

CONFERENCE PROCEEDINGS OF THE SECOND, THIRD AND FOURTH NORDIC CONFERENCE ON PATTERN LANGUAGES OF PROGRAMS VIKINGPLOP

Preamble

| | |
|-------------------------|---|
| Introduction..... | 3 |
| Acknowledgements | 5 |
| Shepherding award | 7 |

VikingPLoP 2003

| | |
|---|-----|
| Pattern language for sharing Systems Engineering "best results" <i>Cecilia Haskins</i> | 11 |
| Two Simple Patterns to Support the Development of Reliable Embedded Systems <i>Chisanga Mwelwa, Michael J. Pont</i> | 17 |
| Architecturally sensitive Usability Patterns <i>Elke Folmer, Jan Bosch</i> | 27 |
| A Pattern Language for Supporting Wireless Communication Between End-Points <i>Franco Guidi-Polanco, Claudio Cubillos F., Guiseppe Menga, Samuel Penh</i> | 47 |
| A System of Patterns for Concurrent request Processing Servers <i>Bernhard Gröne, Peter Tabeling</i> | 61 |
| A Pattern Language for Standardization Work <i>Juha Pärssinen</i> | 95 |
| Factory and Disposal Methods - A Complementary and Symmetric Pair of Patterns <i>Kevlin Henney</i> | 103 |
| The Good, the Bad, and the Koyaanisqatsi - Consideration of Some Patterns for Value Objects <i>Kevlin Henney</i> | 115 |
| Factory and Disposal Methods - A Complementary and Symmetric Pair of Patterns <i>Kevlin Henney</i> | 123 |
| Transformational Patterns for the Improvement of Safety Properties in architectural Specifications <i>Lars Grunske</i> | 135 |

| | |
|--|-----|
| Two sets of Patterns about Group Communication and Dynamics | |
| <i>Ofra Homskey</i> | 153 |
| Analysis Patterns Specifications: Filling the Gaps | |
| <i>Marta Pantoquilha, Ricardo Raminhos, Joao Araujo</i> | 169 |
| VikingPLoP 2004 | |
| A Pattern Language for Participants of Standardization Work | |
| <i>Juha Pärssinen</i> | 183 |
| VikingPLoP 2005 | |
| Patterns for Documenting Frameworks - Part 1 | |
| <i>Ademar Aguiar, Gabriel David</i> | 197 |
| Patterns of Argument Passing | |
| <i>Uwe Zdun</i> | 209 |
| Business strategy patterns for sustainable knowledge based comp | |
| <i>Allan Kelly http://www.allankelly.ne</i> | 235 |
| Load Balancing and High Availability Patterns | |
| <i>Kai Wei, Antti Ylä-Jääsk</i> | 261 |
| Applied MVC Patterns | |
| <i>Sergiy Alpaev</i> | 273 |
| Patterns for ERP-Landscapes | |
| <i>Florian Humplik, Peter Leitner, Wolfgang Zuser, Thomas Grechenig</i> | 295 |
| Patterns of The Mature Customer | |
| <i>Susanne Hørby Christensen</i> | 317 |
| Privilege Separation - A security Pattern | |
| <i>Dan Forsberg</i> | 333 |
| Patterns for Software Release Versioning | |
| <i>Klaus Marquardt</i> | 339 |

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 considered ways to apply Christopher Alexander's ideas of patterns for urban planning and building architecture, to object-oriented software. They started to write object-oriented patterns and discovered an emerging desire to catalog and communicate these themes and idioms. Now, in 2007, 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, to 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.

Since then a yearly VikingPLoP conference has been held in one of the scandinavian countries. Rotating countries and conference chair duties.

2002: Helsingør, Denmark

2003: Bergen, Norway

2004: Uppsala, Sweden

2005: Helsinki, Finland

2006: Helsingør, Denmark

2007: Bergen, Norway

The Shepherding Award

Patterns are the essence of the PLoP conferences. The shepherding process improves the quality of patterns submissions. 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.

At VikingPLoP, the program committee members and the authors of accepted papers had the right to nominate a shepherd.

The Neil Harrison Shepherding Award Recipients At VikingPLoP

- 2002:** Linda Rising
- 2003:** Alan O'Callaghan
- 2004:** Cecilia Haskins
- 2005:** Klaus Marquardt
- 2006:** Andreas Rüping



Conference Chairs

- 2002:** Pavel Hruby and Kristian Elob Sørensen
- 2003:** Cecilia Haskins and Jason Baragry
- 2004:** Rebecca Rikner and Daniel May
- 2005:** Juha Parssinen and Sami Lehtonen
- 2006:** Aino Vonge Corry, Pavel Hruby and Kristian Elob Sørensen
- 2007:** Cecilia Haskins, Lars-Helge Netland and Yngve Espelid

Sponsors

- 2002:** HillsideEurope
Microsoft Business Solutions Aps
PearsonPublishing
- 2003:** HillsideEurope
Norwegian Computer Society
Microsoft Business Solutions Aps
- 2004:** HillsideEurope
- 2005:** HillsideEurope
VTT
- 2006:** HillsideEurope
- 2007:** HillsideEurope
Norwegian Computer Society

VikingPLoP 2003

Pattern language for sharing Systems Engineering “best results”

Cecilia Haskins

Norwegian School of Information Technology

cecilia.haskins@nith.no

Pattern: **Multi-disciplinary teams**

Acknowledgements:

This pattern was submitted to the second VikingPloP after thoughtful shepherding by Jim Coplien. The author wishes to thank Jim and the members of the workshop for their time and comments. There is much room for improvement for this pattern, including integration into a contextual pattern language. These shortcomings are the fault of the author and should not be attributed to others.

Abstract:

The International Council on Systems Engineering (INCOSE) has been making good progress in the compilation of a Systems Engineering Body of Knowledge (SEBOK). The objective of Systems Engineering is to assure the fully integrated development and realisation of products which meet stakeholders' expectations within cost, schedule, and risk constraints (INCOSE).

The INCOSE struggles now with an abundance of good literature augmented by over a decade of symposia proceedings and other good advice collected in their quarterly newsletter *INSIGHT* and the Journal of Systems Engineering. Specifically, INCOSE struggles with the question of how to share this wealth with the world as the organization and its members attempt to communicate the value of SE to the international community, engineers and product developers.

This is not a unique challenge. In 1977, Christopher Alexander tackled this question with the creation of “Patterns” and Pattern Languages. This author proposes that creation of a collection of Pattern Languages to document the wealth of Systems Engineering experience using sources from the SEBOK will yield positive results in the quest to share the value of Systems Engineering. This pattern has been written to illustrate the application of patterns to share a Systems Engineering practice and is intended as the first of many to introduce the reader to the underlying concepts of Systems Engineering.

SE activities incorporate both the process of engineering a system and the actual operational system itself. In 1964 a Manual written by the USAF stated that a system must be designed and tested as a complete entity and then proceeded to list the prime mission equipment, supporting equipment for testing and evaluation, training, facilities and procedures for operation and maintenance, and logistics support (1). Yet recent literature gives the impression such encompassing scope is being rediscovered.

In his foreword to the Wiley Systems Engineering series, Harold Chestnut wrote that Systems Engineering is indispensable in meeting the challenge of complexity, a sentiment that has been repeated frequently in the literature (2).

A brief summary of the Systems Engineering Pattern Languages follows. A set of pattern languages for Systems Engineering will address the following needs:

1. with global recognition of the complexity inherent in today's products and services, address how an organization copes with complexity;
2. with increased attention by consumers and customers in the quality/cost performance of products and services, address how an organization builds in quality for a reasonable end-user price; and,
3. provide a definition of a set of practices that improve responsiveness to stakeholder expectations in today's competitive environments.

The framework for the SE pattern language is grouped into 4 sections entitled Process, Product, Production and Domain. The patterns in the Domain section identify the practices that will be specific to defined applications. The other pattern areas serve to establish the core of best results in SE. Each area will benefit from mining the SEBOK and other documented pattern languages (for example, organizational).

Process

It is a basic tenet of Systems Engineering to establish an organisation that is motivated and capable of realizing the creation of products and services. This area will cover organizational structure, risk, cost and management issues.

1. ***Multi-disciplinary teams***
2. ***unknown*** — *It is envisioned that this section will draw heavily from organizational and risk pattern languages already documented.*

Product

Product patterns will focus on the set of practices used by systems engineers for defining stakeholder needs, requirements management, analysis, design and architecture. Unless otherwise specified this section will cover the creation of both products and services.

3. *unknown*

Production

In the creation of a product many systems are needed to eventually deliver the end-result. This area of the language will cover the supporting systems for a given project. The following list represents a very small subset of the eventual pattern languages (3) required to adequately cover this area.

4. *testing*
5. *manufacturing*
6. *operations*
7. *maintenance*
8. *training*
9. *customer service*
10. *logistics support*
11. *disposal*

Domain

Over time applications developers from various industries (aerospace, transportation, medicine, education) and domains (manufacturing) will contribute patterns that describe successful tailoring of the patterns in the other areas.

