Proceedings of the First Nordic Conference on Pattern Languages of Programs

Edited by Pavel Hruby and
Kristian Elof Sørensen

Published by Microsoft Business Solutions, ApS

VikingPLoP 2002, Proceedings of the First Nordic Conference on Pattern Languages of Programs, edited by Pavel Hruby and Kristian Elof Sørensen.

For more information about VikingPLoP please visit www.plop.dk/vikingplop.

Published by Microsoft Business Solutions, ApS.

Printed in Denmark by DataSats-DigiSource A/S.

ISBN 87-7849-769-8

Contents

Introduction

Patterns and pattern languages are ways to describe best practices, good designs, and capture experience in a way that it is possible for others to reuse it.

In August 1993, Kent Beck, Grady Booch, Ward Cunningham, Ralph Johnson, Ken Auer, Hal Hildebrand and Jim Coplien attempted to apply Christopher Alexander's ideas of patterns for urban planning and building architecture, to object-oriented software. They started to study object-oriented patterns and discovered an emerging desire to catalog and communicate these themes and idioms. Now, in 2003, patterns have arguably become part of the standard vocabulary of the software engineering community, and an essential part of any significant software project.

The first conference on pattern languages of programs, PLoP, was held in August, 1994, at the University of Illinois. Since then, an increasing number of pattern conferences, such as EuroPLoP, ChiliPLoP, KoalaPLoP, MensorePLoP, and SugarloafPLoP, have helped improve pattern expertise in the growing patterns community around the world.

In May, 2001, Linda Rising had the idea of holding a pattern conference in Scandinavia every year – each year at a different venue – to enable people in Scandinavia, who might not otherwise attend a PLoP conference, learn about patterns.

The first VikingPLoP was held in Højstrupgård, a small castle north of Copenhagen, Denmark, in September, 2002. The conference was primarily structured around writers' workshops, supplemented by a focus group on Christopher Alexander and two tutorials: one for first-time PLoP participants, and the other for upcoming shepherds of pattern papers. In total, 25 papers were submitted to the conference, of which 19 were accepted for the writers' workshops. The accepted papers covered the areas of the software architecture and business patterns; development processes and methods; and software design and programming.

This conference would not have been possible without the unstinting help and advice of Linda Rising and Neil Harrison during the planning stage, and Daniel May, author of the VikingPLoP logo, VikingPLoP Web master and dedicated shepherd. We would also like to express our deepest appreciation to Gertrud Bjørnvig, Frank Bushmann, Kevlin Henney, Paul Taylor, Povl Kvols Jensen, Heðin Meitil and Jim Coplien, for their advice during the conference organization; Richard P. Gabriel for leading the focus group on Christopher Alexander; Neil Harrison for conducting the tutorials; all program committee members, shepherds, authors of the submitted papers; Cori N. Johansen for graphical design, Roger Holtum for editorial help; Roar Prip and antenna.nl for hosting the website; Laila Nielsen for coordinating the great facilities at Højstrupgård, and George Platts for helping maintain high spirits during the whole conference.

We would also like to particularly thank to our sponsors, Michael Nielsen of Microsoft Business Solutions for providing essential financial sponsorship of the conference, Jens Coldewey, Jutta Eckstein, Andreas Rüping and Frank Buschmann for the Hillside Europe support, and Victoria Smith of Pearson Publishing for providing pattern literature to the conference participants.

March 2003,
Pavel Hruby and Kristian Elof Sørensen

Conference Chairmen

Pavel Hruby and Kristian Elof Sørensen

Program Committee

Gertrud Bjørnvig, Frank Buschmann, Jim Coplien, Neil Harrison, Kevlin Henney, Daniel May, George Platts, Linda Rising and Paul Taylor.

Shepherds

Jason Baragry, Andy Carlson, Pascal Costanza, Serge Demeyer, Arno Haase, Bob Hanmer, Neil Harrison, Kevlin Henney, Doug Lea, Klaus Marquardt, Daniel May, Michael Pont, Dirk Riehle, Linda Rising, Gustavo Rossi, Peter Sommerlad and Kristian Elof Sørensen.

Sponsors

Microsoft Business Solutions, ApS
Hillside Europe
Pearson Publishing

Shepherding Award

Patterns are the essence of the PLoP conferences. The shepherding process improves the quality of patterns papers. Being a shepherd requires a lot of time and effort. While it usually is a rewarding process both for shepherd and author, it can also include challenges and difficulties. To thank the many people who have laboured as shepherds, an award was instituted. The award is called "The Neil Harrison Shepherding Award". Neil Harrison has guided the VikingPLoP shepherding process, and makes sure that this process succeeds at the PLoP conferences around the world.

The trophy has been Kay Bojesen's Ape – an item possessing characteristics of a good shepherd. A passion for quality, not only at the surface but to the core, being in the game for the long run not just to reap a quick return, as well as the ability to live up to all this while having fun and spreading joy and happiness. Kay Bojesen has been one of the pioneers of Danish industrial design, and the Ape has been on the market since 1951. It has spread joy and playful happiness among people of all ages ever since.

At VikingPLoP 2002, the program committee members and the authors of accepted papers had the right to nominate a shepherd. Eight shepherds were nominated, and the award went to the shepherd who received the most nominations.

The Neil Harrison Shepherding Award
VikingPLoP 2002
Awarded to Linda Rising

Software Architecture, Analysis and Design

A Software Metric Pattern Dialect

Martin Auer
Vienna University of Technology
Research Industrial Software Engineering, Institute of Software Technology
http://www.swt.tuwien.ac.at
Favoritenstr. 9-11, A-1040 Vienna
m.auer@swt.tuwien.ac.at

Abstract

Software patterns are gaining acceptance at many different levels of the software process (implementation, design, architecture, as well as at the organizational level). Especially analysis patterns have proved to be very useful in domain understanding and description.

This paper describes how an existing and established analysis pattern language is reused to derive analysis patterns that are valid for the domain of software metrics. Existing patterns are hereby translated from a source domain to a target domain and modified whenever required by the target domain. Furthermore, new analysis patterns specific to the domain of software measurement are given.

The goals are (i) to efficiently assess main aspects of the domain of software measurement by reusing experiences from domains which have similar concepts of measurement or observation, and (ii) to use the resulting pattern language as a starting point for developing tools and data formats which are able to express the concepts described by the analysis patterns--which should possibly "cover" the domain.

1. Introduction

The concept of patterns is becoming ubiquitous in software engineering and development. Originally introduced in [AIS77] in the context of architecture, it gained widespread acceptance with [GHJ95, Amb98, BMR96], which describe design, process and software architecture patterns.

Especially one family of patterns, analysis patterns, have proved to be a very useful--in describing recurring aspects of specific domains and in communicating efficiently about domain properties and requirements [Fow97].

It is the goal of this paper to define a set of analysis patterns--a pattern language--for the domain of software metrics. This language should support and ease domain understanding and provide the ground for the development of generic tools and data formats to handle metric data.

Instead of creating the patterns from scratch, an existing and established pattern language is reused and translated to the domain of software metrics. This is possible, as other domains are dealing with very similar concepts of measurement and observation, especially the health care domain.

In an on-going research effort at the Institute of Software Technology at the Vienna University of Technology, the pattern language is currently being used to derive design requirements for metric tools and data formats [Aue02].

Section 2 gives pointers to some related work. Section 3 gives an overview, of how an existing pattern language was translated to the domain of software metric collection. Sections 4-6 present new software measurement patterns. Section 7 sums up the results and gives an outlook on further research directions.

2. Related Work

[AIS77] defined many patterns in the context of architecture. [GHJ95] used similar templates in describing software design patterns. Other patterns families include analysis patterns [CNM97, Fow97], process patterns [Amb98] and architectural patterns [BMR96, HBH99].

Analysis patterns have been used successfully to devise domain-specific applications and to share domain knowledge by acting as a common vocabulary [Fow97, Fer98]. Other examples are [Kel98], which presents several patterns from the domain of insurance systems, and [WH98], which describe analysis patterns for e-commerce transactions.

For a general introduction to software metrics, please refer to [FP97]. The mathematical foundations of measurement are described in [KLST71]. Tools for software measurement and measurement automation are described in [KPR01, DFKW96]. The first draft of a metric data exchange format is described in [Aue02]. The general process of translating patterns between similar domains and subsequently relating them to patterns of different families is described in [Aue02c].

3. Create an Initial Pattern Language

3.1 Pattern Reuse

[Fow97] presents several analysis pattern languages from the domains of health care and corporate finance in order to describe recurring analysis models of these domains. One of the presented languages deals with measurement and observation.

This language contains, for example, patterns that deal with compound measurement units like LOC/hour (Compound Unit), with conversions between different measurement units (Conversion Ratio), or with storing measurement meta information like accuracy (Protocol). Other, more complex patterns, describe indirect observations that can be derived from direct ones (Associated Observation), or how rejected observations can be connected to the rejecting observations (Rejected Observation).

The following list should give an overview of the measurement analysis patterns described in [Fow97]:

Pattern, Page	*Intent*
Quantity, 36	To handle measurement units and allowed operations on data types, define an abstraction level above basic numerical data types containing both measurement value and unit.
Conversion Ratio, 38	To handle non-homogeneous measurement units (in use for historical reasons or for the convenience of handling systems with unit magnitudes appropriate to their scale), provide means of converting them by using conversion ratios between units.
Compound Units, 39	To guarantee consistent mathematical data handling, explicitly distinguish between units (s, m…) and compound units (m/s, m^2…) by combining units into compound units.
Measurement, 41	To handle the large number of possible kinds of measurement, separate the measurements from the measured entities and avoiding treating measurements as mere entity attributes.
Observation, Category Observation With Phenomenon, 42, 45	To handle quantitative and qualitative observations, distinguish between numerical measurements and category observations and describe, how they relate to the types of phenomena they are measuring.
Subtyping Observation, 46	To reflect hierarchically structured observations (for example, a general observation "diabetes" along with the more special "diabetes type I" and "diabetes type II" observations), map the potentially deep hierarchies in observation procedures to a corresponding hierarchical structure and allow to propagate negative observations downwards ("no diabetes" implies "no diabetes type I") and positive observations upwards in the hierarchy.
Protocol, 46	To make available the type and accuracy of a measure-

	ment for later evaluation, relate meta information about a measurement to it.
Dual Time Record, 47	To handle changing measurement values over time and to log a measurement history (legally required, for example, in health care), let the model express the time period, where some measurement data is considered valid, in addition to the time point when the measurement took place.
Rejected Observation, 48	To trace back observation changes, connect an observation that rejects a previous one to the rejected observation.
Active Observation, Hypothesis, Projection, 49	To reflect distinct measurement probabilities and the difference between projections and actual observations, distinguish the main measurement and projection types instead of using mere probabilities.
Associated Observation, 50	To express causal relations between observations, derive observations from evidence by mapping initial observations to derived ones using an associative function.
Process of Observation, 51	In order to express temporal and causal dependencies between observations (like observations leading to further ones, or the handling of contradictory observations), model the dependencies in the measurement process.
Enterprise Segment, 59	To relate measurements to entities of an organization, model the static organizational aspects of the environment where the measurement takes place using multiple hierarchies.
Measurement Protocol, 66	To record measurement origins and calculation steps, provide dedicated data structures to express the sources of measurement.
Range, 76	To express common operations on measurement ranges (intersection, isElementInRange…) and to support range types like open or unbound ranges, explicitly treat ranges

	as types of their own.
Phenomenon With Range, 77	To categorize classes of possible measurement values, define non-intersecting ranges of values whose union covers all possible measurement values.

[Fow97] develops his languages by translating patterns between the domains, for example, by applying patterns from the health care domain to the corporate finance one and modifying them according to the specific needs, and states: "By allowing patterns to migrate like this, I hope that more and more useful patterns will emerge, […]".

Indeed, many of the proposed patterns can be translated with little or no change to other domains as well, specifically to the domain of software metric collection. This is much more efficient than devising a pattern language from scratch, as working, existing models can be reused. Another advantage of this reuse or translation becomes visible, when it comes to building *stable* software systems, databases or protocols: In such cases the analysis is required to be as extensive as possible, i.e., to encompass most analysis patterns that might occur in a specific context. By translating an existing set of patterns it becomes less likely to forget or underestimate important analysis issues, the patterns language will instead be more complete or "domain-covering" (in fact, one reason for building this software metrics pattern language is to derive a metric data exchange format and protocol from it).

So in order to devise an initial set of analysis patterns for the domain of software metric collection, some patterns occurring in Fowler's measurement and observation domain are translated to it and, if necessary, changed according to specific domain requirements.

3.2 Pattern Translation

This section gives a very brief overview on the translation of patterns from the domain of measurement and observation given in [Fow97] to the software metrics domain, yielding an initial set of analysis pattern. Please refer to [Aue02d] for a detailed description of the translated patterns' modifications.

The following list describes which pattern were translated, modified and translated, or omitted:

Pattern	*Translation Action*
Quantity	translated
Conversion Ratio	translated with modifications
Compound Units	translated with modifications
Measurement	translated
Observation, Category Observation With Phenomenon	translated with modifications
Subtyping Observation	not translated
Protocol	translated
Dual Time Record	translated with modifications
Rejected Observation	not translated
Active Observation, Hypothesis, Projection	translated with modifications
Associated Observation	translated
Process of Observation	not translated
Enterprise Segment	translated with modifications
Measurement Protocol	translated with modifications
Range	translated
Phenomenon With Range	translated

Some of the analysis patterns proposed in [Fow97] (Quantity, Measurement, Protocol, Range, Phenomenon With Range, Associated Observation) can be translated without change to the domain of software metrics. In fact, these models can be applied to most environments where measuring takes place (experimental environments, polls…). Some of the model's classes might have to be renamed, though--as the models are rooted in health care, [Fow97] often uses the term "Person" instead of, for example, "Entity".

Many analysis patterns can be translated after some slight modifications to them. These changes take into account the specific requirements of the software metrics domain. For

example, metric data is often to be analyzed with sophisticated analysis tools like online analytical processing (OLAP) tools, which can provide interactive data analysis through drill-down and roll-up operations. Such tools' operation heavily depends on structural aspects of the measurement data--so if the tools should be exploited, this must be taken into account at the early analysis stage.

> **Example: Translating the Conversion Ratio Pattern**
> The Conversion Ratio pattern describes a simple model for the conversion of different measurement units. Basically, for a pair of units a conversion factor is stored. Examples for a conversion ratio between software metrics are industry-average LOC per function point values.
> While the proposed pattern is sufficient to express simple "scaling" conversions, it can't express more complex conversions, for example, on interval scale type measurement values like dates (for an introduction to measurement scale types please refer to [FP97]).
> By adding just one additional class Offset to the pattern, instead of being limited to ratio scale type conversions, interval scale type conversions can be expressed as well.

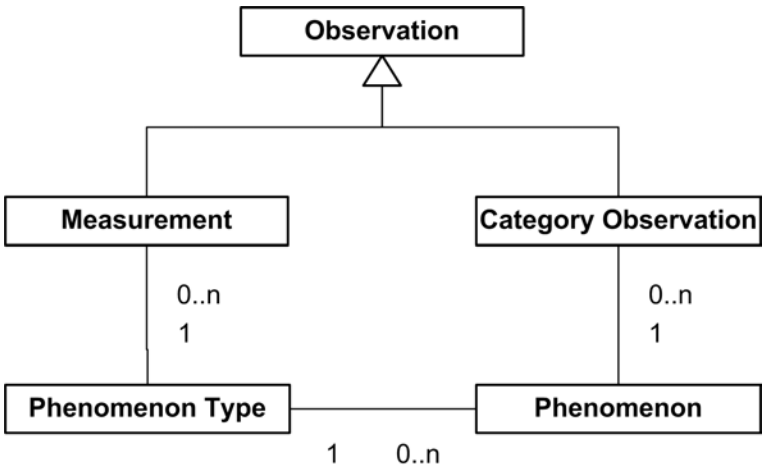
Finally, while many patterns could be translated with no or little modifications, some patterns proposed in the context of health care or corporate finance may be omitted in the area of software metrics collection (Subtyping Observation, Rejected Observation, Process of Observation), largely because of different legal requirements, or because of much more elaborated chains of hypotheses and measurements in health care which need not be modelled in software metrics.

To sum up, reusing an existing pattern language from a similar domain comprises finding out, which patterns can be translated, which can be translated with some modifications and which can't be translated and should rather be omitted. One metaphor of such a translation or reuse of an existing patterns language might be to describe the resulting language as being a "dialect" of the original one.

3.3 Adding New Patterns

After defining an initial set of analysis patterns, new, domain-specific patterns should be added to the derived pattern dialect in order to increase its expressiveness. However, only such patterns must be added, that cannot be expressed elegantly using the translated and potentially modified ones.

The following sections present the new patterns Multiple Choice, Distance and Scale. Some of them closely relate to or extend one of the central patterns proposed in [Fow97]: Category Observation With Phenomenon. This pattern describes the main measurement types (numerical measurement and category observation) as well as their possible relations. In order to better understand the new patterns, it might be helpful to take a brief look at this pattern.



The Category Observation With Phenomenon pattern

A concrete occurrence of this pattern in the domain of software metrics would be the measurement of complexity. Complexity can be measured using McCabe's metrics (which would be an object of the class Measurement in the diagram below); alternatively it can be assessed "manually" by visually interpreting the graphical complexity of a program's flow

graph and mapping this impression on an ordinal scale, i.e., to different perceived complexity levels like “simple”, “complex”, “very complex”. The latter mapping can be expressed with objects of the class Phenomenon.

Now, this pattern obviously can model some aspects of the software metrics domain. Nevertheless, modifications and extensions to it yield new patterns, which can make the pattern language even more expressive. Furthermore, some entirely new patterns are necessary to let the language cover other, highly domain-specific issues. The following sections describe these new patterns.

4. New Analysis Pattern: Multiple Choice

Many times measurements are made using questionnaires, which can contain multiple-choice questions, for example “What operating systems are you using?” Instead of asking a single question for each operating system, the questions are grouped to ease access and understandability, by structuring the questionnaire’s information.

Another issue is, that it is usually desirable to create the data collection forms automatically, for example using the XSL language to create HTML forms out of the questions’ data structure. The layout and the creation process should support multiple-choice questions.

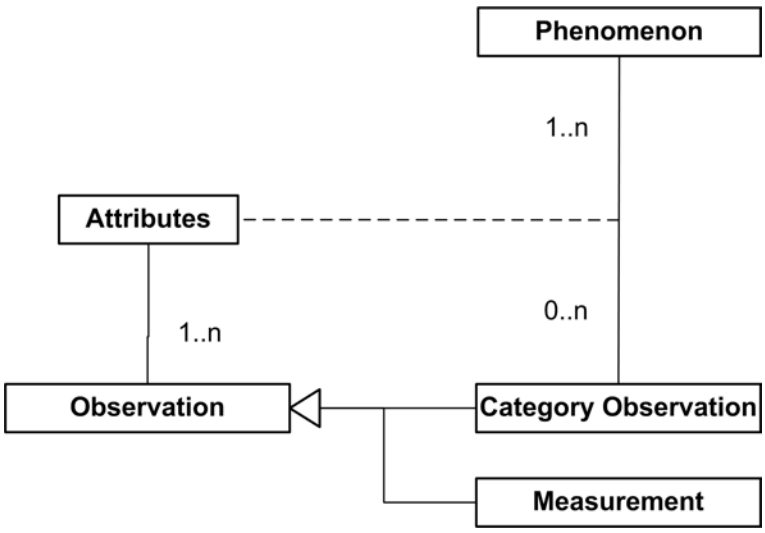
Along with data collection, other issues like storage, verification and analysis of the data should be able to handle multiple-choice questions’ properties.

Yet these tasks quickly become awkward, if unsuitable models and data structures are used to express this form of data. For example, if a single string is used to store a comma-separated list of identifiers (the “answers”) in one database field, it is difficult to analyze the data using standard SQL statements.

There is a second problem: Some multiple-choice questions might even need an additional level of detail. Consider the questions "What operating systems are you using? And for how many years?" Again, the additional question's data handling becomes difficult if the analysis models don't express such questions natively - inelegant workarounds are the inevitable consequence.

One suboptimal solution is to use the translated pattern Category Observation With Phenomenon (see section 3). It supports single choice questions, with an object of class Category Observation (for example, an observation of Carla's blood group) referring to one object of the class Phenomenon (for example, "blood group a"). The concept of multiple-choice questions or multiple measurements could then be expressed by several single observations, i.e., the tuple ("A uses OS 1", "A uses OS 3") could be expressed using the two single observations ("A uses OS 1") and ("A uses OS 3"). However, this structure leads to very inefficient implementation and makes any data verification and handling unnecessarily complicated. Furthermore, additional levels of detail are not part of this model, either.

Therefore, a considerable enhancement, a new pattern, is needed. It natively supports multiple-choice data with a dedicated n-to-n-relation to the set of possible choices, as well as additional information or levels of detail with appropriate data structures.



First, the multiplicity of the relation between observation and category is changed, thus natively allowing for multiple phenomena per observation. Second, an associated class is used to express the potential additional level of detail. Several additional details can be expressed by referencing multiple Observation objects.

Note that attributes consist of several observations, which typically are of type Measurement, but could be of type Category Observation as well. Example: The question "What operating systems are you using? How many months? How satisfied are you with them?" has a Measurement object (number of months) and a Category Observation object ("satisfied", "thrilled"…) associated with each reference to a Phenomenon object ("Windows", "Linux"…) in the Category Observation object (the question's answer).

Some consequences of explicitly modelling multiple-choice measurements are:

- The automatic creation of online questionnaires including multiple choice questions and lists of possible values is facilitated as this type of data is considered right from the start (for example, when designing XSL programs, which are typically used for such tasks).
- Databases storing such data are designed with this measurement data type in mind and are therefore likely to be more stable.
- Later analysis steps to be performed on the data, for example using OLAP tools, are considerably simplified.

Several questionnaire design tools available support multiple choice type questions, like Raosoft's EZSurvey (www.raosoft.com) or SyncForce's SurveyWorld.NET (www.surveyworld.net). Few tools however support multiple-choice questions with additional attributes; one of them is Infopoll Inc.'s Infopoll Designer (www.infopoll.com), whose so-called *matrix type questions* can express them.
Some tools provide additional features like the automatic creation of a database to store the collected data. Perseus Development's SurveySolution XP (www.perseus.com), for example, maps multiple-choice questions to a single table with multiple columns.

5. New Analysis Pattern: Distance

There have been many discussions on the appropriateness of the term „software metrics“, as metrics are an already well-defined mathematical concept of distance. Some argued that instead of “metrics” the term “software measure” should be used to be consistent with existing mathematical terms (with little success).

From a measurement point of view, a metric or distance function is a mapping between a pair of entities to one measurement value, preserving some relations (non-negativity/equivalence, symmetry, and the triangle inequality).

Most real-world distance metrics are simply differences between one-dimensional values or geometrical distances, which usually do not deserve special attention as they can be calculated directly from the original measurement values.

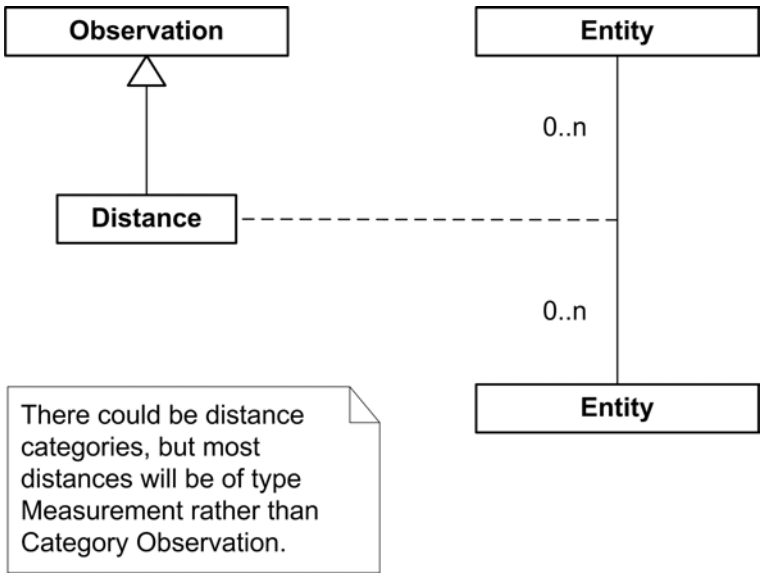
Yet there are some non-trivial distances in software that cannot be calculated from measurement values of isolated entities, but must be derived by measuring attributes of entity *pairs*. Prominent examples are coupling metrics between classes. Note that most coupling metrics or, more specifically, some derived metrics expressing distance, are not metrics by the strict mathematical definition as the mapping usually violates some of the required relations (usually the triangle inequality). But still, the important distance function concept of measuring pair attributes is preserved.

Trying to express such data with existing analysis patterns leads to inelegant and difficult-to-understand structures. For example, by applying the Multiple Choice with Attributes pattern, one could relate entities with coupling values larger than zero (by using entities as objects of type Phenomenon on the to-n-side of the relation), with one additional attribute containing the actual coupling value. Obviously overstretching the original pattern’s expressive capabilities, especially one important attribute of distances, namely symmetry, is not expressed adequately:

- If the distance between entities A and B is stored by making B a multiple choice entity of A, but A is never stored as multiple choice entity of B, than the structure does not reflect the symmetry properties of the distance function. Data analysis hence becomes tedious.
- If both the distances d(A,B) and d(B,A) are expressed using the Multiple Choice pattern, symmetry is preserved, but at the cost of redundant data.

Another problem that occurs with such metric data is that working with large-scale systems and distance metrics on its entities may result in a very large data volume. In order to efficiently handle this kind of information (for example, for interactive visualization using multidimensional scaling methods), special attention has to be given to the data--for example, by using special intermediate data structures for high-performance data handling which can't be achieved using standard relational database structures.

Special attention should therefore be given to such distance metrics. One way of expressing this is to relate an observation with a pair of entities using an associated class:



The object of measurement was renamed Entity, thus referring to all possible measurement entities like artefacts, processes, people etc.

Another consideration can be made on the type of the Distance object. It will typically be of type Measurement, but as type Category Observation could be used as well, the more general Observation class is used in this pattern.

Some consequences of separating the binary distance from simple unary metrics are:

- The distance-intrinsic property of symmetry is expressed in this pattern, thus providing an understandable model. If the implementation of this model preserves this property, data analysis is facilitated considerably.
- There is no redundant data in this model, and there is support for spare data structures (by allowing *zero*-to-n relations to other entities). This expresses two basic requirements for high-performance data handling, and should point a way to efficient implementation.

Several publications propose coupling metrics, the software measurement concept of distance, as it is regarded to have crucial impact on software quality [AK99, HCN98].
Most general statistic tools, for example, the SPSS suite of statistical applications (www.spss.com), are able to handle distance data structures occurring in various field like the social sciences, especially for entity classification and cluster visualization using multi-dimensional scaling methods.
In the domain of software measurement, distance metrics were used to semi-automatically classify modules into subsystems [SL00].

6. New Analysis Pattern: Scale

Scale types are a mathematical concept of classifying the expressiveness of measurements. Formally defined by using so-called *measurement scales* (relation-preserving homomorphisms between sets of real-world entities and numerical symbols) and *admissible transformations* (functions which preserve the homomorphism property of functions when concatenated to

them), scale types describe, how strong the relations are which are preserved by the measurement function.

The following are the most common scale types along with some examples:

Scale Type	*Average*
Nominal scale	Naming, numbering of alternatives, mapping to male/female
Ordinal scale	Wind force, marks, Moh's hardness scale, Richter scale, preference
Interval scale	Temperature in Celsius, time in calendar
Ratio scale	Mass, Temperature in Kelvin, loudness
Absolute scale	Counting, probabilities

Nominal scale type measurements preserve simple unary relations like isMale(..) or a naming of entities; ordinal scale type measurements preserve the binary "greater-than" relation $(..,..)_<$; interval scale type measurements preserve the meaning of differences between measurement values (like, for example, in date information). For an introduction to measurement theory or its application to software metrics, please refer to [KLST71, FP97].

The scale type of a particular measurement affects the kind of statistical methods that can be applied to the data. This might not be useful in domains with low requirements on statistical analysis; however, in the domain of software metrics powerful statistical methods are often applied to the data. If the measurement's scale type is not considered this may lead to the (wrong) application of analysis methods making too strict assumptions on the underlying data's structure.

A few examples of allowed statistical methods are given in the following table:

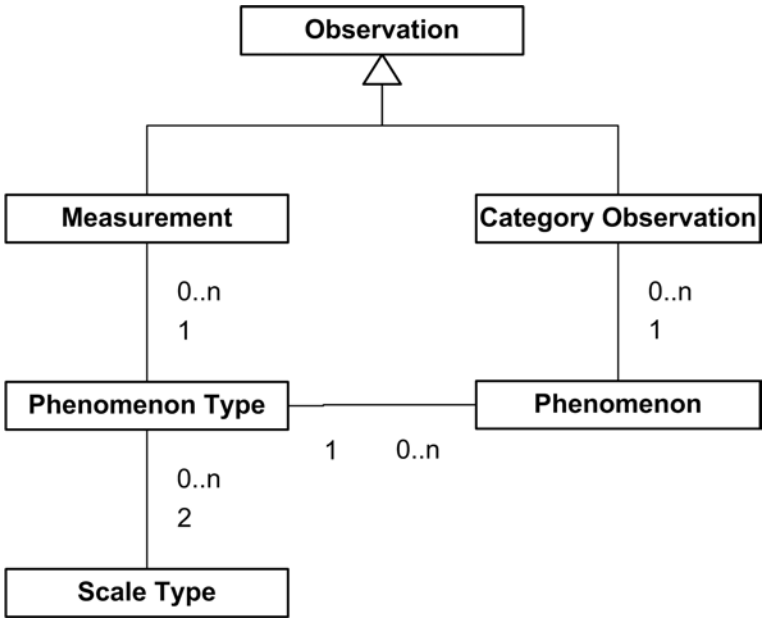
Scale Type	*Average*	*Variability*	*Correlation*
Nominal scale	Mode	Information content	Coefficient of contingency, phi-coefficient
Ordinal scale	Median	Percentiles	Spearman's rho, Kendall's tau

Interval scale	Arithmetic mean	Standard deviation, variance	Product moment correlation, correlation ratio
Ratio scale	Geometric mean, harmonic mean	Variation coefficient	-

In this table, a scale type indicates the weakest scale where a specific method is applicable. More statistical methods, for example, significance tests, are classified in [Ort74].

The main problem that can arise in this context is to use too strong a statistical method for a data set, i.e., to make wrong assumptions about the data's properties. A typical example is to use mean and variance methods on data that is on an ordinal scale, for example grades.

To avoid that, the scale type of measurements and observations should be a part of the model describing the relations between observations and the respective phenomenon type being observed. The concept of scale types can be expressed by introducing a dedicated Scale Type class related to the Phenomenon Type class.



There should be two distinct scale types for Measurement and Category Observation entities. As an example, Phenomenon Type "length" is measured on a ratio scale (in meters), while the corresponding Phenomenon objects "large", "medium" etc. are values on an ordinal scale.

Typical scale types for Measurement objects are ratio or interval, while Category Observation objects are usually on the nominal or ordinal scale. If category observations are made on an ordinal scale, the Phenomenon class should provide means to compare instances of each other in order to express the order relation.

Some consequences are:

- The potential expressiveness of the collected data is considered before the actual data collection takes place.
- To enforce scale type declaration makes the use of wrong statistical methods unlikely. Analysis tools, for example, could provide only allowed statistical methods.

The concept of measurement scales, first presented in [Ste46], was applied to empirical sciences, for example, in classifying behavioural measures in psychology, which often are of interval instead of ratio scale type [Rea92].
Some statistic tools support the measurement scale concept to some extent. The tool ViSta (forrest.psych.unc.edu/research), for example, distinguishes between nominal, ordinal and interval/ratio data types, referring to them as category, ordinal and numeric variables, respectively.

7. Conclusion and Context

Many software design problems like the design of hopefully stable database structures or the definition of data formats and protocols to be used in a variety of environments, share

one common problem: the need to extensively assess present and possibly even future domain properties and structures, as well as entities and their relations.

Elegant designs are able to express a concept in a simple and consistent way (as, for example, the general-purpose data format XML expresses tree structures), while inferior designs quickly have to turn to workarounds (take for example some custom-made and error-prone CSV expression of tree structures with arbitrary delimiter…).

Current research activities at the Institute of Software Technology (Vienna University of Technology) aim to design a data format to exchange software metric data seamlessly between tools and repositories.

In order to maximize the format's expressiveness, first the analysis pattern language presented in this paper was defined. This patterns language should reflect the structure of the software measurement domain, and help to understand and communicate related concepts. The data protocol should be able to cover and express all previously identified analysis patterns consistently, i.e., the data structures should be designed to encode the information types natively, elegantly and without workarounds.

Instead of designing this pattern language from scratch, most patterns were translated and modified from a similar domain. This helped to define a starting set of patterns and made it more likely to achieve an extensive set of patterns. Then, new patterns were added to express domain-specific properties.

Further work will refine this pattern language and relate the analysis patterns to design and implementation patterns/idioms. Another issue will be to define a metric data exchange format and to implement an infrastructure to support data exchange between common development tools.

8. References

[AIS77] C. Alexander, S. Ishikawa, M. Silverstein, M. Jacobson, I. Fiksdahl-King, S. Angel, A Pattern Language, Oxford University Press, 1977

[AK99] E. B.Alan, T. M. Khoshgoftaar, Measuring Coupling and Cohesion: An Information-Theory Approach, Proc. of METRICS'99, Boca Raton, Florida, November 1999

[Amb98] Scott W. Ambler, Process Patterns: Building Large-Scale Systems Using Object Technology, Cambridge University Press, 1998

[Aue02] M. Auer, Measuring the Whole Software Process: A Simple Metric Data Exchange Format and Protocol, Proc. of 6th ECOOP Workshop on Quantitative Approaches in Object-Oriented Software Engineering (QAOOSE 2002), Málaga, June 2002

[Aue02b] M. Auer, XML-Based Metric Data Handling, Tech. Report 02-04, Institute of Software Technology, Technical University of Vienna, March 2002

[Aue02c] M. Auer, Software Decisions with Pattern Relations, accepted for publication in Proc. of Sugarloaf PloP 2002, Pousada Capim Limão, August 2002

[Aue02d] M. Auer, Translating Measurement Patterns to Software Metrics, Tech. Report 02-08, Institute of Software Technology, Technical University of Vienna, September 2002

[BMR96] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, Michael Stal, Pattern Oriented Software Architecture - A System of Patterns, Wiley, 1996

[CNM97] Peter Coad, David North, Mark Maryfield, Object Models: Strategies, Patterns, and Applications, 2nd ed., Yourdon Press, New Jersey, 1997

[DFKW96] R. Dumke, E. Foltin, R. Koeppe, A. Winkler, Softwarequalität durch Messtools, Vieweg Professional Computing, Wiesbaden, Germany, 1996

[GHJ95] E. Gamma, R. Helm, R. Johnson, J. Vlissides, Design Patterns: Elements of Reusable Object-Oriented Software, Addison-Wesley, 1995

[HBH99] J. Hall, L. Barroca, P. Hall, editors, Software Architectures - Advances and Applications, Springer-Verlag, 1999

[Fer98] Eduardo B. Fernandez, Building Systems Using Analysis Patterns, Proc. of the Third International Workshop on Software Architecture, ACM, Orlando, Florida, 1998

[Fow97] Martin Fowler, Analysis Patterns: Reusable Object Models, Addison-Wesley, 1997

[FP97], Norman E. Fenton, S. Pfleeger, Software Metrics: A Rigorous & Practical Approach, PWS Publishing Company, 1997

[HCN98] R. Harrison, S. Counsell, R. Nithi, Coupling Metrics for Object-Oriented Design, Proc. of the Fifth International Software Metrics Symposium, Bethesda, Maryland, November 1998

[Kel98] Wolfgang Keller, Some Patterns for Insurance Systems, PLoP'98, *http://www.objectarchitects.de/ObjectArchitects/papers/index.htm*

[Ker95] Normal L. Kerth, Caterpillar's Fate: A Patterns Language for the Transformation from Analysis to Design, in James O. Coplien, Douglas C. Schmidt, ed., Patterns Languages of Program Design, Addison-Wesley, 1995

[KLST71] D. H. Krantz, R. D. Luce, P. Suppes, A. Tversky, Foundations of Measurement. Vol. I. Additive and Polynomial Representations, Academic Press, New York, 1971

[KPR01] Seija Komi-Sirviö, Päivi Parviainen, Jussi Ronkainen, Measurement Automation: Methodological Background and Practical Solutions – A Multiple Case Study, Proc. of the 7th Int. Software Metrics Symposium, London, 2001

[Ort74] Bernhard Orth, Einführung in die Theorie des Messens, Verlag W. Kohlhammer GmbH, Stuttgart, 1974

[SL00] Frank Simon, Silvio Löffler, Semiautomatische, kohäsionsbasierte Subsystembildung, in Reiner Dumke, Franz Lehner, ed., Software-Metriken: Entwicklungen, Werkzeuge und Anwendungsverfahren, Gabler Verlag, Wiesbaden, 2000

[Rea92] C. C. Reaves, The Theory of Measurement, in Quantitative Research for the Behavioural Sciences, John Wiley & Sons, New York, 1992

[Ste46] S. S. Stevens, On the Theory of Scales of Measurement, Science 161, 1946

[WH98] Hans Weigand, Willem-Jan van den Heuvel, Meta-Patterns for Electronic Commerce Transactions based on FLBC, Proc. of Hawaii Int. Conf. on System Sciences (HICSS'98), IEEE Press, 1998

[Zim95] Walter Zimmer, Relationships Between Design Patterns, in James O. Coplien, Douglas C. Schmidt, ed., Patterns Languages of Program Design, Addison-Wesley, 1995

Framework Patterns for the Evolution of Nonstoppable Software Systems

Walter Cazzola[1], James O. Coplien[2], Ahmed Ghoneim[3], and Gunter Saake[3]

[1] Department of Informatics and Computer Science,
Università degli Studi di Genova
Via Dodecaneso 35, 16146, Genova, Italy
cazzola@disi.unige.it
[2] University of Manchester Institute of Science and Technology, United Kingdom, and
Computer Science, North Central College, Naperville, Illinois
jocoplien@cs.com
[3] Institute für Technische und Betriebliche Informationssysteme,
Otto-von-Guericke-Universität Magdeburg
Postfach 4120, D-39016 Magdeburg, Germany
{ghoneim|saake}@iti.cs.uni-magdeburg.de

Patlet. *The fragment of pattern language proposed in this paper, shows how to adapt a nonstoppable software system to reflect changes in its running environment. These framework patterns depend on well-known techniques for programs to dynamically analyze and modify their own structure, commonly called computational reflection. Our patterns go together with common reflective software architectures.*

Keywords: Pattern, Framework Pattern, Pattern Language, Reflection, Software Evolution, Dynamic Reconfiguration, Reconfiguration of Nonstoppable System.

1 Context Overview and Case Study

A nonstoppable software system with long life span — that is, a software system whose execution can not be halted for allowing system reconfiguration —, has to be able to dynamically adapt itself to changes to its environment, i.e., to evolve itself. To render a nonstoppable software system *self-adapting* to changes to its environment is a topical issue in the software engineering research area. *Computational reflection* [15, 4] is one of the most used mechanisms for getting software adaptability [8, 19, 18]. Two aspects control the evolution of such kind of systems: *behavior*, and *dependencies*. Both of them can be involved in system evolution to comply with changes to system requirements.

When designing *urban traffic control systems* (UTCS), the software engineers must face many issues such as distribution, complexity of configuration, and reactivity to the environment evolution. Moreover, modern urban agglomerates provide a lot of unexpected hard to plan problems such as traffic lights disruptions, car crashes, traffic jam and so on. In [17] all these issues and many others are illustrated.

The evolution of complex modern cities has posed significant challenges to city planners in terms of optimizing traffic flows in a normally congested traffic network. Simulation and analysis of such systems require modeling the behavioral, structural and physical characteristics of the road system. This model includes mobile entities (e.g., cars, pedestrians, vehicular flow, and so on) and fixed entities (e.g., roads, railways, level crossing, traffic lights and so on).

Figure 1, shows a possible object model for the UTCS described by using the UML formalism [2]. Moreover, Fig. 1 shows how the UTCS can be integrated with our reflective approach to evolution [5], i.e., the approach we are expressing as pattern language. This model includes two types of objects:

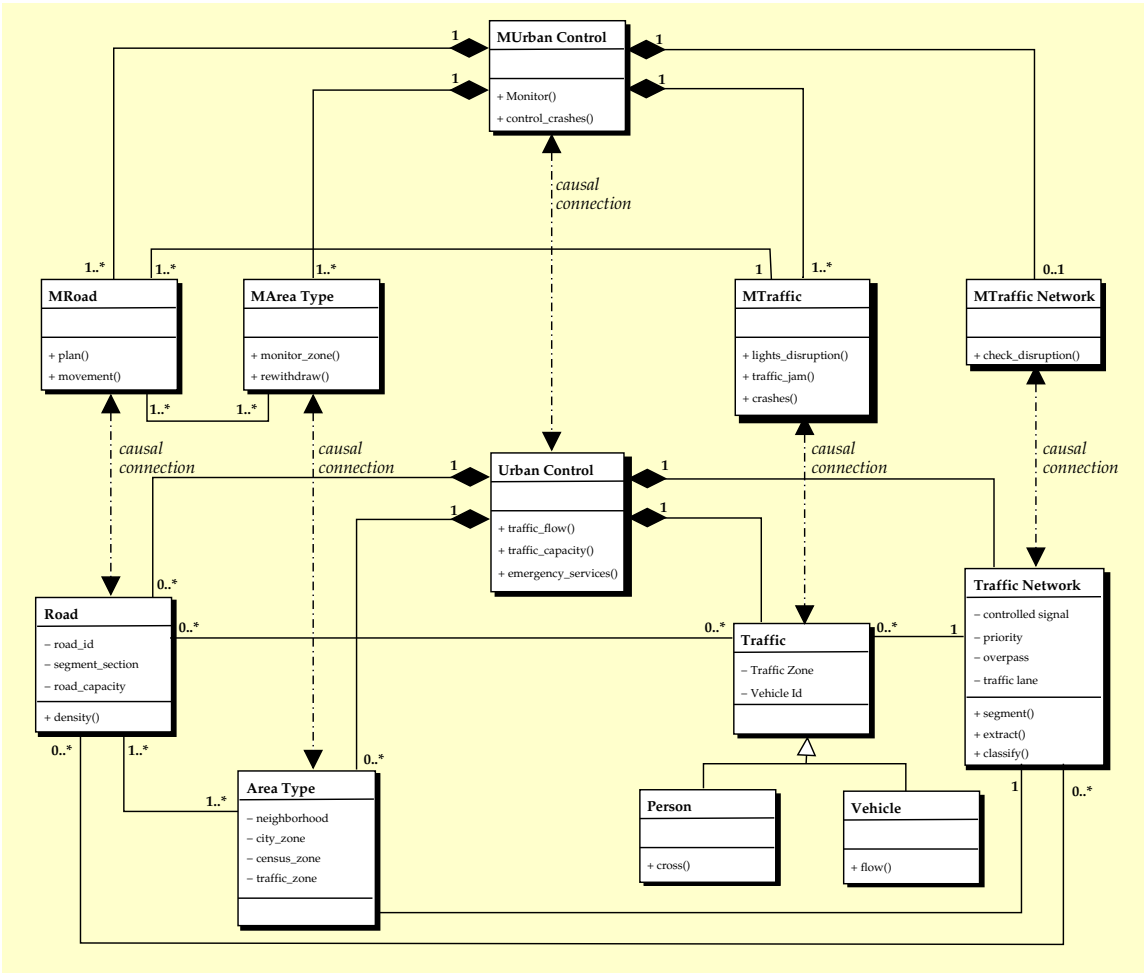


Fig. 1. Urban reflective object model.

– base-objects, e.g., urban control, roads, and so on, whose interactions describe UTCS' structure and its behavior in expected situations;
– meta-objects, associate to the base-objects, whose work consists of evolving the base-objects to deal with unexpected situations.

As mined from the ESCORT project [17], UTCS to deal with all the unexpected problems has to monitor vehicular flowing density[4], and the status of every involved entity (both fixed and mobile). The meta-level detects every anomaly of the UTCS and adapts the current traffic schedule (i.e., the UTCS behavior) to face the problem.

It is fairly evident that modeling and developing a urban traffic control system is an hard job for software engineers. The most important issues they have to deal with are: slowly evolving road situation, that the model must reflect accurately at all times, changes to the road situation that happens with no warning (accidents, broken traffic lights etc.) and that the system must take into account immediately, and of course the ever changing flow of people and vehicles and the dire consequences of restarting the system during rush hour or at all.

[4] UTCS is supported by CCD-Cameras and movement sensors installed in every important nexus [17]. CCD-cameras take a photo every second and by comparing these photos, we can estimate the traffic flowing density. Sensors will notify anomaly events that cannot be detected by CCD-cameras like traffic light disruptions or damages to the road structure.

There are many other nonstoppable systems that have problems similar to the urban traffic control system. Air traffic control, assembly line and nuclear station power are some examples of this kind of systems. Their problems are related to the fact they are nonstoppable and need a higher reactivity to sudden environmental changes. We do not further face these systems but the pattern language we propose can also be used for modeling them as well as the urban traffic control system.

2 Reflection, Reflective Architectures and Evolution of Nonstoppable Systems

Reflection is the ability of a system to watch its own computation and possibly change the way it is performed. Observation and modification imply an "underlay" that will be observed and modified. Since the system reasons about itself, the "underlay" is itself, i.e. the system has a *self-representation* [15].

A *reflective architecture* logically models a system in two layers, called *base-level* and *meta-level*. In the sequel, for simplicity, we refer to the "part of the system working in the base-level or in the meta-level" respectively as base-level and meta-level. The base-level realizes the *functional* aspect of the system, whereas the meta-level realizes the *nonfunctional* aspect of the system. Functional and nonfunctional aspects discriminate among features, respectively, *essential or not* for committing with the given system requirements. Security, fault tolerance, and evolution are examples of nonfunctional requirements[5]. The meta-level is *causally connected* to the base-level, i.e., the meta-level has some data structures, generally called *reification*, representing every characteristic (structure, behavior, interaction, and so on) of the base-level. The base-level is continuously kept consistent with its reification, i.e., each action performed in the base-level is *reified* by the reification and vice versa each change performed by the meta-level on the base-level reification is *reflected* on the base-level. More about the reflective terminology can be learned from [15, 4].

A reflective architecture represents the perfect structure that allows a nonstoppable system to easily evolve. In [5] we have described a reflective architecture for the evolution of nonstoppable systems. In this framework the system running in the base-level is the nonstoppable system prone to be adapted, whereas the nonfunctional feature realized by the meta-level is the system evolution. Evolution takes place exploiting design information concerning the nonstoppable system.

To correctly adapt the base-level system, the meta-level has to face many problems. The most important are: (1) to keep consistent the base-level with its representative in the meta-level, when the base-level evolves, (2) to adapt the reification of the base-level to sudden changes through the design information of the base-level, (3) to verify whether the proposed adaptation would leave the base-level coherent and then to schedule its realization, finally (4) to reflect the modified reification on the base-level. Both reflection and adaptation involve several aspects of a system, the most important are: structure (objects, methods and so on), behavior (state and semantics of the objects) and collaboration (roles, exchanged messages, interfaces and so on).

In this work we describe a fragment of the framework pattern language for dynamically evolving a software system. We talk about framework pattern language because our patterns describe a framework and each of them entrusts part of its execution to other patterns in the

[5] The borderline between what is a functional feature and what is a nonfunctional feature is quite confused because it is tightly coupled to the problem requirements. For example, in a traffic control system the security aspect can be considered nonfunctional whereas security is a functional aspect of an auditing system.

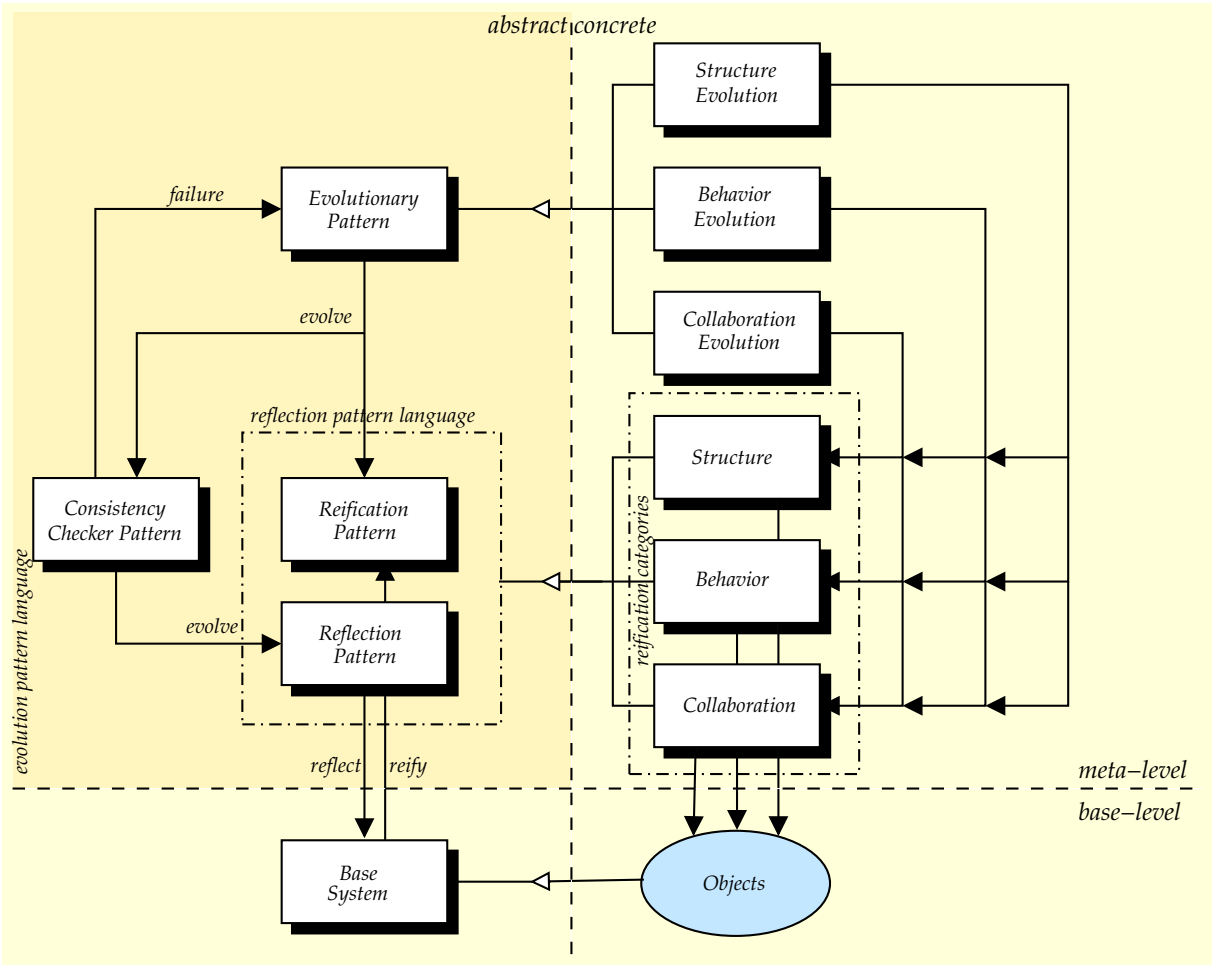


Fig. 2. Framework patterns for the evolution of nonstoppable software systems.

language, i.e., these patterns are tightly coupled. Points (1) and (4) in the above list are entrusted to the pattern language for computational reflection composed of *Reification* and *Reflection Patterns*, the evolution (point (2) in the list above) is entrusted to the *Evolutionary Pattern*, while the *Consistency Checker Pattern* takes care of checking the feasibility of the adaptation and of scheduling its realization (point (3)). Figure 2 sketches our framework, only patterns in the darker box on the left will be explored in this work.

The pattern language we are going to introduce can be used to adapt a nonstoppable system to sudden changes to its requirements or to its environment. This approach to evolution must be integrated with both the adaptation scheme and the consistency constraints used by the *Evolutionary* and *Consistency Checker* patterns; an possible adaptation scheme based on design information is described in [18]. Note that, reflection patterns does not force to choose a programming language with reflective features but its implementation can take advantage from it.

3 Pattern Language for System Evolution

In this section, we define the fragment of pattern language for the evolution of nonstoppable software systems. We have four main patterns: *Evolutionary*, *Reification*, *Reflection* and *Con-*

sistency Checker Pattern. Evolutionary patterns are responsible of adapting the system to incoming requirements. They work on the system reification supplied them by the reification patterns. The consistency checker pattern verifies the feasibility of the adaptation provided by the evolutionary patterns. Finally, the reification and reflection patterns, these patterns are part of the pattern language for computational reflection, are responsible for reifying the base-level system and, for reflecting the adaptation on the base-level, when the consistency checker pattern approves such an adaptation. Both evolutionary and reflection patterns describe a general behavior that can be applied to many domains, as shown in Fig. 2, whereas the consistency checker pattern collaborates with every application of the reflection pattern.

EVOLUTIONARY PATTERN

Intent. *To enable a nonstoppable software systems to adapt itself to dynamic changes to its requirements.*

Problem

Several times a (nonstoppable) software system must evolve to adapt itself to the evolution of the environment it is modeling. For example, in a software system as the UTCS (Sect. 1), this involves changes in the overall structure and behavior of the system, i.e., new components to interact with and a reorganization of the components interactions. If changes are planned a lot of time in advance, it is not a problem to take advantage of a moment when the traffic is low, for stopping the UTCS for a while, just the time for the reconfiguration. This is not a feasible solution when the change suddenly happens such as in case of a road interruption due to a road accident or something similar. In a similar case, we cannot stop the normal execution of the UTCS, creating many problems in other parts of the system, to face the unexpected situation.

Forces

Several forces are involved in the dynamic evolution of a nonstoppable system, obviously the most important is represented by the fact that:

– keeping still for a while (during the reconfiguration) a nonstoppable system could have dire consequences up to and including death.

Other not so important forces are:

– the system has to change whenever the environment it is modeling changes;
– changes to the environment can happen at all times, they are outside the control of the system and can not be foresee during system designing;
– reconfiguring the system in accordance with the changes in the environment is not easy and always feasible; moreover the cost of errors can be very high;
– to limit the problems, system stoppings must be planned in advance (e.g., roads impracticability is notified to drivers weeks in advance) and have to be scheduled in noncritical moments (e.g. signals maintenance is performed during night).

Solution

It is fairly evident that to render a nonstoppable system in compliance with the above forces a specific mechanism for adapting the system to environmental changes is needed. Adaptation takes place on a representative of the system, i.e., a group of *reification categories* [5]. In this

way, the adaptation mechanism does not interfere with the current execution of the system it is adapting, preserving the nonstoppable property of the system. Moreover, working on a representative rather than directly evolving the system provides an implicitly fault tolerance of the adaptation mechanism. Once the adaptation has been completed, the synchronization of the representative with the original system is delegated to the reflection pattern, before really modifying the original system, the soundness of the adaptation is verified by the consistency checker pattern. Examples of this approach can be found in [18].

Implementation

Here we show an algorithm illustrating the basic steps carried out by the evolutionary pattern. This algorithm is realized by using the C++ language.

```
template<typename aspect> class evolutionary {
  public:
    evolutionary(reification_category<aspect> a) : _representative(a) {}
    void do_adaptation(event e) {
          // it retrieves from the plan the rule to face with the event.
      void (*adaptation)(reification_category<aspect>, event);
      adaptation = _plan.get_action(e);
      adaptation(_representative, e);          // it carries out the adaptation.
      _representative.changed();               // it notifies the attempt of evolution
    }
    void inconsistency_detected() {
        // when called an inconsistency has been detected, it tries to solve such an inconsis-
        // tency exploiting its plan.
    }
  private:
    reification_category<aspect> _representative; // representative of the base-level aspect
    plans<aspect> _plan;                          // the rules it follows for adaptations
};
```

The evolutionary class is parametric on the representative it has the intention of adapting. This means that a class describing the aspect to adapt has also to be provided and to be used to instantiate the evolutionary class (see the second row of the code below).

The evolutionary pattern works on a reification category, i.e., on a representative of the software system (more on the representative is explained in section 3.1 when we describe *reification/reflection* patterns). The representative, of course, depends on the aspect of the system this class will deal with.

Another important element are the rules adopted by the algorithm for adapting the representative. These rules are represented by an instance `_plan` of the `plans` class. The `do_adaptation()` of the evolutionary object asks `_plan` for the adaptation rule to apply when an event happens (see row 7 in the code above). Then the evolutionary applies the adaptation rule to the representative.

```
reification_category<structure> str; evolutionary<structure> structure_evolution(str);
reification_category<behavior> beh; evolutionary<behavior> behavior_evolution(beh);

bool on_external_event(event &e) {
     // when an external event has occurred returns true then its argument refers to the
     // occurred event.
}
```

```
int main() {
  event e;
  while (true)
    if (on_external_event(e)) {
       structure_evolution.do_adaptation(e);
       behavior_evolution.do_adaptation(e);
    }
}
```

The evolution takes place when an external event, i.e., an event that has not been generated by the nonstoppable system, happens. The `inconsistency_detected()` method is invoked by the implementation of the consistency checker pattern.

In a complex system, as shown in the code above and in Fig. 2, there will be as many instances of the evolutionary pattern as many aspects of the system have to be adapted.

Applicability

The evolutionary pattern can be used to dynamically reconfigure a system (not necessarily a nonstoppable system) as a reaction to external events, such as anomalies detected by electronic devices. Moreover, the evolutionary pattern can be used to dynamically extend a running system with new features, components, and relations between them. The evolutionary pattern can be used to adapt the behavior as well the structure of the system as well the components interaction.

Known Uses

The evolutionary pattern is applied in the traffic control system realized in ESCORT [17] for controlling the traffic lights in accordance with the density of the vehicular flow as shown in Figure 3. This pattern interacts with elements of the object category like roads, traffic lights, urban control, traffic network, and elaborates the photography survey at real-time. In this case, the adopted evolutionary plan consists of comparing images of the traffic flow taken at different (adjacent) times (t_k). This comparison is done by pruning noises from images by using filters, by segmenting and classifying the images, and estimating the motion for images. Then, evolutionary pattern estimates the density of vehicular flow in a certain road in accordance with these values the traffic lights stay red or green for more or less time.

Collaborations

The evolutionary pattern, as shown in the applicability section, can be used stand alone, but in our work is only a part of a larger overall, so it implicitly interacts with: reflection, reification, and consistency checker patterns.

As explained, the evolutionary pattern observes the environment changes and adapts the base-level representative. The consistency of the representative is validated by the consistency checker pattern when the evolutionary pattern finishes the adaptation. If the validation fails the control returns to the evolutionary pattern for fixing the problem otherwise the base-level is conformed to the representative by the reflection pattern. The representative adapted by the evolutionary pattern is kept up to date by the reification pattern.

Consequences

The evolutionary pattern provides the following advantages:

– provides an implicit mechanism for dynamically evolving a system;

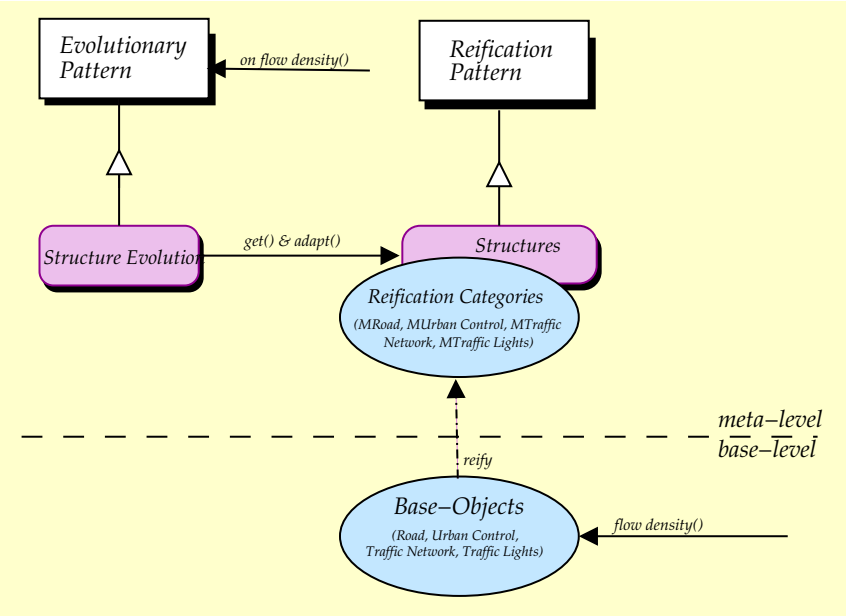


Fig. 3. The application of the evolutionary pattern in the UTCS

– provides a uniform way to evolve every aspect of a system, they could also be evolved separately;

and drawbacks:

– the nonstoppable system has an overhead when external events occurs and adaptation is needed;
– the system need extra code and data structures representing the system, its behavior and the evolutionary rules.

A critical role is played by the adaptation rules, they are the core of the evolutionary pattern and their realization is very hard because badly designed or applied at the wrong time adaptations could have very dire consequences, e.g., consider the chaos generated by stopping the traffic lights in a very busy area during the rush hour.

Related Patterns

The dynamic object model [16], allows the types of a system objects to change at run-time. This has been done by adding new types, changing existing ones, and changing the relationships between them. The work by Foote and Yoder [9], present three evolution patterns, *software tectonics* shows how continuous evaluation can be achieved without failure. *Flexible foundations* presents the need for continual and incremental evolution for systems. *Metamorphosis* presents mechanisms that allow the system to evolve dynamically.

The reflective state pattern [14], that is a refinement of state design pattern [10] based on the reflection architectural pattern [3]. The work by Yacoub and Ammer [20] represents the statecharts patterns and their relation to finite state machine patterns. This is done through defining these patterns: *basic statechart*, *hierarchical statechart*, *orthogonal behavior*, *broadcasting*, and *history-state*.

CONSISTENCY CHECKER PATTERN

Intent. *To verify the feasibility and the soundness of the changes "proposed" by evolutionary patterns. That is, to check if it is possible to apply such changes without rendering inconsistent the base-level system.*

Problem

The delicacy of dynamically changing (part of) a component of a system is fairly evident. Usually changes directly affect only (part of) a component rendering simple to verify the effects of the changes. In complex systems each component cooperates with, integrates/is integrated in, uses/is used by other components, therefore, the effect of changes performed on a component are propagated to many other components not directly involved by the modification. Hazardous changes to a component will conflict with the overall behavior of the system and such conflicts are quite difficult to be detected. This problem is further amplified by the fact that the system can not be stopped hindering an easy reconfiguration and validation of the complete system. For example, in the UTCS, at a crossroads we can not turn green a traffic light without considering the state of the correspondent traffic light for pedestrians.

Forces

The forces involved by the consistency checker are:

- changes to the system are proposed as a reaction to changes in the environment;
- changes to the system are made on a component basis whereas their impact usually affects more than a single component;
- changes to the environment can frequently occur and can impact on many system components;
- inconsistencies due to hazardous changes are difficult to be detected.

Therefore, it is important to verify that environmental changes impacting on many components do not generate conflicts in the overall system and the effect of these environmental changes has only to be propagated to the system components when it is safe, i.e., when the propagation do not leave the system in an inconsistent state. Moreover, it is important to schedule the adaptation before its effects become obsolete or unnecessary.

Solution

It is fairly evident from the problem description that every "proposed" change to a component of the system has to be well planned and validated against inconsistency. Hence we need a mechanism that applies the changes to the system only when the "proposed" changes are proved to leave consistent the system. Moreover, such a mechanism has not only to guarantee against inconsistencies due to erroneous updates but also to choose the right moment for applying the "proposed" changes before their effects become obsolete.

The basic idea consists of gathering many "proposed" changes on representatives (the corresponding reification categories) of the system, checking step by step that replacing such a pool of representatives with the corresponding aspects of the system will leave the system in a consistent situation. Then, the replacement will take effectively place when the system is in a state that can safely be carried out and as long as such a replacement is necessary.

The effective updating of the system is delegated by the consistency checker pattern to the reflection pattern, as shown in the corresponding section this pattern will also choose the right

moment for render effective the update. Whereas, changes, that the consistency checker pattern considers that could render the system inconsistent, are returned to the evolutionary pattern for fixing.

Implementation

The code example below illustrates the basic steps carried out by the consistency checker pattern for verifying and maintaining consistent the base-level system after evolution. This algorithm is realized by using the C++ language.

```
class consistency_checker {
     // consistency checker working only on two reification categories: behavior and struc-
     // ture.
  public:
    consistency_checker(reification_category<structure> s,
       reification_category<behavior> b) : beh(b), str(s) {}
    bool check_consistency() {
       str.reset(); beh.reset(); // reset the notification of a change received by the evolutionary
       return plan.check_consistency(str, beh);
    }
  private:
    reification_category<structure> str;
    reification_category<behavior> beh;
    plans<consistency> _plan; // consistency plan
};
```

The consistency checker works on the whole system. It does not work only on a specific aspect but rather it has to maintain the consistency among every reification category of the system. Therefore, the C++ class describing the consistency checker must access to all the system representatives.

Consistency rules are an important element managed by the consistency checker. These rules are represented by an instance `_plan` of the `plans<consistency>` class and are used to determine if the representatives (in our code are represented by a behavior: `beh` and a structure: `str`) are a consistent snapshot of the system. The method `check_consistency()` of the consistency checker delegates `_plan` for such a check on the representatives (see row 8 of the code above).

```
reification_category<structure> str; evolutionary<structure> structure_evolution(str);
reification_category<behavior> beh; evolutionary<behavior> behavior_evolution(beh);
consistency_checker cc(str, beh);
reflection<structure> obj(str); reflection<behavior> state(beh);

int main() {
  while (true)
    if (str.is_changed() || obj.is_changed())
      if (!cc.check_consistency()) {
        structure_evolution.inconsistency_detected();
        behavior_evolution.inconsistency_detected();
      } else {
        obj.reflect();
        state.reflect();
      }
}
```

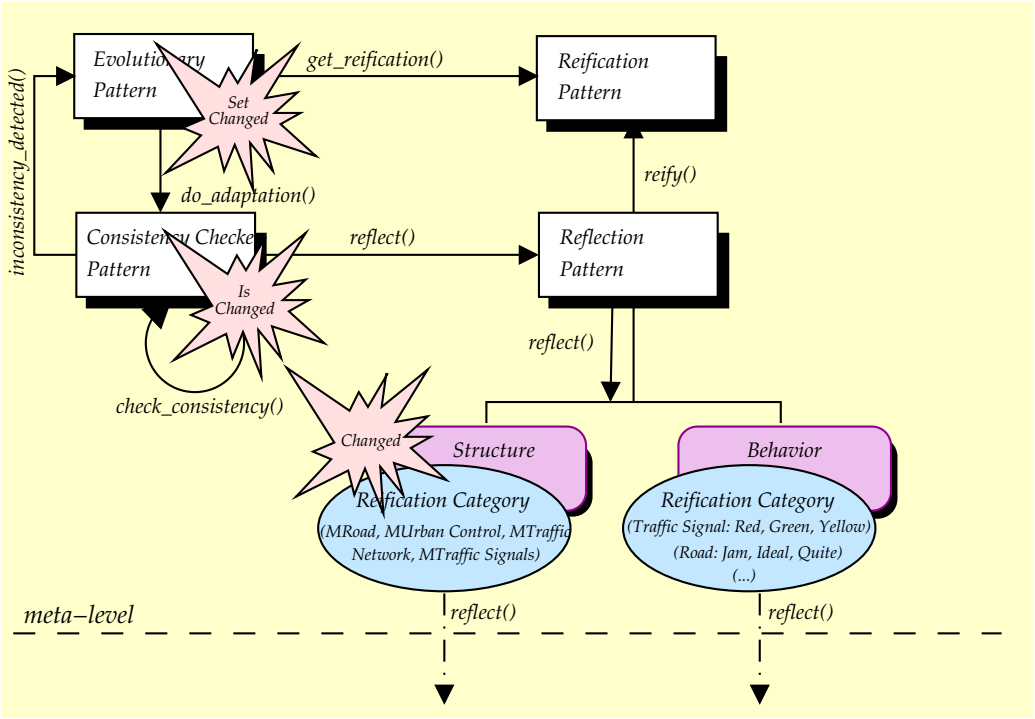


Fig. 4. An example of consistency checking against evolution of the system structure in UTCS.

Both the evolutionary and the consistency checker patterns work on system representatives. Evolutionary objects carry out their work when external events occur whereas the consistency checker performs its work when one of the representatives that it is monitoring is modified, that is, when an evolutionary object proposes a change. If evolutionary objects notify that they have carried out a change to the consistency checker, it is able to simply detect such a change in the representatives.

In Figure 4 we show the integration of the consistency checker pattern with the evolutionary and the reification/reflection patterns. The figure sketches consistency checker role in the UTCS by considering only two reification categories.

Applicability

The consistency checker pattern has to be used when we have to check the consistency of dynamic changes carried out by a system on a representative of another system before effectively performing such changes. The consistency checker pattern has to be used in critical environments to avoid the dire consequences of erroneous and inconsistent updates.

Known Uses

A feasible use of the consistency checker consists of checking the consistency of the base-level of a reflective system against changes performed by the meta-level system before reflection takes place. However this pattern is not mined from existing systems.

Collaborations

The consistency checker pattern can be used stand alone for checking the consistency of a sys-

tem or in collaboration with reification/reflection and evolutionary patterns for safely evolving a system:

- it compares the consistency of the reification categories embodied by the reification pattern. It uses a set of predefined rules;
- it interacts with evolutionary patterns to fix potential inconsistencies between the "proposed" adaptation and the referents;
- it delegates/authorizes the reflection patterns to update the corresponding aspects after the validation of the "proposed" adaptation.

Consequences

The consistency checker pattern provides the following advantages:

- It checks the consistency of the reification categories after evolution and before updating the base-level system in accordance with the adaptation. That is, it checks that carrying out the adaptation proposed by evolutionary patterns will not render the base-level system inconsistent.
- The control flow returns to the evolutionary pattern for fixing the proposed evolution if the consistency check fails. Otherwise, the control flow passes to the reflection pattern which carries out the proposed evolution.
- It looks, in collaboration with the reflection pattern, for the right moment to allow system evolution, i.e., the moment which guarantees that evolution leaves the evolved system working and consistent. It also associates an expiry date to the "proposed" adaptations that must still be applied and applies them only if such a date is not expired yet.

The consistency checker pattern has a few drawbacks:

- It augments the run-time overhead due to its checking and to its cooperation with evolutionary patterns for fixing the inconsistencies.
- Its work is based on a rigorous set of rules establishing when the system can be considered inconsistent. To write this set of rules is a delicate and difficult job.
- Adaptation does not immediately occur, could be postponed for long time and could never occur.

An important point is represented by the quality of the rules composing the validation system. This requirements is a very delicate point which requires a highly skilled software architect because all the efficacy of the consistency checker pattern is based on the quality of the validation system and a bad designed validation system can have dire consequences.

Related Patterns

CHECKS [6] is a pattern language for information integrity, presents two family of patterns. The first family of patterns considers quantities used by the domain model. They check your business logic capturing minimal variations in behavior (*Whole Value* pattern), non-applicable or exceptional quantities (*Exceptional value* pattern), and inappropriate combinations of values (*Meaningless Behavior* pattern). Whereas, the second family enables the direct and transparent manipulation of the domain model. This has been done by using these patterns: *Echo Back*, *Visible Implication*, *Deferred Validation*, *Instant Projection*, and *Hypothetical Publication*.

3.1 Pattern Language for Computational Reflection

This little pattern language has an intrinsic dualistic nature and it is composed of two patterns: *Reification* and *Reflection*. Their combined work allows the system to *export* (to *reify* by using a reflective parlance) a reification of a specific aspect of the system to another system for manipulation and to *import* (to *reflect* by using a reflective parlance) such a reification after the manipulation in the system again. After the terminology we have adopted in [5] reifications are called *reification categories*. These two patterns are not stand-alone but their work has to be integrated with the work of the other system. By the way, the sequence of application is reification then reflection (we have to export before manipulating and importing again).

The pattern language for computational reflection manages the interface between base- and meta-level composing the reflective system architecture. In the framework language for evolution we are describing, this pattern language can be considered as the glue sticking together the evolving system (the base-level using reflection parlance) and the system dealing with evolution (the meta-level) whereas the reification is the representative of one of the aspects of the base-level system. Working on a reification of the nonstoppable system allows the other patterns to "simulate" the adaptation without really affecting the real system.

REIFICATION PATTERN

Intent. *To export a representation of a system aspect both abstract and concrete to another system. Behavior, state, and code are some of the system aspects that can be exported.*

Problem

To monitor and manipulate an inner aspect of another software system means to be able to monitor and manipulate both low-level details (as in the case of system code) and abstract concepts (as in the case of the system behavior) and to access to the inner representation of another software system. This is not a simple job to carry out by using a nonreflective approach because they do not provide a mechanism which allows a system to access the inner representation of another software system forcing the programmer to tightly coupling the code of these two systems (supervisor and supervised). Moreover, it is also missing a high-level representation for some abstract aspects such as the behavior.

Forces

Manipulation of many aspects of a software system from another system is often forbidden due both to a different representation and to protection mechanisms. Moreover, it is a difficult job for a system to manipulate abstract concepts as behavior and collaboration via the API of a traditional programming language. We need to work on representatives without affecting the original system.

Solution

Rather than giving the supervisor access to the inner representation of the supervised, the supervised itself has to provides its own representation to the supervisor. This approach moves the responsibility of representing and therefore interpreting the inner aspects of a system from the supervisor to the supervised system, that is, from a system uncorrelated to such a data to the system owning them, with an obvious simplification. Hence for the system aspect we would reify, the system itself provides a data structure representing such an aspect and some operations

to interpret and manipulate it. Moreover, the representative (that is, the copy of the aspect that is local to the supervisor system) has to be kept constantly consistent with the aspect it represents, that is, the data structure has to be updated when a change in the aspect occurs. For example, in the UTCS, potential reification categories are roads and traffic lights status and their reifications have to be updated every time a road or a traffic light change its state.

Implementation

The implementation of the reification pattern is simple as well as the reflective API of the chosen programming language is articulated, this means that adopting Java instead of C++ the implementation could be simpler.

The code example below illustrates the basic steps carried out by the reification pattern for exporting an aspect of a system to another system. This algorithm is realized by using the C++ language.

```
namespace computational_reflection {

      // reification_category represents the base-level. It is a generic class that can be in-
      // stanced on behavior, structure, collaboration and so on.

  template<typename aspect> class reification_category {
    public:
      reification_category() : _changed(false) {}
      bool is_changed() {return _changed;}
      void changed() {_changed = true; ...}
      void reset() {_changed = false;}
      void local_update() { /* updates the content of the local copy */ }
      void merge_local() { /* merges the local copy to the reified aspect */ }
    private:
      aspect _reified;
      bool _changed;
  };

  template<typename aspect> class reification {
    public:
      reification(reification_category<aspect> a) : rc(a) {}
      void reify() {    // it reifies the corresponding base level aspect.
         rc.local_update();
      }
    private:
      reification_category<aspect> rc;
  };
}
```

Many reification categories can be necessary to represent every aspect of the system and each aspect can need a very different data structure to be represented. However these reification categories provide a common interface for manipulating themselves (such an interface is shown in the code above). The reification categories provide the mechanism used by the evolutionary pattern to notify a change in the representative to the consistency checker pattern (methods `changed()`, `reset()`, and `is_changed()`). Besides, they also provide the methods for keeping synchronized the copy with the original object (methods `local_update()` and `merge_local()`). These two methods directly deal with the base-level and their implementation can be easier if the adopted programming languages has reflective features.

In a system there are as many reification instances as reification categories, and each instance takes care of reifying the corresponding aspect. Therefore, the reification class is parametric on the aspect it is reifying. The reification class provides only the method `reify()`. It is used for reifying the aspect, its implementation does not vary when changes the reified aspect because it delegates the work to the `local_update()` of the corresponding reification category, the `local_update()` is directly related to the reified aspect and its behavior changes when the reified aspect is of a different kind.

```
bool on_event(event &e) {
    // when the base-level changes it returns true then its argument refers to the kind of
    // change occurred.
}

int main() {
  event e;
  reification_category<structure> str;
  reification_category<behavior> beh;
  reflection<structure> obj(str);
  reflection<behavior> state(beh);

  while (true)
    if (on_event(e)) {
       obj.reify(e);
       state.reify(e);
    }
 }
```

The representative of the aspect is updated every time the corresponding aspect changes due to the normal computation of the base-level. The updating takes place reifying the changed aspect on the reification category again.

Applicability

The reification pattern is a basic component for realizing the causal connection between two systems (see section 2 for a brief explanation about the causal connection relation). It can also be used every time it is necessary a local representative of a nonlocal entity, e.g., for implementing a remote method invocation in a client/server system, in this case the client asks the server for services through a representative. It is the server itself which renders available (exports) to the client such a representative (e.g., in Java through the RMI registry and the bind mechanism, see [11]) as in the reification pattern. Moreover, the reification pattern can be used to provide a uniform access to remote and distributed data, e.g., to implement a clustering file system *à la* MOSIX [1]. MOSIX provides to each computer in the cluster a virtual view of the clustering file system. Such a view is a composition of representatives of the singular file systems in the cluster and each change (e.g., to remove files) to the clustering file system affects only the corresponding representative and not the original file system. These representatives are provided by the kernel of the MOSIX system as it is done by the reification pattern.

Collaborations

The reification pattern is tightly coupled with the reflection pattern. Combining reification and reflection patterns grants the causal connection between these two systems. Moreover in the overall of our pattern language for evolution the reification pattern directly collaborates with:

– the evolutionary pattern providing the representatives it will use for evolution, and

– the consistency checker pattern providing a copy of the aspects that the consistency checker will use for validating the adaptation proposed by the evolutionary pattern.

Consequences

To export a representative of (part of) a system to another system has several benefits:

– it permits to simulate the adaptation of a system;
– it easily permits to verify/testing the system before rendering effective the change;
– it allows to modify the reified entity on a representative and testing the modification before rendering effective the change; if the changes are not satisfactory they can be easily discarded, discarding the representative without really affecting the reified entity.

Obviously there are also some drawbacks: the two most relevant are the extra overhead due to keep updated the representatives and the complexity of writing a reification category without using a reflective programming language.

Related Patterns

Reification means to turn something that you would normally not think as an object into an object. Several patterns in the literature approach to reification [10]: *Strategy*, *Mediator*, *Memento*. For example, memento, by widening its application domain to many other system aspects than the system state, can be used to realize our *reification categories*. In [13] a pattern language for implementing communication protocols has been presented, the exchanged data have been reified by the *data-path reification* pattern. The *Delegation pattern* [7], allows objects to share behavior without using inheritance and without duplicating code.

REFLECTION PATTERN

Intent. *To import a representation of a system aspect from another system. Behavior, state, and code are some of the system aspects that can be imported.*

Problem

How to take changes performed on a representative and implement them on the represented entity is not a simple job. We have to face two main problems:

– both represented entity and its representative belong to two separate systems, and
– there is not a simple and well-known mapping between an aspect of a system and its representative in another system.

The representation of an aspect of a system is handled by a component of another system. Each of these systems has its own access rights and usually it does not have the right to access the other system data.

Moreover, represented aspects can be both abstract or concrete concepts, where they are concrete when in the reified system there is a clear part of the code implementing those aspects, such as an object, or a method. Both kind of aspects are represented by a data structure. Therefore, there is no straightforward mapping between a change performed on the representative and its implementation on the represented aspect when it is an abstract concept without a direct counterpart in the system code (e.g., the system behavior).

These facts hinder the reintegration of the changes performed on the representative with the represented aspect.

Forces

The forces involved by the reflection pattern are:

- access rights and protection mechanisms hinder to import the changes made on a representative in the represented entity;
- the knowledge of many details of both systems is necessary to give a mapping between an aspect of a system and its representative in the other system;
- the mapping among these systems tightly couple them together;
- to implement a change carried out on a representation of an abstract concept is not easy;
- to update (or to replace) an aspect of another system is a very intrusive operation;
- we need a wide knowledge of the system to update, of its computational flow and when it is safe interacting with it.

Solution

Rather than enabling the system owning the representative to directly updating the represented aspect of the other system, the represented system itself will import the representative and will merge its content with the represented aspect. This approach overcomes the protection mechanisms because the system surely has all the necessary rights for modifying itself. Moreover, the system has a direct knowledge about its execution (e.g., if it is running or idle) rendering less intrusive the system updating, that is, the system itself will decide when it is time to update itself and if the update is not obsolete with respect to its current state (e.g., it avoids to update an object which does not exist anymore). Then the system itself will ask the supervising system for the content of its representative and will be simpler for it to map the changes on its inner representation than for another system.

Implementation

The consideration we have done for the implementation of the reification pattern still hold for the implementation of the reflection pattern.

The code example below illustrates the basic steps carried out by the reflection pattern for importing the changes done by a system on a representative into the original system. This algorithm is realized by using the C++ language.

```
namespace computational_reflection {
  template<typename aspect> class reflection {
    public:
      reflection(reification_category<aspect> a) : rc(a) {}
      void reflect() {rc.merge_local();}
    private:
      reification_category<aspect> rc;
  };
}
```

Both reflection and reification patterns are part of the same namespace and an instance of the reification class *shares* the reification category with an instance of the reflection class.

The reflection class, as also the reification class, is parametric on the aspect represented by the reification category. Moreover, it provides the method `reflect()` which reflects the changes carried out on the local representative on the corresponding aspect. Obviously the implementation of `reflect()` depends on the aspect we are managing, therefore, on the type of the aspect

and on its operations (`merge_local()` is a method belonging to the class of the aspect).

Applicability

The reflection pattern, as well as the reification pattern, is a basic component for realizing the causal connection between two systems. Moreover, it can be applied to synchronize the content of data structures shared among several processes and for serializing the concurrent writing access to a centralized datum.

Collaborations

The reflection pattern is tightly coupled with the reification pattern and together they grant the causal connection between two systems. Moreover in the overall of our pattern language for evolution the reflection pattern directly collaborates with the consistency checker pattern. The reflection pattern updates the reified system with the changes done to the corresponding local representative after the validation of the local representative performed by the consistency checker.

Consequences

To import a representative of (part of) a system from another system has several benefits:

- it permits to postpone the adaptation performed by another system to the most suitable moment;
- it frees the rest of the application from matters about data representation, protection mechanisms and so on.

The two most relevant problems are the extra overhead necessary for updating the original system and the difficult of modifying the original system in accordance with the content of the local representative above all without using a reflective programming language.

Related Pattern

The *Static Reflection* pattern [12], addresses the particular problem of building wrappers which contain functions which take C function pointers as parameters. The *Reflection Pattern* in [3] on page 193, provides a mechanism for changing structure and behavior of software systems dynamically. It supports the modification of fundamental aspects, such as type structures and function call mechanisms. The *Microkernel Pattern* in [3] on page 171, applies to software systems that must be able to adapt to changing system requirements. It separates a minimal functional core from extended functionality and customer-specific parts.

4 Conclusion and Future Works

In this paper we have addressed the problem of evolving a nonstoppable system at run-time through a reflective architecture. We propose a pattern language that describes the meta-level of the reflective architecture and how the components of the meta-level cooperate for evolving the base-level system. The meta-level system and its connection to the base-level system are modeled by *evolutionary*, *reification/reflection*, and *consistency checker* patterns. They cooperate in evolving the base-level system without affecting its consistency and functionalities. The evolution of a system can be settled by techniques based on different information, whereas the overall structure of the evolving mechanism is quite general. Therefore, in this work we have

just modeled the evolving architecture avoiding to fix a technique and detailing such a mechanism. In general we refer to reification categories, consistency rules, and so on, but their detailed description depends on the adopted techniques for evolution and it is beyond the scope of this work. An example of evolution based on design time information has been given by the authors in [5]. The pattern language for system evolution has been mined from the Escort system [17] which exploits a reflective architecture for supervising the traffic flow in the city of Milan.

Acknowledgments

Authors wish to thank Kristian Elof Søresen who has perfectly shepherded us to shape the paper at its best. They would also thank Mikio Aoyama, Richard Gabriel, Lars Grunske, Kevlin Henney, Juha Pärssinen and Michael J. Pont for the kind words they had for the early version of this pattern language and for their advice to render the paper what it is now.

References

1. Ammon Barak, Shai Guday, and Richard G. Wheeler. *The MOSIX Distributed Operating System, Load Balancing for UNIX*, volume 672 of *Lecture Notes in Computer Science*. Springer-Verlag, 1993.
2. Grady Booch, James Rumbaugh, and Ivar Jacobson. *The Unified Modeling Language User Guide*. Object Technology Series. Addison-Wesley, Reading, Massachusetts, third edition, February 1999.
3. Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal. *Pattern-Oriented Software Architecture: A System of Patterns*. John Wiley and Sons Ltd, 1996.
4. Walter Cazzola. Evaluation of Object-Oriented Reflective Models. In *Proceedings of ECOOP Workshop on Reflective Object-Oriented Programming and Systems (EWROOPS'98)*, in 12th European Conference on Object-Oriented Programming (ECOOP'98), Brussels, Belgium, on 20th-24th July 1998. Extended Abstract also published on ECOOP'98 Workshop Readers, S. Demeyer and J. Bosch editors, LNCS 1543, ISBN 3-540-65460-7 pages 386-387.
5. Walter Cazzola, Ahmed Ghoneim, and Gunter Saake. Reflective Analysis and Design for Adapting Object Run-time Behavior. In Zohra Bellahsène, Dilip Patel, and Colette Rolland, editors, *Proceedings of the 8th International Conference on Object-Oriented Information Systems (OOIS'02)*, Lecture Notes in Computer Science 2425, pages 242–254, Montpellier, France, on 2nd-5th of September 2002. Springer-Verlag. ISBN: 3-540-44087-9.
6. Ward Cunningham. The CHECKS Pattern Language of Information Integrity. In James O. Coplien and Douglas C. Schmidt, editors, *Proceedings of the 1st Annual Conference on Pattern Languages of Programs (PLoP '94)*, Monticello, Illinois, USA, August 1994.
7. Dwight Deugo. Foundation Patterns. In Steve Berczuk and Joe Yoder, editors, *Proceedings of the 5th Annual Conference on Pattern Languages of Programs (PLoP'98)*, Monticello, Illinois, USA, August 1998.
8. Jim Dowling and Vinny Cahill. The K-Component Architecture Meta-Model for Self-Adaptive Software. In Akinori Yonezawa and Satoshi Matsuoka, editors, *Proceedings of 3rd International Conference on Metalevel Architectures and Separation of Crosscutting Concerns (Reflection'2001)*, LNCS 2192, pages 81–88, Kyoto, Japan, September 2001. Springer-Verlag.
9. Brian Foote and Joseph W. Yoder. Evolution, Architecture, and Metamorphosis. In John M. Vlissides, James O. Coplien, and Norman L. Kerth, editors, *Proceedings of the 2nd Annual Conference on Pattern Languages of Programs (PLoP '95)*, Monticello, Illinois, USA, September 1996. Addison-Wesley Software Patterns Series.
10. Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Professional Computing Series. Addison-Wesley, Reading, Ma, USA, 1995.
11. jGuru. Fundamentals of RMI (Short Course). Available on `http://developer.java.sun.com/developer/onlineTraining/rmi/RMI.html`.
12. Bob Jolliffe. The Static Reflection Pattern. In Dwight Deugo and Federico Balaguer, editors, *Proceedings of the 8th Annual Conference on Pattern Languages of Programs (PLoP'01)*, Monticello, Illinois, USA, September 2001. Addison-Wesley Software Patterns Series.
13. Matthias Jung and Ernst W. Biersack. Order-Worker-Entry: A System of Patterns to Structure Communication Protocol Software. In Martine Devos and Andreas Rüping, editors, *Proceedings of the Fifth European Conference on Pattern Languages of Programs (EuroPLoP 2000)*, Irsee, Germany, July 2000.
14. Luciane Lamour Ferreira and Cecília M. F. Rubira. The Reflective State Pattern. In Steve Berczuk and Joe Yoder, editors, *Proceedings of the Pattern Languages of Program Design*, TR #WUCS-98-25, Monticello, Illinois - USA, August 1998.
15. Pattie Maes. Concepts and Experiments in Computational Reflection. In Norman K. Meyrowitz, editor, *Proceedings of the 2nd Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'87)*, volume 22 of *Sigplan Notices*, pages 147–156, Orlando, Florida, USA, October 1987. ACM.

16. Dirk Riehle, Michel Tilman, and Ralph Johnson. Dynamic Object Model. In Eugene Wallingford and Alejandra Garrido, editors, *Proceedings of the 7th Annual Conference on Pattern Languages of Programs (PLoP 2000)*, Monticello, Illinois, USA, August 2000.
17. Andrea Savigni, Filippo Cunsolo, Daniela Micucci, and Francesco Tisato. ESCORT: Towards Integration in Intersection Control. In *Proceedings of Rome Jubilee 2000 Conference (Workshop on the International Foundation for Production Research (IFPR) on Management of Industrial Logistic Systems – 8^{th} Meeting of the Euro Working Group Transportation - EWGT)*, Roma, Italy, September 2000.
18. Francesco Tisato, Walter Cazzola, Andrea Savigni, and Andrea Sosio. Architectural Reflection. Realising Software Architectures via Reflective Activities. In Wolfang Emmerich and Stephan Tai, editors, *Proceedings of the 2nd International Workshop on Engineering Distributed Objects (EDO 2000)*, Lecture Notes in Computer Science 1999, pages 102–115. Springer-Verlag, University of California, Davis, USA, on 2nd-3rd of November 2000.
19. Emiliano Tramontana. Managing Evolution Using Cooperative Designs and a Reflective Architecture. In Walter Cazzola, Robert J. Stroud, and Francesco Tisato, editors, *Reflection and Software Engineering*, Lecture Notes in Computer Science 1826, pages 59–78. Springer-Verlag, Heidelberg, Germany, June 2000.
20. Sherif M. Yacoub and Hany H. Ammer. A Pattern Language of Statecharts. In Steve Berczuk and Joe Yoder, editors, *Proceedings of the 5th Annual Conference on Pattern Languages of Programs (PLoP'98)*, Monticello, Illinois, USA, August 1998.

The Executor Pattern

Decoupling tasks from execution

Eric Crahen
VikingPLOP 2002
crahen@cse.buffalo.edu

Abstract:

Many modern programmers seek to improve the performance of their code by breaking parts of the application down into several tasks that can be executed concurrently. Executing these tasks in separate threads often helps improve the performance, and responsiveness of an application. The Executor pattern describes the decoupling of these tasks from the method of execution. Where the Command pattern [GoF95] focuses on abstracting the details of how to perform a task, The Executor pattern complements it by abstracting the context and the means of execution for a task.

Example:

Consider how requirements can change throughout the lifetime of a web server. In order to maximize the throughput and to increase the responsiveness it is desirable to use a multithreaded design. Using threads to wait for incoming connections and to handle requests can help achieve this goal. To do this, a simple framework for creating tasks that handle these things would be useful.

One method of creating a task to be run in another thread is to extend a Thread class and build the task into that specialization. This is can be effective for long lived tasks, such as ones that listen for and accept incoming connections. This type of heavy-weight task might look something like this,

```
// A very simple heavy-weight task
class ServerThread extends Thread {
  // ...
  public void run() {
    try {
      while(!Thread.interrupted())
        handleIncomingConnections();
    } catch(InterruptedException e) { }
  }
  // ...
}
```

However, for small, short lived tasks this not quite as effective. Once a connection is established, it might be used to fill several requests. Each request would correspond to a separate task. The Reactor and Proactor patterns [POSA99] provide a much more detailed explanation. Creating a series of heavy-weight tasks, binding a different thread to each short task invokes a lot of overhead.

Often, a group of small tasks can be handled by creating a pool of threads. Creating a pool of threads ahead of time helps to avoid the excessive overhead of creating a new thread for each task by creating a set of threads ahead of time and reusing threads to execute each task. A typical thread pool might look like this,

```
public class ThreadPool {
  private LinkedList queue = new LinkedList ();

  public ThreadPool(int size) {
    while(size-- >= 0)
      new Worker().start();
  }

  public void run(Runnable task) {
    synchronized(queue) {
       queue.addLast(task);
       queue.notify();
    }
  }

  private class Worker extends Thread {
    public void run() {
      try {
        for(;;) {
          synchronized(queue) {
            while(queue.isEmpty())
              queue.wait();
            ((Runnable)queue.removeFirst()).run();
          }
        }
      } catch(InterrtupedException x) {
      } catch(RuntimeException e)  { /* shutdown on error */ }
    }
  }

}
```

Not all thread pools are implemented in the same fashion, and often times an implementation may be tailored to a specific purpose. Using threads and some types of thread pools directly can mean placing an additional burden on their users that can ultimately be responsible joining the thread that was spawn or for returning a borrowed thread to the pool. Each thread pool implementations tends to meet slightly different needs, and as a user, you are bound to the interface that pool provides. It makes it difficult to switch from one (thread pools, single threads, no threads) method of execution to another.

As the implementation is being tested and stabilized, an additional set of requirements appears. It may be very desirable to add some external synchronization to the tasks, allowing only a fixed number to run at any given time. Allowing only one to run at a time or adding some kind of pre or post processing action can be very helpful for debugging purposes.

Thread pools themselves aren't always particularly good at is adding some customized behavior to each task. The addition of logging or other performance monitoring of the tasks being executed is often useful. This allows statistics to be collected which can help determine how long a task is taking, which can be helpful in finding tasks that cause errors or in collecting error logs when tasks filling requests fail.

As the number of different types of tasks increase, creating some priority for those tasks may become important. Including some method to assign priorities to each task could be desirable. The needs of an application can grow and change quite a bit as this sort of program is developed. In order to simplify the implementation and limit how much these changing impact the code using it a good abstraction for executing tasks is needed.

Problem:

In order to create an abstraction that can remove the needs to deal with the details of how tasks are executed from the user, the following forces need to be resolved:

Simplicity:

A piece of code that desires to submit a task some object for execution has only one simple desire, to make a request for a task to be executed. Creating an interface which provides a method to do just that without placing any extra burden on the code making the request is important. Requiring any more would encourage binding to a specific means of execution. In other words, clients submitting a task to another object shouldn't be responsible for assisting in the correct execution of that task. Finding a way to simplify a clients' interaction would also be an essential first step in abstracting the means of execution.

Overhead:

Specializing a class that encapsulates a method of execution in order to perform a specific task via that method is a common approach to implementing the Command pattern (Swing Workers, for example Sun00). However, this couples a task directly to a specific vehicle of execution. While this is not necessarily a bad thing in and of itself, binding all of the overhead associated with that vehicle is not always appropriate. When many very short tasks need to be serviced, much time and many resources would be unnecessarily spent creating new threads. Since most of the servers time would be better spent filling requests, finding some way to avoid that overhead would be very desirable.

Mechanism:

Most tasks are run by threads, but not all of them are. Some methods of execution never have to manipulate threads directly. For instance, one method of execution is to use the current thread in conjunction with some form of synchronization to run each task. For example, a semaphore could be used to limit the number of tasks it ever allows to execute at once. Another is to use some form of RPC (CORBA or RMI) to submit tasks to another machine instead of directly running them. Finding a way to make these details interchangeable can be tricky.

Configuration:

Configuration can be somewhat awkward when directly using a resource that provides a method of execution. Often times it would be very convenient to supply some hints about how tasks should be executed. This can involve hints not only for adjusting how resources are utilized, but it also includes providing information about how tasks are serviced. Some kinds of tasks may have different priorities which should be reflected in the order of execution. When designing objects for executing tasks, which are able to adapt to changing demand through controlling the resources allocated to meet that demand, configuration becomes quite important.

Customization:

On some occasions it can be very convenient to associate some extra behavior with each task. For example, keeping a log of the start and stop of each task is one simple use. Notifying other objects as a task is executed, perhaps to update a statistic display, is another more complex example. In keeping with the force of simplicity, a way to achieve this without placing an additional responsibility on the client code is needed.

Solution:

Add a layer of indirection between a client and the execution of a task; instead of a client executing a task directly an intermediate is asked to execute a task for a client. By creating a framework that includes Task objects and Executor objects. A Task object 'knows' how to run a specific task, like a Command object, it exposes a single method to trigger its execution. An Executor object that 'knows' where to execute Task objects; it encapsulates the knowledge to of how to get a Task object into a context where it can be executed.

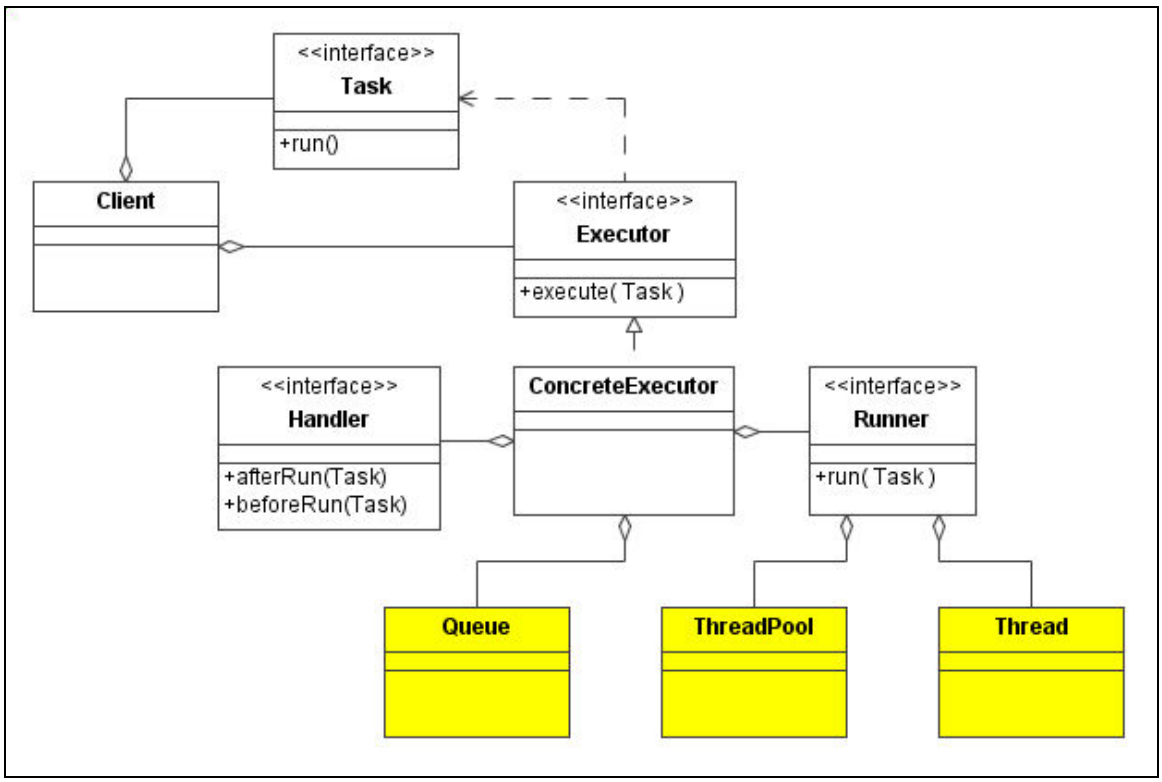
This solution is similar in some respects to the Command Processor pattern. However, Command Processor pattern [PLoPD96] illustrates more specifically how Command objects are managed. The Executor is more concerned with how Command objects are executed.

An Executor allows an application to deal only with light-weight Task objects (tasks) abstracting the any need to directly manipulate any specific thread, network connection or any other resource used to place a Task in the correct context for execution, decoupling the task from the means of execution. This abstraction allows programmers to focus their efforts on creating individual Tasks; the need for the explicit management of any resource is removed.

Tasks are considered light-weight because a task doesn't necessarily bring with it the overhead of an entire thread, or any other method of execution. For example, a task realized by extending a Thread would be considered heavy-weight because it is a Thread, and can't be decoupled from itself.

The context a created by an Executor to run a Task includes not only which thread runs a task, but also what happens before a task is run, what happens when a task is completed. Defining context creates the environment a task will be run in. Shifting these concerns from client into a context managed by an Executor allows changes to be made to the context with regard where a task is executed without impacting any client code. For instance, an Executor may change its context for execution from the current thread to a cached thread from a pool to a new thread for each task.

Multiple executors can be used to address different execution needs. For example, an Executor that uses a pool of threads might be used to execute tasks that download several files simultaneously. A series of transactions might be committed to a log using an executor that will serialize the execution of the tasks submitted to it.



Executors also provide for some customization of the order in which tasks are carried out by including a Queue component. This component can be interchanged easily to select LIFO, FIFO, priority based or some other ordering.

Participants:

A Task defines an interface for some operation. As an application of the Command pattern it is responsible for encapsulating knowledge of how to execute some task.

An Executor provides an interface for scheduling a task to be executed. The Executor encloses the knowledge of how to place a Task in the correct context for execution. This may involve moving the Task to another thread or even another machine

A Client creates both Tasks and Executors; it is responsible for submitting a Task to an Executor for future execution.

A Handler can be associated with an Executor to provide a place to add some pre and post processing options to each Task submitted. This could also have been achieved by wrapping each Task submitted with another Task that would do these things; however providing a separate object to delegate this work to helps to make the purpose clearer.

A Runner is the abstraction for the resource being used to execute these Tasks. For an Executor, this provides access to anything that will directly run a task (a Thread, a set of Threads, a remote object, etc). It hides the portions of the interface of those things which necessarily appropriate to expose in this framework. How a Runner is realized is dependant on the programming language being used and the multithreading facilities it provides. In this paper it will be described with a distinct interface to help clarify the intent behind it.

Variants:

There are a few variants on the core solution that help in creating certain kinds of Executors. These variants typically include the following additional participants,

- *Threads*; a Thread is optional portion of a Runner. Some executors may be designed to dispatch a Task to a separate thread of execution. A Runner might provide customized Thread classes to add different behaviors. How this is accomplished is dependant on platform and language.

- *ThreadPools*; a ThreadPool, like a Thread, is an optional part of a Runner. It is an application of the Pooling pattern, a collection of threads that are used to help execute a Task.

- *Queues*; a Queue is an optional part of an Executor that accumulates Tasks for future execution. It's responsible for accumulating and ordering Tasks before they are placed in the correct context for execution. It also acts a point to address other QoS concerns.

Allowing the user to supply custom threads to the executor is very useful. A factory could be provided for a Runner by a user that creates named threads to assist in debugging. Customized threads could be used for a lot of things, having this option leaves room for a lot of flexibility.

Executors can easily adapt to changing demands by adding or removing threads from its ThreadPool as the frequency of task submission changes. Alternatively, they might create fixed size pools, or pre-allocated sets of threads to speed up response.

Queues help fill any needs an Executor has with regard to the order of execution. By supplying various Queues a user can easily configure an Executor to run tasks in different orders (LIFO, FIFO, Priority, etc.)

As previously mentioned, not all Executors need to use threads, and instead may simple use synchronization to control the context of execution. In these cases, the optional roles described above can be left out, or simply implemented as Null objects. However, most Executors will be multithreaded in nature and use different kinds of Threads, ThreadPools and Queues.

Implementation:

The following is a sample implementation of two very simple Executors, written in Java. One queues and executes tasks in another thread. The other allows only one thread to execute a task at any given time.

```
public interface Executor {
  public void execute(Runnable task);
}
```

The Executor interface shared by all Executors describes it main responsibility, accepting tasks to execute at some point in the future.

```
public interface Handler {
  public void beforeRun(Runnable task);
  public void afterRun(Runnable task);
  public void afterFailure(Runnable task, RuntimeException e);
}
```

The Handler acts as a kind of hooking mechanism, allowing a user to intercept the executing thread at various stages. The interface shown here allows for customization of what happens before and after a task is executed. It also provides a simple way to handle exceptions that might occur when a task is being executed, without having to modify the Executor.

```
public class LockedExecutor implements Executor {
  // An internal object is used to prevent external interference
  // with the synchronization
  private Object lock = new Object();
  private Handler handler;


  public LockedExecutor() {
    this(new NullHandler()); // Null Object
  }

  public LockedExecutor(Handler handler) {
    if(handler == null)
      throw new NullPointerException();
    this.handler = handler;
  }

  public void execute(Runnable task) {
    try {
      synchronized(lock) {
        handler.beforeRun(task);
        task.run();
        handler.afterRun(task);
      }
    } catch(RuntimeException e) { handler.afterFailure(task, e); }
  }
}
```

This LockedExecutor is an Executor that executes each task in isolation from any other task. In other words, only one task can be run by any thread at any time by this Executor.

```
public interface class Runner {
  public void run(Runnable task);
}

public class DaemonThreadRunner implements Runner {
  public void run(Runnable target) {
    Thread worker = new Thread(target);
    worker.setDaemon(true);
    worker.start();
  }
}
```

The Runner interface creates an abstraction for what exactly is being used to run a task. A DaemonThreadRunner is shown here which uses daemon threads to run tasks. It is possible to provide Runners that use regular Threads, subclasses of Thread or even ThreadPools.

```
public class ConcurrentExecutor implements Executor {
  // Work queue
  private LinkedList queue = new LinkedList();
  private Handler handler;

  public ConcurrentExecutor() {
    this(new NullHandler()); // Null Object
  }

  public ConcurrentExecutor(Handler handler) {
    if(handler == null)
      throw new NullPointerException();
    this.handler = handler;
    Runner runner = new DaemonThreadRunner();
    runner.run(new ExecutorTask());
  }

  public void execute(Runnable task) {
    synchronized(queue) {
      queue.addLast(task);
      queue.notify();
    }
  }

  private class ExecutorTask implements Runnable {
    public void run() {
      try {
        synchronized(queue) {
          while(!Thread.interrupted()) {
            // Wait for work to arrive
            while(queue.isEmpty())
              queue.wait();
            Runnable task = (Runnable)queue.removeFirst();
            handler.beforeRun(task);
            task.run();
            handler.afterRun(task);
          }
        }
      } catch(InterrtupedException x) {
      } catch(RuntimeException e) { handler.afterFailure(task, e); }
    }
  };

}
```

This ConcurrentExecutor demonstrates how Runners and Queues can be used to create an Executor that will run tasks in other threads. This implementation uses only a single thread, but it can easily be made to use more than one by replacing the Runner used here with one using a ThreadPool.

Benefits:

Eliminating the direct dependence on any single type of execution mechanism, whether it is a thread, a thread pool or something else, introduces a good deal of flexibility into an application.

- A simple interface is provided for executing commands. A user does not need to do anything other than to ask an Executor to execute() a task to gain these benefits.

- Each task does not necessarily impose a significant amount of overhead. Binding a separate thread to each task is optional. A user may choose an Executor that creates new threads for each submitted task; but when resources are at a premium an Executors implementing a pooling pattern can be selected instead.

- The mechanism of execution is easily interchangeable, by exchanging Executors an application can select between varieties of execution strategies. You can move from running tasks in the current thread, to running tasks in a single worker thread, to running tasks in a thread pool with very little effort.

- Customization, using Handlers, Runners and Queues can be achieved without making any modifications to an Executor. Handlers can be provided to modify some of a tasks behavior, adding pre and post actions without requiring a user to wrap each individual task. Similarly Runners and Queues could be supplied to Executors to provide customized execution resources (specialized Threads, ordered Queues, etc) without having to modify the Executor classes.

Drawbacks:

The portions of the program that are to run concurrently must be broken down, effectively, into a set of light-weight Task objects. There are a few pitfalls that require a programmer's careful attention:

- Using Executors doesn't provide automatic thread safety. It is the responsibility of the programmer to ensure that tasks created are synchronizing access to shared data.

- Tasks should be somewhat thought out; for example, submitting a set of tasks that put the executing threads to sleep may counteract the benefits an Executor provides.

- Some kinds of tasks may not be suitable for all Executors. A task that runs without trouble in a non-threaded or single-threaded Executor may not always behave as expected in a threaded Executor. It is still the responsibility of the programmer to ensure tasks submitted to threaded Executors are actually thread-safe.

- To customize the behaviors of some kinds of Executors may still require a user to interact with the concrete type rather than the general interfaces. For example, to adjust the number of threads being used by an Executor that uses a thread pool would require the user to work with a concrete PooledExecutor (or similar) class since the Executor interface doesn't describe that functionality.

Known Uses:

Executors sometimes can be found in other patterns. The Active Object pattern [POSA99], which provides an asynchronous execution wrapper for an object, uses a kind of executor to schedule the execution of the various *future* tasks it creates for each method invocation.

Other implementations of Executors can be found in some concurrency libraries. For example, Doug Lea's util.concurrent package for Java provides a framework of classes for using thread effectively in Java. As a part of that package, several kinds of Executor classes are included. In fact, the ConcurrentExecutor presented here is very similar to the QueuedExecutor found in this package.
http://gee.cs.oswego.edu/dl/classes/EDU/oswego/cs/dl/util/concurrent/intro.html

Another library containing this abstraction is ZThreads, a portable, object-oriented thread library for C++. In it, a set of Executor templates that are designed to assist in running tasks in different threads.
http://zthread.sourceforge.net

Acknowledgements:

I would like to thank Kevlin Henney who was the shepherd for this paper. His guidance was invaluable in transforming my ideas into the paper you are now reading. Thanks are also extended to Frank Buschmann who oversaw the shepherding process and to all the others who provided me with feedback early on.

References:

[KJ02] Michael Kircher, Prashant Jain, *Pooling*, http://www.hillside.net/patterns/EuroPLoP/papers/Kircher_Jain.zip, 2002.

[GoF95] GoF, Design Patterns, *Command Pattern*, p 233 – 242, Addison-Wesely, 1995.

[Sun00] Sun Microsystems, *Using a Swing Worker Thread*, http://java.sun.com/products/jfc/tsc/articles/threads/threads2.html, 2000.

[PLoPD96] Eds Vlissides, Coplien, Kerth, *Command Processor*, in Pattern Languages of Program Design 1, Addison-Wesley, p63-74, 1996.

[POSA99] Schmidt, Stal, Rohnert, Buschmann, *Active Object*, in Pattern Languages of Program Volume 2, Addison-Wesley, p369-398, 1999.

Automated Determination of Patterns for Usability Evaluations

Michael Gellner

University of Rostock, Department of Computer Science, Software Engineering Group, Albert-Einstein-Str. 21, Rostock 18051, Germany, Phone ++49 (381) 498-34 33

mgellner@informatik.uni-rostock.de

Abstract. Although usability evaluations deal with improving usability of the inspected artifacts the process that leads to these results often misses usable attributes. As a result working in that field is a domain of usability experts only. Since in a lot of environments so called computer-aided solutions are successful in either lowering the efforts for the employees or in filling their gaps by being supported from such a system, this work is one step toward computer aided usability evaluating *(CAUE)*. In a view with eight components beginning with the declaration of evaluation tasks up to reporting the entire evaluation process the following work shows how the second step could be supported by a CAUE system. In that phase the goal is how to determine optimal usability method of many dozens without the necessity to study them before deciding.

1 Introduction

The following work describes a concept that can be used in a computer aided usability evaluation (CAUE) tool. It is based on the eight phase model [9], a process pattern that describes the steps, that has to be done by conducting a usability evaluation and that integrates the common aspects of the published models [6], [8], [11], [12], [16], [20] and [22].

Background Information: What are *process patterns*?
The term *process patterns* comes from Coplien, who defined in [7] that processes often can be considered as patterns of activity within an organization or a project. To differentiate such patterns from the design patterns published in 1995, he created this term. Since then the term *process patterns* is widely used, especially by Ambler in his two books about object oriented software development *Process Patterns* [2] and *More Process Patterns* [3] from 1998. The eight phase pattern that paper deals with is an derivation of this view applied on usability evaluation.

In contradiction to these models the intention of the eight phase model is to create a foundation for a software tool. This model is created for formative purpose, whereas the cited models primary describe usability evaluations as they are. Structurally the eight phase model can be considered as an usability pattern [4], [23] and as a process pattern [2], [3], since it has properties of both ones. Each phase is composed of other patterns. The following approach demonstrates a way to determine methods from a pool of

testing methods. This especially supports the needs of the second phase of the eight phase pattern (see Fig. 1) in which testing methods have to be chosen. At present the literature counts around 60 methods (the exact number depends on the way of counting them; psychologists count every questionnaire as separate evaluation method whereas less well methodically working disciplines often treat the usage of any questionnaire as a general application of the method 'questionnaires').

Background Information: *evaluation methods* and *method patterns* – what's the difference?
The whole literature is about *evaluation methods*. Up to now you find more than 60 methods for usability evaluations. We can assume that even an usability expert won't know them all. They are distributed in various papers, books and reports. Although this is the typical way of publishing in the academic community it is absolutely unhandy to the practitioner. It is possible that even someone who works for a long time in that field has never heard of a testing method that could optimize his work. No one can keep track of this source jungle, maybe with the exception of someone who does nothing else. If so: Usually you do not look for methods but for solutions of your problem (with the testing situation as the problem). For that reason we want represent the testing methods in another way: the situation an evaluation will take place should be given (that can be early in the development cycle) and the solution – that is one or more methods that fit to the situation – should be delivered. The attribute of being solution oriented is fulfilled by patterns. That is the reason why we want to describe and use testing methods as patterns. The content doesn't change: the testing method *heuristic evaluation* e.g. has the equivalent meaning with the method pattern *heuristic evaluation*. Rewriting the evaluation methods and collecting these method patterns in a usability pattern catalogue would ease learning how to conduct usability evaluations and how to apply different methods.

Three preconditions have to be fulfilled in order to realize the concept:

1. The *forces* and the *context* that influence an usability evaluation have to be described sufficiently. Both are summaries of the environments and circumstances but have different main focuses:
   - The *context* determines which kind of pattern can be possibly applied.
   - *Forces* help to decide which one of them fits in optimally.

   For the following procedure attributes of forces and context are summarized *»requirements«*. These requirements include attributes such as funds for equipment of the evaluating department or the progress of a project, the targets of an evaluation and further more are specified (for details see upside).
2. The *method patterns* have to be described sufficiently: specify what equipment is needed to use the method, what is the best moment for a method to be executed, and what priorities have to be set up.
3. The descriptions for the requirements and the method patterns have to be in the same format. Any given data has to match the used criteria.

For the presentation of the requirements and the method patterns a fixed but extensible format is used.

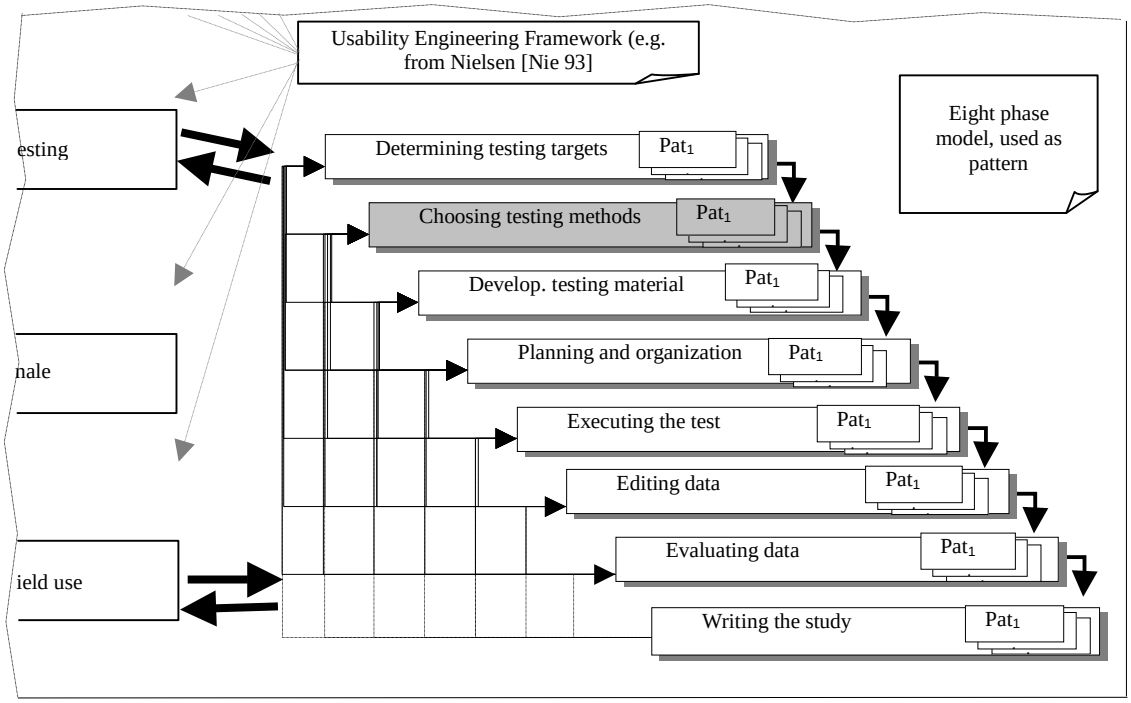


Fig. 1. The eight phase pattern is a process pattern [2] that can contain further patterns in its phases.

Background Information: What is the eight phase pattern and why did we create it?

Conducting an usability evaluation is nothing else than a project. To support this we want to create a catalogue with all necessary patterns (or maybe a *pattern language*) to conduct usability evaluations. We describe the process in pattern format. The evaluation process consists of eight phases, that are structured in a waterfall view. One clear advantage is that software developers should be able to adapt this easily since there also exists a waterfall model that shares a lot with ours. One important difference is that the eight phase model offers ways back from every phase to the beginning. The lack of such ways back in the waterfall model for software development caused a lot of problems in the past. That is one reason why the waterfall model for development today is often considered more as an antipattern than as a wishful foundation for work. Our approach avoids this.

We created the eight phase pattern although there where several models that also tried to capture the evaluation process because none of them seemed to fit to most of the typical situations in that evaluations are done. This is discussed more in detail in [10]. Another important aspect is the integration of the eight phase pattern in the known and applied development models: when do we do an evaluation? Fig. 1 shows parts of the usability engineering model – with this we want to show the integration of our pattern in this model. For connections to other development please have a view in [9].

2 Representing Knowledge about Methods

A lot of approaches describe how differences between usability testing methods and their common elements can be cataloged, but capture only descriptive aspects, e.g. see [13], [14] and [21]. None of these sources offers an algorithm that finds optimal methods for actual problems or that describes which properties are necessary to characterize a testing situation. Since such a procedure is needed for a CAUE system a more detailed view is necessary. The result should match the following specification:

- The procedure should find a testing method for a given situation that could be considered as a good (or even optimal) decision, comparable to the advise of an human expert
- The output should be given in form of a list that contains all analyzed methods. The optimal fitting ones should be listed first, the fewer they fit the later they are listed.
- The pool of testing methods has to be modifiable to add methods or to remove them
- The methods have to be valued multi-dimensional (that means with more criteria)

Important criteria are

- time
- costs
- personnel
- hardware skill
- software skill
- testing version
- comparability
- efficiency
- importance of the production stage
- necessity of following standard
- enjoyment of use
- error frequency

These have to be considered as suggestions for the presentation of the procedure and will be templates for the construction of the method patterns. These criteria will be the basis for semi-automated findings of the matching methods. During the procedure the list of criteria has to be identical with the analyzed methods and the given situation, otherwise a comparison is not possible. In general the number of the items and the content of single items can be changed. For example after a series of tests is made, the criterion *experience with a method* could be considered important, since learning a new and perhaps more efficient method can cost more than one that can be applied without learning or without loss by novice errors. Furthermore a software realization should describe how a recommended method is applied and what results are expected; these parts of the template are not described in that work and are not considered in the procedure.

Another reason for the changeability of the criteria lists is that not every motivation for usability tests can be foreseen. Moreover it would not be useful to maintain a lot of irrelevant data. Maintaining higher specialized criteria are only necessary when it is required to evaluate software that wants to fulfill such them. Examples are the

functionality for special qualifications, the possibility to recognize displays about a wider distance or the support for handicapped persons.

To work with the procedure data are given that represent experience values. All data come in form of a semantic differential with the interval $-n$ to n by 1. To show the procedure we consider arbitrarily $n = 3$ (depending to the required granularity other values could be used). The values are obtained by a mapping of the values to that interval. A priority in further work is to find representative data for all methods and to keep them current. A feature of scaling is a balanced view among all criteria. Otherwise a problem of different valuations comes up: The maximum value e.g. of the criterion *time* for a testing method can hold for hours in one department that is used to test short periods with small systems only whereas another lab measures time in days. A dozen hours is a high number that would be represented by the value $a = 3$ in dimensions the first lab works. In the second one a dozen days is only below average and so might be set on the value $b = 0$. Although the value a is higher than b this comparison would be invalid since there are different dimensions represented. Assuming the dimensions of the first lab the value that represents b could not be set adequately: 12 hours got the maximum value – how to describe 12 days? Assuming the dimensions of the second lab, 12 hours might be set on the value -1 or -2.

A neutral element outside the scale ("*don't know*", "*no statement*", "*can't judge*" etc.) as often given in questionnaires to avoid distortions the results, is not used here but can be introduced if it becomes obvious that non-experts are involved that would need a way to abstain from valuating. The problem that should be avoided in tests with testing persons is that people who feel overburdened or unsure answer arbitrary. With such a neutral channel they can be sorted out and these answers do not mislead the quality of the whole study. In this case, adding and removing testing methods and criteria will be done by an usability expert, a qualified administrator or with an update automatism of an CAUE system, when new information about methods or criteria become available.

Other scales are possible and their support in a software system is desirable. A wide range of absolute values that have to be captured can lead to the need for wider scales, especially when such experiences have to be combined for summaries in one diagram or table. But for this description that aspect is without means, we do not engross this thoughts.

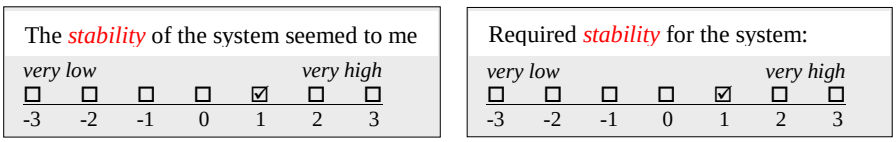


Fig. 2. Application of a semantic differential for the criterion stability (left side) and for requirements analyses.

Semantic differentials, also called polarity profiles, use two contradicting expressions to get an estimation of an asked person (see Fig. 2). With such an amount of estimations of attitudes and mindsets of a certain group to a theme can be measured, such investigations are subject of different psychological disciplines, the sociology, the opinion research, literary studies and many others. Usability evaluations use this technology e.g. when questionnaires are developed. Examples are the often used

ISONORM questionnaire and the System Usability Scale (SUS). Every criterion consists of three components:

- a name for the criterion
- a string for the presentation and
- the chosen value

Similar to that the description of the requirements follows (see Fig. 2):

- a name for the criterion
- a string for the presentation and
- a value of the need

The suggested procedure consists of the comparison of the criteria for the requirements with the criteria of all method patterns. This comparison leads to a value, that is taken as an index for the relation of the method pattern to the requirements. As mentioned the criteria are subject of further works. For the user the criteria and the valuations of the methods have to be considered as given. His work is to value the requirements criteria by criteria for each usability test. This should be done dialog based, every presentation string has the form of a question and must be answered by choosing a value from the given scale.

Table 1. Method patterns in table form

Criterion / *Testing method*	*time needs*	*cost needs*	*personal needs*	*hardware needs*	*software needs*	*efficiency needs*	*comparability needs*	*stage consideration needs*	*standard consideration*	*enjoyment of use needs*	*error frequency needs*	*earlier test consideration*
Example method 1	0	0	-1	-3	-2	-2	0	3	-3	1	-3	3
Example method 2	3	0	3	-3	-3	-1	-1	3	3	-3	-3	0
…												

The list of values for one method characterizes the method, this is called *method pattern*. The other list of values, that is compared with all method patterns are the requirements. Requirements and method patterns have to be in the same format. A certain list of criteria is a *template*. As said this can alter if necessary. Table 1 shows the representation of different method patterns in a table. Beneath a table method patterns can be represented in a graphical form (see Fig. 3). From a mathematical view this graph should contain discrete points (one value for a criterion) but drawing lines eases the usage since those graphs should be read like fingerprints from methods (if someone cannot go with this suggestion also discrete points can be left). A single method could be judged easier with a graphic representation, to compare different methods the tabular form seems to offer a more transparent view.

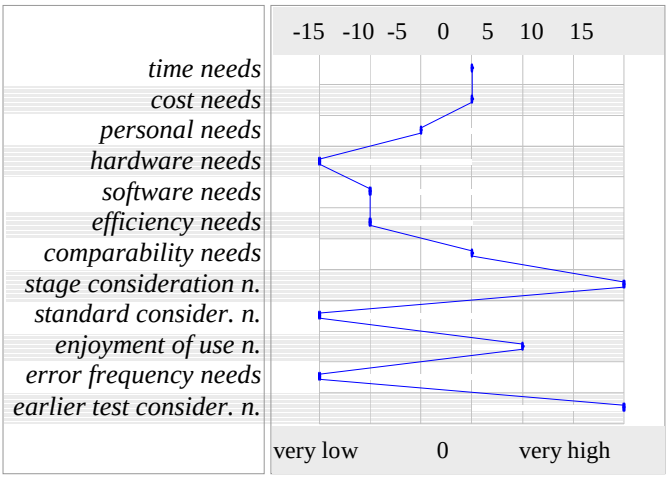


Fig. 3. Representation of the profile of a testing method

3 The Procedure in Detail

3.1 Preconditions

For the following description it is a precondition to have a database with valued methods. The database is part of the further work and will be considered here as given, also a list of criteria is considered as given. Formal the database consists of a list of values for each method as shown in Table 1, where every value stands for the assumed ability of the method to fulfill the assigned criterion.

3.1 Computation of the Differences between Requirements and Method

1. *Analyzing the requirements:* All criteria of a template are presented to an user (here: the one who conducts the usability test), who gives every criterion a value on a scale that describes his available resources, abilities and needs. To simplify this, every criterion requires a string that formulates the criterion as a question. Dialog box by dialog box those questions are asked.
2. *Weighting criteria:* Not all criteria have always the same meaning. The weight they have can change from test to test. If necessary, some criteria have to be stressed, in other cases they have to be belittled. To reach this every criterion can get a factor. This causes that both the value of the method pattern and the value of the requirements criterion are multiplied with the factor before the comparison. Since a difference between the value of the requirements criterion and the value of the method patterns criterion is computed, the amount of the difference increases.
3. *Setting the Always Count Flag: Every* criterion with this flag set, 'positive' differences are ignored. Positive differences are ones that are not caused by failing the requirement value but by over fulfilling it. Both, failing a requirement and over

fulfilling, leads to a difference. The first one is the difference that shall be used for summing up the differences. The higher this sum is the more a method fails the requirements. Normally it seems not to be adequate to treat saving resources in the same way as wasting them. So without setting this flag such positive differences are set to 0. Theoretically (and computationally) it would be possible to compensate a failing with an over fulfilling, practically this would be a misspecification: if there is a lack of personal it is not possible to compensate this with a lot of hardware. If it is needed to test the criterion of enjoyment of use a method have to fulfill this requirement. Another method that is not suitable for this but considers e.g. more exactly the observance of usability standards would simply be a wrong advice. For that reason these two kinds of differences have to be treated separately. In order to find a fitting method pattern for given requirements, in general the negative differences count more than positive ones, since a method cannot be used if special resources are not available, absolutely independent from the plenty of other resources. Applying this flag seems to be useful in two cases:

a) Two methods have the same value: Now it can be analyzed which one of the methods that offer the same efforts can do this in a more economic way.

b) The user wants to force that available resources are used.

4. *Comparing the requirements with the available methods:* Requirements that are analyzed by following topic 1 has to be compared with all method patterns.

   c) The products between the values of the criteria of the method patterns and the weights are computed and the products between the criteria of the requirements and the weights. If a weight is not explicitly given, its value is 1.

   d) For each criterion a difference between the value of the method criterion and the weighted value of the criterion of the requirement value is computed.

   e) If a difference results only during over fulfilling the requirements, that are set up by the requirements, it has to be set to 0 (except the always count flag is set, then the computed amount has to be taken). This step is called the correction.

5. *Summing up:* Now for each criterion there is a number that describes the difference between the requirements and the method for a certain criterion. These numbers (the corrected differences) are summed up. The resulting sum is the called the *Requirements-Pattern Difference (RPD).*

6. *Sorting RPD:* The RPD shows the degree, in that a method differs from the requirements. The smaller the RPD the better a found method patterns fits to the requirements, an optimal RPD has the value 0. The higher a RPD the bigger is the difference between requirements and analyzed method. Sorting the methods by the values of the RPD leads to a list that shows the best fitting method on its top and the most inadequate method for the given requirements at the bottom.

7. In case there is more than one minimal RPD:

   - In a second step the unconsidered positive differences can be computed to find out which one of the methods works more economically, that means the method that uses less resources. Such a second step automation is not hard to realize.
   - Furthermore the system can provide the materials about the concerned methods for the user, so that the user has to choose the methods which are wanted. Identical

RPD must not necessarily result from an identical constellation between method and requirements, so it is not improbable that in spite of the identical score one of those methods could be preferred. At this point the efforts for an automation is hardly to justify, unless this is impossible.

Background Information: What has that to do with patterns?
There are two things to stress:
I. Of the contents the shown procedure helps to value the attributes of the method patterns (in former works: evaluation methods).
II: For the purpose of representation this procedure helps to create visual signatures that can be read fast and easily.
At the first point: we take testing methods and rewrite them in a certain pattern format. Consider the procedure as an operator that supports this translation.
More didactically is the second point: Alexander said, if you cannot draw a picture to it, it is not a pattern. Shortly said: He put some weight on a graphical representation. We share this opinion strongly and so we add to every testing method such a visual »fingerprint«. The procedure might look complicated but simple results often require elaborate background works (e.g. making WinWord was far more complicated than using WinWord). The shown procedure is **not** thought to be applied by usability evaluators. It shows how developers could create a supporting system **for** usability evaluators.

3.2 Constructing an example for the procedure

For the example hypothetical requirements are thought up. We consider a situation in which a software developer has to care for good usability attributes in his product. So first the user of the procedure is a software developer. The situation is characterized from a high pressure of time (the method can only require minimal time, the chosen value is -3), although a maximal efficiency is needed (valued with 3) and a minimal error frequency should be achieved (valued with three), for the complete valuation of the requirements see Table 2. All others resources can be realized. The only importance is to improve the usability of the advanced production, so far that the hypothetical product becomes usable at all.

Following the department of defense (DoD) of the United States this scenario is representative for around 90 percent of their software producers and suppliers. In the stages of the Capability Maturity Model (CMM) about 99 % of the companies are placed in the first two stages (from five), that are marked by problems as in the hypothetical situation. The CMM is a model that allows to judge the *maturity* of a software development process in a company, this stand for the ability of a company to fulfill needs in time, quality and costs reliably and repeatable. The CMM is developed and used in the USA and often compared with the ISO 9000 standard that is favored in the European countries.

In every method analysis all method patterns are compared to the requirements. From all computed RPDs the minimal ones are shown first respectively on top of the list. It can be supposed that not all dozens of further results are interesting, except of the first

five or ten results. For a better transparence the example is limited to two methods that are chosen arbitrarily: the GOMS method introduced in [5] and the heuristic evaluation due to [17], [18]. Table 2 shows the requirements and the method patterns for GOMS and the heuristic evaluation.

Table 2. Hypothetical requirements and the both method patterns to compare with

Criterion / *Testing method*	*time needs*	*cost needs*	*personal needs*	*hardware needs*	*software needs*	*efficiency needs*	*comparability needs*	*stage consideration needs*	*standard consideration*	*enjoyment of use needs*	*error frequency needs*	*earlier test consideration*
Requirements	-3	3	3	1	1	3	3	3	-2	1	3	-3
GOMS pattern	3	0	3	-3	2	-3	-3	3	-3	-3	-3	0
Heuristic evaluation	-3	-2	-2	-3	-3	3	0	2	-2	1	2	0

Since the first point, analyzing and putting in the requirements, is done above, the example starts with weighting the criteria. Important criteria should get higher weights, so that even small differences in that points lead to obviously high RPDs and become to KO criteria). Since time, efficiency and error frequency have the highest meaning, they are weighted with a factor of 5. It can be assumed that similar situations will lead to similar lists of weighting factors. So these lists can be considered as *weighing patterns*.

Weight (factor)	5	1	1	1	1	5	3	1	3	1	5	0

Next *always count flags* can be set, to handle ignored resources the same way as wasting them. This flag should be set if resources are wanted to be used up. In this example one always count flag is set only for demonstrating.

Always Count Flag	-	-	-	-	-	-	-	-	✓	-	-	-

After setting the weights the following values for the requirements and the method patters result:

Requirements	-15	3	3	1	1	15	9	3	-6	1	15	0
GOMS pattern	15	0	3	-3	2	-15	-9	3	-9	-3	-15	0
Heuristic evaluation	-15	-2	-2	-3	-3	15	0	2	-6	1	10	0

The differences between the requirements and the method patterns are calculated criterion by criterion. First this is done for the GOMS method:

Requirements	-15	3	3	1	1	15	9	3	-6	1	15	0
GOMS pattern	15	0	3	-3	2	-15	-9	3	-9	-3	-15	0
|Differences|	30	3	0	4	1	30	18	0	3	4	30	0

And second for the heuristic evaluation:

Criterion / *Testing method*	*time needs*	*cost needs*	*personal needs*	*hardware needs*	*software needs*	*efficiency needs*	*comparability needs*	*stage consideration needs*	*standard consideration*	*enjoyment of use needs*	*error frequency needs*	*earlier test consideration*
Requirements	-15	3	3	1	1	15	9	3	-6	1	15	0
Heuristic evaluation	-15	-2	-2	-3	-3	15	0	2	-6	1	10	0
|Differences|	0	5	5	4	4	0	9	1	0	0	5	0

In both cases the always ignore flag is without influence, since for GOMS as well as for the heuristic evaluation the required value is failed. Further these differences are provisional results, now it must be distinguished between ‘positive’ (to strike) and ‘negative’ (to use for computation) differences:

Requirements	-15	3	3	1	1	15	9	3	-6	1	15	0
GOMS pattern	15	0	3	-3	2	-15	-9	3	-9	-3	-15	0
|Differences|	30	~~3~~	0	4	1	30	18	0	3	4	30	0
	$RPD_{[GOMS,\ Requirements]} = \sum 120$											

Requirements	-15	3	3	1	1	15	9	3	-6	1	15	0
Heuristic evaluation pat.	-15	-2	-2	-3	-3	15	0	2	-6	1	10	0
|Differences|	0	~~5~~	~~5~~	~~4~~	~~4~~	0	9	1	0	0	5	0
	$RPD_{[heur.\ Eval.,\ Requirements]} = \sum 15$											

The remaining differences are summed up to the RPD. Since the both indices are not identical the procedure has reached the end. The RPD of 15 makes the heuristic evaluation seem to be more suitable than GOMS with the RPD of 120.

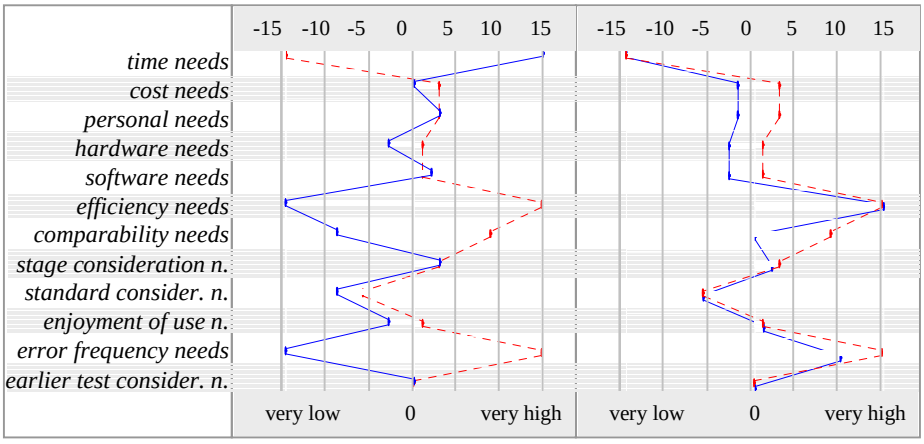


Fig. 4. Graphs of a comparison between the requirements and a method pattern (left: GOMS, right: heuristic evaluation, methods are shown in blue unbroken lines, requirements are in both graphs shown with a broken red line.

This coincides with a very high probability with the advice that usability experts would give in the same situation under the condition that only the same two possibilities are available. Alternatively this computation graphs of the method patterns and the requirements can be compared. This graphs is less exact then the shown computation, but it can be done without technical equipment and it gives at least a fast overview.

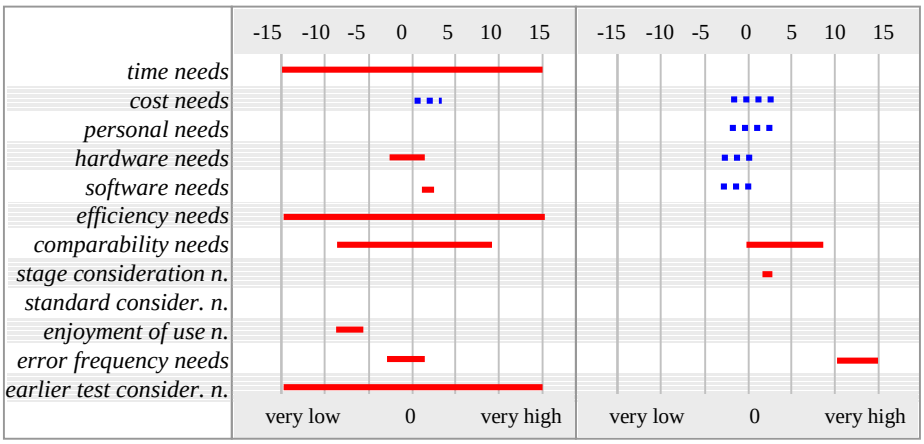


Fig. 5. Visualization of the differences between method patterns and requirements (left: GOMS, right: heuristic evaluation, red unbroken lines symbolizes failing the requirements, blue lines dotted show over fulfilling a criterion).

The enormous differences between the GOMS pattern and the requirements (more than a factor of 8 in comparison to the differences between the heuristic evaluation pattern and the requirements) are easy to recognize in Fig. 4. As clear as here this cannot always be seen. Another graphical view is possible by putting the differences in a diagram (see Fig. 5).

Again, this view doesn't show the exact values of the contradiction between the requirements and the analyzed method, but it displays the tendency.

Background Information: If this is done how will those method patterns look?

At the moment this is not absolutely sure, but possibly something like this:

Heuristic Evaluation Pattern (similar: Next Door Testing, Discount Testing)

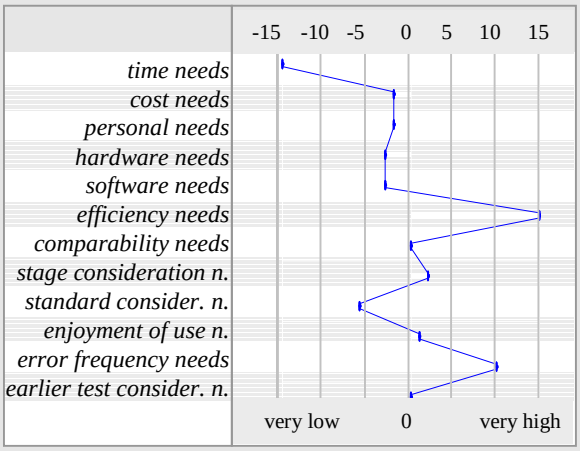
Author: Jakob Nielsen (Heuristic Evaluation Method), Michael Gellner (Pattern)

Problem: You have to conduct an usability evaluation but only small funds for hard- and software. The evaluation comes very late since there is already done a lot of work. Although this is not a wishful situation you have to take it as it is and to take care for the best usability attributes that can be realized at this point.

Examples: - A piece of software should be tested separately for its usability attributes.
- An usability evaluation has to be conducted spontaneously.

Context: There is not much time, not much funds for buying equipment and there are hardly human resources except of some other developers.

Solution: Use the heuristic evaluation method. You need no real testing persons and no special equipment (hard- and software). A handful of developers are enough. Act as follows:
Here will follow a "how to"-description of the heuristic evaluation. We skipped this since we want to show how our patterns could look like and do not want to focus on one usability evaluation method.
The common signature of the heuristic evaluation pattern in a nutshell:



That *signature* is the part of the pattern what the procedure is for.

4 Conclusion

The shown procedure offers a way to compare different usability evaluation methods. The idea of patterns as tools and as alternative to guidelines (discussed by van Welie et al. [25]) is applied in this concept. This is an important step to guide and advice non-expert users in their workflow by evaluations. In internal workshops further areas of usage could be named. They are given below.

5 View

A further usage beneath the finding and application of methods could be to support documentation and communication. These abilities are already published for the design patterns in an early work from Johnson [15]. A transfer of his results to the presented usability evaluation patterns would be helpful.

One more important point is the described database. The literature gives a lot of reference numbers which have to be extracted but not all data is given there. For many important criteria such numbers are not presented until now, that means a lot of testing activity is necessary to get them. A cooperative network with usability oriented departments that organize testing series with such special methods could find out a lot of this missing data rapidly.

As a consequence the comparability of usability departments has to be improved. An article from Nielsen shows the necessity: Nielsen tried 1994 to compare different

usability departments. As criteria he uses the number of used rooms and the possibility to use a scan converter [19]. It is true that such points are easy to answer exactly, otherwise it seems really questionable if those formalities are valid criteria to compare usability departments. Wiklund avoided such criteria and edited a book in that different usability departments are shown completely in their own words [24]. This really gave a good imagination about how the labs in really known companies work, but on the other hand he avoided completely any comparison. Obviously both authors stood for the problem of finding satisfying criteria.

The given idea of method patterns can be used to mark the abilities of a usability lab. A method pattern gives an index about the attributes of a testing method by correct usage in a correctly described situation. But be aware: by no means these indices are guarantied specifics or absolute values. Bigger differences from the given indices indicate either an index that has to be corrected or a weakness in the applying environment. In partnerships of usability labs such indices could be compared to find possible lacks of efficiency.

By comparing with others the measurement of the own environment becomes possible: how far are the new hard- and software or the new approach really useful? The pattern approach in combination with the testing methods make quality management in the area of usability testing possible.

References

1. Alexander, C., Ishikawa, S., Silverstein, M., Jacobson, M., Fiksdahl-King, I. and Angel, S., *A Pattern Languag*e. Oxford University Press, New York, 1977.
2. Ambler, S. W., *Process Patterns: Building Large-Scale Systems Using Object Technology*. SIGS Books/Cambridge University Press, New York, 1998.
3. Ambler, S. W., *More Process Patterns: Delivering Large-Scale Systems Using Object Technology*. SIGS Books/Cambridge University Press, New York, 1998.
4. Borchers, J., *A Pattern Approach to Interaction Design*. John Wiley & Sons, Chichester, England, 2001.
5. Card, S. K., Moran, T. P. and Newell, A., *The Psychology of Human Computer Interaction*. Hillsdale, New Jersey, Lawrence Erlbaum Associates, 1983.
6. Constantine, L. L. and Lockwood L. A. D, Software for use, a practical guide to the models and methods of usage-centered design. Addison Wesley Longman, Inc., Reading, Massachusetts, 1999.
7. Coplien, J., *A Generative Development-Process Pattern Language*. In: Coplien, J.O. und Schmidt, D. [Hrsg.], Pattern Languages of Program Design, Addison-Wesley, Reading, Massachusetts, pp. 183 - 237.
8. Dumas, J. S. and Redish, J. C., *A Practical Guide to Usability Testing.* Revised Edition. Intellect Books Limited, Exeter, England, 1999.
9. Gellner, M., *Modellierung des Usability Testing Prozesses im Hinblick auf den Entwurf eines Computer Aided Usability Engineering (CAUE) Systems*. In: Rostocker Informatik-Berichte, Vol. 24, pp. 5 – 21. Rostock, 2000.
10. Gellner, M. and Forbrig, P., *Modeling the Usability Evaluation Process with the Perspective of Developing a Computer Aided Usability Evaluation (CAUE) System*.

Proceedings of INTERACT 2001, Workshop on “Usability throughout the entire systems development lifecycle”, 2001, Tokyo, Japan.

11. Good, M., Spine, T. M., Whiteside, J. and George, P., *User-derived impact analysis as a tool for usability engineering*. In: Mantei, M. und Oberton [Hrsg.], Human Factors in Computing Systems, CHI’86, Conference Proceedings, pp. 241 - 246, ACM Press, New York, 1986.
12. Hix D. and Hartson H.R., *Developing User Interfaces: Ensuring Usability Throught Product and Process.* John Wiley & Sons, New York, 1993.
13. Hom, J., *The Usability Methods Toolbox*. URL: http://www.best.com/~jthom/ usability/usable.htm, San Jose State University, Department of Industrial and Systems Engineering, 1998.
14. Hüttner, J., Wandke, H. and Rätz, A., *Benutzerfreundliche Software, Psychologisches Wissen für die ergonomische Schnittstellengestaltung*. Bernd-Michael Paschke Verlag, Berlin 1995.
15. Johnson, R. E., Documenting Frameworks using patterns. In: Object-Oriented Programming Systems, Languages, and Applications Conference Proceedings, publiziert als ACM SIGPLAN Notices, volume 27, number 10, Vancouver, British Columbia, Canada, ACM Press, 1992, pp. 63 - 76.
16. Mayhew, D. J., *The Usability Engineering Lifecycle*. Morgan Kaufmann Publishers, Inc., Kalifornien, San Francisco, 1999.
17. Nielsen, J., und Molich, R. (1990). *Heuristic evaluation of user interfaces*. Proc. ACM CHI'90 Conf. (Seattle, WA, 1-5 April), pp. 249 - 256.
18. Nielsen, J., *Usability Engineering*. AP Proffessional, New Jersey 1993.
19. Nielsen, J., Usability Laboratories: A 1994 Survey. In: Behaviour & Information Technology 13, 1&2, S. 3 – 8.
20. Preece, J., *Human-Computer-Interaction*, Addison-Wesley, Harlow, 1994.
21. Rubin, Jeffrey, *Handbook of Usability Testing: How to Plan, Design, and Conduct Effective Tests*, John Wiley & Sons, Inc., 1994.
22. Stary, C., and Riesenecker-Caba, T., *EU-CON II – Software-ergonomische Bewertung und Gestaltung von Bildschirmarbeit.* Schriftenreihe der Bundesanstalt für Arbeitsschutz und Arbeitsmedizin [Hrsg.], Forschung FB 826., Wirtschaftsverlag NW, Dortmund/Berlin, 1999.
23. Tidwell, J., *A Pattern Language for Human-Computer Interface Design*. Washington University Tech. Report WUCS-98-25, 1998, (Basierend auf der Präsentation *Interaction Design Patterns*, Conference on Pattern Languages of Programming IV. (PloP’98), Monticello, Italien.
24. Wiklund, M. E. [Hrsg.], Usability in Practice, How Companies Develop User-Friendly Products, Academic Press, Inc., London, 1994.
25. van Welie, M., van der Veer, G. and Eliëns, A., Patterns as Tools for User Interface Design. In: International Workshop on Tools for Working with Guidelines, pp. 313-324, 7- 8 October 2000, Biarritz, France.

Transformational Pattern for High-Level-Architectural Connectors

Lars Grunske
Department of Software Engineering
Hasso-Plattner-Institute for Software Systems Engineering at the University of Potsdam
Prof.-Dr.-Helmert Strasse 2-3
D-14482 Potsdam (Germany)
+49(0)3315509152
grunske@hpi.uni-potsdam.de

ABSTRACT
Today's software systems are often built from a set of independent components. For interconnecting these components the interaction mechanisms are capsulated in connectors. This leads to an architectural description, which uses both components and connectors as first class modelling entities. Existing research provides a large fundament for the component construction. For the ability to build connectors, some work is still needed. Therefore, this paper presents a set of patterns for high-level architectural connectors, such as:

- secure transmission connector
- error detection or correction connector
- compressed transmission connector
- split bi-directional transmission connector
- redundant channel connector
- adapter connector.

These patterns restructure existing software architectures, so they meet their non-functional requirements, such as maintainability, security and safety.

Keywords
software architecture, components, connectors, pattern language, high-level architectural connectors, construction of concrete connectors

1. INTRODUCTION
In the past decades, the development of software architectures has received increasing attention by researchers and practitioners [Hofmeister et al. 99], [Shaw, Garlan 96]. Thus specifying the software architecture addresses problems in the engineering of large and complex systems. This results in a description of the "big picture" which is associated with software architecture. Using software architecture, developers can clarify their understanding of the system and communicate with each other [Medvidovic, Rosenblum 99]. Furthermore the construction process of the system can be eased and the code generation can be done automatically or semi automatically with well-defined architectural models [OMG 01]. The effect is a decreased time to marked and a reduction of the development costs. Thus, there is high interest to apply this in industrial software engineering projects.

Furthermore, with the designing of software architecture, important aspects are modeled early in the development process. Thus problems can be identified and removed cost and time efficiently. To remove the identified problems the architectural model can be restructured. For this, transformational patterns can be applied, which are similar to code level refactorings as proposed by [Folwer 99]. A transformational pattern is a refactoring at an architectural level, which only change parts of the architectural model without alternation of the provided external behavior of the system. So transformational patterns can be viewed as recipes for improving the quality of the software architecture. This paper presents the concepts of transformational patterns. To illustrate this concept the paper introduces some transformational patterns, which especially address high-level architectural connectors.

These high-level architectural connectors are introduced in Section 2 where it is described how to specify and implement them. An introducing example is given after this, which shows the practical relevance of high-level architectural connectors. Furthermore, it gives some examples of how an architectural model can be restructured and it motivates the usage of transformational connector patterns. The concepts of these transformational connector patterns are described in section 4. Section 5 illustrates this concept, by presenting commonly used patterns for restructuring high-level architectural connectors. Finally, section 6 concludes the paper and makes remarks on further work in this area. In the appendix of this paper the used architectural notation is presented.

2. CONNECTORS

2.1 What Are Connectors and Why Are They Useful?

Today's software systems are often constructed with a set of components [Scipersky 98][Balek, Plasil 00]. Thus, to specify the software architecture of a system, the fundamental components should be described by a well-defined interface and a concrete behavior specification. This is the purpose of many published works [Allen, Garland 97, Scipersky 98].

Based on research in component technology, it is pointed out that the communication between the components should be encapsulated by other architectural elements [Balek Plasil 00, Allen, Galand 97, Shaw, Garlan 96]. These elements are called connectors. With the introduction of these connectors in architectural specification a separation of computation (components) from control and communication (connectors) is possible [Hofmeister et al. 99]. The benefit of this separation is the reusability of the components, because of the looser coupling between the components and the possibility to specify adaptation mechanisms in a connector [Garlan et al. 95, Medvidovic et al. 97]. In addition to this, the introduction of connectors has benefits for the maintenance, flexibility and scalability. Furthermore, in distributed systems location transparency and mobility of the components can be achieved with connectors. Finally, the usage of connectors solves the deployment anomaly described in [Balek 02]. This addresses the problem of hard coded communication mechanisms in the components.

To summarize this, the software architecture needs two first class architectural elements, the components and the connectors. [Perry,Wolf 92]. The components capsule the functionality of the system and the connectors capsule the control and communication aspects of the system [Hofmeister et al. 99].

2.2 Specification of Connectors

For the specification of high-level architectural connectors, two kinds of connectors exist, basic and composite. Basic connectors provide common communication and/or control primitives. The communication primitives are for example remote procedure calls, asynchronous message passing, synchronous message passing, rendezvous etc. [Mehta et al. 00] [Eugster et al. 01]. The control primitives define which components get which information [Mehta et al. 00]. This is relevant if more than two components are connected with one connector, as presented in figure 2-1. In this example the information generated by component *A* can be forwarded to component *B* or to component *C* or to both. For the specification of these communication and control primitives an architectural description language (ADL) or a glue description language [Alan, Garlan 97] can be used.

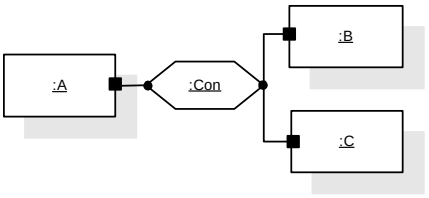


Figure 2-1 Example topology (one connector and tree components)

Composite connectors are built of nested components and connectors. A topology specification must be used to describe the formation and the interconnection of these nested components and connectors. This topology specification describes the internal structure of a connector with internal elements and their interactions.

For the topology specification the notation of [Hofmeister et al. 99] is utilized in this paper. The relevant parts of this notation and the underlying model are presented in the appendix.

2.3 Implementation and Realization of Connectors

To apply the concepts of high-level architectural connectors in the construction of systems, it is necessary to point out how to realize them at the code level. For the realization the following two possibilities exist:

- Connectors are implemented by the code level modules of the associated components or,
- Connectors are implemented in their own code level modules

The first variant is used for basic connectors, which only implement simple communication primitives. For example, if two components communicate with procedure calls, the sending component uses a reference, which identifies the receiving component.

The second variant is used for connectors that implement control primitives. As an example for this, the connector in Figure 2-1 can be implemented with a code level module, which follows the mediator pattern [Gamma et al. 96]. In this case the connector is realized with a mediator class and the components are colleagues of this mediator class. This works well for static relationships between connector and components. But if the number of associated components is changing dynamically, the solution with the mediator pattern is not suitable. In this case the connector can be implemented with the observer pattern [Gamma et al. 96]. According to this, the

connector contains a subject class, which can dynamically attach the new components.

To realize composite connectors the second variant should always be used. In any case these connectors contain nested components, which are implemented in own code level modules. These code level modules can be used for the implementation of the control and communication primitives of the nested components.

3. MOTIVATING EXAMPLE: A TELEPHONE BUSINESS SYSTEM

An architectural example is utilized to clarify the understanding of a connector and to motivate the usefulness of transformational connector patterns.

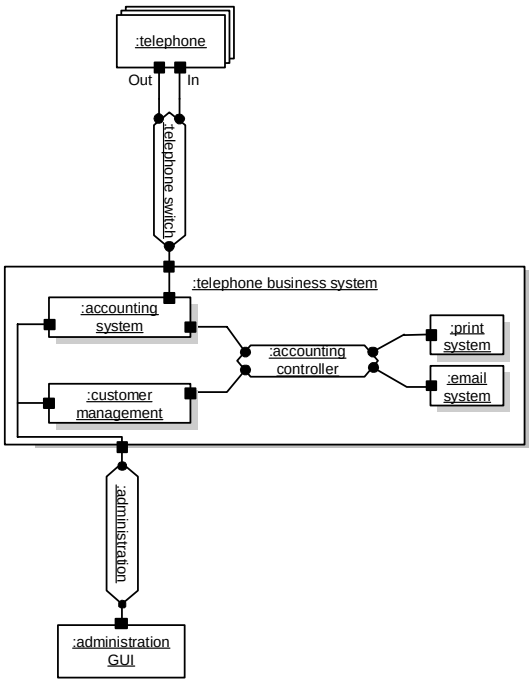


Figure 3-1 Example telephone business system

The chosen example is a telephone business system (figure 3-1). In this example *telephones* are connected with a *telephone switch*, which is responsible for connecting phones in order to allow users of the phones to communicate with each other. Every time a telephone call is completed the *telephone switch* transfers the corresponding data (source number, destination number, length of call, etc.) to the *telephone business system*. The *telephone business system* consists of the *accounting system*, the *customer management*, the *accounting controller*, the *print system* and the *email system*. The *accounting system* stores the call data. In order to create invoices the *accounting controller* gets customer data from the *customer management*, which is responsible for maintaining addresses, names, telephone numbers and other customer related data and the corresponding call data from the *accounting system*. The invoices are sent to a *print system* and/or an *e-mail system* component in order to be transferred to the customers. The whole *telephone business system* can be administrated via an *administration GUI* that is connected via an *administration* connector with the *telephone business system*.

Now we have a more detailed look at the *administration* connector. The linked components *administration GUI* and *telephone business system* are executed on different hardware platforms with different processor types, so they communicate by means of a public network. Thus, the connector is implemented with a middleware, such as CORBA or DCOM. The information flow through the connector is security relevant, because it contains the data of the customers and their bills. To prevent security problems, the connector must be replaced by a connector which encrypts/decrypts the transferred information. This is an often-recurring problem, which can be solved with a transformational pattern. As a solution the architecture and especially the connector can be transformed with the secure transmission connector pattern, which is described in section 5.2. The transformation substitutes the *administration* connector with the *secure transmission* [*administration*] connector, which is presented in Figure 3-2.

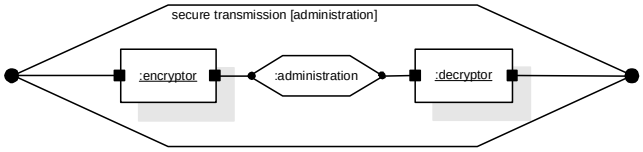


Figure 3-2 The improved connector *secure transmission* [*administration*]

4. CONCEPTS OF TRANSFORMATIONAL PATTERNS

In the introducing example an improved connector substitutes the former connector. This substitution does not change the functionality of the whole system. This is exactly the intention of a transformational pattern. Thus it can be easily applied in the software architectural phase and in later "lifecycle" phases. The usage of transformational patterns in later phases in the software system lifecycle is often necessary. This is because many problems, which are initiated from connectors, are identified during the runtime of the system. For example security problems in the telephone business system can only be detected when the system is operating. Thus the transformational patterns must be applied in the maintenance of the system

For the systematic construction of the improved connector it is necessary to use the former connector, as it

is presented in the introducing example (Figure 3-2). Therefore the mechanisms of high-order connectors from [Wermelinger et al. 2000] and [Garlan 97] can be taken as a suitable solution. This concept can be adapted for the transformational connector pattern. Thus, for the construction of the improved connector a pattern takes the former pattern as parameter. Therefore the topology of the patterns should contain at least one placeholder, which are used for the inclusion of the former connectors. The placeholders are represented in the used notation as connectors with a dashed line (cp. Figure 4-1).

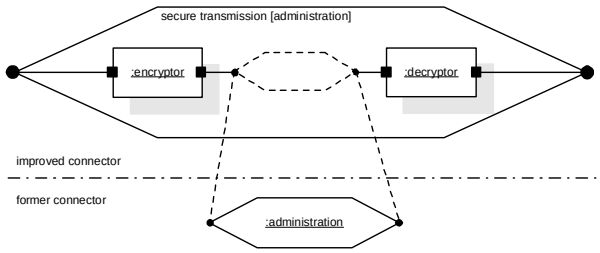


Figure 4-1 Inclusion of the former connector (topology)

For the inclusion of a connector into a placeholder it is necessary to pay attention to a set of rules. For example, a substitution is only feasible if the obeyed protocols are compatible. In addition to this, all roles of the included connector must be connected or bound. To ease this the presented pattern language only contains patterns and placeholders with two roles. Furthermore the substitution is not associative. This means that the order of nesting is relevant for the quality of the resulting connector. Furthermore the quality improvement is not always additive. That means, if for instance an architectural connector is transformed twice with the secure transmission pattern, the quality (secureness of the transmitted information) does not improve double.

5. TRANSFORMATIONAL CONNECTOR PATTERNS

In this chapter a set of possible transformational patterns for high-level architectural connectors is presented to confirm the presented concepts. These patterns can be found in many current applications. They all have a static topology, which is presented for each pattern. Furthermore all presented patterns and the containing placeholders own two roles, which eases the substitution.

The description of a single pattern in this chapter consists of the following two points:

- Problem description and context
- Solution.

The problem description points out the problem the pattern has to deal with. The context gives some impression about the relevant applications types and points out under which circumstances the problem occurs. Then, the main part of the pattern description presents the solution for the problem. This solution includes a textual and a topology description of the pattern. Furthermore, in the solution part the participating components and connectors are described.

5.1 Compressed Transmission Connector

5.1.1 Problem description and context

When a message is sent between distributed hardware nodes and messages contain a large amount of data the transmission of these messages needs a lot of time. This leads often to a missing of economical and soft real time requirements.

This problem can be found in modern web applications, where the amount of data constantly increases. Other relevant application types are for example distributed tools for processing video streams.

5.1.2 Solution

To solve this problem the messages should be compressed before sending and decompressed after receiving. For the compression and decompression an application specific algorithm should be used (i.e. MPEG for audio and video data or Zip/Rar for normal data). In figure 5-1 the general topology for this pattern is presented. The participating components are an instance of the component type *compressor* and of the component type *decompressor*. These two component types implement the algorithms for the compression and decompression. The fact that the message can only be sent from the *compressor* to the *decompressor results in an unidirectional* message flow in this pattern .

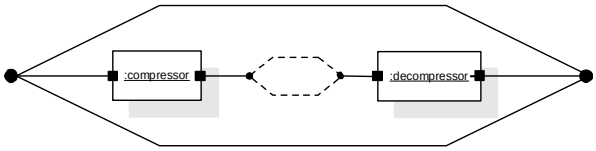


Figure 5-1 Topology of the pattern compressed transmission

If this pattern is used, a trade-off between the required time for the compression and decompression and the saved time by the transmission of the message should be considered. That is because for small and incompressible data the pattern and the compression algorithms are only an overhead.

5.2 Secure Transmission Connector

5.2.1 Problem description and context

If a message is sent via a connector in a public network, an unauthorized person can intercept this message. Hence a person who can read the message is capable of getting the contained information.

The problem can be found in the context of applications, which must secure the sending of confidential information. Typical fields of applications can be found especially in the military, business and banking sector.

5.2.2 Solution.

To solve this problem the messages should be encrypted. For this purpose, several algorithms exist (i.e. the RSA encryption standard), which encrypt a message with a special encryption key. These encryption algorithms must be implemented in the component type *encryptor* (cp. Figure 5-2). This component encrypts an arriving message and sends it to a component of the type *decryptor*. This component has to know the encryption key in order to restore the original message. Thus, an unauthorized person who intercepts the message and does not know the encryption key and the encryption algorithm is not able to get the contained information.

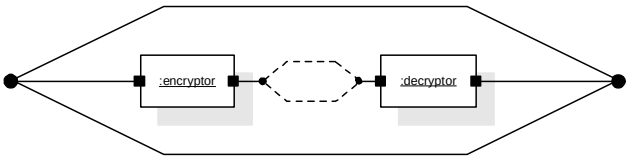


Figure 5-2 Topology of the pattern secure transmissions

5.3 Error Detection or Correction Connector

5.3.1 Problem description and context

The information contained in a message can be corrupted via sending this message over a connector. This may appear by a failure of the hardware channel, which is responsible for the transmission of the message. Such failure might be a bit error. Another way to corrupt a message is to intercept it and send a manipulated message to the receiver. If a message contains critical information a corrupted message leads to a failure of the system. These failures must be avoided especially for safety critical systems.

5.3.2 Solution.

As a solution for this problem a message should contain additional information. This information is redundant and describes the original message. For instance checksums like CRC (Cyclic Redundancy Checksums), parity bits and hamming codes can be used [Birolini 99]. The redundant information is generated in the instance of the component type *message-encoder* (compare the topology description of this pattern in figure 5-3). In the component *message-checker* the correctness of each arriving message is checked. Two handling strategies exist for incorrect or invalid messages. The first one identifies the failure and sends requests component a repeated transmission from the *message-encoder*. This is an optimistic strategy. The second strategy is a pessimistic one and requires a positive reply for each received valid message . If this sending component does not receive the positive reply after a specified amount of time, the message is sent again.

But not only an error detecting information can be sent. The receiving component could also correct a non-valid message. For this purpose an error correcting code should be used. This code needs a larger amount of redundant information than the error detecting code. Thus, an increase of the message transfer time and a trade-off between the correctness of the information and the message transfer time is necessary. Furthermore the decoding and error correction of the message times need computation time, which must be considered.

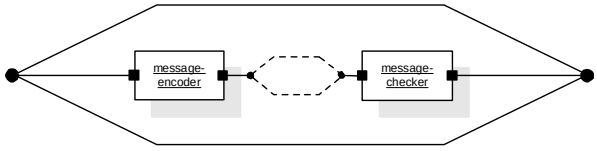


Figure 5-3 Topology of the pattern error detection or correction

5.4 Redundant Channel Connector

5.4.1 Problem description and context

Even though single message missing can lead to an accident, the transmission of the message must not fail necessarily. A loss of a message can occur for instance if the message sending hardware channels fail. The problem must be especially addressed in many safety critical technical systems. Such systems can be for instance railroad, avionic, military or automotive systems.

5.4.2 Solution.

In this pattern the messages are sent over a set of independent redundant connectors. In order to achieve this a component of the type *redundant-sender* sends the message to *n* independent channels. In the figure 5-4 three independent channels are used. The messages are received by a voting component (*voting-receiver*). This component implements a comparison strategy, which forwards a message only if from *m* channels identical messages are received. This leads to a *m*-from-*n* voting strategy. Typical voting strategies are two-from-two, two-from-three and three-from-five[Birolini 99].

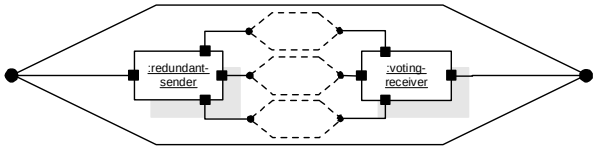


Figure 5-4 Topology of the pattern redundant channel

If you apply this transformational pattern, consider also the reliability of the sending and voting component.

5.5 Adapter Connector

5.5.1 Problem description and context

The reuse of large grained components has a benefit for the development cost and time. But this reuse is problematic if the reused components are incompatible with the existing components. This incompatibility is based on the incompatibility of the obeyed protocols. Occurring problems are messages with different names or different data types. More problematic than this are incompatibilities in the order in which the messages have to be sent.

5.5.2 Solution.

This problem can be solved if the connectors are enriched by an adapter mechanism. This adapter mechanism must be implemented in the component type *adapter*. For each problem the adapter mechanism must be defined separately, so that no common code can be used for the component type *adapter*.

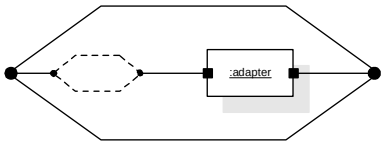


Figure 5-5 Topology of the pattern adapter

The original adapter pattern can be found in [Gamma et al. 95]

5.6 Split Bi-directional Transmission Connector

5.6.1 Problem description and context

Many of the presented patterns can only support a unidirectional message flow. Therefore it is necessary to split the bi-directional message flow into two unidirectional message flows.

5.6.2 Solution.

In figure 4-1 the general topology for this pattern is presented. The instances of the component type *sender* select all messages and send them to the opposite instance of the component type *receiver*. Thus, in this topology the two directions of the message flow can be identified. The two containing placeholders represent these directions. They can be substituted by a high-level architectural connector pattern, which only supports a unidirectional message flow.