12. Unknown

Related patterns – to be provided

References

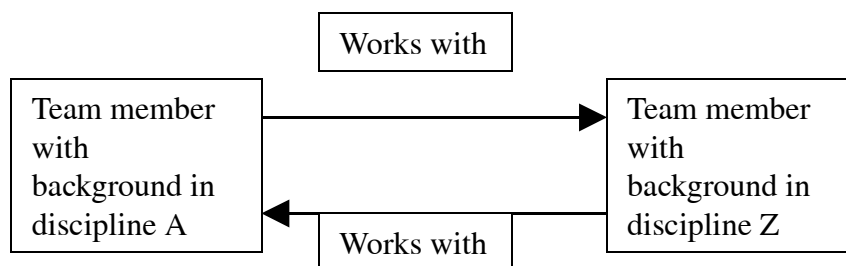
To be provided

| | |
|---------------------|--|
| Pattern Name | Multi-disciplinary teams |
| Context | Modern systems are more complex and of greater scope than was possible to realize as recently as a century ago. Today, no single person can design a modern system. Such systems are generally designed and realized through projects. Projects are generally organized into small groups with specialized talents that allow them to accomplish a specific task. Complex systems require that the results of these efforts are integrated into a final solution. This requires timely and precise communication between the groups. |
| Problem | Groups of people working on a complex modern system design must be technological experts but often lack the authority and skills to communicate effectively with other specialists in order to create a workable solution. |
| Forces | Groups of people who must work closely together are often confronted with the following challenges: Confusion about roles and responsibilities of the participants, both at the individual and group level, Confusion about responsibility for task that lie in the interfaces with other groups, Insufficient teamwork and poor inter-team communication, and Barriers inherent in organizational structures inhibit information exchange. |
| Solution | When Systems Engineering is done first, it is possible to design out unnecessary complexity. Establish multi-disciplinary teams (MDT), often implemented as integrated product teams (IPT), to perform concurrent engineering. Create a responsibility assignment matrix that maps the tasks identified in the Work Breakdown Structure onto an organization's project teaming chart to avoid duplication of efforts. |

| | |
|---------------------------------|---|
| <p>Resulting Context</p> | <p>MDTs are able to resolve project issues quickly through direct communication between team members. Such intra-team communication shortens the decision-making cycle and is more likely to result in improved decisions because the multi-disciplinary perspectives are captured early in the process. Studies have shown that group decisions are often “riskier” resulting in the potential for greater innovation.</p> <p>In a multi-disciplinary team, each member comes from a discipline with its own perspective and focuses on representing that viewpoint. An integrated team is generally multi-disciplinary but each member is expected to establish the necessary relationships with the other members and to confront the team with challenging ideas with the focus on the final result.</p> <p>As a member of the MDT, today’s informed customer becomes a partner in a structured and disciplined process that enables the team to meet stakeholder expectations.</p> |
| <p>Rationale</p> | <p>Large organizations must cope with large numbers of people and concurrent activities. Bureaucracy is common in these organizations to preserve order. However, the benefits of aligning groups of people such that they share a common focus (usually determined by technological boundaries) also can have an adverse affect on the goals of an integrated program. Specialization leads to over- or under-design at the expense of considerations about the relevance of the design to achieve a given purpose. When design tasks are assigned to specialized groups, lateral communication is either non-existent or very difficult. Too often, systems engineering is practiced as a form of crisis management to perform after-the-fact integration of separate components to make them “play together.”</p> <p>The barriers that inhibit close “day-to-day” working relationships also inhibit the transfer of essential information for decision-making. The influence of political-economic-technical-human factors plague decision-making in bureaucratic environments thereby jeopardizing the resulting design. (5)</p> <p>From Chestnut (4, p.37) “The generation of a balanced design requires that each major design decision be based on proper consideration of system variables... This necessitates the closest coordination of select personnel skilled in SE who work as a homogenous system design team.”</p> <p>From Chase (5, p.14) “Only if clear communications among the varied specialized efforts is established can there be an integrated coherent program effort, such as is required to design and develop a system composed of complex subsystems that must function effectively together as a unified entity.”</p> |

| | |
|-------------------------|---|
| | <p>Chase also says that in organizing for success it is critical to facilitate communications and “system designs are dependent upon the effective integration of multidisciplinary efforts.”</p> <p>From Chase (5, p.22) The organization of a system project should provide opportunity for all disciplinary specialists to work together continuously on a face-to-face basis and, most importantly, to acquire the systems viewpoint and understanding of the role that their specific knowledge can provide in deriving a particular system design.</p> <p>Chase advocates mapping the tasks of the milestone schedule to the WBS and identifying the lines of communication among tasks in terms of interdependencies and mutual constraints to reveal that different phases of the lifecycle call for different tasks and different personnel skills. Properly used, this allows management to acquire and properly utilize the proper combination of specialist and generalist skills. A project avoids “bureaucratization” of the design approach by streamlining the organization and integrating the various specialist backgrounds into common system-oriented task groups with loyalties directed toward the systems design effort.</p> <p>Rummler and Brache (6, p.177) refer to cross-functional processes as the <i>horizontal system</i> within an organization. Management looking for performance improvements is advised to focus on the effectiveness and efficiency of the horizontal organization. This approach recognizes individuals and groups who understand the “big picture” and the business of other functional areas with which they need to collaborate. Interactions focus on win-win decision-making.</p> |
| Related Patterns | <p>Many patterns are related to this one and will be identified and included in later versions of this pattern. Eventually both CE and IPT/IPPD pattern languages must be written.</p> |
| Known Uses | <p>J. Parker, 1989, “<i>Peacekeeper IFSS – A TQM success story</i>,” National TQM Symposium 1989, DC: AIAA. Parker writes of Martin Marietta’s use of “tiger teams” in crisis situations. These are cross-functional teams that dropped traditional differences, focused on the task at hand and were able to achieve remarkable innovations very quickly. The challenge for MM was to use this approach in a non-crisis situation. Parker is very clear that a major contributor to the success of the project was the creation of opportunities for people from different disciplines to work together face-to-face to achieve dramatic reductions in cycle time.</p> |

| | |
|---------------------------------------|--|
| Relevant references from SEBOK | <ol style="list-style-type: none"> 1. Martin, James N. 2000. Systems Engineering Guidebook: a process for developing systems and products. CRC Press LLC. 2. Forsberg, Kevin et al. 2000. Visualizing Project Management: A model for business and technical success (2.ed). John Wiley & Sons, Inc. 3. Blancard, Benjamin S. and Wolter J. Fabrycky. 1997. Systems Engineering and Analysis, 3.ed. NJ: Prentice Hall. 4. Chestnut, Harold. 1967. Systems Engineering Methods. John Wiley & Sons, Inc. 5. Chase, Wilton P. 1974. Management of Systems Engineering. John Wiley & Sons, Inc. 6. Rummier, Geary A. and Alan P. Brache. 1995. Improving Performance: How to manage the white space on the organization chart. San Francisco, CA: Jossey-Bass Inc. |
| Definitions of terms: | <p>Concurrent Engineering – the concept that all stakeholders need to be considered throughout the project lifecycle in order to produce the best product. CE promotes simultaneous development of both product and process. (2)</p> <p>IPPD – Integrated Product and Process Development – a management technique that simultaneously integrates all essential acquisition activities through the use of multidisciplinary teams to optimize the design, manufacturing and supportability processes. (3)</p> |
| Diagram: | Below |



Two Simple Patterns to Support the Development of Reliable Embedded Systems

Chisanga Mwelwa¹ and Michael J. Pont {cm55, M.Pont}@le.ac.uk

*Embedded Systems Laboratory, Department of Engineering, University of Leicester,
University Road, LEICESTER LE1 7RH, UK*

<http://www.le.ac.uk/eg/embedded/>

Introduction

As the title suggests, this paper is concerned with the development of software for embedded systems. Typical application areas for this type of software range from passenger cars and aircraft through to common domestic equipment, such as washing machines and microwave ovens.

We have previously described a “language” consisting of more than eighty patterns, which will be referred to in this paper as the “PRES Collection²” (see Appendix A). This language is intended to support the development of reliable embedded systems using low-cost embedded hardware with severe memory constraints. Typical implementations will employ embedded microcontrollers with a few kilobytes of available RAM.

Over the last few years, we have had the chance to observe many people use this collection when developing a range of different systems: these observations have included industrial projects and various university research projects. In this paper, we present two patterns that have resulted from these observations: HEARTBEAT LED and ERROR LED.

¹ Primary contact

² Our original patterns focused on “time-triggered” designs and were known as the “PTTES collection” (after the original book title: “Patterns for Time-Triggered Embedded Systems”). Since then, this collection has expanded and has subsequently been revised, and – while the focus remains on reliable design – not all of the patterns are time-triggered. The collection has therefore been renamed the “PRES collection” (Patterns for Reliable Embedded Systems).

The two patterns are related. HEARTBEAT LED provides a simple, low-cost mechanism for providing feedback on the overall health of your system: if the LED is flashing, the core of the system is running correctly. ERROR LED goes one step further and provides a mechanism for error reporting.

Format

The other patterns referred to in this paper are from the PRES collection and are presented in a distinctive style e.g. CO-OPERATIVE SCHEDULER. See Appendix A for a complete list of these patterns.

Acknowledgements

The authors are grateful to Alan O' Callaghan (our Shepherd at Viking PLoP 2003) for comments and suggestions on the first drafts of this paper. We are also grateful to the participants in our workshop at Viking PLoP (Neil Harrison, Klaus Marquardt, Bernhard Grone and Peter Tabeling) who all provided further useful comments.

Copyright

Copyright © 2003 Chisanga Mwelwa and Michael J. Pont. Permission is granted to copy this paper for the purposes of Viking PLoP 2003. All other rights are reserved.

HEARTBEAT LED

Context

- You are developing (or maintaining) an embedded application based on a microcontroller or microprocessor.
- You are programming in C (or a similar language).
- Your application has an architecture based on some form of scheduler.

Problem

How can you tell, at a glance, if your system is “alive”?