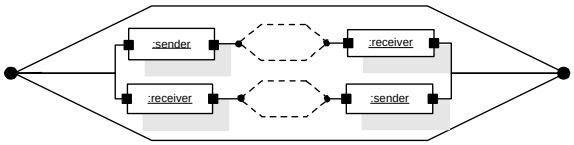


Figure 5-6 Topology of the pattern Split Bi-directional transmission connector

6. CONCLUSION AND FUTURE WORK

In this paper a set of transformational patterns for high-level architectural connectors was presented. These patterns address several often-occurring problems in the interconnection of architectural components. Furthermore, with these patterns and a set of basic connectors and components it is possible to construct systematically an improved connector, which substitutes the former connector.

Besides, this work was not intended to be complete since it can be extended by other high-level architectural patterns. These extensions are necessary especially for patterns, which implement multicast functionalities. The presented pattern cannot handle these at the time.

For future work in this area an implementation for each of the participating components and connectors in a pattern should be given. Afterwards, the improved connector can be constructed automatically.

7. REFERENCES

[Agha 88]
G.A. Agha. Actors: a model of concurrent computation in distributed systems, MIT Press, 1988.

[Agha 97]
G. A. Agha, Abstracting Interaction Patterns: A Programming Paradigm for Open Distributed Systems, Formal Methods for Open Object-based Distributed Systems, IFIP Transactions, E. Najm and J.-B. Stefani, Eds., Chapman & Hall, 1997.

[Alfaro,Henzinger 01]
L. de Alfaro and T.A. Henzinger. Interface automata. Proceedings of the 9th Annual Symposium on Foundations of Software Engineering (FSE), ACM Press, pp. 109-120, 2001.

[Allen, Garlan 97]
R. Allen and D. Garlan. A Formal Basis for Architectural Connection. ACM Transaction on Software Engineering and Methodology, July 1997.

[Bálek 02]
D. Bálek. Connectors in Software Architectures, Ph.D. Thesis, March 2002.

[Bálek, Plasil 00]
D. Bálek and F. Plasil. Software Connectors: A Hierarchical Model. Tech. Report No. 2000/2, Dep. of SW Engineering, Charles University, Prague, 2000.

[Buschmann et al. 96]
F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal, Pattern-Oriented Software Architecture - A System of Patterns, John Wiley & Sons, 1996.

[Eugster et al. 01]
P. Eugster, R. Guerraoui, and C. Damm. On objects and events. Proceedings for OOPSLA 2001, Tampa Bay, Florida, October 2001.

[Fowler 99]
M. Fowler. Refactoring: Improving the Design of Existing Code. Addison-Wesley, 1999.

[Gamma et al. 95]
E. Gamma, R. Helm, R. Johnson, J. Vlissides. Design Patterns, Elements of Reusable Object-oriented Software, Addison-Wesley 1995.

[Garlan et al. 95]
D. Garlan, R. Allen, J. Ockerbloom, Architectural mismatch or why it's hard to build systems out of existing parts, Proceedings of the 17th international conference on Software engineering, p.179-185, Seattle, Washington, 1995.

[Garlan 98]
D. Garlan, "Higher-order connectors", Position paper for the Workshop on Compositional Software Architectures, January 1998.

[Hofmeister et al. 99]
C. Hofmeister, R. Nord and D. Soni, Applied Software Architecture, Reading, MA: Addison Wesley Longman, 1999

[Medvidovic et al. 96]
N. Medvidovic, P.Oreizy, J. Robbins and R. Taylor. Using Object-Oriented Typing to Support Architectural Design in the C2 Style, Proceedings of ACM SIGSOFT'96: 4th Symposium on the Foundations of Software Engineering (FSE4), pp. 24-32, San Francisco, California, October 1996.

[Medvidovic et al. 97]
N. Medvidovic, P. Oreizy, and R. N. Taylor. Reuse of Off-the-Shelf Components in C2-Style Architectures. Proceedings of the 1997 Intemational Conference on Software Engineering (ICSE'97), Boston, MA, May 1997.

[Medvidovic, Rosenblum 99]
N. Medvidovic and D. S. Rosenblum. Assessing the Suitability of a Standard Design Method for Modeling Software Architectures. Proceedings of the First Working IFIP 52 Conference on Software Architecture (WICSA1), San Antonio, TX, February 1999.

[Mehta et al. 00]
N. Mehta, N. Medvidovic and S. Phadke. Towards a Taxonomy of Software Connectors, Proceedings of the 22nd International Conference on Software Engineering, Limerick, Ireland, June 2000.

[OMG 01]
OMG ormsc/01-07-01: Model Driven Architecture (MDA). 2001.

[Plasil et al. 01]
F. Plasil, S. Visnovsky and M. Besta, Behavior Protocols, Tech. Report No. 2000/7, Dep. of SW Engineering, Charles University, Accepted for publication for the IEEE Transactions on Software Engineering. Prague, 2001.

[Shaw 93]
M. Shaw. Procedure calls are the assembly language of system interconnection: Connectors deserve first-class status. In Proceedings of the Workshop on Studies of Software Design, May 1993.

[Shaw, Garlan 96]
M. Shaw & D. Garlan. Software Architecture: Perspectives on an Emerging Discipline. Prentice Hall, 1996

[Birolini 99]
A. Birolini. Reliability engineering: theory and practice (third ed.), New York, Springer, 1999.

[Szyperski 98]
C. Szyperski: Component Software. Beyond Object-Oriented Programming. ACM Press/Addison Wesley, 1998

[Taylor et al. 96]
R. N. Taylor, N. Medvidovic, K. M. Anderson, E. J. Whitehead, J. E. Robbins, K. A. Nies, P. Oreizy and D. L. Dubrow. A Component- and Message-Based Architectural Style for GUI Software. IEEE Transaction on Software Engineering, 22(6), 1996.

[Wermelinger et al. 00]
M. Wermelinger, A. Lopes and J. L. Fiadeiro. Superposing Connectors. Proc. 10th International Workshop on Software Specification and Design. IEEE Computer Society Press 2000.

APPENDIX

8. COMPONENT-CONNECTOR-MODEL

As motivated in the paper, components and connectors should be used as first class entities [Perry, Wolf 92] of the architectural specification. The usage of the two architectural element types leads to a model for the software architecture known as component-connector model (CCM). A detailed description of the CCM can be found in [Hofmeister et al. 99]. Further information about this topic are presented in [Medvidovic et al. 97, Medvidovic et al. 96, and Taylor et al. 96]. They describe the C2 architecture style, which uses the component-connector-model in restricted form and focuses on GUI-specific applications.

8.1 General Overview

The meta-model of the component connector-model is presented in Figure 2-1. It is shown, that the software architecture is described by a set of components and connectors. The only way to interact with the environment of these connectors and components is to use interfaces. Thus, to each component a set of ports and to each connector a set of roles is assigned. These ports and roles obey exactly one protocol, which specifies the order of the incoming and outgoing messages.

To build more complex systems, component and connector can be hierarchically decomposed. Thus, in each component and connector other architectural elements can be nested. This leads to a composition hierarchy that can be represented as a tree. In this tree the leaf nodes are basic architectural elements, which cannot be further decomposed. The top node is the system as a whole.

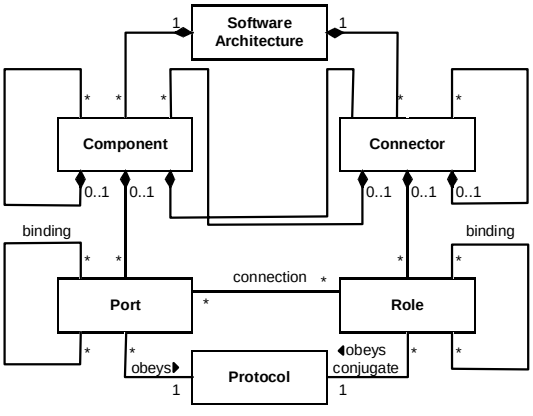


Figure 8-1 Meta-model of a component-connector-model [Hofmeister et al. 99]

Furthermore, the components and connectors must be interconnected. For this, two general mechanisms exist. The first one is the connection mechanism, which interconnects a component and a connector through ports and roles. For the connection mechanism the component and the connector must be in the same level in the composition hierarchy. The second one is the bind mechanism, which is used to bind an inner port to a port of the enclosing component, or to bind an inner role to a role of the enclosing connector respectively. To ease the understandability of the figures in this paper, these connection mechanisms are not used in a strict way.

8.2 Construction of a composite connector

For the construction of a composite connector the following BNF (Backus-Naur-Form) is used, which introduce the in chapter 4 presented placeholder to the CCM:

<connector>	:: =	<composite connector> | **<basic connector>**
<composite connector>	:: =	(<**component**> | <connectors>)+ | <connector pattern>
<connector pattern>	:: =	(component | <connectors> |<placeholder>)+
<placeholder>	:: =	connector

The language contains two terminal symbols, the basic connector and the components. They are marked bold and in angel brackets In addition to this the language contains with the connector, the composite connector, the connector pattern and the placeholder four non-terminal symbols. With the BNF it is shown that a connector can be constructed with the terminals (the basic connector and the components). Furthermore, the substitution mechanism is presented. Thus, a connector can substitute a placeholder. This connector can be a composite or a basic connector. The composite connector can be build as postulated in the meta-model of the CCM (cp. figure 8-1) or it can be a build with a connector pattern.

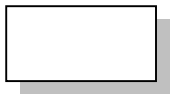
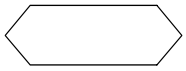
The components are terminal symbols to ease the language. In a complex architectural language the components can also decomposed and so they must be represent by non-terminal symbols. This is also described in section 2 and the meta-model of the CCM.

8.3 Representation

To specify the software architecture with the component-connector-model a notation is necessary. For this notation the UML can be tailored with stereotypes. A suggestion for the stereotypes is given in [Hofmeister et al. 99]. The stereotypes can also be represented by

graphical symbols. These symbols are presented in Table 8-1, and are used in this paper.

Table 8-1 Types of the component-connector-model

Type	Stereotype	Graphical Symbol
Component	<<component>>	
Connector	<<connector>>	
Placeholder	<<connector-placeholder>>	
Port	<<port>>	
Role	<<role>>	
Protocol	<<protocol>>	

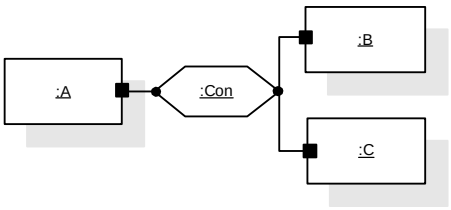


Figure 8-2 Example topology

Now we reuse the abstract example presented in Figure 2-1, to explain the usage of the notation (cp. Figure 8-2). In this example topology the three components *A* ,*B* and *C* are represented by shadowed rectangles. The symbol used for the connector *Con* is an elongated hexagon. These components and connectors have interfaces, called ports and roles, which are represented by small black squares and circles and placed on the rectangle's and hexagon's edges. A connection between ports and roles is shown with a UML association.
For further interest in the usage of this notation and their elements in [Hofmeister et al. 99] some concrete examples are given.

Methods for States

A Pattern for Realizing Object Lifecycles

Kevlin Henney
kevlin@curbralan.com
kevlin@acm.org

March 2003

Abstract

Substance doesn't change. Method contains no permanence. Substance relates to the form of the atom. Method relates to what the atom does. In technical composition a similar distinction exists between physical description and functional description. A complex assembly is best described first in terms of its substances: its subassemblies and parts. Then, next, it is described in terms of its methods: its functions as they occur in sequence. If you confuse physical and functional description, substance and method, you get all tangled up and so does the reader.

Robert M Pirsig, *Zen and the Art of Motorcycle Maintenance*

The intent of the METHODS FOR STATES pattern is to encapsulate modal behavior of an object within a single class. In stateful objects with strongly modal lifecycles, the behavior of a given method can be history sensitive, differing significantly according to the current state of the object. Simple lifecycle models can be implemented in terms of flags and conditional statements in each method; an approach whose code comprehensibility scales poorly. More sophisticated modal behavior can be realized through object delegation, drawing on a community of dependent classes to express the behavior. However, the larger the community, the more pronounced coupling and comprehensibility problems become.

Using METHODS FOR STATES a class is able to express all of the different behaviors as ordinary methods. It can do so without either the control coupling and reduced readability of large conditional statements or a large supporting cast of ancillary classes. Indirection, based on referring to methods as objects, is used to both represent the state and dispatch from a public method request to the correct underlying method for the state.

A structured but lightweight pattern form is used in the paper: the *problem* is summarized; a worked *example* with code presented in C++ and Ruby follows, exploring some possible solutions but not the pattern's proposed solution; the *forces* that circumscribe the problem are listed; the pattern *solution* is described; a *resolution* of the example is presented; the pattern *consequences* are then detailed; an *appendix* lists problem–solution thumbnails for related patterns; *acknowledgments* and *references* are listed at the end.

Problem

Some types of stateful object can be said to have strongly modal lifecycles. In such an object the behavior of some of its methods appears to alter significantly over the course the object's life. It is possible to distinguish different *modes* of operation — sometimes known as *macrostates* or, more ambiguously, *states* — that collectively describe such behaviors. A method's behavior depends on the object's internal state, including its current mode, and a change in mode comes about as a response to an event, such as a method call.

Assuming that each mode does not have associated state of its own, what is the most effective configuration for expressing the mode and its dependent behaviors?

Example

Consider a simple digital clock:

- It displays hours and minutes in normal 24-hour time.
- Two buttons can be used to adjust the current time: one button to change the mode and another to increment the hour or minute, depending on the mode.
- A heartbeat event is generated internally once a second to allow the clock to update. To keep the example simple, no latency is assumed.

The clock has three specific modes:

- Displaying the time, in which the increment button is ignored and the update event advances the stored time.
- Setting the hours, in which the clock ignores the update event and the increment button increments the displayed hour value.
- Setting the minutes, in which the clock ignores the update event and the increment button increments the displayed minute value.

This is not exactly a complex state machine, but there is enough here to get your teeth into. What are the options available for implementing such a state model?

Flags for States in C++

Let's start what might be termed the FLAGS FOR STATES pattern (see *Appendix*), expressed using a flag with a `switch` accompaniment in C++. The following code shows the essential public interface and a simple representation:

```
class clock
{
public:
    void change_mode();
    void increment();
    void tick();
    ...
private:
    enum mode
    {
        displaying_time, setting_hours, setting_minutes
    };
    mode behavior;
    int hour, minute, second;
};
```

The clock has a simple lifecycle model: the only significant behavioral changes come from the change-mode button. The change in mode is independent of data values or any intermediate behavior. Therefore, only the `change_mode` function affects `behavior`:

```
void clock::change_mode()
{
    static const mode next[] =
    {
        setting_hours, setting_minutes, displaying_time
    };
    behavior = next[behavior];
}
```

This particular implementation takes advantage of an `enum`'s implicit conversion to an integer, using it to lookup the next state in a table. More verbosely, a `switch` statement could have been used to achieve the same effect. More tersely, the next state can be calculated by incrementing the current one, wrapping around from last to first as necessary, because the state transition model forms a simple cycle.

The realization of `increment` is less open to alternatives:

```
void clock::increment()
{
    switch(behavior)
    {
    case displaying_time:
        break;
    case setting_hours:
        hour = (hour + 1) % 24;
        break;
    case setting_minutes:
        minute = (minute + 1) % 60;
        break;
    }
}
```

The empty `case` for `displaying_time` is included for completeness, demonstrating statewide coverage for the event explicitly. The event response for `tick` is complementary:

```
void clock::tick()
{
    switch(behavior)
    {
    case displaying_time:
        if(++second == 60)
        {
            second = 0;
            if(++minute == 60)
            {
                minute = 0;
                hour   = (hour + 1) % 24;
            }
        }
        break;
```

```
    case setting_hours:
        break;
    case setting_minutes:
        break;
    }
}
```

There is a temptation to tow an orthodox object-oriented hard-line and come down against flags and `switch`es, assuming them to be an indicator of deficient design. However, this line is a narrow and not always convincing one. For the scope and scale of the given problem almost any other solution not based on explicit selection will be longer and more intricate.

However, if we imagine a slightly broader version of the problem, the context shifts and the flag and `switch` solution becomes increasingly — indeed, in terms of lines of code, exponentially — inappropriate. Consider adding an alarm feature to the clock, and perhaps a date feature. Taking it further, a digital watch typically offers all these features and more: stopwatch, multiple time zones, phone book, etc. The FLAGS FOR STATES design will collapse under its own weight, acquiring the flexibility of a block of concrete, the cohesiveness of loose sand, and the plot comprehensibility of a telephone directory. Such consequences would justify the pursuit of alternative approaches.

Objects for States in Ruby

The OBJECTS FOR STATES pattern† (see *Appendix*) offers itself up as a likely candidate. The organizing principle behind the pattern is the introduction of an object to represent the behavior of the main object — the clock in this case — in each mode. The behavioral object offers a method for each event the main object can respond to — in this case `change_mode`, `increment`, and `tick`.

Here is a sketch of this from the main object's perspective in Ruby:

```
class Clock
    def initialize(hour, minute, second)
        @now      = TimeOfDay.new(hour, minute, second)
        @behavior = DisplayingTime.new
    end
    def change_mode
        @behavior = @behavior.change_mode(@now)
    end
    def increment
        @behavior = @behavior.increment(@now)
    end
    def tick
        @behavior = @behavior.tick(@now)
    end
end
```

All the responsibility for behavior is forwarded to the behavior object. For each mode of behavior there is a class that implements the same method interface, namely `change_mode`, `increment`, and `tick`. Explicit `switch`-like selection has been replaced with runtime polymorphism, and each delegated method selects the next behavior. When there is a state

† OBJECTS FOR STATES is also known as the STATE pattern, but this name is misleading and not particularly descriptive. It is misleading because it is often mistaken as definitive: *the* state pattern. It is not particularly descriptive because there is no suggestion of the solution structure in the name, just a hand-waiving reference to the problem. The OBJECTS FOR STATES name is listed [Gamma+1995] as a synonym for STATE. It more accurately captures the pattern's intent and structure, and should be preferred.

transition the delegated method will return the mode object for the new state; when there is no transition, the delegated method simply returns `self`. The behavior for each mode has been localized and encapsulated rather than scattered across different `cases` in many functions. The reason for passing the `@now` instance variable becomes apparent when you consider that the behavioral objects need to work with their context, so they must also affect the state of their associated `Clock` object's data. The type of the current time is a simple data structure with fields for the current hour, minute, and second:

```
TimeOfDay = Struct.new(:hour, :minute, :second)
```

Objects are handled by reference rather than by copy, so the behavioral objects modifying the passed data object affect the state of the main clock object. Keeping to a fairly conventional OO style:

```
class DisplayingTime
    def change_mode(time_of_day)
        SettingHours.new
    end
    def increment(time_of_day)
        self
    end
    def tick(time_of_day)
        if (time_of_day.second += 1) == 60
            time_of_day.second = 0
            if (time_of_day.minute += 1) == 60
                time_of_day.minute = 0
                time_of_day.hour   = (time_of_day.hour + 1) % 24
            end
        end
        self
    end
end

class SettingHours
    def change_mode(time_of_day)
        SettingMinutes.new
    end
    def increment(time_of_day)
        time_of_day.hour = (time_of_day.hour + 1) % 24
        self
    end
    def tick(time_of_day)
        self
    end
end

class SettingMinutes
    def change_mode(time_of_day)
        DisplayingTime.new
    end
    def increment(time_of_day)
        time_of_day.minute = (time_of_day.minute + 1) % 60
        self
    end
    def tick(time_of_day)
        self
    end
end
```

The behavioral object is sometimes also known as the *state object*, but this name is confusing: as you can see from the code, the state object is often stateless. In this case, because the state object is stateless and immutable, a single instance for each concrete class will suffice, avoiding the need for dynamic object creation. The appropriate solution in this case is simply to use a class variable. There is a temptation to use a SINGLETON [Gamma+1995], but this temptation should be resisted. In this case, as in many others, it overcomplicates the design. The use of a single, shared instance for each mode is the business of the `Clock` class, not the mode's, and an ordinary class variable is both simple and sufficient.

Another temptation that would offer little in return would be to introduce a common superclass for the `DisplayingTime`, `SettingHours`, and `SettingMinutes` classes. The classes currently share the same implicit interface, but no effort is made to factor out any common behavior. In some designs this may make sense, but there is little to be gained by adding an extra class to the current example. The only duplicated code is the empty `tick` method in both the `SettingHours` and the `SettingMinutes` classes. Such light and incidental duplication of nothingness does not really warrant extracting a superclass.

The implementation of the state model using objects and polymorphism is elegant, albeit longer than the explicit-conditional approach. Its broader benefits are not as easily discernible as they would be in a more extensive state model. A relatively large community of specific classes and methods seems to have sprung up to solve a comparatively simple problem.

Forces

History-sensitive method behavior implies that additional object state is required to track the current mode. This state needs to be clear and easy to manage. Handling the additional state should also not weigh down the implementation of each method with additional complexity.

Simple lifecycle models can be implemented in terms of flags and conditional statements in each method; an approach whose code comprehensibility scales poorly. More sophisticated modal behavior can be realized through object delegation, drawing on a community of dependent classes to express the behavior, typically using OBJECTS FOR STATES [Gamma+1995]. However, the larger the community, the more pronounced coupling and comprehensibility problems become. If only a single method is history sensitive, elaborating a whole class hierarchy certainly seems like overkill.

Particular method behaviors may be shared across different methods in different states: for example, methods that do nothing or methods that trigger a particular event, such as an exception. Avoiding duplicate code is generally considered a good — indeed, fundamental — practice. Common behavior can be factored out into private methods and called from the relevant `case` when using FLAGS FOR STATES. Alternatively, a class hierarchy, based typically on nesting of states, allows common method implementations to be pulled up the hierarchy. However, this does not accommodate behaviors that are common but not related so simply, crosscutting the hierarchy.

Some developers have made the mistake of assuming that OBJECTS FOR STATES is the "one true way" to realize state models in code. This is partly because of its inclusion in *Design Patterns* [Gamma+1995] as the only object lifecycle pattern and partly because of its branding as *the* STATE pattern. The pattern is powerful and certainly not trivial; applying it

uniformly to code as a cure-all for all state models is sure to complicate the source and confuse the reader.‡

The benefit of a flag-based approach is that all the behavior is defined in a single class rather than across many. Access to the context of the main object is also simple: ordinary methods have such direct access, which is generally not the case for separate objects. However, the effect on each individual method is to obfuscate rather than clarify intent. Each history-sensitive method suffers from strong control coupling to the flag variable.

What is needed of a solution is the ability to express and select each different behavior simply, without interference from explicit conditional statements or separation across multiple classes.

Solution

Represent an object's mode as a simple data structure containing references to methods. Each history-sensitive public method of the object forwards a call, along with any arguments received, to a corresponding entry in the data structure. Each different behavior for the object is implemented as its own private method. Each mode is associated with its own data structure instance, which holds references to the relevant private methods.

The method references may be true direct method references, such as member function pointers in C++, or they may take the form of the symbolic method names that are resolved using reflection.

The data structure holding the method references can be a record-like data structure with named fields, such as a C++ `struct`. Alternatively, a dictionary object can be used to look up the private method reference corresponding to each history-sensitive public method. In effect, this configuration emulates the normal method lookup table (*vtable*) mechanism, with a little added customization, evolution, and intelligence. Where only a single public method is history sensitive, no intermediate data structure is needed to represent the mode: a single method reference will suffice. Global, module, or class-wide variables can be used to hold the single instance of the data structure (or method reference) required for each mode.

Resolution

Returning to the clock example, the forces occurring in both the C++ and Ruby designs can be resolved conveniently and idiomatically with METHODS FOR STATES.

Methods for States in C++

In C++ the design can be modified so that each public member function forwards its call to the relevant member function pointer in a `struct` indicative of the current mode:

```
class clock
{
public:
    void change_mode()
    {
        (this->*(behavior->change_mode))();
    }
```

‡ Misapplication of OBJECTS FOR STATES only presents itself as a force as a consequence of common programmer design pattern knowledge: a few years ago, before *Design Patterns* became widely read as an OO design book, it would not be considered a force; it is possible that the same may be true in the future for different reasons.

```
    void increment()
    {
        (this->*(behavior->increment))();
    }
    void tick()
    {
        (this->*(behavior->tick))();
    }
private:
    typedef void (clock::*function)();
    struct mode
    {
        const function change_mode, increment, tick;
    };
    static const mode displaying_time
    static const mode setting_hours;
    static const mode setting_minutes;
    const mode *behavior;
    int hour, minute, second;
    ...
};
```

A palette of suitable behavior is provided by private member functions:

```
class clock
{
    ...
private:
    ...
    template<const mode *next_mode>
    void change_to()
    {
        behavior = next_mode;
    }
    void next_hour()
    {
        hour = (hour + 1) % 24;
    }
    void next_minute()
    {
        minute = (minute + 1) % 60;
    }
    void update_time()
    {
        if(++second == 60)
        {
            second = 0;
            if(++minute == 60)
            {
                minute = 0;
                hour   = (hour + 1) % 24;
            }
        }
    }
    void do_nothing()
    {
    }
};
```

And the rest is down to initialization, letting the data do the work:

```
const clock::mode clock::displaying_time =
{
    &clock::change_to<&setting_hours>, &clock::do_nothing, &clock::update_time
};
const clock::mode clock::setting_hours =
{
    &clock::change_to<&setting_minutes>, &clock::next_hour, &clock::do_nothing
};
const clock::mode clock::setting_minutes =
{
    &clock::change_to<&displaying_time>, &clock::next_minute, &clock::do_nothing
};
```

Methods for States in Ruby

To implement the clock example in Ruby, class variables — prefixed with `@@` — and `Struct` objects are used:

```
Mode = Struct.new(:increment, :tick)

class Clock
    @@displaying_time = Mode.new(:do_nothing, :update_time)
    @@setting_hours   = Mode.new(:next_hour, :do_nothing)
    @@setting_minutes = Mode.new(:next_minute, :do_nothing)
end
```

Method calls are forwarded by accessing the appropriate method name from the corresponding `Struct` attribute, and resolving it against the current object:

```
class Clock
    def initialize(hour, minute, second)
        @hour, @minute, @second = hour, minute, second
        @behavior = @@displaying_time
    end
    def change_mode
        send(@behavior.change_mode)
    end
    def increment
        send(@behavior.increment)
    end
    def tick
        send(@behavior.tick)
    end
end
```

A lookup table simplifies the `change_mode` method, so that only a single implementation is required:

```
class Clock
    @@mode_changes =
    {
        @@displaying_time => @@setting_hours,
```

```
        @@setting_hours   => @@setting_minutes,
        @@setting_minutes => @@displaying_time
    }
    def next_mode
        @behavior = @@mode_changes[@behavior]
    end
    def next_hour
        @hour = (@hour + 1) % 24
    end
    def next_minute
        @minute = (@minute + 1) % 60
    end
    def update_time
        if (@second += 1) == 60
            @second = 0
            if (@minute += 1) == 60
                @minute = 0
                @hour   = (@hour + 1) % 24
            end
        end
    end
    def do_nothing
    end
end
```

Because the underlying implementation of `change_mode` is the same no matter what the mode, i.e. `next_mode`, it need not participate in the dynamic lookup. It has therefore been excluded for the method referencing structure.

Consequences

Using METHODS FOR STATES allows a class to express all of its different behaviors in ordinary methods on itself. It achieves its behavior without either the control coupling and reduced readability of large conditional statements or a large supporting cast of ancillary classes.

Indirection, based on referencing methods as objects in their own right, is used both to represent the state and to dispatch from a public method request to the correct underlying method for the current mode. The overall control flow and effect is that of DOUBLE DISPATCH (see *Appendix*). In terms of performance, METHODS FOR STATES requires an additional two levels of indirection to resolve a method call.

The behavior of the class's objects is fully encapsulated within the class, the same unit of code to which the behavior is coupled, rather than fragmented across multiple small classes. The only additional data type required is for the data structure holding the method references, which may be defined as a nested or module-level type, or may exist as an associative collection, or may be nothing more than a single method reference. There is no proliferation of classes, nested or otherwise. On the other hand, if there are many distinct behaviors, the main class may end up far longer than was intended or is manageable — a shopping list of different options.

Each distinct behavior is assigned its own method. This allocation of responsibility is clearer and more cohesive than asking the reader or class author to wade through potentially large rambling selection statements. The independence of public methods from their runtime implementation also allows more fluidity in the modes and their transitions, and improves opportunities for sharing of common behavior between different modes:

- `switch` statements are notoriously tedious and error prone for such extension — especially in the large — leading to both bugs and duplicate code.
- Representing modes in a class hierarchy supports simple addition of new modes, but allows convenient sharing of common implementation only between a macrostate and its superstate. This relationship is mirrored in the class hierarchy, so method sharing across unrelated states is inconvenient.

With METHODS FOR STATES, common behavior is capitalized on more readily than in other designs, whether in the form of doing nothing or changing mode in a consistent fashion. The more that event responses between different modes overlap, the more suitable this pattern becomes.

The potential independence of public methods from the underlying private methods means that method names cannot always be relied upon to indicate the context of calling. The initialization of each mode's data structure instance must be read to understand what methods are used in what modes and to what end.

Other than the one-off initialization of each mode's corresponding lookup data structure, no additional object creation is needed to support METHODS FOR STATES. The data structure instances can be initialized at the earliest opportunity — program startup or class load time — and remain unchanged. Because they are immutable the sharing is intrinsically thread-safe.

No object context needs to be passed around because ordinary methods can already see the internal state of the object they represent the behavior of. This directness makes the development and comprehension of each specific method behavior simpler: no additional arguments are required for the underlying methods; no tricks with indirection or data visibility are required to deal with whatever restrictions the language places on an object's access to the internals of another. Although the use of closure-based objects — such as Java's inner classes — simplify the context-access issues from one dependent object on another, such an approach requires a great deal more object creation to work; a one off at-startup initialization will not do the trick.

METHODS FOR STATES is most suited to languages that support simple method references and resolution. For example, in C++ a member function pointer is resolved against an object pointer using the `->*` operator. Similarly, in C plain function pointers support the simple implementation of this pattern in a non-object-oriented context. In Ruby a method's symbolic name can be invoked on an object using the `send` method that is common to all objects. In contrast, Java's reflection mechanism requires more gymnastics — and correspondingly more obscurity and less convenience — to call a method on an object given its name as a string. C# could be said to suffer a similar problem, but it also has features that support a simple enough workaround: `delegates` and `static` methods. Instead of expressing the behaviors as ordinary private methods, they can be expressed as private `static` methods that take an additional argument to an instance of the class, effectively emulating the `this` reference implicit in ordinary methods. For `static` methods, `delegate` variables behave much as function pointers do in C, allowing a cleaner and more efficient implementation of METHODS FOR STATES than is possible with reflection.

METHODS FOR STATES can often provide a simpler and more manageable alternative to OBJECTS FOR STATES (see *Appendix*), a pattern with similar intent but markedly different structure and options. Incorporated into existing state pattern languages [Dyson+1998, Yacoub+2000], METHODS FOR STATES would expand the vocabulary, options, and range of designs available to a programmer beyond the narrower set offered by an OBJECTS FOR STATES view of state machines. Where per-mode state is needed, OBJECTS FOR STATES offers a more cohesive design than trying to shore up METHODS FOR STATES with extra optional variables. In languages, such as Java, that require awkward code to realize METHODS FOR

STATES, or that lead to the creation of further objects and classes, an OBJECTS FOR STATES solution is often preferable.

METHODS FOR STATES is not a viable substitute in situations that better suit COLLECTIONS FOR STATES (see *Appendix*), but it can be used as a complement. COLLECTIONS FOR STATES groups multiple objects together according to their mode, representing the concept of the mode extrinsically. Objects in the same mode are held in the same collection, reducing the need for intrinsic mode representation. However, where COLLECTIONS FOR STATES is being applied as a speed optimization, e.g. acting collectively on modal objects that already have an internal lifecycle model, retaining METHODS OF STATES internally may still make sense.

Appendix

The following table lists thumbnails for patterns external to this paper that are related in some way to METHODS FOR STATES. OBJECTS FOR STATES provides a common alternative to METHODS FOR STATES, and vice-versa. Similarly FLAGS FOR STATES can be used in limited scenarios where either METHODS FOR STATES or OBJECTS FOR STATES might be regarded as overkill. COLLECTIONS FOR STATES is a complementary pattern: it may be applied in its own right or in conjunction with METHODS FOR STATES, OBJECTS FOR STATES, or FLAGS FOR STATES, but not as a substitutable alternative. DOUBLE DISPATCH describes the most generalized form of the dispatch used in the heart of METHODS FOR STATES.

Name	*Problem*	*Solution*
COLLECTIONS FOR STATES [Henney1999]	A number of objects are managed and held in a collection, and operated on according to their common state. What is a suitable expression of the state with respect to each object?	Represent each state of interest with a separate collection that refers to all objects in that state. State transitions become transfers between collections.
DOUBLE DISPATCH [Beck1997]	How can you select a method based on the type of the target and the type or value of one other variable without hardwiring the selection as a conditional statement?	Delegate the selection of the actual method via a helper object that then calls back on the main object. The type of the helper object determines which method is selected. The helper object is normally the other variable in the interaction.
FLAGS FOR STATES	How can an object significantly change its behavior for only a couple of methods based on only one or two alternative internal states?	Represent the behavioral state of the object explicitly using a flag. In each of the history-sensitive methods, use a conditional to check the flag and act accordingly.
OBJECTS FOR STATES [Gamma+1995, Dyson+1998]	How can an object significantly change its behavior, depending on its internal state, without hardwired multi-part conditional code?	Separate the behavior from the main class, which holds the context, into a separate class hierarchy where each class represents the behavior in a particular state. Method calls on the context are forwarded to the mode object.

Acknowledgements

This paper is derived, in part, from a previously published article [Henney2002].

I would like to thank Pascal Costanza for his excellent shepherding and his patience, Jon Jagger for his additional comments on the preconference version of the paper, and to the participants in the writer's workshop at VikingPLoP 2002: Mikio Aoyama, Walter Cazzola, Lars Grunske, Juha Parssinen, Michael Pont, and Kristian Elof Sørensen.

References

[Beck1997] Kent Beck, *Smalltalk Best Practice Patterns*, Prentice Hall, 1997.

[Dyson+1998] Paul Dyson and Bruce Anderson, "State Patterns", *Pattern Languages of Program Design 3*, edited by Robert Martin, Dirk Riehle, and Frank Buschmann, Addison-Wesley, 1998.

[Gamma+1995] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1995.

[Henney1999] Kevlin Henney, "Collections for States", *Proceedings of the 4th European Conference on Pattern Languages of Programs*, 1999, `http://www.curbralan.com`.

[Henney2002] Kevlin Henney, "State Government", *C/C++ Users Journal* C++ Experts Forum, June 2002, `http://www.cuj.com/experts/2006/henney.htm`.

[Yacoub+1998] Sherif M Yacoub and Hany H Ammar, "Finite State Machine Patterns", *Pattern Languages of Program Design 4*, edited by Neil Harrison, Brian Foote, and Hans Rohnert, Addison-Wesley, 2000.

Universal Enterprise Model: Business Pattern Language

Pavel Hruby

Microsoft Business Solutions
Frydenlunds Allé 6
DK-2950 Vedbaek, Denmark
E-mail: phruby@acm.org

Abstract

Have you ever tried to describe an object model of a business system and struggled to find the right relationships between business entities, such as customers, business partners, products, sales and purchase orders, invoices and credit memos? Have you ever wanted to know a simple rule for modeling the business system in a consistent manner? The Universal Enterprise Model is a pattern language for building extensible models of business systems. The fundamental structure of the business system is derived from the resources-events-agents (REA) pattern, which is extended by a number of behavioral patterns, such as roles, due dates, addresses, classifications and accounts.

Context

Relationships between business partners are at the core of business. Various business software solutions implement functionality that focuses on various aspects of relationships between business partners, such as customers and vendors, employers and employees, service providers and service receivers. Almost all of these relationships are in some way intended to, or directly related to exchanges of economic resources, such as the purchase and sale of products and services, corresponding payments. Some relationships specify commitments and constraints for the exchanges.

Some examples of such relationships are as follows:

- *Purchase order*: a commitment for the vendor to deliver goods and obligation for the customer to pay for it. In addition, a purchase order specifies payment and delivery terms, such as the delivery date and what happens if the delivery date is not met.

- *Invoice*: a declaration of the claim that the buyer owes a specific amount of money to the seller. In addition, an invoice typically specifies other properties such as payment terms.

- *Employment contract*: specifies details about relationship between employee and employer. In addition, the employment contract specifies other conditions of the employment, such as position and compensation.

- *Shipment*: a movement of materials between business partners, warehouse sites, or between a warehouse site and a business partner.

- *Payment*: a transfer of money from one business partner to another.

I call the abovementioned artifacts *business relationships*. The term *business relationship*, as used in this paper, covers relationships between parties in various scopes and at various levels of abstraction. An example of a general scope relationship is a contract for providing a maintenance service in a given period of time. This contract can result in more specific relationships, such as service orders, and they result in more specific relationships, such as material movements, payments and other business transactions.

I call the business partners participating in a business relationship *parties*. In keeping with the authors of other publications [1], [3], [4], [6], [7], I use the term *party* to mean a business entity that can participate in a business relationship with another party, such as a person, company, legal entity, team, or organizational unit.

The *resource* is a subject of trade. I use the term resource to mean a concrete physical product, asset, inventory, or service that has identity. For example, a product that has a serial number, or a service that has a start time and end time.

Problem

Have you ever tried to describe an object model of a business system and struggled to find the right structure of the model and the right relationships between business entities? Have you ever wanted to know a simple rule for modeling the business or economic system in a consistent manner?

Forces

1. Parties have relationships between each other. We can model these relationships as associations between the parties. However, these relationships often have specific attributes, describing details of this relationship rather than details of one of the parties. Examples of such attributes are delivery due, validity period, payment terms and employment position. Because of these attributes, the relations between parties cannot be modeled as pure associations.

2. Relationships between parties involved in business are manifested in various documents, such as purchase and sales orders, quotes, invoices, payments, and delivery receipts. These documents vary in complexity and it is not possible to determine a complete set of business documents that fit all situations in all businesses. However, you want to capture all these relationships in a uniform way in the object model.

3. If a relationship exists between parties, this relationship may, under certain conditions, create or cause the creation of other relationships between the same parties. For example, a purchase order may result in a delivery of goods. You want to capture this fact in the object model. However, traditional object-oriented modeling techniques do not give any hints for how to describe the fact that one relationship between objects can imply another.

Solution

Encapsulate the relationship between parties in an entity called *business relationship*. The business relationship entity is related to at least two *party* entities: the supplier and the consumer. Examples of parties are customer, vendor, or employee. Each business relationship is related to one or more

resources. Examples of resources are an asset, a physical product, service, work, or another subject of trade.
The conceptual structure of the business relationship pattern is illustrated in Fig.1.

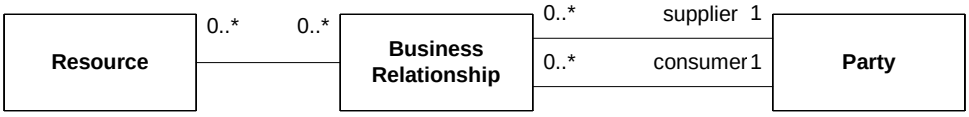


Fig.1. Conceptual structure of the business relationship pattern

Known Uses

Purchase order is a business relationship that specifies a commitment for the vendor to deliver goods and an obligation for the customer to pay for it. Purchase order is related via the reconciliation to the business relationships shipment and payment.

Invoice is a business relationship that declares the claim that the buyer owes a specific amount of money to the seller.

Customer phone call requesting a sales quotation is a business relationship, related to certain resources. It is related via the reconciliation relationship to the quotation.

Resulting Context

If you are starting to look at your enterprise through the perspective of business relationships, then you will need to decide what relationships between parties exist, and model these relationships as entities.

The business relationship pattern allows you to define the fundamental infrastructure of the enterprise model, which can be extended by other patterns. See the section Related Patterns for details. However, besides this structure, the business relationships pattern does not specify any constraints and modeling rules. The system compliant with economic rules is described in the REA, COMMITMENT and CLAIM patterns.

Related Patterns

The pattern BUSINESS TRANSACTIONS specifies how to record history of business relationships. The patterns REA, COMMITMENT and CLAIM extend the BUSINESS RELATIONSHIPS pattern with general economic rules valid for every economic system. These patterns determine the structure and skeleton of the enterprise model.

The rest of the patterns extend the structural patterns with specific functionality and features. Currently, in this group are DUE DATES, ROLES, ACCOUNTS, CLASSIFICATIONS, and ADDRESSES. This is not a final list, and the pattern language can (and will) be extended by adding more patterns to this group. The pattern map is illustrated in Fig. 2.

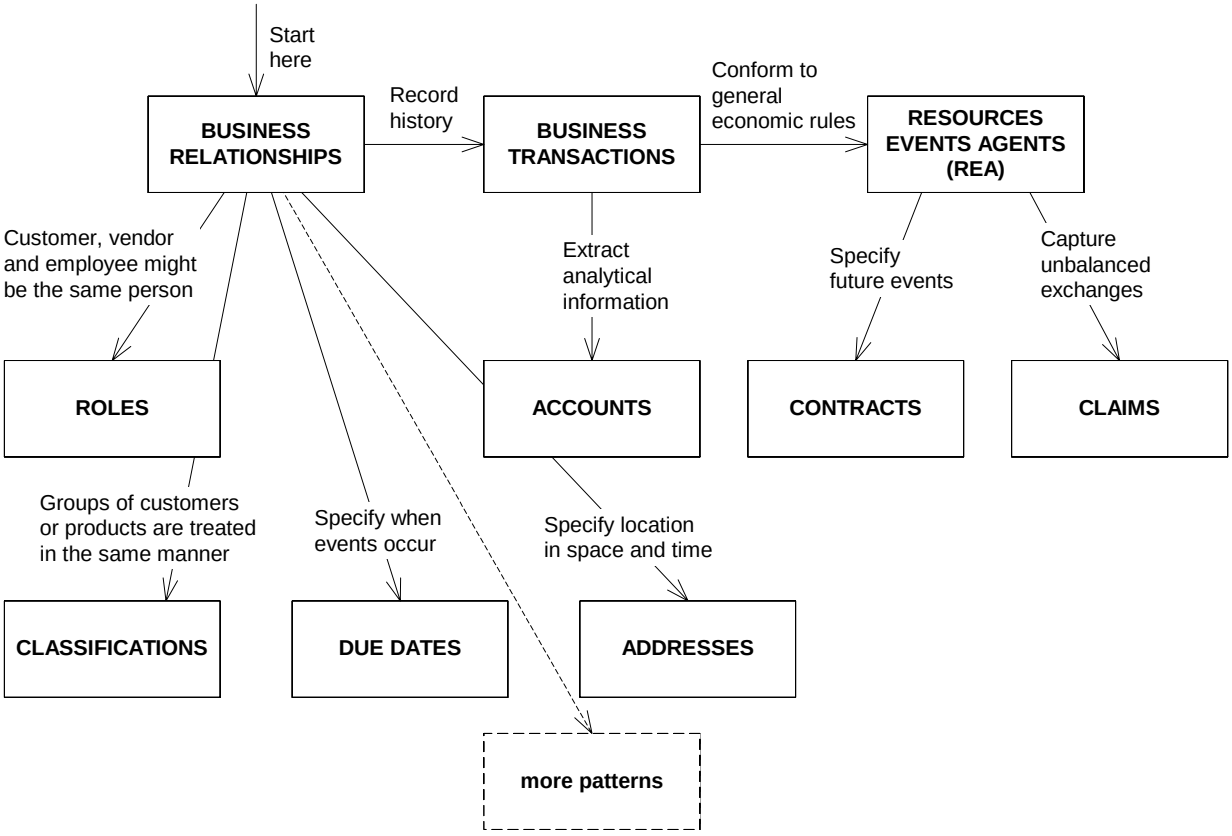


Fig. 2. Pattern Map

Credits and Sources

1. Christian Vibe Scheller of Denmark, personal communication, 1998 – 2000, Lars Hammer, Jesper Kiehn, Henning Kjersgaard Nielsen, Erik B. Jakobsen of Microsoft Business Solutions, personal discussion, 2000 – 2002.

2. David Hay in [6] describes the business relationships Purchase and Sale Orders, Lease, Permit and Employment Contracts. David Hay uses the terms *contract* for business relationships, *line item* for business relationship line and *asset* for the resource. The business relationship pattern, as described in this paper, is more general than David Hay's contact. It also covers material movements, invoices, etc.

3. Peter Coad in [1] uses this pattern to model various business relationships, including Request for Quote, Purchase and Sales orders, Delivery, Invoice, Service, Production Requests, Manufacturing process. The complete list is available at http://www.oi.com/services/publications/jm_book.zip, Peter Coad uses the terms *Moment-Interval* for Business relationship, *Description* for resource and *Party Role* for party. The universal enterprise model pattern language uses this pattern as a skeleton to bind together additional business patterns.

4. Customer Contact Pattern by Dirk Riehle is a specific instance of this pattern, in which one of the parties is the customer. For details, see http://www.riehle.org.

Business Transactions

Context

Registering and analyzing history of the business relationships between business partners is an important part of the functionality of most business software solutions. The history of business relationships is typically related to realized or intended exchanges of economic resources, such as purchase and sale of products and services, invoices and corresponding payments.

Problem

How do we keep track of the history of business relationships?

Forces

1. You want a system that records all relevant information about relationships between business partners.

2. You want to model the fact that an event that affects the business relationship might imply other events to occur. For example, shipment of goods implies payment to occur.

3. If user of the system made an error when making a record, there are often legal requirements for correction of the error. The original (erroneous) information should often not just be deleted or overwritten, but instead a new record that eliminates the effect of the error should be made.

Solution

Encapsulate the event that changed the relationship between parties in an entity called *business transaction*. The business transaction is related to two *party* entities: the supplier and the consumer. Each business transaction is related to one or more *resources*. Business transaction might have additional relationships to other entities, such as to the category of party or category of resource (see the CLASSIFICATIONS pattern for details).

The operation *commit()* persists the business transaction and makes is immutable, that is, no changes in the properties and relationships are allowed after this operation has successfully finished. If the business transaction contained erroneous information, the only way to undo the effect of this transaction is to create and commit another transaction that eliminates the effect of the error. See reference [5] for discussion on adjusting transactions.

The attribute *when occurred* denotes a time interval or point in time when the transaction occurred, and *when committed* denotes the time when the transaction was registered.

The *reconciliation* relationship represents a cause-and-effect association between the business transactions. For example, a quotation might be followed by an order, shipment by payment, or payment by shipment. Please note that *initiator* and *terminator* of the reconciliation do not imply any time order. That is, the terminator business relationship can take effect before the initiator business relationship and vice versa.

The structure of the business transaction pattern is illustrated in Fig.3.

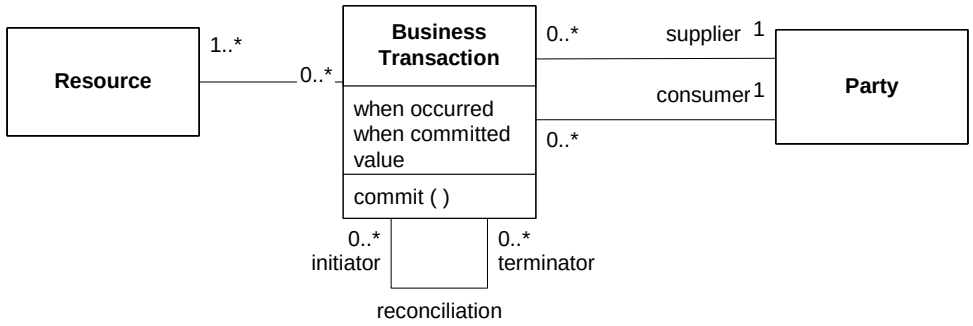


Fig.3. Structure of the business transaction pattern

An example of the business transaction pattern is illustrated in Fig. 4. The figure illustrates two business transactions: invoice and payment. The parties are Customer C and Vendor V. The invoice line specifies four pieces of Product A. The reconciliation relationship links together the payment and the invoice.

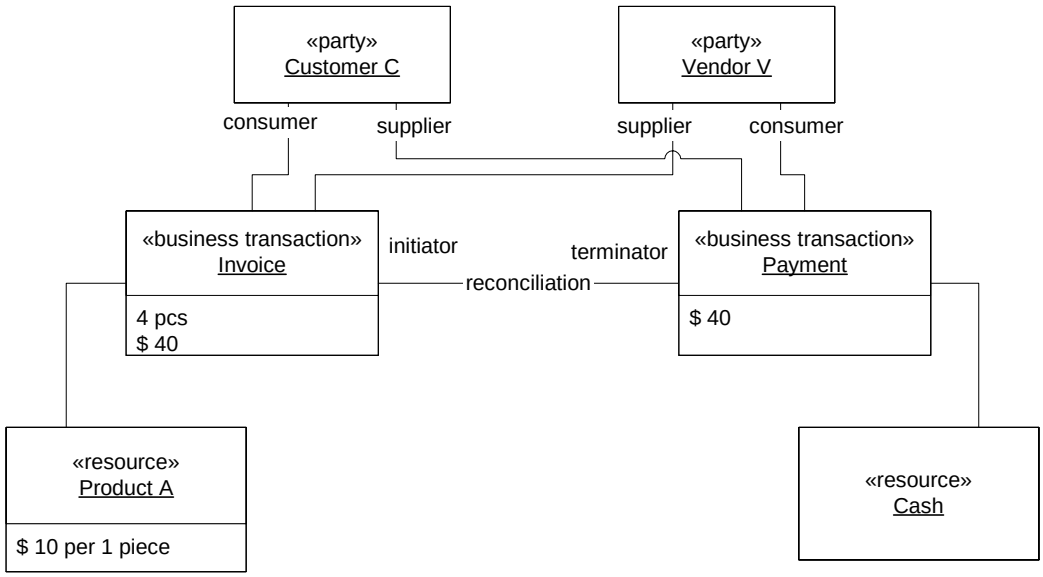


Fig. 4. Example of business transactions

Known uses

Material movement. The warehouse locations are the parties, and the responsibility for the material is the resource.

Sales order line. Sales order line is related (via sales order header) to the customer and seller. Sales order line specifies a product or service, which are the economic resources for both parties.

Rental. The premises is the resource; yielding the right to occupy to the tenant is an outflow of resources. Payment of rent is an inflow of resources. Rental is an example of business transaction that occurs over an interval of time.

Deposits, withdrawals of cash, and interest payments are the transactions in a banking system. Bank and customer are the parties, and responsibility for cash, and interest payments are the resources.

Resulting Context

The BUSINESS TRANSACTIONS is a special case of BUSINESS RELATIONSHIPS. It covers those business relationships, for which all changes are recorded, especially the exchanges of economic resources between business partners, and commitments for such exchanges.

Some business transactions last over a period of time, and recording these changes is not always natural. Some changes, like usage of the equipment or shrinkage of the inventory, happen continuously, and not at discrete points in time. Nevertheless, the accounting practice has standard way of solving these issues.

Please note that a business transaction does not need to have a navigable relationship to financial accounts. See the ACCOUNTS pattern, and the reference [6] for more information. However, many traditional accounting systems specify an account already when the transaction is committed. See the reference [5] for more information on this approach. Although this is commonly used in the accounting practice, this solution has its limits for very large business information systems. Please see the reference [8] for discussion on scalability.

Related Patterns

The patterns REA, COMMITMENTS and CLAIMS extend the BUSINESS TRANSACTIONS pattern with general economic rules valid for every economic system. The ACCOUNTS pattern shows how to extract analytical data from the recorded transactions.

Credits and Sources

1. Business transactions are part of the Transactions and Accounts pattern language, available at http://c2.com/cgi-bin/wiki?TransactionsAndAccounts

2. Martin Fowler' paper Accounting Patterns has a concept of event that is close to business transaction. The paper is available at http://martinfowler.com/apsupp/accounting.pdf.

The REA (Resources Events Agents) Pattern

Context

Business is based on exchanges of economic resources. For example, a customer at a shop buys a product. The product is the resource and the sale is the outflow of resources. The payment for the product is an inflow or resources and the cash is the resource. If the deal is fair, both partners agree that the value of the exchanged resources is the same. That is, the agreed value of the product is the same as the amount of cash received in return.

For business or legal reasons, it is important to keep track of what resources have been exchanged between business partners and when the exchange of resources occurred.

Some exchanges occur over an interval of time, for example, providing or receiving services, or labor work. Even the sale in a shop might have a non-zero duration, if we want to keep track of various stages of the selling process. It is not that important to distinguish whether exchange occurred instantaneously or during an interval of time. It depends on the point of view. From this pattern perspective it is important that the exchange occurred and that the resource changed ownership.

Problem

How do we model exchanges of economic resources?

Forces

1. You want to model the rules that apply to *all* economic systems. However, most analysis patterns and data models are often domain dependent, because they reflect experience of specific software consultants.

2. You want to model things that are shareable and reusable across various application domains, and where details of specific applications are ignored. You could build your object model from user requirements, but it would be very difficult to find the right abstractions valid for *all* economic systems.

3. You want to build an extendable system, which can be extended by new concepts without changes to the foundations of the system. This is difficult, because you must sort out or generalize the domain specific knowledge. Otherwise, your system would result in changes, instead of extensions.

4. You want well-defined modeling rules that enable you to ensure that your model is consistent. For example, how do you keep track of a party who gives resources away, and eventually receives other resources in return? Other examples of consistency questions include: if company sells a product, how does the company obtain it? Or what does a company get in return for providing a customer with certain benefits?

Solution

Consider the economic activities as a sequence of exchanges of resources – the process of giving up some resources to obtain others. The following entities model the exchanges of resources.

Economic event represents the interval in time, or a moment in time when economic exchange occurs. Further more, economic event keeps track of the *value* of exchanged resources. The *time interval* specifies a moment or interval in time when the exchange occurred. Some exchanges occur instantaneously, such as sales of goods; some occur over interval of time, such as rentals or services. Note that the change of ownership (the economic event) sometimes occurs at a different time to that of the physical movement of goods.

The economic event represents either *inflow* or *outflow*. Inflow is an event when a party receives ownership of resources, for example, incoming payment. Outflow is an event when a party yields ownership of resources, for example, a shipment of goods.

Resource represents the subject of trade. Resource is something of value that is under the control of the party.

Party represents an economic unit, or legal entity capable of exchanging economic resources with other parties.

Duality is a special kind of reconciliation. It is a relationship between inflow economic events and outflow economic events. When the deal is closed and fulfilled, the values of inflow and outflow economic events are the same. It is *duality's* responsibility to keep track of the balance between the outflow and inflow. Duality is a many-to-many relationship. For example, several sales can be paid by one check, and one sale can be paid by several installments.

The model is illustrated in Fig. 5.

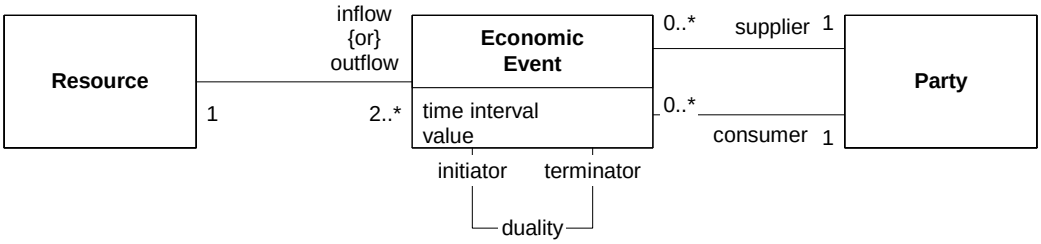


Fig. 5. The economic event pattern as a special case of business transaction

The following three rules apply for the model. They can be used to ensure consistency of a specific instantiation of this pattern.

1. At least one inflow and one outflow economic event exist for each resource. Conversely, each inflow and outflow economic event must be related to a resource.
For example, goods related to the sales event must also be related to the purchase or production event (or some other event specifying how those goods are going to be obtained).

2. Each outflow economic event must have a duality relationship to an inflow economic event, and vice versa. For example, shipment to a customer (outflow) must be related to a customer payment (inflow).

3. Each economic event must have a relationship to two parties participating in the exchange. For example, each economic event must be related to the customer and vendor, employer and employee, and so on.

An example of a simple system is illustrated in Fig. 6. In this example, a wholesaler buys products from vendors and sells them onto customers. The *Customer*, *Vendor* and *Wholesaler* are the parties; the *Product* and *Cash* are the resources; *Goods Receipt*, *Payment*, *Shipment* and *Cash Receipt* are the economic events.

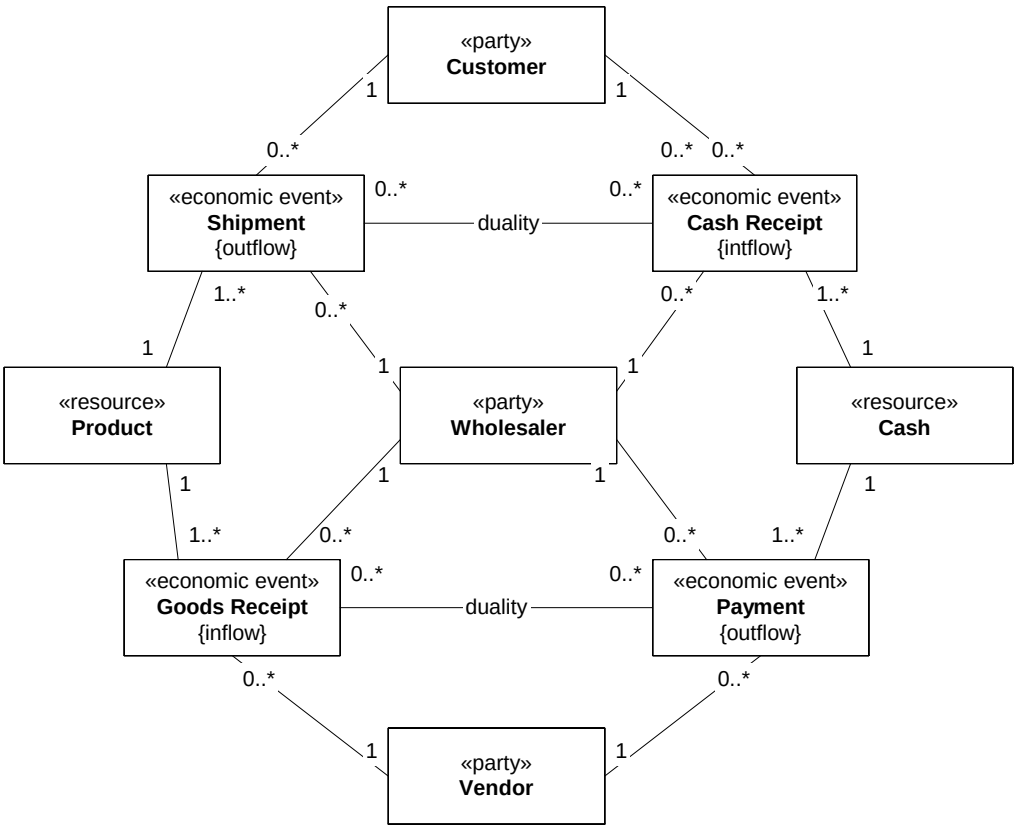


Fig. 6. Economic events for a wholesale system

Fig. 7 illustrates and example, in which the upper part of the example from Fig. 6 is enhanced by two economic events, Return of Goods and Cash Refund. The duality here is a ternary relationship between these four economic events. The duality ensures that when the deal is closed, the sum of values of the initiator events is equal to the sum of the terminator events.

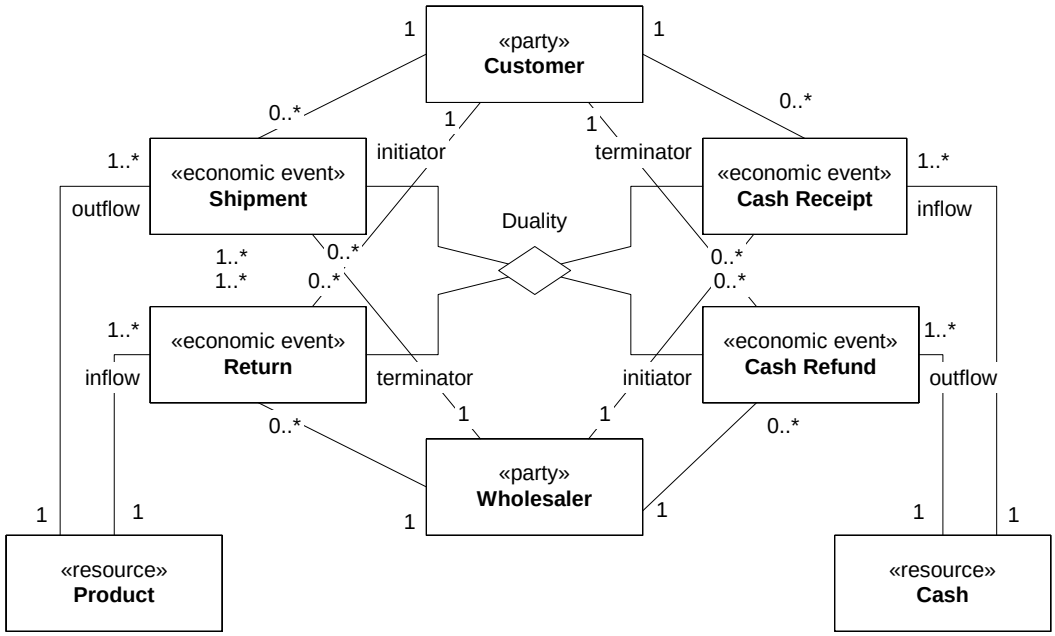


Fig. 7. Return of goods

Known uses

Cash sale. Product and cash are the resources; delivery and payment are the economic events.

Rental. The premises is the resource; yielding the right to occupy to the tenant is an outflow of resources. Payment of rent is an inflow of resources. Rental is an example of economic event that occurs over an interval of time.

Employment. From employer's perspective, an employee's time is the inflow of resources, salary is the outflow. Employment is another example when an economic event occurs over an interval of time.

Resulting Context

The economic event is a special case of business transaction, see BUSINESS TRANSACTIONS pattern. It covers those business transactions that change ownership of resources. Moreover, this pattern defines rules valid for *all* economic systems. This pattern forces developer to make a *complete* model of the economic exchanges, and think about non-obvious questions like "what do I get in return for paying my taxes?" While this is often useful, sometimes the overall burden from increased complexity overweighs the benefits of getting a complete and economically correct model.

As the economic event may occur over interval of time, sometimes it is useful to model interactions between business partners within the scope of one economic event. The LIFECYCLES[1] pattern addresses this issue.

Related Patterns

The COMMITMENTS pattern is a special kind of BUSINESS TRANSACTIONS that specifies the economic events scheduled to occur in the future.
The CLAIMS pattern is a special kind of BUSINESS TRANSACTIONS that deals with unbalanced exchanges of resources.

The ADDRESSES pattern specifies where the exchange occurred. The ACCOUNTS pattern specifies how to create a report that aggregates values across sets of economic events. The CLASSIFICATIONS pattern specifies how to handle various categories of economic events.
The DUE DATES pattern specifies the time interval in the case the economic event occurs over interval of time.

Credits and Sources

1. Bob Haugen, e-mail discussion
2. Guido Geerts, personal communication
3. William E. McCarthy and Jesper Kiehn, teleconference
4. Henning Kjersaard Nielsen and Erik B. Jakobsen, personal discussion
5. McCarthy, W.E. 1982. “The REA Accounting Model: A Generalized Framework for Accounting Systems in A Shared Data Environment.” The Accounting Review (July), pp. 554-578.

[1] The LIFECYCLES pattern is not part of this pattern language, yet.

Contracts

Context

Some relationships between business partners specify promises of future exchanges of resources. For example, a purchase order is a promise to receive goods from a vendor, and pay for the goods. An employment contract is a promise that an employee will give his time to the employer and a promise by the employer to provide compensation in return.

Problem

How to model promises of future exchanges of economic resources?

Forces

1. Most economic events do not occur unexpectedly. They have been agreed between business partners beforehand. You would like to have a mechanism specifying details about the commitments of economic events.

2. If a party commits itself to give resources away (to pay for a product or accept a purchase order), the party would like to be informed about whether it will actually have the resources available at the time specified by the commitment.

Solution

Consider the economic contracts between business partners as a collection of commitments to trigger economic events in a well-defined future. The following entities model the contracts and commitments.

Commitment entity specifies an economic event or events to occur in the future. *Commitment* has a relation to *Resource*, and either reserves the resource for future stock outflow, or expects a resource for future stock inflow.

Fullfillment (a special kind of reconciliation) is a one-to-many relationship between the commitment and the economic event. One commitment can be fulfilled by several economic events.

Economic contract is a set of commitments, which specify the future inflow or outflow of resources. At least one commitment in this set should be related to the inflow economic event, and at least one to the outflow economic event.

Reciprocity (a special kind of reconciliation) is a many-to-many relationship between commitments. Reciprocity guarantees that the total value of expectation claims is equal to the total value of the reservation claims.

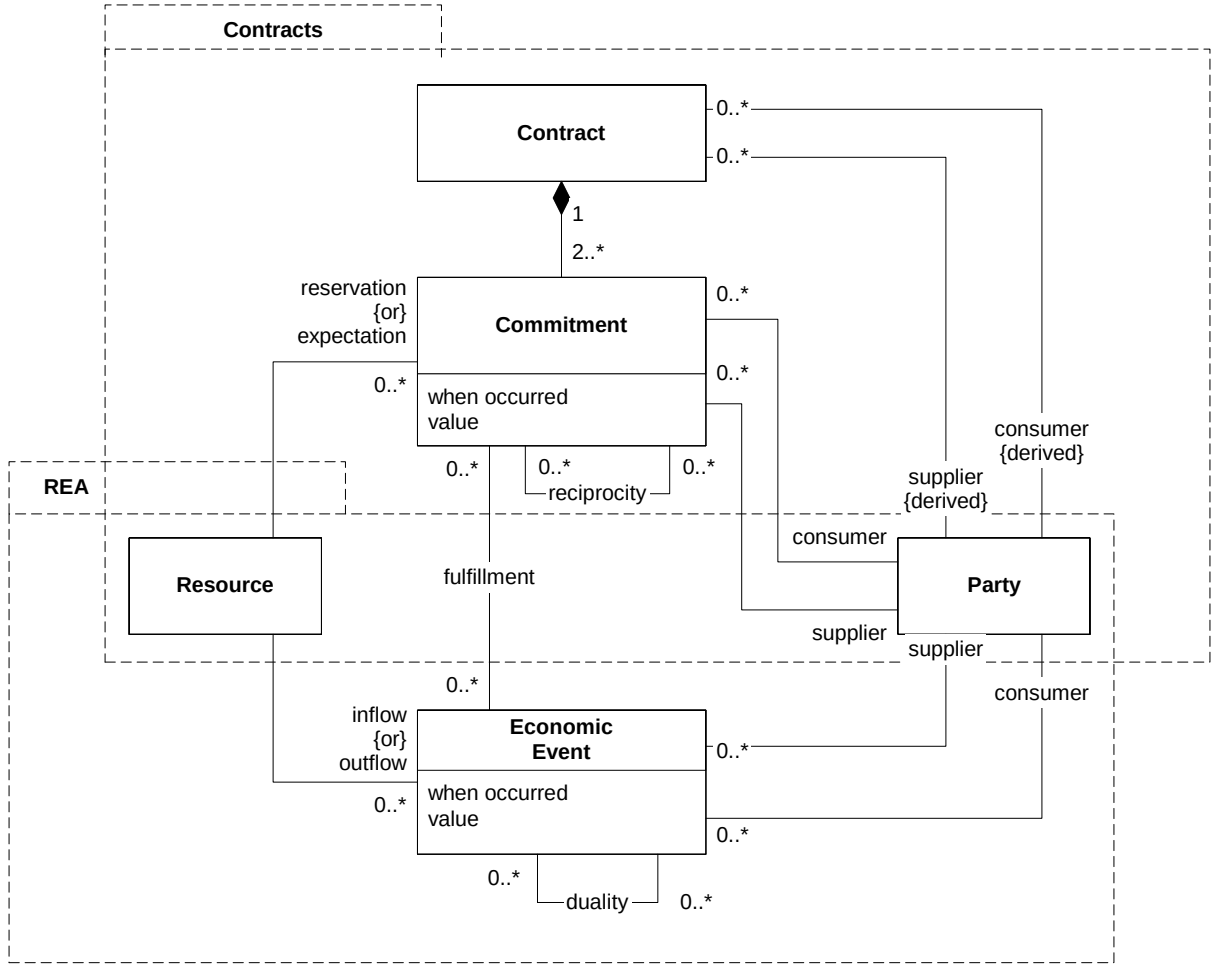


Fig. 8. The commitments pattern as a special case of business transaction

The following rules apply for the model. They can be used to ensure consistency of a specific instantiation of this pattern.

1. Each commitment must be related to a resource. For example, a sales order line must specify the goods to be sold.

2. Each commitment must have two relationships to parties that agree on the future exchange of resources. The customer and vendor, the employer and employee are the examples of parties related in contractual relationship.

3. Each commitment must have the fulfillment relation to at least one economic event.
For example, the purchase order line specifying the goods must be related to the shipment of these goods.

Fig. 9 illustrates an example of a sales order. Sales order is a contract composed of two commitments. The reservation commitment is to sell 4 pieces of product A, of a value of $ 40. The expectation commitment is to receive $ 40 compensation in cash.

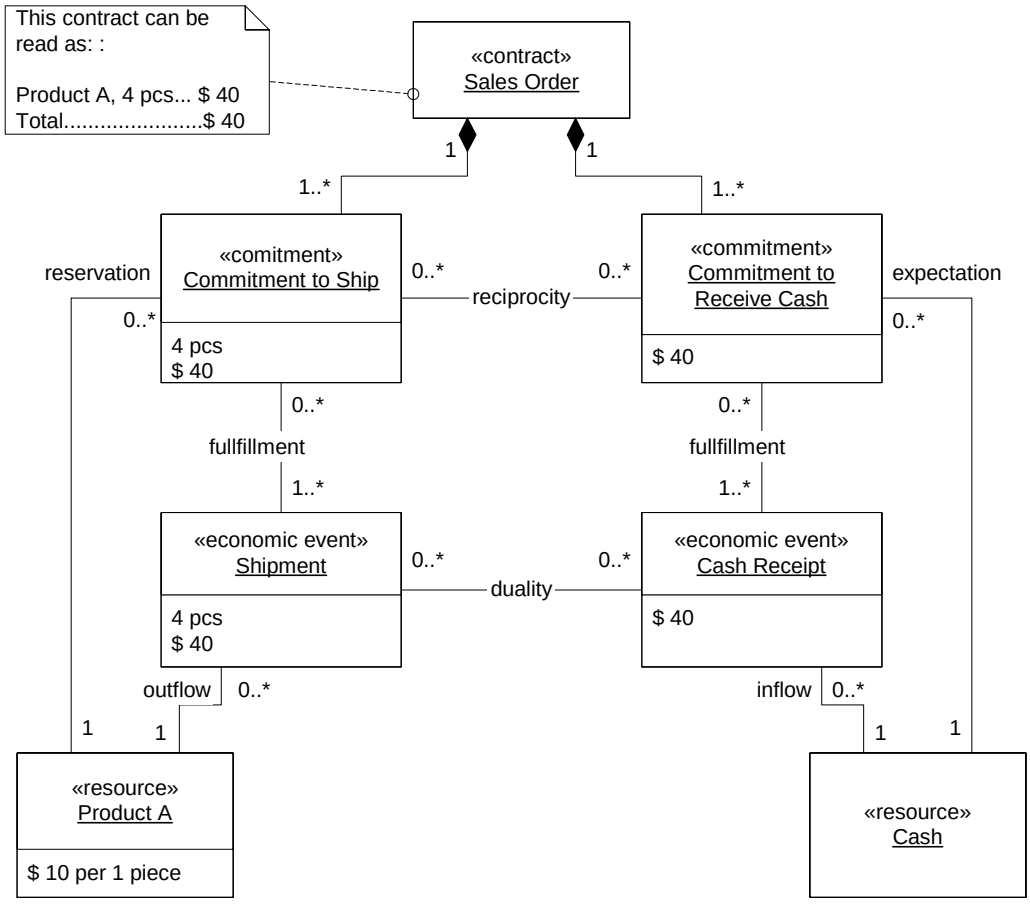


Fig. 9. Sales order contract with commitments and economic events

Known uses

Purchase order is a contract that consists of two commitments: a commitment to receive the ordered goods and a commitment to pay for the goods.

Employment contract represents a commitment by the employee to give his or her time to the employer, and a commitment by the employer give compensation in kind. Employment contract is an example of a contract when both parties agree on a stream of outflow economic events and a stream of inflow economic events.

Resulting Context

The commitment is a special case of business transaction, see BUSINESS TRANSACTIONS pattern. It covers those business transactions that are commitments, promises for changes in ownership of resources. Moreover, the contract pattern defines rules valid for *all* economic systems.

Prognosis of future events, for example budgeting, is not covered by this pattern.

Sometimes, this pattern forces developer to include to the model resources that are difficult to quantify and feel redundant. For example, the commitment to pay taxes to the government are reservation of resources, however, resources expected to be received in return from the government are difficult or even impossible to quantify precisely. The solution to this problem is called IMPLEMENTATION COMPROMISE: developers develop an ideal model of the enterprise that ensures that they have not forgot anything. Then, some contracts, resources or parties are omitted from the model. The resulting model is sufficient for the purpose of the software solution, and simpler than the complete model.

Related Patterns

The ADDRESSES pattern specifies at what place the commitment occurred and where the exchange of resources is supposed to occur.
The ACCOUNTS pattern specifies how to represent unbalances between commitments and the corresponding economic events.
The CLASSIFICATIONS pattern specifies how to handle various categories of commitments.
The DUE DATES pattern specifies when the economic event is scheduled to occur.

Credits and Sources

1. Geerts, G.; McCarthy, W.: The Ontological Foundation of REA Enterprise Information Systems, November 1999, March 2000, and August 2000.

2. Guido Geerts, The University of Delaware, Newark; personal communication, April 2002

Claims

Context

When a vendor ships goods, it usually does not receive customer payments at the very same moment. Outflow and inflow economic events usually do not occur simultaneously, and the duality relationship between the economic events is out of balance for certain period of time. In this case, a common practice is to send an invoice, a requirement to the business partner to settle the owed amount.

Problem

How can we model unbalanced economic exchanges?

Forces

1. Exchange of future economic resources must be fair, that is, agreed by both parties. How can we keep track of the fact that a party gives resources away and will eventually receive some resources in return?

2. Both business partners might not automatically know the exact unbalanced value between the inflow and outflow events. For example, a service contract might specify payments according to consumption. Or a vendor sometimes adds shipping fee to the price of products, whose value might not be known to the purchaser. You want to assure that both parties agree upon the unbalanced value.

3. Legal reasons might require a document specifying the unbalanced value. For example, VAT (value added tax) in Denmark is calculated as a percentage from the invoiced amount.

4. You want your model to be consistent from economic viewpoint. You could model the claim as a basic business relationship following the Fig.1, but you want your model give answers to questions like: if a customer receives a discount for early payment, what benefits does the vendor get in return?

Solution

Encapsulate the unbalanced value in the *duality*. The *claim* entity sums the value over one or more *dualities*. For example, when a vendor ships goods and the customer has not paid yet, the vendor can *materialize* a claim, that is, create an invoice. Each invoice line is related via the *materialization* relationship to the shipment events that have not been fully paid, yet. When the customer eventually pays for the goods, each invoice line will be related via *settlement* relationship to the payment. The *claim* entity is a placeholder for additional attributes such as DUE DATE.
This pattern is illustrated in Fig. 10.

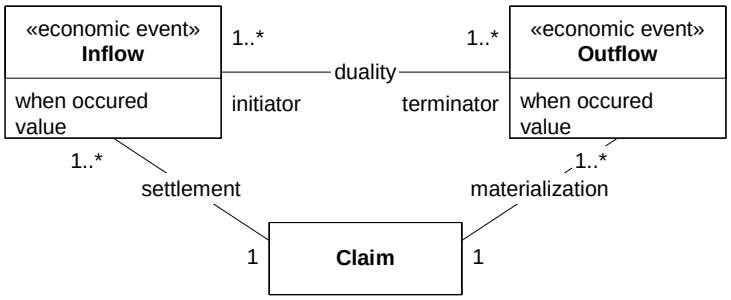


Fig. 10. The claims pattern

Related Patterns

The CLAIMS extends the REA pattern, as CLAIMS is applicable together with the duality relationship. Claim is usually contains due date, see the DUE DATES pattern.

Known uses

Invoice is a claim: pays us the money for the goods or services we provided to you.

Library's late notice that is sent out to you is a claim: bring back those books you owe us!

Credits and Sources

1. Bob Haugen, Logistical Software, communication via e-mail, November 2001- April 2002.

2. Guido Geerts, The University of Delaware, Newark; personal communication, April 2002

3. Daniel May, The University of Southern Denmark, personal communication, summer 2002.

Roles

Context

Sometimes, an employee of the company can also be a customer. For example, a bank clerk opens a bank account, or a nurse at the hospital becomes a patient. A vendor can become a customer, if he buys the company's product, and employees can become vendors, if the employer refunds them their purchases.

Similarly, a specific product can be considered as a raw material or finished goods. Or, the same working hour can be seen as an employee's time, or a consultancy hour sold to the customer. Although the resource is the same, when related to different business relationships, some of its properties will differ, for example, the unit price.

Problem

How can we model situations in which the same physical entity participates in different types of business relationships?

Forces

1. You used the business relationship pattern and your data model contains various kinds of business relationships, such as purchase orders, sales orders and employment contracts. A possible solution could be to replace all these kinds of business relationships by a "universal" business relationship entity, but this would not adequately capture the problem domain. For example, it is not very useful to have a single data entity representing both the sales order and the employment contract, because they have significantly different attributes.

2. The same party can be related to different business relationships, for example, to purchase and sales order. However, the party has different properties when participating in different business relationships. For example, the default ship-to-address is relevant on a party related to the sales order, but not on the party related to the employment contract.

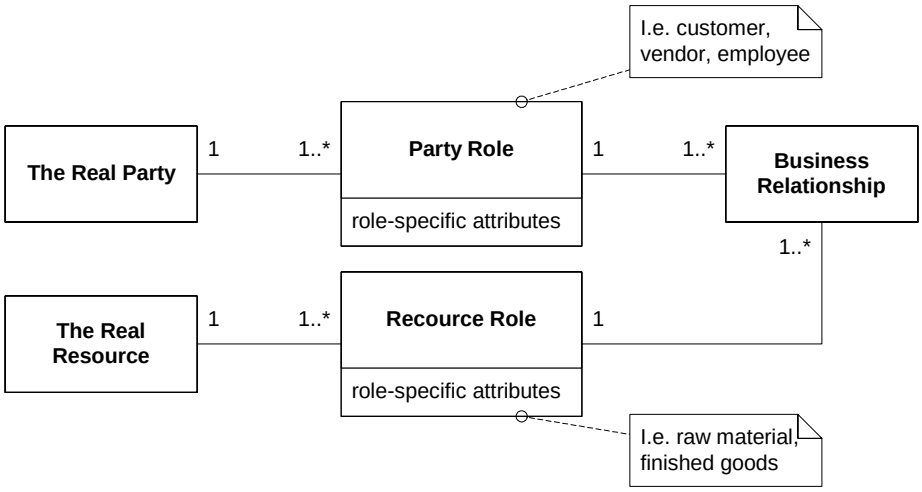
3. As a result of the second force, you could model customers, vendors and employees as parties in your model. However, you want to capture the information that all these three parties can be, in the real world, the same physical entity.

Solution

Split the party entity into *the real party* and the set of *party roles*. Split the resource entity into *the real resource* and the set of *resource roles*. The solution is illustrated in Fig. 11.

The real party and the *real resource* represent the physical objects participating in the business relationship.

The *party role* and the *resource role* are a way that *the real party* and *the real resource* can participate in a business relationship. Each real party and real resource have at least one role.



.
Fig. 11. The role pattern

Some might argue that the real party and resource have attributes common to all their roles. However, conceptually more correct would be to consider the common *role* attributes in the implementation of the *role* entity, for example, moving common attributes to the role superclass. Various implementation approaches to roles are discussed in [5].

Known uses

1. Specific product can have the roles of a material used in work, or finished goods sold to the customer. These two roles have different attributes: the product as material might be used together with mounting fittings. The same product as a goods sold is used without the fittings. Moreover, the unit price for the same product is different if used as a material or if sold directly. Source: Country Union of Danish Electricians (ELFO).

2. A person can play a role of a customer, vendor and employee.

Resulting Context

The roles pattern adds flexibility and extensibility to the model and prevents from duplicated information. However, this is not always the desired behavior. For example, in a hospital information system a user requirement was to physically separate the patient data and the employee data, even if they represent the same person (source: Ralph Johnson, personal communication).

Credits and Sources

1. Martin Fowler's paper "Dealing with roles" focuses on implementation issues of the roles. For details see http://martinfowler.com/apsupp/roles.pdf

2. Dirk Riehle's paper Role Object describes a generalization of the role concept for any component (a particular key abstraction). For details see www.riehle.org.

Accounts

Also known as balances

Context

The BUSINESS TRANSACTIONS pattern describes how to keep track of and registers business relationships, economic events, and commitments. However, merely keeping track of these entities is usually not the main interest of an enterprise's decision makers. Decision makers are mostly interested in *reports*, the aggregated data that summarize registered entities, and provide the information about the state of the enterprise.

Problem

How can we model the aggregated values across sets of business transactions?

Forces

1. The business transaction pattern allows for tracking of business transactions and their values. However, users of enterprise planning systems would like to get information about aggregated values from the sets of business transactions, economic events and commitments.

2. You could manually write an algorithm that aggregates the values from business transactions, for example, in the form of SQL statements. However, you would like to know a rule, valid for *all* economic systems, that would automatically provide for aggregated values across sets of business transactions, events, and commitments available in the system.

Solution

If a party or resource is related to at least two business transaction entities, then the party or resource has a property called *account*. The *account* summarizes the values of all related instances of one business transaction and subtracts them from a sum of the values of all related instances of the other business transaction. The solution is illustrated in Fig. 12.

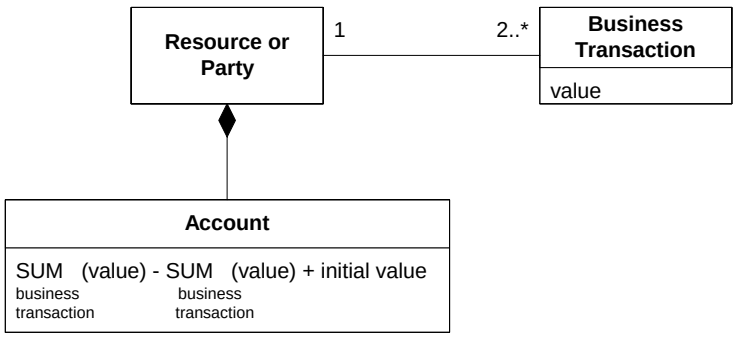


Fig. 12. The accounts pattern

The following figure shows an example of the system with four business transactions: the purchase order, receipt, sales order and shipment. The resource *goods* has three accounts. The account *goods*

on stock is a sum of all goods receipts minus all shipments. The account *goods on order* (goods to be received) is a sum of all purchase order lines minus sum of all the receipts related to purchase order lines. The account *goods to ship* is a sum of all sales order lines minus the sum of all the shipments related to sales order lines.

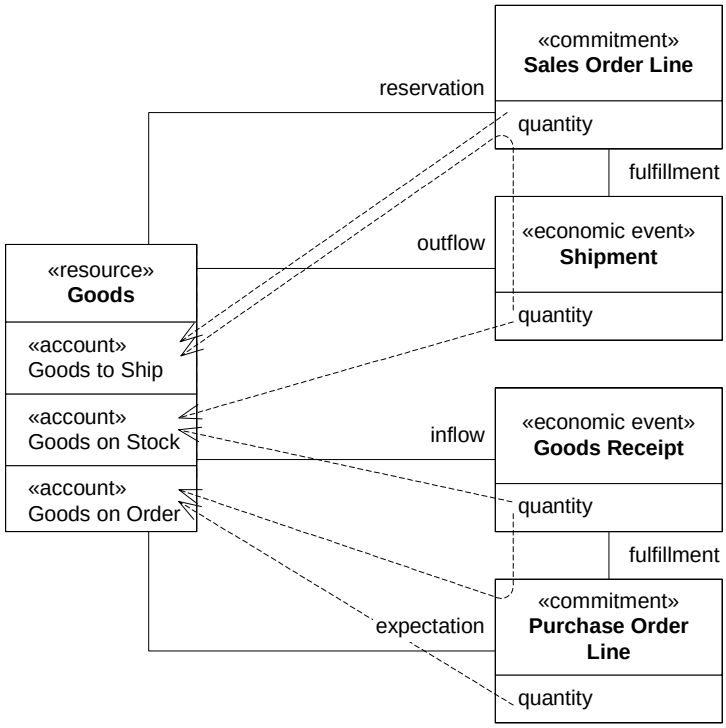


Fig. 13. Examples of accounts on resource

Please note this solution does not describe *all* accounts that exist in the accounting practice.

Known uses

Accounts receivable. For example, the customer balance is an account. It is a sum of all sales orders minus sum of all customer payments.

Accounts payable. For example, vendor balance is an account. It is a sum of all purchase orders minus the sum of all payments to the vendor.

Assets. For example, cash is an account on the company itself. It is a sum of all cash receipts minus sum of all cash disbursements.

Resulting Context

If the business transaction, in addition to its value property, specifies additional information, attributes and properties, the account pattern can be extended to provide selective sums, and aggregate balances for certain values of these additional attributes. For example, if a payment contains due date, see the DUE DATES pattern, the customer account can be refined to reflect all expected payments that are over due and the expected payments that are not over due.

Due Dates

Also known as Deadline

Context

Commitments are promises of the future economic events. Sometimes it is useful to specify when, at what time, the future economic events shall occur. Starting date, last payment date and renewal date are examples of due dates.

Problem

How can we model when should future economic events occur?

Forces

1. Dates and time intervals are attributes of business relationship, business transaction, economic event, commitment and claim. You could add the date-time attribute to all of them, but you want to have a uniform behavior for all these entities in your model.

2. Some future events can be recurrent. Examples of recurrent events are periodic shipments, or meetings that occur every week. There might be complicated rules specifying the recurrence.

3. You want to specify what happens when due date expires.

Solution

Encapsulate the attributes necessary to setup the due date in the *Due Date* entity. The time interval for due date is specified, for example, by *Start date* and *Duration*. The *has expired* attribute specifies whether the due date has passed. Examples of *recurrence rules* are “every day”, “every week”, “first Monday in a month”, etc. Examples of *reminder rule* are “on due date”, “15 minutes before start”, “one day later”, etc. The TYPE OBJECT often extends this pattern by providing for *due date type* that encapsulates a list of various rule types that might be set by the *due date*. The structure of the Due Date pattern is in Fig. 14.

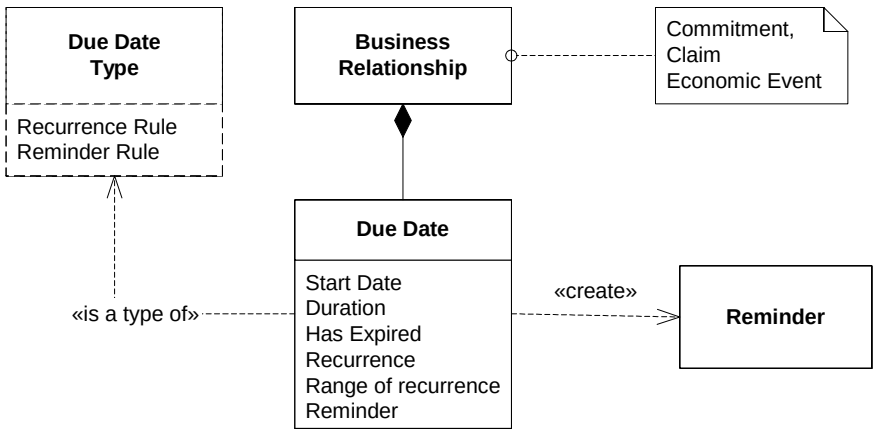


Fig. 14. The due date pattern

Known uses

Registration deadline for a conference is an example of a due date.
Payment schedule, or recurrent meeting in Microsoft Outlook and other personal information managers are examples of a recurrent due date.

Resulting Context

The due date defines the necessary attributes, behavior and rules relevant for setting up the due dates. But what actually happens when due date expires? This information has to be somehow forwarded to the rest of the system or users. The NOTIFICATIONS[2] pattern solves the user notification.

Related Patterns

TYPE OBJECT by Ralph Johnson et al. provides for flexibility by listing applicable recurrence and reminder rules in a *Due date type* entity.
CALENDARS[3] of a party or resource provides an aggregated view over all due dates of all business relationships related to this party or resource.

[2] The NOTIFICATIONS pattern is not part of this pattern language, yet.
[3]The CALENDARS pattern is not part of this pattern language, yet.

Addresses

Context

Parties and resources are usually located in a certain places in the real world. For example, the ship-to address specifies the location of the customer within the scope of the shipment event. However, the physical address is not the only way to contact a party. Phone number, e-mail and URL are examples of communication addresses. These addresses can change in time, for example, a person at the office can be contacted by direct phone number, but during meetings can be contacted by phone via the receptionist. Some communication addresses are public and accessible by any party, whilst others are available only to a restricted set of parties. Resources have communication addresses as well. For example, a product is placed in a certain warehouse location, and its description can be found through its URL.

Problem

How can we contact a party or locate a resource?

Forces

1. You want a uniform mechanism of establishing communication with a party or locating a resource. You can decorate the party or resource object with attributes such as e-mail, url, but this solution does not provide any information regarding when each is address valid.

2. You want to capture the fact that some communication mechanisms are restricted to a certain set of parties and not available to everybody.

3. You want to specify communication addresses for a party, such as ship-to address and bill-to address, but these addresses are often specific to the business relationship, rather than to the party.

Solution

Encapsulate the functionality about how to locate a resource or party in the *address* entity. The *address* is an abstract concept for the *communication address* and the *geographical address*. Examples of communication addresses are the phone number, e-mail and URL. Examples of geographical addresses are the postal address and geographical coordinates. The *business relationship* specifies the validity time interval of the address. The business relationship also specifies what parties the address is available to. The solution is illustrated in Fig. 15.

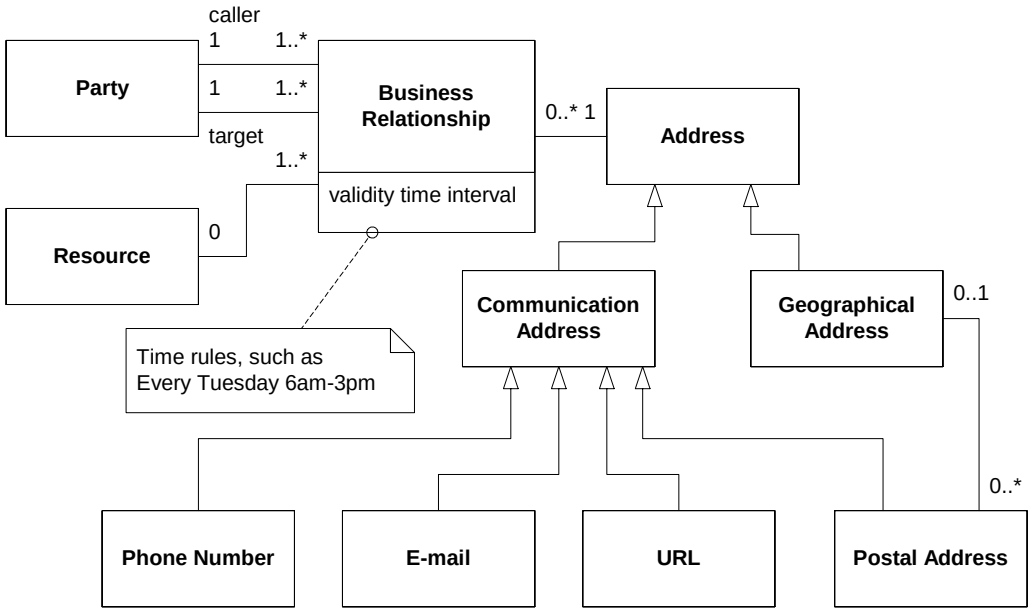


Fig. 15. Addresses

Known uses

A sales order must have a sell-to, bill-to, and ship-to address. These can all be postal addresses, as it may be necessary to send information material such as a confirmation to the customer. However, depending on the company and the customer these addresses could also be electronic addresses.

Within Human Resources department of a company, all addresses of employees must be stored. These addresses must be both postal addresses and phone numbers. The employee's phone numbers and physical locations during working hours and outside working hours are different.

Resulting Context

This pattern specifies that parties and resources are not directly related to their addresses, they are related via the business relationship entity. This is a consequence of keeping track of valid addresses over time, and focusing on the communication aspect of the address. In simple cases, this extra complexity would not pay off the benefits of using this pattern, and the address will be related directly to the party or resource.

Related Patterns

The CLASSIFICATIONS pattern can be used to specify a group of the caller parties. For example, in the postal address, which is public, the caller is usually the "any" party category. For e-mail address or direct phone number, which might be restricted, the caller might be a category "trusted party", which would imply that only parties with the "trusted party" classification might be related to (that is, to know about) this e-mail address or phone number.

Credits and Sources

1. The paper “How Do I Find You? (Let Me Count The Ways.)” by Terry Moriarty and Dwight Seeley describes the main ideas of this pattern. For details see www.inastrol.com

2. The address pattern is a part of IBM SanFrancisco framework, for details, see: http://www.ibm.com/software/ad/sanfrancisco

Classifications

Context

Sometimes, a company partitions its customers into various categories, for example, high-volume customers and small-yield customers. Sometimes, there is a hierarchy of categories. For example, the furniture products can be classified into the categories office and home. The home category has subcategories, such as sofas, chairs, tables, and beds. The sofas category has subcategories leather sofas, cloth sofas and sleeping sofas. Sometimes, the classified object can belong to several categories; for example, a specific sofa can be both sleeping sofa and leather sofa.

Problem

How can we model categories of products or parties in a uniform way?

Forces

1. You want to make a uniform model for, for example, event categories, party categories, product categories and business relationship categories.

2. It should be possible to categorize different business relationships, resources or parties using the same classification hierarchy.

3. When an object changes its category, it still belongs to the same class. That is, it still has the same attributes, properties and methods. It might change the values of the attributes, but, for example, does not get a new attribute when it changes its category. Note: the TYPE OBJECT should be used if this force is not applicable.

4. The entity typically does not change its category when its attributes change. For example, let's suppose that the payment entity belongs to different priority categories. If due date of the payment is over, the payment entity still belongs to its original priority category. Note: use the LIFECYCLES pattern if this force is not applicable.

Solution

The business relationship, economic event, commitment, contract, claim, resource and party can be related to the *category* entity. The category entity can belong to a hierarchy of category entities, that is, a category is related to a *supercategory* and *subcategories*. In general, the relationships between them are many-to-many. The business relationship, resource or party can belong to several categories. The category can belong to several supercategories and might have several subcategories. Solution is illustrated in Fig. 16.

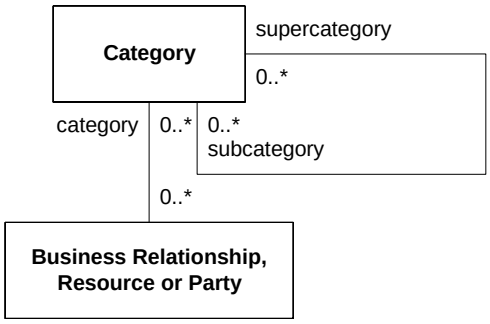


Fig. 16. The classifications pattern

Known uses

Subject is the classification of information resources into categories such as Business & Economy, Computers & Internet, Entertainment, Recreation & Sports, etc.

Skill (qualification) is a classification that assigns employee to certain skill groups. Skill is an example of a classification in which party can be assigned to more than one category.

VAT (value added tax) group is a classification of the product into the VAT groups.

Resulting Context

The business relationship, resource or party can be assigned into several categories simultaneously. The consequence of using this pattern is that the classified object does not change when its category changes. For example, if a product changes its VAT group, it is still the same product. On the contrary, we might consider a “party” object, categorize it into a “customer” or “employee” categories. Application of this pattern would imply that the attributes and methods are the same both for “customer” and “employee”. If it is not the case, the TYPE OBJECT pattern should be applied instead of classification in this particular situation.

Related Patterns

The LIFECYCLES can be used to categorize the objects as well. For example, imagine that we create the following categories for the order: quotation, accepted, shipped, paid. We could use the classification pattern. However, it would be impossible to specify the rules like: the quotation is first accepted, then shipped and then paid; or the rules like: If the order is in the category “paid”, it is not allowed to change the category to “quotation”. In this case, the LIFECYCLES should be used, instead of CLASSIFICATIONS.

TYPE OBJECT can be used to categorize the objects as well. For example, imagine that we create the following categories for the sales order: quotation, accepted order, shipment, and payment. If the sales order in these four categories has different attributes and behavior, for example, the shipment specifies the products but not prices, and payment specifies the prices but not products, then TYPE OBJECT should be used. That is, the quotation, accepted order, shipment, payment should be different types, and not categories of the same type.

Credits and Sources

The classification pattern is a part of IBM SanFrancisco framework, for details, see: http://www.ibm.com/software/ad/sanfrancisco
.

Acknowledgements for the whole Pattern Language

I would like to thank to Daniel May of Maersk Mc-Kinley Institute, University of Southern Denmark, Odense, Denmark, for shepherding this paper, and for his useful suggestions and comments. I do, of course, take full responsibility for any omission or errors.

I would like to thank to Jesper Kiehn, Microsoft Business Solutions, Copenhagen, Denmark, for fruitful discussions, valuable comments, observations and ideas.

References

[1] Coad, P., Lefebre, E., DeLuca, J.: Java Modeling in Color with UML, Prentice Hall PTR, 1999.
[2] David, J. S.: Three events that defined an REA methodology for systems analysis, design and implementation
[3] Ericsson, H., Penker, M.: Business Modeling with UML, John Wiley & Sons, 2000.
[4] Fowler, M.: Analysis Patterns, Addison Wesley Longmann, 1997.
[5] Fowler, M.: Analysis patterns (articles on the web), http://martinfowler.com/articles.html#
[6] Guido L. Geerts and William E. McCarthy: The Ontological foundation of REA Enterprise Information Systems, 1999-2000.
[7] Hay, D.: Data Model Patterns, Conventions of Thought, Dorset House Publishing, 1996.
[8] Hollander, A., Denna, E.L., Cherrington, J.O.: Accounting Information Technology and Business Solutions, Irwin McGraw-Hill, 2000.
[9] Riehle, D.: Association object patterns, http://www.riehle.org/practical-matters/patterns/business/association-objects/index.html
[10] Riehle, D.: Role Object. http://www.riehle.org/computer-science-research/1997/plop-1997-role-object.html
[11] Silverston, L., Inmon, W. H., Graziano, K,: The data model resource book, John Wiley & Sons, Inc. 1977.
[12] William E. McCarthy: The REA Accounting model: A generalized framework for accounting systems in a shared data environment. The accounting review (July) pp. 554-578, 1982.

A First Approach To Design Web Sites By Using Patterns

Francisco Montero, María Lozano, Pascual González, Isidro Ramos[τ]

LoUISE Research Group
Escuela Politécnica Superior de Albacete
Universidad de Castilla–La Mancha
02071 – Albacete – Spain
{fmontero, mlozano, pgonzalez}@info-ab.uclm.es

[τ]Dpto. de Sistemas Informáticos y Computación
Universidad Politécnica de Valencia
Camino de Vera s/n
E-46071 Valencia - Spain
iramos@dsic.upv.es

Abstract. This paper presents a first approach of a web design pattern language. Its main goal is to gather the experience on web design and provides a communicative tool than can be used by every stakeholder in a project. The pattern language distinguishes between three design levels: the web site, a web page and the ornamentation level. The recurring principle through the pattern language is supporting users to achieve usability improvement.

Introduction

The World Wide Web has rapidly become the dominant Internet tool, combining hypertext and multimedia to provide a network of multidisciplinary resources. It is important to make sure that every part of a Web site is useful. A user will come to a site expecting to be able to perform a particular task, or read a particular piece of information. When we are designing a Web site we want to make sure that the user can find that resource quickly and easily. If they can't find the information quickly then they may leave our site, and proceed to another site where they can find the resource. The Web is a new medium and requires a new approach [Nielsen 99].

[Shneiderman 98] commented that "*It will take a decade until sufficient experience, experimentation, and hypothesis testing clarify issues*" and warned that meanwhile "the paucity of empirical data to validate or sharpen insight mean that some guidelines are misleading". Nevertheless, many sets of web design guidelines have been published. There are many [Guidelines] that can be used for improving the designing of our Web sites. Most of these recommendations for Web site designers are however not based on research but on intuition. They are based in the experience of the designer. Traditionally, interface design experiences are gathered with guidelines but patterns can be used too. The concept of a pattern language has been developed by Christopher Alexander and his colleagues in architecture and urban design [Alexander 77, 79]. In brief, a pattern language is a network of patterns of varying scales; each pattern is embodied as a concrete prototype, and is related to larger scale patterns, which it supports, and to smaller scale patterns which support it. The goal of a pattern language is to capture patterns in their contexts, and to provide a mechanism for understanding the non-local consequences of design decisions [Erickson 97].

A Pattern Language for web design

A pattern language has to have a structure of hierarchical network. So, an essential component in the definition of a pattern is the relationships with others patterns. In the diagram of the proposed pattern language (see Figure 1), arrows between patterns introduce these relationships. The pattern language distinguishes between three levels, these levels are inspired by Christopher Alexander's pattern. His pattern language describes a highly structured collection of patterns, intended as a practical guide for architectural designers. This idea is extrapolated to web site designing by introducing patterns related with **web sites**, **web-pages** and **ornamentation** details.

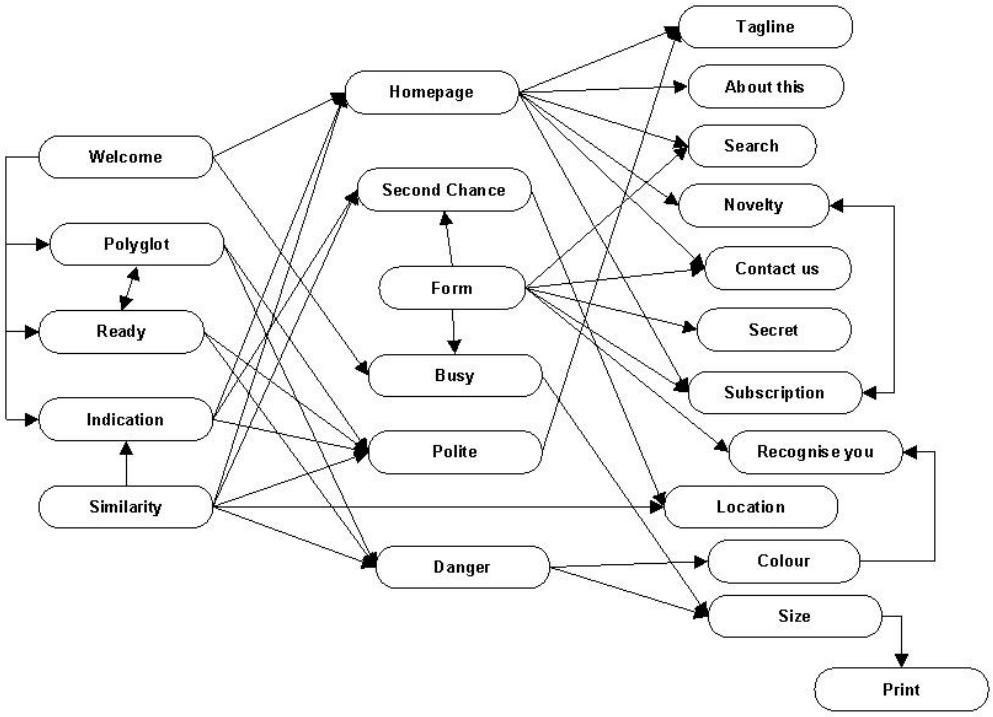


Figure 1. Proposed pattern language to design web sites

In the following sections of this paper, each level starts with a brief summary, which introduces the patterns described in the section. The patterns in the entire collection are depicted graphically in Figure 1 and summarised at the end of this paper in the Summary section.

How could I use this collection?

The algorithm that describes how this pattern language can be used is the following:

1. Read the resumed list of patterns.
2. Scan down the list, and find the pattern, which best describes the overall scope of the project or the problem that you want to solve.
3. Read the starting pattern. Tick all of the low order patterns and ignore all the high order patterns.
4. Turn to each pattern and now tick only relevant low order patterns, if they exist else 6.
5. Keep going like this, until you have ticked all the patterns you want for your project.
6. Adjust the sequence by adding your own material where you haven't found a corresponding pattern.
7. Change any pattern where you have a personal version, which is more relevant.

Web Site Level

This section introduces design patterns related with Web site design. These patterns are associated with common features that can be found on many Web sites and are extrapolated from another different context. The user requires know where he/she is (**Welcome**) and where he/she can go (**Indication**). The user wants to visit the Web site in a suitable way (**Polyglot**, **Ready**, **Similarity**).

Welcome

Motivation:
When a user arrives at a Web site, like he/she arrives at a city, town or any important building needs to know where he/she is, what can he do there, and what he need for visiting that place.

Problem:
How does the user know where he/she is?

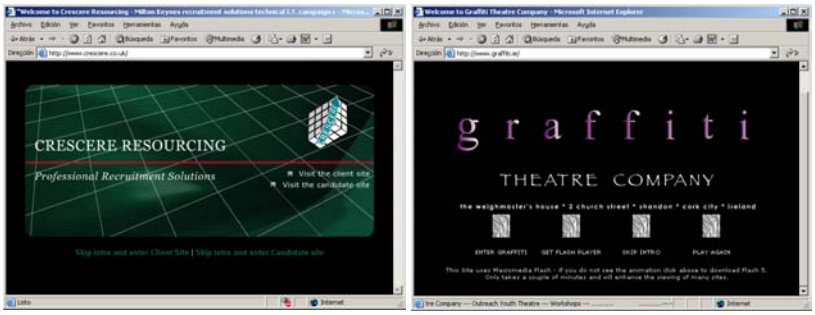
Forces:

- Users need know where they are
- User wants to know where they can go next
- A complex Web site can be very disorienting for users
- Users who are familiar with the structure and content of a Web site should can jump straight to the space where they want to go

Solution:
Provide a reception place where user access conditions can be evaluated. From this welcome point, user will be able to enter to **Homepage** and to another **Indication**s. User's information, such as language or monitor size should be gathered to the provision of Web site's services to user (**Ready**). In its defect, the user should be informed about the best conditions for the visiting Web site or these conditions should be offered directly (**Polyglot**). User can find information about content (**About this**) and owner (**Contact us**) of the Web site in this page. **Welcome** and **Homepage** is the same one in many occasions.

Consequences:
Provide improvements on the navigation, functionality and feedback

Examples & implementation details: http://www.alanismorissette.com These web sites, and many others on the Web, have got a initial page where users are received. These pages have as main features their low-load time (**Busy**), offer the possibility to customise the language or browser properties, and provide information on who, what, when and where the user can find on the web site. Ex.: http://www.crescere.co.uk/, http://www.graffiti.ie/



Indication (aka. Index)

Motivation:
A Web site is a navigational space where users want to achieve goals, such as finding information or buying a product. In a similar way, as occurs in supermarkets, museums or important buildings users need to know where they can go and what they can do once they get there.

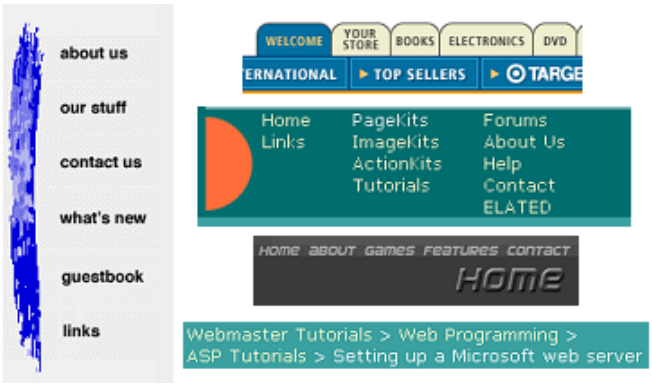
Problem:
How does the user know where he can go and what will he/she find there?
Forces:

- The user should know what places are available
- Doing actions accidentally may be disoriented
- Putting together a collection of objects may take time and mental effort
- The links on Web site should be organised well enough so that the user can find what's needed

Solution:
Web site musts provide the needs mechanisms (meaningful links) that allow any user to move from one place to another places. User can be disoriented and should receive feedback information about his/her Location and has the possibility of coming back (**Second chance**) to a safe place (**Homepage**). Important links should be placed high on the page and descriptive link labels should be used (**Polite**) and if you use frames you should title each frame to facilitate frame identification and navigation.

Consequences:
Provide improvements on the functionality and navigation

Examples & Implementation details: http://www.fourthcube.com, http://www.amazon.com These pages, and others like them provide navigation information by menus, breadcrumbs, buttons or simply links. The situation or appearance of these elements of navigation information can be very varied (left, top, right, etc.). Different kinds of navigation tools can be seen in the following figures:



Polyglot

Motivation:
"The power of the Web is in its universality. Access by everyone regardless of disability is an essential aspect" (Tim Berners-Lee. Director of the W3C). Many factors should be considered: hardware, software, aesthetic, etc.

Problem:
How can the user do a useful use of the Web site and access information at your own pace?

Forces:

- The user wants easy access to information
- The user is principally doing something else, and this shouldn't interfere with it
- The user has little or no incentive to spend time learning technical details
- The user wants full access to everything at once, even if the Web site is complex

Solution:
Speak user's language is "design for all"[Schneiderman 98] [Constantine 99]. Kids, older or disabled people can visit our Web site and universal design techniques can be applied in the design of Web site and his services. These people must know if they are **Ready**. Monitor size, user's screen resolution, connection speed and download time should be considered when you design a Web site, but font sizes and familiar fonts too (**Danger**). Information should be provided of a suitable manner by considering several kinds of peoples and technical features and by using **Polite** language.

Consequences:
Provide improvements on the functionality, language and consistency

Examples & Implementation details:
http://europa.eu.int, http://wap.uclm.es these pages has several links or buttons associated with different ways of visualisation. Basically the user can select between several languages or levels.

Since many people prefer to read printed text many web sites provide printed version of articles or papers (**Print**).

It's important to keep in mind that even if you specify a particular font in your HTML code, it can't display on your viewer's display unless the font exists on your viewer's hard drive. For this reason, it's best to use common fonts such as Arial®, Times New Roman® and Courier™ New. Is important to provide a text equivalent for every non-text element (via ALT, LONGDESC or in element content). This includes: images, graphical representations of text, image map regions, animations, applets and programmatic objects, ascii art, frames scripts, images or videos. For data tables, identify row and column headers.

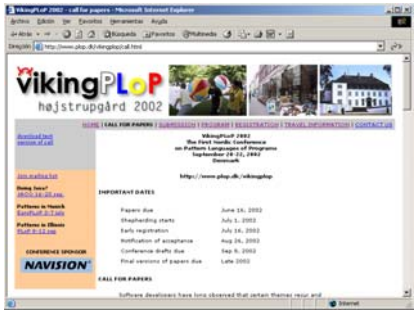
Similarity

Motivation:
When an user is navigating across the Web site must know if he/she is still there or not. The Web site can be very complex and many links can be external to Web site.
Problem:
How does the user know that is visiting the same Web site?
Forces:

- Users need know where they are
- A complex Web site can be very disorienting for users
- Doing actions accidentally may be disoriented

Solution:
Web site should be designed by using the same criteria: colours, fonts, navigation location and layout. Use a single style sheet for all the pages on your site. One of the main benefits of style sheets is to ensure visual continuity as the user navigates your site, but documents should be organized so they may be read without style sheets (**Polyglot**). The user always must be informed by using a suitable way (**Polite**) where he/she is (**Location**) and where he can go (**Indication**). Offering undo/redo mechanisms is advisable (**Second chance**), so as avoiding to use disoriented components (**Danger**).
Consequences:
Provide improvements on the navigation, consistency and feedback

Examples & Implementation details: http://www.ibm.com The implementation of this pattern can be done by using style sheets. Style sheets are a way to separate style from content in Web pages. In an ideal world, you would put all your content (e.g. text and graphics) in one place, and define how that content is laid out (the style) in another. Straight HTML mixes style with content. Style sheets allow you to modify the default attributes of many standard HTML tags. You can create a style sheet in a separate file and then link one or more web pages to it - this is called linking, amazingly enough. Or, you can embed style definitions directly into the <head> section of individual web pages, using the <style> tag - we humans call this embedding. Ex. http://www.plop.dk/VikingPLoP/



Ready

Motivation:
You can use everything to design your Web site, but the user who wants to visit your web will have to have installed the needed plug-ins, and you should remember that you should speak your language (**Polyglot**).

Problem:
How does the user know that he can visit the web site without problems?

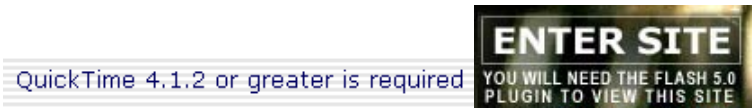
Forces:

- The user wants easy access to information
- A complex Web site can be very disorienting for users
- The user wants to have control over the actions
- The user doesn't want to be interrupted by collateral aspects related with design

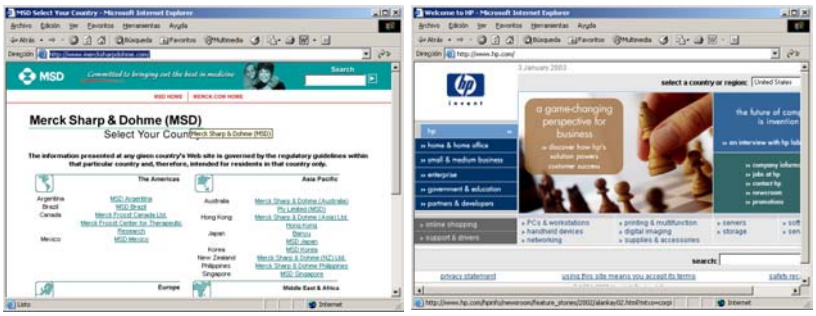
Solution:
Provide tools or needed information to visit the web site of suitable manner. Web site must detect if the user has everything needed and provide links to download places where he will get needed plug-ins. The user does not need to know technical aspects (**Polite**). Ensure that pages are usable when scripts, applets, or other programmatic objects are turned off or not supported. If this is not possible, provide information on an alternative accessible page (**Polyglot**).

Consequences:
Provide improvements on the functionality, control and navigation

Examples & Implementation details:
http://www.mercksharpdohme.com/, http://www.hp.com There are many web sites where plug-ins are required, or minimum monitor resolution is needed (**Danger**). Frames are not supported in other situations. The user needs know that he needs.



For best viewing of this site, I recommend a minimum of 640 x 480 monitor resolution and 16 bit High Colour. Shockwave plug-ins is required. And for those who hate frames, sorry guys! this site relies heavily on frames. If you find any part of this site is not working let me know!



Web Page Level

This section introduces design patterns related with Web page design. They are habitual elements and considered features when we are designing Web sites. In these hierarchical structure a **Homepage** is necessary. In some occasions, the user needs provide information then he/she must fill a form (**Form**) and always the user wants to have the control (**Busy**, **Second Chance**) and to visit web sites to his/her own pace (**Polite**, **Danger**).

Homepage

Motivation:
A Web site can be achieved by random way, but always must have a point of reference. When an user arrives at a Web site, like he/she arrives at a city, town or any important building needs to know where he/she is, what he can do there, and what he need for visiting that Web site. Home Page is an essential component of a Web site. On it questions such as: who?, what?, when? and where? Should have answer.

Problem:
How does the user know where the user is?

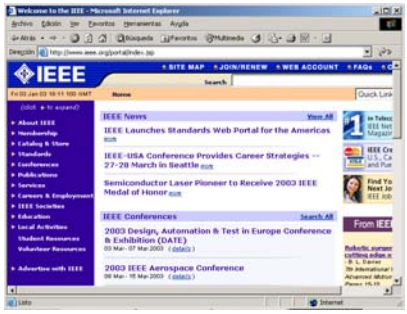
Forces:

- Users need know where they are
- User wants to know where they can go next
- A complex Web site can be very disorienting for users
- Users who are familiar with the structure and content of a Web site should can jump straight to the space where they want to go

Solution:
Provide a starting page where the user feels like at home. Homepage is a place where the user can go back if he is disoriented. Its layout puts important information at top (**Novelty**), includes logos (**Tagline**), search approaches (**Search**) and information contact (**Subscription**, **Contact us**, **About this**).

Consequences:
Provide improvements on the functionality, control and navigation

Examples & Implementation details: http://www.apple.com, http://www.ieee.org Any Web site has a homepage. It is a specific page that introduces distinctive features. The most critical role of the homepage is to communicate what the company is, the value the site offers over the competition and the physical world, and the products or services offered. The challenge is to design a homepage that allows access to all important features without cramming them onto the page itself, too often overwhelming new users. References to homepage should be included in every pages of the web site (**Similarity**).



Polite

Motivation:
For many years, technical communications have stressed the need to use language that's meaningful to readers. That this would be helpful to people seem intuitively obvious. However a difficulty in accomplishing this may be less obvious: people differ widely in terms they choose to describe particular concepts [Evans 98].

Problem:
How can the user access the content of the Web page in a simple and proper way?

Forces:

- Users can be slowed when they must ponder the difference between similar link labels
- There may times when no terms are meaningful to all users of a Web site

Solution:
Use the clearest and simplest language appropriate for a site's content. Create documents that validate to published formal grammars. Associate labels explicitly with their controls (**Indication**). Express only one idea in each sentence (**Tagline**). Long, complicated sentences often mean that you aren't clear about what you want to say. Because asking users seems to be an especially effective way to choose option names, use cards sorting, participatory design, or other methods that involve users whenever possible.

Consequences:
Provide improvements on the functionality, feedback, language and consistency

Examples & Implementation details:
http://www.rae.es, http://ox.ac.uk Languages often have alternative expressions for the same thing ('car' and 'auto'), and a given word can carry different senses ('river bank' vs. 'savings bank') or function as different parts of speech ('to steal'--verb; 'a steal'--noun). Because languages naturally adapt to their situations of use and also reflect the social identities of their speakers, linguistic variation is inevitable and natural. Some words can create problems for you, especially, when you use them without thinking about their true meaning. The way we think and the words we use determine our reactions to life.

Hyperlinks from each chunk should clearly state where it leads when clicked. Links should include a clear, easy-to-ready explanation. Avoid puns and wordplay. Make sure the link it of sufficient length for the reader to click — avoid small one-word links as these will frustrate tired surfers or those with poor eyesight. Aim to assist your readers.

On the Web, underlining should *only* be used to indicate that a particular piece of text is also a link. I've seen a lot of Web pages use underlining for emphasis, which confuses viewers when they click on the underlined text and nothing happens. For this reason, instead of using underscores for emphasis, it's best to use bold type and/or italics.

Busy

Motivation:
Web sites are places where users can download information, images, files or applications, but this downloading can take a lot of time, create significant delays or be accomplished in different ways.

Problem:
How does the user know when his/her operations have finished or the finished state of them?

Forces:

- The user wants to know how long they have to wait for the process to end
- The user wants to know how fast the progress is being made, especially if the speed varies
- Sometimes its impossible to tell how long the process is going to take

Solution:
Provide feedback of the user action (sendings, loadings, downloadings, etc.) Images, files and any element that the user can download should have got information about size, so users can know how long have to wait for the download process. Images and text should be downloaded on-demand (**Size**).

Consequences:
Provide improvements on the functionality, feedback and error prevention

Examples & Implementation details:
http://www.google.com, http://www.acrobat.com Many Web pages needs load a plug-ing to get a correct visualisation, a progress bar is used to provide such information. Sometimes a task running within a program might take a while to complete. A user-friendly program provides some indication to the user about how long the task might take and how much work has already been done. If you don't know or don't want to indicate how complete the task is, you can use a cursor or an animated image to indicate that some work is occurring. If, on the other hand, you want to convey how complete the task is, then you can use a progress bar like this one (http://java.sun.com):

Sometimes, you can't immediately determine the length of a long-running task. You can show this uncertainty by putting the progress bar in *indeterminate mode.* In this mode, the progress bar displays animation to indicate that work is occurring. As soon as the program determines the length of the task, you should switch the progress bar back into its default, determinate mode. In the Java look and feel, indeterminate progress bars look like this:

Second Chance

Motivation:
When the user is navigating on the Web site, he wants to feel the control of his/her operations. He needs know that any operation can be cancelled and that he can return to a previous state.

Problem:
How can the user be sure of his actions?

Forces:

- Doing actions accidentally may be disoriented
- The user wants security and error prevent
- The user wants to explore and not to learn
- The user is in a hurry

Solution:
Provide elements for undo/redo, backing and clearing. These mechanisms in a Web environment consist of providing links to previous page, previous location or **Homepage**. In **Form** is necessary to provide two buttons: "submit" and "reset".

Consequences:
Provide improvements on the control, functionality and error prevention

Examples & Implementation details:
http://www.iomega.com, http://www.acrobat.com. The pages that implements this pattern has got links or buttons that provide undo command. So links to previous sections (page, up, back) or homepage are usual in any page of a web site. Reset form button in Forms (The form) is other example of using this pattern. Browsers provide this functionality too by using back button.

Form

Motivation:
The user has to provide information, usually short answers to questions
Problem:
How can the user provide information to the web site owner?
Forces:

- The user needs to know what kind of information to provide
- Users generally do not enjoy supplying information this way
- It should be clear what is required, and what is optional
- The user is in a hurry

Solution:
Provide appropriate "blanks" to be filled in, which clearly and correctly indicate what information should be provided [Tidwell 98]. **Search, Contact Us** and **Subscription** are examples of forms. In occasions, a form fills a complete page. The user needs know if his/her submit was correctly processed (**Busy**).
Consequences:
Provide improvements on the functionality
Examples & Implementation details: http://www.iomega.com, http://www.iberia.es These pages and other pages where the user can provide information implements this pattern. An HTML form is a section of a document containing normal content, markup, special elements called *controls* (checkboxes, radio buttons, menus, etc.), and labels on those controls. Users generally "complete" a form by modifying its controls (entering text, selecting menu items, etc.), before submitting the form to an agent for processing (e.g., to a Web server, to a mail server, etc.)

Danger

Motivation:
There is a plethora of plug-ins for sound, animation and all kinds of things. But you can't assume that anyone is going to have them, or can use them with their particular computer set-up.

Problem:
How can the user visit a web site without getting confused, being interrupted or being disoriented?

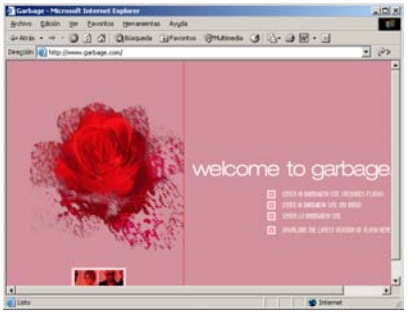
Forces:

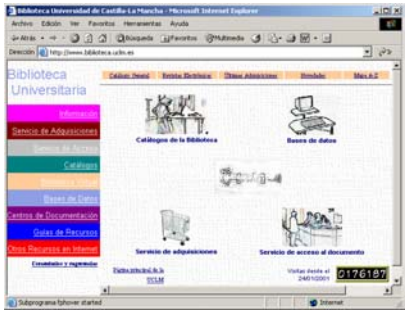
- Users generally have not got the plug-in that he/she needs
- It should be clear what is required to visit the Web site
- Users that can visit Web site is unknown
- Everybody is disability in one way or another

Solution:
Be careful with using disoriented components. For example, you can use readable font size, consider monitor size, use well-designed headings, limit number of frames, limit use of animated gifs, flash, applets, music, rollovers, reduce user's workload, not use blink or marquee elements, limit maximum page size (**Size**, **Colour**). Use style sheets to control layout and presentation.

Consequences:
Provide improvements on visual clarity, control, functionality and navigation.

Examples & Implementation details:
The fundamental design of the Web is based on having the page as the atomic unit of information, and the notion of the page permeates all aspects of the Web [Nielsen, 00].
http://www.garbage.com, http://www.biblioteca.uclm.es are bad examples, antipatterns in this respect. Frames pose many problems, for instance, navigation does not work with frames since the unit of navigation is different from the unit of view. Frames sites are either hard or impossible for search engines to index. If you use frames, people will have a hard time finding your Web site. Frames cause printing issues on older browsers, which tend to print the frame clicked in last, or if not default to top-left, which is generally not the frame you want to print.
The BLINK and MARQUEE elements are not defined in any W3C specification and should not be used".
Internet is a great search engine, and Macromedia Flash is a great visual impact tool but search engines cannot read the text within a Flash 'Movie' or within any other image files like GIF's or JPEG's.





Ornamentation Level

This section introduces decoration features of a Web site. These features provide improvements on general usability of any Web site. They are related with using of **Colour**, **Size**, security (**Secret**) and providing location references (**Location**, **Contact us**) and information (**Subscription**, **Recognize**, **Novelty**).

Tag Line

Motivation: When you are designing a web site you should provide information about the purpose of it.
Problem: How can the user know the purpose of the web site?
Forces:

- Users are in a hurry
- Users don't read web pages, they have a look at pages

Solution: *Include a tagline that explicitly summarise what the site or company does*. Its should be brief, simple and to the point. Include a short description of the site in the window.
Consequences: Provide improvements on visual clarity, functionality and feedback
Examples & Implementation details: http://www.coolhomepages.com, http://www.bbva.es These pages has images or taglines that implements this pattern. A tagline is a short phrase that communicates the "who" and "why' of your Web site. The following elements create effective taglines: subject + audience + organization.

Print

Motivation: Most people read online text differently from how they read printed texts rather than reading word by word, most people quickly scan blocks of online text. When reading text on pages within a Web site, most people also move quickly among pages
Problem: How can the user get a suitable print of information?
Forces:

- Readers appreciate short chunks that can be located quickly
- Most users either save documents to disk or print them out

Solution: *Provide a text version of web pages directly printable*, or offer a downloadable, formatted version of the document to be printed [Lyardet 00].
Consequences: Provide improvements on functionality and control.
Examples & Implementation details: http://www.borland.es, http://www.sport.es This pattern is used when content of long documents is broken up into smaller chunks and linked. Then providing one large page for printing, a file to download (.pdf, .ps, .doc) or the ability to print all of a sectioned document in one step is useful to the user.

Subscription

Motivation: Users want not to visit a web site everyday, they prefer to be informed when new products or news arrive.
Problem: How can the user be informed with meaningful information for him? How can the user have got access to periodic information?
Forces:

- User is in a hurry
- User wants to be informed

Solution: *Provide an approach to user can book on-line* by providing an email. So Web site owner can send information to registered users about **Novelties**. The user should be sure that your email address is not shared with anyone (**Secret**).
Consequences: Provide improvements on feedback.
Examples & Implementation details: http://www.prenhall.com, http://www.sun.es This pattern is implemented by using a simple **Form** where user usually only have to provide an email. In other occasions is necessary provide more information related with preferences of the user and your profile, so is possible provide personalised information. Unsubscribe option should be provided too.

Contact us

Motivation: All business Web sites need to provide a clear way to contact with Web site owner.
Problem: How can the user get additional information on products or documents?
Forces:

- People like to know with whom they are doing business
- Getting company information might be the sole reason that users come to the site
- Many users want to know how is behind the service

Solution: Provide a **Form**, a place or a link in the web site where the user can get additional information about the web site owner and his poducts.
Consequences: Provide improvements on feedback.
Examples & Implementation details: http://www.intel.com, http://www.lucent.com This pattern is implemented by including a page or a section where user can find contact information, in many occasions this information is included in bottom of all pages of the web site. In others cases, a form is provided to the user. This **form** contains features like a textarea or textfields in order to the user can provide his email and questions. Data provided by the user are **Secrets**.

Search

Motivation: Search is one of the most important elements of a Homepage, and it's essential that users be able to find it easily and use it effortlessly.
Problem: How can the user know if a web site can provide specified information that he wants?
Forces:

- User wants to know if the searched information is on the Web site
- User doesn't read web site. He/she has a look at it.

Solution: *Provide a search engine or overview page*. Give users an input box on the **Homepage** to enter search queries, instead of just giving them a link to a search page [Nielsen 02]. Search on the homepage should search the entire site default. [see Welie's patterns 01]
Consequences: Provide improvements on functionality and control.
Examples & Implementation details: http://www.paginasamarillas.es, http://www.microsoft.com. This pattern is implemented by providing a search form. Search forms are the user interface of the search engine. It can consist of a very simple **Form** with just a text field and a button, it can be a page and add a lint to it in your navigation. Advanced search capabilities can be worth adding. An advanced search page with options for phrases, multiple fields, special collections or zones, and date ranges allows them to perform more precise searches.

Recognize

Motivation: When a user comes back to a Web site he needs know what places he has visited, what documents he has downloaded and if there are modifications from last visit.
Problem: How does the user know where he/she has been?
Forces:

- User does not want to loose his time
- User wants to receive personalised information

Solution: *Keep information about user actions, visited places, logins, etc, for instance by using cookies*. Since HTTP is a non-persistent protocol, it is impossible to differentiate between visits to a web site, unless the server can somehow mark a visitor.
Consequences: Provide improvements on feedback and error prevention
Examples & Implementation details: http://www.kinkos.com, http://www.americanairlines.com. This pattern is implemented by using cookies. Web cookies are simply bits of software placed on your computer when you browse Web sites, so the web site will recognise the user's computer when he comes back to visit again. Cookies have some beneficial things. For example, when you log on or purchase online to certain sites, did you ever notice that when you return again you do not have to sign on the next time? That's because it stored your password and id on your machine in a cookie. So user's workload is reduced.

Colour

Motivation: Many web designers overlook the importance of colour when designing a web site. Colour should be one of your first concerns when it comes time to start your web site design.
Problem: How can the user access to information in a suitable way?
Forces:

- Web browsers can only see 256 colours
- People reading light characters on dark backgrounds for long periods reported less visual fatigue

Solution: *Provide information by using suitable colours in fonts, backgrounds and image.* For example, the default colours for Web page links are blue for non-visited links and purple for visited links. Ensure that foreground and background colour combinations provide sufficient contrast when viewed by someone having colour deficits or when viewed on a back and white screen. You should use yellow and red colours sparingly in your Web site itself. Only use them in areas where you want the visitor to focus on. Do not make large parts of your web site with bright colour.
Consequences: Provide improvements on feedback, functionality, consistency and visual clarity
Examples & Implementation details: http://www.biblioteca.uclm.es, http://www.trashclub.com. Initially, one of the limiting aspects of designing for the Web can be the 216-colour palette. The idea behind the Netscape-created colour-set was to maintain a consistent appearance for Web pages viewed on a Windows, Macintosh, or Unix machine. Creating a consistent Web site, it's best to base the site's colours on the Websafe ones. Luckily, most HTML editors now have that palette built in. This pattern is implemented by choosing colour combinations where readability and visual clarity is improved and ensuring that all information conveyed with colour is also available without colour for example from context or markup. Backgrounds should not detract from readability. The first reference is an example of bad use of colour.

Location

Motivation: When a user arrives at a Web site, like he/she arrives at a city, town or any important building needs to know where he/she is.
Problem: How does the user know where he/she is?
Forces:

- Users need know where they are
- A complex Web site can be very disorienting for users

Solution: Provide feedback information about location of the user in the web site.
Consequences: Provide improvements on navigation, consistency and feedback
Examples & Implementation details: http://java.sun.com, http://www.acrobat.com This pattern is implemented by placing references in all web pages of the web site: using titles and breadcrumbs. Usually complex Web site includes a sitemap.

Novelty

Motivation: Users want to know if there are new features in the Web site. Users admit suggestions and want to know offers and promotions.
Problem: How can the user know novelties and latest news or suggestions of a web site?
Forces: User doesn't read web site. He/she has a look at it
Users are in a hurry
Solution: *Provide novelties of the web site in a clear and intuitive manner* where users will have rapid access to new services offered by web site
Consequences: Provide improvements on functionality and navigation
Examples & Implementation details: http://www.microsoft.com, http://www.terra.es This pattern is implemented by placing latest news or suggestions in an outstanding place in the **Homepage**.

Size

Motivation: Design for the WWW is a balancing act between the graphics "wow" and the real time "now". The more graphically intense a site the longer it can take to download.
Problem: How can the user access to information in a suitable way?
Forces:

- With a 28.8k connection, your computer can receive, on average, 2K per second. No one wants to wait 30 seconds just to see your site logo
- Developing fixed-size Web pages is a fundamentally flawed practice

Solution: *Provide information by using suitable sizes in images, fonts, and pages.* Animations, images, long files should be provided if the user really wants it (on-demand). Page length, scrolling vs. paging needs, font size are important aspects.
Consequences: Provide improvements on control, consistency and visual clarity
Examples & Implementation details: http://www.number-10.gov.uk, http://www.tagheuer.com This pattern is implemented when thumbnails are used, when width and height attributes of the <IMG> tag are used, using standard fonts, or when large volume of information is broken into chunks and written in several pages avoiding scrolling. Organize documents so they may be read without style sheets is useful.

About this

Motivation: All business Web sites need to provide a clear way to find information about the company no matter how big or small the company is
Problem: How does the user know which is the purpose of the site?
Forces:

- People like to know with whom they are doing business
- Getting company information might be the sole reason that users come to the site
- Many users want to know who is behind the service

Solution: Provide a place or a link where the user can get information about the web site's content.
Consequences: Provide improvements on functionality and feedback.
Examples & Implementation details: http://www.sunspot.net, http://www.ireland.com This pattern is implemented by adding a section where information about owner of the page can be found. Normally a link to this section is situated in the **Homepage**.

Secret

Motivation: If user provides private information, he/she will need to have the right to expect confidentiality. Rapid advances in communication technology have accentuated the need for security in the Internet.
Problem: How can the user be sure that information which he provides is protected?
Forces:

- Users want to security
- Users do not need to know technical aspects

Solution: *Provide security needed mechanisms (access and privacy) and inform to the user of security conditions and terms of use*. Users should be registered (**Subscription**) and so access to private sections on the Web site is allowed, but only a login and password is not sufficient sometimes [Yoder98].
Consequences: Provide improvements on feedback and control
Examples & Implementation details: http://www.bankofamerica.com, http://www.cdnow.com This pattern is implemented with login form where we'll ask the user for their username and password by using php or asp. But unless your form is located on a secure server, the information is transmitted in clear text, and encryption won't occur until the php script runs.
Include links where users can read web site Privacy Policy Statement | Terms of Use

Summary

The following tables summarise the patterns in this pattern catalogue for reference purposes. These patterns could be integrated on a methodology like a checklist to develop user interfaces like IDEAS [Lozano 01]. There are patterns of requirements, like these, that can be used in beginning of an usability-based iterative life cycle. So patterns can be used to improve a participatory design, evaluate web site under usability criteria and facilitate communication between stakeholders involved in Web site developing.

Problem	Solution	Pattern name
How does the user know where he is?	Supply a reception place where conditions of user access to web site can be evaluated	Welcome
How does the user know where he can go and what he will find there?	Provide meaningful links to the different pages of the web sites	Indication
How can the user visit the web site at his own pace?	Provide information of a suitable way by taking into account users	Polyglot
How does the user know that he is in the same web site?	Provide an uniform aspect of the web site (colours, sizes, distribution, etc.)	Similarity
How does the user know that he can visit the web site without problems?	Provide tools or information needed to visit the web site of suitable manner	Ready

Problem	Solution	Pattern name
How does the user know where he is?	Provide a reference point of the web site	Homepage
How can the user access the content of the Web page in a simple and proper way?	Provide information by using a simple language and jargon is avoided	Polite
How does the user know when his operations have finished or the finished state of them?	Provide feedback of the user action (sending, loadings, downloading, etc.)	Busy
How can the user be sure of his actions?	Provide elements for undo/redo, backing and clearing	Second chance
How can the user provide information to the web site owner?	Provide appropriate “blanks” to be filled in, which clearly and correctly indicate what information should be provided	Form
How can the user visit a web site without getting confused, being interrupted or being disoriented?	Be careful with using disoriented components (frames, animated gifs, floating, windows, banners, applets, flash, etc.)	Danger!

Problem	Solution	Pattern name
How can the user know the purpose of the web site is?	Provide a slogan or image that identifies the web site and its purpose	Tag Line
How can the user get a suitable print of information?	Provide information on several ways and formats and give the possibility of printing or downloading wide documents	Print
How can the user be informed with meaningful information for him?	Provide a Form where the user can get information that he wants automatically	Subscription
How can the user request for additional information about the content of the web site?	Provide a Form, a place or a link in the web site where the user can get additional information about the web site owner and his products	Contact us
How can the user know if a web site can provide the information he wants?	Provide a search engine or overview page	Search
How does the user know where he/she has been?	Keep information about user actions, visited places, logins, etc.	Recognize
How can the user access to information web site in a suitable way?	Provide information by using suitable colours in fonts, backgrounds and image.	Colour
How can the user access to information web site in a suitable way?	Provide information by using suitable sizes in images, fonts, and pages	Size
How can the user be sure that information that he provides is protected?	Provide security mechanisms (access and privacy) needed to protect user data and the web site and inform to the user of security conditions	Secret
How can the user know novelties and latest news of a web site?	Provide suggestions and news of the web site in a clear and intuitive manner	Novelty
How can the user know where he is, or what is the section that he is visiting?	Provide feedback information about location of the user in the web site	Location
How can the user get additional information about web site owner?	Include a link to an “About Us” section	About this

Acknowledgements

We would like to thank our shepherd Serge Demeyer for his valuable suggestions and comments on this paper. This work is supported in part by the Spanish CICYT TIC 2000-1673-C06-06 and CICYT TIC 2000-1106-C02-02 grants.

References

[Alexander 77] Christopher Alexander. "A Pattern Language", Oxford University Press, 1977.

[Alexander 79] Christopher Alexander. "The Timeless Way of Building", Oxford University Press, 1979.

[Constantine 01] Constantine Stephanidis, Anthony Savidis. "Universal Access in the Information Society: Methods, Tools and Interaction Technologies." Springer-Verlang. 2001.

[Erickson 97] Thomas Erickson, "Supporting Interdisciplinary Design: Towards Pattern Languages for Workplaces". 1997. http://www.pliant.org/personal/Tom_Erickson/Patterns.Chapter.html

[Evans 98] Mary Evans. "Web Design: An Empiricist's Guide". University of Whasington, Seatle, Washington. 1998.

[Lozano 01] María Lozano, Isidro Ramos, Pascual González. "User interface Specification and Modeling in an Object Oriented Environment for Automatic Software Development". IEEE 34th International Conference on TOOLS USA. 2001.

[Lyardet 00] Fernando Lyardet, Gustavo Rossi. Web Usability Patterns.2000.

[Meszaros & Doble 94] Gerard Meszaros, Jim Doble, "A Pattern Language for Pattern Writing", in Martin, Riehle, Buschmann, Pattern Languages of Program Design 3. Reading, Mass: Addison-Wesley, 1994.

[Nielsen 00] Jakob Nielsen, "Designing Web Usability: The Practice of Simplicity". New Riders Publishing. 2000.

[Nielsen 02] Jakob Nielsen, Marie Tahir. "Homepage Usability: 50 Websites deconstructed ". New Riders. 2002.

[Shneiderman 98] Ben Shneiderman. "Designing the User Interface: Strategies for Effective Human-Computer Interaction". Addison Wesley. 1998.

[Tidwell 98] Jenifer Tidwell. "Common Ground: A Pattern Language for Human-Computer Interaction". http://www.mit.edu/~jtidwell/. 1998/99

[Welie 01] Martijn van Welie. Interaction design patterns. http://www.welie.com/. 2001.

[Yoder 98] Joseph Yoder, Jeffrey Barcalow. "Arquitectural Patterns for Enabling Application Security". PloP'97 D-4 book. 1998.

[Guidelines] Yale C/AIM WWW Style Manual
http://info.med.yale.edu
Apple's Web Design Guide
http://applenet.apple.com
IBM Web Design Guidelines
http://www.ibm.com/IBM/HCI/guidelines

Using watchdog timers to improve the reliability of single-processor embedded systems: Seven new patterns and a case study

Michael J. Pont[1] and Royan H.L. Ong
Control & Instrumentation Research Group, Department of Engineering, University of Leicester, University Road, LEICESTER LE1 7RH, UK.

Background

We have recently described a "language" consisting of more than seventy patterns, which will be referred to here as the "PTTES Collection" (see Appendix 1). This language is intended to support the development of reliable embedded systems: the particular focus of the collection is on systems with a time triggered, co-operatively scheduled (TTCS) system architecture.

In the PTTES Collection, the pattern HARDWARE WATCHDOG was presented: this pattern was intended to describe how to use watchdog timers (such as the ubiquitous '1232' chip) in any embedded application. At the time the PTTES Collection was assembled, HARDWARE WATCHDOG was viewed as a very basic pattern, and it was presented in an introductory part of the book. However, in recent months (as we have helped other people use the PTTES Collection in a number of projects), it has become clear that the range of ways in which watchdog timers can be used in TTCS applications was not adequately described in the original, rather superficial, pattern. We therefore began work on a new version.

The patterns in this paper

The seven patterns described in this paper, together, form a replacement for HARDWARE WATCHDOG. In this new collection, WATCHDOG RECOVERY is the entry point. This pattern describes, in general terms, how to use a watchdog with TTCS applications: it also provides links to the various other patterns in this paper.

More specific applications for watchdog timers are detailed in SCHEDULER WATCHDOG (which describes how you can detect the failure of the scheduler in your system), and PROGRAM-FLOW WATCHDOG (which describes how to detect program-flow errors).

Using SCHEDULER WATCHDOG and / or PROGRAM-FLOW WATCHDOG allows you to detect certain important types of error: the "recovery" patterns may then prove helpful. The first such pattern (RESET RECOVERY) is also the simplest: this describes how and when to use a simple system reset to recover from some forms of error. The second recovery pattern is Fail-Silent Recovery: this describes how and when to shut down your system in the event of a serious error. The third recovery pattern is Limp-Home Recovery: this describes mechanisms that will let your system "limp home" - in the event of an error - by running a simple version of the original algorithm.

[1] To whom correspondence should be addressed: M.Pont@le.ac.uk.

Finally, we have met many developers (some with considerable experience) who believe that general-purpose watchdog timers can form the basis of techniques for detecting oscillator failure. One reason for including OSCILLATOR WATCHDOG in this paper is to help dispel this (sometimes dangerous) myth.

Case study

A short case study is presented at the end of the paper. This employs an example (an automotive cruise-control system) to illustrate how many of the patterns presented in this paper can be used in a realistic embedded application.

Acknowledgements

We are very grateful to Bob Hanmer (our Shepherd at VikingPLoP), who provided numerous useful suggestions during the evolution of this paper prior to the conference. We are also grateful to the members of our workshop (Mikio Aoyama, Walter Cazzola, Lars Grunske, Kevlin Henney, Juha Pärssinen, Kristian Elof Sørensen), who provided further suggestions for improving this paper at the conference itself.

Responsibility for all remaining bugs and errors rests with the authors.

WATCHDOG RECOVERY

Context

- You are developing a single-processor embedded application using a member of the 8051 family of microcontrollers (or similar hardware).
- You are programming in C (or a similar language).
- The application has a time-triggered architecture, constructed using a scheduler (e.g. CO-OPERATIVE SCHEDULER [Pont, 2001, page 254]).

Problem

How can you make best use of a watchdog timer in your TTCS application?

Background

Suppose there is a hungry dog guarding a house (Figure 1), and someone wishes to break in . If, during the burglary, an accomplice repeatedly throws the guard dog small pieces of meat, then the animal will be so busy concentrating on the food that he will ignore his guard duties, and will not bark. However, if the accomplice run out of meat or forgets to feed the dog for some other reason, the animal will start barking, thereby alerting the neighbours, property occupants or police.

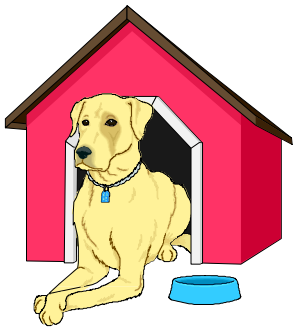


Figure 1: The origins of the 'watchdog' analogy. See text for details.

This type of canine behaviour is mirrored (to an extent) in computerised "watchdog timers" used in microcontroller-based systems. More specifically, the watchdog timers used to implement WATCHDOG RECOVERY will - usually - have the following two features:

- The timer must be refreshed at regular, well-defined, intervals.

  If the timer is not refreshed at the required time it will overflow, an process which will usually cause the associated microcontroller to be reset.

- When starting up, the microcontroller can determine the cause of the reset.

  That is, it can determine if it has been started 'normally', or re-started as a result of a watchdog overflow. This means that, in the latter case, the programmer can ensure that the system will try to handle the error that caused the watchdog overflow.

As we will see, the features of watchdog hardware are a good match for the needs of TTCS systems, and - with a little care - a watchdog timer can form the basis of a simple but effective way of improving your system's ability to handle a range of different faults.

Solution

Understanding the basic operation of watchdog timer hardware is not difficult. However, making good use of this hardware in a TTCS application requires some care. As we will see in this section, there are three main issues which need to be considered:

- Choice of hardware;
- The watchdog-induced reset;
- The recovery process.

We begin by considering the choice of hardware.

Choice of hardware

We have seen in many previous cases (in Pont, 2001) that, where available, the use of on-chip components is to be preferred to the use of equivalent off-chip components. Specifically, on-chip components tend to offer the following benefits:

- Reduced hardware complexity, which tends to result in increased system reliability.
- Reduced application cost.
- Reduced application size.

These factors also apply when selecting a watchdog timer. In addition, when implementing WATCHDOG RECOVERY, it is usually important that the system is able to determine - as it begins operation - whether it was reset as a result of normal power cycling, or because of a watchdog timeout. **In most cases, only on-chip watchdogs allow you to determine the cause of the reset in a simple and reliable manner.**

With appropriate on-chip hardware, determining the cause of a reset is usually straightforward: we give an example at the end of this pattern to illustrate this.

The watchdog-induced reset

We consider time-based error detection, handling program-flow errors, and other - more general - uses for watchdog resets in this section.

(a) Time-based error detection

A key requirement in applications using a co-operative scheduler is that, for all tasks, under all circumstances, the following condition must be adhered to:

$$Duration_{Task} < Interval_{Tick}$$

Where: $Duration_{Task}$ is the task duration, and $Interval_{Tick}$ is the system 'tick interval'.

The pattern SCHEDULER WATCHDOG [this paper] describes techniques that will help you to meet this condition.

(b) Responding to program-flow errors

Timer-based error detection requires the watchdog timer to do two things:

1. Detect time-related errors;
2. Cause a system reset (and, thereby, invoke an error-recovery process).

Time-based error detection is not the only possibility. When the system uses a watchdog timer, we can use this timer to force a system reset at any time, through the use of an endless loop:

```
// One way of forcing a watchdog-induced reset
while(1)
   ;
```

Use of a watchdog in this way is particularly appropriate in situations where you have detected an error, and the nature of this error means that you cannot be sure what state the system is currently in.

One form of error that gives rise to such concerns is the program-flow error, which can occur as a result of electromagnetic interference. When such errors occur the program flow may be diverted to a "random" address in code memory. By the time you manage to detect that such a random jump has taken place, it is generally impossible to predict what damage has been done, and you therefore cannot be sure that, if you call an error-handling function, it will operate as intended.

In these circumstance, we can use the watchdog timer to perform a system reset, after which we call the error handler. In doing this, we assume that the system is more likely to operate correctly after it is reset, and that the error-handling function will therefore work more effectively when called in this way.

This technique for dealing with program-flow errors is discussed in detail in PROGRAM-FLOW WATCHDOG [this paper].

(c) Other uses for watchdog-induced resets

If your system uses watchdog-induced resets to handle program-flow errors, and / or it uses timer-based error detection techniques, then it can make sense to also use watchdog-induced resets to handle other errors. Doing this means that you can integrate some or all of your error-handling mechanisms in a single place (usually in some form of system initialisation function). This can - in many systems - provide a very "clean" and approach to error handling that is easy to understand (and maintain).

Note that this combined approach is only appropriate where the recovery behaviour you will implement is the **same** for the different errors you are trying to detect: an examination of the possible error-recovery mechanisms (which are summarised in the next section) may help you to decide if this is the case for your system.

Here are some suggestions for the types of errors that can be effectively handled in this way:

- Failure of on-chip hardware (e.g. analogue-to-digital converters, ports).
- Failure of external actuators (e.g. DC motors in an industrial robot; stepper motors in a printer).
- Failure of external sensors (e.g. ultraviolet sensor in an art gallery; vibration sensor in an automotive system).
- Temporary reduction is power-supply voltage.

We illustrate the use of this approach to error handling in the case study at the end of this paper.

Recovery behaviour

Before we decide whether we need to carry out recovery behaviour, we assume that the system has been reset. If the reset was "normal" we simply start the scheduler and run the standard system configuration.

If, instead, the cause of the reset was a watchdog overflow, then there are three main options:

- We can simply continue **as if** the processor had undergone an "ordinary" reset. This option is discussed in the pattern RESET RECOVERY [this paper].
- We can try to "freeze" the system in the reset state. This option is discussed in the pattern FAIL-SILENT RECOVERY [this paper].
- We can try to have the system run a different algorithm (typically, a very simple version of the original algorithm, often without using the scheduler). This option is discussed in the pattern LIMP-HOME RECOVERY [this paper].

Hardware resource implications

The main resource implication is that a suitable watchdog timer is required.

Reliability and safety implications

We consider a number of key features in this section.

Risk assessment

In safety-related or safety-critical systems, this pattern should not be implemented before a complete risk-assessment study has been conducted (by suitably-qualified individuals).

Successful use of this pattern requires a full understanding of the errors that are likely to be detected by your error-detection strategies (and those that will be missed), plus an equal understanding of the recovery strategy that you have chosen to implement. Without a complete investigation of these issues, you cannot be sure that implementation of the pattern you will increase (rather than decrease) the reliability of your application.

The limitations of single-processor designs

It is important to appreciate that there is a limit to the extent to which reliability of a single-processor embedded system can be improved using a watchdog timer.

For example, LIMP-HOME RECOVERY is the most sophisticated recovery strategy considered in this paper. If implemented with due care, it can prove very effective. However, it relies for its operation on the fact that - even in the presence of an error - the processor itself (and key support circuitry, such as the oscillator, power supply, etc) still continues to function. If the processor or oscillator suffer physical damage, or power is removed, LIMP-HOME RECOVERY cannot help your system to recover.

In the event of physical damage to your "main" processor (or its support hardware), you may need to have some means of engaging another processor to take over the required computational task. One way to perform this type of activity is to use WATCHDOG SLAVE[2].

Time, time, time …

Suppose that the braking system in an automotive application uses a 500 ms watchdog and the vehicle encounters a problem when it is travelling at 70 miles per hour (110 km per hour). In these circumstances, the vehicle and its passengers will have travelled some 15 metres / 16 yards - right into the car in front - before the vehicle even begins to switch to a "limp-home" braking system.

In some circumstances, the programmer can reduce the delays involved with watchdog-induced resets, and thereby improve the system reliability. For example, many systems force a watchdog reset using code like this:

```
// One way of forcing a watchdog-induced reset
while(1)
   ;
```

[2] WATCHDOG SLAVE is a based on the "shared-clock scheduler" architecture (see Pont, 2001, Part F). The pattern is still under development, and details will be released at a future PLoP conference.

Some hardware allows you to adjust the watchdog delay while the watchdog is active. This can be a useful means of reducing the delays involved in the watchdog-induced reset. For example, using the Infineon C515C, the watchdog reload register can be changed at any time, thereby altering the overflow period. This allows the programmer to do the following:

```
// Set up the watchdog for "normal" use
// - overflow period = ~39 ms
WDTREL = 0x00;

...

// Adjust watchdog timer for faster reset
// - overflow set to ~300 µs
WDTREL = 0x7F;

// Now force watchdog-induced reset
while(1)
   ;
```

On-chip watchdogs and 'idle' mode

In most applications based on CO-OPERATIVE SCHEDULER [Pont, 2001, page 254], the microcontroller enters 'idle' mode between scheduler ticks, after executing the Dispatcher function.

You need to be aware that - when entering idle mode - some microcontrollers **disable the (on-chip) watchdog timer**. If this happens, then none of your watchdog-based error-handling mechanisms will operate correctly.

If your chosen microcontroller disables the watchdog timer in idle mode, it may be necessary to avoid using this mode. Please note that the scheduler will still operate correctly in these circumstances; however, the power consumption of your system will increase.

Portability

This pattern does not rely (in any way) on features which are unique to the 8051 family: it can be applied in systems based on any microcontroller (e.g. PIC, AVR, HC08, C16x, ARM, etc).

Overall strengths and weaknesses

☺ Watchdogs can provide a 'last resort' form of error recovery. If you think of the use of watchdogs in terms of 'if all else fails, then we'll let the watchdog reset the system', you are taking a realistic view of the capabilities of this approach.

☹ Use of this technique usually requires an on-chip watchdog.

☹ Used without due care at the design phase and / or adequate testing, watchdogs can reduce the system reliability dramatically. In particular, in the presence of sustained faults, badly-designed watchdog "recovery" mechanisms can cause your system to repeatedly reset itself. **This can be very dangerous.**

☹ Watchdogs with long timeout periods are unsuitable for many applications.

Related patterns and alternative solutions

We consider a number of related patterns and alternative solutions in this section.

Related pattern: SCHEDULER WATCHDOG [this paper]

This pattern describes how you can you use a watchdog timer to ensure that the scheduler in your TTCS application is operating correctly.

Related pattern: PROGRAM-FLOW WATCHDOG [this paper]

This pattern provides a description of a popular technique for dealing with program-flow errors in embedded systems: such errors are often thought to arise from electromagnetic interference.

Related pattern: OSCILLATOR WATCHDOG [this paper]

This pattern describes how to deal with oscillator failures in a single-processor embedded system.

Related pattern: RESET RECOVERY [this paper]

This pattern describes a very simple recovery strategy that can be used after a watchdog-induced reset.

Related pattern: FAIL-SILENT RECOVERY [this paper]

This pattern describes how to shut down your system after a watchdog-induced reset.

Related pattern: LIMP-HOME RECOVERY [this paper]

This pattern describes how you can re-start your system (and run a different - usually very simple - algorithm), after a watchdog-induced reset.

Other simple watchdog solutions

Bruce Powel Douglass has described an alternative watchdog pattern (WATCHDOG [Douglass, 1999, p.646]): note that this pattern is not tailored for use with TTCS applications.

A software watchdog?

In certain restricted circumstances, a software watchdog may also be useful. This can be created from two components:

- A Timer ISR;
- A refresh function.

Essentially, we set a timer to overflow in (say) 60 ms. Under normal circumstances, this timer will never overflow, because we will call the “refresh” function regularly, and - thereby - restart the timer. If, however, the program is ‘jammed’, the refresh function will not be called. When the timer overflows, the ISR will be called: this can be used to implement an ‘appropriate’ error recovery strategy.

The main advantage of a software watchdogs is that different forms of error recovery (not necessarily involving a complete chip reset) are possible.

The main concern with this approach is that some errors (for example, those induced by EMI) may disrupt the "software" timer as well as the main application code: hardware watchdogs **appear** to be more robust in these circumstances.

Note that some hardware provides a way of obtaining a combination of "software" and "hardware" watchdogs. Specifically, the DS87C520 (and similar family members) allow the programmer to invoke an interrupt service routine (ISR) a short time before the chip undergoes a full reset. This provides a mechanism for trying to deal with the source of the error in an ISR and - if unsuccessful - allowing a full reset to take place.

When one processor is not enough

As noted in "solution", there is a limit to the extent to which reliability of a single-processor embedded system can be improved using any form of watchdog timer.

Using a shared-clock scheduler (see Pont, 2001, Part F) can sometimes be a useful alternative to the techniques discussed in this pattern.

Other patterns

Some alternative patterns for fault tolerance and error recovery which may be of interest were presented recently by Saridakis (2002).

Example: Automotive cruise control.

Use of WATCHDOG RECOVERY is illustrated in the case study at the end of this paper.

Example: Determining the cause of a watchdog reset

As noted in "Solution", most implementations of WATCHDOG RECOVERY rely on an ability to determine the cause of a system reset. Fortunately, this is usually easy to do. For example, in the Infineon C515C, the WDTS flag (bit 6 in the register IP0) is set if the reset was caused by a watchdog timer overflow. Having determined the status of this bit, it should be cleared in software:

```
// Determine if reset was caused by watchdog overflow (C515C)
if (IP0 & 0x40)
   {
   // WDTS flag is set - reset *was* caused by watchdog
   Watchdog_reset_G = 1;

   // Clear the IP0 flag
   IP0 &= 0xBF;
   }
else
   {
   Watchdog_reset_G = 0;
   }
```

SCHEDULER WATCHDOG

Context

- You are developing a single-processor embedded application using a member of the 8051 family of microcontrollers (or similar hardware).
- You are programming in C (or a similar language).
- The application has a time-triggered architecture, constructed using a scheduler (e.g. CO-OPERATIVE SCHEDULER [Pont, 2001, page 254]).

Problem

How can you detect that the scheduler in your TTCS application has stopped operating correctly?

Background

General background

SCHEDULER WATCHDOG can be seen as an implementation of the more general pattern WATCHDOG RECOVERY [this paper]: please refer to WATCHDOG RECOVERY for background information that will assist in the understanding of the present pattern.

Features of co-operative schedulers

As we discussed in CO-OPERATIVE SCHEDULER [Pont, 2001, page 254], a key requirement in applications using a (co-operative) scheduler is that, for all tasks, under all circumstances, the following condition must be adhered to:

$$Duration_{Task} < Interval_{Tick} \qquad \text{- Eq. 1}$$

Where: $Duration_{Task}$ is the task duration, and $Interval_{Tick}$ is the system 'tick interval'.

Simply satisfying **Equation 1** is not sufficient to guarantee the integrity of the scheduling, since we also need to take into account the CPU overheads imposed by the running of the scheduler itself. We can represent this as follows:

$$Duration_{Task} < \left(1 - \frac{CPU_{Scheduler}}{100}\right) \times Interval_{Tick} \qquad \text{- Eq. 2}$$

Where: $CPU_{Scheduler}$ is the percentage of the available CPU time consumed by the scheduler itself.

Equation 2 will be applicable where (only) one task is scheduled to execute at any tick interval. If this condition is not satisfied, then we also need to take into account the duration of *all* tasks that are scheduled to run in the same tick interval. Thus - at every tick interval - we need to ensure that:

$$\sum_{i=1}^{N} Duration_{Task\ i} < \left(1 - \frac{CPU_{Scheduler}}{100}\right) \times Interval_{Tick} \qquad \text{- Eq. 3}$$

Where: $\sum_{i=1}^{N} Duration_{Task\ i}$ is the sum of the duration of all the tasks scheduled to run at a particular tick interval.

We have previously discussed (Pont, 2001) a number of techniques[3] which can help you meet the condition summarised in **Equation 3**.

In many systems, the designers apply SCHEDULER WATCHDOG in order to develop a final safety net for their system.

Solution

We will start by considering time-based error detection techniques.

Time-based error detection

It is possible to use a watchdog timer to test the condition summarised in **Equation 3** (in Background), as follows:

- Set the watchdog timer to overflow at a period greater than the tick interval.
- Create a task that will update the watchdog timer shortly before it overflows.
- Start the watchdog.

Under normal circumstances, the watchdog timer will never overflow, and your system will operate as normal. However, if the duration of a task (or the duration of a sequence of tasks, scheduled to execute in the same tick interval) cause the scheduling to be significantly disrupted, the watchdog timer will reset the system.

Selecting the overflow period

Selecting the watchdog overflow period requires some care, since the choice of the overflow period will depend on the system characteristics.

(a) Systems with 'hard' timing constraints

For systems with "hard" timing constraints for one or more tasks, it is usually appropriate to set the watchdog overflow period to a value slightly greater than the tick interval (e.g. 1.1 ms overflow in a system with 1 ms ticks). In this way, you will very rapidly detect scheduling problems.

---

3 A summary of these techniques is given in "Related patterns and alternative solutions".

Please note that to do this, the watchdog timer will usually need to be driven by a crystal oscillator (or the timing will not be sufficiently accurate). In addition, the watchdog timer will need to give you enough control over the timer settings, so that the required overflow period can be set.

(b) Systems with 'soft' timing constraints

The 'hard timing' approach is very effective, but before deciding on this option, you should bear in mind the fact that many ('soft') TTCS systems continue to operate safely and effectively, even if - at times - the duration of the task(s) that are scheduled to run at a particular time exceeds the tick interval.

To give a simple example, a scheduler with a 1 ms tick interval can - without problems - schedule a single task with a duration of 10 ms that is called every 20 ms.

Of course, if the same system is also trying to schedule a task of duration 0.1 ms every 5 ms, then the 0.1 ms task will sometimes be blocked. Often careful design will avoid this blockage but - even if it occurs - it still may not matter because, although the 0.1 ms will not always run on time, it will always run (that is, it will run 200 times every second, as required).

For some tasks - with soft deadlines - this type of behaviour may be acceptable. If it is, then it is appropriate to use a watchdog timer with a longer time-out period.

Typically, this will be done as follows:

- Set the watchdog to overflow after a period of around 100 ms.
- Feed the watchdog every millisecond, using an appropriate task.
- Only if the scheduling is blocked for more than 100 ms will the system be reset.

Recovery strategies

In the event that the watchdog timer has overflowed, we know that something has disrupted the scheduling.

A range of suitable recovery strategies are discussed in RESET RECOVERY [this paper], FAIL-SILENT RECOVERY [this paper] and LIMP-HOME RECOVERY [this paper].

Hardware resource implications

Using SCHEDULER WATCHDOG requires an appropriate watchdog timer: please see WATCHDOG RECOVERY [this paper] for details.

Reliability and safety implications

In safety-related or safety-critical systems, this pattern should not be implemented before a complete risk-assessment study has been conducted (by suitably-qualified individuals).

Successful use of this pattern requires a full understanding of the errors that are likely to be detected by your error-detection strategies (and those that will be missed), plus an equal understanding of the recovery strategy that you have chosen to implement. Without a complete investigation of these issues, you cannot be sure that implementation of the pattern you will increase (rather than decrease) the reliability of your application.

Please see WATCHDOG RECOVERY [this paper] for further discussion of the reliability and safety implications associated with watchdog timers.

Portability

The approach to error detection a recovery described in SCHEDULER WATCHDOG is not in any way specific to the 8051 microcontroller family: it can be used with any device.

Overall strengths and weaknesses

☺ **SCHEDULER WATCHDOG** provides a useful "safety net" in the event that problems in the system disrupt the scheduling.

☹ Use of this technique usually requires an on-chip watchdog.

☹ Used without due care at the design phase and / or adequate testing, watchdogs can reduce the system reliability dramatically. In particular, in the presence of sustained faults, badly-designed watchdog "recovery" mechanisms can cause your system to repeatedly reset itself. **This can be very dangerous.**

☹ Watchdogs with long timeout periods are unsuitable for many applications.

Related patterns and alternative solutions

Please refer to WATCHDOG RECOVERY [this paper] for references to other, general, watchdog patterns.

In this section, some patterns directly related to SCHEDULER WATCHDOG are mentioned.

Other mechanisms for detecting (or avoiding) time-based errors

As noted in "Solution" there are a number of other patterns in the PTTES Collection that can help you satisfy the time constraints described in Equation 1, Equation 2 and Equation 3. For example:

- The processor patterns (STANDARD 8051, SMALL 8051, EXTENDED 8051) allow selection of a processor with performance levels appropriate for the application.
- The oscillator patterns (CRYSTAL OSCILLATOR and CERAMIC RESONATOR) allow an appropriate choice of oscillator type, and oscillator frequency to be made, taking into

account system performance (and, hence, task duration), power-supply requirements, and other relevant factors.

- The various Shared-Clock schedulers (SCC SCHEDULER, SCI SCHEDULER (DATA), SCI SCHEDULER (TICK), SCU SCHEDULER (LOCAL), SCU SCHEDULER (RS-232), SCU SCHEDULER (RS-485)) describe how to schedule tasks on multiple processors, which still maintaining a time-triggered system architecture. Using one of these schedulers as a foundation, the pattern LONG TASK describes how to migrate longer tasks onto another processor without compromising the basic time-triggered architecture.
- LOOP TIMEOUT and HARDWARE TIMEOUT describe the design of timeout mechanisms which may be used to ensure that tasks complete within their allotted time.
- MULTI-STAGE TASK discusses how to split up a long, infrequently-triggered task into a short task, which will be called more frequently. PC LINK (RS232) and LCD CHARACTER PANEL both implement this architecture.
- HYBRID SCHEDULER describes a scheduler that has most of the desirable features of the (pure) co-operative scheduler, but allows a single long (pre-emptible) task to be executed.

Before implementing SCHEDULER WATCHDOG, you should consider whether these patterns meet the needs of your application.

Example: Automotive cruise control.

Use of SCHEDULER WATCHDOG is illustrated in the case study at the end of this paper.

Example: A library for the watchdog timer on the Infineon C515C

A simple code "library" supporting the use of the watchdog timer on the Infineon C515C is presented in Listing 1, Listing 2 and Listing 3.

```
/*-------------------------------------------------------------------*-

  WATCHDOG_C515C_Init()

  This function sets up the watchdog timer.

-*-------------------------------------------------------------------*/
void WATCHDOG_C515C_Init(void)
   {
   // Watchdog timer prescaler (1/16) enabled
   // Watchdog timer reload value is 0x6B
   // Oscillator is 10 MHz -> watchdog period is ~103 ms
   WDTREL = 0xEB;

   // Start watchdog timer
   WDT  = 1;
   SWDT = 1;
   }
```

Listing 1: Part of a small "watchdog" library for the Infineon C515C.

```
/*-------------------------------------------------------------------*-

  WATCHDOG_C515C_Refresh()

  Feed the internal C515C watchdog.

-*-------------------------------------------------------------------*/
void WATCHDOG_C515C_Refresh(void)
   {
   WDT = 1;
   SWDT = 1;
   }
```

Listing 2: Part of a small "watchdog" library for the Infineon C515C.

```
/*-------------------------------------------------------------------*-

  WATCHDOG_C515C_Cause_of_Reset()

  Returns 1 if last reset was caused by watchdog (and clears flag)
  Returns 0 if last reset was "normal".

-*-------------------------------------------------------------------*/
int WATCHDOG_C515C_Cause_of_Reset(void)
   {
   // Determine if reset was caused by watchdog overflow (C515C)
   if (IP0 & 0x40)
      {
      // Clear the IP0 flag
      IP0 &= 0xBF;

      return 1;
      }

   return 0;
   }
```

Listing 3: Part of a small "watchdog" library for the Infineon C515C.