Design constraints

Many embedded systems have little or no user interface. There is not generally a screen on which you can display error messages or warnings to the user.

If you are working on a system prototype, or performing maintenance in the field, how can you tell that the system is “alive” – that it has power and (at least) the scheduler is running?

You could, of course, hook up a debugging link (e.g. a JTAG link), or a simpler serial link (based on RS-232), but this takes time and including suitable ports on your production system may not be practical or cost effective. Often a very simple, low-cost solution is required.

Solution

Every time we implement an embedded system, the first task we include is one that flashes a “heartbeat” LED. Wherever possible, this LED stays with the system, right into production.

We tend to use a 50% duty cycle and a frequency of 0.5 Hz (that is, the LED runs continuously, on for one second, off for one second, and so on) but this is – of course – up to you.

Use of this simple technique provides the following key benefit:

- The development team, the maintenance team and, where appropriate, the users, can tell at a glance that the system has power, and that the scheduler is operating normally.

In addition, during development, there are two less significant (but still useful) side benefits:

- After a little practice, the developer can tell “intuitively” - by watching the LED - whether the scheduler is running at the correct rate: if it is not, it may be that the timers have not been initialised correctly, or that an incorrect crystal frequency has been assumed.
- By adding the “Heartbeat” task to the scheduler array *after* all other tasks have been included, the developer can tell immediately if the task array is large enough to match the needs of the application (if the array is not large enough, the LED will never flash).

Implementation

Numerous possible implementations are possible. We give an example of one possible version at the end of this pattern.

Reliability and safety implications

Use of this simple technique may help to improve system reliability since it provides those developing the system with an indication of its health throughout the development lifecycle.

Hardware requirements

HEARTBEAT LED has minimal hardware requirements. The only requirements are a port pin connected to an appropriate LED (with a matching resistor if required).

Cost implications

As noted above, the hardware requirements are very limited. The time taken to implement this pattern is also likely to be minimal. Overall, the costs are very low.

Maintenance

HEARTBEAT LED gives a developer an indication of a systems “health” during its maintenance.

Portability

Highly portable – can be implemented on a wide range of hardware platforms.

Related patterns and alternative solutions

See NAKED LED³ for hardware details.

Overall strengths and weaknesses

- ☺ HEARTBEAT LED provides a simple, low-cost way of determining whether your system is “alive”.
- ☹ Uses a port pin and associated LED hardware.

Example: A “Heartbeat LED” task for an 8051 microcontroller

```
/*-----*-  
    Heartbeat_LED.C  
-----  
    Simple 'Heartbeat LED' task for an Infineon C515C microcontroller.  
    If everything is OK, flashes at 0.5 Hz  
-----*/
```

³ Page 254 in M. J. Pont (2001), *Patterns for Time-Triggered Embedded Systems*, Addison-Wesley

```
#include "Main.H"
#include "Port.H"
#include "Heartbeat_LED.H"

// ----- Private variable definitions -----

static bit Heartbeat_led_state_G;

/*-----*--

HEARTBEAT_LED_Init()
Prepare for HEARTBEAT_Update() task.

--*-----*/
void HEARTBEAT_LED_Init(void)
{
    Heartbeat_led_state_G = 0;
}

/*-----*--

HEARTBEAT_LED_Update()

Flashes an LED on a specified port pin.

Must schedule at twice the required flash rate: thus, for 0.5 Hz
flash (on for 1 second, off for 1 second) must schedule at 1 Hz.

--*-----*/
void HEARTBEAT_LED_Update(void)
{
    // Change the LED from OFF to ON (or vice versa)
    if (Heartbeat_led_state_G == 1)
    {
        Heartbeat_led_state_G = 0;
        Heartbeat_led_pin = 0;
    }
    else
    {
        Heartbeat_led_state_G = 1;
        Heartbeat_led_pin = 1;
    }
}

/*-----*--
---- END OF FILE -----
--*-----*/
```

Listing 1: The source file defining the task responsible for the flashing LED

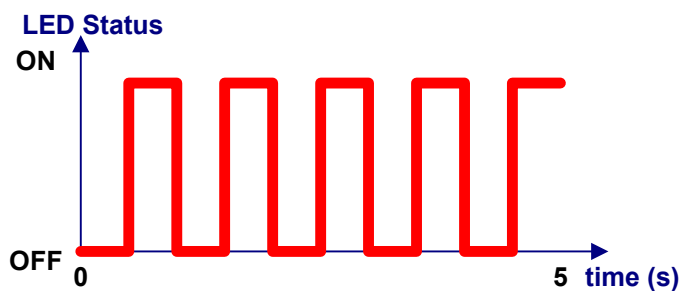


Figure 1: Graph depicting flash rate of an implemented HEARTBEAT LED task scheduled to run every 1 second

ERROR LED

Context

- You have implemented HEARTBEAT LED (see HEARTBEAT LED).

And

- You now require a means of reporting errors.

Problem

If your embedded system is not working correctly, how can you tell what is wrong?

Design constraints

See HEARTBEAT LED for the design constraints.

HEARTBEAT LED can provide a very cost-effective way of telling whether your system is “alive”. If the system is functioning, but has detected some errors, a HEARTBEAT LED may not be of great help.

How can you report errors, without significantly increasing the system (or development) costs?

Solution

To implement ERROR LED a single LED is used to report error codes to the developer or (if appropriate) the user. In most cases, we like to base the Error LED on a Heartbeat LED so that, if there are no errors, we see the usual (comforting) 0.5 Hz signal. If there is a problem, the display changes, and – by observing the different pulse rates – we can often identify the cause.

Implementation

There are many possible ways of implementing ERROR LED.

We use a (global) error variable, and maintain a list of error codes (in Main.H). In the event of an error, we adjust the output of the ERROR LED accordingly.

We give an example of a suitable implementation at the end of this pattern.

Reliability and safety implications

Most forms of error reporting – like ERROR LED – provide a means of improving system reliability.

Hardware requirements

See HEARTBEAT LED hardware requirements.

Cost implications

Implementing a basic implementation of ERROR LED will cost you little more than implementing HEARTBEAT LED. However, it takes time to include error reporting in your program code, and this may add to the development costs.

Maintenance

ERROR LED can be very valuable during system maintenance as it can be used to debug reported bugs.

Portability

Highly portable – can be implemented on a wide range of hardware platforms.

Related patterns and alternative solutions

See HEARTBEAT LED for the related patterns.

As an alternative solution one could easily substitute a buzzer for the LED, and thereby draw the attention of developers (or users) to errors using various sounds or different pulse frequencies.

Overall strengths and weaknesses

- ☺ ERROR LED provides a low-cost, non-invasive, means of error reporting.
- ☹ Uses a port pin and associated LED hardware.
- ☹ Adding error reporting takes time and hence may increase development costs.

Example: An “Error LED” task for an ARM microcontroller

```
/*-----*-
Error_LED.C
-----*

Simple 'Error LED' task for a Philips LPC2106
ARM microcontroller.

If everything is OK, flashes at 0.5 Hz

If there is an error code active, this is displayed.

-----*/
#include "Main.H"
#include "Port.H"

#include "Error_LED.H"

// see Scheduler for definition
extern int Error_code_G;

/*-----*-

ERROR_LED_Init()

Prepare for ERROR_LED_Update() function.

-----*/
void ERROR_LED_Init(void)
{
    // Set up Heartbeat_pin as GPIO
    PINSEL0 &= ~Heartbeat_pin;
```

```

    // Set Heartbeat_pin to output mode
    IODIR |= Heartbeat_pin;
}

/*-----*/

ERROR_LED_Update()

Flashes at 0.5 Hz if error code is zero.

Otherwise, displays error code.

Must schedule every second (soft deadline).

-----*/
void ERROR_LED_Update(void)
{
    static int LED_state = 0;
    static int Error_state = 0;

    if (Error_code_G == 0)
    {
        // No errors recorded
        // - just flash at 0.5 Hz

        // Change the LED from OFF to ON (or vice versa)
        if (LED_state == 1)
        {
            LED_state = 0;
            IOCLR = Error_pin; // Set to 0
        }
        else
        {
            LED_state = 1;
            IOSET = Error_pin; // Set to 1
        }
        return;
    }

    // If we are here, there is an error code ...
    Error_state++;

    if (Error_state < Error_code_G*2)
    {
        LED_state = 0;
        IOCLR = Error_pin; // Set to 0
    }
    else
    {
        if (Error_state < Error_code_G*4)
        {
            // Change the LED from OFF to ON (or vice versa)
            if (LED_state == 1)
            {
                LED_state = 0;
                IOCLR = Error_pin; // Set to 0
            }
            else
            {
                LED_state = 1;
                IOSET = Error_pin; // Set to 1
            }
        }
        else
        {
            Error_state = 0;
        }
    }
}

/*-----*/
---- END OF FILE -----
/*-----*/

```

Listing 2: The source file defining the task responsible for the flashing error LED

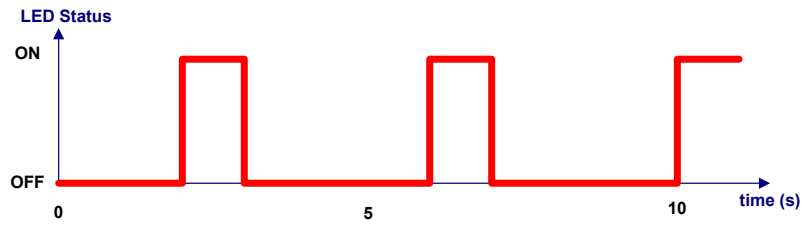


Figure 2: Graph depicting flash rate of implemented ERROR LED when an error has been flagged for an error code set to 1 (compare this flash rate with that of HEARTBEAT LED in **Figure 1**)

Appendix A: The PRES Collection

A complete list of the current patterns in the PRES collection is given in Table 1.