PROGRAM-FLOW WATCHDOG

Context

- You are developing a single-processor embedded application using a member of the 8051 family of microcontrollers (or similar hardware).
- You are programming in C (or a similar language).
- The application has a time-triggered architecture, constructed using a scheduler (e.g. CO-OPERATIVE SCHEDULER [Pont, 2001, page 254]).

Problem

How can you recover from program-flow errors in an embedded processor?

Background

This pattern is concerned with reducing the impact of 'program flow' errors on embedded applications. Such errors can occur as a result of electromagnetic interference.

Arguably, the most serious form of program-flow error in an embedded microcontroller is corruption of the program counter (PC), also known as the instruction pointer[4]. Since the PC of the 8051 is a 16-bit wide register, we make the reasonable assumption that – in response to PC corruption – the PC may take on any value in the range 0 to 65535. In these circumstances, the 8051 processor will fetch and execute the next instruction from the code memory location pointed to by the corrupted PC register. This code memory location may contain:

- program code,
- data constants, or,
- "nothing" (that is, it is unprogrammed, and contains neither code nor meaningful data).

We discuss each of these possibilities in the sections that follow.

Vectoring to program-code locations

Clearly, corruption of the instruction pointer that causes the program flow to be diverted to a "random" address is likely to cause severe side effects. However, the precise impact of such diversions can be very difficult to predict.

[4] The PC is only one of many registers in an embedded processor and there is no evidence to suggest that this particular register is any more or less susceptible to EMI than the others. However, the impact of corruption to the PC is arguably the most serious result of EMI, as it can result in disruption to the program flow.

A particular problem arises because in the 8051 (and many other processors), more than half of the instructions are “multibyte instructions”, such as “POP” and “ACALL” that occupy two or three memory locations, respectively. PC corruption may cause the program flow to be diverted to any of these locations.

To illustrate the nature of the resulting “multibyte instruction trap” (MIT), consider the assembly code shown below:

```
0100 759850    MOV    98H,#050H
0103 438920    ORL    89H,#020H
0106 758DFD    MOV    8DH,#0FDH
```

If the PC is corrupted and takes on the value 0x0101, then the code above will be interpreted as follows:

```
0101 98      SUBB    A,R0
0102 5043    JNC     #43H
0104 8920    MOV     20H,R1
0106 758DFD  MOV     8DH,#0FDH
```

In this example, the first three instructions of the original program code have been misinterpreted while the rest remain unchanged. Of course, the number of instructions that will be misinterpreted depends on the instruction sequence, the corrupted PC value and the state of the processor at the time of PC corruption. In short, the precise impact **is impossible to predict** in most practical situations.

Vectoring to data locations

The problems with misinterpretation of instructions also apply to data values stored in the code area since, to the processor, data constants - such as digital filter coefficients stored in the code area - are indistinguishable from program code.

Again, the results of this are - again - very difficult to predict.

Vectoring to unprogrammed memory locations

Unprogrammed memory locations will usually (by default) have the contents 0xFF, which corresponds to the “MOV R7,A” in the 8051 instruction set (“Copy the contents of the accumulator to register R7”).

In many applications, the program code will occupy the lower code memory addresses, and the remainder of the memory will be unprogrammed. In these circumstances the processor will execute “MOV R7,A” instructions until the PC reaches the end of the physical code memory. The processor will then continue executing program code at location 0x0000. This can have an impact similar to a processor reset[5].

[5] It is important to appreciate that this is NOT the same as a processor reset. For example, when the processor is reset, register values take on well-defined values: this has important implications for several aspects of system behaviour, such as initial port settings.

In other applications there may be unprogrammed "gaps" in the memory maps, followed by constant data or program code. Execution of this code (or data, treated as code) is likely to have less predictable side effects, as we discussed earlier.

Solution

The technique we discuss here has previously been described and assessed in a number of studies (Campbell, 1995; Campbell, 1988; Niaussat, 1998; Ong and Pont, 2001; Ong *et al.*, 2001; Ong and Pont, 2002).

An overview of the approach

The most straightforward implementation of PROGRAM-FLOW WATCHDOG involves two stages:

- We fill unused locations at the end of the program code memory with single-byte "No Operation" (NOP), or equivalent, instructions.
- We place a "PC Error Handler" (PCEH) at the end of code memory to deal with the error[6].

The operation of Program-Flow Watchdogs may be easily predicted. When an PC error occurs **and the PC points to a memory location within the Program-Flow Watchdog area**, the processor will repeatedly execute NOP instructions until the PC points to the start of the PCEH. The error handler will then carry out its intended recovery function.

Dealing with errors

Here, we will assume that the PCEH will consist mainly of a loop:

```
// Force watchdog timeout
while(1)
   ;
```

This means that, as discussed in WATCHDOG RECOVERY [this paper] the watchdog timer will force a clean system reset.

Please note that, as also discussed in WATCHDOG RECOVERY, we may be able to reduce the time taken to reset the processor by adapting the watchdog timing. For example:

```
// Set up the watchdog for "normal" use
// - overflow period = ~39 ms
WDTREL = 0x00;

...


// Adjust watchdog timer for faster reset
// - overflow set to ~300 µs
WDTREL = 0x7F;

// Now force watchdog-induced reset
while(1)
```

6 Note that, except in the event of PC corruption, the PCEH is unreachable.

;

After the watchdog-induced reset, we need to implement a suitable recovery strategy. A range of different options are discussed in RESET RECOVERY [this paper], FAIL-SILENT RECOVERY [this paper] and LIMP-HOME RECOVERY [this paper].

Hardware resource implications

As noted above, PROGRAM-FLOW WATCHDOG can only be guaranteed to work where the corrupted PC points to an "empty" memory location. Maximum effectiveness will therefore be obtained with comparatively small programs (a few kilobytes of code memory), and larger areas of empty memory.

If devices with less than 64kB of code memory are used, a problem known as "memory aliasing" can occur (see Figure 2).

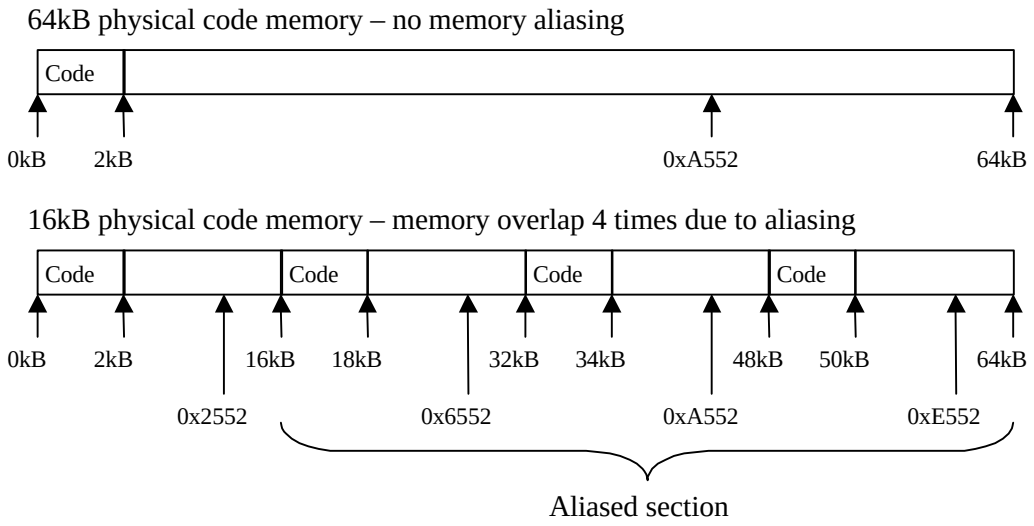


Figure 2: Problems caused by memory aliasing. See text for details.

Figure 2 shows a 2kB program on microcontrollers with 16kB and 64kB of physical code memory. In this example, the PC is addressing location 0xA552 on the 64kB device. **In the 16kB device, the lack of the upper two address lines means that addresses 0x2552, 0x6552, 0xA552 and 0xE552 will all address the same physical code location.**

Memory aliasing has an important impact on the effectiveness of Program-Flow Watchdog. For example, suppose your program code is 2 kB in size:

- If you use a 64 kB code memory to store your program, you are likely to trap (on average) 62/64 * 100% (= 96.9%) of program-flow errors.
- If you use a 4 kB code memory to store your program, you are likely to trap (on average) 2/4
  * 100% (= 50%) of program-flow errors.

The implications are clear. If you want to increase the chances of detecting program-flow errors using this approach, you need to use the maximum amount of (code) memory that is supported by your processor. In the case of the 8051 family, this generally means selecting a device with 64 kB of memory. Clearly, this choice will have cost implications.

Reliability and safety implications

Use of PROGRAM-FLOW WATCHDOG may help to improve reliability of your system in the presence of program-flow errors (which may, in turn, result from EMI).

Under normal conditions, neither the filling of unused memory locations with NOP instructions, nor the addition of an error handler will have any impact on your program.

> Please note, however, that - if a watchdog timer is used as part of your error-recovery strategy - you need to ensure that you have a full understanding of the implications this can have for the reliability of your system. Please see WATCHDOG RECOVERY [this paper] for further details.

Portability

The technique used in this pattern is applicable with any microcontroller family (however, the particular instructions used for the "NOP" behaviour will - obviously - need to match the hardware).

Overall strengths and weaknesses

☺ A low-cost technique that can be effective in the presence of program-flow errors.

☹ For maximum effectiveness, a significant amount of "empty" code memory is required.

Related patterns and alternative solutions

We consider a number of alternative solutions and related patterns in this section.

Speeding up the response

We stated in "Solution" that the most straightforward implementation of PROGRAM-FLOW WATCHDOG involves two stages:

- We fill unused locations at the end of the program code memory with single-byte "No Operation" (NOP), or equivalent, instructions.
- Second, a small amount of program code, in the form of an "PC Error Handler" (PCEH), is placed at the end of code memory to deal with the error.

Suppose that we implement this solution, and that our microcontroller therefore has a large amount of ROM, filled with NOP instructions. Further suppose that a program-flow error throws the PC to the start of this NOP area. It may then take an appreciable period of time for the processor to reach the error handler. In addition, the time taken to recover from an error is highly variable (since it depends on the value of the corrupted PC).

An alternative is to fill the memory not with "NOP" instructions but with "jump" instructions. In effect, we want to fill each location with "Jump to address X" instructions, and then place the error handler at address X.

In practice, such a jump instruction will occupy more than one byte, but this problem is not insurmountable. In the 8051, the simplest implementation is to fill the empty memory with "long jump" instructions (0x02). As a result (almost) every time the PC lands in this area, the processor will execute the instruction: "Jump to 0x0202". The error handler will then be located at address 0x0202.

We give an example of this process below.

The recovery operation

A range of suitable recovery strategies are discussed in RESET RECOVERY [this paper], FAIL-SILENT RECOVERY [this paper] and LIMP-HOME RECOVERY [this paper].

Hardware-based alternatives

To deal with EMI-related problems, hardware solutions, including device shielding, wiring screening and input/output filtering are widely used. However, hardware solutions are expensive, can suffer physical damage, and – in applications such as Hall-effect sensors – can interfere with normal device operation.

Despite this, in most cases, it does not make sense to abandon hardware protection completely. Software-based techniques can be effective as an adjunct to hardware-based techniques.

Example: Automotive cruise control.

Use of PROGRAM-FLOW WATCHDOG is illustrated in the case study at the end of this paper.

Example: Implementing Program-Flow Watchdog (NOP fill)

We summarise how to implement a basic PROGRAM-FLOW WATCHDOG on the 8051 microcontroller, using the Keil compiler, below:

1. Compile and link the program, as normal, with an error handler.
2. From the .M51 file, determine the length of the error function (e.g. 45 bytes, 0x2D bytes).
3. Determine the size of the code memory you will use (e.g. 0x2000 = 8K); ideally, this will be 64 kbytes.
4. Subtract the size of the error function from the code-memory size (e.g. 0x2000 - 0x2D = 0x1FD3)
5. Use the compiler / linker options to move the error handler to this location.
6. EITHER: Use your device programmer to fill the memory with NOP instructions.
   OR: Use the .M51 file to determine the required size, and use the startup.A51 file to set the values to "NOP".
7. Re-compile and link the code, and program the chip.

Example: Implementing Program-Flow Watchdog (Jump version)

We summarise how to implement a "jump" version of PROGRAM-FLOW WATCHDOG on the 8051 microcontroller, using the Keil compiler, below:

1. Write the program (including error handler).
2. Use the compiler / linker options to move the error handler to location 0x0202.
3. Use your programmer to fill the memory with 0x02 instructions - or use the .M51 file (again) to determine the required size, and use the startup.A51 file to set the values to "0x02".
4. Compile and link the code, and program the chip.

RESET RECOVERY

Context

- You are developing a single-processor embedded application using a member of the 8051 family of microcontrollers (or similar hardware).
- You are programming in C (or a similar language).
- The application has a time-triggered architecture, constructed using a scheduler (e.g. CO-OPERATIVE SCHEDULER [Pont, 2001, page 254]).

and:

- You are using techniques described in WATCHDOG RECOVERY [this paper] - or similar approaches - in order to improve the fault-tolerance of your system.
- A watchdog-induced reset has occurred.

Problem

How can you ensure that your processor re-starts safely after a watchdog-induced reset?

Background

Please see WATCHDOG RECOVERY [this paper] for background information on watchdog timers.

Solution

As we discussed in WATCHDOG RECOVERY, all of the error-recovery strategies discussed in this paper begin with a system reset, which has been caused by the overflow of a watchdog timer.

What are we trying to achieve?

Using RESET RECOVERY we assume that the best way to deal with an error (the presence of which is indicated by a watchdog-induced reset) is to re-start the system, in its normal configuration.

Implementation

RESET RECOVERY is very to easy to implement. We require a basic watchdog timer, such as the common "1232" external device, available from various manufacturers (we show how to use this device in an example below).

Using such a device, the cause of a system reset cannot be easily determined. However, this does not present a problem when implementing RESET RECOVERY. After any reset, we simply start (or re-start) the scheduler and **try** to carry out the normal system operations.

Hardware resource implications

As noted in "Solution", it is not necessary to distinguish between a 'normal' system reset, and a reset caused by a watchdog overflow when implementing **RESET RECOVERY**. One consequence of this is that any type of watchdog hardware (internal or external) can be used.

Reliability and safety implications

In safety-related or safety-critical systems, this pattern should not be implemented before a complete risk-assessment study has been conducted (by suitably-qualified individuals).

Successful use of this pattern requires a full understanding of the errors that are likely to be detected by your error-detection strategies (and those that will be missed), plus an equal understanding of the recovery strategy that you have chosen to implement. Without a complete investigation of these issues, you cannot be sure that implementation of the pattern you will increase (rather than decrease) the reliability of your application. Please see WATCHDOG RECOVERY [this paper] for further discussion of the reliability and safety implications associated with watchdog timers.

The particular problem with RESET RECOVERY is that, if the error that gave rise to the watchdog reset is permanent (or long-lived), then you are likely to lose control of your system as it enters an endless loop (reset, watchdog overflow, reset, watchdog overflow, …).

This lack of control can have disastrous consequences in many systems.

Portability

This approach can be used with any processor or microcontroller.

Overall strengths and weaknesses

☺ Very easy to implement.

☹ MUST BE HANDLED WITH EXTREME CARE (see "Reliability and safety issues").

Related patterns and alternative solutions

Two alternative recovery strategies are discussed in FAIL-SILENT RECOVERY [this paper] and LIMP-HOME RECOVERY [this paper].

Example: A library for the '1232' external watchdog timer

In this example we present a very simple library which will allow the use of an external '1232' watchdog chip.

The use of the 1232 is very straightforward:

- We wire up the watchdog to the microcontroller reset pin, as illustrated in Figure 3.

- We choose from one of three (nominal) possible timeout periods, and connect the TD pin on the 1232 to select an appropriate period (see Table 1).
- We pulse the /ST0 line on the 1232 regularly, with a pulse interval less than the timeout period.

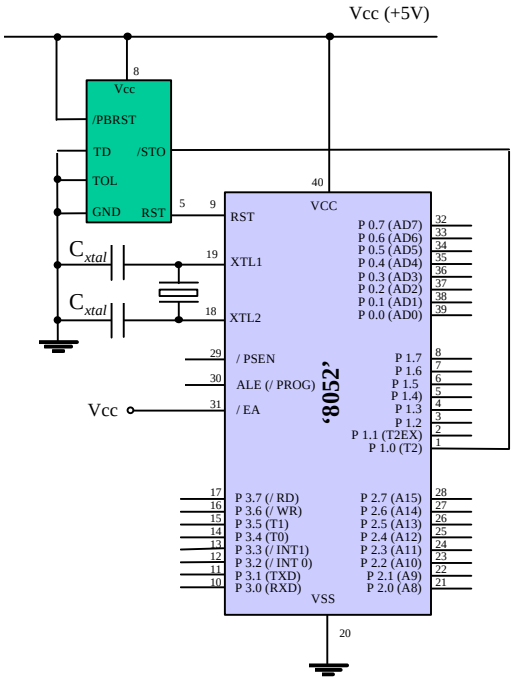


Figure 3: Simple demonstration circuit for 1232 watchdog.

	Minimum timeout	Typical timeout	Maximum timeout
TD to GND	62.5 ms	150 ms	250 ms
TD floating	250 ms	600 ms	1000 ms
TD to Vcc	500 ms	1200 ms	2000 ms

Table 1: Timings for the ubiquitous '1232; watchdog.

The key part of a suitable code library is given in Listing 4.

```
/*-------------------------------------------------------------------*-

   Dog_1232.C

  ------------------------------------------------------------------

   Watchdog timer library for external 1232 WD.

-*-------------------------------------------------------------------*/

#include "Dog_1232.h"

// ------ Port pins -----------------------------------------------

// Connect 1232 (pin /ST) to the WATCHDOG_pin
sbit WATCHDOG_pin = P1^0;
```

```
// ------ Private variables ---------------------------------------

// Current state of the watchdog pin
static bit WATCHDOG_state_G = 0;

/*---------------------------------------------------------------------*-

  WATCHDOG_Feed()

  'Feed' the external 1232-type watchdog chip.

-*---------------------------------------------------------------------*/
void WATCHDOG_Feed(void)
   {
   // Change the state of the watchdog pin
   if (WATCHDOG_state_G == 1)
      {
      WATCHDOG_state_G = 0;
      WATCHDOG_pin = 0;
      }
   else
      {
      WATCHDOG_state_G = 1;
      WATCHDOG_pin = 1;
      }
   }

/*---------------------------------------------------------------------*-
  ---- END OF FILE ------------------------------------------------
-*---------------------------------------------------------------------*/
```

Listing 4: Part of a central heating demo using 'Super Loop' and 'Hardware Watchdog'.

As noted above, most TTCS applications will use this library by feeding the watchdog at the start of the dispatcher function, as follows:

```
/*---------------------------------------------------------------------*-

  SCH_Dispatch_Tasks()

  This is the 'dispatcher' function.  When a task (function)
  is due to run, SCH_Dispatch_Tasks() will run it.
  This function must be called (repeatedly) from the main loop.

-*---------------------------------------------------------------------*/
void SCH_Dispatch_Tasks(void)
   {
   tByte Index;

   // Feed the watchdog here
   WATCHDOG_Feed();

   // Dispatches (runs) the next task (if one is ready)
   for (Index = 0; Index < SCH_MAX_TASKS; Index++)
      {
      ...
```

FAIL-SILENT RECOVERY

Context

- You are developing a single-processor embedded application using a member of the 8051 family of microcontrollers (or similar hardware).
- You are programming in C (or a similar language).
- The application has a time-triggered architecture, constructed using a scheduler (e.g. CO-OPERATIVE SCHEDULER [Pont, 2001, page 254]).

and:

- You are using techniques described in WATCHDOG RECOVERY [this paper] - or similar approaches - to improve the fault-tolerance of your system.
- A watchdog-induced reset has occurred.

and:

- Simply re-starting the system in the event of an error (as discussed in RESET RECOVERY [this paper]) is not an appropriate response, since there is a significant risk that the error is either "permanent", or that it will re-occur. This could leave your system stuck, out of control, in an endless "reset, watchdog overflow, reset, watchdog overflow, …" loop.

Problem

How can you ensure that your processor re-starts safely after a watchdog-induced reset?

Background

Please see WATCHDOG RECOVERY [this paper] for background information on watchdog timers.

Solution

When using Fail-Silent Watchdog, our aim is to shut the system down after a watchdog-induced reset. This type of response is referred to as "fail silent" behaviour because the processor becomes "silent" in the event of an error[7].

As indicated in "Context", we assume that simply re-starting the system in the event of an error (as discussed in RESET RECOVERY [this paper]) is not an appropriate response, since there is a significant risk that the error is either "permanent", or that it will re-occur. We also assume

[7] This type of behaviour is often contrasted with what is known as "babbling idiot" failure (particularly in multi-processor systems), where a damaged or faulty processor remains active (and continues to interfere with the rest of the system, particularly by transmitting "noise" to other nodes). For example, use of RESET RECOVERY in the presence of sustained faults can give rise to such a "babbling idiot".

that "freezing" the system is a known (safe) state in the event of an error will increase (rather than decrease) the reliability of our system.

Software architecture

FAIL-SILENT RECOVERY is implemented after every "Normal" reset as follows:

- The scheduler is started and program execution is normal.

By contrast, after a watchdog-induced reset, FAIL-SILENT RECOVERY will typically be implemented as follows:

- Any necessary port pins will be set to appropriate levels (for example, levels which will shut down any attached machinery).
- Where required, an error port will be set to report the cause of the error,
- All interrupts will be disabled, and,
- The system will be stopped, either by entering an endless loop or (preferably) by entering power-down or idle mode.

This effectively freezes the processor, in a known - safe - state.

Please note that power-down or idle mode is used because, in the event that the problems were caused by EMI or ESD, this is thought likely to make the system more robust in the event of another interference burst.

Hardware resource implications

The main resource implication is that a suitable watchdog timer is required.

Reliability and safety implications

In safety-related or safety-critical systems, this pattern should not be implemented before a complete risk-assessment study has been conducted (by suitably-qualified individuals).

Successful use of this pattern requires a full understanding of the errors that are likely to be detected by your error-detection strategies (and those that will be missed), plus an equal understanding of the recovery strategy that you have chosen to implement. Without a complete investigation of these issues, you cannot be sure that implementation of the pattern you will increase (rather than decrease) the reliability of your application.

Please see WATCHDOG RECOVERY [this paper] for further discussion of the reliability and safety implications associated with watchdog timers.

Portability

The main constraint on portability is that the watchdog timer used to implement this pattern must allow the programmer to determine the cause of a reset.

Overall strengths and weaknesses

☺ If implemented in appropriate circumstances and with care, this pattern can help to improve the reliability of your system.

☹ Increases the system complexity.

Related patterns and alternative solutions

Two alternative recovery strategies are discussed in RESET RECOVERY [this paper] and LIMP-HOME RECOVERY [this paper].

Example: Automotive cruise control.

Use of FAIL-SILENT RECOVERY is illustrated in the case study at the end of this paper.

Example: Fail-Silent behaviour in the Airbus A310

In the A310 Airbus, the slat and flap control computers form an 'intelligent' actuator sub-system. If an error is detected during landing, the wings are set to a safe state and then the actuator sub-system shuts itself down (Burns and Wellings, 1997, p.102).

Please note that the mechanisms underlying this "fail silent" behaviour are unknown: they may not be the same as the techniques described in this paper.

Example: Fail-Silent behaviour in a steer-by-wire application

Suppose that an automotive steer-by-wire system has been created that runs a single task, every 10 ms. We will assume that the system is being monitored to check for task over-runs (see SCHEDULER WATCHDOG [this paper]). We will also assume that the system has been well designed, and has appropriate timeout code, etc, implemented.

Further suppose that a passenger car using this system is being driven on a motorway, and that an error is detected, resulting in a watchdog reset. What recovery behaviour should be implemented?

We could simply re-start the scheduler and "hope for the best". However, this form of "reset recovery" is probably not appropriate. In this case, if we simply perform a reset, we may leave the driver without control of their vehicle (see RESET RECOVERY [this paper]).

Instead, we could implement a fail-silent strategy. In this case, we would simply aim to bring the vehicle, slowly, to a halt. To warn other road vehicles that there was a problem, we could choose to flash all the lights on the vehicle on an off (continuously), and to pulse the horn. This strategy (which may - in fact - be far from silent) is not ideal, because there can be no guarantee that the driver and passengers (or other road vehicles) will survive the incident. However, it the event of a very serious system failure, it may be all that we can do.

LIMP-HOME RECOVERY

Context

- You are developing a single-processor embedded application using a member of the 8051 family of microcontrollers (or similar hardware).
- You are programming in C (or a similar language).
- The application has a time-triggered architecture, constructed using a scheduler (e.g. CO-OPERATIVE SCHEDULER [Pont, 2001, page 254]).

and:

- You are using techniques described in WATCHDOG RECOVERY [this paper] - or similar approaches - to improve the fault-tolerance of your system.
- A watchdog-induced reset has occurred.

and:

- Simply re-starting the system in the event of an error (as discussed in RESET RECOVERY [this paper]) is not an appropriate response, since there is a significant risk that the error is either "permanent", or that it will re-occur. This could leave your system stuck, out of control, in an endless "reset, watchdog overflow, reset, watchdog overflow, …" loop.
- "Freezing" the system in a known state in the event of an error (as discussed in FAIL-SILENT RECOVERY [this paper]) is not appropriate behaviour, as it is likely to decrease (rather than increase) the reliability of our system. For example, completely shutting down a piece of essential medical equipment is something we would wish to do only as a last resort. Similarly, shutting down an aircraft control system during takeoff is (highly) undesirable.

Problem

How can you ensure that your processor re-starts safely after a watchdog-induced reset?

Background

Please see WATCHDOG RECOVERY [this paper] for background information on watchdog timers.

Solution

As we discussed in WATCHDOG RECOVERY, all of the error-recovery strategies presented in this paper begin with a system reset, which has been caused by the overflow of a watchdog timer.

What are we trying to achieve?

In using LIMP-HOME RECOVERY, we make four assumptions about our system:

- A watchdog-induced reset indicates that a significant error has occurred.

- Although a full (normal) re-start is considered too risky, it may still be possible to let the system "limp home" by running a simple version of the original algorithm.

Overall, in using this pattern, we are looking for ways of ensuring that the system continues to function - even in a very limited way - in the event of an error.

Software architecture

LIMP-HOME RECOVERY is implemented after ever "Normal" reset as follows:

- The scheduler is started and program execution is normal.

By contrast, after a watchdog-induced reset, LIMP-HOME RECOVERY will typically be implemented as follows:

- The scheduler will not be started.
- A simple version of the original algorithm will be executed.

Hardware resource implications

At the processor level, the main resource implication is that a suitable watchdog timer is required.

However (rather more substantial) hardware costs may also arise if "backup" hardware (such as sensors or actuators) is required in the limp-home system.

Reliability and safety implications

In safety-related or safety-critical systems, this pattern should not be implemented before a complete risk-assessment study has been conducted (by suitably-qualified individuals).

Successful use of this pattern requires a full understanding of the errors that are likely to be detected by your error-detection strategies (and those that will be missed), plus an equal understanding of the recovery strategy that you have chosen to implement. Without a complete investigation of these issues, you cannot be sure that implementation of the pattern you will increase (rather than decrease) the reliability of your application.

Please see WATCHDOG RECOVERY [this paper] for further discussion of the reliability and safety implications associated with watchdog timers.

Portability

The main constraint on portability is that the watchdog timer used to implement this pattern must allow the programmer to determine the cause of a reset.

Overall strengths and weaknesses

☺ If implemented in appropriate circumstances and with care, this pattern can help to improve the reliability of your system.

☹ Increases the system complexity.

Related patterns and alternative solutions

Two alternative recovery strategies are discussed in RESET RECOVERY [this paper] and FAIL-SILENT RECOVERY [this paper].

Some alternative patterns for fault tolerance and error recovery which may be of interested were presented recently by Saridakis (2002).

Example: Limp-home behaviour in a steer-by-wire application

In FAIL-SILENT RECOVERY [this paper], we considered one possible recovery strategy in a steer-by-sire application.

As an alternative to the approach discussed in the previous example, we may wish to consider a limp-home control strategy. In this case, a suitable strategy might involve a code structure like this:

```
while(1)
   {
   Update_basic_steering_control();
   Software_delay_10ms();
   }
```

This is a basic software architecture (based on SUPER LOOP [Pont, 2001, p.162]).

In creating this version, we have avoided use of the scheduler code. We might also wish to use a different (simpler) control algorithm at the heart of this system. For example, the main control algorithm may use measurements of the current speed, in order to ensure a smooth response even when the vehicle is moving rapidly. We could omit this feature in the "limp home" version.

Of course, simply using a different software implementation may still not be enough. For example, in our steer-by-wire application, we may have a position sensor (attached to the steering column) and an appropriate form of DC motor (attached to the steering rack). Both the sensor and the actuator would then be linked to the processor.

When designing the limp-home controller, we would like to have an additional sensor and actuator, which are - as far as possible - independent of the components used in the main (scheduled) system. This option makes sense because it is likely to maximise the chances that the Slave node will operate correctly when it takes over (for example, it could have been a failure of the DC motor that made it necessary to shut down the original processing).

This approach has two main implications:

1. The hardware **must** 'fail silently': for example, if we did add a backup motor to the steering rack, this would be little use if the main motor 'seized' when the scheduler task was shut down.

   Note that there may be costs associated with obtaining this behaviour. For example, we may need to add some kind of clutch assembly to the motor output, to ensure that it could be disconnected in the event of a motor jam. However, such a decision would need to be made only after a full risk assessment. For example, it would not make sense to add a clutch unit if a failure of this unit (leading to a loss of control of steering) was more likely than a motor seizure.

2. The cost of hardware duplication can be significant, and will often be considerably higher than the cost of a duplicated processor: this may make this approach economically unfeasible.

   When costs are too high, sometimes a compromise can prove effective. For example, in the steering system, we might consider adding a second set of windings to the motor for use by the Slave (rather than adding a complete new motor assembly). Again, such a decision should be made only after a full risk assessment.

OSCILLATOR WATCHDOG

Context

- You are developing a single-processor embedded application a member of the 8051 family of microcontrollers (or similar hardware).
- You are designing an appropriate hardware foundation for your application.

Problem

How do you detect the failure of the oscillator in your embedded application (and what should you do in these circumstances)?

Background

What is a watchdog timer

To understand this pattern, you'll need to understand what a watchdog timer is: WATCHDOG RECOVERY [this paper] provides the necessary background.

If we have watchdog timers, why do we need oscillator watchdogs?

People sometimes assume that watchdog timer is a good way of detecting oscillator failure. However, a few moments thought quickly reveals that this is very rarely the case.

> When the oscillator fails, the associated microcontroller will stop. **Even if (by using a watchdog timer, or some other technique) you detect that the oscillator has failed, you cannot execute any code to deal with the situation.**

In these circumstances, you may be able to improve the reliability of your system by using an *oscillator watchdog*.

Solution

Software-based techniques can be used to solve many problems in embedded applications which are traditionally handled by adding hardware: for example, switch debouncing can be carried out using external hardware components or through software (see SWITCH INTERFACE (SOFTWARE) [Pont, 2001, page 399]). In general, where it is possible to use software, this results in a more flexible and lower-cost solution.

In some cases, software is not an option and hardware **is** required: dealing with oscillator failure is such a case. Specifically, to implement Oscillator Watchdog, you need to select a microcontroller with on-chip 'oscillator watchdog' hardware.

Oscillator-watchdog hardware is not part of the 8051 core, and implementations vary slightly. However, the behaviour is always the same: if an oscillator failure is detected, the microcontroller is forced into a reset state: **this means that port pins take on their reset values**.

The state of the port pins can be crucial, since it means that the developer has a better chance of ensuring that hardware devices controlled by the processor (for example, dangerous machinery) will be shut down if the processor's oscillator fails. Please note that - as discussed in PORT I/O [Pont, 2001, page 174] - the port reset values must be taken into account when making use of an oscillator watchdog: failure to do so will render the use of such a watchdog meaningless.

In most cases, the microcontroller will be held in a reset state "for ever". However, most oscillator watchdogs will continue to monitor the clock input to the chip: if the main oscillator is restored, the system will leave reset and will begin operating again.

Hardware resource implications

Use of an oscillator watchdog requires no hardware resources (apart, of course, from the watchdog hardware itself).

Reliability and safety implications

Quartz-based oscillators

As discussed in CRYSTAL OSCILLATOR [Pont, 2001, page 54], quartz crystals form the basis of almost all stable oscillator circuits. A common source of failure for such components is physical damage (for example, through sustained vibration or from a sudden sharp impact). Use of OSCILLATOR WATCHDOG can be particularly effective in systems employing crystal oscillators where physical damage can occur.

In addition to their fragility, crystal oscillators also have one other feature: they can take a comparatively long time to start up (up to around 10 ms). Until this oscillator starts, most 8051 devices are "in limbo: they cannot enter their normal reset state - and the ports will be at an undefined value. 10 ms may seem a long period of time if high-power machinery is connected to a port pin.

Use of an oscillator watchdog can assist in these circumstances too. In order to drive the processor into a reset state, the oscillator watchdog needs to contain its own oscillator. This will usually be a low-frequency (and low stability) RC oscillator. Under normal circumstances, the "watchdog" behaviour will be invoked if the frequency of the main oscillator is lower than that of the RC device.

RC oscillators usually have a very rapid start time. As a result, when an oscillator watchdog is available, the RC oscillator will start first and will (with many microcontrollers) very rapidly detect that the main oscillator is not operating. The oscillator watchdog will then force the

system into a reset state. The system will only leave this state when the main oscillator has become active.

The impact of such an oscillator can be very significant. For example, the time to reach a reset state in a standard 8051 can be around 10 ms. Using an Infineon C515C, with oscillator watchdog, the maximum time to reach reset state becomes 34 µs. This is a very significant speed improvement (approximately 300x faster).

The limitations of single-processor designs

Use of an oscillator watchdog will simply leave your system "frozen", albeit in a well-defined state. This is much better than leaving the system in an indeterminate state. However, it may not be enough.

For example, suppose your system is controlling the brakes or steering in a moving vehicle. In these circumstances "freezing" the system may be highly undesirable. Instead, you may wish to switch in a 'backup' microcontroller, in order to try and return control of the vehicle to the driver. WATCHDOG SLAVE[8] describes one way in which you can achieve this.

Portability

Oscillator Watchdog hardware is available in only a comparatively small number of microcontrollers. Assuming the presence of such hardware will restrict the portability of your design.

Overall strengths and weaknesses

☺ Can improve reliability in situations where oscillator failure occurs (for example, due to system vibration).

☹ Can only be used where processors have appropriate hardware support.

Related patterns and alternative solutions

Please see "Reliability and Safety Implications" for information about WATCHDOG SLAVE.

Example: Automotive cruise control.

Use of OSCILLATOR WATCHDOG is illustrated in the case study at the end of this paper.

Example: The Philips P87LPC760 family

The Philips P87LPC760 family of (8051) microcontrollers have on-chip oscillator watchdogs. Please refer to the data sheets for these devices for further details.

Example: The Infineon C868 family

The Infineon C868 (8051) microcontrollers have on-chip oscillator watchdogs. Please refer to the data sheets for these devices for further details.

---

8 WATCHDOG SLAVE is a based on the "shared-clock scheduler" architecture (see Pont, 2001, Part F). The pattern is still under development, and details will be released at a future PLoP conference.

CASE STUDY: Implementing an automotive cruise-control system

We consider the use of many of the patterns presented in this paper in an automotive cruise-control system (CCS)[9]. The CCS will (it is assumed) be required to take over the task of maintaining the vehicle at a constant speed even while negotiating a varying terrain involving, for example, hills or corners in the road.

The CCS will be implemented using a single Infineon C515C microcontroller (an example of an EXTENDED 8051 [Pont, 2001, p.46]).

To operate the CCS, the driver will do the following:

- Enter the required speed on a rotary speed-control dial[10]. This will be read, via a potentiometer and analogue-to-digital converter: see ONE-SHOT ADC [Pont, 2001, p.757].
- Press a "cruise" switch: see SWITCH INTERFACE (SOFTWARE) [Pont, 2001, p.399].

The CCS will stop cruising "on demand" (and return control to the driver), if:

- The driver presses the cruise switch again, or,
- The driver touches the brake pedal (which is also assumed to have a switch attached).

While cruising, the CCS will then repeatedly perform the following tasks:

- It will check the status of the brake pedal and the cruise switch, to ensure that the driver wishes to keep cruising: if not, it will return control to the driver.
- It will check for system errors (see below): if an error is detected, it will "fail silently" and return control to the driver.
- It will read the desired cruise speed (entered by the driver).
- It will measure the current vehicle speed by counting pulses from magnetic sensors attached to two wheels (see HARDWARE PULSE COUNT [Pont, 2001, p.728]).
- It will use PID CONTROLLER [Pont, 2001, p.861] to determine the required throttle setting.
- It will use an appropriate hardware interface to control the throttle (possibly using HARDWARE PWM [Pont, 2001, p.808] and MOSFET DRIVER [Pont, 2001, p.139]).

The error-handling functions will use many of the patterns presented in this paper:

- WATCHDOG RECOVERY [this paper] will provide the basic framework for the error handling.
- SCHEDULER WATCHDOG [this paper] will provide a means of detecting scheduling errors.
- PROGRAM-FLOW WATCHDOG [this paper] may be appropriate here. Program-flow errors can occur as a result of electromagnetic interference, and the typical passenger car contains numerous electromechanical devices such as contact breakers, alternators, relays and

9 This application is builds on the simple PID-based CCS described in PID CONTROLLER [Pont, 2001, p.861].

10 Note that the driver is assumed to be able to alter the cruise speed while cruising, by means of this dial.

ignition coils that are excellent sources of high-energy, wideband, electromagnetic noise that is capable of corrupting electronic circuits.

- OSCILLATOR WATCHDOG [this paper] will provide a means of dealing with oscillator failures.
- We need to ensure that the speed sensors, throttle actuator, speed-control dial, cruise switch and brake switch all operate correctly while cruising. If any of these devices fail then, using the approach discussed in WATCHDOG RECOVERY [this paper], we will force a watchdog reset.

  There are various checks we can perform. For example: (1) the speed sensors can be tested by looking for sudden (impossibly rapid) changes in speed, and checking that neither sensor reads zero while cruising. (2) The throttle actuators can be checked by including a sensor to detect that the throttle setting actually changes on demand. (3) The speed-control dial can employ a dual-gang potentiometer to provide two measures of the desired speed: if they don't agree, or are out of range, we assume there is an error. (4) The switches can be "doubled up", so that both a "normally open" and "normally closed" switch is used in each case: see SWITCH INTERFACE (SOFTWARE) [Pont, 2001, p.399] for further details.
- In terms of recovery behaviour, we assume that the driver is able to take over control of the vehicle in the event of an error. This means that neither RESET RECOVERY [this paper] nor LIMP-HOME RECOVERY [this paper] will be appropriate in this system and that, instead FAIL-SILENT RECOVERY [this paper] will be used.

The basic system architecture will be based on CO-OPERATIVE SCHEDULER [Pont, 2001, p.255]: the core of the resulting main function is shown in Listing 5.

```
void main(void)
   {
   // Determine cause of system reset
   // (*** IF RESET CAUSED BY WATCHDOG, THEN FAIL SILENTLY HERE ***)
   CCS_Check_Cause_Of_Reset();

   // Set up the scheduler
   SCH_Init_T2();

   // Prepare the CCS
   CCS_Init();

   // Add the tasks - TIMING IS IN TICKS (10 ms tick interval)
   SCH_Add_Task(CCS_Read_Cruise_and_Brake_Switches, 1, 2);
   SCH_Add_Task(CCS_Get_Required_Speed, 2, 10);
   SCH_Add_Task(CCS_Calculate_Current_Speed, 3, 10);
   SCH_Add_Task(CCS_Compute_and_Apply_Throttle, 4, 10);

   SCH_Start();

   while(1)
      {
      SCH_Dispatch_Tasks();
      }
   }
```

Listing 5: The function `main()` *for the CCS demo.*

At the start of the program, the function `CCS_Check_Cause_Of_Reset()` is called: this function is key to the watchdog-based error handling. A possible implementation of this function (assuming the use of Infineon C515C hardware and the Keil compiler) is given in Listing 6.

```
void CCS_Check_Cause_Of_Reset(void)
   {
   // Determine if reset was caused by watchdog overflow (C515C)
   if (IP0 & 0x40)
      {
      // WDTS flag is set - reset *was* caused by watchdog
      Watchdog_reset_G = 1;

      // We shut the system down here - fail silently

      // Try to tell the driver
      CCS_Beep()

      // Disable any / all interrupts
      EA = 0;

      // Enter power-down mode
      PCON   |= 0x02;
      PCON   |= 0x40;

      // Safety net - just in case we are thrown out of power-down
      // (Should NEVER be needed)
      while (1);
      }

   // ------------------------------------------------
   // If we get this far, then this was a "normal" reset
   // - carry on and start the scheduler, etc.
   // ------------------------------------------------
   }
```

Listing 6: A function for handling watchdog-related errors in the CCS system. See text for details.

References and further reading

Burns, A. and Wellings, A. (1997) *"Real-time systems and programming languages"*, Addison-Wesley.

Campbell, D. (1995) "*Designing for Electromagnetic compatibility with Single-Chip Microcontrollers*", Motorola Application Note AN1263.

Campbell, D. (1998) "Defensive Software Programming with Embedded Microcontrollers", IEE Colloquium on Electromagnetic Compatibility of Software, Birmingham, UK, Nov 1998 (Conference code 98/471).

Douglass, B.P. (1999) *"Doing hard time: Developing real-time system with UML, objects, frameworks, and patterns"*, Addison-Wesley. ISBN: 0-201-498375.

Niaussat, A. (1998) "*Software techniques for improving ST6 EMC performance*", ST Application Note AN1015/0398.

Ong, H.L.R and Pont, M.J. (2001) "Empirical comparison of software-based error detection and correction techniques for embedded systems", Proceedings of the 9th International Symposium on Hardware / Software Codesign, April 25-27 2001, Copenhagen, Denmark. Pp.230-235. Published by ACM Press, New York. ISBN: 1-58113-364-2.

Ong, H.L.R and Pont, M.J. (2002) "The impact of instruction pointer corruption on program flow: a computational modelling study", *Microprocessors and Microsystems*, **25**: 409-419.

Ong, H.L.R, Pont, M.J. and Peasgood, W. (2001) "Do software-based techniques increase the reliability of embedded applications in the presence of EMI?" Microprocessors and Microsystems, **24**: 481-491.

Pont, M.J. (2001) *"Patterns for time-triggered embedded systems: Building reliable applications with the 8051 family of microcontrollers"*, Addison-Wesley / ACM Press. ISBN: 0-201-331381.

Pont, M.J. (2002) *"Embedded C"*, Addison-Wesley. ISBN: 0-201-79523X.

Saridakis, T. (2002) "A system of patterns for fault tolerance", paper presented at EuroPloP 2002, July 2002, Germany.

Appendix:
The PTTES Collection

A complete list of the patterns in the PTTES collection is given in Table 1.

STANDARD 8051	SMALL 8051	EXTENDED 8051
CRYSTAL OSCILLATOR	CERAMIC OSCILLATOR	RC RESET
ROBUST RESET	ON-CHIP MEMORY	OFF-CHIP DATA MEMORY
OFF-CHIP CODE MEMORY	NAKED LED	NAKED LOAD
IC BUFFER	BJT DRIVER	IC DRIVER
MOSFET DRIVER	SSR DRIVER (DC)	EMR DRIVER
SSR DRIVER (AC)	SUPER LOOP	PROJECT HEADER
PORT I/O	PORT HEADER	HARDWARE DELAY
SOFTWARE DELAY	HARDWARE WATCHDOG	CO-OPERATIVE SCHEDULER
HARDWARE TIMEOUT	LOOP TIMEOUT	MULTI-STAGE TASK
MULTI-STATE TASK	HYBRID SCHEDULER	PC LINK (RS232)
SWITCH INTERFACE **(SOFTWARE)**	SWITCH INTERFACE **(HARDWARE)**	ON-OFF SWITCH
MULTI-STATE SWITCH	KEYPAD INTERFACE	MX LED DISPLAY
LCD CHARACTER PANEL	I^2C PERIPHERAL	SPI PERIPHERAL
SCI SCHEDULER **(TICK)**	SCI SCHEDULER **(DATA)**	SCU SCHEDULER **(LOCAL)**
SCU SCHEDULER **(RS-232)**	SCU SCHEDULER (RS-485)	SCC SCHEDULER
DATA UNION	LONG TASK	DOMINO TASK
HARDWARE PULSE-COUNT	SOFTWARE PULSE-COUNT	HARDWARE PRM
SOFTWARE PRM	ONE-SHOT ADC	ADC PRE-AMP
SEQUENTIAL ADC	A-A FILTER	CURRENT SENSOR
HARDWARE PWM	PWM SMOOTHER	3-LEVEL PWM
SOFTWARE PWM	DAC OUTPUT	DAC SMOOTHER
DAC DRIVER	PID CONTROLLER	255-TICK SCHEDULER
ONE-TASK SCHEDULER	ONE-YEAR SCHEDULER	STABLE SCHEDULER

Table 2: The 72 patterns we have assembled in order to support the development of embedded systems. From: Pont, M.J. (2001) "Patterns for time-triggered embedded systems"*, Addison-Wesley.*

Object-Oriented Remoting - Basic Infrastructure Patterns

Markus Völter
voelter
Ingenieurbüro für Softewaretechnologie
Germany
voelter@acm.org

Michael Kircher
Siemems AG
Corporate Technology
Software and System Architectures
Germany
michael.kircher@siemens.com

Uwe Zdun
New Media Lab
Department of Information Systems
Vienna University of Economics
Austria
zdun@acm.org

This pattern language describes the building blocks of typical distributed object frameworks, such as Java RMI, CORBA, .NET Remoting, web object systems, or web services. The patterns cover the basic infrastructure of such distributed object frameworks in a rather abstract manner, as it can be observed by developers using a distributed object systems for object-oriented remoting. The patterns presented in this paper are used in almost every distributed object framework application.

Introduction: Remoting Applications

Distributed systems are probably the most common way of building complex software systems today. Many major systems – those that are really large, complex, and expensive – are distributed systems. They are used for many different purposes, including the Internet, reservation systems, in-vehicle software, telecommunication networks, air traffic control, video streaming, and many more.

Many critical issues, such as performance, predictability, parallelism, scalability, partial failure, etc., have to be taken into account (see [TS2002]). Three fundamentally different remoting paradigms are used in today's software systems: there are those systems that use the metaphor of a remote procedure call, those that use the metaphor of posting and receiving messages, and those that use continuous streams of data. Note that this paper looks mostly at the first of these three different paradigms.

In remote procedure call (RPC) systems, two different roles are distinguished: clients and servers. A server provides a (more or less well-defined) set of operations which the client can invoke. These operations look like normal local operations: they typically have a name, parameters and a return type, as well as a way to signal exceptions in some systems. The goal of remote procedure call middleware is to provide clients with the illusion that a remote invocation is the same as a local one. Here, we concentrate on object-oriented remote procedure call systems, where a server application hosts a set of objects that provide the operations to clients as part of their public interface.

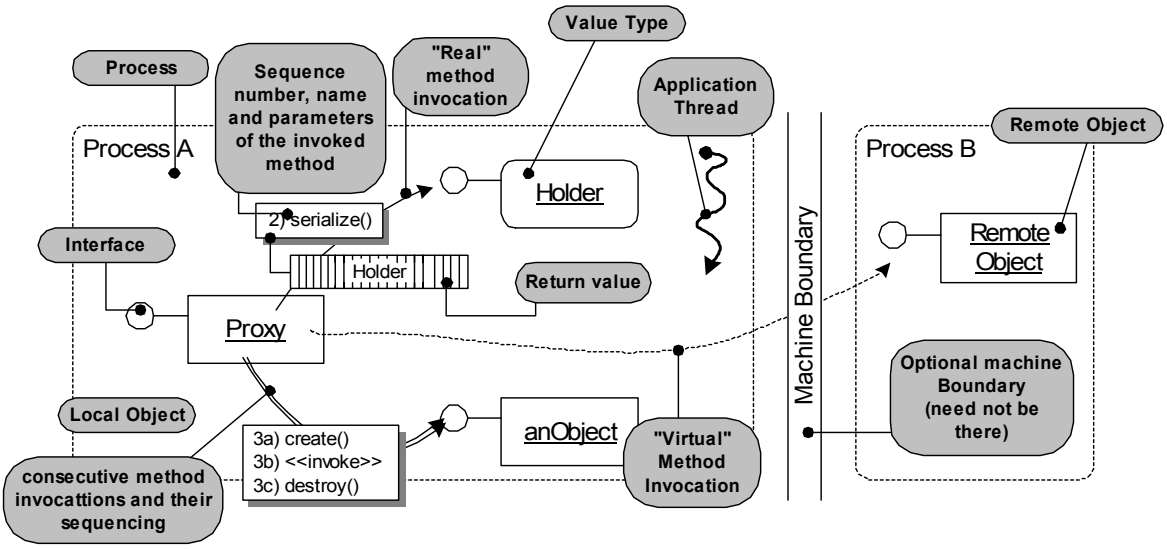
Whichever remoting paradigm is used today in an application, the application developer should be shielded from the details of the underlying metaphor. That is, in (OO-)RPC systems, the developer should only see a local method call, and not need to care about locating the server object, marshalling the request, or detect certain remoting-specific error conditions. A middleware provides a simple, high-level programming model to the developers, hiding all the nitty-gritty details as far as possible (but no further). Middleware specifications and products are available for all remoting paradigms.

In this paper, we will discuss a pattern language consisting of the basic infrastructure patterns that one has to know and understand when working with (or constructing) an (OO-)RPC middleware (called a "distributed object framework").

Pattern Form

The form of our patterns is Alexandrian, without examples. That is, each pattern starts with a name. It is followed by the context of the pattern in the language, and then three stars follow. After that, the problem is described in bold face, and then in plain face, the problem is described in more detail with the forces of the pattern. Then, following the word "Therefore" follows the solution, again in bold face. A detailed solution with emphasis on the related pattern in the language comes next (in plain face). Finally, after another three stars, the consequences of the pattern are given.

Each pattern is illustrated with a "collaboration diagram." As we also display containment structures in these diagrams, the illustrations are not really UML diagrams. The following example illustration is a legend annotated with comments.



Basic Distributed Object Patterns: Overview

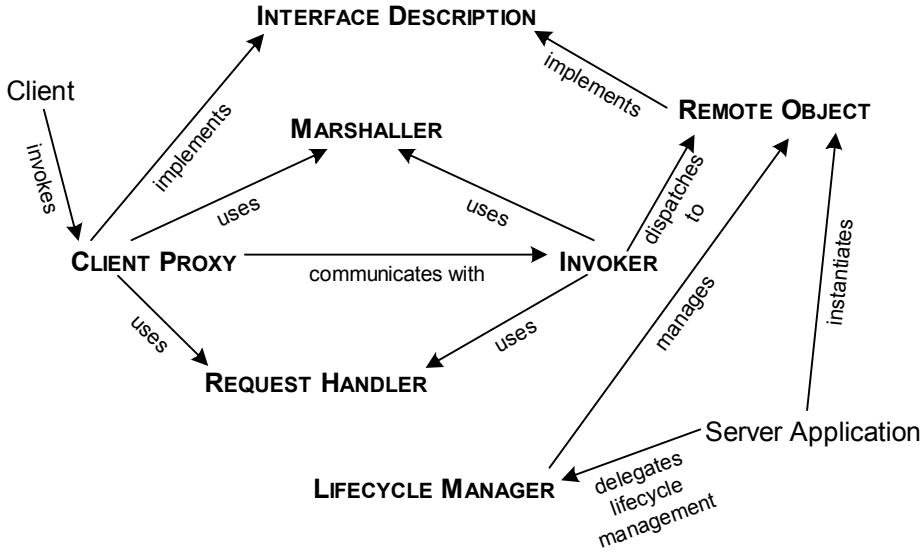
The reason for building or using distributed object frameworks is to allow clients to communicate with objects on a remote server. On the server side the invoked functionality is implemented as a REMOTE OBJECT. The client invokes an operation of a local object and expects it to be executed by the REMOTE OBJECT. To make this happen, the invocation crosses the machine boundary, the correct operation of the correct REMOTE OBJECT is obtained and executed, and the result of this operation invocation is passed back across the network. These basic communication tasks between client and REMOTE OBJECT are handled by the patterns described in this chapter.

A CLIENT PROXY is used by a client to access the REMOTE OBJECT. The CLIENT PROXY is a local object within the client process that offers the REMOTE OBJECT'S interface. This interface is defined using an INTERFACE DESCRIPTION.

The client can use a CLIENT REQUEST HANDLER to handle network communication. On the server side, the remote invocations are received by a SERVER REQUEST HANDLER. It handles the message reception and forwards invocations to the INVOKER, after the message is received completely. The INVOKER dispatches remote invocations to the responsible REMOTE OBJECT using the received invocation information.

On client and server side complex types are serialized and de-serialized using a MARSHALLER.

The LIFECYCLE HANDLER manages lifecycle issues of a group of REMOTE OBJECTS.



In most distributed object frameworks, these patterns are integrated with a few typical components. A SERVER APPLICATION instantiates and controls the REMOTE OBJECTS. The FRAMEWORK FACADE shields the distributed object framework and provides a simple API to developers using the distributed object framework.

The COMMUNICATION FRAMEWORK implements the layer beneath the distributed object framework, and it handles the low-level details of network communication. In object-oriented systems it is usually built using a set of common patterns for concurrent and networked objects. In the distributed object framework, the COMMUNICATION FRAMEWORK is primarily used by the REQUEST HANDLER both on client and server side. A PROTOCOL PLUG-IN can be used by developers to exchange or adapt the protocol implemented by the REQUEST HANDLER.

Remote Object

You are using (or building) a distributed object framework – a framework to access objects remotely. Clients access functionality provided by a remote SERVER APPLICATION.

❊ ❊ ❊

In many respects, accessing an object over a network is different from accessing a local object. Machine boundaries, process boundaries, network latency, network unreliability, and many other distinctive properties of network environments play an important role and need to be "managed." How to access an object in a distant process, separated by a network?

For a remote invocation, machine boundaries and process boundaries have to be crossed. An ordinary, local operation invocation is not sufficient, because additionally the operation invocation has to be transferred from the local process to the remote process, running within the remote machine.

Object identities, unique in the address space of one process, are not necessarily unique across process boundaries and machine boundaries.

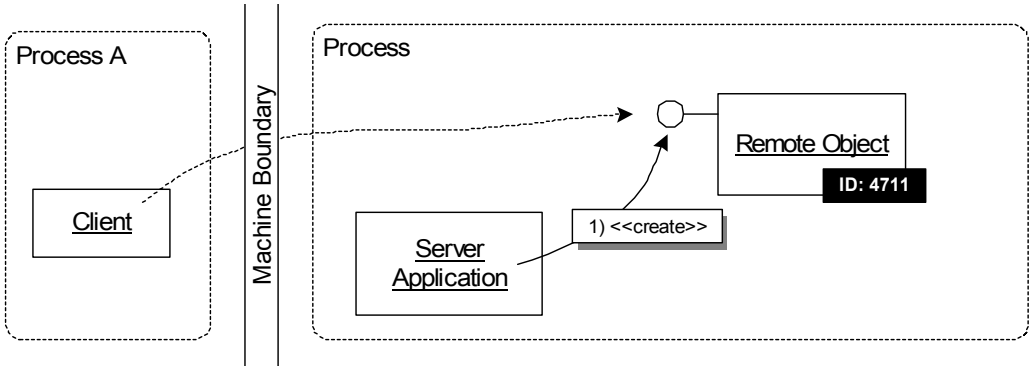
Compared to local invocations, invocations across a network involve delay and unpredictable latency. Because networks must be considered to be unreliable, clients must deal with new kinds of errors. Also, you want to minimize the number of (slow and thus expensive) network hops.

The interface of an object that is provided remotely is different to the local interface. The local interface can contain additional operations that should not be invoked by remote clients, but only by local clients. Thus an interface has to be defined on which remote clients can rely.

The distributed object framework should provide solutions for these fundamental issues of accessing objects remotely.

Therefore:

Provide a distributed object framework on the client side and server side. This framework transfers local invocations from the client side to a REMOTE OBJECT, running within the server. These REMOTE OBJECTS are used as the building blocks for distributed applications. Each REMOTE OBJECT provides a well-defined interface to be accessed remotely; that is, the remote client can address the REMOTE OBJECT across the network and invoke (some of) its operations.



The SERVER APPLICATION instantiates a REMOTE OBJECT. Then clients can access the REMOTE OBJECT using the functionality provided by the distributed object framework.

❊ ❊ ❊

The client and the REMOTE OBJECT usually reside within different processes and possibly also within different machines. The SERVER APPLICATION manages the lifecycle of the REMOTE OBJECTS.

It is responsible for instantiation and destruction. The distributed object framework provides the infrastructure to let clients access REMOTE OBJECTS.

The distributed object framework provides a REMOTE OBJECT type or interface. This is used to distinguish REMOTE OBJECTS from other, local objects. It also provides means to access the functionalities of the distributed object framework from the REMOTE OBJECT.

REMOTE OBJECTS have an unique *object ID* in their local address space, as well as means to construct a *global object reference*. The *global object reference* is used to reference and subsequently access a REMOTE OBJECT across the network.

Clients need to know the remotely accessible interface of a REMOTE OBJECT. A simple solution is to let remote clients access any operation of the REMOTE OBJECT. But perhaps some local operations should be inaccessible for remote clients, such as operations used to access the distributed object framework. Thus each REMOTE OBJECT type defines or declares its remotely accessible interface. Often this definition or declaration is given as an INTERFACE DESCRIPTION.

The REMOTE OBJECT'S interface is also supported by the CLIENT PROXY that handles remote communication on client side. CLIENT PROXIES communicate with an INVOKER on server side that dispatches invocations to the addressed REMOTE OBJECT.

REMOTE OBJECTS are a solution to extend the object-oriented paradigm across process and machine boundaries. However, accessing objects remotely always implies a set of inherent problems, such as network latency and network unreliability, that cannot be completely hidden. Different distributed object frameworks hide these issues to a different degree. When designing distributed object frameworks, there is always a trade-off between possible control and ease-of-use.

Client Proxy

You want to access a REMOTE OBJECT, running in a server, from a remote client.

❋ ❋ ❋

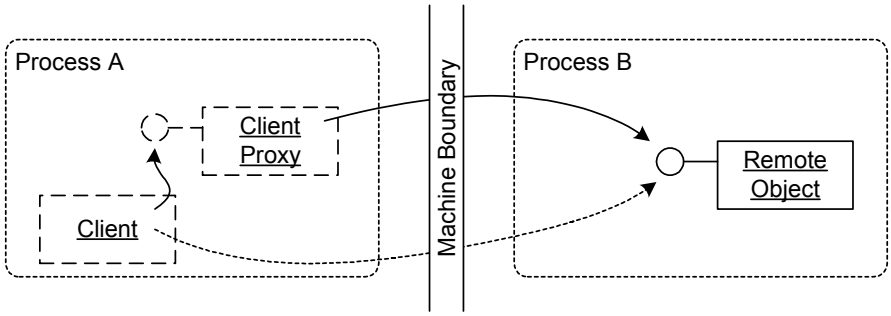
A primary goal of using REMOTE OBJECTS is to support a programming model for REMOTE OBJECTS that is similar to accessing local objects. A client developer should not have to deal with issues like access to the network, transmission of invocations, marshalling, and similar basic remoting functionalities. Of course, some fundamental properties of remote programming, such as unreliability and slowness of network calls, cannot be completely hidden. These should be presented to clients with the proper abstractions from the exploited object-oriented programming model.

The main purpose of distributed object frameworks is to ease development of distributed applications. Thus, developers should not necessarily leave their accustomed "way of programming." In the ideal case, they can simply invoke operations of the REMOTE OBJECTS just as if they were local objects.

Of course, some issues of remote activation and remote error conditions have to be considered by the client developer. Fundamental network properties such as network unreliability and latency also make an invocation of remote operations different to a local invocation. In cases where the client developer needs control over some remoting properties, appropriate APIs should be provided and these should integrate well with the accustomed programming model. In general, however, a remote invocation should be very similar to a local invocation. Issues of transporting an invocation across the network, such as mapping invocations to the REMOTE OBJECT'S address or marshalling the invocation data, should be hidden from client developers.

Therefore:

In the client application, use a CLIENT PROXY object for accessing the REMOTE OBJECT. The CLIENT PROXY object supports the interface of the respective REMOTE OBJECT. For remote invocations, clients only interact with the local CLIENT PROXY object. The CLIENT PROXY primarily forwards invocations to the REMOTE OBJECT. It is responsible for the details of accessing the REMOTE OBJECT via the distributed object framework. Only those remoting details that cannot be handled automatically are exposed to the client developer.



To invoke an operation of a REMOTE OBJECT, the client invokes a operation of the CLIENT PROXY that supports the REMOTE OBJECT'S interface. The CLIENT PROXY forwards this invocation to the REMOTE OBJECT. It handles the details of crossing the machine boundary and process boundary.

❋ ❋ ❋

A CLIENT PROXY is a variant of the *proxy* pattern [GHJV95] which is also documented in a variant supporting remoting [BMR+96]. CLIENT PROXIES do not access the REMOTE OBJECT directly. Instead they call an INVOKER on the server that dispatches the request to the correct target object. Thus

the INVOKER is responsible for transforming the remote invocation into a local invocation inside the REMOTE OBJECT'S server process.

Developers of clients cannot assume that the REMOTE OBJECT is reachable all the time, for instance, because of network delays, network failures, or server crashes. The CLIENT PROXY abstracts these remoting details for clients using *remoting errors*.

Instantiation of REMOTE OBJECTS requires the involvement of the SERVER APPLICATION. Sometimes client and CLIENT PROXY can also be involved, if a client-dependent instance is required. A related issue that involves the CLIENT PROXY is distributed garbage collection. In the context of client side failures, it has to be ensured that client-dependent instances are cleaned up, when they are not required anymore. *Leases* provide a possible solution to this problem.

The CLIENT PROXY is required on client side, and thus it has to be deployed to the client somehow. The simplest solution are CLIENT PROXIES that are hard-wired in the client code. The liability of this simple solution is that a complete, new client has to be deployed every time some interface or remote access operation of one REMOTE OBJECT changes. In some systems a client does not even know before runtime to which REMOTE OBJECT it connects, and thus the CLIENT PROXY cannot be hard-wired in the client. How to allow clients to use server objects without knowing about them at compile time? To resolve these problems, two CLIENT PROXY variants provide a solution:

- *Exchangeable client proxy:* The CLIENT PROXY class can be designed to be exchangeable in the client. Then a CLIENT PROXY class that corresponds to a specific INVOKER is delivered to the client at runtime. This usually happens during startup of the client. Distributing the CLIENT PROXY can be automatically handled by *naming*. This solution has the advantage that CLIENT PROXIES can be exchanged transparently, but incurs the liability that it requires the CLIENT PROXY classes to be sent (in binary form) across the network.
- *Dynamic invocation interface:* Alternatively, a more generic CLIENT PROXY can be used. That means, the CLIENT PROXY does not provide the REMOTE OBJECT'S interface as its own interface, but invocations are dynamically constructed. That means, the interface of the REMOTE OBJECT is not known in advance (before the invocation reaches the CLIENT PROXY), and thus has to be looked up for each invocation. Dynamic invocations on the CLIENT PROXY are very flexible but they incur some performance overhead for lookup of the REMOTE OBJECTS interface.

For dynamic invocations, there are again two variants; either the CLIENT PROXY or the INVOKER can lookup the interface of the REMOTE OBJECT:

- *Interface repository:* If the CLIENT PROXY looks up the interface dynamically, it has to query an interface repository (a variant of the pattern INTERFACE DESCRIPTION) to retrieve the operation signature.
- *Dynamic dispatch on the invoker:* Alternatively, the CLIENT PROXY can send a symbolic invocation (like strings containing *object ID*, operation name, and arguments) across the network. Then it is the burden of the INVOKER to dynamically look up the interface. Note that this CLIENT PROXY variant has to deal with a special *remoting error* type that is raised when the dynamic dispatch was not successful.

The CLIENT PROXY uses a MARSHALLER to marshall the invocation and to de-marshall the result received.

CLIENT PROXY code for accessing the COMMUNICATION FRAMEWORK and invoking the MARSHALLER can be reused to a large degree. Only interface-dependent parts are custom (for instance per REMOTE OBJECT type). These interface-dependent parts can be generated or retrieved from the REMOTE OBJECT'S INTERFACE DESCRIPTION automatically.

Invoker

A remote invocation reaches the server side and should be delivered to a specific REMOTE OBJECT.

❋ ❋ ❋

When a CLIENT PROXY forwards invocation data across the machine boundary to the server side, somehow the targeted REMOTE OBJECT has to be reached. The simplest solution is to let every REMOTE OBJECT be addressed over the network directly. For large numbers of REMOTE OBJECTS this solution does not work. For one, there may be not enough network addresses (that is: network ports) for each REMOTE OBJECT. The client developer would have to deal with the network addresses to select for the appropriate REMOTE OBJECT, which is cumbersome and too complex. There is no way for the SERVER APPLICATION to control the access to its REMOTE OBJECTS (centrally). How to avoid these server side problems of invoking REMOTE OBJECTS?

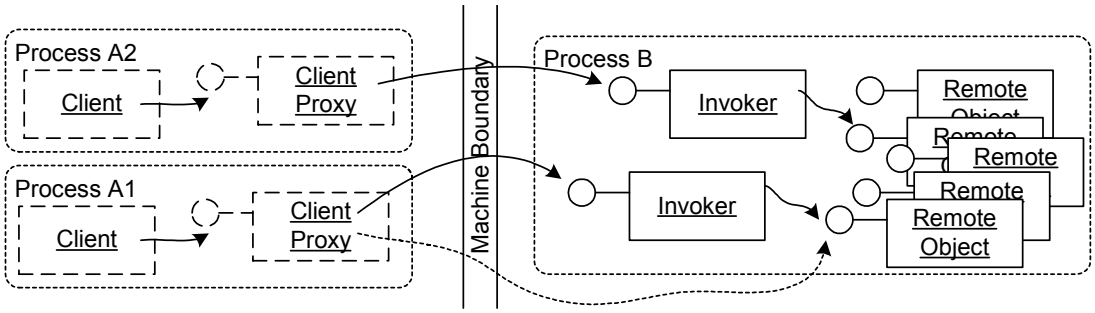
Consider a SERVER APPLICATION providing a large number of sensors as REMOTE OBJECTS to remote clients. If each of the sensor instances would be addressed over the network directly, a first problem might be that the possible number of sensors might exceed the number of free network ports. Then the sensors cannot be provided anymore by directly binding the REMOTE OBJECTS to network ports. Thus, if REMOTE OBJECTS are directly addressed over the network, scalability might be limited, due to the number of free ports.

But even if there is only a limited number of clients, there are still other issues left open, when directly associating REMOTE OBJECTS with network ports. Client developers would have to be aware of (the large number of) sensor network addresses. This is a considerable complexity that should be avoided. Instead, a client should only have to provide the necessary information required to select the appropriate sensor's REMOTE OBJECT, and the SERVER APPLICATION should have to deal with finding and invoking that object.

In many application scenarios a SERVER APPLICATION requires some central control over the invocation process of its REMOTE OBJECTS. Consider you want to implement some functionality that operates for a set of REMOTE OBJECTS of a SERVER APPLICATION. Examples are logging remote invocations or access control of REMOTE OBJECTS. In both cases it is required to deal with the invocation before or after it reaches the REMOTE OBJECT. The developer of the REMOTE OBJECT should not have to implement code for these issues; instead, these issues should be handled by the SERVER APPLICATION solely.

Therefore:

Provide an INVOKER that is remotely accessed by CLIENT PROXIES. The CLIENT PROXIES send invocations across the network, containing the actual operation invocation information and additional contextual information. The INVOKER has to de-marshal the operation invocation information to obtain the ID of the targeted REMOTE OBJECT and an identifier of operation to be invoked. Then the INVOKER dispatches the invocation to the REMOTE OBJECT; that is, it looks up the correct local object and operation implementation, corresponding to the remote invocation, and invokes it. The INVOKER can also be used to extend or control the invocation process.



The CLIENT PROXY sends a request to the INVOKER with invocation information for a REMOTE OBJECT. The INVOKER dispatches the local object address and operation implementation. Then it invokes the correct operation implementation for this object.

❄ ❄ ❄

The INVOKER is part of the SERVER APPLICATION. Possibly one SERVER APPLICATION can provide more than one INVOKER. The task of receiving messages via the network is handled by a REQUEST HANDLER. The REQUEST HANDLER is responsible for the details of receiving remote messages, such as threading, connection pooling, and accessing the operating system APIs. It hands over control to the INVOKER after a message is received completely.

INVOKERS specifically handle the message dispatching task for a group of REMOTE OBJECTS. Such a group of REMOTE OBJECTS might, for instance, consist of all REMOTE OBJECTS in a SERVER APPLICATION, or of the object in a specific REMOTE OBJECT configuration groups.

To reduce the amount of memory resources needed, the server can temporarily evict REMOTE OBJECTS instance from memory or use different *pooling* strategies. The INVOKER is used to implement this functionality.

Message dispatching means to de-marshal the symbolic information in the message, then to use this information to determine the target object and operation, and finally invoke this operation. De-marshalling is handled by a MARSHALLER.

For determining and invoking the target object and operation, different context information can be used, such as the *object ID*, an operation identifier, and the REMOTE OBJECT type. Context information of a remote invocation is implemented using *invocation contexts*. The *invocation context* at least contains the *object ID* and operation to be invoked.

Determining and invoking the target object and operation can be either handled dynamically or statically:

- *Server stubs* (also called skeletons) are a static dispatch mechanism. The part of the INVOKER that is responsible for actually invoking the object is generated from the INTERFACE DESCRIPTION. For each REMOTE OBJECT type one server stub is generated. Thus the INVOKER „knows“ its REMOTE OBJECT types, operations, and operation signatures in advance. The INVOKER can directly invoke the called operation, and it does not have to find the operation implementation dynamically. Of course, the corresponding CLIENT PROXY has to address the correct INVOKER and determine the operation implementation statically. In comparison to dynamic dispatch variants, static dispatch eliminates the performance overhead of looking up operation implementations at runtime.
- *Dynamic dispatch:* Consider a situation in which the REMOTE OBJECTS interfaces are not known at compile time. Then it is not possible to generate or write (static) server stubs for the INVOKER. Instead the INVOKER has to decide at runtime which operation of which remote object should be invoked. To do this, first the INVOKER has to extract the *object ID*, operation name, and arguments from the invocation information sent by the CLIENT PROXY. Next it has to find the corresponding REMOTE OBJECT and operation implementation in the local process,

for instance, using runtime dispatch mechanisms, such as reflection [Mae87] or dynamic lookup in a table. Finally, the target instance and operation are invoked. This form of dispatch is called dynamic dispatch, as the INVOKER dispatches each invocation at runtime (for more details of this implementation variant see the pattern *message redirector* [GNZ01]). Dynamic dispatch is more flexible than static dispatch but has a performance penalty. As an invocation can possibly contain a non-existing operation or use a wrong signature, a special *remoting error* has to be delivered to the CLIENT PROXY in case the invocation fails.

Note that for using an INVOKER it is necessary to create a corresponding CLIENT PROXY. The CLIENT PROXY has to provide the invocation information required by the INVOKER. However, it is not necessarily required that the dynamic dispatch variant of INVOKER is combined with a dynamic invocation interface of CLIENT PROXY, or that (static) server stub requires static CLIENT PROXIES. In existing systems, often matching variants (like server stubs together with static CLIENT PROXIES) are used merely out of practical reasons. For instance, a code generator can directly generate matching server stubs and static CLIENT PROXIES from an INTERFACE DESCRIPTION. Of course, dynamic and static dispatch can also be mixed, and many existing distributed object systems provide static and dynamic INVOKERS simultaneously.

There is a trade-off between dynamic and static dispatch variants: dynamic variants are more flexible because they can dynamically be modified or extended with new object and operation types. Static variants are more efficient but they usually require recompiling INVOKERS and CLIENT PROXIES. That means new CLIENT PROXIES have to be distributed to the client side. Deploying CLIENT PROXIES to clients can be a problem; in such cases, a (more) dynamic solution can help to implement modifications only on server side.

An INVOKER bundles the REMOTE OBJECTS for which it dispatches messages, for instance, all objects of a specific REMOTE OBJECT type. Using INVOKERS instead of direct network connections to the REMOTE OBJECTS, reduces the number of required network addressable entities, and the REQUEST HANDLER can use this information to share connections from the same client. The INVOKER can be used to implement behavior affecting a group of REMOTE OBJECTS.

Request Handler

You are providing REMOTE OBJECTS in a SERVER APPLICATION, and INVOKERS are used for message dispatching. A client invokes REMOTE OBJECTS using a CLIENT PROXY.

❋ ❋ ❋

For sending invocations from the client to the server side, many tasks have to be performed on client side: connection establishment and configuration, result handling, timeout handling, and error detection. On server side similar tasks have to performed before the responsible INVOKER can dispatch the respective operation to the targeted REMOTE OBJECT: the server has to listen to the respective port, handle (socket) communication via the communication protocol, and manage the communication resources. Especially for larger and more performance-critical systems it is necessary to allow for global optimization and coordination of the communication resources. That means to coordinate communication setup, resource management, threading, and concurrency in a central fashion. If more than one INVOKER or CLIENT PROXY is used, the request or reply has to be forwarded to the responsible INVOKER or CLIENT PROXY.

Consider a typical scenario in the embedded systems domain: a SERVER APPLICATION manages a large number of different types of controller objects. Each controller receives measurements from sensor objects in clients and forwards actions to actuator objects. Consider further that TCP/IP is used for network communication. In a naive implementation, each controller object type would have its own INVOKER, which would also manage connections with the clients. If a client connects to several different controller objects on the same server, an unnecessarily high number of TCP/IP connections needs to be maintained. If the SERVER APPLICATION would use a (blocking) thread to handle requests for each connection, a high number of threads would be necessary, impeding server performance.

In this scenario, similar problems can occur on client side, when the CLIENT PROXY that handles every detail of the network communication on its own. This would only work for simple, single-threaded clients with only a few requests. In more complex clients, possibly a large number of requests are sent simultaneously. Each CLIENT PROXY may have to handle multiple invocations at the same time.

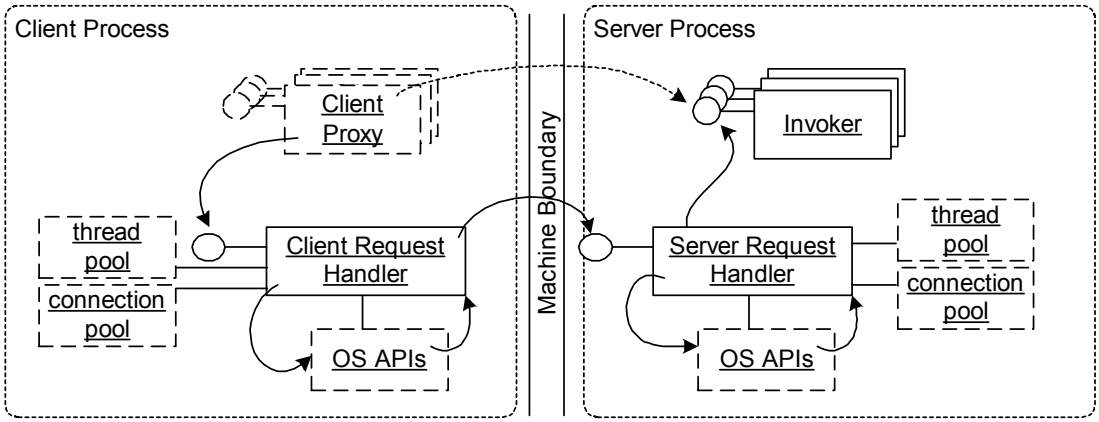
In general, a way to coordinate communication handling and management is required to arrange the network traffic and resource consumption effectively. In most applications it cannot be predicted at which time a particular REMOTE OBJECT or INVOKER (or CLIENT PROXY respectively) has which workload. In such cases, the fair allocation of shared communication resources, as well as quality of service (QoS) constraints, should be handled centrally. Thus the allocation of the communication resources should not be realized by individual CLIENT PROXIES, REMOTE OBJECTS, or INVOKERS. This is also important for reducing the number of used network connections: potentially, network connections to the same client can be shared for the whole server process.

Configuration of network protocols, add-on services, and general asynchrony has to be done on a per invocation basis. For instance, only some operations may be logged, need access control, or need a special Quality of Service. Often client and server side have to work in sync to handle these tasks.

Therefore:

Provide a SERVER REQUEST HANDLER together with a respective CLIENT REQUEST HANDLER. The CLIENT REQUEST HANDLER is used for client side invocations by the CLIENT PROXIES. It is responsible for opening and closing the network connection. It also sends the invocation across the network, waits for results, and dispatches them to the CLIENT PROXY. Additionally, it needs to cope with timeouts and errors on a per invocation basis. The SERVER REQUEST HANDLER is responsible for dealing with the network communication on server side. Messages

are received via network connections (incrementally). The SERVER REQUEST HANDLER waits until the full message has arrived, and then, if required, the message is (partially) demarshalled to find the responsible INVOKER. Next, the message is forwarded to this INVOKER for further processing. Finally, the REQUEST HANDLER has to clean up the connection and other resources it has used. The REQUEST HANDLER typically uses efficient, optimized mechanisms of the underlying operating system. Both CLIENT and SERVER REQUEST HANDLER can use *pooling* for connection handles and request threads.



Connections are handled by the REQUEST HANDLER on client and server side. These are used by CLIENT PROXIES and INVOKERS. The REQUEST HANDLER makes effective use of the communication resources and the operating system APIs, and it pools connections handles and threads.

❋ ❋ ❋

The primary responsiblity of a REQUEST HANDLER is to handle network communication. For this purpose, it instantiates a connection handle per connection. The connection handle is responsible for opening and closing the socket connection and storing the file descriptor of the conncection.

The CLIENT REQUEST HANDLER has to manage network connection on client side. It sends invocation requests and informs the client when the result arrives. For a synchronuous invocations this simply means to return to the waiting CLIENT PROXY operation. For asynchronuous invocations the CLIENT REQUEST HANDLER can invoke a *result callback*, a *poll object*, or use other asynchrony strategies.

When timeouts have to be supported, the CLIENT REQUEST HANDLER informs the CLIENT PROXY of timeouts, as it already does this for general network errors. For this purpose, the CLIENT REQUEST HANDLER sets up a timeout event when the invocation is sent to the REMOTE OBJECT. If the reply does not arrive within the timeout period the CLIENT PROXY is informed.

A SERVER REQUEST HANDLER generically handles the communication across the network. INVOKERS, in contrast, are specific for a group of REMOTE OBJECTS. Therefore, SERVER REQUEST HANDLER and INVOKERS are implemented as separated components because the SERVER REQUEST HANDLER should handle the communication resources independently of particular INVOKERS or REMOTE OBJECTS. Implementing both INVOKER and SERVER REQUEST HANDLER as one component makes only sense for small SERVER APPLICATIONS or if there is just one INVOKER per SERVER APPLICATION.

Both CLIENT and SERVER REQUEST HANDLER have to deal with network events: the CLIENT REQUEST HANDLER has to wait for the result of its invocations, the SERVER REQUEST HANDLER has to wait for arriving invocations. Both REQUEST HANDLERS typically use the *reactor* [SSRB00] pattern for demultiplexing and dispatching events from the network. The REQUEST HANDLER receives the events dispatched by the *reactor* and handles them. The same event dispatching infrastructure

usually can be used on client and server side (for more details see the description of the COMMUNICATION FRAMEWORK at the end of this chapter), whereas CLIENT and SERVER REQUEST HANDLER are different implementations.

The REQUEST HANDLER can also use *half-sync/half-async* [SSRB00] and/or *leader/followers* [SSRB00] to manage network connections and threading efficiently. The patterns CLIENT PROXY, REQUEST HANDLER, and INVOKER together build a *broker* as described in [BMR+96]. REQUEST HANDLERS are also responsible for making effective use of the operating system APIs.

For each particular connection, the REQUEST HANDLER has to instantiate a connection handle. To optimize resource allocation for connections, the connections can be shared in a pool as well (using the pattern *pooling* [KJ02]). If one particular client process sends or receives more than one message to the SERVER APPLICATION at the same time, the network connection can be shared among these messages.

If a client communicates with a REMOTE OBJECT in a SERVER APPLICATION, in some application scenarios it can be expected that this client will potentially communicate again with the same SERVER APPLICATION. Thus the connection can be held open for a certain period of time and used in case of a continued communication. This form of continued communication is called a persistent connection [FGM+99] and, if required, it is implemented by the REQUEST HANDLER. Persistent connections eliminate the overhead of establishing and destroying connections for continuous client requests.

The REQUEST HANDLER provides framework functionality for PROTOCOL PLUG-INS. Also other add-on services, such as access control or logging, can be implemented by REQUEST HANDLER. Note that many of these tasks require the CLIENT and SERVER REQUEST HANDLER to work in concert.

Although the REQUEST HANDLER is described as a single component that all invocations have to pass, usually it is not a bottleneck because with connection pools and thread pools it is highly concurrent. The pattern allows to effectively use the communication resources, independently of the REMOTE OBJECT, CLIENT PROXY, or INVOKER structures. Because of this, a REQUEST HANDLER is reusable for many different clients and SERVER APPLICATIONS. The same REQUEST HANDLER instance is shared by multiple CLIENT PROXIES on client side and multiple INVOKERS on server side. REQUEST HANDLERS hide connection and message handling complexity from the CLIENT PROXIES and INVOKERS.

However, REQUEST HANDLERS impede a slight overhead. In smaller applications with only few network connections and high performance requirements, the performance overhead may matter. Another liability may be the memory overhead of the connection and thread pools, especially in limited computing environments such as embedded systems. Using connection and thread pools in REQUEST HANDLERS does only make sense if the instantiation times for connection handles and threads may have a significant performance impact in terms of response times. This means that there should be a considerable number of messages expected to be received at the same time. For example, simplistic clients can also handle network communication on their own (for instance, simply with a blocking request).

Marshaller

You make REMOTE OBJECTS available to clients. The invocation data is transported over the network. CLIENT PROXY, REQUEST HANDLER, and INVOKER handle the basic communication issues.

❊ ❊ ❊

The data to describe operation invocations of REMOTE OBJECTS consist of the target object's *object ID*, the operation identifier, the arguments, and possibly other context information. All this information has to be transported over the network connection. For transporting it over the network only byte streams are suitable as a transport format.

For sending invocation data across the network there has to be some concept for transforming invocations of remote operations into a byte stream format. There are some exceptional cases that make this task complicated, as for instance:

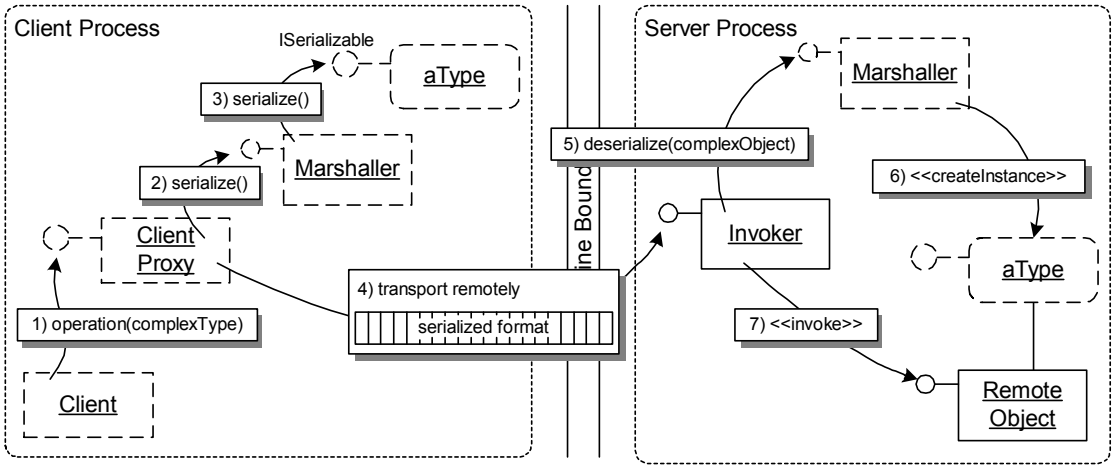
- How do you handle references occurring multiple times?
- How do you handle object identity for non-remote objects?
- How do you determine wether an attribute of an operation should be transformed into a byte stream or not (for instance, references to GUI objects or local resources will usually not be transmitted)?
- How do you handle domain-specific or application-specific specialties of the REMOTE OBJECTS (for example, for objects that are persistent in a database the relationships with the database have to be handled)?
- How do you handle complex, user-defined type objects? These have references to other instances, possibly forming a complex hierarchy of objects. Such hierarchies might even contain multiple references to the same instance, and in such cases, logical identities have to be preserved after the transport to the server.

Generating and interpreting a byte stream representation should not require additional programming efforts per instance, but should only be defined once per type.

Sometimes even the process of generating and interpreting transport formats should be extensible for developers. Additionally, sometimes the used data formats have to be extensible as well.

Therefore:

Require each type used within REMOTE OBJECT invocations to provide a way to serialize itself into a transport format that can be transported over a network as a byte stream. The distributed object framework provides a MARSHALLER (on client and server side) that uses this mechanism whenever a remote invocation needs to be transported across the network. A MARSHALLER also implements some scheme how to preserve object identities and deal with complex data types. In many systems, developers can provide custom MARSHALLERS to customize this scheme or to use other transport formats. The MARSHALLER also provides operations to de-marshal a given byte stream. Make sure the CLIENT PROXY, INVOKER, and REQUEST HANDLER invoke the respective MARSHALLER at the appropriate times.



An operation with an object as argument is invoked. The CLIENT PROXY uses the responsible MARSHALLER to serialize this object. The serialized format is transported across the network. This format is de-serialized by the MARSHALLER on server side and the instance of the respective type is created. Finally, the REMOTE OBJECT is invoked and uses this object.

❋ ❋ ❋

A MARSHALLER converts remote invocations into byte streams in a way that is non-specific for REMOTE OBJECT types.

Complex type object should not be referenced remotely, but marshalled by value. To transport such a type across the network, a generic transport format is required. For this purpose, a MARSHALLER uses the *serializer* pattern [RSB+98]. The serialization of a complex type can be done in multiple ways:

- The programming language can provides a generic, built-in facility. This is often the case in interpreted languages which can use reflection to introspect the type's structure.
- Tools generate serialization code directly from the INTERFACE DESCRIPTION, assuming that the structure of such types is also expressed in the INTERFACE DESCRIPTION.
- It also can be the developer's burden to provide the serialization functionality. In this case, the developer usually has to implement a suitable interface that declares operations for serialization and de-serialization.

The concrete format of such serialized data depends on the distributed object framework used. In principle, everything is a byte stream as soon as it is transported over the network. It is often more convenient to use a structured format such as XML, CDR, or ASN.1 to represent complex, structured data.

A MARSHALLER can support a hook to let developers provide a custom MARSHALLER. Reasons are that generic marshalling may become a rather complex and performance-consuming operation, when you need to marshal complex data structures, such as graphs. Some serialization formats might be better than others for certain environments. Thus one generic marshalling format can never be optimal for each data structure. You might also need to optimize data packets for bandwidth.

A custom MARSHALLER might, for instance, transport all attributes of an object, or it might only transport the public ones, or it might just ignore those it cannot serialize instead of throwing an exception. As a consequence, exchanging a MARSHALLER is not necessarily transparent for the application on top.

Different serialization formats have their benefits and liabilities. For instance, the byte stream format can be application-specifically optimized for efficient processing or memory or bandwidth usage. But such formats are hardly human-readable, and harder to create and parse, because standard tools are usually missing. Standard representations such as XML or CDR can be created and parsed easily with standard tools, but may result in a less efficient (because more or less generic) representation. This means these formats require more processing power to be created or parsed. Some formats, especially XML, are very bloated, because they use a human-readable, text-based representation. While there is a large set of tools to process XML, it requires a lot of memory and significantly more network bandwidth than more condensed formats, such as binary data. Standard representations are usually more interoperable than application-specific formats.

Interface Description

A client invokes an operation of a REMOTE OBJECT using CLIENT PROXY and INVOKER.

❋ ❋ ❋

When a CLIENT PROXY and an INVOKER are used together for remote communication, you need to align the interface of CLIENT PROXY and REMOTE OBJECT. Also, you need to align marshalling and de-marshalling. Client developers need to know the interfaces of REMOTE OBJECTS they use.

If a CLIENT PROXY should be used as a local representative of a REMOTE OBJECT in the client process, it exposes the interface provided by the REMOTE OBJECT. If the CLIENT PROXY is responsible for ensuring that the interface is used correctly, the type, operation, and signature information of the REMOTE OBJECT have to be known before a invocation is sent across the network.

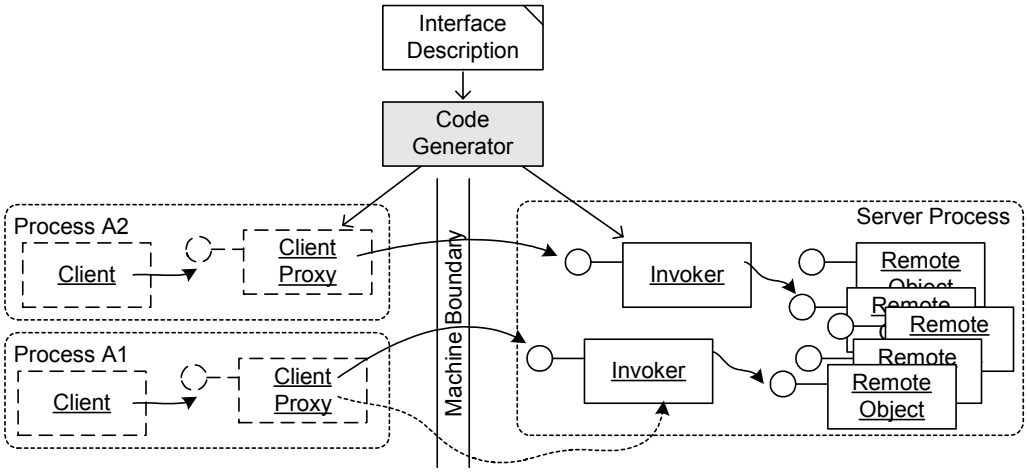
On server side, the INVOKER has to dispatch the invoked operation of the REMOTE OBJECT. If the INVOKER does not dispatch all type, operation, and signature information dynamically, it requires these information before an invocation takes place.

Either CLIENT PROXY or INVOKER should ensure that no violation of the REMOTE OBJECT'S interfaces occurs, or, if violations are possible, CLIENT PROXY or INVOKER have to handle them.

It should not be required that client developers or REMOTE OBJECT developers have to deal with propagating and/or ensuring REMOTE OBJECT interfaces manually. Instead, the distributed object framework should provide suitable means for partly automating these issues.

Therefore:

Provide an INTERFACE DESCRIPTION in which you describe the interface of a REMOTE OBJECT type required for CLIENT PROXY and INVOKER, as well as information for marshalling and dispatching.From the INTERFACE DESCRIPTION interface-related parts of CLIENT PROXY and INVOKER can be derived (either with code generation or runtime configuration techniques). The INTERFACE DESCRIPTION also documents the remoting interfaces for client developers.



Using an INTERFACE DESCRIPTION in a separate file a code generator generates code for CLIENT PROXY and INVOKER. The CLIENT PROXY is then used by the client. It contacts the INVOKER that dispatches the invocation to the REMOTE OBJECT. This way, details of the distributed object framework are hidden from client and REMOTE OBJECT developers, as they only see client code, INTERFACE DESCRIPTION and REMOTE OBJECT code.

❋ ❋ ❋

Usually an INTERFACE DESCRIPTION contains interface specifications including their operations and signatures, as well as marshalling and dispatching information. The INTERFACE DESCRIPTION itself can be given in various forms:

- *Interface description language:* The INTERFACE DESCRIPTION is separated from the program text, for instance, in an additional file written by the REMOTE OBJECT developer. An interface description language is used to provide these information. A code generator generates the interface-related parts of CLIENT PROXY and INVOKER. That means (many) interface violations can be automatically detected when compiling client and CLIENT PROXY.
- *Interface repository:* The INTERFACE DESCRIPTION is provided at runtime to remote clients using an interface repository exposed by the SERVER APPLICATION (or by some external interface repository). Note that an interface repository is required for implementing the dynamic invocation interface variant of CLIENT PROXY. An interface repository is not sufficient for code generation purposes solely, as code generators require the information before runtime.
- *Reflective interfaces:* The INTERFACE DESCRIPTION is only provided on server side by means of reflection [Mae87]. The dynamic dispatch variant of INVOKER is used to obtain the interface information and handle the complete invocation process. A simplistic CLIENT PROXY only sends symbolic invocations to the INVOKER, but does not know the actual interface of the REMOTE OBJECT. Note that this variant requires the INVOKER to handle interface violations and other exceptions. This variant is not suited for code generation, as the INTERFACE DESCRIPTION is not exposed before runtime.

Note that the different variants of INTERFACE DESCRIPTION correspond to the used variants of CLIENT PROXY and INVOKER. Static variants of CLIENT PROXY and INVOKER can exploit code generation techniques and thus primarily use separated interface descriptions in interface description languages. Dynamic variants require INTERFACE DESCRIPTIONS either on client or on server side at runtime, thus reflective interfaces or interface repositories are used. In many distributed object frameworks more than one INTERFACE DESCRIPTION variant is supported, as more than one variant of CLIENT PROXY and/or INVOKER are supported.

operation signatures offered to clients should generally be designed to stay stable. However, changes cannot be avoided. Especially, in a distributed setting, where SERVER APPLICATION developers have no control over client code (and deployment of it), there should be some common way to provide modified remoting interfaces to client developers. INTERFACE DESCRIPTIONS primarily separate interfaces from implementations. Thus the software engineering principle “separation of concerns“ is supported, as well as exchangeability of implementations. That means clients can rely on stable interfaces, while REMOTE OBJECT implementations can be exchanged.

Lifecycle Manager

Your SERVER APPLICATION provides different kinds of REMOTE OBJECTS. Each of these objects has a lifecycle.

❄ ❄ ❄

A SERVER APPLICATIONS has to handle the lifecycle of its REMOTE OBJECTS. In first place, that means to create each object when it is needed and ensure that objects are destroyed (using their destructor) when the server (or thread) terminates. At runtime, the SERVER APPLICATION has to control its resources. For instance, it should ensure that only those objects that are actually needed are active (in memory). All others should be either destroyed or passivated to a database (according to the REMOTE OBJECT'S activation strategy).

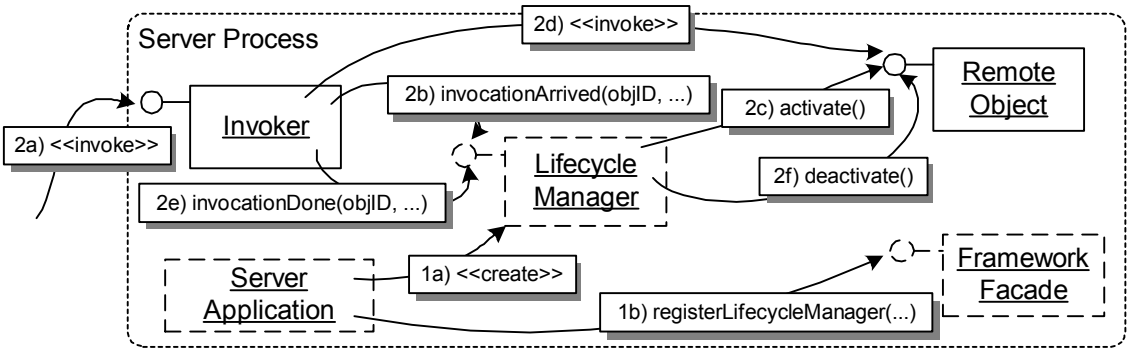
Consider a web community portal with about one hundred thousand registered users. Only a small number of these users will be active at the same time. Thus for reasons of scalability, only those user objects that are actually needed at a certain point in time should be active objects in memory. All other objects should be created or activated on demand, according to the activation strategy of their REMOTE OBJECT type.

Implementing standard activation, sharing, and eviction strategies requires triggering certain lifecycle events. A generic mechanism used for implementing these lifecycle events would also leverage the integration of these patterns and code reuse in their implementations.

Besides standard activation, sharing, and eviction strategies, sometimes developers require custom lifecycle *strategies*. Consider an application in which a REMOTE OBJECT can be paused; that is, the messages for this object are sent to a message queue and processed later on. Some instance has to redirect the message for certain *object IDs* to the message queue instead of the paused REMOTE OBJECT.

Therefore:

Provide a LIFECYCLE MANAGER to handle the lifecycle of a set of REMOTE OBJECTS. It also stores the current lifecycle state of each REMOTE OBJECT. The REMOTE OBJECTS implement lifecycle operations corresponding to the possible lifecycle events. These allow the LIFECYCLE MANAGER to modify the lifecycle state of an object. Different LIFECYCLE MANAGERS can be present in the same SERVER APPLICATION, implementing different lifecycle *strategies*. A custom LIFECYCLE MANAGER can extend both lifecycle states and lifecycle operations. Before and after an invocation the INVOKER calls the LIFECYCLE MANAGER to ensure that the invoked object is active.



The responsible LIFECYCLE MANAGER is created by the SERVER APPLICATION during startup and it is registered with the distributed object framework using the FRAMEWORK FACADE'S API. Before an invocation the LIFECYCLE MANAGER is informed by the INVOKER. If the REMOTE OBJECT is not active, the LIFECYCLE MANAGER activates it. Then

the invocation is performed. After it returns, the LIFECYCLE MANAGER is informed again and can deactivate the object.

❋ ❋ ❋

The INVOKER invokes the LIFECYCLE MANAGER before and after each invocation so that the LIFECYCLE MANAGER can handle the lifecycle events. Informing the LIFECYCLE MANAGER of events in the INVOKER can be hard-coded in the INVOKER code, or it can be implemented with an *invocation interceptor*. Often it is also necessary to let REMOTE OBJECTS invoke their associated LIFECYCLE MANAGER to inform it about certain lifecycle events. Thus it is necessary that each REMOTE OBJECT can obtain a reference to its LIFECYCLE MANAGER.

Note that it might be necessary to let the LIFECYCLE MANAGER work asynchronously, for example, to scan for objects that should be deactivated because some timeout has been reached. The LIFECYCLE MANAGER can, for instance, be informed of the timeout with a callback operation.

The possible lifecycle events of a LIFECYCLE MANAGER are implemented as lifecycle operations by the REMOTE OBJECTS. The REMOTE OBJECTS have to provide lifecycle operations that fit to the LIFECYCLE MANAGER'S lifecycle *strategy*. The LIFECYCLE MANAGER can invoke these lifecycle operations to change the lifecycle state of an object. Typical lifecycle states are: not existing, inactive, virtual, and active.

Consider implementing *passivation*. A persistent object may have an additional lifecycles state *virtual,* meaning that the REMOTE OBJECT is currently not an active instance, but its *object ID* still can be passed to clients. Upon an invocation, some entity has to map the requested *object ID* to the serving, virtual REMOTE OBJECT, and it has to be re-activated on demand.

The LIFECYCLE MANAGER needs to know the REMOTE OBJECTS under its control. For this purpose, it can maintain an object map which associates *object IDs* with REMOTE OBJECTS, or it uses a list of objects maintained elsewhere in the SERVER APPLICATION. The current lifecycle state is also stored in the LIFECYCLE MANAGER'S object map.

The LIFECYCLE MANAGER can be used to implement activation, sharing, and eviction *strategies*. Especially, the activation patterns can be implemented this way. Providing new, custom activation policies makes SERVER APPLICATIONS customizable in terms of performance tuning and custom resource management.

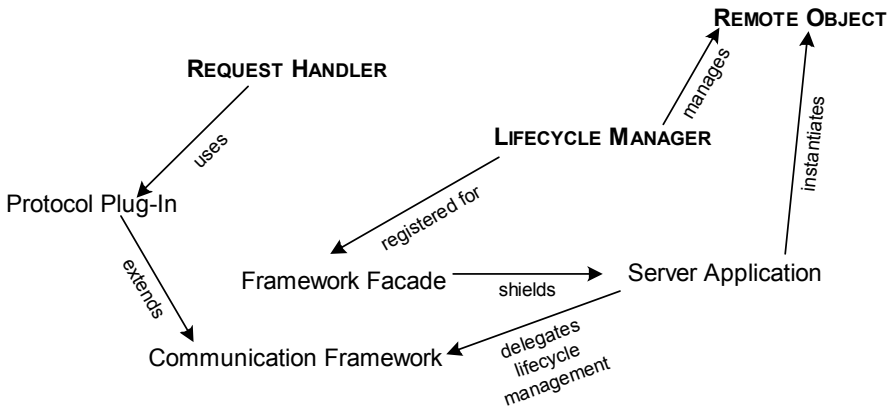
However, a LIFECYCLE MANAGER also incurs the liability of a slight performance overhead, as it has to be informed of every invocation and return of a message.

A LIFECYCLE MANAGER can be seen as a combination of *activator* [SSRB00] and *evictor* [Jai01].

Integrating the Patterns

There are many interactions among the patterns presented and with the patterns presented in later chapters. Components of the environment of a distributed object framework, as well as the use of the patterns in these components, are discussed in the remainder of this chapter. In particular:

- A SERVER APPLICATION is a central instance that manages its REMOTE OBJECTS. Its most important task is to bundle all the objects that belong to one application or service. During startup the SERVER APPLICATION initializes the distributed object framework, if necessary, and obtains references to well-known REMOTE OBJECTS, such as *naming*. Also, it instantiate REMOTE OBJECTS according to their activation strategy and delegates lifecycle management of REMOTE OBJECTS to its LIFECYCLE MANAGER.
- A FRAMEWORK FACADE shields the distributed object framework from direct access. The distributed object framework is a complex piece of software and only parts of it are relevant for developers of SERVER APPLICATIONS. Necessary configuration parameters are offered with a concise interface, the FRAMEWORK FACADE.
- The COMMUNICATION FRAMEWORK implements the low-level details of network connections. The distributed object framework patterns largely abstract these details. However, sometimes it is important to understand the COMMUNICATION FRAMEWORK elements. We explain them with a set of common pattern for concurrent and networked objects.
- A PROTOCOL PLUG-IN is plugged into the CLIENT and SERVER REQUEST HANDLER. It substitutes the communication protocol, used per default by the REQUEST HANDLER, if necessary. This can be used for optimizing the network protocols used for particular applications.



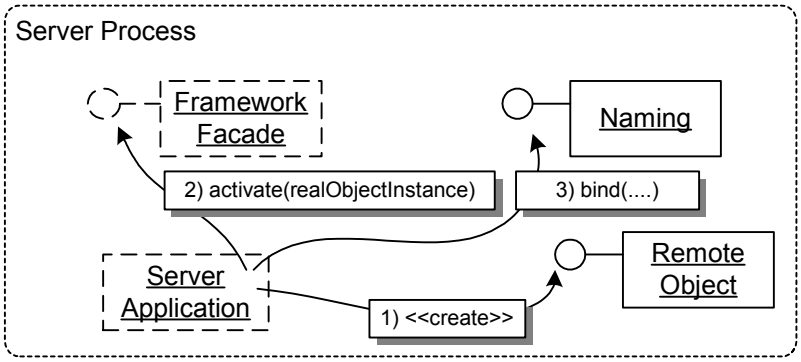
Server Application

The REMOTE OBJECT pattern describes how services are offered remotely. In order to achieve this, there are many tasks that have to be fulfilled in the server:

- The distributed object framework has to be initialized.
- Like any other object, a REMOTE OBJECT has to be instantiated. In a distributed object framework there are different strategies defined when and how remotely accessible objects are instantiated.
- When *naming* is used, a REMOTE OBJECT can be registered with it. To make this possible, a *global object reference* to *naming* has to be resolved in advance.
- A set of related REMOTE OBJECTS form together one remote application. In one server process more than one application may run.

- An application has not only a lifetime responsibility for creating the REMOTE OBJECTS, but also for destroying them. If the application terminates, its REMOTE OBJECTS have to be destroyed, too.

Thus, we provide a SERVER APPLICATION whose job it is to initialize and configure the distributed object framework. The SERVER APPLICATION uses the FRAMEWORK FACADE for this task. Moreover, it resolves initial, pre-configured references such as *naming*. Then it instantiates REMOTE OBJECTS that are static instances, or prepares for instantiating other REMOTE OBJECTS, according to their activation strategy.



For each REMOTE OBJECT to be instantiated, the SERVER APPLICATION creates the object according to the activation strategy, activates it in the distributed object framework, and optionally binds it with *naming*.

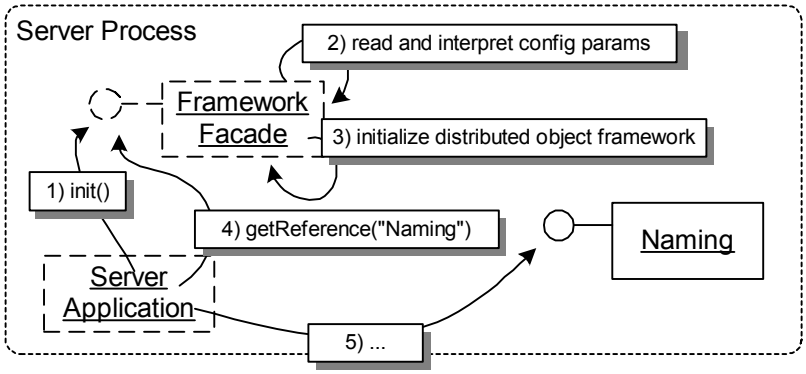
The SERVER APPLICATION bundles REMOTE OBJECTS that conceptually belong together, and handles all object management tasks with respect to the distributed object framework. If a lifecycle manager is used, the SERVER APPLICATION object delegates the actual management of the REMOTE OBJECT'S lifecycle to a LIFECYCLE MANAGER.

Framework Facade

The SERVER APPLICATION has to initialize the distributed object framework and provide access to its COMMUNICATION FRAMEWORK. A distributed object framework is a rather complex piece of software. It needs to coordinate several components and their interactions, all need to be initialized and configured properly and consistently.

Some (simple) way for developers using the distributed object framework is required for configuring the COMMUNICATION FRAMEWORK and interacting with it. Developers also need a way to resolve initial, well-known references to *pseudo objects*. References that must be initially resolved include *naming* and other well-known REMOTE OBJECTS available in a distributed object framework. These "well-known" REMOTE OBJECTS have to be configured somewhere.

As a solution, the distributed object framework provides a central FRAMEWORK FACADE. For developers using the distributed object framework, it serves as the single access point and administration API to the distributed object framework and its services.



A FRAMEWORK FACADE is initialized by a SERVER APPLICATION. It reads and interprets the configuration parameters. Then it initializes the distributed object framework. It also serves as the central place to obtain references to central services, such as *naming*.

The FRAMEWORK FACADE is typically a *pseudo object,* meaning that it behaves like any other managed REMOTE OBJECT but it is not remotely accessible.

A FRAMEWORK FACADE is used in server and client applications. You might want to provide different implementations, as the performance and scalability requirements for servers are usually significantly higher than those for pure clients.

A FRAMEWORK FACADE provides a single access point (a *facade* [GHJV95]) to the distributed object framework. It reduces complexity and shields the constituent parts from direct access. A central FRAMEWORK FACADE avoids multiple initialization and configuration of the same distributed object framework.

However, situations that require (unforeseen) access to the distributed object framework's intricacies are hard to handle, as bypassing the *facade* would be the only way for application developers to deal with such situations.

Communication Framework

In this section we give a brief overview of (patterns of) the COMMUNICATION FRAMEWORK shielded by the distributed object framework. On server side interaction with the COMMUNICATION FRAMEWORK is especially handled by the pattern REQUEST HANDLER.

Usually a COMMUNICATION FRAMEWORK is designed using *layers* [BMR+96]. The lowest *layer* is an adaptation *layer* to the operating system and network communication APIs. Using an adaptation *layer* has the advantage that higher layers can abstract from platform details, and therefore use an platform-independent interface.

The operating system APIs are (often) written in the procedural C language; thus, if the COMMUNICATION FRAMEWORK is written in an object-oriented language, *wrapper facades* [SSRB00] are used for encapsulating the operating system's APIs. The higher layers only access the operating system's APIs via the *wrapper facades.* Each *wrapper facade* consists of one or more classes that contain forwarder operations. These forwarder operations encapsulate the C-based functions and data within an object-oriented interface. Typical *wrapper facades* provide access to threading, socket communication, I/O event handling, dynamic linking, etc.

On top of the *wrapper facades* the COMMUNICATION FRAMEWORK is defined. Connection establishment is handled by the *acceptor/connector* pattern [SSRB00]. The pattern separates the connection initialization from the use of established connections. A connection is created, when the connector on client side connects to the acceptor on server side. On the basis of *acceptor/connector*, connect strategies and concurrency strategies become exchangeable. Once a connection is estab-

lished, further communication is done based on a connection handle, returned from successful connection establishment.

The SERVER APPLICATION starts an event loop, and new connections are handled as events. A *reactor* [SSRB00] reacts on the communication events raised on the connection handles. Its task is to efficiently demultiplex the events and dispatch all requests to connection handler objects in the SERVER APPLICATION.

Concurrency is dealt with using concurrency patterns [SSRB00] (see also [Lea99]). There are two alternatives for synchronizing and scheduling concurrently invoked remote operations:

- *Active object* decouples the executing from the invoking thread. In the case of a distributed object framework, the invoking thread belongs to the REQUEST HANDLER, whereas the executing thread belongs to the INVOKER. A queue is used between those threads to exchange requests and responses.
- A *monitor object* ensures that only one operation runs within an object at a time by queueing operation executions. It applies one lock per object to synchronizes access to all operations.

The mentioned patterns are used within a *half-sync/half-async* architecture. The pattern decouples asynchronous and synchronous processing by defining an asynchronous and a synchronous service processing layer. A queue between these layers maps asynchronous invocations to synchronous execution.

A *component configurator* [SSRB00] supports component configuration and dynamic reconfiguration of components in an application at runtime. Typical components include reusable implementations of common services used in distributed applications, such as *naming* or logging. A *component configurator* uses a *factory* [GHJV95] to create the service objects according to configuration parameters.

CLIENT PROXIES, REQUEST HANDLER, and INVOKERS build together a *broker*, as it is documented in [BMR+96].

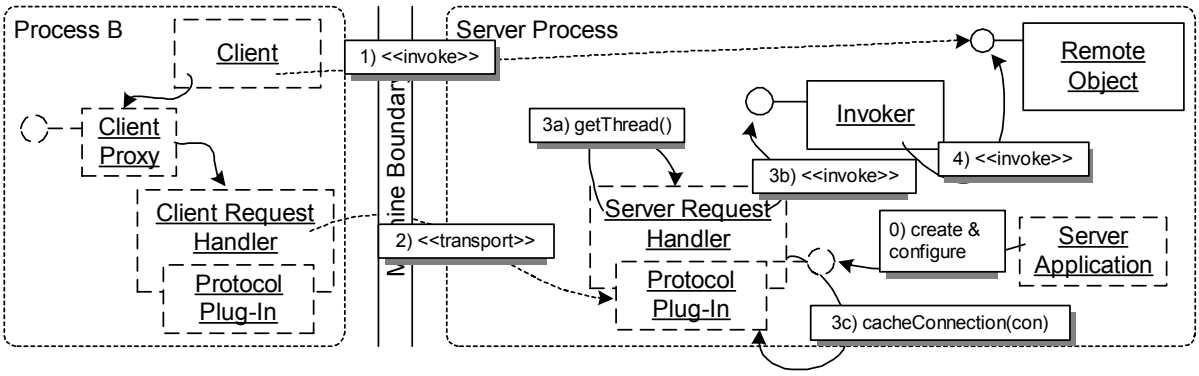
Protocol Plug-in

Usually, a developer using the distributed object framework can abstract from the implementation details of the COMMUNICATION FRAMEWORK. However, there are some situations in which these details matter and it is necessary to provide some way to influence them:

- Sometimes the same application should be able to operate with different protocols. Thus the protocol realization should be exchangeable.
- The developer provides a custom MARSHALLER that provides an optimized serialization mechanism. It is important to make sure that the protocol used by the COMMUNICATION FRAMEWORK can actually transport the serialized data.
- The SERVER APPLICATION needs to fulfil varying QoS requirements. To effectively fulfil these QoS requirements, the facilities provided by the network protocol have to be used differently and perhaps need to be optimized at a rather low level.
- You have to serve many clients. Thus it might be beneficial to optimize the protocol to open a new connection each time a request is transported, because this reduces resource consumption of the SERVER APPLICATION.

Other low-level aspects you might need to take care of are threading, invocation priorities, or caching of certain data. Such issues should be handled transparently for the application logic.

PROTOCOL PLUG-INS are provided as an extension mechanism of the REQUEST HANDLER. The PROTOCOL PLUG-INS handle the low-level networking issues in cooperation with the COMMUNICATION FRAMEWORK and operating system.



The SERVER APPLICATION creates and configures the PROTOCOL PLUG-IN for the REQUEST HANDLER. For an invocation, the CLIENT PROXY transports the message to this PROTOCOL PLUG-IN. The REQUEST HANDLER then forwards the invocation to the INVOKER. It may cache the connection in the PROTOCOL PLUG-IN.

Often, a PROTOCOL PLUG-IN and a custom MARSHALLER go hand in hand, because a specific protocol requires specific marshalling or vice versa.

Although we focused on the server side, the client obviously also requires something that adapts its CLIENT PROXY to the networking details. The protocol used by client and server must match, obviously.

A PROTOCOL PLUG-IN offers an API to customize low-level protocol details of the COMMUNICATION FRAMEWORK. There is no need to integrate these low-level details into the application logic. In principal, protocols can be exchanged transparently. However, it is not always possible to plug-in new protocols without changes in REQUEST HANDLER, INVOKER or SERVER APPLICATION. If more than one protocol are used together, the provided APIs either can only provide a common denominator of these protocols, or the other components have to be aware of the used protocol.

Conclusion

In this paper, we have presented a pattern language consisting of basic infrastructure pattern of distributed object frameworks. These patterns are required for building almost any RPC-based distributed object framework. Moreover, they have to be understood to build applications with these frameworks as well. Note that, for more complex distributed object framework applications many other patterns have to be applied, say, to achieve high performance, scalability, and multi-threading.

Acknowledgements

We wish to thank our Viking Plop 2002 shepherd Kristian Elof Sørensen for his valuable comments on this paper, as well as the participants in the Viking Plop 2002 writers workshop: Kevlin Henney, Michael Pont, Valter Cazzalo, Mikio Aoyama, Juha Pärsinen, and Lars Grunske.

References

[BMR+96] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerland and M. Stal, *Pattern-Oriented Software Architecture: A System of Patterns*, John Wiley and Sons, 1996.

[FGM+99] R. Fielding, J. Gettys, J. Mogul, H. Frysyk, L. Masinter, P. Leach, and T. Berners-Lee. *Hypertext transfer protocol – HTTP/1.1. RFC2616*, 1999.

[GHJV95] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[GNZ01] M. Goedicke, G. Neumann, and U. Zdun. Message Redirector. In *Proceedings of EuroPlop 2001*, Irsee, Germany, July 2001.

[Jai01] P. Jain. Evictor Pattern , In *Proceedings of 8th Pattern Languages of Programs Conference*, Illinois, USA, Sep. 11-15, 2001

[KJ02] M. Kircher, and P. Jain. Pooling Pattern, In *Proceedings of EuroPlop 2002*, Irsee, Germany, July, 2002.

[Lea99] D. Lea. *Concurrent Java: Design Principles and Patterns*, Second Edition, Addison-Wesley, 1999.

[Mae87] P. Maes. *Computational reflection*. Technical report 87-2, Free University of Brussels, AI Lab, 1987.

[RSB+98] Dirk Riehle, Wolf Siberski, Dirk Bäumer, Daniel Megert, and Heinz Züllighoven. Serializer. In *Pattern Languages of Program Design 3*. Edited by Robert Martin, Dirk Riehle, and Frank Buschmann. Reading, Massachusetts: Addison-Wesley, 1998. Chapter 17, page 293-312.

[SSRB00] D. C. Schmidt, M. Stal, H. Rohnert, and F. Buschmann. Patterns for Concurrent and Distributed Objects. Pattern-Oriented Software Architecture. John Wiley and Sons, 2000.

[TS2002] A. Tanenbaum and M. van Steen. *Distributed Systems: Principles and Paradigms*. Prentice Hall, 1995.

[VSW02] M. Voelter, A. Schmid, and E. Wolff. *Server Component Patterns*. John Wiley and Sons, 2002.

Design Patterns for Evolutionary Robotics

Esben H. Østergaard
esben@mip.sdu.dk
Adaptronics Group
The Maersk McKinney Moller Institute for Production Technology
University of Southern Denmark
Campusvej 55, 5230 Odense M., Denmark

The construction of controllers for robots performing simple tasks in well-known and well-structured environments is reasonably well understood, and well established theories and communities exist. On the other hand, construction of controllers for robots placed in dynamic and unstructured environments can be very challenging, and no good general methodologies exist. Often, finding good solutions relies on the experience and intuition of the system engineer. Evolutionary robotics offers an approach to automate this process. However, designing an evolutionary robotic system is a non-trivial process, that requires perhaps even more experience and intuition than designing the controller itself. The development of a toolbox in the form of a design pattern language might significantly reduce the problems related to designing such a system.

In this paper, four design patterns regarding evolutionary robotics is presented that could become part of design pattern language for evolutionary robotics. The four presented patterns focus on the principles and dynamics of evolutioanry robotics, rather than on implementation aspects. The first pattern can be regarded as a context for the subsequent three patterns.

Name: Artificial Evolution of Real Robots

Context: A lot of progress has been made in techniques for developing robotics for production linies, where the task the robot performs is monotonic and where the robot designer can engineer the robots environment. In the 70's-90's, automated production lines appeared all over the world.

In non-monotonic and complex environment, no techniques for desinging robot controllers are available at present time. Considering expectations from the 1960's, very little progress has been made in producing robots that can function in non-engineered environments and solve a large array of different

tasks. The typical scenario for a "hard" robot task is a mobile robot performing in a real world environment such as an office or an outdoor environment.

Problem: Engeneering controllers for robots can sometimes be a very tricky and challenging task. In some cases, no general applicable approaches exist that can be applied to ensure that a robot controller reliably and robustly performs the task. This is especially true for robots performing real time in complex and unstructured environments.

To increase robustness for such tasks, it has been proposed to write controllers as "loosely coupled parallel processes"[11] [1]. Such controllers might exhibit behavior that is not programmed into any of the processes, but emerges from the interaction between the processes. Controllers that exhibit emergent behavior can be hard to engineer. How do we deal with the complexity of writing such controllers?

Forces: A central issue in the problem described above, is dealing with complexity. Controllers are especially hard to write when

- The robots environment is complex
- The robots hardware is complex

When a controller consists of loosely coupled parallel processes, dealing with the complexity of the controller is also an issue. Such controllers might exhibit emergent phenomena arising from the interaction between the physical parts of the robot, the robot controller and the environment. These emergent properties play a major role in the design of controllers for complex robots.

Solution: Artificial evolution is known to be able to cope with emergent phenomena in complex systems. The "Evolutionary Robotics" approach suggests automating the process of writing robot controllers by appling artificial evolution to the problem [2]. An artificial evolution starts with a population of randomly generated candidate solutions to the given problem. These solutions are evaluated, and a the best performing candidates are used as bias to generate a new generation in the population. This procedure is repeated until a sufficiently capable solution is found.

For using this approach to evolve robot controllers, the following aspects must be considered:

- **Task to be Solved.** Consider the task that you want the robot to perform. Is it physically possible for the robot to perform the task? Are the sensors and actuators of the robot sufficient?
- **Fitness Function.** It is necessary to find a suitable way of evaluating performance of the candidate solutions, by measuring their *fitness* according to some succes criteria. These evaluations should be fully automated,

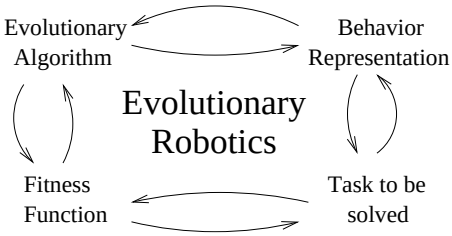


Figure 1: *Design aspects of an evolutionary robotics system. Arrows indicates influences between choices made for each aspects. Performance of the resulting system is a function of the interplay between the choices made.*

since a large number of evaluations will be required. Perhaps the robots own sensors can be used to evaluate the robots performance, or perhaps it is necessary to provide the environment or the robot with extra sensors. The fitness function compiles information from these sensors, and assigns a scalar fitness value to each candidate solution.

- **Behavior Representation.** An evolutionary algorithm typically works on a string of numbers, resembling a gene string. A mechanism is needed that translates this gene string into a controller on the robot. This controller is said to represent the robot's behavior. As an example, the controller can be a neural network where genes code for the synaptic weights. The behavior representation must be powerfull enough to express solutions to the task to be solved.

- **Evolutionary Algorithm.** Potentially, any guided search algorithm can be used. It is important that the search algorithm is capable of rapidly searching large spaces by using information about the candidate solutions fitness. An evolutionary algorithm is an obvious choice since it has these properties.

Figure 1 illustrates how design considerations of these aspects influence each other.

A conceptual model of an evolutionary robotics system is shown in figure 2. Genotypes generated by the evolutionary algorithm are translated into phenotypes, that are tested on the real robot. Performance is fed back to the evolutionary algorithm, to give guidelines for selection. After a sufficient number of generations, it is expected that the best performing phenotype can be used as a controller for performing the desired task.

Evolutionary methods is by nature stochastic, and no guarantees can be given as to whether or not applying evolutionary methods to a given problem

will come up with a suitable solution. Only running the system and testing the resulting controller, can reveal whether or not the desired behavior is produced. It might be very hard to predict which solution will be generated. Evolutionary systems tend to produce solutions that surprise even the system engineer. Overall, evolutionary methods should probably not be applied if there is a requirement for guaranteeing system behavior. Another thing to consider is that in evolutionary robotics systems it often is necessary to design a simulator of the robot and its environment. This is further described in the patterns *Simulated World* and *Shaking Simulation*. Finally, time consumption for designing and applying an evolutionary system should be weighed against time for handwriting the robots controller directly, and then use a series of iterative refinements of testing and rewriting to produce a working robot. Advantages of the evolutionary approach is that after one evolutionary system has been implemented, it can potentially automaticly produce controllers for related tasks throught only very minor changes. The main advantages of the approach is the potential to reduce number of man-hours used to construct the system, and the potential of the system to produce innovative solutions [8] to a given problem.

Resulting Context: As a result of applying the solution of this pattern, you have a system that in theory should generate robot controllers for the task at hand. You might however find, that it is infeasible to use the system due to the time required, wear and tear of the robots hardware, problems of assessing fitness or simply because of battery issues. In these cases, you might want evolution to take place in a simulation of the robots environemnt, as described in the pattern "Simulated World". You might also find, that there seems to be no progress in evolution. In that case you could experiment with the parameters for the evolutionary algorithm, or you could look at the "Shaping Evolution" pattern.

Example: An example of the use of this approach can be found in Adrian Thompsons paper "Artificial Evolution in the Physical World" [12]. The components of his system are:

- **Task to be Solved.** A two-wheeled robot has to navigate to the center of an arena, using only feedback from two on-board sonars.
- **Fitness Function.** The fitness for the robot while moving is given as: $fitness = \frac{1}{T}\int_0^T (e^{-k_x c_x(t)^2} + e^{-k_y c_y(t)^2})dt$, where $c_x(t)$ and $c_y(t)$ are the robots x and y position relative to the center at time t, and k_x and k_y are such that the maximum fitness is 1 and the lowest fitness is 0.1 (at the point furthest away from the center).
- **Behavior Representation.** The robot was equipped with a field programmable gate array (FPGA) taking input from the sonars and pro-

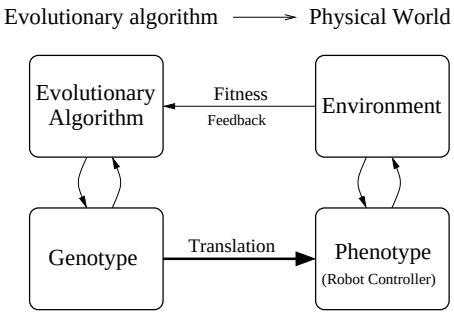


Figure 2: *Information flow in an evolutionary robotics system. The evolutionary algorithm manipulates gene strings. Gene strings are translated into candidate robot controllers, that are evaluated on the robot (Physical world). Performance of the candidate controllers is fed back to the evolutionary algorithm, where it is used to guide selection and reproduction of new candidate solutions. After a sufficient number of generations, the best candidate should be able to perform the task.*

ducing output to the two motors driving the wheels. The FPGA can be reconfigured at run-time to change the electronic computation it performs.

- **Evolutionary Algorithm.** A standard evolutionary algorithm was implemented, that searched through the space of possible configurations for the FPGA.

Using these components, a controller for driving the robot towards the center of the arena was generated.

Related Patterns: In the context of this paper, look at Shaped Evolution, Simulated World. Also look at patterns related to implementing and applying evolutionary algorithms (for example [9]). For alternative approaches to the evolutionary robotics approach you might look at Fuzzy Logic , Expert Systems, or Machine Learning Approaches [6], [11].

Name: Shaped Evolution

Context: You have designed, implemented and tested an evolutionary robotics framework, as described in the Artificial Evolution of Real Robots pattern. You have tried several different parameter setting for the evolutionary algorithm to optimize evolutionary progress.

Problem: There is no progress in evolution. Solutions never become better than random, even though several different parameter settings for the evolutionary algorithm have been tried. The evolutionary algorithm fails to generate controllers that makes the robot perform the task.

Forces: In some cases, the chance of finding a controller that performs the task is virtually zero. This happens especially in cases where the task is complicated and requires the robot to perform a sequence of actions, where the environment is highly unreliable, or where the behavior representation and evolutionary algorithm are such, that it is very unlikely that any random mutation will result in a better controller. In that case, evolution does not work. Computing power might not be sufficient to do brute force search of the gene space and maybe no better behavior representation is readily available. It could also be a problem that evolutionary pressure drowns in noise of the uncertainty of fitness evaluations.

Solution: One approach to dealing with this problem is to shape evolution by feeding your knowledge about the problem into the system, so the system can use this knowledge to guide the evolutioanry process.

Shaping can be applied to different parts of the system. Depending on the circumstances, either might be most applicable:

- **Fitness Function Shaping.** Change the fitness function so that small scores are given for partial solutions. It is important the fitness function gives "better" partial solutions higher scores than "worse" partial solutions.
- **Task Shaping.** Make the task easier, and evolve a controller for the easier task. While evolving on the same population, make the task gradually harder until the robot is performing the desired task.
- **Behavior Primitives Shaping.** Provide evolution with higher level behavior primitives. The primitives can be obtained by evolving controllers

to perform the basic actions of the robot, such as turning and moving forwards. Providing these higher level primitives for the evolutionary search should make it easier to find a good solution. [7]

Rationale: If none of the solutions generated by the evolutionary algorithm come close to performing the task, and thus all get the same low fitness score, the selection mechanism doesn't work. The system must be changed, so that selection pressure can kick in. By changing a combination of the fitness function and the behavior representation, hopefully a gradient of fitness in gene space can be created, which evolution then can follow.

Resulting Context: Shaping evolution increases the chance that a solution is found. However, shaping also potentially reduces the innovative strength of the evolutionary system. If too much shaping is applied, there is little difference between evolving a controller and hand writing a controller for the job. However, if too little shaping is applied, there is a risk that evolution never finds a good solution.

Related Patterns: Artificial Evolution of Real Robots, Simulated World

Name: Simulated World

Context: You have choosed the evolutionary robotics approach to design controllers for the robot to perform the task to be solved. You have ideas about which fitness function to use, which behavior representation to use, and which evolutionary algorithm to use. You are concerned about time consumption or other practical issues regarding the evolutionary system.

Problem: Evolutionary searches often require tens of thousands of evaluations. Performing these evaluations on the robot is very strenuous on the robots mechanical components, and requires a vast amount of time. Numbers of 1000 generations with 1000 evaluation per generation is common. If an evaluation takes about 1 minute, the total time usage for an evolutution is $1000\frac{evaluations}{generation} \times 1000\frac{generations}{evolution} \times 1\frac{minutes}{evaluation}$, which is about two years. This is enough to stress the durability of the robot components and the patience of the engineer.

Forces: Things to consider regarding evolution in the real world is time consumption, difficulties in automating fitness evaluations, risk of doing damage to the robot or its surroundings during evolution, durability of the robot, complexity of the robot, the environment and robot-environment interactions.

Solution: Construct a simulatior of the robot, the environment and robot-environment interactions. Use the model to simulate and evaluate candidate controllers for performing the desired task. Figure 2 and 3 illustrates the idea. Three approaches seems to dominate the field[1]

- **Table based simultion**, where the robots own sensors and actuators is used to sample a model of the environment.
- **Geometrical simulation**, using vector calculations to model the environment and the robots sensing and actuation.
- **Minimal simulation**, which is a cut-to-the-bone simulation, where only the most important aspects of the robot-environment interaction is modeled, and the rest is made unreliable by applying random noise.

Either approach or hybrid approaches can be used for any given problem. Roughly, the table based approach seems best suited for "small" enviornments

[1] Table Based, Geometrical and Minimal simulation might be regarded as their own patterns, but in the interest of keeping this paper short and focused, writing these patterns has been left as an exercise for the reader.

(small state space), the geometrical model suited for "clean" environments (expensive hardware, indoor factory environment) and the minimal model suited for complex robots doing simple tasks (eight legged robot walking). In many cases it can be very hard to guess which model resolution to use, and which parts of the world that are relevant, and which that are not important. Also modeling changes to robot component performance at different battery voltage levels can be hard to model.

It is often hard to construct a good model of the robots and its environment [3]. You need to asses which aspects of the real world that are important to model, and which are not. Often you will get it wrong the first time, and the resulting robot controller will fail to perform in the real world. Some aspects of the world, such as collisions between several objects, can be very hard to model. Another thing to consider is the amount of available computing power. Depending on the resolution of the simulator, much more computing power than what is available on the robot will be needed. Using a simulation might be the only viable approach in dangerous environments or if the robot might damage the environment.

One of the critique points of the simulation approach is, that it introduces the need for a model. This contradicts one of the main arguments for using the evolutionary robotics approach, namely that controllers can be produced without using complex models of the robot/environment. Using a model to simulate the robot is somehow against the spirit of evolutionary robotics.

Rationale: Carefully written computer simulations of the robot and its environment can be used to evaluate candidate solutions during evolution. In many cases, the simulator can evaluate solutions at greatly accelerated time, thereby reducing overall evolution time. Problems related to automating fitness measurements are avoided, and in case of mobile robots, also power issues and issues regarding robot hardware failure are avoided. If the fidelity of the simulation is high enough, the controller evolved in simulation will perform the task once transferred to the real world.

Resulting Context: The result is a system as the one illustrated in figure 3. The system evaluates candidate controllers in the simulator. After certain criteria are fulfilled, the best candidate controller is transferred to the real robot, which then hopefully performs the task.

In some cases it might be a good idea to continue evolution for a few generation on the real robot, to compensate for differences between the simulator and the real world.

Related Patterns: Artificial Evolution of Real Robots, Shaking Simulation. Possible future related patterns are Table Based Simulation, Geometrical Simulation and Minimal Simulation.

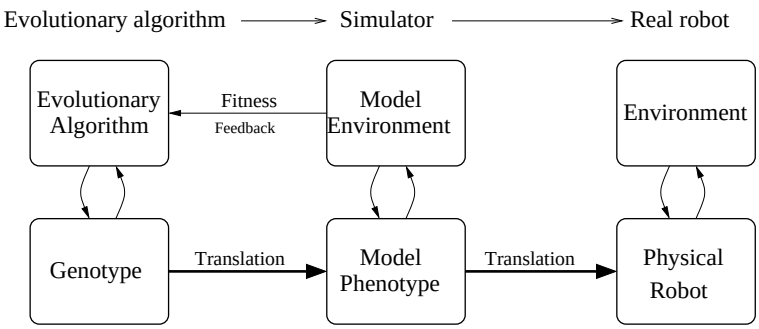


Figure 3: *Components of an evolutionary robotics system using simulation to evaluate controllers.*

Name: Shaking Simulation

Context: An evolutionary robotics system has been implemented, which uses a simuator for evaluating candidate controllers. The system has been tried, and it has been found that the evolved controllers perform poorly once transferred to the real robot, even though they perform well in the simulation.

Problem: When using a simulation approach to evolutionary robotics, it often happens that robots evolved in the simulator performs significantly worse in the real world, or maybe even fails completely. It is impossible to write a perfect simulation of any real environment. Evolved controller will very likely rely on aspects of the simulated environment that are not present in the real world, such as the walls having specific colors or the robot having perfect wheel odometry, or possibly even bugs in the simulator. This becomes evident when a controller evolved in simulation is transferred to the real robot. This step is referred to as *crossing the reality gap* [10] [4]. How how do we make controllers cross the reality gap?

Forces: One must consider the complexity of the environemnt, "Discreteness" of robot and environment, reliability of sensors and actuators, performance fluctuations due to varying battery levels, varying light conditions. In short: all the dirt that makes reality different from nice and tidy computer models.

Solution: Apply noise to all levels of the simulation. The noise should be applied "adequately" and be "correctly profiled" [5]. The profile and level can be

obtained from repeated samplings of the world through the robots own sensors and actuators. To increase robustness further, the profile of the noise should be varied during evolution. It is important that the noise is such, that the evolved controller cannot come to rely on certain aspects of the simulation that do not apply in the real world.

Rationale: The problem arises due to the fact that it is impossible to write an 100% accurate model of the real world, combined with the fact that evolved robot controllers tend to exploit any aspect of their environment that can be exploited. If the simulation offers opportunities for exploitation that do not exist in the real world, the evolved controllers will most likely fail when transferred to the real world. Injecting noise into the simulation makes evolved solutions more robust. The noise serves two purposes. Firstly, the noise models all of the aspects of the real world that are below the fidelity of the simulator. Secondly, the noise blurs effects caused by bad or wrong models of the world.

One danger of applying noise to the simulation is, that it might blur the robots sensing to a degree that renders the robot unable to perform the task at all.

Resulting Context: The simulator now has noise applied to it, forcing evolved controllers to cope with uncertainty, thereby increasing the cahnch that the evolved controllers can cross the reality gap and perform the task in the real world. However, the increased complexity of performning the task in the simulation due to the added noise might delay or halt the evolutionary process. Applying additional shaping (see *Shaping Evolution*) might help overcome this problem.

Related Patterns: Artificial Evolution of Real Robots, Simulated World and Shaping Evolution.

Acknowledgment

Thanks to Daniel May (Maersk Institute, University of Southern Denmark) for shepherding this paper, and to my workshop group at the VikinPLoP conference.

References

[1] Rodney A. Brooks. Evolutionary robotics; where from and where to. *Evolutionary Robotics: From Intelligent Robots to Artificial Life ER'97*, pages 1–19, 1997.

[2] Dario Floreano. Reducing human design and increasing adaptability in evolutionary robotics. In T. Gomi, editor, *Evolutionary Robotics*. Ontario (Canada): AAI Books, 1997.

[3] I. Harvey, D. Cliff, and P. Husbands. Issues in evolutioanry robotics. In Roitblat, H. Meyer, J.-A. and Wilson, S., editors, *Proceedings of SAB92*. MIT Press Bradford Books, Cambridge, MA, jul 1992.

[4] Nick Jacobi. Running across the reality gap: Octopod locomotion evolved in a minimal simulation. *Lecture Notes in Computer Science*, 1468:39–??, 1998.

[5] N. Jakobi, P. Husbands, and I. Harvey. Noise and the reality gap: The use of simulation in evolutionary robotics. *Lecture Notes in Computer Science*, 929:704–720, 1995.

[6] Bart Kosko. *Neural Networks and Fuzzy Systems, a dynamical systems approach to machine intelligence.* Prentice Hall International, 1992.

[7] Wei-Po Lee, John Hallam, and Henrik Hautop Lund. Learning complex robot behaviours by evolutionary approaches. *6th European Workshop on Learning Robots, EWLR-6*, aug 1997.

[8] Henrik Hautop Lund and John Hallam. Sufficient neurocontrollers can be surprisingly simple. *Research paper no. 824. Department of Artificial Intelligence, University of Edinburgh*, 1996.

[9] Zbigniew Michalewicz. *Genetic Algorithms + Data Structure = Evolution Programs.* Springer Verlag, Berlin, 1999.

[10] Orazio Miglino, Henrik Hautop Lund, and Stefano Nolfi. Evolving mobile robots in simulated and real environments. *Artificial Life 2*, pages 417–434, 1995.

[11] Rolf Pfeifer and Christian Scheier. *Understanding Intelligence.* The MIT Press, Cambridge Massachusetts, 1999.

[12] Adrian Thompson. Artificial evolution in the physical world. In *Evolutionary Robotics: From Intelligent Robots to Artificial Life (ER'97)*, pages 101–125. AAI Books, 1997.

Software Development Processes and Organization

Patterns for the Role of Use Cases

Gertrud Bjørnvig
Microsoft Business Solutions
(Formerly Navision)
Frydenlunds Allé 6
DK - 2950 Vedbæk
+ 45 45 67 94 53
gbjrnvig@microsoft.com

Content

Version: VikingPLoP 2002 proceedings.

Last edited 15 January, 2003

Introduction

These patterns are about use cases. Or maybe they are not…

When I returned from VikingPLoP my mind was preoccupied with two comments from the writers' workshop. The first comment was something like: *Take a look at Steve Adolph's "Patterns for Effective Use Cases". It has just been published*. Aha! A whole new book with use case patterns! Interesting. When I began to work with my patterns I tried to figure out if there was any other pattern work going on in relation to use cases; I mean with such a proven technique I couldn't be the only one. And I wasn't, but I didn't know about the book. So what did that mean for my use case pattern work?
The second comment was *Maybe this isn't about use cases*. Not about use cases? I was confused!

I bought the book [1] to see how their patterns were related to my patterns. Were they overlapping? Were they complementary? Or were they something different?

First of all: It is a good book! I can recognize most of the patterns.
When I tried to compare these patterns with my patterns the statement *Maybe this isn't about use cases* began to make more sense. The majority of the effective use case patterns are about the more technical part of writing use cases. Patterns like *BreadthBeforeDepth*, *MultipleForms*, *VerbPhraseName*, and *LeveledSteps* are all addressing good techniques for writing use cases. It seems like my patterns are more about the use cases' role in the development process culminating in the pattern *UseCaseAsCenter* and the candidate pattern *SizeTheIterations*. I know this is an overall consideration. There are several effective use case patterns addressing the process, for example, *SpiralDevelopment*. And my pattern *NarrativeAndGeneric* is an example of a pattern addressing a more technical issue. So the conclusion is not that simple; in some degree the patterns are overlapping, in some degree they are complementary, and in some degree they are something else.

Where does this bring me?

I have changed the name from "Patterns of Use Cases" to "Patterns for the Role of Use Cases".
And then I have chosen to use and refer to the effective use case patterns where I thought it would improve my patterns. In one case – *GoalsDefineNumber* – I actually use four of the patterns from the book to describe the solution, and I hope it has improved the pattern.

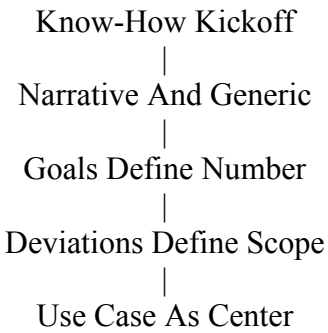
Somehow it feels like my patterns act like a middle layer between Jim Coplien and Neil Harrisons's organizational patterns [6] and the effective use case patterns. We need organizational patterns as *UnityOfPurpose* and *HolisticDiversity* in order to utilize the potential of the use case technique. But we should also be able to manage the different parts of the technique that the effective use case patterns cover, such as naming the use case (*VerbPhraseName*) and leveling the steps properly (*LeveledSteps*).

I am referring to *Navision* in the paper, despite that the name of the company has changed. Primarily because the things referred happened when the name was Navision. But it is actually still correct since *Navision* now is the name of the product line where most of the referred use case experiences are captured.

Summary

Know-How Kickoff:	Begin the use case work.
Narrative And Generic:	Separate detailed story telling from functional requirements.
Goals Define Number:	Limit the number of use cases through goal-oriented use case definition.
Deviations Define Scope:	Optimize the knowledge of the use case scope.
Use Case As Center:	Let the use cases be the teams' center for feature development.

The natural sequence of the patterns follows the order:



The Patterns

Know-How Kickoff

… you have decided to apply use cases in the project.

***

The use case work is stuck before it is started; the use cases seem uninteresting or the team cannot see why they should do use cases.

Sometimes when we begin applying use cases we are guessing more than identifying and analyzing. We can only define one actor "the user" because we really don't know who the actors are. Or we define detailed and trivial use cases such as "Update Customer Information" or "Set Up Vendor Type" instead of focusing on the real problems we want to address. Maybe we haven't been able to involve people with sufficient knowledge or maybe we're trying do use cases too early. The result is use cases that aren't substantial enough to encourage further exploration.

We can also begin too late, then it can be hard to get team members' attention and participation, as I realized in this situation:

> The whole team is gathered to a use case session. A drafted design exists. The project manager and the usability specialist think that lack of use cases is a problem for the project. The developers are eager to begin coding. The product managers are eager to see results. They feel that they know what they are going to develop and cannot see any reason for doing use cases now. Most of the session is spent on discussing why we should do use cases now.

It was too late for the team. It was obvious that most of the team members had a clear picture of what they were going to do – but they didn't have the same picture! And they didn't want anything or anybody to cloud this picture; "I know what to do, and I don't want anything that can delay progress now."
If a team does not have consensus about the purpose of the project, it can be very hard doing use cases. The lack of consensus can be an obstacle for the work – especially later in a project.

So we have to get the team's attention in time, and we need to identify the use cases that really matter.

Therefore:

Before starting any design activities: Gather persons with know-how about market, customers, users and domain to a use case work session. Base the session on a product vision or business case.

A well-timed use case work session – with qualified know-how represented – is an effective way to ensure that we identify the important use cases from the beginning. And it is a good kickoff for the subsequent use case work.

***

It is a good idea to *EngageCustomers* [6] in the session. If it isn't possible, ensure that you have a *SurrogateCustomer* [6] role to fill in the role of the customer. At Navision we have a product manager role as a part of the team. The product manager represents the customer, and it is crucial for the success of a *Know-HowKickoff* to have this role present.

It is preferable that the majority of the participants are project team members – it can give the nice side effect of a mutual buy-in to the use cases. It could also be other stakeholders; it will support the principle of *ParticipatingAudience* [1] where you involve customers and internal stakeholders in the use case work. But avoid having too many people in the session – no more than ten. It can be very effective with just three people, if they are the right people.

My experience is that with the right people – representing sufficient knowledge about the project – in the room, you can identify the 5 to 12 most important use cases for a project in a few hours. Of course, this depends on the number of people and the scope of the project.

A use case diagram is effective as a mean to communicate and achieve consensus about the overall goals and scope during the first use case session.
During the session is it also a good idea to list all open issues and questions, instead of trying to resolve everything right away. It reduces the time used on discussions, and it prevents too much guessing. Such a list will be your first version of a *ReadinessReflectionList* (see "Candidate Patterns").

UnityOfPurpose [6] is often a precondition for a successful *Know-HowKickoff*, but it can also go the other way – the preliminary use case work can help achieve *UnityOfPurpose*.

Know-HowKickoff uses the principle of *HolisticDiversity* [6]. It gathers a few people with diversified skills and lets them communicate directly. The same goes for *DiversityOfMembership*[i] [6] where the principle is used for requirement teams. A similar pattern is *BalancedTeam* [1].

The Navision feature team model supports these patterns very well; the team core roles (product manager, developer, tester, user assistance, usability, project manager) are gathered in the same team from the beginning of a project. The team is sitting physically close to each other. The Navision feature team model is based on the team model described in Microsoft Solutions Framework [9].

There is also an element of *Lock'EmUpTogether* [6]; we need to gather different people in the same room to do a very focused task.

A *Know-HowKickoff* can also work as a kind of working *Face-To-FaceBeforeWorkingRemotely*, where you "begin a distributed project with a face-to-face meeting for everyone [6]." A typical distributed project involves geographic

distance and different time zones. But even a short geographic distance as the one between buildings and floors can have a challenging impact on the communication. Not to mention the mental distance there can be when roles belong to different organizational units with different value set – for example business roles and development roles. A project involving subcontracting is another example of a distributed project where a *Know-HowKickoff* can work as a kind of working *Face-to-FaceBeforeWorkingRemotely*.

These dimensions of *Know-HowKickoff* make it support *UseCaseAsCenter*.

Know-HowKickoff is a good beginning, but it doesn't ensure further progress in the use case work in itself. The next challenge is to write the use cases. One way to handle this is to establish a *SmallWritingTeam* [1] who are writing *NarrativeAndGeneric* use cases with *ConsolidatedNormality* (see "Candidate Patterns"). Review their work through a variation of *GroupValidation* [6] with focus on requirements instead of design. It can be done as a *TwoTierReview* [1] where the review process is divided into two types of review: First one or more smaller reviews with few people involved, and then a group validation with the complete group. Don't forget that *DeviationsDefineScope*.

Narrative And Generic

… use cases capture the functional requirements. A use case has been identified in a *Know-HowKickoff,* but not yet described.

***

Often use cases are too detailed to be useful as functional requirements, or they are too abstract to be understandable.

Business people tend to have a lot of details in their use case descriptions. They like to tell the story.

Domain experts tend to write more abstract, sometimes too abstract, use cases. Everything is so familiar that things become implicit.

Developers tend to begin design in the use case description. They are focusing on the solution.

We love stories and details, but we need precision and brevity.

Therefore:

Begin the use case with a narrative that illustrates the intention of the use case. The concrete details that are good in a narrative can then be separated from the more generic functional requirements where too many details are disruptive and cause imprecision and inconsistency.

The use cases can concisely describes functional requirements and at the same time, we can get a clear picture of a real context of use. This can be seen in the following example:

> Part of narrative: Christian finds the customer through the handheld device and checks out the name of the contact person at the company, John Jensen. The last time he visited John, he was just about to celebrate his 25th jubilee. Christian wants to ask about that at the meeting.
> Corresponding part of use case: The sales person checks information on contact person.

***

To make qualified narratives (and qualified use cases!) you need sufficient data about the users' situation. The usability domain offers several good techniques of how to gather valid user data – for example from site visits.

My experience from use case workshops is that it can be hard for a team to describe a use case from scratch. But when we start with a narrative we get a common picture

that makes it easier for the team to describe the more generic part of the use case. I have not seen it work the other way around; when the generic part of the use case has been written it seems to be very hard to tell the story.

One team in Navision describes their vision as a story presented as a role-play and documented in power point slides. It was easy to derive use cases from that story, since everyone had a common picture of the idea before the use cases were identified.

Many use case descriptions in Navision are supplemented with a storyboard as the narrative.
Most narratives include at least one deviation from the main scenario. Narratives can also go across use cases.

Narratives can give life to actor descriptions as well. Describe a real user by giving a user profile with age, name, tasks etc. to supplement the more generic actor descriptions.

NarrativeAndGeneric can help us write *PreciseAndReadable* use cases that are "readable enough so that the stakeholders bother to read and evaluate it, and precise enough so that the developers understand what they are building [1]."

Goals Define Number

… the candidate use cases have been identified through a *Know-HowKickoff*. Narratives illustrate the intention of the use cases.

***

The project has too many use cases. The use cases don't give any overview. Team members get lost in details.

Your background before use cases will probably decide what traps you risk falling into.
What inspires you to do use cases? Visual modeling and UML? Or did you seek an alternative to the traditional requirement specification? Are you a modeler or a writer?

The modeling trap is decomposition – an extension here and an include there, and you end up with 50 use cases before the day is over.

The writing trap is completeness – the solution is described in detail and repeated in many use cases. The result is the same as in the modeling trap: you miss the important goals and have too many use cases.

Too many use cases mean a lack of overview and a lack of clarity as for what is important.

Therefore:

Keep only those use cases that support the user in meeting their work-based goals and the corporate business goals.

You need to identify the *UserValuedTransactions* which are "the valuable services that the system delivers to the actors to satisfy their business purposes [1]." Knowing the *UserValuedTransactions* can help you define a *CompleteSingleGoal* [1] for each use case. Ensure *ForwardProgress* [1] by eliminating or merging steps that do not advance the actor. *MergeDroplets* by "merging related tiny use cases or use case fragments into the use cases that relate to the same goal [1]", and *CleanHouse* [1] by removing those use cases that do not add value to the system.

You can also prioritize your use cases in order to identify the *FocalUseCases* [3]. It is the use cases that are most important to users, but the prioritization also includes other parameters, such as stakeholders' interests and risks expressed by development.

Focusing on user goals help us to reduce the number of use cases, and it helps us to deliver useful software!

***

Finding the right user goal level for a use case can be hard. One project in Navision handled it this way:

> The overall goal is formulated as a positioning statement: "Returns Management transforms customer dissatisfaction into customer satisfaction". One of the *UserValuedTransactions* is to "Register Compensation Agreement with Customer". This transaction satisfies a *CompleteSingleGoal* for the user and the business, so it is defined as one of the use cases. A part of this transaction is to "Register compensation agreement for a special item going to be repaired by the vendor". It could be seen as a *CompleteSingleGoal* in itself, but it is also a sub-goal to "Register Compensation Agreement with Customer".

Having sub-goals like this at the step level in the use case can help us manage the number of use cases. I define steps broadly as steps in the main scenario plus steps representing deviations from the main scenario, including extensions. Very few extensions need to be described as full use cases. A row in the deviation list will often be sufficient.

To solve the problem of too many CRUD (Create/Retrieve/Update/Delete) use cases in administrative systems, you can use parameterized use cases as described by Alistair Cockburn [5]. Alistair Cockburn was also the first to describe the principle of goal oriented use cases [4].

GoalsDefineNumber helps you to constrain your number of use cases. A controllable number of use cases is a precondition for having *UseCaseAsCenter*. My experience is that a project should have no more than 15 use cases for a 6-9 month time period with a team size at no more than ten people. See *SizeTheOrganization* [6] and *SizeTheIterations* (see "Candidate Patterns").

Deviations Define Scope

… the core use cases have been identified and are characterized by *GoalDefineSize*. The use cases are *NarrativeAndGeneric* with *ConsolidatedNormality* (see "Candidate Patterns") as the main scenario.

***

You don't feel that you can base your time and resource estimations on the use cases, or it has taken much longer to implement a use case than expected.

How do you estimate your use cases?

This question can actually mean two things: 1) How do I estimate? 2) How do I ensure that the use cases provide a good foundation for estimation?

If it is the first question, the use cases cannot help you. It is just as hard to estimate use cases as anything else.

The second question is more relevant. We feel that the use case is finished when we have written the main scenario. It represents the core functionality, so we think we can go on with design and implementation. But we realize that the main scenario is a minor part. All the extra things that the user should be able to do in this context must be implemented. And all the things that can go wrong must be taken care of, too. Sometimes these things have an impact on the design that we haven't even been aware of. So at the end of the day, we have spent much more time on implementing all the things that have *not* been a part of the main scenario than on the main scenario itself. If we aren't aware of it in beforehand we have a very poor basis for estimation.

A use case with nothing but a main scenario – no extensions, no exceptions, and no variations – is only a beginning. The main scenario is the framework for deviations – the *ConsolidatedNormality* (see "Candidate Patterns") – and represents in many cases less than the half of the scope. Each step in the main scenario is a potential source for several deviations.

At the same time we can't know everything before we start coding. Some things will not be discovered before we begin to test. But we still have to optimize our knowledge about the scope to estimate and prioritize. We have to be able to prioritize in order to meet our schedules.

Therefore:

Define the potential scope of the use case by listing the deviations from the main scenario in one or more deviation identification sessions. For each step in the main scenario ask questions like: What can go wrong here? What else does the user want to do here? What is happening if…?

Involve different roles in order to get a qualified list of deviations. A tester is good at focusing on things that can go wrong. A usability expert is good at focusing on user experience. This is the principle of *HolisticDiversity* [6].

The deviation list provides a realistic feeling for the potential scope of a use case. It can help us decide if we want everything on the list in scope or not, and we improve our basis for estimating.

***

Deviations can be called variations, extensions, exceptions, or something else. I see no practical benefit of separating the different kinds of deviations into different categories.

At Navision, we have had good experiences with gathering domain experts in a workshop where a first draft of the use cases – including the main scenario – is used as a basis for structured deviation identification.

ExhaustiveAlternatives has a similar solution: "Capture all alternatives and failures that must be handled in the use case [1]." But the focus here is more to avoid that the developers will misunderstand the system's behavior, so the system will be deficient. *DeviationsDefineScope* focuses more on how to manage the scope. Anyway; achieving both can only be good!

Sometimes it isn't worthwhile doing the deviation identification for all use cases at once. It is sufficient to do it for the use cases for the next development iteration. This is closely related to *SpiralDevelopment* where the use cases are "developed in an iterative, breadth-first manner, with each iteration progressively increasing the precision and accuracy of the use case set [1]." See also *SizeTheIterations* (see "Candidate Patterns").

To implement *UseCaseAsCenter* you need *DeviationsDefineScope*.

Use Case As Center

… the team has a shared understanding of the goals and the scope of the project achieved through *Know-HowKickoff*, *NarrativeAndGeneric*, *GoalsDefineNumber* and *DeviationsDefineScope*.

***

You have invested the effort to do use cases, but during design and coding you realize that the use cases are not used. Things are not implemented consistently with the agreements documented in the use cases. Changes are not documented, and maybe not communicated at all.

Very few people like updating existing documentation. Especially if the document has been a part of an approval procedure, where the goal more is to get the approval than to have a useful document during development. But outdated use cases are not useful for coding, testing, or user documentation purposes.

Use cases are often used in the beginning of a project to capture agreements about direction and then they die. This can be okay and has a value in itself. But it means that it can be hard to know what is being implemented and why. The developers may be the only ones who are updated, and other roles have to disturb them in order to do their job.

I have heard statements like "Use cases are only for developers" or "We only do use cases in order to do test cases". If the use cases are written for a specific role – and probably by that role, too – this role, and nobody else will use them.

Other statements like "I cannot test from these use cases" or "It isn't possible to design from the use cases" are signs of a role that hasn't been involved in developing the use cases, but now is expected to use the use cases as a basis for their work. Involvement is everything – and *ResponsibilitiesEngage* [6].

Use cases have the potential to be the repository of important agreements about functionality, but often they end up being a one-man or one-role show or they just die.

Therefore:

Let the use cases be the center or placeholder for other work products. Ensure that all team roles contribute to the use cases. Organize test cases around use cases, insert references between for example use case and user interface, list tasks in relation to use cases, and plan the development iterations based on use cases.

If other work products are organized around the use cases the team will have a common interest in having readable and updateable use cases.

When use cases synchronize the work of the team, they become the natural driver of the development iterations.

Use case driven development is an effective way to control iterative and incremental development, and if you succeed doing it you will get the full benefit of your investment in use cases.

***

Having *UseCaseAsCenter* can help preventing the problem of deceased use cases, but it can be very hard to repair the situation.

I have only seen real use case driven projects when the leader (formal or informal) of the team wants it like that. It has to be planned that way.

Putting the *UseCaseAsCenter* certainly has its strengths. An outsourced development project illustrates this:

> At Navision a development project was outsourced to a development partner. The project manager from the partner and the project manager from Navision met to plan the development based on use cases. The project manager from the partner should prepare by listing tasks. He has made a list, but hasn't related them to the use cases. We wrote the use case names on posters, and did the same with all the tasks. Each task was then placed below a use case, and what surprised us all was that it was possible to relate every task to a use case, even though they were identified independently of the use cases.
>
> Then the use cases were grouped into iterations. At first the project manager from the partner thought that there were too many dependencies to divide the development into several iterations, but we realized that a lot of these dependencies weren't a real problem. Maybe some things had to be implemented as stubs (for example a piece of code that simulates a non-implemented function) in the first iterations, but it could be done. The advantages outweighed the disadvantages. We planned five iterations, executed three, and the project was delivered on time.

Architecture and system design will go across use cases and is not naturally linked to use cases. Some projects need a more technology and architecture driven approach, but the use cases will still be a good way to connect the development work with test and user documentation (a user doesn't have to be an end user – some times the user is another developer).
In *FeatureAssignment* features are assigned to people for development, but it is emphasized that this should be coupled with the role of *CodeOwnership* (each code module in the system is owned by a single developer) to "strike a balance between maintaining architectural integrity and getting the job done [6]." The same is true for *UseCaseAsCenter*; you need a balancing mechanism to ensure architectural integrity. Use case driven iterations can be used to avoid designing more than what is actually needed for a given iteration. See also *SizeTheIterations* (see "Candidate Patterns").

Adornments [1] have similarities with *UseCaseAsCenter* by recommending that nonfunctional information should be associated with the use cases in supplementary sections.

There is a relation between *UseCaseAsCenter* and *WorkFlowsInward*. *WorkFlowsInward* means that "Work should flow in to developer from stakeholder, especially customers. Work should not flow out from managers [6]." The use cases are an agreement between stakeholders and the team about the functionality of the system. When we have *UseCaseAsCenter*, most of the work can be derived from the use cases instead of being generated by a manager role. It means that *UseCaseAsCenter* can support that *WorkFlowsInward*.

The principle in *UseCaseAsCenter* is essentially the same as Ivar Jacobson's *A Use Case Driven Approach* [7] [8]. While Jacobson is advocating for a modeling approach with traceability through the different models, *UseCaseAsCenter* is focusing more on use cases as a common center for the team's work.

You need *UseCaseAsCenter* to *SizeTheIterations* (see "Candidate Patterns").

Candidate Patterns

During the work with the five first patterns I have identified three candidate patterns.

Consolidated Normality

The strength of the use case description structure is related to the relation between the main scenario and the deviations. This structure allows us to handle complexity in a simple and consistent manner. But it requires that we are able to define the "normal" scenario – without any "if's" at all. Many use case authors have a hard time doing that, but my experience is that it is always possible to define the normal – and it is necessary in order to get structured use case descriptions.

This is similar to *ScenarioPlusFragments* [1]. Maybe it is too much overlap to write this pattern? Express the *ConsolidatedNormality* as *LeveledSteps* [1].

Readiness Reflection List

It is very effective to maintain an open issue list per use case (plus one in general). The character and number of open issues gives us a good picture of the situation. Is it a big issue? Is it a minor issue? How long is the list? It is always hard to know when we are ready to begin a new activity. The open issues at the list can help us to reflect about our readiness; look at the issues and you will probably know if you are ready. Ready for design, ready for code, ready for test, ready for shipping…

A *ReadinessReflectionList* can also help us identifying the areas where we have high confidence as we need to *GetOnWithIt* [6].

Size The Iterations

Empirically it seems like we shouldn't have more than 15 use cases for a project (or sub-project), divided into 3 to 5 development iterations, based on 1 to 3 use cases each. I don't know why it works that way – but I guess it is a pattern.

SizeTheIterations builds on the principle of *SpiralDevelopment* [1].

Acknowledgements

First of all I would like to thank Jim Coplien for opening my eyes to the pattern world beyond the more technical patterns related to (system) architecture and object-oriented programming. It has been a great source of inspiration. It inspired me to begin the work of writing these patterns. And it also inspired me to spend three days in the shadow of a palm with a view to the Greek sea reading Alexander's "The Timeless Way of Building". It was after I had finished the patterns for VikingPLoP. My first impulse was to go home and rewrite it all! I didn't have time for that, but I got great insights in the philosophy of patterns and pattern languages.

Thanks to the teams in Navision that have provided me with the stories and examples that I'm using in the patterns – I hope you find that I have used the material in a decent manner.

Special thanks to my Navision colleagues Dan Henriksen, Susan Wiingaard, and Diana Velasco for commenting on early versions of the patterns.
And to Brian Jay Godkin for helping me improve the English language as well as the content for this version of the patterns (any inappropriate use of the English language is most likely caused by my last moment updates!).

And thank you to all the nice people who workshopped the patterns at VikingPLoP; I appreciated your comments very much.

Most of all I would like to thank my shepherd, Neil Harrison, who guided me through the challenging work of improving the patterns up to VikingPLoP.

References

[1] Steve Adolph, and Paul Bramble. *Patterns for Effective Use Cases*. Addison-Wesley 2002.
[2] Christopher Alexander. *The Timeless Way of Building*. New York, Oxford University Press 1979.
[3] Robert Biddle, James Noble, and Ewan Tempero. *Patterns for Essential Use Cases*. Technical Report CS-TR-01/02, April 2000.
[4] Alistair Cockburn. *Structuring Use Cases with Goals*. JOOP September and November 1997.
[5] Alistair Cockburn. *Writing Effective Use Cases*. Addison-Wesley 2001.
[6] Jim Coplien and Neil Harrison. *Organizational Patterns*. Org Patterns web site January 2003: http://www.easycomp.org/cgi-bin/OrgPatterns?BookOutline
[7] Ivar Jacobson et al. *Object-Oriented Software Engineering*. Addison-Wesley 1992.
[8] Ivar & Sten Jacobson. *Use case Engineering: Unlocking the Power.* Object Magazine October 1996.
[9] *Microsoft Solutions Framework*. MSF Resource Library web site January 2003: *http://www.microsoft.com/technet/treeview/default.asp?url=/technet/itsolutions/tandp/innsol/msfrl/default.asp*

[i] *This pattern cannot be found directly from the link in the references section, but it can be found here: http://www.easycomp.org/cgi-bin/OrgPatterns.book?HarrisonPatterns*

Agile Environments – Some Patterns for Agile Software Development Facilitation

Klaus Marius Hansen

ISIS Katrinebjerg
University of Aarhus
Aabogade 34
DK-8200 Aarhus N

Abstract

Agile software development demands an agile working environment in contrast to traditional, specification-oriented software development. Pair programming, continuous user involvement, and ad-hoc meetings are examples of work practices that require facilitation for new ways of working. We present patterns that facilitate the creation and evolution of a working environment for agile development.

1 Introduction

Many system development projects are facing uncertainty with respect to which solutions to create, how to create these solutions, or even which problem to solve. Traditional prescriptive and specification-oriented development processes such as the early object-oriented development methods (Booch, 1991), (Jacobson et al., 1992), (Rumbaugh et al., 1991) fail to address such commonly occuring project situations.

There have been continouous reactions to such (earlier and later) processes starting with the early participatory design work (Nygaard and Bergo, 1974), going over Joint Application Development (Wood and Silver, 1995), to current methods and processes such as Adaptive Software Development (Highsmith, 2000), Crystal (Cockburn, 2001), Dynamic System Development Method (Stapleton, 1997), Extreme Programming (XP; (Beck, 1999)), Feature-Driven Development (Coad et al., 1999), and SCRUM (Schwaber et al., 2002). These processes share a number of

Email address: marius@daimi.au.dk (Klaus Marius Hansen).

characteristics: They aim at providing working software in uncertain and complex situations through a focus on, e.g., iterative development, incremental addition of functionality, and active stakeholder involvement.

Based on the observation of such common characteristics, the agile manifesto (Beck et al., 2001) establishes four fundamental values for agile software development:

- *individuals and interactions* over processes and tools,
- *working software* over comprehensive documentation,
- *customer collaboration* over contract negotiation, and
- *responding to change* over following a plan.

The environment, physical and virtual, in which agile software development is performed is important in that it may contribute significantly to working with these values: A work environment that favors personal communication may help facilitate individuals and interactions, the physical environment may help developers focus on developing thus aiding in making working software, locating a development team close to a customer site may enhance customer collaboration, flexible arrangements of project rooms may help in responding to change by allowing dynamic reconfiguration. Since the principles of agile software development go back to the early days of participatory design and since the work environment may make a significant difference to agile software development projects, it makes sense to capture and express the experience of agile software development faciliation, i.e., the practices of helping bring about agile software development easier, through. Thus, this paper tries to do that through the formulation of a set of patterns for agile software development facilitation.

1.1 Form

The major part of this paper consists of a set of patterns for agile software development facilitation. They are written for software developers and managers, a distinction that may become blurred, however, in agile software development.

For the description of patterns, we use a slight variation of the narrative Portland Form (Cunningham, 2002): Each pattern name is followed by a problem and a set of associated forces for that problem. The solution to the problem that the pattern represents is started by a *Therefore:* and a short solution. Finally, the resulting context is discussed. References to patterns are shown in italics.

The last part of this paper consists of a problem/solution summary of the presented patterns.

2 Patterns for Agile Environments

The following patterns provide a set of starting points for relating experience in agile software development facilitation. Currently, the patterns are divided into four categories:

- *Context* encompasses patterns that are not specific to facilitation within agile software development, but are prerequisites for agile faciliation.
- *Place* concerns social properties of locations: One location may be used for concerts during the week and church ceremonies on Sundays yielding different places (Harrison and Dourish, 1996). The patterns in this category are then concerned with facilitating social aspects of the physical development environment.
- *Space* is concerned with the physical properties of locations: How much room is there, how is the lightning, where is the furniture placed etc (Harrison and Dourish, 1996).
- *Detail* contains faciliation patterns that are additions to the Space and Place patterns.

Within each category, the patterns are organized according to importance.

2.1 Context

2.1.1 Team Chooses

You need to provide the best working environment for your development team. Each project is unique. Your team is motivated and dedicated since they will be working in an agile manner, moreover you may be working with a process (such as XP) that requires and should maintain high discipline.

Agile software development favors individuals and interactions.

Therefore: The team decides how their work environment should be and sets rules for how development should be facilitated.

This pattern is at the core of agile software development: responding to change requires that individuals are able to make changes as required.

With respect to facilitation, this means that the team - including a project manager and possibly a customer - should be able to decide how to balance the forces of the subsequent patterns in this collection. If your process requires high discipline (such as XP) this is not in contradiction to that. This pattern just acknowledges that

project members are competent professionals and allow them to inform their own work situation.