The present version of this collection consists of 71 patterns (see: M. J. Pont (2001), *"Patterns for Time-Triggered Embedded Systems"*, Addison-Wesley) plus a further 7 patterns from Pont and Ong (2002). Please note that the 2002 patterns are identified thus [VP]. Please also note that the later patterns, together, form a replacement for HARDWARE WATCHDOG (presented in *"Patterns for Time-Triggered Embedded Systems"*): HARDWARE WATCHDOG is therefore not listed in this table.

Table 1: The "PRES Collection"

| | | |
|--------------------------------------|--------------------------------------|-------------------------------|
| 255-TICK SCHEDULER | 3-LEVEL PWM | A-A FILTER |
| ADC PRE-AMP | BJT DRIVER | CERAMIC OSCILLATOR |
| CO-OPERATIVE SCHEDULER | CRYSTAL OSCILLATOR | CURRENT SENSOR |
| DAC DRIVER | DAC OUTPUT | DAC SMOOTHER |
| DATA UNION | DOMINO TASK | EMR DRIVER |
| EXTENDED 8051 | FAIL-SILENT RECOVERY [VP] | HARDWARE DELAY |
| HARDWARE PRM | HARDWARE PULSE-COUNT | HARDWARE PWM |
| HARDWARE TIMEOUT | HYBRID SCHEDULER | I ² C PERIPHERAL |
| IC BUFFER | IC DRIVER | KEYPAD INTERFACE |
| LCD CHARACTER PANEL | LIMP-HOME RECOVERY [VP] | LONG TASK |
| LOOP TIMEOUT | MOSFET DRIVER | MULTI-STAGE TASK |
| MULTI-STATE SWITCH | MULTI-STATE TASK | Mx LED DISPLAY |
| NAKED LED | NAKED LOAD | OFF-CHIP CODE MEMORY |
| OFF-CHIP DATA MEMORY | ON-CHIP MEMORY | ONE-SHOT ADC |
| ONE-TASK SCHEDULER | ONE-YEAR SCHEDULER | ON-OFF SWITCH |
| OSCILLATOR WATCHDOG [VP] | PC LINK (RS232) | PID CONTROLLER |
| PORT HEADER | PORT I/O | PROGRAM-FLOW WATCHDOG [VP] |
| PROJECT HEADER | PWM SMOOTHER | RC RESET |
| RESET RECOVERY [VP] | ROBUST RESET | SCC SCHEDULER |
| SCHEDULER WATCHDOG [VP] | SCI SCHEDULER (DATA) | SCI SCHEDULER (TICK) |
| SCU SCHEDULER (LOCAL) | SCU SCHEDULER (RS-232) | SCU SCHEDULER (RS-485) |
| SEQUENTIAL ADC | SMALL 8051 | SOFTWARE DELAY |
| SOFTWARE PRM | SOFTWARE PULSE-COUNT | SOFTWARE PWM |
| SPI PERIPHERAL | SSR DRIVER (AC) | SSR DRIVER (DC) |
| STABLE SCHEDULER | STANDARD 8051 | SUPER LOOP |
| SWITCH INTERFACE (HARDWARE) | SWITCH INTERFACE (SOFTWARE) | WATCHDOG RECOVERY [VP] |

Architecturally Sensitive Usability Patterns

Eelke Folmer and Jan Bosch

Department of Mathematics and Computing Science
University of Groningen, PO Box 800, 9700 AV the Netherlands
mail@eelke.com, Jan.Bosch@cs.rug.nl
<http://www.rug.nl/informatica/search>

Abstract

The work presented in this paper is motivated by the increasing realization in the software engineering community of the importance of software architecture for fulfilling quality requirements. Practice shows that for current software systems, most usability issues are still only detected during testing and deployment. Some changes that affect usability, for instance changes to the appearance of a system's user interface, may easily be made late in the development process without incurring too great a cost. Changes that relate to the interactions that take place between the system and the user such as, for example, usability patterns, are likely to require a much greater degree of modification. The reason for this shortcoming is that the software architecture of a system restricts certain patterns from being implemented after implementation. Several of these usability patterns are "architecture-sensitive", in the sense that such modifications are costly to implement afterwards due through their structural impact on the system. Our research has argued the importance of the relation between usability and software architecture. Software engineers and usability engineers should be aware of the importance of this relation. One of the results of this research is a collection of usability patterns. The contribution of this paper is that it has tried to capture and describe several usability patterns that may have a positive effect on the level of usability but that are difficult to retrofit into applications because these typically require architectural support. Our collection of patterns can be used during architectural design to determine if the architecture needs to be modified to support such patterns. The usability patterns identified so far can be used as requirements during architectural design

1. Keywords

Usability, software architecture, patterns.

Introduction

In recent years, usability has been increasingly recognized as an important consideration during software development. Issues such as whether a product is easy to learn, to use or whether it is responsive to the user and whether the user can efficiently complete tasks using it, may greatly affect a product's acceptance and success in the marketplace. Not only does the software need to implement its functionality, it must also satisfy its usability requirements in order to be accepted by its users. However, how software systems can be designed so they are usable is still a topic of debate and currently there is much ongoing research activity around this question.

A problem with many of today's software systems is that they do not meet their quality requirements very well, many well-known software products still suffer from poor usability. Lack of usability may have several causes. One hypothesis is that it results from the subjective perceptions of designers, whose view on the system is generally different from that of an end user [Berkun, 2002].

Another reason for poor usability may be the high costs associated with fixing usability issues during the later stages of development. Studies of software engineering projects [Nielsen, 1993], [Lederer and Prasad, 1992] reveal that organizations spend a relatively large amount of time and money on fixing usability problems. 80% of software life-cycle costs occur during the maintenance phase, most maintenance costs are associated with "unmet or unforeseen" user requirements and other usability problems [Pressman, 1992]. These figures show that a large amount of maintenance costs are spent on dealing with usability issues such as frequent requests for interface changes by users, implementing overlooked tasks and so on [Lederer and Prasad, 1992]. These high costs prevent developers from making the necessary adjustments for meeting all the usability requirements.

We believe these high costs are due to the fact that many of the necessary adjustments require changes to the system that cannot be easily accommodated by the software architecture. Sometimes it is very difficult to apply certain usability improving design solutions *after* the majority of a system has been implemented because some of these design solutions are 'architecture-sensitive', for example:

- Some changes that improve usability, for instance changes to the appearance of a system's user interface, may easily be made late in the development process without incurring too great a cost.

- Changes that relate to the interactions between the system and the user; for example adding undo or cancel to a task or system wide changes that improve usability such as providing a consistent interface require a much greater degree of modification.

Restructuring the system at a late stage will typically be extremely and possibly prohibitively, expensive. The further in back in the development process the designers go to make a change, the more it will cost [Brooks, 1995] to implement such changes. The reason for this is that such changes typically require changes to the software architecture, i.e. “the fundamental organization of a system embodied in its components, their relationships to each other and to the environment and the principles guiding its design and evolution” [IEEE, 1998]

Within the software engineering community, it is generally understood that to a large extent, the quality attributes (e.g. performance or maintainability) of a software system are determined by its software architecture [Bosch, 2000]

The work presented in this paper is motivated by the fact that this also applies to usability. In years of research, the HCI community has invented many clever solutions such as usability patterns to all sorts of usability design problems. However, many of these solutions are architecture-sensitive and are hard to retrofit in an existing software architecture. Consequently, it is often hard to address usability problems in already implemented systems.

To address the issue, the HCI community has been equally innovative in development methodology and nowadays such techniques as prototyping and rapid application development that depend on involving end-users early in the development are commonly used for the development of new systems. While these techniques can be very effective in addressing some common usability issues, they do not really solve the retrofit problem. Even using such techniques, it is still possible that usability problems will surface after the system has been implemented, that require major changes in the software architecture.

Summarizing, in our opinion poor usability and high development costs may be due to the following problems:

- The high costs associated with fixing certain usability issues during the later stages of development, prevent developers from making the necessary adjustments for meeting all the usability requirements.
- Usability is still often associated with interface design [Berkun, 2002]. Interface design is often performed during the last stages of software development. With this approach, we run the risk that if the interface is designed last; many assumptions are built into the design of the architecture that unknowingly affect the interface design, which makes systems less usable.

These two problems prompted us and the STATUS¹ (SoFTware Architecture That supports USability) project that is funding our research, to closely investigate the relationship between usability and software architecture to gain a better understanding of how the architecture may restrict usability.

The contribution of this paper is an integrated set of architecturally sensitive usability patterns, that, in most cases, have a positive effect on the level of usability but that are difficult to retro-fit into applications because these design solutions may require architectural support. For each of these architecturally sensitive usability patterns we have analyzed the usability effect and the potential architectural implications.

The remainder of this paper is organized as follows. The next section discusses the relationship between usability and software architecture. Section 3 presents the architecturally sensitive usability patterns we have identified. Finally we discuss related work in section 4 and conclude in section 5.

2. Architecturally sensitive usability patterns

One of the products of the research into the relationship between software architecture and usability is the concept of an architecturally sensitive usability pattern. We determined that the implementation of a usability pattern is a modification that may solve a specific usability problem in a specific context, but which may be very hard to implement afterwards because such a pattern may have architectural implications.

We define the term “architecturally sensitive usability pattern” to refer to a technique or mechanism that should be applied to the design of the architecture of a software system in order to address a need identified by a usability property at the requirements stage (or an iteration thereof).

The purpose of identifying and defining architecturally sensitive usability patterns is to capture design experience to inform architectural design and hence avoid the retrofit problem.

There are many different types of patterns. In the context of this paper, we use the term pattern in a similar fashion as [Buschmann et al, 1996]: “patterns document existing, well-proven design experience”. With our set of patterns, we have

¹ STATUS is an ESPRIT project (IST-2001-32298) financed by the European Commission in its Information Society Technologies Program. The partners are Information Highway Group (IHG), Universidad Politecnica de Madrid (UPM), University of Groningen (RUG), Imperial College of Science, Technology and Medicine (ICSTM), LOGICDIS S.A.

concentrated on capturing the architectural considerations that must be taken into account when deciding to implement a usability pattern.

Our architecturally sensitive usability patterns have been derived from two sources:

- Internal case studies at the industrial partners in the STATUS project.
- Existing usability pattern collections [Brighton, 1998], [Common ground, 1999], [Welie, 2003], [PoInter, 2003].

Only those patterns are selected or defined that to our analysis require architectural support. We have merely annotated existing usability patterns for their architectural sensitiveness.

2.1 Pattern format

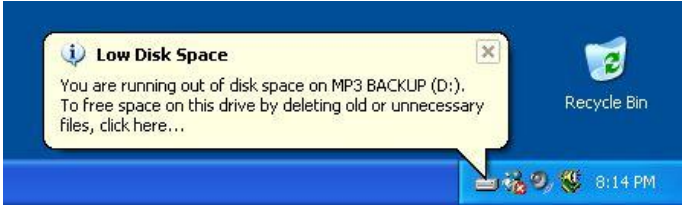
As identified by [Granlund et al, 2001] patterns are an effective way of capturing and transferring knowledge due to their consistent format and readability. To describe our patterns we use the following format:

- **Name:** Whenever possible we use the names of existing patterns. However, some patterns are known under different names and some patterns are not recognized in usability pattern literature.
- **Usability context:** A situation giving rise to a usability problem; the context extends the plain problem-solutions dichotomy by describing situations in which the problems occur. This is similar to the context used in the patterns defined in [Buschmann et al, 1996].
- **Intent:** A short statement that answers the following questions: what does the pattern do and what are its *rationale* and *intent* [Gamma et al 1995].
- **Architectural implications:** It may be possible to use a number of different methods to implement the solution presented in each usability pattern. Some of our architecturally sensitive usability patterns such as undo (table 5.9) may be implemented by a design pattern. For example, the Memento pattern should be used whenever the internal state of an object may need to be restored at a later time [Gamma et al 1995] Alternatively, an architectural pattern may be used. For example providing multiple views (table 2.8) by using a model view controller pattern [Buschmann et al, 1996]. Our patterns do not specify implementation details in terms of classes and objects. However, we are contemplating a case study to analyze how companies implement the patterns we discuss in this paper. Furthermore, we present which architectural considerations must be taken into account to implement the pattern. In the case of the wizard example there may need to be a provision in the architecture for a wizard component, which can be connected to other relevant components, the one triggering the operation and the one receiving the data gathered by the wizard. This leads to architectural decisions about the way that operations are managed.
- **Relationship with usability:** For each pattern we state which usability attributes are affected by it. A comprehensive survey [Folmer and Bosch, 2002] revealed that different researchers have different definitions for the term usability attribute, but the generally accepted meaning is that a usability attribute is a precise and measurable component of the abstract concept that is usability. After an extensive survey of the work of various authors, the following set of usability attributes has been identified for which the software systems in our work are assessed:
 - Learnability - how quickly and easily users can begin to do productive work with a system that is new to them, combined with the ease of remembering the way a system must be operated.
 - Efficiency of use - the number of tasks per unit time that the user can perform using the system.
 - Reliability in use - this refers to the error rate in using the system and the time it takes to recover from errors.
 - Satisfaction - the subjective opinions of users of the system.
- **Examples/known uses:** Similar to patterns described in [Gamma et al 1995] and [Buschmann et al, 1996] we present three known uses of the pattern in current software (not necessarily implemented in an architecture-sensitive way).

Our pattern format is not intended to be exhaustive. We intend to add to the collection in future work and actively engage in discussions with the usability and software engineering communities through e.g. workshops and our website². Future work on this project will lead to the expansion and reworking of the set of patterns presented here. This includes work to fill out the elements of each pattern to include more of the sections, which traditionally make up a pattern description, for instance what the pros and cons of using each pattern may be, forces that lead to the use of the pattern, aliases etc.

² www.designforquality.com

2.2 System Feedback

| | |
|-------------------------------------|---|
| Usability context: | Situations where the user performs an action that may unintentionally lead to a problem [Welie, 2003]. |
| Intent: | Communicate changes in the system to the user. |
| Architectural implications: | To support the provision of alerts to the user, there may need to be a component that monitors the behavior of the system and sends messages to an output device. Furthermore, some form of asynchronous messaging (e.g. events) support may be needed to respond to events in other architecture components. [Buschmann et al, 1996] suggests several architectural styles to implement asynchronous messaging (e.g. the blackboard style). |
| Relationship with Usability: | Informing the user about effects of actions that occur in the system raises user <i>satisfaction</i> , as users learn what is going on. On the other hand, satisfaction may also be affected by the decreased system performance due to alert processing. User <i>efficiency</i> may increase, because they are alerted about given situations and do not have to waste time checking the system state under these circumstances. |
| Examples: | <ul style="list-style-type: none"> • If a new email arrives, the user may be alerted by means of an aural or visual cue. • If a user makes a request to a web server that is currently off line, they will be presented with a popup window telling them that the server is not responding • If a user is running out of disk space, windows XP will alert the user with a popup box in the system tray.  <p style="text-align: center;">Figure 1: Windows XP low disk space alert</p> |

2.3 Actions for Multiple Objects


| | |
|-------------------------------------|---|
| Usability context: | Actions need to be performed on objects, and users are likely to want to perform these actions on two or more objects at one time [Tidwell 1998]. |
| Intent: | Provide a mechanism that allows the user to customize or aggregate actions |
| Architectural implications: | A provision needs to be made in the architecture for objects to be grouped into composites, or for it to be possible to iterate over a set of objects performing the same action for each. |
| Relationship with Usability: | Providing a mechanism that allows the user to customize or aggregate his actions increases user's control over the system. Providing the ability to perform the same action on a number of objects at once reduces the time that it will take the user to complete a task, as the system should be much faster at repeating actions than the human. The number of clicks (or equivalent actions) that the user has to make to complete the task is reduced. |
| Examples: | <ul style="list-style-type: none"> • In a vector based graphics package such as Corel Draw, it is possible to select multiple graphics objects and perform the same action (e.g. change color) on all of them at the same time. • Copying several files from one place to another. • Outlook allows the selection of different received emails and forward them all at once.  |

Figure 2: Actions on multiple objects in word

2.4 Cancel


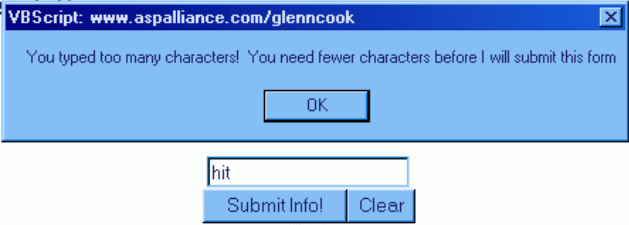
| | |
|-------------------------------------|---|
| Usability context: | The user invokes an operation, then no longer wants the operation to be performed [Bass et al, 2001] |
| Intent: | Allow the user to cancel a command that has been issued but not yet completed, to prevent reaching an error state. |
| Architectural implications: | There needs to be provision in the architecture for the component(s) monitoring the user input to run independently from and concurrently with the components that carry out the processing of actions. The components processing actions need to be able to be interrupted and the consequences of the actions may need to be rolled back. |
| Relationship with Usability: | Being able to cancel commands helps with error management. If the user realises that they have done something wrong then the user can interrupt and cancel an action before the reaching the error state. It also gives the user the feeling that it is in control of the interaction. |
| Examples: | <ul style="list-style-type: none"> In most web browsers, if the user types a URL incorrectly, and the web browser spends a long time searching for a page that does not exist, the user can cancel the action by pressing the “stop” button before the browser presents the user with a “404” page, or a dialog saying that they server could not be found. When copying files with windows explorer the user is able to press the cancel button to abort the file copy process Norton antivirus allows the user to interrupt or cancel the virus scanning process  |

Figure 3: Cancel operation in windows commander

2.5 Data Validation

| | |
|-------------------------------------|--|
| Usability context: | <p>The user needs to supply the application with data, but may be unfamiliar with which data is required or what syntax should be used. [Welie and Trætteberg, 2000]</p> <p>Users have to input data where errors are likely to occur.</p> |
| Intent: | Verify whether (multiple) items of data in a form or field have been entered correctly. |
| Architectural implications: | To ensure that the integrity of the data stored in the system is maintained, a mechanism is needed to validate both the data entered by the user and the processed data. Solutions that may be employed include the use of XML and XML schemas. Furthermore, a data integrity layer consisting of business-objects may be implemented to shield application code from the underlying database. Finally, there may be some client or server components that verify the data entered by users. |
| Relationship with Usability: | This pattern relates to a provision for the management of errors. The application of this pattern reduces the number of errors, increasing <i>reliability</i> and user <i>efficiency</i> . |

| | |
|------------------|---|
| Examples: | <ul style="list-style-type: none"> • This pattern is often employed in forms on websites where the user has to enter a number of different data items, for example, when registering for a new service, or buying something. • Large content management systems often use XML to define objects. Some WYSIWYG tools that allow the user to edit these objects use the XML definition (DTD or schema) to prevent users from entering invalid data. • Use of a data integrity layer in multi tiered applications to shield user interface code from database.  <p style="text-align: center;">Figure 4: Form Validation</p> |
|------------------|---|