The agile values should constrain what the team is able to decide. Focus on individuals and interactions, working software, customer collaboration, and responding to change is mandatory and decisions on faciliation should not compromise this. Having the *Customer Close By* (Section 2.2.2) or *One War, One Room* (Section 2.2.1) are examples of patterns that are almost required when applicable.

2.2 Place

2.2.1 One War, One Room

Developers and stakeholders on your project need to communicate as effectively and efficiently as possible. The frequency of and need for communicating is high. The team is small.

Therefore: Place developers in the same room. Also put *Customer Close By* (Section 2.2.2).

For large projects it is impractical to place all developers in the same room, but for smaller projects (at least up to ten developers), it maximizes the potential face-to-face communication and awareness. This arrangement is also ideal for pair programming. If corporate culture makes it impossible to have such a room, creating a persistent design room (Beyer and Holtzblatt, 1997) in which design meetings and, e.g., *Writing on the Wall* (Section 2.2.3) may take place, may serve as a compromise. Geographically separated teams makes faciliation more difficult.

Even if the team is heterogeneous through having, e.g., object-oriented developers, ethnographers, usability experts, and customers on it, co-location may work very well (Christensen et al., 1998).

Care must be taken in also providing space for more private activities, such as reading and phone calls through, e.g., *Public and Private Spaces* (Section 2.3.2).

2.2.2 Customer Close By

Stakeholder requirements are initially vague and are constantly prone to change. You need the development team to discover and keep in sync with requirements. The customers needs to know the progress of the team.

Therefore: Place the customer and developers close by and make room for that.

If customers are co-located with, or readily accessible to, the development team, there should be a workspace for the customer for doing ordinary work. The development team should have access to the customer at all times. XP advocates this kind of arrangement.

A complimentary approach is to locate the development team within the customer organization. If this is possible, this is more effective in terms of learning about and designing for the practice of the customer (Christensen et al., 1998). DSDM advocates this kind of arrangement.

This kind of stakeholder involvement in the project will probably not suffice. It may, e.g., be beneficial to include ethnographic studies of real work when dealing with complex problem domains (Christensen et al., 1998).

The nature of the project may make customer co-location difficult: for contract development it may be hard economically to convince the contracter of this kind of arrangement. For in-house development co-location may be easier, in particular if provisions can be made for the customer to work effectively on her day-to-day work while located near the development team. For off-the-shelf products, it may be impossible to be co-located with customers.

2.2.3 Writing on the Wall

Developers and customers should have easy access to writing material for discussions, thinking, and visualization. You want to maintain awareness and access to the results. You want to capture ideas and suggestions as they evolve.

Therefore: Place whiteboards and paper for writing in development rooms.

Make sure you have dedicated space for these activities so that it is possible to quickly go to and from such discussions without interrupting the work of others.

For whiteboards, you will need plenty of space for sketching and modelling. Also, if you have *Space for Pairs* (Section 2.3.4) you may consider having a whiteboard for each pair. An option is also to use brown paper or even whiteboard wallpaper to cover an entire project room with writing surface...

Mynatt (1999) investigated the use of whiteboards in a personal and a public office setting. She points to four characteristics of office whiteboard use:

(1) Whiteboards are effective for thinking and pre-production tasks: ideas and thoughts such as how a graphical user interface should be designed or which responsibilities a class should have can quickly and easily be written or erased.
(2) Use of whiteboards leads to clusters of persistent and short-lived content: most of what is written on a whiteboard is really ephemeral, but some content such

as to do list persist for much longer.
(3) Whiteboards contain everyday content as well as formal drawings.
(4) Whiteboards are and can be used for information that ranges from being semi-public to private.

An extensive use of whiteboards or other analog writing material may cause problems, however: content cannot be saved or restored and can only be manipulated in very primitive ways. For saving, and partly for restoring, a digital camera may help. It is common practice to capture contents of whiteboards using such means and later use the images for recall or for more formal documentation (Andersen et al., 2000). If you frequently need to update your content, you will either need massive amounts of whiteboard space or these tools may be too *Simple Artefacts* (Section 2.2.4). Another problem is that whiteboard and paper are not good at documenting precise decisions due to their ephemeral character. Also, although these materials are better at representing the development of a solution, or discarded proposals, they are not good at capturing rationales.

Using electronic whiteboards can help overcome some of the weaknesses of analog whiteboards. Electronic whiteboards capture the pen input made by users and transfers this input to a computer. There are two major variants of electronic whiteboards: Whiteboard replacements, such as the SMART Board (http://www.smarttech.com) are complete, standalone replacements of traditional whiteboards. They are typically combined with a computer and a projector so that a computer image may be projected onto their screens. These technologies are expensive and mostly applicable if the writing requires a lot of interactivity. Whiteboard augmenters, such as the Mimio (http://www.mimio.com), can be plugged onto any existing whiteboard after which they can capture what the user draws on the whiteboard if the special pens are used. They can be used either in non-projected or projected mode. In non-projected mode, the whiteboard acts just as an ordinary whiteboard, but enables the saving of drawings on the whiteboard. In projected mode, the whiteboard augmenters work much like the electronic whiteboard replacements. Whiteboard augmenters are much less expensive than whiteboard replacements and enable the reuse of existing whiteboards. The are, however, more problematic to set up, particularly in projected mode. In either case, if project mode is to be used, consider using *Simple Artefacts* (Section 2.2.4) in this mode.

Ambler (2002) introduces the Agile Modeling methodology which has a number of detailed suggestions for supporting object-oriented modelling in an agile manner by organizing project rooms with plenty of simple, readily available, and flexible writing surfaces and writing material. Concretely, Ambler advocates that a room to be used for modelling should contain dedicated space, significant whiteboard space, a digital camera, modelling supplies, a bookshelf or storage cabinet, a large table, a computer, chairs, wall space to attach paper, a projector, reference books, food, and toys. In general, Ambler stresses the use of simple and flexible tools, such as whiteboard and paper, over more complex tools, such as CASE tools. This

works well for simple situations that require more writing that reading; if there is need for extending previous models or using previous models for reference, it becomes problematic to use just analog material. For this Ambler recommends having a project, a computer with a CASE tool, and a "CASE jockey" who operates the CASE tool to capture changes drawn on the whiteboard and to show previous models. Electronic whiteboards, as introduced above, may improve on this sitatuation and make the choice between of tools much more flexible.

2.2.4 Simple Artefacts

You want to make sure developers focus on producing working software instead of, e.g., just working with software. Using Complex artefacts used in software production may detract the developers' attention from this.

Therefore: Provide and use simple artefacts, physically as well as virtually. The simplest tool that solves a problem should be used.

Index cards are ubiquitous tools also in agile development processes. They are cheap, readily available and sufficiently simple as not to detract developers from their primary focus.

Whiteboards and paper, e.g., for *Writing on the Wall* (Section 2.2.3), are simple tools that are ideal for collaboration and communication. For some situations these may be too simple, cf. *Writing on the Wall* (Section 2.2.3).

Use virtual tools that fit your work processes as closely as possible, preferably maximizing support for communication, feedback, and customer collaboration with a focus on working software. For modelling, Ideogramic UML (Damm et al., 2000), which combines electronic whiteboards and computer-aided software engineering, may be ideal. For programming, an Integrated Development Environment (IDE) that is flexible and allows you to combine just the needed components, may be ideal.

It may be argued that this pattern is actually the XP value of Simplicity used on faciliation.

2.3 Space

2.3.1 Flexible Furniture

You want *Space for Pairs* (Section 2.3.4) and *Space for Groups* (Section 2.3.3). Ordinary office equipment and arrangements makes flexible adaptation of spaces for these purposes difficult.

Therefore: Provide flexible furniture and allow developers to rearrange it through *Team Chooses* (Section 2.1.1).

Chairs should be movable so that people can pair and participate in group meetings. You might also want space for "standing" meetings (as advocated by DSDM).

Tables with adjustable legs are good for flexible pairing when programming.

Artefacts used for *Writing on the Wall* (Section 2.2.3) should also preferably be movable.

2.3.2 Public and Private Spaces

You want to facilitate group and pair communication while still catering for the needs for private communication and thoughts. You may have *One War, One Room* (Section 2.2.1).

Therefore: Provide a mix of public and private spaces.

These spaces should intermix with *Space for Groups* (Section 2.3.3) and *Space for Pairs* (Section 2.3.4).

An effective arrangement in XP has been to provide small, semi-private bull-pens along one wall of the room and have the middle of the room be common.

Often, if this is not provided, developers tend to create their own, virtual privacy by, e.g., listening to music using headphones. This practice hampers communication within the team.

2.3.3 Space for Groups

Communication with customers and brainstorming among developers is essential. You want to provide awareness of such (ad-hoc) meetings to all project members.

Therefore: Make space for groups. Make the space visible to all.

Meeting with an on-site customer, for, e.g., the planning game of XP, may be one of the reasons for having Space for Groups. This space can be combined with *Customer Close By* (Section 2.2.2). Awareness of results of group work may be made through *Writing on the Wall* (Section 2.2.3).

2.3.4 Space for Pairs

Developers need to communicate, collaborate, and coordinate effectively also in the small. Private offices or inflexible personal office space makes hampers this.

Therefore: Make room for pairs so that pair discussions and pair programming are facilitated.

This may involve having Flexible Furniture and is facilitated by having *One War, One Room* (Section 2.2.1).

2.4 Detail

2.4.1 Perks

Morale is high because of *Team Chooses* (Section 2.1.1) and the agile work process. You want to keep morale high and you want to support lateral thinking in the project group.

Therefore: Give developers perks in the form of food and toys.

Create a place in which it is possible and welcome to relax and where it is possible to eat food, drink coffee, play with toys, read magazines etc. The team should make sure they do not spend too much time on this.

3 Problem/Solution Summaries

Problem	*Solution*	*Pattern Name*
You need to provide the best working environment for your development team. Each project is unique	The team decides how their work environment should be and sets rules for how development should be facilitated	*Team Chooses*
Developers and stakeholders on your project need to communicate as effectively and efficiently as possible	Place developers in the same room	*One War, One Room*
Stakeholder requirements are initially vague and are constantly prone to change	Place the customer and developers close by and make room for that	*Customer Close By*
Developers and customers should have immediate access to writing material to be used for discussions and for thinking	Place whiteboards and paper for writing in development rooms	*Writing on the Wall*
You want to make sure developers focus on producing working software instead of, e.g., just working with software	The simplest tool that solves a problem should be used	*Simple Artefacts*
You want *Space for Pairs*, *Space for Groups*, and want to allow for adaptation to the current project situation	Provide flexible furniture and allow developers to rearrange it through Team Chooses	*Flexible Furniture*
You want to facilitate group and pair communication while still catering for the needs for private communication and thoughts	Provide a mix of public and private spaces	*Public and Private Spaces*
Communication with customers and brainstorming among developers is essential	Make space for groups. Make the space visible to all	*Space for Groups*
Developers need to communicate, collaborate, and coordinate effectively also in the small	Make room for pairs so that pair discussions and pair programming are facilitated	*Space for Pairs*
You want to keep moral high and you want to support lateral thinking in the project group	Give developers perks in the form of food and toys	*Perks*

References

Ambler, S. (2002). *Agile Modeling: Effective Practices for Extreme Programming and the Unified Process*. Wiley.

Andersen, C., Hansen, K., Sandvad, E., Thomsen, M., and Tyrsted, M. (2000). Tool support for iterative system development activities: Issues and experiences. In *Proceedings of NWPER'2000*, pages 1–21.

Beck, K. (1999). *Extreme Programming Explained: Embrace Change*. Addison-Wesley.

Beck, K., Beedle, M., van Bennekum, A., Cockburn, A., Cunningham, W., Fowler, M., Grenning, J., Highsmith, J., Hunt, A., Jeffries, R., Kern, R., Marick, B., Martin, R., Mellor, S., Schwaber, K., Sutherland, J., and Thomas, D. (2001). Manifesto for agile software development. http://www.agilemanifesto.org.

Beyer, H. and Holtzblatt, K. (1997). *Contextual Design: A Customer-Centered Approach to Systems Designs*. Academic Press/Morgan Kaufmann.

Booch, G. (1991). *Object-Oriented Design with Applications*. Benjamin/Cummings.

Christensen, M., Crabtree, A., Damm, C., Hansen, K., Madsen, O., Marqvardsen, P., Mogensen, P., Sandvad, E., Sloth, L., and Thomsen, M. (1998). The M.A.D. experience: Multiperspective Application Development in evolutionary prototyping. In Jul, E., editor, *ECOOP'98 – Object-Oriented Programming. Proceedings of the 12th European Conference*, pages 13–40. Springer Verlag.

Coad, P., Lefebrve, E., and de Luca, J. (1999). *Java Modeling in Color with UML*. Wiley.

Cockburn, A. (2001). *Agile Software Development: Software Through People*. Addison-Wesley.

Cunningham, W. (2002). About the Portland Form. http://c2.com/ppr/about/portland.html.

Damm, C., Hansen, K., and Thomsen, M. (2000). Tool support for object-oriented cooperative design: Gesture-based modeling on an electronic whiteboard. In *Proceedings of CHI 2000, ACM Conference on Human Factors in Computing Systems*, pages 518–525.

Harrison, S. and Dourish, P. (1996). Re-place-ing space: the roles of place and space in collaborative systems. In *Proceedings of CSCW'1996, Proceedings of the Conference on Computer-Supported Cooperative Work*, pages 67–76.

Highsmith, J. (2000). *Adaptive Software Development: A Collaborative Approach to Managing Complex Systems*. Dorset House.

Jacobson, I., Christerson, M., Jonsson, P., and Övergaard, G. (1992). *Object-Oriented Software Engineering: A Use Case Driven Approach*. ACM Press.

Mynatt, E. (1999). The writing on the wall. In *Proceedings of INTERACT'99*, pages 196–204.

Nygaard, K. and Bergo, O. (1974). *Planning, Control and Data Handling: Textbook for the Trade Unions. Part 1 Initiation. (In Norwegian)*. Tiden Norsk Forlag.

Rumbaugh, J., Blaha, M., Premerlani, W., Eddy, F., and Loresen, W. (1991).

Object-Oriented Modeling and Design. Prentice Hall.

Schwaber, K., Beedle, M., and Martin, R. C. (2002). *Agile Software Development with SCRUM*. Prentice Hall.

Stapleton, J. (1997). *DSDM Dynamic Systems Development Method : The Method in Practice*. Addison-Wesley.

Wood, J. and Silver, D. (1995). *Joint Application Development*. Wiley.

Pattern language for conducting a successful niche conference

Cecilia Haskins
Lecturer at the Norwegian School of Information Technology
cecilia.haskins@nith.no

Pattern: Personal Connection

Abstract: This is the first pattern for a pattern language that will eventually build on precedent patterns and add the essential elements that are required by the organizing committees for smaller conferences. This pattern addresses the challenge of attracting quality speakers on a shoestring budget. The author is very greatful to Neil Harrison and Linda Rising for their patience and advises during the shepherding process, and to the members of Workshop 2 for their helpful commentary during VikingPLOP 2002. It is the hope of the author that the foundation for the entire language will emerge with the help of the shepherding and future workshops.

Summary of the Language
Total number of patterns = 12
Grouped into 4 sections entitled Vision, Atmosphere, Roles and Customs. The patterns in the Vision section identify the core values of the rOOts conferences, to provide a forum for the discussion of new thinking – a spirit of learning. The other pattern areas serve to reinforce the core.

Vision

Create a rewarding, mutually nurturing environment for the exchange and discussion of trends in the software community linked to "object" oriented themes.

1. *Marketplace of new ideas (Day 1 Panel)*
2. *News from the front (experience reports)*
3. *Environmentally-conscious*
4. *Something to take away and use tomorrow*
5. *Mixing up the formula*

Atmosphere

Establish a setting for the conference that is comfortable, intimate and conducive to formal and informal dialogues.

6. *Comfortable beds*
7. *Time to talk*
8. *Night on the town*

Roles

Leadership and participation are important to a successful symposium.

9. *Enthusiastic committee*
10. *Distinguished speakers*
11. *Engaged participants*
12. *Credible web site*

Customs

Repetition of certain formulas re-enforces the comfort level for returning delegates and speakers and creates a format for "continuing" dialogues between and within each conference event.

13. *Something to remember us by (gift selection)*
14. *Personal connection*

Related patterns

Antipattern: Greedy Conference

Notes.
The conference should have a vision that defines the purpose and establishes the reason that the speakers have been invited. Speakers are informed of the nature of the funding and sponsorship and become a part of the foundation for the conference by their presence. In turn, they help create a forum for the exchange of new ideas and not just another conference.

The essence of this solution is captured in a set of steps.
The organizing committee should create a wish list of persons they wish to hear during the conference. Individual members of the committee volunteer to take the role of personal host to each speaker.

Once the theme of the conference is determined, the speakers are asked to suggest what presentations or tutorials they would most like to deliver. (see "Mixing up the formula")

After the speaker arrives in town, every effort is made for the host (or another member of the organizing committee) to greet the speaker. On the evening before the conference starts a small welcoming dinner with the speakers and 2-3 of the hosts is scheduled so that the speakers do not have to find a meal, navigate a strange city, or eat alone (unless they so choose). This serves to break the ice for persons who have not yet met, but will spend the better part of 2-3 days together. Those already acquainted often pick up unfinished conversations from their last meeting, and the mood is generally light and jovial.

Speakers are encouraged to be approachable to the small audience during lunch and conference social events, and a book signing is recommended. A unique gift with the conference logo is presented to each speaker as thanks and a memento of the occasion. (see "Here's a little something to remember us by")

Element	Description of the Pattern
Name	**Personal Connection**
Problem	Most conferences strive to offer a program to their delegates with the most qualified speakers available. The organizing committee of a small conference with a limited budget may find it difficult to offer large honoraria to their speakers, and therefore fail to attract the caliber of speaker that the delegates deserve.
Context	The organizing committee for a small, intimate niche conference has additional challenges from those of better-funded conference committees.
Forces	▪ Speakers of international caliber have hectic schedules and can command high fees ▪ Speakers generally find conferences highly impersonal ▪ The decision to attend a conference is highly dependent on the content and speakers ▪ For most delegates the ability to attend a conference is related to an affordable price
Solution	Establish a Personal Connection with the speakers for your event. This is done by creating a personal invitation, assigning a personal host and allowing a personal relationship to grow. Begin with a personal invitation. Sometimes a committee member has met the person identified as a candidate speaker. In those instances, that committee member should write the invitation and build on that acquaintance. Other speakers are identified because they are authors of a book that has been referred by another rOOts speaker or read by a committee member. In those cases, this is the basis for the personal invitation. Every invitation refers to the web site of the conference to establish credibility and when possible, indicated which other speakers are already on the program with whom this speaker may wish to spend some time. Simplify and personalize the interactions between the committee and the speakers by assigning a Personal Host. The Personal Host is responsible for all communications before and after the conference. During the conference, the Personal Host will greet the speaker upon arrival, make sure that room reservations and meals are suitable and be available to assist with any questions that arise while the speaker is in town. Speakers are informed of the nature of the funding and sponsorship and become a part of the foundation for the conference by their presence. The extra attention compensates the speaker for a reduction in their normal honorarium. All travel expenses and any expenses that would be incurred during their visit are fully covered by the committee. When requested, the presence of a spouse is encouraged, and the spouse is included in all social activities. At the end of the conference, a special, speakers-only event is scheduled, designed to show off some aspect of the host town, and allow those speakers that can afford the extra day to relax, confer and continue to enjoy each other's company before going home. In many instances, friendships begin between the invited speakers and the committee members. These relationships become the basis for planning future conferences.
Examples	From conferences rOOts 2000 and 2001 and 2002 In 1999 the vision was generated to create a small conference in Norway that expanded on the realm of thinking spawned by the creation of Simula and the promotion of object-orientation for software design and development. Four prominent authors were considered critical to the success of the first event. They were invited and asked to defer or reduce their usual honoraria. Members of the committee accepted the role as "host" to one speaker and this approach has lead to the background for this pattern. Today, the rOOts committee members have good friends who in turn can suggest other persons with interesting and new ideas. ROOts speakers have shaped much of the vision that motivates this conference. In 2000, Cope helped the committee articulate a niche position for the conference resulting in the current formula with tracks for both technical and leadership delegates. Martin Fowler gave us many helpful tips on what worked best from his perspective as a speaker, and advised the committee not to grow the conference into the thousands of delegates, as was the then-current plan.

Element	Description of the Pattern
	In 2001 a world-class author approached the committee and offered to be a speaker. All of our keynote speakers have deferred honorariums. All of our speakers who receive honorariums reduce their rates and spend lots of time with the delegates.
Resulting Context	The end result is that for every 2 speakers who say "no thanks" there is one that says yes. And often those who decline in one year because of a conflict in schedule make it clear they would like to be invited again. A small conference achieves its primary goal of creating a program filled with highly qualified presenters. The presenters actually enjoy the conference because they are relaxed and allowed time to exchange ideas with their peers. As a consequence, they are very generous in the time they spend with the delegates. Delegates leave satisfied and looking forward to returning next year. Conference sponsors enjoy the prestige of being associated with an event with world-class presenters and continue to support the conference in future years, which in turn helps the committee to pay the speakers some money and at the same time keep the conference fees low. Speakers are willing to consider repeating the experience, and the relationship is born. The relationships established from prior years work to the advantage of both the speakers and organizing committee. There is always the possibility that a speaker will not accept the committee offer, but that is the only negative consequence encountered to-date. There have been 2 experiences where rOOts was scheduled against a "bigger" event and lost potential speakers to that event.
Rationale	▪ Speakers have an altruistic side that prompts them to help a worthwhile event. They are also human and value having a positive experience and a bit of a good time. They like to take holidays and meet new people. ▪ Components of a successful conference include sponsorship and good attendance both of which are enhanced by the presence of world-class presenters ▪ Delegates are inevitably drawn to conferences at which published authors or renowned speakers are giving presentations and tutorials ▪ Delegates' employers are interested in spending their training money in the most efficient way possible (minimized travel and conference fees, maximized conference content)
Related Patterns	Antipattern: Greedy Conference
Known Uses	Conferences rOOts 2000 and 2001 and 2002

Patterns for the Practicing Software Architect

Klaus Marquardt, Käthe-Kollwitz-Weg 14, 23558 Lübeck, Germany

Email: marquardt@acm.org or pattern@kmarquardt.de

Large software systems and projects have a lot of internal dependencies that comprise not only software design, but also the process and the organization. Some of these dependencies can be introduced intentionally, others will grow undetected and prevent the system or project to evolve in an optimal manner. A software architect may detect undesired dependencies and try to cure the system from them. The presented patterns provide techniques that enable a focused work, and refine and redefine the role of an architect within the project.

Introduction

There is no commonly accepted definition of the profession of a software architect yet. Most approaches focus on the initial up-front activities needed for large projects. Key success factors are the creation and sharing of a common vision of the system, and a focus on the system parts responsibilities and their mutual dependencies. However, some important qualities of the system cannot be planned for in advance, but need constant care and attention. Here another working attitude can serve the architect, an attitude towards problem solving. The system becomes more consistent and exhibits more internal qualities when the initial architect also is the most frequently consulted problem solver.

A different metaphor offers new perspectives both for agile development and for projects facing problems – which every project will at some time. The software architect then takes a role similar to a medical doctor. She examines the system and makes a diagnosis, identifies the underlying causes and starts with treatment. This metaphor complements the more common engineering metaphors and shows its strength in different situations.

This paper collects patterns about activities of an architect, and the role that she might take within an organization. It is part of a larger effort to make problems and solutions accessible in a medical form, as symptoms, diagnoses, and therapies. The diagnoses are typical entry points and combine different therapies and show their relations and interdependencies. Some diagnoses from design structures and organizations have been published before [*Marquardt01, Marquardt02a*] and are referenced throughout the paper; see the appendix for thumbnails.

Therapy Pattern Format

The pattern form used for the therapies deviates in some respect from canonical forms. First, a therapy might fit different diagnoses, thus it starts with a “meeting point” more general than a usual context.

Especially for therapies it is essential to know their mechanisms, and to learn about side effects, cross effects, counter indications etc. – just as you would expect for any medication. The respective sections extend the pattern form and are introduced by corresponding symbols:


the effective mechanisms and the related diagnoses or pathogens;


scope, costs, roles involved;


side effects, counter indications, overdose effects;


cross effects and related therapies.

Each therapy pattern reflects on its implementation and relevance and closes with an example.

Contents

Divide et Impera[1]

> Consider a project that is too large and complex to be handled by a single developer. You are asked to sketch a system architecture.

In the initiation of a large software project, you need to provide a starting point for system development and for management. You are aware of possibly unstated expectations like

- the system structure must allow for parallel development and easy integration;
- you need to identify a reasonable order of tasks and the effect of changes;
- you need to tell which parts of the system are completed, and how they can be packaged and tested.

In short, you are about to define a system architecture.

A monolithic software system is the structure with the least effort in the beginning,	… but a carefully partitioned system scales better with increasingly complex requirements and a growing scope.[2]
The system is developed with the least effort when you define the architecture right at the projects' start,	… but before halfway into the project you hardly learned enough to finalize the interfaces and architecture.
Late changes to the system structure require rework effort and time,	… but you do not know the optimal partitioning in the beginning.
You do not like to spend effort for tasks that are not visible to the end user,	… but non-functional expectations need to be addressed even if they come at a high engineering cost.
You need to answer questions about the order and distribution of assigned tasks,	… but the system structure should be able to stand independent of the tasks, for changes, reassignment or maintenance.

Therefore, divide the software system in distinct parts. Consider and evaluate cuts both along the lines of application domains, and of technical subsystems. Identify the intended subsystems, define their dependency graph, their responsibilities, and rules for their division.

With this solution, you scale down the problem: you get to a number of smaller scopes that are each used and maintained by a much smaller group of developers, and become stable and finished in a more reasonable time. You do not need to complete the entire system to close down each single part of it [*Caesar*].

While the partitioning of each system is unique and hard to predict, here are some starting points that have proven effective in other projects:

- IDENTIFY APPLICATION MODEL. The essence of the system is its meaning to the user. Before you can do any division, your analysis must be sure of the semantic core.

[1] Latin: Divide and conquer
[2] see [*Dyson+02*] for an example evolving from scratch to a complex internet server architecture

- SEPARATE VIEW FROM MODEL. Make sure that whatever you present does not influence your data model, and that the specifics of the presentation are not reflected in your application model. This has proven particularly useful when multiple distinct views had to be supported for different use cases.
- SEPARATE MODEL FROM TECHNOLOGY. While you might be certain to use a particular infrastructure throughout the project's lifetime, your system division, testing and integration becomes much easier when large parts of the software do not depend on it.
- SEPARATE APPLICATION FROM PROTOCOL. You gain flexibility and testability when your application does not depend on some specific protocol, but treats each protocol connection as an additional view on the model. Applicable in most technical systems and where a high degree of interoperation with other systems is required.
- APPLICATION DECOMPOSITION. Within your application model, you will be able to define distinct areas that have little overlap or commonalities. Separate these areas explicitly, and address shared areas distinctly. The defined application areas are a valuable starting point for a development order and sub-team division.
- FOLLOW ORGANIZATION. Like the architecture, the organization divides and structures its components according to its needs. When your division of responsibilities contradicts the organization's, you create a friction that can be severe enough to effectively replace any integration with finger pointing. Divide the system so that only complete parts are given to an organizational unit, and that the dependencies are in sync.
- DEFINE CLEAR AND MINIMAL INTERFACES. Check your division whether the identified components have clear and potential minimal interfaces to each other. The emphasis lies on clear here – unclear interfaces can tear your project down, while extensive interfaces basically require a steeper learning curve for new programmers.
- INSTALL ISOLATION LAYERS. Each part of your software will have to solve problems that are unknown to other parts of the system. The solution is a private property that should not shine through the interfaces, otherwise your system becomes untestable and might need frequent rework when particular solutions are exchanged.
- ISOLATE RISKS. From the very beginning, you will suspect that some software parts are particular suspect with respect to performance, hardware or driver changes, or other risky properties. Divide the system so that these risks are treated in only one component each.
- IDENTIFY SHARED CODE. While you divide the system, you will often see that some of the cuts you like to establish cannot be made clearly because both sides need some shared knowledge or code. Define this shared area as a component on its own, and start developing its interfaces early in order to stabilize them quickly.

The list above indicates that your system will be divided along multiple dimensions. Each package of your system should sit at exactly one coordinate point of these dimensions, so that you do not introduce conflicting forces within individual architectural granules.

When dividing your system, do not forget that you need a plan to conquer. Make sure that the identified divisions can be set together again and integrate into a functioning, consistent system. Play an integration planning game where you combine possibly available components, and check whether the partial system would be executable and testable. You might consider replacing particular components with dummy or trivial implementations.

———————

This pattern contains an essence of the art and craft of software architecture. While from a philosophical viewpoint software architecture is about creating and sharing a vision, as in BIG PICTURE ARCHITECTURE, most clients of architecture demand more concrete guidance. A software architect must resolve three key tensions within a project: the tension between a top-down approach where the architect defines what is to be implemented, and a bottom-up approach where the architecture is the result of the finished development; between desired up-front activities and pragmatic refactoring of not-so-perfect subsystems; finally, the tension between the immediate need for task assignment and the deferred need for a consistent maintainable system structure.

The art of applying DIVIDE ET IMPERA lies in three aspects with the component definition: they should be defined along application defined lines, exhibit appropriate mutual dependencies, and be of an adequate size. Define the components in a way that they need as little programmatic access to one another as possible. When you need that access, order the component dependencies in a manner that the most frequently used component becomes stable first.

An appropriate component size depends on the total system size. It is usually not useful to distinct more than about a dozen components per level of abstraction. This may give you a rough idea; however, the application coherence of the components is definitely more important.

The main mechanism is to establish a separation of concerns, and to get to smaller system pieces that you can manage more easily. By forcing you to structure the system, DIVIDE ET IMPERA is preventive against most structural diseases such as QUICKSAND BASES, CROWDED PACKAGES, FOUNDLINGS, and DEPENDENCY CYCLES. Applied on a small scale, it can become effective against CROWDED PACKAGES and DEPENDENCY CYCLES.

The effort of DIVIDE ET IMPERA depends on your and the teams' abilities to abstract, communicate, and learn. It is about proportional to system and the team size, and can be scaled down by simple timeout criteria. It can be lower when a preventive BIG PICTURE ARCHITECTURE is in place. Involved roles are developers and architects, with a limited involvement of management.

There are no counter indications. You might experience an overdose effect when you choose too many and too small components, that you become busy trying to master the complexity of component relations instead of gaining the profits of a clear system structure. The most severe side effect is that you might get the division wrong, providing more irritation than guidance and effectively preventing system integration and maintenance.

A number of other therapy patterns help to increase the effectiveness of DIVIDE ET IMPERA: BIG PICTURE ARCHITECTURE, NEGLECT THE LEVEL BELOW, COMPONENT MAINTENANCE RULES, and MAP COMPONENTS TO EXECUTION. Soothe possible overdose and side effects by applying IMPERA SED AUTEM DIVIDE frequently from the very beginning which complements the top-down approach by a bottom-up attitude.

An example is given with the IMPERA SED AUTEM DIVIDE therapy pattern

Impera sed autem Divide[3]

> Consider a project that is too large and complex to be handled by a single developer. While the initial system is quite aged, it needs further maintenance and functional additions.

In a large software project that grew without restructuring for some time, you are faced with one of the following problems – or more likely, with several of them within a short period:

- The changes you introduce to the system have unexpected effects for unrelated functionality, or no effect at all.
- You are unable to assign a task that does not interfere with other tasks.
- You find no way to complete an assignment without changing already finished system parts.
- You cannot test a single functionality without testing the entire system.

In short, you suffer from the fact that your system is a monolith, or Big Ball of Mud [*Foote+00*], that you are no longer able to manage. The problem at hand can be in any area like parallel development, closure, testability, or packaging.

You do not like to spend effort for tasks that are not visible to the end user,	… but further functions are increasingly hard to add and come at a high engineering cost.
Late changes to the system structure require rework effort and time,	… but you do not know the optimal partitioning in the beginning.
A monolithic software system is the structure with the least effort in the beginning,	… but in the meantime you learned about your system and sense an emerging structure.
You know that you need to spend time on revising your systems' structure,	… but you can not afford to put the development to a hold.

Therefore, identify potential components that you could finish independent of future system extensions, and separate them from the rest of the software system. Finish these components before you continue with functional expansions. If you cannot complete the components themselves, declare their interface as a component on its own and finish that one. Thus, the application becomes completed in terms of these defined independent, or at least well ordered building blocks.

Consider and evaluate cuts both along the lines of application domains, and of technical subsystems. The topics described in DIVIDE ET IMPERA can give you a starting point. Define the components' responsibilities and interfaces and check for dependencies with other component candidates. Start dividing the system where the current monolithic structure hinders you most, and introduce further structure in a piecemeal growth manner.

With this solution, you define some area of the system for which you can solve your problem. You get a smaller scope that can be handled by a smaller group of developers, and become stable and finished in a more reasonable time. Neither do you need to complete the entire

[3] Latin: Conquer, but also divide

system to close down these specific parts, nor do you need to finish all separation efforts before you continue to develop user relevant functionality.

The art of applying IMPERA SED AUTEM DIVIDE lies with the component definition, just as in DIVIDE ET IMPERA. However, you are in refactoring [*Fowler99*] mode now and might face some unpleasant facts of the de facto system. Take special care to limit mutual access between the components. When all components access each other, you will not win the division (and might lose the imperative), and most liabilities of a thicket like structure become re-introduced. As always with refactoring, also take care not to break existing functionality (your imperative), preferably by maintaining an automated test suite for user visible features.

One major advantage of IMPERA SED AUTEM DIVIDE is that it can be introduced into an existing code base, often with surprisingly little changes and at a low cost. On the downside, the division is by convention only and can in general not be enforced in code. Applying IMPERA SED AUTEM DIVIDE requires buy-in by most of the developers. Some languages and environments offer constructs that help them to stick to their conventions.

The main mechanism is to separate different concerns where they stand in your way, and to factor out smaller system pieces that you can manage more easily. IMPERA SED AUTEM DIVIDE is thus effective against structural diseases on a limited scale, such as QUICKSAND BASES, CROWDED PACKAGES and FOUNDLINGS with a chance for remission of most symptoms.

The effort of IMPERA SED AUTEM DIVIDE is proportional to the amount of code, and can be scaled down to the most suffering components. It can be lower when a preventive BIG PICTURE ARCHITECTURE or a operational structure according to DIVIDE ET IMPERA has been in place. Involved roles are limited to developers and architects. However, when you introduce IMPERA SED AUTEM DIVIDE late in your project you need to re-consider your system partitioning. This can become a fundamental architectural change – to the positive, but at the price of rethinking and refactoring.

There are no side effects or counter indications. An overdose effect can occur when you apply IMPERA SED AUTEM DIVIDE against a grown and working system at the end of its life cycle.

IMPERA SED AUTEM DIVIDE is a natural counterpart to DIVIDE ET IMPERA. You need both of them in different stages of the project. A successfully established BIG PICTURE ARCHITECTURE helps to minimize the dose you need of IMPERA SED AUTEM DIVIDE.

---

*The software of an integrated workplace needed to integrate a number of different applications. Each of these applications brought along its own user interface and data model (PLUG-IN PACKAGE [*Marquardt99*]). The system became divided along these packages, plus the common software parts they depended on, and the software parts that employed the plug-ins.*

The packages as well as the common framework turned out to be of a too large granularity. They were subdivided during development: most packages according to technical aspects as user interface, communication, persistence; one plug-in package became subdivided along sub-applications within the initial application to support the existing know-how of the development team.

Big Picture Architecture

> Consider a project that is too large and complex to be handled by a single developer. You are asked to sketch a system architecture.

In a large software project that is started from scratch, you need to provide an architectural outline that all decisions can be measured against.

A detailed architecture provides the closest decision guidance,	… but a late architecture delays the project, or becomes avoided by decision makers.
A concise architecture is easy to transport and understand and provides consistency by itself,	… but a short Metaphor[4] may have multiple meanings and mislead the projects' participants, and … a Metaphor answers too little questions beyond its particular domain.
Communicating and arguing about the architecture is tedious and may become frustrating,	… but an architecture not known to the team will be ignored and forgotten.

Therefore, define a compact architecture outline and make it become part of the project jargon. The software architecture outline must cover the top level of the technical structure, the key domain abstractions, interfaces and interactions, and the order and stability of development.[5]

You need to illustrate the most important issues in a simplified way. The simplifications should match with the developers' experience and scale up to a large extent. Examples from the technical domain are a Document-View or a layered architecture. The domain model typically comprises less than 20 classes and their relations. The most relevant interfaces can also be categorized and sketched. The order and stability of development can be expressed in packages and their dependencies.

Metaphors can help you to outline parts of the BIG PICTURE ARCHITECTURE with a few words that evoke guiding associations. Graphics and diagrams are typically most appropriate for components, interfaces and interactions. Resist the temptation to use buzzwords and technology phrases as they do not help you to distinct your unique system.

Influencing the project jargon is most easy when you can refer to common vocabulary. This can be pattern names in technical domains. Key domain classes usually creep into the commonly used language from the customer. Packages and dependencies may require more thought and more discussion within the team, before a common terminology becomes established.

A BIG PICTURE ARCHITECTURE typically is specific to your system and its domain. However, you make your life (and the life of your peers) easier when you refer to a common and rather precise vocabulary. Some architectural patterns as MODEL-VIEW-CONTROL [*Buschmann+96*] can transport significant ideas with a few words.

---

[4] as in over-simplified XP

[5] This is an extension to the SHAMROCK pattern, thumbnailed in [*O'Callaghan99*]

Where you need to introduce specific high-level concepts, try to maintain a level of detail and abstraction that can provide guidance. Well-understood and well-understandable systems typically contain SMALL FAMILY SYSTEMS [*Kerth95*], that is a limited number of participants at about the same level, who know each other well and play defined yet distinct roles. You may use this as an completion criterion for this level of detail.

While patterns are most useful for technical issues and interfaces, the specific small families often occur in top level domain classes. Relate these aspects to each other, and explain why the combination of different views on the system is important, and which view is most appropriate when.

The dependencies within these views and between them must be explicitly modeled, so that all current and foreseeable questions can be evaluated against all of them. Views from different aspects should be orthogonal and complement each other well. Be prepared to even address issues of proposed team structure, project scheduling and task assignment.

The main mechanism is to establish means and ways of communication. BIG PICTURE ARCHITECTURE is effective against all diseases related to communication problems, with a chance for remission of most symptoms.

A BIG PICTURE ARCHITECTURE requires time to develop and communicate. The architect needs to involve all roles of the project. The effort is comparable to other team building processes, and could be spend in parallel to similar processes that cause friction but form a team and consensus.

There are no side effects or counter indications. An overdose effect can be “micro-architecture”, similar to micro-management. The most severe side effect is that you might choose an inappropriate big picture, providing more irritation than guidance and effectively preventing a consistent system development.

BIG PICTURE ARCHITECTURE is one of the essential architectural techniques you need from the very start of the project. In conjunction with DIVIDE ET IMPERA and NEGLECT THE LEVEL BELOW it transports the architectural vision of the system.

In lack of other common vocabulary, an architect introduced an extensible architecture with the notion of “colored boxes”. Each box represented an extension component, the color indicated its particular purpose with respect to techniques and application. Within each component, a Model-View-Control pattern (MVC) came into place, and within the MVC participants one more level of substructure was defined. After some time, the vocabulary and dependencies became obvious to the team, and each developer was able to place a given class at the correct logical location – or to tell what was wrong about it.

Explicit Dependency Management

> Consider a project in which each new functionality is designed in a small group, and for each of its aspects the most appropriate place in the system is determined. An architect observed whether the agreed relations are kept during development.

In a large software project, the internal structure erodes while the project focuses on functionality and delivery dates.

Functions and dates are externally visible and easy to track,	… but the internal structure is visible to developers only – or incomprehensible to everybody.
You do not like to spend effort for tasks that are not visible to the end user,	… but non-functional expectations need to be addressed even if they come at a high engineering cost.
Functions and dates determine how valuable the software is initially,	… but deficits in the non-functional internal structure limit the development speed, testability, and maintainability.
Non-functional properties are plenty, and they are incredibly hard to measure objectively,	… but a focus on very few important issues like dependencies and responsibilities[6] gives you a usable grip to the system's internal quality.

Therefore, track the internal system dependencies as a manager would track dates and delivered functionality. Use your influence skills and introduce strong forces so that the management of the structure becomes a similar visibility. Establish restrictive rules for the key dependencies within your system. Enforce that all developers stick to them as they do to coding conventions and project schedules.

However, hold a minute before weighting your influence against others. First make sure that you are detailed about the desired structure, that it is well communicated, and that the components interfaces are clear, consistent, and unambiguous with respect to dependencies and their direction. Use DIVIDE ET IMPERA to outline and define the structure, but also use NEGLECT THE LEVEL BELOW to define your stop criteria, reduce your workload and avoid appearing draconian.

Check the dependencies on a regular basis, and establish this check as part of the commonly accepted development process. Make sure that you have tool support for the checks! Remember that compiler and linker already do a similar job, they typically can be instrumented to fit parts of your needs.

You need to create a dependency model and maintain it through the project. Update both the model and the checking tools with each change. As a rule of thumb, do not check for mutual dependencies in more than three dimensions or about three dozen components. If further divisions are needed and in place, they are probably less relevant for frequent checking.

[6] for a motivation of this, see [*Marquardt01*], "A view on software architecture"

EXPLICIT DEPENDENCY MANAGEMENT can most successfully be done by strong personalities. It is hard to introduce additional forces onto the developments, and it is even harder when they are orthogonal to the manager's (short term) forces.

The main mechanism is to bring the internal structure and its relevance to the project's success into everybody's consciousness. EXPLICIT DEPENDENCY MANAGEMENT is effective against all diagnoses caused by too little attention on structural issues, such as DEPENDENCY CYCLES.

The related costs depend heavily on the project and its participant, and grow facultatively with the project size. This duty should be a significant part of the job definition of the software architect. Other roles involved are software developers, but they should hardly notice their involvement.

The only counter indication is a project of small size, i.e. when you are likely to overdose this agent. An overdose effect can be "micro-architecture", similar to micro-management. When an architect cares for too many details, the developers perceive both mistrust and that their work is not valued.

The key therapy for combination with EXPLICIT DEPENDENCY MANAGEMENT is NEGLECT THE LEVEL BELOW. The combination can help you to cope with the combinatorial explosion of component relations with increasing project size, and avoid the micro-architecture overdose effect.

A project used a visual modeling tool and established a process to keep this model consistent with the actual code. Custom-made scripts checked the model for various design metrics, among them for prohibited usage between dedicated system areas, and for cyclic dependencies among packages.

Neglect the Level Below

Also known as: Defined Neglection Level

Consider a project with a detailed architecture designed by an architecture team. The developers see a huge pile of paper, prepare for trouble, implement the system according to personal taste – and ignore the paperwork and the ideas buries within.

For an ongoing project, you have defined a well-thought concise architecture. You need to establish the architecture in the implementation without compromises in key issues.

An architecture is expected to cover everything relevant for system design and implementation,	… but a voluminous architecture is neither read nor implemented without overly tight restrictions.
The architecture has prepared for every reasonable issue arising,	… but during development of sufficiently large projects new, unforeseen issues will arise.
The system is most consistent when everything is controlled by the architecture,	… but oftentimes consistency in details is less important than a clear overall structure.
An intentional limitation of your efforts helps you to reach your major goals,	… but you need to be aware where a slip in details can be crucial for the project's success.

Therefore, decide on the level of detail that is controlled by the architect, and neglect all levels below that. While you are serious about the architectural rules at high levels, relax your control on all levels below the one that is of architectural interest. For each interest area, you might define a separate neglection level.

This is an important rule of thumb when you care for DIVIDE ET IMPERA and a BIG PICTURE ARCHITECTURE. For example, take care for packages and the order of components; but neglect cycles among classes within a single package or components.

While there is no commonly valid rule across all projects and domains, here are some starting points how to determine your initial level of neglection. Be aware that this neglect is subject to change during the project, according to identified risks, workload, and customer needs.

- ONE LEVEL BELOW BASICS. As a first approximation, look at the components you defined in DIVIDE ET IMPERA and take care of no more than one level below that one. Include especially risky components in full detail unless you are swamped with work.
- LISTEN. Except when compiled from newcomers, the team usually knows quite well where critical points are and what to take care of.
- DON'T INTERFERE EXPERIENCE. Where the most experienced developers work, you may neglect the most activities – provided you have clearly communicated and reasoned about your priorities and architectural goals, and received a high degree of buy in. Stay in close contact to understand when you need to get rid of your neglect.
- BE ARROGANT. After accepting other people's competence and wisdom, you need to be sufficiently arrogant to go for your viewpoint in case of any doubt.

Like some other patterns about the software architect, NEGLECT THE LEVEL BELOW requires a delicate balance. To the inexperienced, it can turn into bad advice for at least two reasons. First, in medium sized project with experienced developers, an architect's credibility may partly depend on his ability to discuss even on details. Second, decisions on detailed levels may strike back to higher levels, if they are not in line with the big picture. You need to know which decisions are of key importance, and follow them to the implementation. If you really neglect these, somebody else will take over and become the de-facto architect of the project. And this is probably OK, because the knowledge of an appropriate neglection level comes with experience, and this is what being an architect should be related to.

The main mechanism is a defined focus for the architect's work and a relief from tasks that do not increase success probability. Additionally, NEGLECT THE LEVEL BELOW applies a divide-et-impera strategy with respect to responsibilities in the project, allowing other experienced developers to grow with their tasks. It is preventive and palliative against all diagnoses caused by overly tight architectural control or by ignored architectural rules.

NEGLECT THE LEVEL BELOW is especially useful in large, complex projects. There are no costs related, and only developers and architects need to be involved.

Lack of experience is a clear counter indication. It is important that you know what is important, what is not, and which issues may become important at which time during the project. The overdose effect of NEGLECT THE LEVEL BELOW would be to have no meaningful architecture at all, whilst the underdose effect is the same – though from ignorance from the other side.

NEGLECT THE LEVEL BELOW is best accompanied by BIG PICTURE ARCHITECTURE, in which you already define what are the key architectural issues to you. JOINT DESIGN can help you notice when you are about to miss important issues, and PART TIME ARCHITECT can help you to really let go and focus on key issues only.

A large Plug-In based project introduced design conventions that banned bidirectional or cyclic dependencies among packages. These rules were checked with automated tool support. Within a package, dependency cycles were explicitly accepted. This allowed for a refactoring when cycles were appropriate to the solution, and kept the high-level dependency structure manageable.

Component Maintenance Rules

> Consider a project that is too large and complex to be handled by a single developer. You are asked how the system architecture supports tasks assignment and parallel development.

In a large software project that is started from scratch, the architecture needs to support a distribution of tasks assigned by project management.

The order and distribution of assigned tasks is a project management issue,	… but the technical system partitioning is an architectural issue.
The task assignment must be related to the technical solution,	… but the architecture itself does not depend on a particular (arbitrary) assignment, and the overall solution does not change with tactical project decisions.
You need to answer questions about the order and distribution of assigned tasks,	… but the system maintenance depends on the logical components and structure alone.

Therefore, define rules about your system structure that developers can follow while they fulfill assigned tasks. These rules provide guidance to maintain and update the system consistently. Adding functionality must always be paralleled by questioning and updating the system structure [*Parnas94*].

Define which subsystems and packages the system needs to distinguish, and why – as you do in a DIVIDE AND IMPERA. Define each major package's responsibilities, so that it is easy to determine which classes or functions belong into it, and which must be kept out. Also provide criteria on when a new package should be created, and how and where this finds its appropriate place within the top level architecture.

Again, the components defined in DIVIDE AND IMPERA form a good starting point. A reasonable, though obvious rule would be that each package belongs to exactly one side of each separation. Each functional addition or further separation enforces the creation of additional packages. Further rules come from intentional abstraction layers, and an analysis of expected changes, configurations, and system extensions.

Another important hint for a component maintenance problem is unclear ownership. For each package, exactly one or one team of developers should feel responsible. Mutual access problems indicate a missing separation and typically require splitting the component.

The larger the system, the more you need to care for a consistent component structure. In small projects, you might be well of with a much simpler system division oriented along the lines of use cases or stories (as used in Extreme Programming [*Beck99*]). Here, each use case forms both a task and a component – driven to an extreme, the system could consist primarily of Use-Case Controllers [*Aguiar+01*]. This is surprisingly similar to the functional decomposition with its known liabilities [*Martin96*], and it hardly scales for larger systems.

Besides finding appropriate rules, the implementation of COMPONENT MAINTENANCE RULES has another hard part: communicating the rules in a way that developers adhere to them.

The probably most effective way for communication is JOINT DESIGN [*Marquardt02b*], where you participate in the developers' work and can give both rules and the reasoning behind

them. Seek for discussions where you can combine answers to arising questions with explaining the goals behind your advice. You may even write a short document of a few pages that summarizes the rules and can be a usable reminder. Unless the project is really huge, I am not in favor of presentations – except to upper management, and people outside of the project.

COMPONENT MAINTENANCE RULES is effective because it defines the scope of the architecture and its scaling mechanism. It is curative for all diagnoses that are caused by structure erosion, such as CROWDED PACKAGE, FICTITIOUS MARRIAGE, and FOUNDLING.

COMPONENT MAINTENANCE RULES is the daily task of a software architect. Its effort depends on the quality of the guidelines and the willingness of the developers to follow them. Involved roles are limited to developers and architects.

There are no counter indications or overdose effects. A possible side effect occurs when there is no architect's position defined, in this case typically a senior developer suffers from trying to fulfill this task by working overtime.

DIVIDE ET IMPERA and BIG PICTURE ARCHITECTURE are necessary first steps for COMPONENT MAINTENANCE RULES. To know where to stop with rule definition, apply NEGLECT THE LEVEL BELOW. Another related diagnosis is FUNCTION FOLLOWS FORM, which can be avoided by an agent combination of COMPONENT MAINTENANCE RULES and MAP COMPONENTS TO EXECUTION.

Possible rules and heuristics could include:

Every function is placed as low in the package graph as possible. Potentially commonly usable classes must be accessible from potential clients with respect to the global division of your system. Use abstraction layers to increase the reuse potential, and put specific implementations at a higher level.

If frequent conflicts between different developers occur, check whether some packages need to be split because they combine incoherent responsibilities.

Application related classes must not depend on technical issues, such as persistence or serialization. Nothing depends on a presentation.

It is easier to do performance tuning in a clear logical structure, than to debug in a performance optimized system.

Generic Situation

> Consider a component within a project that has a limited and well understood functionality. All its states or error conditions can be related to a few generic situations, and its responsibilities described by a few generic interfaces.

In a software component under construction, you need to describe all states that a specific property can take. The software itself needs access to these properties. You can not come to closure before you know how to deal with all possible states and situations.

It is convenient to have each single condition mentioned uniquely,	… but you can adopt to new situations more quickly when you have the ability to categorize them in familiar terms.
It requires little thought to change a system whenever you learn more about it,	… but it is expensive to couple all your development efforts tightly, and update or recompile your existing code base with essentially unrelated changes.
Anticipation of yet undiscovered system properties is error prone,	… but you can use your application knowledge to come to separation and closure quickly.

Therefore, abstract from the concrete situations and describe each situation as an incarnation of a general one. Instead of describing each possible state individually, your system might allow you to abstract to a small number of states of a more general nature.

If you manage to define and live with a few, at maximum a few dozens of general situations, your basic requirements may be fulfilled while you come to closure with the remainder of your software component and its dependees.

Apply GENERIC SITUATION whenever you or a colleague is tempted to enumerate some property on system scope. When there is no definite criteria to end this enumeration, step back and look for commonalities among the already available entries. Combine similar descriptions to a common one.

GENERIC SITUATION is a standard technique to software engineering as to all scientific disciplines, and has become especially popular with object-oriented approaches. Though it is mostly expressed using base classes and inheritance, it is not limited to that. The general nature of GENERIC SITUATION is to retreat from the problem at hand, check for similarities with other known or expected problems, and describe the commonalities among them. Domain specific application level protocols for standardized data exchange, e.g. based on XML, are examples for successful GENERIC SITUATION.

The ability to find and communicate abstractions and GENERIC SITUATIONS comes with experience, and depends on individual personalities. When you are insecure, do not start with GENERIC SITUATION but approach your system in a piecemeal growth manner, learning about the system while you complete it. However, you miss chances for an early decomposition and independent completion of different subsystems.

It is important to control changes to your generalized software parts. Each change violates the system's closure, and reduces the trust its clients have into it. Especially avoid bloat – for a successful GENERIC SITUATION, less is always more.

Occasionally, you will find that your GENERIC SITUATION falls short of describing a particular situation. If necessary, you can accompany the general description by some optional context information, like class name or source file.

The main mechanism is to reduce the systems' complexity by using your knowledge of the application domain, rather than by technical engineering means. GENERIC SITUATION is effective against QUICKSAND BASES, with a chance for remission of all symptoms.

The effort of GENERIC SITUATION is in the up-front considerations, and can not be scaled. Involved roles are limited to developers and architects. The most significant side effect is that you have an increased effort for late corrections after wrong guesses.

Counter indications are a project that is not in an early stage, or a component whose domain is not well known to the development team. You might experience an overdose effect when you try to describe everything in a generic manner, and forget that software is about timely delivery of real functionality.

The C++ STL defines fourteen exception classes that it throws in error situations.

A framework project decided to report only a limited number of error codes to their clients, because the analyzed error situations offered only very few meaningful reactions of the client code. They defined a class `ReturnCode` *that was returned instead of an integer and basically consisted of a definition of 18 error codes. All situations were described in terms like* `Timeout`, `InvalidArgument`, `InvalidState`, *or* `Concurrency`. *It was found that three of the 18 initially defined general codes were never actually returned from a framework function.*

Part Time Architect

> Consider a large project lasting for several years, now in its second half. The initial architectural work is done, and the roles, duties and assignments of the lead technologists are to be changed.

After the initial architectural effort, the remaining questions to the architect will arise in ad hoc situations. The architect can not continue to work in the same role, trying to answer questions before they arise.

Architecture is a key factor in software development,	… but it is only one of a number of key success factors.
The architectural outline is done early in the project,	… but most architectural work is done in a piecemeal growth manner.
You are the best architect the software system could possibly have,	… but the system needs more than just you, and you need more than just this system.
You are valued for your expertise and advice,	… but other people also love to contribute and be recognized for their work.

Therefore, allow yourself only a limited time to care for the system in the architect's role. Let go as soon as the system can prosper with less care, and stay in control of only the very essentials of the architecture.

This can be done in several ways that also give other advantages. You could spend more time doing actual coding (as in ARCHITECT ALSO IMPLEMENTS), helping to finish the system. You could move forward to another project or other unrelated tasks. You could invite other team members to take some architectural responsibilities, giving them adequate career opportunities.

When your time as full time architect of the particular system has passed, strange things will happen. Colleagues and managers will pay less attention to your suggestions, some colleagues might try to occupy your position, your focus on structure might lead to over-design and unnecessarily slow down the development pace – in short, you potentially harm the system and your career.

The difficult part is to identify the right moment to let go. Similar to movie stars and politicians, you can tell that the moment was there. It takes a very self-conscious and ego-less person to declare herself partially superfluous, and to admit that the project would run faster without her work. If you initiate a change actively, you avoid that others have this impression first, and you can decide yourself to go back into full time mode when integration problems arise or major extensions are planned.

Whatever you decide to change, you face a difficult situation that you need to balance. The system still requires some guidance and a firm hand for consistent answers to arising questions, often on an ad-hoc basis and with a low response time, especially for integration problems near the project's end. To be able to fill this demand, you should keep off the critical path in other assignments[7]. On the other hand, most assignments only make sense and

[7] see also: LAZY LEADER, [*Coldewey98*]

offer learning and career opportunities when you are able to spend time and commitment – including the willingness to work on the most important issues.

Doing architecture on a part-time basis has a large political aspect and requires agreement with your manager. I have seen more than one project where project manager and architect were competing for their influence. All suggested measures take some control within the current project away from you, so you need to carefully consider your career and learning opportunities. The longer you stay with a project, the more thorough insights you can gain, and you gather experience-based knowledge of aspects related to admission, packaging, and shipping. However, you are also more likely perceived as "ordinary developer", which might not be your favorite role within the organization. So you need to balance your opportunities. In any case, you are better off if you suggest your favorite choice before somebody else does.

PART TIME ARCHITECT is intended to help when the architecture hinders the system development, such as in DESIGN BY SPLINTER. It can also be applied during the normal course of the project, especially when you start to REVERT THE ORGANIZATION [*Marquardt02a*].

This can start as a personal practice for architects, but in the mid term it requires management support. The costs depend on your organization and on the teams' ability to fill missing parts of the architecture by themselves. The costs are lower with teams used to agile development methods [*Cockburn02*] compared to teams following a traditional, waterfall-like process.

A side effect is the potential influence to your career. PART TIME ARCHITECT can be fostered by JOINT DESIGN, where more developers take over architectural tasks. The overdose effect can be lethal for a project: having no consistent architecture, and stopping being an architect. Beware of the overdose effects of any agent that decrease the influence of architecture. Do not lose your strengths! This is in no way a recommendation to drop all architectural efforts and to let loose the dilettantes.

One particular implementation of PART TIME ARCHITECT is ARCHITECT ALSO IMPLEMENTS. Note that PART TIME ARCHITECT goes nicely together with agile development methodologies where parts of the architecture emerge, and the architect's role can be filled by different project participants.

For a significant functional extension of a medical product, an architecture team was formed of four developers. One of them was the initial architect of the product. All architects remained the lead developers of their respective teams. This lead to a very quick exchange of experience, and the team established a productive and informal working basis. Product development proceeded smoothly and combined a successful follow-on product with a consistent architectural vision of the complete system. The four architects found a productive way of cooperation. While the initial architect acted as a MENTOR and improved his team and change management skills, the developers raised their architectural skill level.

Architect Also Implements[8]

> Consider a project where the architects have their desks among the developers, consult them for decisions and are consulted in turn, and suffer from the system's insufficiencies just like everybody else does.

After some initial effort in high-level system structure and guiding rules, you need a commonly shared vision of the software's architecture throughout the entire development.

Good architects have special skills in abstract thinking and communicating,	… but they are often also the most experienced developers.
An invisible outsider as architect is not able to convince the development team,	… but mere presence does not make a vision a shared one.
Architects might perceive the need for structural maintenance tasks,	… but these task are to be fulfilled by individual developers on a daily base.

Therefore, make the architect a developer, a primo inter pares[9] - in addition to her architectural tasks. Assign development tasks to her that are influenced by architectural decisions. It is common practice to let the architect implement the most difficult system parts, but take care to keep him off the critical path and plan some slack time for unforeseen architectural issues.

Leading by example brings a lot of short and long term benefits. You get a consistent system and a well educated development team, and a number of high quality feedback loops. This will most likely increase your system's development speed, its internal quality, and decrease its maintenance costs. On the other hand, an architect switching between different tasks might be less effective, and some of the tasks can not be completed as quickly as usual.

In huge industrial projects, it is fairly uncommon to have an architect do anything but architecture – whatever your organization may mean with this term. However, ARCHITECT ALSO IMPLEMENTS is a common policy in smaller teams. When different teams and organizations begin to cooperate, for example in projects crossing geographical and cultural borders, stating this policy explicitly is of major importance. Otherwise, misunderstandings will occur with respect to roles, responsibilities, influence and availability that can seriously hinder a successful cooperation.

ARCHITECT ALSO IMPLEMENTS helps you to keep the system in sync with the architecture, and the architect in sync with the development team. It is effective against problems that come with diminished architectural consequence, such as CROWDED PACKAGES.

This is common practice for a large number of small to medium sized teams. When introducing it anew, it requires management decisions, a clear definition of the architect's role, and the willingness to take the risk of delays in architectural decisions, or of delayed task completion respectively.

[8] initial version published by J.O.Coplien [*Coplien95*]
[9] Latin: first among peers

A side effect is the potential influence to your career. While you gain influence within the project team, your influence on a company level might be smaller than before.

ARCHITECT ALSO IMPLEMENTS is probably the most common one of the possible implementations of PART TIME ARCHITECT. It can help to overcome possible overdose effect introduced by NEGLECT THE LEVEL BELOW.

A large bi-national project established an architecture team, consisting of one architect from each site. One architect focused mainly on this formally mighty architecture team and did not participate in the daily development work. The other architect took more care for concrete daily questions than for the central architecture team. After eight months, the first architect had lost contact to his team to a degree that the team went astray and did not manage to deliver useful software. Finally the project was restructured, based only on results of the second team. The other development department was dissolved.

Reduce Architectural Broadness

> Consider a large project in tough times. While the architect is busy clarifying issues on a company level, important components are designed by the team with little to no architectural influence.

The architect begins to hinder the project's progress because the team relies too heavily on his time and expertise.

It is important to have an architecture in place to develop a consistent system,	… but architecture is not the only thing that is valuable about a system.
The architect is often the most experienced technician in the project,	… but not all tasks require the same level of expertise.
A good, consequent architect can be a MENTOR [*Marquardt02b*] to other developers,	… but an overly rigorous architect takes away self confidence of other developers.

Therefore, identify areas that are not central to the system's structure and keep them apart from the architects, from meticulous design reviews, and from discussions about their structure.

Let the architect focus on issues that cross the boundaries of the project or the organization, and that are of key importance for future projects. In these areas, an architect's expertise is mandatory. Allow small teams of seasoned developers to design large subsystems independently, leave them alone for a while and trust in their ability to finish successfully. The architect needs to work overtime less, and other developer gain experience.

As a starting point for your non-central areas outside of the architect's scope, again take a look at the components identified during DIVIDE ET IMPERA. Candidate components need to fulfill two criteria: they need to be leafs in the dependency tree that are not suppliers for several other components, and the developers working on them need to be both experienced with the task at hand and familiar with the BIG PICTURE ARCHITECTURE.

Experienced architects in a less experienced team tend to control other developers too tightly, even after those have learned a lot and could walk alone. The hardest part is not to find the right system part to let go, but the right time. It might help you to observe how the team and system evolves during the architect's vacation. If the team discusses similar issues and questions decisions, they are ready to gather their own experiences.

Make sure to enable very short feedback loops when requested by the developers, and plan for milestones when the architect and the team come together and discuss their achievements.

The main mechanism is to share responsibilities, and to allow yourself to let go. REDUCE ARCHITECTURAL BROADNESS is effective against all pathogens related to team paralysis without an architect's presence, and against over-managed dependencies as in DESIGN BY SPLINTER.

REDUCE ARCHITECTURAL BROADNESS can be a personal practice for architects, or a management decision. Its effort is neglectable, but you should prepare for the risk associated with the side effects.

Inexperienced developers outside of the architect's influence sphere are a clear counter indication. Check for this especially when REDUCE ARCHITECTURAL BROADNESS is initiated by management. A side effect is that you might need to get some system parts back into the architecture later, and need to do some severe refactoring then. Another side effect is a potentially lower prestige of your job – given that you strive for a technical career path. The overdose effect can be lethal for a project: having no consistent architecture.

REDUCE ARCHITECTURAL BROADNESS can be successfully combined with JOINT DESIGN [*Marquardt02b*]. This agent combination minimizes the individual side effects and negative consequences, and reduces the project's risk. It can go together with Part Time Architect, although the initial intent there is not to protect the team and system, but to unbind resources from less critical tasks.

While the architects of a large system struggled with a company wide component strategy in an inhomogeneous environment, seasoned developers developed a significant part of the GUI concepts. This way the system development proceeded, each developer did an important job in accordance to his abilities, and the architecture focused only at the most critical structures.

Conclusion

During the lecture of this article, you have learned about several therapies that can help you to prevent and cure architectural deficits of your system. Most of them have a broad spectrum of applicability that is only outlined by the context example. The presented therapies cover the key aspects relevant to the software architecture and the architect herself, although references and links to the surrounding environment are addressed where necessary.

The pattern literature has dealt with the presented topics before. Jim Coplien and Neil Harrison have published patterns about architecture and organization [*Coplien95*, *Coplien+*], Alistair Cockburn has written project management patterns using a pharmaceutical analogy [*Cockburn98*]. These works try to cover similar needs, but the approach presented here allows for a better linkage between the different therapy patterns and a broader range of different aspects via the more general diagnoses as entry points [*Marquardt01*].

However, the selected therapies leave more gaps than they fill so far. Besides the vast amount of obviously missing therapies, they need to cover more field experience, and a comparison between alternative agents. In parts, these comparisons are covered by other publications cited above. Future work is needed to collect the therapies in the distinct areas to reach completeness, and to explore alternatives in relation to each other. If you could point me to gaps or know how to fill some, please contact me at pattern@kmarquardt.de.

Acknowledgements

Many thanks to Andy Carlson, the VikingPLoP 2002 shepherd for these patterns. He provided a distant view on the patterns and forced me to improve my casual writing style, so that I could possibly get my points across to those readers who are not already familiar with my ideas. I owe the workshop participants at VikingPLoP a great debt, especially Neil Harrison for his constructive workshop lead.

Additional thanks to the shepherds for my EuroPLoP 2001 and 2002 submissions, Robert Hanmer, Neil Harrison, and Arno Haase, as well as to the participants of the respective workshops. This work would not have been possible without their encouraging, valuable and valued feedback.

References

AgileManifesto online at: http://www.agilealliance.org

Aguiar+01 Ademar Aguiar, Alexandre Sousa, Alexandre Pinto: Use-Case Controller. In: Proceedings of EuroPLoP 2001

Beck99 Kent Beck: Extreme Programming Explained: Embrace Change. Addison-Wesley 1999

Buschmann+96 Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, Michael Stal: Pattern-Oriented Software Architecture. Wiley 1996

Caesar Gaius Iulius Caesar: De bello gallico

Cockburn98 Alistair Cockburn: Surviving Object-Oriented Projects. Addison-Wesley 1998

Cockburn+01 Alistair Cockburn, Laurie Williams: The Costs and Benefits of Pair Programming. In: Succi, Marchesi: Extreme Programming Examined, Addison-Wesley 2001

Cockburn02 Alistair Cockburn: Agile Software Development. Addison-Wesley 2002

Coldewey98 Jens Coldewey: Lazy Leader. In: Proceedings of EuroPLoP 1998

Coplien95 James Coplien: A Generative Development-Process Pattern Language. In: Pattern Languages of Program Design, Addison-Wesley 1995

Coplien+ James Coplien, Neil Harrison: (to be published).
Online at: http://i44pc48.info.uni-karlsruhe.de/cgi-bin/OrgPatterns.book

DeMarco+92 Tom DeMarco, Timothy Lister: Peopleware. Dorset House 1992

Dikel+01 David Dikel, David Kane, James Wilson: Software Architecture. Organizational Principles and Patterns, Prentice Hall 2001

Dyson+02 Paul Dyson, Andy Longshaw: Patterns for Internet Architecture. Submitted to EuroPLoP 2002

Foote+00 Brian Foote, Joseph Yoder: Big Ball of Mud. In: Pattern Languages of Program Design 4, Addison-Wesley 2000

Fowler99 Martin Fowler: Refactoring. Addison-Wesley 1999

Malveau+01 Raphael Malveau, Thomas Mowbray: Software Architect Bootcamp, Prentice Hall 2001

Marquardt99 Klaus Marquardt: Patterns for Plug-Ins. In: Proceedings of EuroPLoP 1999

Marquardt01 Klaus Marquardt: Dependency Structures. Architectural Diagnoses and Therapies. In: Proceedings of EuroPLoP 2001

Marquardt02a Klaus Marquardt: Diagnoses from Software Organizations. To be published in: Proceedings of EuroPLoP 2002

Marquardt02b Klaus Marquardt: Patterns for the Treatment of System Dependencies. (Provisional title) To be published in: Proceedings of EuroPLoP 2002

Martin96 Robert Martin: Designing Object-Oriented C++ Applications using the Booch Method, Prentice-Hall 1996

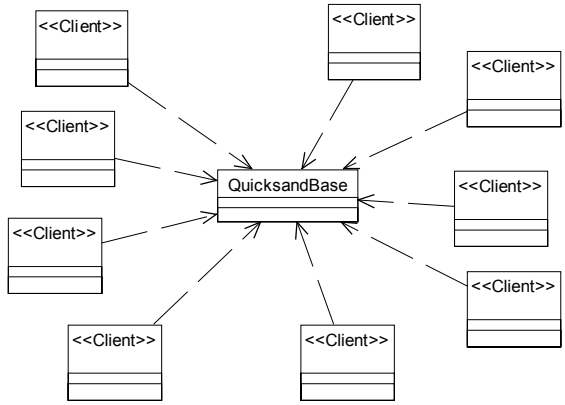
O'Callaghan99 Alan O'Callaghan: Patterns for Change. In: Proceedings of EuroPLoP 1999

Parnas94 David Lorge Parnas: Software Aging, Invited plenary talk, ICSE 1994

Rüping99 Andreas Rüping: Project Documentation Management. In: Proceedings of EuroPLoP 1999

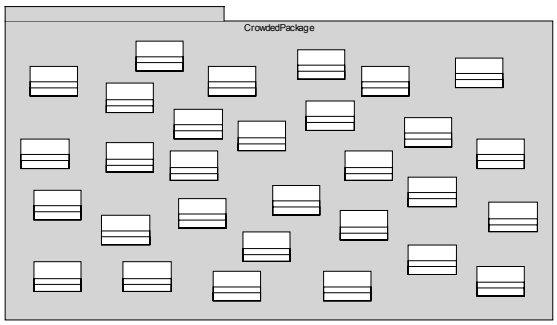
Appendix: Diagnoses Reference

For your convenience, here are some thumbnails of referenced diagnoses and patterns from previous publications [*Marquardt01*, *Marquardt02a*].



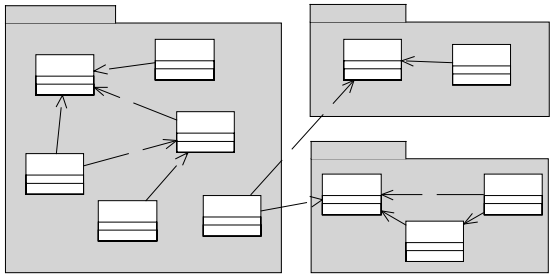
QUICKSAND BASE

A class is designed in a way that it is fundamental for the system and highly visible, but will be frequently changed during development



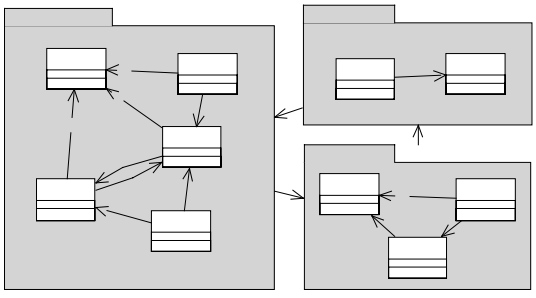
CROWDED PACKAGE

Missing architectural influence leads to unmanaged packages that are so filled up with classes that all overview or task parallelism becomes impossible



FOUNDLING

A class has few relations to other classes within the same package, but it is closely related to several classes from one or more other packages

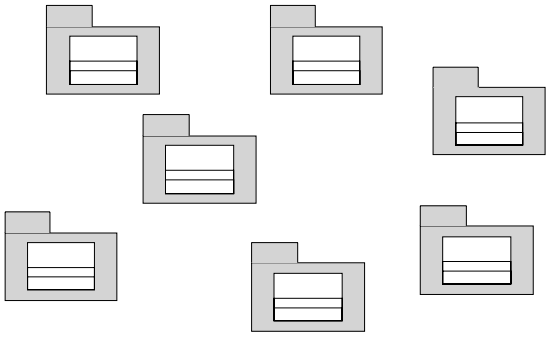


DEPENDENCY CYCLES

Packages or classes depend on each other directly or indirectly, forming one big chunk that must be used, reused, and maintained as a whole

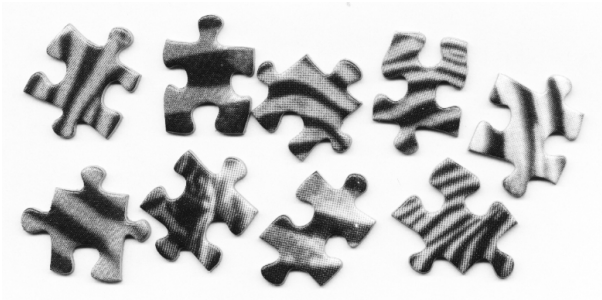
DESIGN BY SPLINTER

The static dependencies of a system are over-managed, leading to a complex of very small packages that is increasingly hard to understand



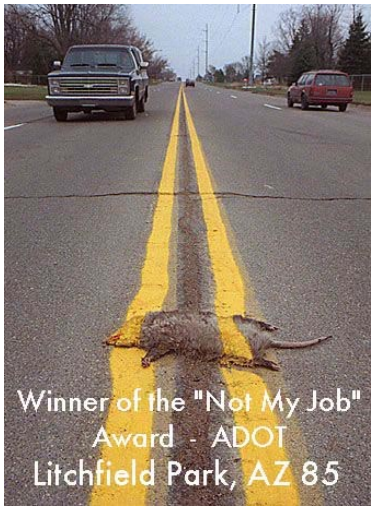
FUNCTION FOLLOWS FORM

Many classes are designed with a physical context in mind, like the process or device they will be executed on, and exhibit inconsistent responsibilities



UGLY INTEGRATION

Developers lead a careless life taking responsibility only for their own code. System responsibility is with persons considered “poor guys”



A language fragment of social antipatterns in systems development

Met-Mari Nielsen
mmn@daimi.au.dk
DAIMI
University of Aarhus
Ny Munkegade, DK-8000 Aarhus C, Denmark

Abstract

This paper presents a language fragment in the field of social antipatterns in systems development, most of which relate to the antipattern **Corncob**. The refactored solutions present ways to counter the negative consequences of **Corncob**s in different situations.

A minor issue of this paper was to find a form where the victims of the antipattern gain the knowledge relevant to a successful refactoring of their situation. To this end I have defined (yet another) antipattern format.

1. Introduction

The patterns presented in this paper all came together as part of a one-hour workshop on antipatterns at PLoPÅrhus the 28th of May 2002. As the talk concerned antipatterns based on examples from [Brown et. al. '98] and amongst them the **Corncob**[1], it is perhaps no surprise that **Corncob**s are in some ways instrumental in bringing about most of these antipatterns. What surprised me was how closely related these antipatterns actually were, even though the people present were a mix of developers and managers within systems development and none of the evening's five groups came up with identical antipatterns. The result was what I call a language fragment; by no means exhaustive, but closely related antipatterns.

I might note that the **Corncob**s in this paper refer to advocates of the pluralist organization [Morgan '86]: persons whose one and only goal is to gain and execute power to further their own position, and in this process causing problems for your project. Section 4 gives a more detailed description of the **corncob** antipattern.

2. A good solution is hard to find

One of the experiences of PLoP Århus so far was that most design patterns are relatively hard to identify or use in your own projects, whereas the antipattern discussion set us free to discuss solutions and implementations.

[1] All uses of existing pattern (or antipattern) names are boldfaced throughout the paper, for easy identification.

When it comes to social situations it can be difficult to explain to outsiders why and how things are done as they are, but antipatterns presented us with a workable way to explain (by example) how some roles and attitudes can become problematic and what the organizational consequences are. Somehow the bad examples were an easier way to access the total sum of our expert knowledge and what we thought was "good" project management (this may, however, be inherent to the Danish way of thinking).

The ideal company probably does not exist, and sometimes you just "put up" with some persons or practices because other aspects of your job make it worthwhile. While the larger context of your company or industry might cause some ingrained problems, the solutions I have chosen to present should all be possible to implement in small scale within a single team or project.

Most of the refactored solutions mirror my own opinion of good organizational culture. This also leads me to empathize that even though there might be other solutions viewed as equally good and stable, they would not lead to an organization where I would want to work (given the choice).

3. About the form

First of all antipatterns should present "easy" and "bad" solutions to a general problem and include some form of refactoring of the situation at hand, to reach a "better" solution [Long'01] [Brown et. al. '98].

3.1 Name

There must be appropriate names that easily convey meaning to both novices and experts: If a name already has been coined for some part of the antipattern then it might do for the whole. If a "victim" coined it, it is likely to be shared by other victims.
If it dips into some general culturally shared knowledge, then it might even take with those who haven't experienced the antipattern (yet).

> One of my favorite examples of easily understood names is Copliens pattern **Sacrificial lamb** [Coplien'95]. Describing why it is sometimes needed to make an example (of an perhaps otherwise innocent person), to place the blame and get on with a project.

One of the hardest problems in writing these patterns was finding valid English names (The patterns were initially conceived in Danish). A name carries great value especially if it can dig into some common knowledge or essence about the problem at hand. It simply helps focus the discussion and understanding of the pattern, not to mention that commonly shared concepts *are* a way of expressing patterns of human behavior. I therefore decided to include both the initial Danish name and the corresponding English one as pattern title.

3.2 Context

Since the antipatterns documented herein present a good deal of organizational/human behavior it seemed to me that the ethnographic pattern form might feasible for this format. Martin notes that design patterns are too problem-oriented and therefore skips the problem section, entirely concentrating on the contextual and organizational situation as seen, leaving an appropriate design to the computer scientists (or whoever is in charge of decisions) [Martin et. al. '01]. However, this approach will obviously not work when the main concern are for those affected by the antipattern to be able to refactor a solution. And hereby I reveal myself; because I wish these antipatterns to be refactored and understood

by those affected, not just as tools to be used by experts in organizational change (although they are welcome).

Even if a good name has been found, you need to convince people that this *is* a valid problem. Stories are a good way of relaying a context without preconceptions of former training [Erickson'96]. This is in a way close to Alexander's way of writing patterns because the stories as told often relay information of both the context and forces that must shape the refactoring process [Alexander et. al. '79].

> One of my favorite ways of describing shaping forces and conveying refactoring solutions come from Alexander's pattern **Connected play**, where studies of emotional problems show that psychosis correlate with number of childhood playmates[2]. The study has nothing to do with architecture but conveys at least to me how a viable neighborhood should look for families with children. But moreover it gives you power to change a potentially "bad" situation, "just" by making sure your child has a network of playmates (not necessarily including path systems between a minimum of 64 houses).

3.3 Refactoring

This leads me to the refactoring part of the antipattern. I have chosen those generic solutions acceptable to me as a person (and an employee), leaving my interpretation of "the good (work) life" open for discussion without really having any theory to back it up (just happy/unhappy co-workers to prove the point).

4. About the *Corncob*

As many of the antipatterns in the paper revolve around the Corncob, I would like to document what I find as important aspects within this antipattern.

Brown's general form states that: "[...] Corncobs focus much more on politics than technology, they are usually experts at manipulating politics at personal and/or organizational levels. Technology people can become unwilling easy victims of the Corncob's tactics." [Brown et. al. '98, p236].

Such people would find themselves at home in what Morgan names a "pluralist organization". A pluralist has the following worldview [Morgan '86, p188]:

- "Emphasis on the diversity of individual and group interests. The organization is regarded as a loose coalition which has just a passing interest in the formal goals of the organization."
- "Regards conflict as an inherent and ineradicable characteristic of organizational affairs and stresses its potentially positive or functional aspects."
- "Regards power as a crucial variable. Power is the medium through which conflict of interest are resolved. The organization is viewed as a plurality of power holders drawing their power from a plurality of sources."

The pluralist focus on conflict can actually be healthy for your organization! Morgan states that "[...] in group decision-making situations, where the absence of conflict provides conformity and "groupthink." The existence of rival points of view and of different aims and objectives can do much to improve the quality of decision making." ([Morgan '86] p.191).

[2] zero playmates giving 85% chance of severe emotional problems, more than four giving a 30% chance.

If such views are prevalent in your organization, then maybe it is you who are misplaced. However, Corncobs come forward in many situations, and thereby gives reason for the following antipattern:

4.1 Corncob a.k.a. Den Stadde

Many views exist on the relevance of the name corncob, in my search for something more tasteful, I came across the Danish expression a "stadde" (mid-jutlandish dialect for a kind of stubborn horse). Describing the situation where one horse in a four-in-hand does not pull it's worth, and therefore looks magnificent when the tour is over. This describes to me at least another aspect of the corncob, he is no team player.

Problem: "A difficult person (the Corncob) causes problems through destructive behaviours for a software development team or, even worse, throughout an enterprise. This person may be a member of the team or a member of external senior staff." [Brown et. al. '98, p236]

Antisolution: Sticking to rules and procedures, waiting for the corncob to recognize the need for at common goal. Any conflict resolved to the corncobs satisfaction will only support his view and use, of organizational politics.

Vignette 1, "The good":
A large company had a true corncob in charge of one of two sales divisions, that is not only responsibility of sales but also in charge of a number of semi-independent partners doing actual installation and support. The division's sales rate was terrific and earnings were the best of all departments. Unfortunately some of the splendor came from the fact that the **corncob** in question bullied the manager of the central R&D department into allocating ad hoc support hours for incomplete installations (billed to R&D). Every techie in sales thought him a great manager because he always managed to acquire the needed resources to complete an installation. Guess what the R&D people thought.

Vignette 2, "The bad & the ugly":
I haven't come across any easy vignettes, but they all involve long histories of small manipulations and unsuccessful meetings.

Refactoring:

- The antipattern for **corncob** states in known exceptions, that some situations can benefit from a corncob manager ([Brown et. al. 98] p. 237).
- Install procedures that emphasize the need for a common goal, and minimize the **corncobs** possibilities of impacting your own goals.

5. The Language fragment

The relations between the antipatterns presented in this paper are depicted in Figure 1. The antipatterns cover different strategies; there are the individual roles where for example. **Corncob** is instrumental in creating **faithful few,** and there are the symptoms or cultural strategies whereupon the **corncob** tries to manage his (or her) surroundings.

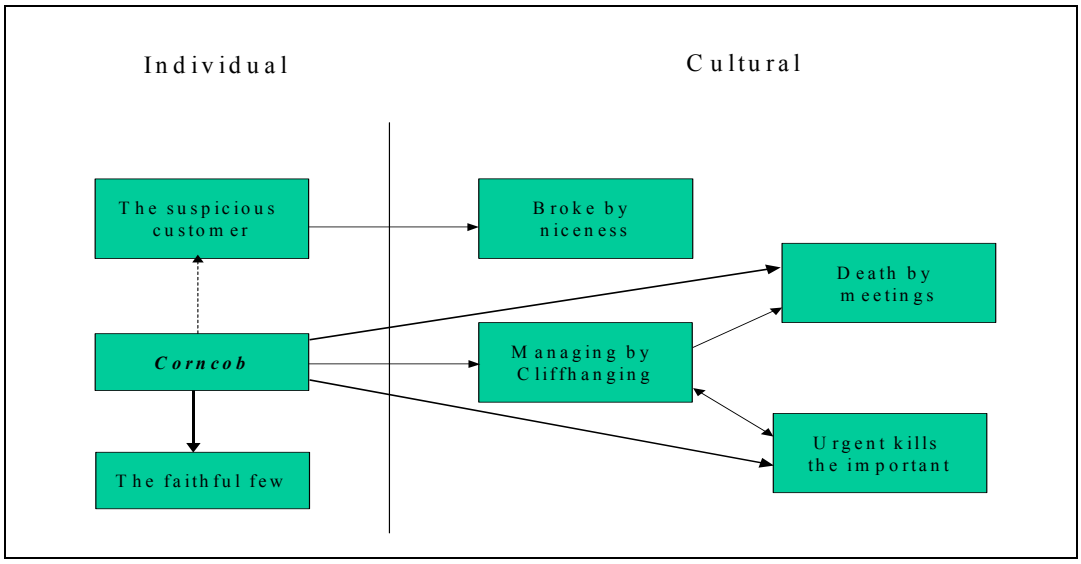


Figure 1: A language fragment of social antipatterns in systems development

Most of the antipatterns presented come in different flavors. A given strategy usually holds benefits for somebody (hopefully fore those implementing it). Therefore the antipatterns presented are chosen to show different consequences of a given strategy (as extremes of a continuum). From "the good" where strategy might even have a positive effect on the organization, to "the bad" where the majority involved wants a change and it is achievable, to "the ugly" where only extreme measures (like suicide or maybe murder) will cause a change.
As previously stated the problems and refactorings are chosen to achieve a situation perceived as "good" and "beneficial" for me and my friends.