2.6 History logging

| | |
|-------------------------------------|---|
| Usability context: | <ul style="list-style-type: none"> • How can the software help save the user time and effort? [Tidwell 1998] • How can the artifact support the user's need to navigate through it in ways not directly supported by the artifact's structure? [Tidwell 1998] • The user performs a sequence of actions with the software, or navigates through it. [Tidwell 1998] |
| Intent: | Record a log of the actions of the user (and possibly the system) to be able to look back over what was done. |
| Architectural implications: | In order to implement this, a repository must be provided where information about actions can be stored. Consideration should be given to how long the data is required. Actions must be represented in a suitable way for recording in the log. Additionally, such features may have some privacy/security implications. |
| Relationship with Usability: | Providing a log helps the user to see what went wrong if an error occurs and may help the user to correct that error. Being able to refer to actions that were carried out may help with "recognition rather than recall". The provision of this pattern improves <i>reliability</i> in use, as it provides the user with information on how to correct errors. It also has a positive effect on <i>learnability</i> , as the user learns how to work with the system |

- Examples:

- Web browsers create a history file listing all the websites that the user has visited. Most web browsers also include functionality for purging this data.
- Windows XP keeps track of recently accessed documents.
- Automatic Form completion in Mozilla and Internet Explorer based upon previously inserted information.

The screenshot shows the Internet Explorer interface. The address bar displays "http://www.google.com". The left sidebar contains "History" and "Favorites". The main window area lists several websites under the heading "Today":

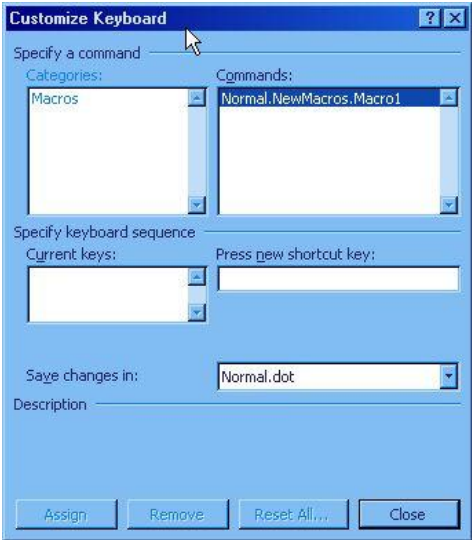
- 01320000100
- 0679243100
- emissiden [www.emissiden.com]
- esplance [www.esplance.com]
- eudigale [www.dogag.com]
- ezsearch [http://ezsearch.in.com]
- cirne [www.cirne.it]
- cs [www.cs.rug.nl]
- dicipratis [http://dicipratis.hawg.com]
- qjset [http://www.qjset.com]
- qjset [http://www.qjset.de]

A vertical blue arrow points from the caption below to the list of websites.

Figure 5: Browser history in Internet explorer

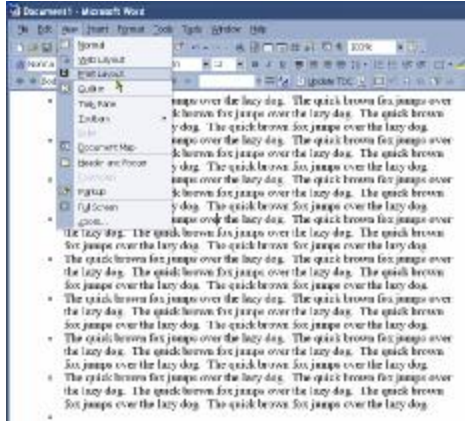
2.7 Scripting

| | |
|-------------------------------------|--|
| Usability context: | The user needs to perform the same sequence of actions over and over again with little or no variability [Tidwell 1998]. |
| Intent: | Provide a mechanism that allows the user to perform a sequence of commands or actions to a number of different objects. |
| Architectural implications: | A provision needs to be made in the architecture for grouping commands into composites or for recording and playing back sequences of commands in some way. There needs to be an appropriate representation of commands, and a repository for storing the macros. Typically, some sort of scripting language is often used to implement such functionality. This implies that all features must be scriptable. |
| Relationship with Usability: | Providing the ability to group a set of actions into one higher-level action reduces the user's cognitive load, as the user does not need to remember how to execute the individual steps of the process once the user has created a script. The user just needs to remember how to trigger the script. |

| | |
|------------------|---|
| Examples: | <ul style="list-style-type: none"> • All Microsoft's Office applications provide the ability to record macros, or to create them using the Visual Basic for Applications language. • Mozilla Firebird allows users to install extensions that extend the features of the program using scripts. • Open Office has Java bindings, that allows users to write Java programs that extend open office. Open office also supports a subset of VB.  <p>Figure 6: Record macros in Microsoft Word</p> |
|------------------|---|

2.8 Multiple views

| | |
|-------------------------------------|--|
| Usability context: | The same software functionality is required to be presented using different human-computer interface styles for different user preferences, needs or disabilities. [Brighton, 1998] |
| Intent: | Provide multiple views for different users and uses. |
| Architectural implications: | The architecture must be constructed so that components that hold the model of the data that is currently being processed are separated from components that are responsible for representing this data to the user (view) and those that handle input events (controller). The model component needs to notify the view component when the model is updated, so that the display can be redrawn. Multiple views is often facilitated through the use of the MVC pattern [Buschmann et al, 1996] |
| Relationship with Usability: | <p>Separating the model of the data from the view aids consistency across multiple views when these are employed. Separating out the controller allows different types of input devices to be used by different users, which may be useful for disabled users. Having data-specific views available at any time provides the user with guidance and will contribute to error prevention.</p> <p>Error prevention improves user <i>efficiency</i>, and increases <i>satisfaction</i>. Additionally, specific views usually consume fewer resources than the original action, which increases user efficiency.</p> |

| | |
|-------------------------|---|
| <p>Examples:</p> | <ul style="list-style-type: none"> • Microsoft Word has a number of different views that the user can select (normal view, outline view, print layout view...) and switch between these at will, which all represent the same underlying data. • Rational Rose uses a single model for various UML diagrams. Changes in one diagram affects related entities in other diagrams. • Nautilus file manager of the Gnome desktop software for Linux allows multiple views on the file system.  <p style="text-align: center;">Figure 7: Multiple views in Microsoft Word 2002</p> |
|-------------------------|---|

2.9 Multi-Channeling

| | |
|-------------------------------------|--|
| Usability context: | Users want or require (e.g. because of disabilities) access to the system using different types of devices (input/output). |
| Intent: | Provide a mechanism that allows access using different types of devices (such as desktop/laptop/WAP/web/interactive TV). |
| Architectural implications: | There may need to be a component that monitors how users access the application. Depending on which channel is used, the system should make adjustments. For example, by presenting a different navigation controls or by limiting the number of images/data sent to the user. |
| Relationship with Usability: | This pattern improves system accessibility by supporting different devices. User <i>satisfaction</i> may be increased by enabling access to the system through different devices. However, if system performance falls because of having to manage these devices, satisfaction may decrease. |

| | |
|------------------|--|
| Examples: | <ul style="list-style-type: none"> • Auction sites such as eBay or weather forecasts can be accessed from a desktop/laptop, but this information can also be obtained using interactive TV or a mobile phone. • Some set top boxes allow users to surf the Internet using an ordinary TV. • Disabled users may use special input devices that allow them to control the application. <div data-bbox="842 461 1082 779" data-label="Image"> </div> <p data-bbox="676 792 1257 824">Figure 8: EBay access through a mobile phone (WAP)</p> |
|------------------|--|

2.10 Undo

| | |
|-------------------------------------|--|
| Usability context: | Users may perform actions they want to reverse. [Welie, 2003] |
| Intent: | Allow the user to undo the effects of an action and return to the previous state. |
| Architectural implications: | In order to implement undo, a component must be present that can record the sequence of actions carried out by the user and the system, and also sufficient detail about the state of the system between each action in order that the previous state can be recovered. |
| Relationship with Usability: | Providing the ability to undo an action helps the user to correct errors if the user makes a mistake. It helps the user to feel that it is in control of the interaction. This pattern improves <i>reliability</i> , as it makes it possible to correct any errors made by the system and improves user <i>efficiency</i> . Users may become more comfortable exploring the application knowing that undo can be applied to any action, improving learnability. |
| Examples: | <ul style="list-style-type: none"> • Microsoft Word provides the ability to undo and redo (repeatedly) almost all actions while the user is working on a document. • Emacs allows all changes made in the text of a buffer to be undone, up to a certain amount of change. • Photoshop provides a multi level undo, which allows the user to set the number of steps that can be undone. This is necessary because storing the information required to do the operations requires a substantial amount of memory. <div data-bbox="719 1559 1214 1933" data-label="Image"> <p data-bbox="906 1720 1214 1742">Relationship with Usability Properties</p> <p data-bbox="906 1765 1214 1798">Providing the ability to undo an action helps the user to feel that it is in control of the interaction.</p> <p data-bbox="906 1821 981 1843">Example</p> <p data-bbox="906 1865 1214 1910">Microsoft Word provides the ability to undo almost all actions while working on a document.</p> </div> <p data-bbox="836 1944 1094 1975">Figure 9: Undo in word</p> |