5.1 The faithful few
a.k.a. Tordenskjolds soldater

"Tordenskjolds soldiers" is an expression from Danish history. The hero Tordenskjold let his soldiers march in ring, and thereby convinced the enemy that he had a very large army when in fact they only saw the same soldiers again and again (possibly a little out of breath from running around the back alleys).

Problem: Too much knowledge ends up in a few brains

- "The good": By divine hand only a few know what *it* is really about and therefore needed in almost any decision regarding their field. When the persons concerned literally are the only ones knowing what the company does or how the system works.
- "The bad": The same persons always ends up doing what others don't want to, documenting, testing... making coffee, mainly because others with more political savoir faire manages to push these tasks away from their desks.

Antisolution: Others "forget" to do the work themselves and rely on **The faithful few** to close the gaps. When such a person disappears (either by death or other reasons) the gap is sorely felt because of the missing knowledge. If you ever have tried to leave a job and kept getting requests and questions from your former colleagues then you was probably one of **The faithful few**.

Vignette 1, "The good":
In the company where I currently work we have a truck number of one[3]. Bo knows all about customers, installations and technology, and moreover he holds a lot of goodwill with the customers. The rest of the employees on the other hand know mainly about their chosen field of functionality (Even the director of the company has trouble defining what we do in general). We all want a bit of Bo's time because he knows all the critical bits and pieces of the business, from design to customer contacts. Therefore he doesn't get to do any work of his own. All day is spent responding to everybody else's needs. As Bo's greatest value to the company is his R&D, drastic measures went into ensuring than this was what he actually spent his time on. First he was sent on a lot of holidays just to relax and "think", but somehow customers got hold of his hotel number or there were "critical" problems to solve on the road.
The end solution possessed almost every refactoring they could think of at once. They continued the holidays, they hired me to document all their customer installations and generic modules (mainly Bo's knowledge), they restricted customers to call their product managers, Bo got official telephone hours once a week and he was sent home to work (without phone) twice a week. This has created a "partway" barrier between Bo and the customers but also between Bo and those in company who used to depend on him. While this happened some customers allocated new contact persons within their companies, which also helped a lot in "educating" them not to phone Bo at home at 5'oclock in the morning. These measures have gone a long way to distribute knowledge in the organization (People has to learn to do things themselves now). Unfortunately Bo likes to know everything there is to know, and this part of the antipattern has not been completely solved yet.

Vignette 2, "The bad":
Torben is a systems developer in a company where manpower is scarce, mainly because sales keeps selling more solutions than there are resources to develop. In the beginning Torben had great commitment to his company and was quickly known as somebody reliable to put in some extra hours and put out some fires. After a while all his hours was spent on the closure of **Fire drills**[4]. This became a boon to some of the previous programmers on the projects, because when installation time came Torben was a good place to put the blame for faulty functionality. As a consequence he also ended up with a lot of critical knowledge about the projects and therefore often was chosen to do further maintenance when needed (and again as the faults was his he deserved to right them, right?).

Refactoring: Even though this pattern doesn't seem to be a large organizational problem it is my experience that **The faithful few** only really appear in a wider context of antipatterns and therefore refactoring depends in part of recognizing these patterns as well.

Where there are **Corncob**s you usually have some of the faithful as well (I haven't seen **Corncob**s in relation to a project without their "faithful" e.g. those who do the work and take the blame). If this is the case you might try to refactor the **Corncob**s themselves. There are, though, cases of faithfuls without any real **corncobs** around

[3] How many people "need" to be hit by a truck before your project/company grinds to a halt.
[4] The **fire drill** minipattern, empathizes mismanagment [Brown et. al. '98]. From a system developers view the pattern results in a frantic implementation and testing phase within a ridiculuosly short timeline.

- A "soldier" can be successfully discharged if resources are allocated so that people have time to acquire an appropriate level of knowledge. The simplest strategy is to limit his exposure within the company (simply by letting him work at home, without a phone a few days a week); this forces people to cope without him.
- If the "soldiers" knowledge is really unique it might be an idea to document it in some way, maybe even hiring a cheap student to take interviews and do the tedious job of writing it down
- There might be many reasons why a particular job becomes undesirable and mostly this is connected with the organization culture. If such is the case, measures should be taken to make the job more desirable. Mostly these functions gets collected into a job-description, and someone are hired to do the "cleaning", after a while the person usually gets the idea that the job they are doing are not desirable. Maybe this is an antipattern in itself?
- Another way of dealing with the problem of to much knowledge in to few brains is to implement some pair techniques; indeed a suggestion at the PLoP meeting was to counter the pattern with pair programming and/or reviewing.

Related patterns: Corncob is usually a precondition for **faithful few**. As **managing by cliffhanging** can be a strategy to elicit sympathy sometimes **faithful few** employ this as a means to avoiding confrontation. When a job is undesirable it usually falls on the **faithful few**. All instances of **Sacrificial lamb** [Coplien'95] I know of, has been in highly political organizations where the lamb was one of the faithful ones.

5.2 The suspicious customer a.k.a. Den mistænksomme kunde

Sometimes you encounter a customer bent on having it "his way", calling you (or your company) a thief and a liar is just a part of the process. Sometimes the acquirement of something new transforms people into **corncobs** instead of resolving into a trust-relationship[5].

Problem: Your customer might have chosen this company as a provider, but that doesn't mean he has to trust your decisions.

Antisolution: Sometimes the solution becomes that of **Indulge the child;** if the customer has the money and wants to pay why not let him.

Vignette 1, "The good":
A large organization decided that they basically needed a database of a specific type of assets; with the demand to update and access locally and still have a central point to do cost statistics. They hired some consultants to make a requirement specification and later a provider to deliver the actual solution. When we (the project team) sat down to read the requirements, several design decisions were noted, which would severely impede performance. Most notably it required a copy of the database for each level of access rights and then synchronizations for every single update. This seemed unnecessary. When we tried to change this (and some other problems) the contact persons within the institution protested wildly. No change was to be made to the design, because then there wouldn't be any guarantee that the software would fulfill functionality demands. Months of discussions

[5] Linda Risings customer interaction patterns points the right way [Rising '99].

and pleading were the result. Every meeting, every phone conversation during this phase was concluded with the fact that they wouldn't trust any changes to the initial design.
The turning point came when the four *future* system administrators were presented with the administrator interface (we included demonstrations with both one and three databases). When responses came back from the meeting we were allowed to make whatever design changes we wanted (even though the four in question were not high ranking within their company or the project).

Vignette 2, "The bad":
A friend of mine was in charge of a project where the customer's system administrator was convinced she was upping the price. Every change request or update involved long negotiations whether this *really* was the time or money needed and usually the exchanges ended with the system administrator going back and trying to reformulate his demands so they would appear to be less demanding (but still resulting in the same functionality).

Refactoring: If you want to make beautiful code and systems that in a smooth way supports the customer's needs, you would want to refactor this situation. In my experience the access points, e.g. the customers contact persons are probably in their own way **Corncob**s, playing for power and stalling for time.
The way to deal with such situations is to partly play their game, circumvent the corncob and tap into another level of the organization.
This can be done as part of the development process by arranging one or more meetings where, for instance, the future users or those responsible are invited. Try to show them a lot of sensible solutions (not necessarily related to the real issue), build trust with others than the **corncob**.

Another solution is to let the customer go. Saying that you cant/won't provide what he wants... This might be necessary if it isn't possible to refactor as above (in "the bad" there was literally nobody else to access within the customer company).

Related patterns: This is basically **Corncob** as a customer. An objection to the proposed refactoring is that if you don't take the money and provide the product then you might end up **Broke by niceness**, making a product that the customer needs but doesn't want to pay for.

5.3 Broke by niceness

You could describe this pattern as the difference between sales and development in many organisations.
Problem: When trying to get a perfect result many project teams loose sight of the monetary issues involved.
Antisolution: Extend on the existing solution.

Vignette 1, "The good":
A company provided a solution ten years ago to a major customer. The system still has such a nice track record that the customer doesn't want to change hardware platform. The downside is that they are stuck with maintaining otherwise obsolete technologies and needing to keep track of the relic, when all other customers (indeed the industry) long since have discarded both the hard- and software.

Vignette 2, "The bad":

A switch company provided a hardware box to pick and choose automatically between the cheapest phone companies. They invented (and distributed) a piece of software that automatically updated their boxes. This made hardware updates practically unnecessary. However, as the company made their profit from these hardware updates, they went broke.

Refactoring: This is a point for which I have yet to see any clear-cut solution. Maybe because deep down I myself would like to provide a perfect system. The overall danger is just that you (and your company) get stuck in old technologies, supporting the same old solutions instead of spending time on developing new ones. Which of course in the long run might be bad.
The proposed solution is to abandon the perfect solution in favor of an appropriate one.

Related patterns: Trying to satisfy **the suspicious customer** might end up you in **broke by niceness.**

5.4 Managing by cliffhanging

Remember the old Zorro-series? Each episode would end with our hero in some sort of predicament, sometimes even falling over a cliff edge. This was the cliffhanger. The next episode would start with an explanation of how Zorro managed to survive his trouble, just to be thrown into some new adventure (of course helping the downtrodden along the way).

Problem: An unstable stressful situation "needs" to be prolonged.

When a situation is becoming critical, that is, filled with "unsolvable" problems with regard to the time available, people tend to get critical of the one in charge of relaying the bad news, instead of focusing on a possible solution.
Cliffhangers can be used to divert attention and create sympathy for the messenger and the cause while still keeping peoples attention

- "The good": Create unity/sympathy in a difficult situation
- "The ugly": To keep the employees/fellow co-workers from revolting

Antisolution: The messenger chooses to preface all news with something good (hopefully a solution to something) and ends with the "unsolvable" problems thereby postponing discussion (possible critique) and a probable solution, to a later meeting.

Vignette 1, "The good":
A company had a period when it was bought, first by one and then another cooperation. Amidst the organizational changes imposed by this situation, their product started to sell rather well. Every Friday employees sat down to a new dose of procedures, new projects, hiring rounds or hiring stops, and the way of coping was to weave these problems and sometimes solutions (for these came as well) into an ongoing cliffhanger which kept us with a notion of "we're all in this together..."

Vignette 2, "The ugly":
I know a company where management for four years has been on the cliffhanging premises. Manpower for systems development and support are short and products relying on designed, but not developed, software keeps getting sold. Every time there is an announcement from the department manager, people tense up. Afterwards they laugh it off, because every "solution" tends to be of the "do something worthwhile for your company..." and "two bottles of wine to those who helps finishing the product by Friday". Relief

prompts the laughter, this time they were not the ones sent to Brazil for a week of 25-hour workdays (and then promptly sent to England for a three-day **fire drill**). After years and years of it, most of the department finally said quits and started looking for work elsewhere, but management managed to keep their attention for another four months with promises of management and organizational change (later it was revealed that it wasn't their department that was subject to change).

Refactoring:

- If you wish to stop a cliffhanger it is necessary to see what the story is and identify the factors that cause the stress. This is mainly a question of company policy, whether to hire more people or to let down expectations.
- When a **Corncob** employs cliffhanging, his motivation is to prolong a situation in their favor. Try to deal with and solve the underlying problems, and the cliffhanger ends (just as a new and fair minded governor would have put Zorro out of the job).
- In some instances though this is not a bad pattern though, as it simply helps people cope with a situation full of immediate "unsolvables" and a lot of stress.

Related patterns: Cliffhanging can be employed by **Corncob**s to prolong a situation in their favor. This is also a form of antipattern to **Compensate success** [Coplien'95], as when rewards are used instead of allocating time/manpower.
Cliffhanging can also be used to uphold a permanent state of **Urgent kills the important** (As was the case of the company described in "The ugly"). Meetings are often the place where cliffhangers are told, and **death by meetings** can be a consequence of cliffhanging.

5.5 Death by meetings

Nothing happened, lets have a meeting!

Problem: To many meetings ends up with nothing to show

- "The bad": Nobody wants to take responsibility
- "The bad": Meetings are often seen as places to further personal agendas

Antisolution: Lets have a meeting to decide what to do and how to do it.

Vignette 1, "The bad":
A project in crisis was deemed "prestigious", meaning that the director really wanted to see the customer happy on this one. A programmer (incidentally one of **The faithful few**) was shipped overseas and even provided with a suit on the company credit card so as to appear keen and professional. When he arrived he was eager and prepared to work round the clock to smooth the bugs and bruised egos within the system. But what happened? Because the problems weren't easy to solve nothing much happened in the first four hours after the first welcome meeting. Then another meeting was called to clear why nothing had happened, and yet another and yet another. No problems were actually cleared on this trip because all the time was spent on meetings and preparing for them instead of actual problem solving. The time spent went to meetings about what the problems were and why nothing had happened since last meeting.

Vignette 2, "The ugly":
A company had the policy to have strategy meetings every three to four months, where management and key programmers sat together to prioritize present and coming projects. Karsten went to the first few with great hopes of incorporating the development of next

generation of company software (which was his initial job description) into the company plan. He ended up getting caught in power plays, and long discussions about who said what at the meeting and waiting for the next meeting where issues would be clarified.[6]

Refactoring:

- It seems to me that endless meetings is a hallmark of **Corncob**s, this is where power is demonstrated. The power to stall a decision is particular in this context.
- One source has reported that having stand-up meetings (as described in Extreme Programming) is a good way to solve some of the problems (but on the other hand then the winner is the one with most stamina).
- A well-defined meeting agenda, clear-cut roles and responsibilities.

Related patterns: Corncobs of course as they particularly indulge in this pattern.

5.6 Urgent kills the important

If your company continually keeps impossible schedules, every new customer or project is seen as important as the previous, the problem is just that the last (many) project isn't finished yet.
Note: This pattern corresponds to the root cause **Haste** in [Brown et. al. '98].
It is included as an antipattern because I have seen successful strategies of dealing with this situation in relation to some of the previously mentioned patterns.

Problem: Last in first out has become your way to deal with impossible workloads. The consequence in all the companies was that the systems delivered were somehow amputated and almost impossible to support.

Antisolution: This pattern ties deeply into cliffhanging. If for one reason or another you want to keep **urgent kills the important** as a permanent situation then **management by cliffhanging** can help you make a culture devoted to rush decisions and solutions.

Vignette 1, "The bad":
I know a company where documentation isn't considered worthwhile because what matters is getting the code out. Therefore no matter what your preferences were when first hired, nobody really documents anything anymore (nobody allocates time to do it).

Vignette 2, "The bad":
At another company, project plans were impossible to meet but forced through anyway by management with the notion to do as much as humanly possible.

Vignette 3, "The bad":
At a third company it was sales that had the final word in allocating resources. The result was that each new sale was given priority over all other projects (and that meant a new system every month or so).

Refactoring:

[6] This is the only case I know where attendees really wanted to do summaries just to ensure that their particular words were quoted "correctly".

If you want to be proud of the end result (and not the money earned or the amount of work done in a very short while) there are different strategies given the wider context:

There are a number of different ways, not necessarily mutually exclusive, to deal with this pattern:

1. Become a **Corncob** yourself and start digging into a power base of your own.
2. Make yourself a niche as somebody who goes against the culture.
3. If it is possible and desirable to change company policy then new procedures may be implemented in a few projects or with a few key persons for a start, but this demands that all involved are willing to change their way of working (beware of the **Corncob**s).
4. Last but not least: change company.

Note that both 2) and 3) can be successfully implemented with a shield of procedures or a **Gatekeeper** [Coplien'95].

6. Epilogue: Into the harbor

This paper is about antipatterns. Many projects can be successfully refactored, whereas others could not. When a company does not resolve their antipatterns, projects die. The process of not dealing with such an amount of antipatterns is covered by this last antipattern, with an unproven refactoring, which I present here:

6.1 Into the harbour
a.k.a. Kør projektet i havnen

Named after the Danish custom of "getting rid" of a car by driving it into the harbour and then claiming it was stolen. The name was coined to cover projects, which were swamped in antipatterns and other problems, where you sincerely meant that the system under development was a piece of junk never to be installed.

Problem: Too many antipatterns has killed the project
Antisolution: 1) start *all* over again or 2) abandon the project, and (crucially) place the blame somewhere else. Remember: make the customer believe it was unavoidable or other people's personal problems that caused the fiasco.

Vignettes: Unfortunately no company I know of wants to admit to such procedures in public (which is, of course, was the whole point of this antipattern).
Refactoring: I have never seen a dead project cause any real change. My only advice would be to build yourself **a beautiful company**.[7]

7. Acknowledgements

The (new) antipatterns presented in this article were partly workshop effort and partly formulated by individuals prior to the meeting. Not knowing where to place the blame I thus invite all the attendees of PLoPÅrhus to share the honor of this paper.
Thanks to Karsten Nielsen and Torben Jacobsen for good story material.
Last but not least thanks to Ulrik P. Shultz.

[7] *Patterns for Building a Beautiful Company*, Linda Rising, Caroline King, Daniel May, Steve Sanchez forthcoming in this publication.

8. References

[Martin et. al. ’01] Martin, D., Rodden, T., Rouncefield, M., Sommerville, I., Viller, S. *Finding patterns in the fieldwork*. In proceedings of the 7. conference on ECSCW, 2001

[Erickson’00] Erickson, T. *Lingua francas for design: Sacred places and pattern languages*. DIS ’00, Brooklyn New York, ACM 2000

[Erickson’96] Erickson T. *Design as storytelling*. in methods and tools 1996

[Long’01] Long J. *Software reuse antipatterns*. ACM SIGSOFT, software engineering notes vol. 26 no. 4, July 2001, p 68.

[Brown et. al.’98] Brown, Malveau, McCormic III, Mowbray *Anti patterns*. John Wiley & sons, Inc., 1998

[Alexander et. al. ‘79] Alexander, C., Ishikawa, S., Silverstein, M., Jacobson, M., Fiksdahl-King, I., Angel, S. A. *A pattern language*. Oxford University Press, Oxford, 1979

[Coplien’95] Coplien, J. O. *A generative development-process pattern language*. In Pattern Languages of Program Design (eds. J. O. Coplien and D. C. Schmidt). Reading, Mass: Addison-Wesley, 1995

[Morgan ‘86] Morgan, G. *Images of organization* SAGE Publications 1996

[Rising ‘99] Rising L. *Customer Interaction patterns,* In Pattern Language of Program Design 4 (eds. Foote, Harrison and Rohnert). AG communication systems corporation 1999

Patterns for Building a Beautiful Company

A work in progress. Last updated January 7, 2003

Linda Rising, risingl@acm.org
Caroline King, IAMCKING@aol.com
Daniel May, danielmay@yahoo.com
Steve Sanchez, Steve@mastermarble.com

When you see a beautiful company, the different parts mesh together. There is a sense of balance and peace in the meetings. There is a quiet efficiency in the hallways, offices and rooms. There is a sense of well−being and community among workers, management, customers, and partners. There is trust between management and workers and between employees. There is a vitality that arises out of the shared sense of purpose. In an ugly company, there is fear in the troops. Management is harsh and bureaucratic. Workers feel like they are on a death march.
Bruce Whitenack

Introduction

The patterns in this collection define a toolbox for building a beautiful company. We believe that these patterns will enable entrepreneurs to produce more than just a place where people turn up to punch in and out. We hope to create workspaces where people feel they're making a difference at some level, where they are free to be their best.

As we watch the Enron and WorldCom dramas unfold, we see a disturbing leadership pattern and its result. What's missing are whole (in the sense of complete) firms built on a foundation of possibility, integrity, and beauty. In many instances what we see today is that profit is everything.

We believe that to make money, you have to believe in the product or service you offer and care for the customers or clients you serve. That isn't a religious argument; it's a business lesson. When a company cares passionately about what they do and the people they do it for, magic can happen.

Beautiful companies value individuals−both customers and employees. These organizations are transparent and collaborative and respect relationships as the bedrock of all good businesses. Their structure is more a network than a top−down hierarchy. In these companies, fairness is a given; they value what's ethical above what's expedient. [Hefferman02]

"All happy families resemble one other," wrote Tolstoy in Anna Karenina, "but each unhappy family is unhappy in its own way." We have uncovered commonalities of beautiful companies in a series of interviews where the first question we ask is, "Do you believe you are trying to build a beautiful company?" The rest of the interview reveals what beauty means to the person being interviewed. The outcome is a collection of common solutions to the problems of running a small business.

Arie de Geus [deGeus97] states that there are two kinds of companies. The first he calls an economic company, run for purely economic purposes. People are treated as assets and the goal is to produce maximum results with minimum resources. The economic company is a corporate machine whose purpose is the production of wealth for a small group of managers and investors and feels no responsibility to the membership as a whole. In the second kind of company, return on investment is important but must complement the optimization of people. To produce profitability and longevity, care is taken to build a community: defining membership, establishing common values, recruiting people, developing their capabilities, assessing their potential, living up to a human contract, managing relationships with outsiders and contractors, and establishing policies for exiting the company gracefully. The values of the company coexist with the values of the individuals within the company and every member is aware of this coexistence. De Geus calls this organization a "Living Company." Our preference is for "Beautiful Company."

There's a lot of advice for entrepreneurs. The business world is drowning in self−help books. It's difficult to know where to turn. The world is rapidly changing. The rules are being rewritten as fast as we can learn them. The fundamentals are being replaced and the foundations are shifting. Success is being redefined. Even the well intended need guidance. How to begin? How to proceed?

In the beginning of our entrepreneurial journey, we brought all the books and learned to want more of the same kind of thing. The books set up a one−way conversation. They say, "Look what we did at Lucent, or IBM, or GE, or HP. But this information can be daunting and it doesn't always apply. Our intention is to reach entrepreneurs of small or mid−size companies in the early stages of developing their businesses.

There are no 100% beautiful companies. We don't intend to present a utopian viewpoint but an ideal to be approached by imperfect people. Along the way, there will be struggles and problems that must be solved. This work is a toolbox for building a beautiful company and the stories of how these tools have helped others.

Although we believe that many of the patterns could be applied to any business, our experience is mostly with small companies. As a result, we don't directly address issues with stockholders, for instance, or other concerns that small business owners usually don't face.

Profitability and success are of concern to any business. We believe that regardless of the product, regardless of the domain, regardless of the success criteria held by the entrepreneur, these patterns can be used to reach the desired goals. Usually people believe that their problems are unique, but what we find is a tremendous commonality among the challenges that small business entrepreneurs face. The exact product or widget doesn't make the difference. The common denominator is people.

Experienced patterns writers, Linda Rising and Daniel May, have teamed with Caroline King and Steve Sanchez, business coaches, to capture knowledge about running a successful small

business. Caroline has over 20 years experience as a business consultant for small business owners. Steve has been the owner of Master Marble in Phoenix, Arizona since 1990 and has been involved in business coaching and consulting for over seven years.

Acknowledgements

Thanks to our shepherd for PLoP ’02, Bruce Whitenack, for his encouragement and valuable suggestions. Thanks to our shepherd for VikingPLoP ’02, Klaus Marquardt, for adding an international perspective and helping our patterns grow.

The Structure of the Pattern Language

Beautiful Purpose. Chart the direction of your beautiful company by defining your purpose−not a product definition−a description of something deeper.

Beautiful Leadership. To build a beautiful company, lead from the heart and grow a workplace community.

> **It's a Small World.** Keep an open mind; look for opportunities to share interests; look for connections.
>
> **Know Your Limits.** In a small organization, you think you have to know everything but it's important to realize that no one can do everything.
>
> **The Right Coach.** When you're stuck and don't understand what's holding up progress, find a good business coach.

No Ordinary Workplace. Create a workspace for your beautiful company where people will feel like they're part of a happy family, where they will feel safe and free to grow.

Organizational Integrity. Creation of an extraordinary workplace depends on the balance between business reality and preservation of human values. Your company and everyone in it must live by its values.

Beautiful People. To grow a happy family, treat your people as volunteers and allow them to share in the business.

> **The Right Person for the Job.** To find Beautiful People, let everyone know the kind of person you want. It's a Small World. Trust your instincts and look beyond the resume. Set up an Audition.
>
> **Audition.** Let the applicant work for a short time to see if it's the right fit for both sides.
>
> **Be All That They Can Be.**
>
> **Graceful Exit.** Organizations are made up of people and people are constantly changing. Set up the expectation that separation is a natural occurrence.
>
> **Changing Conversations.** When dealing with people whose values seem different from yours, change the conversations you have. Instead of reflecting the unpleasantness, create a new intention and let them know you are sincere.

Beautiful Customers. Be authentic and expect your customers to be authentic.

Beautiful Outsiders. Treat those your company will interact with by explaining your intention and living up to it.

Beautiful Purpose

I'm the owner of Master Marble, a natural stone fabrication company in Phoenix, Arizona. I founded the company in the fall of 1990, with two employees. I think of myself as a creative entrepreneur able to make something out of nothing or do a lot with a little, but my real strength is to create a vision, share it with others, and help bring it to life. Steve Sanchez

???

You realize that there's more to creating a beautiful company than just creating a product, delivering services, and making money.

How do you chart your direction without a map?

Entrepreneurs get stuck in the details of the company and lose sight of the big picture. It's hard to see what's going wrong. People just keep making the same old meatloaf−maybe they try to make it faster or cheaper, but they stay inside what they know. They hope for a different new taste but keep using the same ingredients.

Companies need to know what they stand for. They need nonnegotiable, minimum standards. They need to be able to say, "We will not accept work that goes against our standards, because that's not who we are." [Webber02] Nineteenth century philosopher Thomas Carlyle said, "A man lives by believing something, not by debating and arguing about many things." Once you decide to decide, life becomes surprisingly simple. You don't have to think about certain issues or questions again. You simply get on with things and don't waste time and energy rehashing, debating, and arguing the problems and possibilities. [Marriott+97]

"To start with, unless we can define a purpose for this organization that we can all believe in, we might as well go home. That's 'purpose' as in, 'We the people of the United States of America, in order to form a more perfect union ...' The purpose has to be an authentic statement of what the organization is about. [Waldrop96]

Therefore:

Clearly define what your company is all about. This is not just a product definition−but a description of something deeper.

A beautiful purpose lights up, inspires, moves, touches, and creates a feeling of well being, comfort, safety and excitement within the company.

Identifying and capturing your purpose is a difficult task. You will need time for this activity. Start by answering the questions:

Who are we as a company?
What is our purpose?
What products/services do we provide?
What do we want to be/become/be known for?
Why are we special?
What are we proud of?
What difference do our products/services make?
What do we care about this?

David Packard posed the following questions: "I want to discuss why a company exists in the first place. In other words, why are we here? I think many people assume, wrongly, that a company exists simply to make money. While this is an important result of a company's existence, we have to go deeper and find the real reasons for our being." An effective way to get at purpose is to pose the question, "Why not just shut this organization down, cash out, and sell off the assets?" and to push for an answer that would be valid now and 100 years in the future. [Collins+94]

Don't just preach these values, institute concrete organizational mechanisms to stimulate change and improvement. [Collins+94] Once your purpose has been identified, when you know your purpose, you're clear on your goals, now articulate Vision and Mission Statements that bring your purpose to life. People will notice and respond to your purpose and alignment.

Century Roofing, Inc., with over 30 years experience in the Arizona roofing industry, is dedicated to providing the highest quality roof systems at a competitive price, while building lasting relationships based on friendship and trust.

Master Marble Ltd., the industry leader in providing the beauty and durability of natural stone, stands for excellence in customer service, craftsmanship and innovation. As a team we all flourish in a win/win environment.

Ginger L. Price, D.D.S., is a professional and personable dental team committed to providing the most comprehensive quality care in a gentle manner to individuals who value themselves and their health.

Beautiful Leadership

The leader of the company is like the captain of a ship. Even though not everyone on the ship deals directly with the captain, everyone has an eye and ear on the captain. The captain leads by example and sets the tone for the rest of the company. Not all captains are up to it. I've seen captains that organize meetings, maintain the status quo, always seek consensus, stabilize the environment, retain tight control, and remain hidden in the background. But isn't this more like a caretaker or administrator? Surely there must be something more. A captain leads, challenges, and inspires. A captain shouldn't make dangerous decisions, but if no risks are taken to push forward, then I don't think that's leadership. I think most employees understand that not every decision made will be perfect or correct, but they expect the captain to show leadership.

???

You're an entrepreneur who wants to build a beautiful company. You've defined a Beautiful Purpose.

How can you grow and maintain a challenging and nurturing environment?

I look back at the leaders I admired and they all had hearts as big as houses. It's the heart that is the essence of leadership. Create an atmosphere where healthy interactions have their best chance to happen. You do that in various ways. Maybe you make a cult of quality work or you instill in people the sense that the group is, in some sense, at least, an elite, the best in the world. You get them to think about integrity and all the baggage that word carries. Whatever it is, there has to be something that unifies the group. I think of it as soul. The human creature has built into its firmware a need to be part of a community. People who feel part of a community do better. And in today's sterile modern world, there isn't much community to be had. For most of us, the best chance of a community is at work. So building soul into the organization is really an exercise in community building. The soul you foster in the organization is the seed around which community begins to form. [DeMarco97]

The art of leadership is liberating people to do what is required of them in the most effective and humane way possible. The leader removes obstacles that prevent them from doing their jobs. The true leader enables his followers to realize their full potential. To do this, leaders must be clear about their own beliefs. They must have thought through their assumptions about human nature, the role of the organization, the measurement of performance, and a host of other important issues. Leaders must have the self−confidence to encourage contrary opinions, an important source of vitality. The true leader is a listener. The leader listens to the ideas, needs, aspirations, and wishes of his followers and then, within the context of his well−developed system of beliefs, responds appropriately. That is why the leader must know his own mind. That is why leadership requires ideas. [DePree89]

The first responsibility of a leader is to define reality. The last is to say thank you. In between, the leader must become a servant of his followers and a debtor. That sums up the progress of an artful leader. Leadership is learned over time, a belief, a condition of the heart, much more an art than a set of things to do. [DePree89]

There's a terrible defect at the core of our thinking about people and organizations today. There is little or no tolerance for the character−building conversations that pave the way for meaningful change. We're stuck, lost, riveted by the objective domain. That's where our metrics are; that's where we look for solutions. It's the come−on of the consulting industry and the domain of all the books, magazines, and training programs out there. That's why books and magazines that

have numbers in their titles sell so well. We'll do anything to avoid facing the basic, underlying questions: How do we make truly difficult choices? The problem is, when you're stuck, you're not likely to make progress by using competence as your tool. Instead, progress requires commitment to two things. First, you need to dedicate yourself to understanding yourself better−in the philosophical sense of understanding what it means to exist as a human being in the world. Second, you need to change your habits of thought: how you think, what you value, how you work, how you connect with people, how you learn, what you expect from life, and how you manage frustration. Changing those habits means changing your way of being intelligent. It means moving from a non−leadership mind to a leadership mind. [LaBarre00]

Real leadership conveys vision, engenders confidence and encourages striving toward common goals. Leadership is the ability to enroll people in your agenda. Meaningful acts of leadership usually cause people to accept some short−term pain to increase the long−term benefit. We need leaders for this because we all tend to be short−term thinkers. [DeMarco01]

Therefore:

Lead from the heart and create a happy family within your workplace.

Real leadership needs the following:

- Clear articulation of a direction.
- Frank admission of the short−term pain.
- Follow−up.
- Follow−up.
- Follow−up.

When we're presented with the first of these and none of the others, it's not leadership; it's nothing more than posturing. [DeMarco01] The follow−up, of course, should include a committed action plan that will produce what you want to produce.

The people at Herman Miller have become my second family. [DePree89]

It's a Small World

Here's how Erik's Bike shop finds employees. There is a small pool of individuals who love to tinker with bikes. It's not a get−rich industry, so you have to love bikes to work in it. Where do you find employees with those characteristics? They come into the store all day long! Erik has hired so many customers that all the managers look at customers as potential employees. When they get to know one who would fit into the company, they ask, "How would you like to work with us?"[Dauten99]

???

You're an entrepreneur who wants to create a beautiful company. You're trying to practice Beautiful Leadership.

How can you get what you need, acquire new insights, and connections to new people?

Organizations, like all human groups, operate through conversation. [Senge+99]

In the first issue of *Psychology Today*, back in 1967, Stanley Milgram described his "small world" experiment. If you choose any two people in the world at random, how many acquaintances would be needed to create a chain between them? He started from places such as Nebraska and asked subjects to send a folder through the mail to a target person in cities like Boston. The starters had to send the folder to someone they knew on a first−name basis. That person was to send the folder to someone closer, and so on. Incredibly, Milgram reported that it took only five people in six jumps to reach a random stranger. But Milgram's startling conclusion turns out to rest on scanty evidence. The idea of "six degrees of separation" may, in fact, be plain wrong−the academic equivalent of an urban myth. There is some evidence that Milgram might be right in spite of his own research. When we say, "It's a small world," we are not talking about the chances of connection between two people taken at random. We are talking about the chances of meeting a person who can help us meet a goal. Over a lifetime, these chances are high, especially for educated people who travel in similar networks. And when an especially unlikely connection occurs, the world does feel small whether or not scientific evidence supports it. [Kleinfield02]

Therefore:

Look for opportunities to share interests; look for connections.

The people you meet everyday are the quickest way to get what you want. If you're unemployed, they'll help you find a job. Most jobs are found through friends or personal connections. If you're looking for an employee, personal contacts will help you find the right person.

When you make a connection, maintain it. Don't be shy about making your interests and needs known to friends and acquaintances. They may not have what you need now but if they know what you're looking for, they'll pass your name along or refer you to a useful contact.

We had signed a lease to go into a new building but the tenants below blocked construction−so we had to abandon the project. We had already told our landlord that we were moving, so from August through November we were without office space. The first thing I did was to call everyone I knew. One friend didn't use her office on Fridays. Another let me work on Tuesday and my partner work on Wednesday and then we both worked at another office on Friday. The patients really got involved. It was a real test. If we hadn't been well established and well

connected, patients would have gone somewhere else. I guess it says a lot about who we are and a lot about our friends. [Price02]

It never fails to amaze me how so many opportunities and problems are resolved in a serendipitous fashion. I talk to people about some problems we're facing or what we're doing in our company. Invariably, someone will say, "I know about this" or "Maybe I know someone who may be able to help." It's uncanny how often problems are solved and new ideas, contacts, and opportunities are created in this way. You don't always end up in the place that you thought you would, but you end up learning some valuable things and meeting interesting people−sometimes, new partners and employees in your company. Someone once said that a company is really just a set of conversations; I think there's some truth in that. Daniel May

Know Your Limits

I rarely have all the answers or knowledge to do everything myself but I believe that if I recognize that I don't know something or can't do it then I can find someone who does and give it away to them. I still hold the responsibility for the result but it creates a kind of unlimitedness for me. It is from this place that I now manage my company. Steve Sanchez

???

You're an entrepreneur who wants to build a beautiful company. You're trying to exercise Beautiful Leadership.

In a small organization, you think you have to know everything.

When you're an entrepreneur, you think you know everything already, and you're nervous about letting go. But that's what you have to do to build the company. Some leaders won't understand the day−to−day details of what their people are doing and might worry that their lack of understanding will undermine their ability to manage. Any job has numerous facets. Don't despair. You achieved your position because of your strengths, not in spite of your weaknesses. Those who are successful are honest about what they know, use their skills to their fullest, and actively seek to grow their skills in new directions. Not understanding the details doesn't get in the way if you use your strengths well. [Kruger99]

The hallmark of great leadership is knowing your limits. That's the essence of where I go wrong. I get so convinced of my knowledge that I blind myself to evidence proving that what I seem to know is wrong. [DeMarco97]

If you're too busy hiding your lack of knowledge, you won't feel comfortable asking questions. You'll think you should already know the answers. Let me tell you a secret: no one knows all the answers. No one. Not knowing the answers isn't a sign of weakness−not asking the questions is. Even if you don't understand the answers to your questions, keep asking questions until you get the answers in terms you can understand. Asking questions helps you understand what your staff does on a day−to−day basis. It also enables you to gauge the effort your staff puts into solving a given problem. Finally, asking questions causes your staff to questions their own assumptions− and that helps them improve their own work. [Hendrickson02]

It is inefficient to make up for the "imperfections" of others by working harder or longer hours yourself. By doing so, you may increase your output by 25−30% over a limited period. You might gain 1/3 of a man−year. If you create the conditions under which 10 people will each produce 10% more you will have gained one full man−year. [deGeus97]

Therefore:

Know your limits. No one can do everything.

The first thing to do is admit what you don't know. Misrepresenting your skills to your staff will always backfire. Learn something about the work your people do. You don't need to become so proficient that you could take over for any one of them. You just need to understand the ins and outs of the job well enough that you can anticipate what your people need and understand the importance of what they're saying. If you're unsure about where to start learning, ask your people. Not knowing where to put the semicolons isn't a big deal. Knowing how to lead people, now that's a big deal. [Hendrickson02]

One clue for recognizing limitations: when you're taking too long to decide something. When the issue is well within your expertise, decision−making is faster. Long deliberations usually mean that you lack information. Ask for Help [Manns+02].

Embrace the fact that there is an entire body of knowledge that you don't know. You might have looked at not knowing something as a weakness or that something is wrong. Two sides of the same coin. Acknowledging that you are keeping an "open mind" gives you a sense of freedom and power. Exactly the opposite of how you feel when you don't know the answer.

By acknowledging the areas where you need help and delegating to an expert you become unlimited. You retain responsibility for the outcome but your limitation does not become a hindrance. You have room to move.

In the early years I would hang on tooth and nail to every micro management decision and then get all worked up when things went wrong. I couldn't see that I was afraid of not looking good or that something would happen in the company that would not live up to my inner vision. I tried to force everyone to do things my way. I spent more energy making sure they did it my way than on the result. In the product and company I produced, I was scared something would go wrong that would make me look bad, so I got mad at my staff to scare them too! That's how I started. It only took about 10 years to get it. I never said I was smart. Steve Sanchez

Here's something that trips up a lot of entrepreneurs: You can't do it all. I made a list of my strengths and weaknesses to determine where I should devote my energies and whom I should hire to help. [Kurtzig91]

My manager didn't know a macro from a make file, but he knew what the customers needed and how to motivate the developers to give it to them. He helped us understand what the customers needed and improved the relationships between sales and development. [Hendrickson02]

The Right Coach

When I started my business, I worried about being on track in the whole scope of the business. I heard about SCORE, the Service Corps of Retired Executives, a resource partner with the U.S. Small Business Administration. They paired me with a mentor, a great guy who had owned a steel fabrication company. He tried to compare our companies' products but the analysis didn't fit. We were talking about widgets. We worked for 6 months until we both lost interest. A friend introduced me to Caroline. When she came in, she asked for different information. She asked me who I was trying to be and what was working and what wasn't working. We started by transforming me and translating that to the business. She still continues to coach me−even after 7 years. I still have breakthroughs. Now I understand that widgets don't matter. It's the organization, the management, the environment, and most of all, the people that matter. Steve Sanchez

???

You're an entrepreneur trying to build a beautiful company and exercise Beautiful Leadership.

You don't understand what's holding up progress. You are worried about whether you are on the right track.

Mastery is a three−stage process. The first is superficial understanding. (This seems logical enough. I get the idea. Let's give it a try.) The next stage comes when practice is attempted. This can cause stress and apprehension. (I didn't understand the full implication of this. I'm not sure we're up to it.) The third stage causes introspection, personal distress, and, if you are persistent, inner change. (This is not just about the organization; it's about me.) This third stage is difficult because the focus is on you. It is demanding but holds the most promise. When your organization faces a challenge and you're part of it, you're always the part that's easiest to change−as hard as that is! [Pascale+00] Entrepreneurs have problems due to their lack of experience and expertise in running a company. They get to a certain point and cannot go beyond it.

A coach is like a mirror. You could dress yourself without a mirror but you'd risk not getting it right. Tiger Woods has several coaches. Michael Jordan said he would leave the game if he lost his coach. Even Pavarotti has an acting coach, a voice coach, a music coach, a language coach, and a personal fitness coach. Pavarotti wears 50−100 pounds of costume, sings for 2 hours, raises his voice to be heard in perfect pitch without amplification by 5,000 people, while acting in a very demanding role, frequently in a language other than his native Italian. Coaching took Luciano Pavarotti from just a good voice to legendary operatic status. A coach enables progress by helping you realize where you are and how you are behaving and then moving you to make new choices.

Therefore:

Find the right business coach.

Ask people at other companies for recommendations. Interview several coaches and find The Right Person for the Job, for you and your business. Set up an Audition: enter into a short−term agreement−6 months to a year, set some strategic goals and see what you accomplish in that time frame.

Master Marble has spawned six entrepreneurs that set up competing businesses. These were not friendly competitors. Of the six, none had coaching, and only one survived. All the other

businesses died within 2 years. The lone successful entrepreneur had been with Master Marble a long time and had received coaching while an employee. It is important to get the correct support, a coach is not the same as a mentor or casual advisor. Remember that I went through many, many advisors and others before Caroline was able to open up the possibilities that finale transformed Master Marble. I believe nothing takes the place of a coach. Just getting "help" could just be more of the same old meatloaf especially if the person seeking assistance doesn't go outside of their comfort zone to find "unattached" input. Steve Sanchez

CK had a client who worked with her for 3−6 months but let her go because he felt he couldn't afford her fee. Six months after she left, he closed his business and went to work for another employer. He didn't fail because of not making profit, he just was not willing to do the things that were needed to make his business a success. Some clients shrink right up, once they leave coaching. Coaching is confrontational − you confront yourself. You have to be willing to leave yourself behind for who you can become. I think coaching is valuable for any business large, small or in between, and essential for all entrepreneurs. I think it's not just "get a coach," but find the Right coach. All coaches don't fit all companies or all small business owners and different coaches can work at different times for the same company. Caroline King

When I started my company, I felt something inside but didn't really follow it. Now I realize that and that's what creates beauty in our company. There's something within us−you have to access that−otherwise you're just doing, doing, doing, and never getting anywhere. The business world is about profit, forcing things, pushing. In the beginning the spiritual side and the business side were separate. Now they are connected. It happened because I got the right coach. First there was a lot of crying, then uncovering integrity and alignment with my greater self. Then there was a lot of cleaning up of my business transactions to include personal integrity and honoring my word. It affects every part of your business. If you take a stand, it's going to affect everything. I also thought about whether I'm doing what I love to do. When you look at the service you're providing, you have to decide whether or not it's what you really love. You're always exploring and always learning and always in transformation. As you find new interests, you need to incorporate them. Coaching can help you do that. [Rike02]

The controller of a company made a financial error in the company's books. The owner of the company was very angry and exploded at the controller but then forgot the incident. After this, the controller felt the owner no longer trusted her and decided to quit her job. She turned in her resignation, but did not discuss her decision with her employer. At this point, I was having a coaching session with the owner, who was surprised and dismayed with the controller's decision to leave the company; but hadn't talked to her about it. I brought all parties together and encouraged them to talk about the incident. That allowed them to express their concerns. The controller was reminded of how her father had treated her when she made a mistake−−he withheld his love, but never talked about it. She had decided that it was easier to leave (home) than to talk about it. The owner was not aware of treating her differently after the incident. He knew that he hadn't told her that he forgave her for the mistake and when he did, she withdrew her resignation and stayed with the company. Until coaching allowed this incident to be discussed, the owner did not realize that his reaction to the mistake had made such an impact on the controller, so she felt she had to leave because she thought he was disappointed in her performance. The controller didn't know that the owner had forgiven her and he never would have communicated without the coaching, because he felt it was "no big deal", since "everyone makes mistakes." Caroline King

Changing Conversations

I gave a quote to an ongoing client−a general contractor. He presented the contract to his client−the homeowner−and it was accepted. The next day the homeowner called to try to lower the contract amount by demeaning our price. When I remained firm, she was not happy. She wanted to quibble over $250 in a $13,000 contract. If she had asked for a reduction in the contract, I would have agreed but not when she was nasty. The mirror operates this way. We can reflect what an individual projects or replace it with something else. I chose to reflect the same energy she sent out. I have since reconsidered and gave the general contractor a $350 concession. I called the homeowner to say, "I want to apologize for being rude the last time we spoke. I also want to set an intention that when we are finished with this project, you will love me, my company, and our product." I think she was a little taken back but said, "Thank you." I have had several conversations with her about the project and we are communicating nicely. I gave up having to be right and reflecting rudeness back to the client. I know that my company will provide excellent service on time and on budget with fantastic quality and that this will be the start of an exceptional relationship. Steve Sanchez

???

You're trying to build a beautiful company and look for Beautiful People and Beautiful Customers and Beautiful Outsiders but it doesn't always happen.

Sometimes you have to deal with people whose values don't seem to match yours.

"Inside−out" means to start first with self; even more fundamentally, to start with the most inside part of self−with your paradigms, your character, and your motives. If you want to have a happy marriage, be the kind of person who generates positive energy and sidesteps negative energy rather than empowering it. If you want to have a more pleasant, cooperative teenager, be a more understanding, empathic, consistent, loving parent. If you want to have more freedom, more latitude in your job, be a more responsible, a more helpful, a more contributing employee. If you want to be trusted, be trustworthy. [Covey89]

Therefore:

Change the conversations you have. Instead of reflecting unpleasantness, create a new intention and let others know you are sincere.

Change your own point of view to see things in a positive light. There is always something good to see in everyone. It isn't necessary to be in perfect agreement to focus on the good. Our greatest lessons come from those who push us and challenge us and force us to take a different point of view.

You can say, "If a person of your intelligence and competence and commitments disagrees with me, then there must be something to your disagreement that I don't understand, and I need to understand it. You have a perspective, a frame of reference I need to look at." When someone disagrees with you, you can say, "Good! You see it differently." You don't have to agree with them; you can simply affirm them. And you can seek to understand. [Covey89]

You can't win them all. Sometimes the best solution is to let the person in the mirror take himself out.

When someone comes along who isn't so nice−they just take themselves out. We had one patient who didn't like women. I don't know why he chose a dentist that was a woman. You think that if you're nice then they'll be nice but they just get nastier and nastier. I explained that dentistry wasn't his area of expertise and that I knew more than he did. We did one procedure on him and he never came back. [Price02]

At patterns conferences, we have writers' workshops, where small groups of people consider papers. We follow a strict process in the workshop and, over time, I've come to see the wisdom in it. After the introduction, the first step is to identify things you like about the paper. There have been times when I have read through some very poor papers and had a real struggle finding something to bring up at this point in the workshop. When I offer a meager positive comment, I am always surprised to hear what others have to say. They always see something good that I missed but that I agree with and, further, that causes me to see something else that's good about the work. Since everyone in the workshop has this experience, the positive comments take on a life of their own and grow to easily fill the space allowed for this step. Even though this is the rule, I'm always astonished and pleased when I see this happen. Linda Rising

My whole family worked for the company but I had just forced my brother out. We didn't see eye−to−eye. It wasn't working. I told him he couldn't be in sales−he could be in manufacturing if he wanted and stay with the company. He wouldn't have it and left. He was the top salesman in the company. I really didn't know what would happen when he left. He gave them deals they couldn't refuse. He was cutting prices. I worried that the company would go bankrupt. I was struggling to make sense of the contracts he left behind. The phone rang and I snapped a nasty "Hello!" It was Caroline. I'm afraid I was very short and, OK, I was nasty! Didn't she know that I was overwhelmed? What she told me brought me around. She said that if I saw myself as overwhelmed and things as hopeless, well, that's what they would be. She suggested that I say "I'm challenged but I can handle it." It made a difference. If you want a better meatloaf, you've got to change the ingredients. Steve Sanchez

No Ordinary Workplace

I always enjoyed coming to our office, seeing familiar faces. I knew that I could relax with a drink in the conference room and catch up with the war stories from other people, or talk to them about a tough technical problem or difficult client. Our cubicles had whiteboards so we could express ourselves with diagrams (as consultants do!) and the open office space let you see who was in. We had a library where you could find references, or hide with someone to talk about something confidential. It was certainly more than just a working environment or an office; it was a place and collection of people that I looked forward to seeing. Every time I walked through the door, I'd be thinking: what's up today? And it was always a bit exciting. Daniel May

???

You're an entrepreneur who wants to build a beautiful company. You've defined a Beautiful Purpose, started to exercise Beautiful Leadership, and are ready to start hiring.

How can you attract and keep Beautiful People, Beautiful Customers, and Beautiful Outsiders?

The environment should enable and empower people to do their best. The environment should encourage a rising level of knowledge about corporate life: literacy about business, the competition, relationships, and ownership. The environment must encourage lavish communication. The environment should be a place of realized potential. It should be a "high touch" place, a place where we connect persons to each other and to technology in an effective and human way. [DePree89]

Therefore:

Create a space where people will feel like they're part of a happy family, where they will feel safe and free to grow.

The quality of the environment will be determined by Organizational Integrity.

We try to make work more fun: after−hours outings to baseball games and bowling alleys, a basketball tournament, and floorwide "super loader" contests. We know that these are monotonous jobs. We want to make it less mechanical and more social. People don't want to feel like robots. And if they're happy, they'll take the missorts seriously. They'll treat other people right, and the quality will go up. Because, hey, they know that guy −− they played volleyball with him. [Hammonds02]

In dentistry, people are afraid. What our patients tell us is that when they come into our office, the environment makes them feel good. This is a "Cheers" kind of place−where everybody knows your name. We try to build relationships with our patients. It starts in the front office. The first time we do a procedure on a patient, the team goes the extra mile to make that person comfortable. We customize things to take the edge off. [Price02]

Organizational Integrity

A small computer software company developed some software that they sold in a 5−year contract to a bank. The bank president was excited about it, but his people weren't really behind it. A month later a new president took over. He was uncomfortable with the software conversion. The software company was in deep financial trouble. The president of the software company knew he had every legal right to enforce the contract but he had become convinced of the value of integrity. He told the bank president, "We have a contract. Your bank has secured our products and services to convert your organization to the new application but we understand that you're not happy with it. We will give you back the contract and your deposit. If you are ever looking for a software solution in the future, come back and see us." Walking away from this contract was almost financial suicide (loss of an $84,000 contract) but the president of the software company felt that in the long run, if the principle held, it would have a payback. Three months later, the bank president called and said, "I'm going to make some changes in our data processing system, and I want to do business with you." He signed a contract for $240,000. [Covey89]

???

How do we cope with business reality and still preserve human values?

The professional learns how to diagnose, how to understand. He also learns how to relate people's needs to his products and services, and he has to have the integrity to say, "My product or service will not meet that need," if it will not. This requires a purpose, a clear sense of direction and value, a burning "Yes!" inside that makes it possible to say "No," to other things. It also requires independent will, the power to do something when you don't want to do it, to be a function of your values rather than a function of the impulse or desire of any given moment. It's the power to act with integrity. [Covey89]

When times are tough, integrity matters most. Four in five employees say their organization's integrity is an important reason to stay. Misconduct drops and satisfaction rises when leaders model by ethical behavior. [Bentley]

By "alignment" we mean simply that all the elements of a company work together within the context of the company's purpose. Research shows that individuals pick up on all the signals in their work environment as cues for how they should behave. People want to believe in their company's vision, but will be ever watchful for the tiny inconsistencies that allow them to say, "Aha! See! I knew management was just blowing smoke. They don't really believe their own rhetoric." [Collins+94]

Therefore:

Your company and everyone in it must live by its values.

People are either committed to integrity or they're not. When employees practice integrity in their personal life, they will most likely perform with integrity at their job. However, even with the best intentions, sometimes the integrity goes out of our business or personal life. We take our attention off the details, we "go for the goodies"−a big contract, a shortcut that isn't exactly above board−and our integrity slips. If we are committed to keeping our integrity, we make the correction as soon as we see that it has slipped out. Even when a company has integrity−high quality products, good value for the price, excellent customer service, taking care of it's

employees, paying all taxes and licensing fees−keeping that integrity is an on−going process across the board, and from the top−president, CEO−to the bottom−mail room or janitor.

When coaching begins at a company, any lack of organizational integrity naturally surfaces. You begin to separate the chaff from the wheat−those committed to integrity, and those committed to something else−making a fast buck, sloughing off, shuffling papers, and collecting a paycheck. The Right Coach can help you find out what's working and what's not, and who's working and who's not. When people lacking integrity see that they will be uncovered, they usually quit before they get fired, because they know about their lack of integrity before you discover it.

A team at one company decided at a "visioning" meeting that their values should include "functioning with integrity." Someone asked, "Does 'integrity' apply to us alone or does it include customers?" "Of course, we're not going to be honest with our customers," said another. They looked at each other in silence. In their industry, vendors routinely promised customers delivery dates they knew they could not meet. The team began a three−hour dialogue without a break. When it ended they concluded, "If we're putting up integrity as a value, we need integrity in all aspects of our business." Current reality, however, presented them with a dilemma: if they changed immediately, they would be unable to match their competitors' delivery promises, and they'd be out of business. So they developed a strategic migration plan. They visited key customers and said, "This industry is based on exchanges of false promises. You know it. We know it. Nobody likes it, but we all feel stuck with it. We would like to change that. We would like to start by being honest with you." Thenceforth, every delivery date they offered those customers was realistic−and honored. Within a year, their business was growing exponentially and their profits skyrocketed. [Senge+94]

Shared ideals, shared ideas, shared goals, shared respect, a sense of integrity, a sense of quality, a sense of advocacy, a sense of caring−these are the basis of Herman Miller's covenant and value system. [DePree89]

The key to beauty is mastery. You can't skip levels. There's an entry level and each level is a pre−requisite for the next. If you think you can skip a level, it will come back to haunt you. You can fake integrity but it will come back to haunt you. You won't make it. You can't go on to the next step until you've mastered the current one. [Rike02]

Beautiful People

If I had to say what it is about our company that makes it beautiful I would have to say that it's fairness. It's the way we treat people in this company. We have a high respect for the individuals in it. It happens all the time−a contractor or a vendor will take odds with a price or schedule and I'm drawn into it. I always defend my people first. I'm not interested in winning and the other side losing but I never want my own people to lose−even if they're wrong. [Rike02]

???

You're an entrepreneur who wants to build a beautiful company. You've defined a Beautiful Purpose, started to exercise Beautiful Leadership, and established that you have No Ordinary Workplace. You want to work with people with a passion for what they do, because you know that people are your most important asset.

Many companies treat their employees as "heads." You know that's not what you want to do.

Business isn't about shuffling numbers or rearranging organizational charts or tallying the latest business school formulas. It's about people. [Brown85] Whereas a management curriculum has no place for human beings, the workplace is full of them. [deGeus97]

Control is an illusion. People never "do as they're told." People get paid so they're willing to give some control to the boss but they don't give up all control. You can't pay them enough for that. Many managers assume they have all the control−that it's their job to control everything. To manage the kind of person who forms the heart and soul of effective organizations, you have to give up control to keep control. You have to use your authority so sparingly that no one notices that it's being used. You have to create a real sense that control is not completely centralized but spread generously over the organization. Like a gifted helmsman, who knows that all use of the rudder increases drag and holds the vessel back, you have to steer with the lightest possible touch. The slack that you cut for your employees is essential to a healthy organization. [DeMarco01]

The best people working for you are like volunteers. Since they could probably find good jobs elsewhere, they choose to work for reasons less tangible than salary or position. [DePree89] As a manager you have to work with people as you find them. Your role is to create the conditions in which they will voluntarily give their best. [deGeus97]

Treat people as employees and that's what you'll get. They leave as soon as they can get fifty cents an hour more somewhere else. Treat them as co−workers and everything changes. [Dauten99]

Therefore:

Make employees part of the business. Give them a part of the operation to run. Put one employee in charge of displays or men's shirts or the check−in process. Involve them in training others in what they know.

The people you groom and train in your business stand on your shoulders. They will be more creative and more productive and more successful because they learn from you and add their own creative juices. This is markedly different from those who look for people to control. Instead, look for people you can partner with−people you can "dance" with.

Management is based on attaining predetermined objectives with and through the voluntary cooperation and effort of other people. Too many managers fall into the trap of believing that their employees are there to serve them, when in reality employees want to fulfill their own needs. Communicate to your employees that you do not want slaves. Rather, you want to work together in a mutual relationship so that everyone involved can fulfill his own needs. Effective leadership requires that your people cooperate voluntarily, not as a result of manipulative action on your part. [Brown85]

Treat employees as volunteers just as you treat customers as volunteers, because that's what they are. They volunteer the best part−their hearts and minds. [Covey89]

Give away control. Allow your employees to chart their course, while riding alongside to offer support and reassurance when needed. The shared purpose and culture that binds the company together will encourage your employees to perform, driven from within.

Try being a coach of a Little League team or Girl Scout troop. Most people think of management as control but in a volunteer situation you can't discipline a volunteer and you can't offer monetary rewards. You're forced to be a leader.

A potential client and his general contractor came to our plant. We went over their blue prints and specification for a large custom home. During their visit several of our staff came by. Our receptionist came in to give us information and make copies. Our production coordinator gave the visitors a virtual tour of the production schedule and explained how we do things. As the visitors were leaving, I introduced them to my dad, who has been with the company almost from the beginning. As we were walking out together, the general contractor turned to me and said, "You know, everybody here is so nice." It took a moment before I realized what a wonderful compliment he had paid my company. Steve Sanchez

I once had an employee who had a gift for turning returns into new sales. If someone wanted to return an item, they'd always walk out with more than they brought back. I told her I wanted her to teach everyone else. That meant she had to figure out what she was doing in order to teach it. She helped make us all better at our jobs. [Dauten99]

Why would I say we have a beautiful company? The thing that sets us apart is the way people are treated−we respect our employees and our clients. That's really the biggest thing−respect−it's about the Golden Rule−how you want to be treated. We don't treat some better than others. [Price02]

The Right Person for the Job

Word of mouth is a powerful thing. The people who seemed to work out best with our company were those who were recommended by existing employees or close associates of the company. Because they knew about the needs of our company, they acted as a reputation filter for potential employees. It was in their interest to make sure that these people have a good fit−after all, they don't want to come off as passing off bad people to you! These potential employees also judge you from what they hear about you out there in the marketspace; your good reputation will attract and your bad reputation will repel. People talk.

???

You're an entrepreneur who wants to hire Beautiful People. You have a clear idea of the person you want.

You know the right person is out there but you don't know how to find him.

Word of mouth is still the most important form of human communication. Many advertising executives believe that because of the tremendous marking efforts these days, word−of−mouth has become the only kind of persuasion most of us respond to anymore. One study showed that most job connections are made through acquaintances or weak ties, not close friends. Masters of the weak tie understand what word of mouth is. It's not me telling you about something and your telling a friend and that friend telling a friend. Word of mouth begins when somewhere along that chain, someone tells a Connector. [Gladwell00]

Resumes are the wrong way to hire. They cause both those who apply and those who hire to focus on the wrong things like degrees and certifications and number of years of experience. Hiring ought to be based on knowledge and experiences, on dreams and passions, on courage and curiosity; in other words, on things that can't be quantified and set forth on a piece of paper. [Dauten02b]

Therefore:

Advertise−but not in the usual channels. Remember, It's a Small World. Let everyone know the kind of person you want. When you finally meet a potential candidate, listen to your guts. You need people with heart. You better hire it, because you sure can't put it in.

Don't go through the haystack looking for another needle. Become a magnet to draw the needle to you. Have a clear image of the person you want, so you'll be able to describe him to others. Imagine you had a magic wand and could instantly create the person you wanted. Articulate these things in your description so everyone can understand what you're talking about. When you are clear about what you want and what you can offer, you have a better chance of meeting the potential employee's expectations.

When you finally meet a potential candidate, listen to your guts. Give some credibility to your intuition. The interviewee who makes you say, "Aha!" is the one for the job. This person may not be the best qualified on paper or have the most impressive credentials.

How do I attract beautiful people? People come to me. I never run an ad. I pray and then just talk to everyone. I have a picture in my head of the person I want. The old Jim would want to do the elaborate interview thing but the new Jim would want to ask a few appropriate questions and then take it to a quiet place and trust his intuition. [Rike02]

It isn't about hiring the person with the best credentials it's more about the attitude of that person. You could overlook a lot of things but not attitude. We have a technical person who has a lot of technical knowledge but it goes by the way when they deal with the customer. All the technical skill in the world won't help if attitude turns off a customer. Steve Sanchez

Intention. You "intend" to have the perfect staff show up. I feel there is something you have inside that makes it happen. Somehow there's a mutual attraction when you both have needs and you meet the needs of the other. Even when you hire someone you have reservations about they just hold the spot until the right person comes along. [Price02]

At Master Marble we're searching for the perfect receptionist. In the past we used the "meatloaf" approach. This was our recipe for "meatloaf receptionist."

- *Send job description to personnel staffing company with more detail than the last−necessary because we didn't get the person we wanted last time.*
- *Write clever ad for classifieds. Use current buzz words.*
- *Conduct numerous multi level interviews with every conceivable person on staff and some from other companies.*
- *Hire the person with the best match of their buzzwords to our buzzwords.*
- *Cook in the Master Marble oven for 4−6 months at medium stress. Vuwella! Meatloaf!*
- *Reconsider, reload, and repeat once a year for ten years or until sick of the loaf.*

We noticed something. We are tired of meatloaf. Now we create a new dish. Keep the job description, the clever ad with buzzwords, and add a magic wand, "If you could have anything you wanted in that position what would it be?" The person would:

- *Have the attitude that anything is possible.*
- *Be resourceful.*
- *Be flexible.*
- *Work well with our clients.*
- *Have a good sense of humor. Have them tell a joke during the interview?*
- *Get something personal from what they are doing.*
- *Of course, have all the other stuff, job history, accomplishments, blah, blah, blah.*

How do we find out if the interviewee has these qualities? Ask them on the spot for examples and to provide someone to corroborate the information. Get the answers in a fun way. We can't keep making meatloaf from the same old ingredients−wanting different outcomes but not willing to change the way we do things. Steve Sanchez

Audition

It's hard to catch everything about a person in an interview. There are so many courses, mentors, and books that can train you to ace an interview. It's like testing something in a lab: it might work in an artificially constrained environment, but the real world is much more volatile and unpredictable, so you'd better test it there too. You see a lot from a person when they're working, in the little things they say and how they respond to problems. Do they rub everyone the wrong way? Do they understand what the company's really about? Maybe the new employee just doesn't feel like it's working out for him. Neither of you can see all of this in an interview. We always had a one−month probationary period for our new employees during which they could be asked to leave. And people were asked to leave. Daniel May

???

It's hard to tell in an interview whether the person you hire will work out.

What's the hardest job in management? People. Getting the right people for the right job. That's what makes the difference between a good manager and a drone. Get the right people. Then, no matter what all else you may do wrong after that, the people will save you. That's what management is all about. People selection, task matching, motivation, team formation−the four most essential ingredients of management. All the rest is Administrivia. [DeMarco97]

Hiring is rarely based on just a job interview or referral−both sides understand that these are artificial. They want to see the other person's work, and the person at work, to see if they are talent "kin." [Dauten99]

Therefore:

When you decide to hire someone, let him work for a short time to see if you have the right fit.

You'll learn about the new hire and he will learn about you. The time frame and the tasks depend on the position. Sometimes a day is enough; sometimes it takes several months.

Consulting Co was a boutique consulting company that only hired senior consultants (8+ years experience). In partnership with universities, it created a one−year internship program for high−performing graduates. These graduates would combine their intellectual capability with real−life senior experience during this year, after which they would return to complete further studies. Over the course of this year, Consulting Co put itself into the position of intimately knowing these graduates, so that they could occasionally draw on their expertise for consultancy assignments or employment. Daniel May

Master Marble hires individuals and lets them work in the shop for a day to determine their skills. The shop foreman reviews their performance. Steve Sanchez

I was a consultant for a small software company where new hires were given a mentor and a task and a few months to evaluate the fit. This provided the employee a chance to evaluate the company and the company and the rest of the team a chance to evaluate the programmer's skills. Linda Rising

Southwest Airlines is famous for its hiring practices but no one says much about its firing practices. They consider the first six months a trial period, in which any employee can be fired at will−including union employees. [Dauten99]

Nordstrom loses nearly a quarter of its salespeople each year. The leaders of the company accept this turnover−maybe even welcome it−regarding it as a natural selection process. At Nordstrom's, they say, "Your performance is your review." Given that much of the employee's income is from commissions and that expectations for hard work are very high, the employees tend to select themselves out. [Dauten99]

Be All That They Can Be

My father was the founder of Herman Miller. In the furniture industry of the 1920s, the machines of most factories were run by pulleys from a central drive shaft, which was powered by a steam engine. The boiler for the steam engine was fueled by sawdust and other waste from the machine room−a beautiful cycle. The millwright oversaw that cycle. He was a key person. One day the millwright died. My father, a young manager at the time, thought he should visit the family. The widow asked my father if it would be all right if she read aloud some poetry. Naturally he agreed. She went into another room and came back with a bound book and for several minutes read selected pieces of beautiful poetry. When she finished, my father said how beautiful the poetry was and asked who wrote it. She replied that her husband, the millwright, was the poet. I learned from that that it is fundamental that leaders endorse a concept of persons. This begins with an understanding of the diversity of people's gifts and talents and skill. [DePree89]

???

You're trying to build a beautiful company and keep the Beautiful People you've hired.

Sometimes people are in disguise−their talents are hidden, even from themselves.

Everyone comes with certain gifts, but not the same gifts. True participation and enlightened leadership allow these gifts to be expressed in different ways and at different times. [DePree89]

In a living company, cohesion and diversity exist together. The company is clearly a unit, with a single identity; but the people and substructures within that unit show a rich variety. The substructures do not need to be uniform for the whole to keep together. On the contrary, there is value in diversity. It is the manager's duty to that organization to find the employee's strong points and help bring the most out of that employee. [deGeus97]

The Golden Rule says to "Do unto others as you would have others do unto you." While on the surface that could mean to do for them what you would like to have done for you, I think the essential meaning is to understand them deeply as individuals, the way you would want to be understood, and then to treat them in terms of that understanding. As one successful parent said about raising children, "Treat them all the same by treating them differently." [Covey89]

The more we can see people in terms of their unseen potential; the more we can use our imagination rather than our memory when interacting with them. We can refuse to label them− we can see them in new fresh ways when we're with them. We can help them become independent, fulfilled people capable of deeply satisfying, enriching, and productive relationships with others. Goethe taught, "Treat a man as he is and he will remain as he is. Treat a man as he can and should be and he will become as he can and should be." [Covey89]

My experience with people is that they generally do what you expect them to do! If you expect them to perform well, they will; conversely, if you expect them to perform poorly, they'll probably oblige. I believe that average employees who try their hardest to live up to your high expectations of them will do better than above−average people with low self−esteem. Motivate your people to draw on that untapped 90% of their ability, and their level of performance will soar! [Brown85]

If you want to be a leader, you must realize that a manager is not God. A manager does not create people in his own image. As a manager you take people as they come, the way God created them, and you learn to work with them. [deGeus97]

Therefore:

Make sure that each of your employees has a meaningful job −−one that is challenging and requires his full attention all day long.

Assess each individual in terms of experience, ability, personality type, and interest and then match each person with a particular job. Don’t assume too much. Make sure you talk to each person. Use common sense.

This enables you to think in a new way about the strengths of others. Not just "expert" others−world−class designers and people with university degrees−but all your employees. Understanding and accepting diversity means that everyone feels needed and accepted. Recognizing diversity helps connect the great variety of gifts that people bring to the work and service of the organization. Diversity allows us all to contribute in a special way, to make our special gifts part of the corporate effort. [DePree89]

Recognizing diversity helps us to understand the need we have for opportunity, equity, and identity in the workplace. Recognizing diversity gives us the chance to provide meaning, fulfillment, and purpose. These must not be relegated solely to private life any more than such things as love, beautify, and joy. [DePree89]

Make sure that these diverse gifts are involved in your company from the beginning. If you wait too long, they may become integrated into your organizational culture and you’ll find they’ll start thinking more and more like everyone else.

In each person is a range of personae. How you treat them will determine which of these personae you see−and whether you bring out the best or the worst in that person.

I met a man who hired troublemakers−people other managers were ready to give up on because they were difficult to manage. He established his department as a place for ’difficult people.’ They weren’t difficult for him because he’d set them loose instead of trying to manage them and they turned out amazing work. [Dauten99]

During an interview, I like to ask prospective employees what is the "gift" they bring? Who are they going to be? I share my vision of a beautiful company and ask, "How will you contribute?" This is a real tool! I think it gives a person a moment to consider what he’s up to, and perhaps bring up something that would not have otherwise come up. Steve Sanchez

Graceful Exit

One gifted boss tells them, "I know you've done your best, but you haven't found what you were meant to do." Then he helps them focus on their strengths and weaknesses and uses his network to help them find a new job. [Dauten99]

???

You're building a beautiful company that's No Ordinary Workplace and hiring Beautiful People.

Organizations are made up of people and people are constantly changing.

Working with employees is like dancing with them. When they decide to leave, we both know it. That's when I sit down and talk to them about it. Some employees decide to change their behavior after I called the game, and others decide to leave. [Price02]

Failures in the employee relationship occur not because the image in the initial interview was wrong or the interviewee was faking it but because some life circumstance changed: marriage, falling in love, family problems. As a result, the focus goes away from the job and the employee relationship is changed.

My father was a doctor. He had employees that had been with him forever. I think sometimes you need to let them go. He had a lot of loyalty but sometimes they abuse it. The people you have in your company are a reflection of what's in your head. They unconsciously know when things are changing and will decide on their own not to move with you. Last year I went to San Francisco to get cosmetic training, and when I came back I discovered that some of the staff were thinking of leaving−they couldn't move with the changes. [Price02]

When you first begin to define your company as founded on integrity, some employees can't align with the integrity and they have to leave. If they don't it's only a matter of time before they are asked to leave. Beauty is in the eye of the beholder. For a company to be beautiful, it must be seen that way by everyone in the organization. Over time, the goals of individuals in the company and the goals of the organization will shift and when there is a mis−alignment, the beauty goes away.

Firing can be an important management tool. You can use firing and hiring to set new standards, to demonstrate to other employees what you are looking for. People are smart. They pick up on it. Employees see what you do and think "I understand−here's the performance and work ethic that's rewarded." They see the person who left and think, "That's the performance and work ethic that's unacceptable." It's part of the process of getting people to understand what the company wants. You have to accept the need to fire people. But that doesn't mean you jump to it. If it's easy to fire someone, it's probably the wrong decision. If it's easy, you're not caring enough. Treating people with dignity sends another message: Employees want to know that if they have to leave, they'll be able to feed their families and have time to find a new job. [Dauten99]

Therefore:

Understand and help your employees understand that separation is a natural occurrence.

The Right Coach can help uncover the problem and where the breakdown occurred. Most issues can be resolved if the employer and employee are willing to talk openly about what happened to the relationship, which must have been good or the employee would not have been hired in the

first place. Try to discover what happened, where the problem began. Is there a misunderstanding, a withholding, an undelivered communication, an unfulfilled expectation? Is there out integrity on the part of the employee or did the employer promise something that was not delivered? Are there hurt feelings, or does one person feel the other one doesn't care anymore? These questions must be answered before you consider firing a beautiful employee.

Sometimes it is necessary to end a relationship, but only when both parties are clear that it is for the best for all concerned. If the relationship with the employee is built on trust, the exit should not be painful. No bridges should be burned and a win−win situation should result.

When you find The Right Person for the Job, you can begin with the Graceful Exit already in place. This transforms the entire context of the relationship. It is a promise between both parties that establishes the level of integrity for the time when it is usually at its lowest. Defining high integrity for the exit means an even higher level during the course of the relationship.

No one stays with a company forever. When someone leaves, especially valuable contributors, we worry about how we will get along without them. Start that conversation when they come in. Tell them not to be concerned about moving on. If you think of your small company as an environment for growth, then when they've outgrown that, they can move up in your company or they can leave perhaps to create their own companies. It provides a lot of freedom when people know that whatever happens, if they outgrow their jobs, that's OK. As people grow they're ready for a bigger challenge and if your company doesn't have that challenge that's OK.

Ask the potential employee to agree to consider only a Graceful Exit should the time come. Ask them to describe what this would be for them. Some of the components might be: not leave in a huff, training their replacement, or even finding a new employee. This changes the context for the exit to one of leaving on the best possible terms. Leaving everything complete, perhaps co−creating a referral for the next job.

This approach removes the fear. The company doesn't feel it owns anyone and isn't afraid when the employee thinks of leaving. The employee isn't afraid to consider leaving and the experience is complete.

Instead of making enemies, make allies with those who have been fired. If you care you attract quality people. Winners attract winners and leaders attract leaders. But sometimes people aren't performing and they've got to go. Deal with it kindly and with compassion. Help them to do better in the next job. Many fired employees are hired back. They leave and try something new and come back grateful and recharged. [Dauten99]

The expression, "I had to fire him" is the right one. When there is no other option and everybody knows it: boss, employee, coworkers−that's the right time. When it's that time, the gifted boss and great employee alliance need not end, just be suspended. [Dauten99]

One employee was moody and acting out little dramas. She was playing a game. It would go on and gain momentum unless I "call it." If you let one kid stay up late, then they all want to do it! I say that I "call the game"−we're both playing a game and we need to talk. I asked her if she was aware of what her behavior was saying and I told her that I needed to have things be different. I suggested we meet again in two weeks. She came in ten days later and quit. [Price02]

Beautiful Customers

In January 2001 I was skiing with friends in Northern Arizona. I fell and seriously injured my knee and had to see an orthopedic surgeon, Dr. Dave. We started off like old friends. He has taken great care of my rehabilitation and me through two knee surgeries. Dr. Dave became a customer, buying granite kitchen countertops for his mountain home. I worked out all the details with his wife and completed the project but Dr. Dave was never able to see the material before we installed. It was 2 months after it was completed before he saw the finished product. I had an appointment with him just as we were submitting the final invoice. When he came into the examination room he said, "I have something for you." He left the room and returned with a check and handed it to me. It was signed but the amount was blank. He said, "I'm not sure of the balance, so you fill it in." When I returned to the office I took the check around to my employees to see their response. They all understood what it said about our beautiful company. Steve Sanchez

???

You're trying to build a beautiful company. You have a Beautiful Purpose and are trying to exercise Beautiful Leadership. You've hired Beautiful People.

How do you get and keep beautiful customers?

In business, you want a vendor you can forget about−where you don't have to follow up and you know the specs are going to be right. We want customers to feel that way about us, who light us up, are fun to do business with, stretch us and make us grow, pay their bills on time, and we're proud to be associated with.

Therefore:

Be authentic and expect your customers to be authentic.

How do you do that? Make a promise that is a powerful statement, for example, I tell new clients: once you make a contract, you can forget about the rest. We will take care of all the details. Then we live up to it. This generates the relationship that creates beautiful customers. Even if you make a mistake−when the trust is high−they just say, "OK, you made a mistake." They know we will never stop until they are satisfied. I've never left a client swinging in the wind. Steve Sanchez

It happens all the time in my office. People will say, "I was afraid but now I feel OK." The biggest thing is the painless shot. People write notes. I've saved them over the past twenty years. I call this huge notebook my "Office thank you." [Price02]

One of my clients is a general contractor who builds multi−million dollar homes. We don't bid his jobs. We negotiate them. He comes up with a budget and then we figure out how we can do what he wants within his budget. Steve Sanchez

Beautiful Outsiders

Our vendors are all personal friends. They're almost more important than our patients. Kevin has been our equipment guy for 15−16 years. We buy all our supplies from him. He repairs all our equipment. We have total trust. I don't have to shop prices. I don't have to check his invoice. I don't like to do business with people I don't like−when I don't trust their integrity. I make sure they really like me and I really like them. I pay their bills on time. I take care of them and their families. No matter what, we take care of each other. We can resolve any issue. [Price02]

???

You're trying to build a beautiful company. You've defined a Beautiful Purpose and you're trying to exercise Beautiful Leadership. You hire Beautiful People to work in your company but you also have relationships with individuals and companies outside your company who are not customers.

How do you treat those your company will interact with who are not your employees or customers−vendors, partners, consultants, sub−contractors?

"Trust, but verify," is a Russian proverb, "Doveryai, no proveryai" that was often recited by President Ronald Reagan concerning nuclear weapons control during the peak of the Cold War. When asked how he could be sure that the Soviets were going to abide by the agreement, he said he would trust them−but also verify. Profound mistrust of the Soviet Union led him to demand strict measures for confirming that arms reduction had, in fact, been carried out.

In the Greek poet Hesiod's *Works and Days,* "Let the wage promised to a friend be fixed; even with your brother smile−and get a witness; for trust and mistrust, alike ruin men."

Trust is a process, not a product. It's an endless, arduous, thankless process. We only notice when something goes wrong.

Therefore:

Negotiate an agreement with outsiders, and then live it.

The agreement aligns with your Beautiful Purpose. Once you know what you want, you simply tell them and set the intention. If you know what you're looking for you can identify it when you see it. They can decide they like it or they can do business elsewhere.

You want to do business with companies that are professional, fun, pay their bills on time, stretch you and work with you as partners. If they break the agreement, let them know something is not working and ask if the problem is on your side. Ask them if they want to change the agreement, and if so, negotiate a new one. Keep in mind what you want to happen, quid pro quo−give something in return for something of equal value−tit for tat. If the relationship continues to deteriorate, realize that you are only as good as your weakest partner and take steps to ensure you deliver on your agreement to your clients. This could mean that you replace the outsider. Always go back to the agreement; this keeps it clean, simple, and professional.

Over time a relationship is built on trust with an outside partner. After working with some outsiders the need for vigilance could be reduced. There's a lack of fear. That's a component of a beautiful company. You know it's going to get done.

If you replace an outsider, you might consider giving them another chance in the future based on their performance record with other companies that were verifiable but maintain a high degree of vigilance.

In one company that developed a large piece of software with multiple teams, each team defined an interface document with teams that defined shared data and access routines. We were always careful because some teams would violate this document and try to go their own way. In some cases we had regular meetings with the rogue teams, since we didn't have a choice about working with them! Linda Rising

Chris, our computer guy, has been taking care of us for about 6−7 years. We installed a new software system−anytime you change the software you have to change the settings. Our network had some problems with the new server. Chris spent a lot of time to reconfigure it. He sent me the bill and then called and said he would reduce the price because he felt some responsibility for it. I didn't have to say anything. That's the kind of relationship you want. [Price02]

References

[Bentley] Bentley College Center for Business Ethics
http://ecampus.bentley.edu/dept/cbe/newresearch/2.html

[Brand99] Brand, S., *The Clock of the Long Now*, Basic Books, 1999.

[Brown85] Brown, W.S., *13 Fatal Errors Managers Make and How You Can Avoid Them*, Berkley Books, NY, 1985.

[Carbonara97] Carbonara, P., "Wealth and Poverty," *Fast Company*, December 1997, 60.

[Collins+94] Collins, J.C. and J.I. Porras, *Built to Last: Successful Habits of Visionary Companies*, HarperBusiness, 1994.

[Dauten99] Dauten, D., *The Gifted Boss*, William Morrow and Company, 1999.

[Dauten02a] Dauten, D., "The Best Companies are Best to Employees," *The Arizona Republic*, February 12, 2002.

[Dauten02b] Dauten, D., "Devil curses resume dependents, falsifiers," *The Arizona Republic*, Tuesday, August 13, 2002, D2.

[Davenport+98] Davenport, T.H. and L. Prusak, *Working Knowledge: How Organizations Manage What They Know*, Harvard Business School Press, 1998.

[deGeus97] deGeus, A. *The Living Company*, Harvard Business School Press, 1997.

[DeMarco97] DeMarco, T., *The Deadline*, Dorset House Publishing, 1997.

[DeMarco01] DeMarco, T., *Slack: Getting Past Burnout, Busywork, and the Myth of Total Efficiency*, Broadway Books, 2001.

[DePree89] DePree, M., *Leadership is an Art*, Dell, 1989.

[Edler95] Edler, R., "*If I Knew Then What I Know Now: CEOs and other smart executives share wisdom they wish they'd been told 25 years ago*," G. P. Putnam's Sons, 1995.

[Gilbreath93] Gilbreath, R.D., *Escape from Management Hell*, Berrett−Koehler Publishers, 1993.

[Gladwell00] Gladwell, M., *The Tipping Point*, Little, Brown and Company, 2000.

[Hammonds02] Hammods, K.H., "Handle With Care," *Fast Company*, August 2002, 102−107.

[Hefferman02] Hefferman, M., "The Female CEO," *Fast Company*, August 2002, 58−66.

[Hendrickson02] Hendrickson, E., "Managing Technical People (When You're No Techie), *STQE*, May/June 2002, 58−60.

[Jr02] *The Arizona Republic*, Friday, June 28, 2002, D2.

[Kleinfeld02] Kleinfeld, J.S., "Six Degrees of Separation: An Urban Myth?" *Psychology Today*, Mar/Apr2002. http://www.uaf.edu/northern/six_degrees.html

[Kurtzig91] Kurtzig, S.L. with T. Parker, *CEO: Building a $400 million company from the ground up*, Harvard Business School Press, 1991.

[LaBarre00] LaBarre, P., "Do You Have the Will to Lead?" *Fast Company*, March 2000, 222.

[LaBarre01] LaBarre, P., "Who's Fast 2002: Feargal Quinn," *Fast Company*, November 2001, 88−94.

[Manns+02] Manns, M.L. and L. Rising, *Patterns for Introducing Patterns (or any Innovation) into Organizations*, Addison−Wesley, in press.

[Mariott+97] Marriott, J.W., Jr. and K.A. Brown, *The Spirit to Serve: Marriott's Way*, Harper Business, 1997.

[McMahon02] McMahon, J.T., "Enron's leaders still don't get it," *Houston Chronicle*, February 1, 2002. http://www.chron.com/cs/CDA/story.hts/editorial/outlook/1236695

[Overholt02] Overholt, A., "True or False: You're Hiring the Right People," *Fast Company*, February 2002, 110−114.

[Pascale+00] Pascale, R.T., M. Millemann, and L. Gioja, *Surfing the Edge of Chaos*, Crown Business, 2000.

[Price02] Interview by Caroline King, Linda Rising, and Steve Sanchez of Ginger Price, D.D.S., July 30, 2002.

[Rike02] Interview by Caroline King, Linda Rising, and Steve Sanchez of Jim Rike, President, CalibeR Construction, Inc., June 26, 2002.

[Rothman01] Rothman, J., "Other People's Problems," *Software Development*, September 2001, Project and Process Management Column, 49−50.

[Senge+94] Senge, P. and A. Kleiner, C. Roberts, R.B. Ross, and B.J. Smith, *The Fifth Discipline Fieldbook: Strategies and Tools for Building a Learning Organization*, Doubleday, 1994.

[Senge+99] Senge, P., Kleiner, A., Roberts, C., Ross, R., Roth, G. Smith, B. *The Dance of Change: The Challenges to Sustaining Momentum in Learning Organizations*, Doubleday, 1999.

[Waldrop96] Waldrop, M.M., "Dee Hock on Management," *Fast Company*, October 1996, 79.

[Webber02] Webber, A.M., "Are All Consultants Corrupt?" *Fast Company*, May 2002, 130−134.