2.11 User Modes

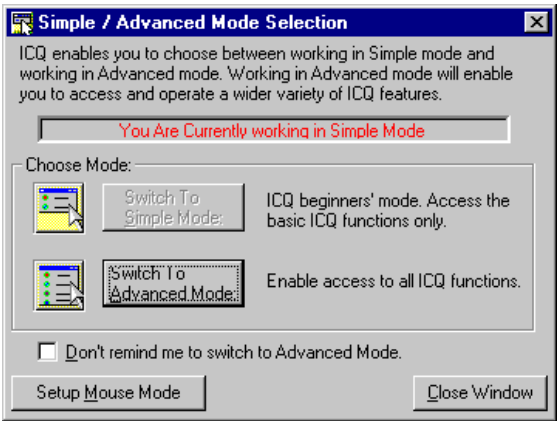
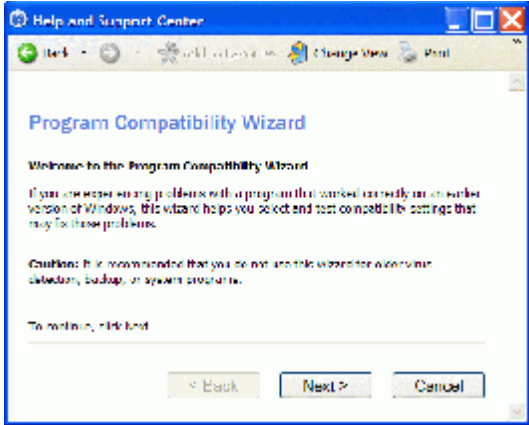
| | |
|-------------------------------------|---|
| Usability context: | The application is very complex and many of its functions can be tuned to the user's preference. Not enough is known about the user's preferences to assume defaults that will suit all users. Potential users may range from novice to expert [Welie, 2003] |
| Intent: | Provide different modes corresponding to different feature sets required by different types of users, or by the same user when performing different tasks. |
| Architectural implications: | Depending on the mode, the same set of controls may be mapped to different actions, via different sets of connectors, or more or fewer user interface components may be displayed. Using e.g. [Buschmann et al, 1996] Broker style may be required to implement this. |
| Relationship with Usability: | Supporting different modes allows personalization of the software to the user's needs or expertise. Expert users can tweak the application for their particular purposes, which increases <i>satisfaction</i> and possible <i>performance</i> , but this solution decreases <i>learnability</i> [Welie, 2003] |
| Examples: | <ul style="list-style-type: none"> WinZip allows the user to switch between "wizard" and "classic" modes, where the wizard mode gives more guidance, but the classic mode lets the expert user work more efficiently. Many websites have different modes for different people, e.g. guests, normal, logged-in users or administrators. ICQ allows the user to switch from novice (limited functionality) to advanced enabling all functionality.  |

Figure 10: User modes in ICQ

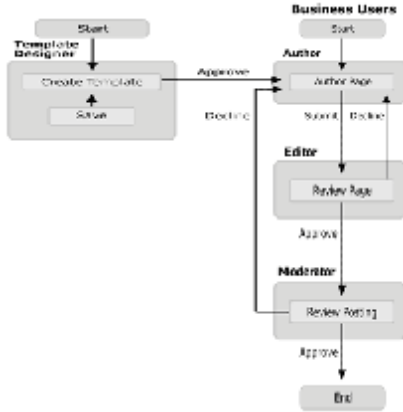
2.12 User Profiles

| | |
|-------------------------------------|---|
| Usability context: | The application will be used by users with differing abilities, cultures, and tastes [Tidwell 1998]. |
| Intent: | Build and record a profile of each (type of) user, so that specific attributes of the system (for example the layout of the user interface, the amount of data or options to show) can be set and reset each time for a different user. Different users may have different roles, and require different things from the software. |
| Architectural implications: | A repository for user data needs to be provided. This data may be added or altered either by having the user setting a preference, or by the system. User profiles often have a security impact that has major architectural implications. |
| Relationship with Usability: | Providing the facility to model different users allows a user to express preferences which is a form of adaptability. <i>Satisfaction</i> is raised because users are allowed to customize the application to their needs and wishes. |

| | |
|-------------------------|--|
| <p>Examples:</p> | <ul style="list-style-type: none"> • The install wizard used by most Windows programs guides the user through choosing various options for installation. • When partitioning hard disks during Mandrake Linux install a user can use Disk Druid, which is a disk partition wizard. • Blogger.com allows a user to create a new web log (online publishing system) in four simple steps using a wizard. Advanced users may customize their web log afterwards by editing templates. <div data-bbox="699 546 1230 969">  </div> <p>Figure 12: Program compatibility wizard in windows XP</p> |
|-------------------------|--|

2.14 Workflow model

| | |
|-------------------------------------|--|
| Usability context: | A user who is part of a workflow chain (based on some company process), should perform its specific task efficiently and reliable. |
| Intent: | Provide different only the tools or actions that they need in order to perform their specific task on a piece of data before passing it to the next person in the workflow chain. |
| Architectural implications: | A component or set of connectors that model the workflow is required, describing the data flows. A model of each user in the system is also required, so the actions they need to perform on the data can be provided (see also user profile). |
| Relationship with Usability: | This pattern improves user <i>efficiency</i> and <i>reliability</i> , as the user will only see the information and tasks corresponding to the operations to be done. |

| | |
|-------------------------|---|
| <p>Examples:</p> | <ul style="list-style-type: none"> • Most CMS and ERP systems are workflow model based. • A typical example of an administrative process that is workflow based is the handling of an expense account form. An employee fills in the proper information; the form is routed to the employee's manager for approval and then on to the accounting department to disburse the appropriate check and mail it to the employee. • Online publishing: a journalist writes an article and submits it online to an editor for review before it is published on the website of a newspaper. This process is often automated in a workflow model.  <p>Figure 13: Workflow model in Microsoft Content Management Server 2001</p> |
|-------------------------|---|

2.15 Emulation

| | |
|-------------------------------------|--|
| Usability context: | Users are familiar with a particular system and now require consistency in terms of interface and behavior between different pieces of software |
| Intent: | Emulate the appearance and/or behavior of a different system. |
| Architectural implications: | Command interfaces and views but also behavior needs to be replaceable and interchangeable, or there needs to be provision for a translation from one command language and view to another in order to enable emulation. This differs from the providing multiple views diagram because the behavior of the application should be replaceable. |
| Relationship with Usability: | Emulation can provide consistency in terms of interface and behavior between different pieces of software, which may aid <i>learnability</i> |

| | |
|-------------------------|---|
| <p>Examples:</p> | <ul style="list-style-type: none"> • Microsoft Word 97 can be made to emulate WordPerfect, so that it is easier to use for users who are used to that system. • Windows XP offers a new configuration menu; however, it is possible to switch to the “classic view” for users more familiar with windows 2000 or windows 98. • Jedit (Open Source programmer's text editor) can have EMACS and VI key bindings modes. <div data-bbox="715 521 1219 893" data-label="Image"> </div> <p style="text-align: center;">Figure 14: Windows XP control panel</p> |
|-------------------------|---|

2.16 Context Sensitive Help

| | |
|--|---|
| <p>Usability context:</p> | <p>When help in the context of the current task would be useful.</p> |
| <p>Intent:</p> | <p>Monitor what the user is currently doing, and make documentation available that is relevant to the completion of that task.</p> |
| <p>Architectural implications:</p> | <p>There needs to be provision in the architecture for a component that tracks what the user is doing at any time and targets a relevant portion of the available help.</p> |
| <p>Relationship with Usability:</p> | <p>The provision of context sensitive help can give the user guidance. This pattern will improve <i>reliability</i> and <i>efficiency</i>, as well as <i>learnability</i> for non-expert users.</p> |

Examples:

- Microsoft Word includes context sensitive help. Depending on what feature the user is currently using (entering text, manipulating an image, selecting a font style) the Office Assistant will offer different pieces of advice (although some users feel that it is too forceful in its advice).
- Depending upon what the cursor is currently pointing to; Word will pop up a small description or explanation of that feature.
- Eclipse (a popular Java development environment) allows the user to consult context sensitive info (such as specific API specifications)



Figure 15: Context sensitive help

3. Related work

In our work, the concept of a pattern is used to define an architecturally sensitive usability pattern. Software patterns first became popular with the object-oriented Design Patterns book [Gamma et al 1995]. Since then a pattern community has emerged that produced specifies patterns for all sorts of problems (e.g. architectural styles [Buschmann et al, 1996] and object oriented frameworks [Coplien and Schmidt, 1995].

An architecturally sensitive usability pattern as defined in our work is not the same as a design pattern [Gamma et al 1995] Unlike the design patterns, architecturally sensitive patterns do not specify a specific design solution in terms of objects and classes. Instead, we outline potential architectural implications that face developers looking to solve the problem the architecturally sensitive pattern represents.

One aspect that architecturally sensitive usability patterns share with design patterns is that they capture design experience in a form that can be effectively reused by software designers to improve the usability of their software, without having to address each problem from scratch. The aim is to capture what was previously very much the “art” of designing usable software and turn it into a repeatable engineering process.

Previous work has been done in the area of usability patterns, by [Tidwell 1998], [Perzel and Kane 1999], [Welie and Trætteberg, 2000]. Several usability pattern collections [Brighton, 1998], [Common ground, 1999], [Welie, 2003], [PoInter, 2003] can be found on the web³. Most of these usability patterns collections refrain from providing or discussing implementation details. Our paper is not different in that respect because it does not provide specific implementation details. However, we do discuss potential architectural implications. Our work has been influenced by the earlier work, but takes a different standpoint, concentrating on the architectural impact that patterns may have on a system. We consider only patterns that should be applied during the design of a system’s software architecture, rather than during the detailed design stage.

[Bass et al, 2002] give examples of architectural patterns that may aid usability. They have identified scenarios that illustrate particular aspects of usability that are architecture-sensitive and present architectural patterns for implementing these aspects of usability

4. Conclusions

This paper describes an integrated set of architecturally sensitive usability patterns, that in most cases, are considered to have a positive effect on the level of usability but that are difficult to retro-fit into applications because these design solutions may require architectural support. For each of our architecturally sensitive usability patterns we have analyzed the usability effect and the potential architectural implications. The architecturally sensitive usability patterns have been

³ for a complete overview: http://www.pliant.org/personal/Tom_Erickson/InteractionPatterns.html

derived from internal case studies at the industrial partners in the STATUS project and from existing usability pattern collections.

We believe that it is vital that usability issues are taken into account during the architecture design phase to as large an extent as is possible to prevent the high costs incurring adaptive maintenance activities once the system has been implemented. Our collection of patterns can be used during architectural design to heuristically evaluate if the architecture needs to be modified to support the use of such patterns. However, we do not claim that a particular pattern will always improve usability. It is up to the architect to assess whether implementing a pattern at the architectural level will improve usability. In addition, the architect will have to balance usability optimizing solutions with other quality attributes such as performance, maintainability or security.

Future research should focus on verifying our assumptions concerning the architectural sensitiveness of the usability patterns. Proving the architecture sensitivity of a usability pattern is difficult because the patterns we presented may be implemented in different ways, influencing architectural sensitiveness.

Practice shows that patterns such as cancel, undo and history logging may be implemented by the command pattern [Gamma et al 1995], emulation and providing multiple views may be implemented by the MVC pattern [Buschmann et al, 1996]. Actions for multiple objects may be implemented by the composite pattern [Gamma et al 1995] or the visitor pattern [Gamma et al 1995]. Investigating how our usability patterns may be implemented by design patterns or architectural patterns is considered as future work.

In addition to the patterns that we identified, there are some techniques that can be applied to the way that the development team designs and builds the software and that may lead to improvements in usability for the end user. For example, the use of an application framework as a baseline on which to construct applications may be of benefit, promoting consistency in the appearance and behavior of components across a number of applications. For instance, using the Microsoft Foundation Classes when building a Windows application will provide “common” Windows functionality that will be familiar to users who have previously used other applications build on this library. This is not a pattern that can be applied to the architecture in the same way as those presented in section 5, but it is nonetheless something which will be considered during the further study of the relationship between software architecture and usability during the remaining parts of this project.

5. Acknowledgements

We would like to thank the partners in the STATUS project for their input and their cooperation.

6. References

[IEEE, 1998]

IEEE Architecture Working Group. Recommended practice for architectural description. Draft IEEE Standard P1471/D4.1, IEEE, 1998,

[Bass et al, 2001]

Bass, Lenn; Kates, Jessie & John, Bonnie. E. Achieving Usability through software architecture, 2002, <http://www.sei.cmu.edu/publications/documents/01.reports/01tr005.html>

[Berkun, 2002]

Berkun, S., The list of fourteen reasons ease of use doesn't happen on engineering projects, <http://www.uiweb.com/issues/issue22.htm>

[Bosch, 2000]

Bosch, J., 2000. Design and Use of Software Architectures: Adopting and Evolving a Product Line Approach, Pearson Education (Addison-Wesley and ACM Press).

[Brooks, 1995]

Brooks, F. P. jr., 1995. The Mythical Man-Month: Essays on Software Engineering, Twentieth Anniversary Edition, Addison-Wesley.

[Buschmann et al, 1996]

Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P., Stal, M., 1996. Pattern-Oriented Software Architecture: A System of Patterns, John Wiley and Son Ltd.

[Coplien and Schmidt, 1995]

Coplien, J. O., Schmidt, D. C., 1995. Pattern Languages of Program Design, Addison-Wesley (Software Patterns Series).

[Folmer and Bosch, 2002]

Folmer, E. & Bosch, J. Architecting for usability; a survey. Journal of systems and software 0-0, 2002.

[Gamma et al 1995]

Gamma, E., Helm, R., Johnson, R., Vlissides, J., 1995. Design Patterns Elements of Reusable Object-Orientated Software., Addison -Wesley.

[Granlund et al, 2001]

Granlund, Å.; Lafrenière, D. & Carr, D. A., 2001, Pattern-Supported Approach to the User Interface Design Process, Proceedings of HCI International 2001 9th international Conference on Human-Computer interaction.

[PoInter, 2003]

Lancaster University, PoInter: Patterns of INTERaction collection,
<http://www.comp.lancs.ac.uk/computing/research/cseg/projects/pointer/patterns.html>

[Lederer and Prasad, 1992]

Lederer, A. L. P. J. Nine Management Guidelines for Better cost Estimating. Communications of the ACM 51-59, 1992.

[Nielsen, 1993]

Nielsen, J., 1993. Usability Engineering, Academic press, San Diego CA.

[Perzel and Kane 1999]

Perzel, K. & Kane, D., 1999, Usability Patterns for Applications on the World Wide Web.

[Pressman, 1992]

Pressman, R. S., 1992. Software Engineering: A Practitioner's Approach, McGraw-Hill, NY.

[Brighton, 1998]

The Usability Group at the University of Brighton, UK., The Brighton Usability Pattern Collection.
<http://www.cmis.brighton.ac.uk/research/patterns/home.html>

[Tidwell 1998]

Tidwell, J., 1998, Interaction Design Patterns, Conference on Pattern Languages of Programming 1998.

[Common ground, 1999]

Tidwell, J., Common ground: Pattern Language for Human-Computer Interface Design,
http://www.mit.edu/~jtidwell/interaction_patterns.html

[Welie, 2003]

Welie, M., GUI Design patterns, <http://www.welie.com/>

[Welie and Trættemberg, 2000]

Welie, M. & Trættemberg, H., 2000, Interaction Patterns in User Interfaces, Conference on Pattern Languages of Programming (PloP) 7th.

A Pattern Language for Supporting Wireless Communication Between End-Points

Franco Guidi-Polanco, Claudio Cubillos F., Giuseppe Menga, and Samuel Penha

Dip. Automatica e Informatica, Politecnico di Torino, Italy

C.so Duca degli Abruzzi 24, I-10129 Turin, Italy

Phone: +39 011 564 7084 Fax: +39 011 564 7099

E-mail : {franco.guidi, claudio.cubillos, menga}@polito.it

I. Introduction

Development in information and communication technologies has made possible the vision of pervasive computing as the paradigm for the beginning of the 21st century [12]: computing devices are evolving to “wearable” forms, everyday objects are becoming “intelligent”, and their traditional functions are being shared through omnipresent networks, linking local environments with global worldwide services. Different physical communication technologies, such as Wireless LAN (IEEE 802.11b), General Packet Radio Service (GPRS), or Universal Mobile Telecommunication System (UMTS) [8], are currently available, and multi-interface devices are taking advantage of them. Universal communication is possible due to the diffusion and adoption of Internet protocols as open inter-applications standards.

As consequence of these trends in information and communication technologies, we envisioned the concept of “Global Automation” [2]. Global Automation means transferring and extending classical process control and factory automation ideas to large scale distributed environments, allowing the creation of flat interconnections of autonomous, decentralized and highly interacting decision making/control modules. As enabling architecture for the implementation of global automation systems we started developing the Global Automation Platform (GAP), a middleware framework made up by a communication infrastructure that conceives, under a common architectural reference model, the integration of fixed and mobile services, using a wide range of communication technologies. From the point of view of the execution environment, the GAP platform has been designed to integrate a network of heterogeneous devices, from embedded processors with limited processing power, memory, networking capabilities, to large enterprise servers. The programming languages that we are using to implement the architecture are Java and G++ (our extension of C++ for automation systems) [7], C++ for Windows CE, G++ for Windows CE, Java 2 Standard Edition (J2SE), and Java 2 Micro Edition.

The pattern language presented in this article is the result of our efforts towards the design of the communication infrastructure for the GAP platform, and the experience we have made with it in the fields of automation systems and mobile robotics. In particular, our “laboratory” experiences regarded the integration of mobile robots that have to interact among them and with external services -as directory services, human or automated controllers, etc.- using mainly wireless connections.

Finally, the resulting architecture –and the design patterns that describe it- can be applied in other networking scenarios, not necessary related to the field of mobile robotics. In particular, we believe that the patterns described in this article can be useful for developers interested in the integration of distributed applications using different networking technologies, especially wireless networks.

II The Pattern Language

The concrete problems addressed by this pattern language involve the design of software systems that requires establishing and managing communication between two end-points relying on different (mainly wireless) networking technologies.

The first pattern, called *Communicator/Session/Channel* describes an architecture that provides high-level connectivity for applications, maintaining the state of the message transmission independently from the state of the underlying network channel. The adoption of this pattern allows the development of applications that do not require awareness of both, the network channel stack, and the state of the communications in case of brief interruptions.

The second pattern, *Channel with Configurable Protocols* describes how a communication channel can be structured in terms of different combinations of messaging formats and networking protocols, that can be separately plugged-in the system.

The third pattern, the *URL-Based Connection Factory* presents a common interface for the specification and initialization of the appropriate combination of network classes in terms of network protocols and message formats.

Figure 1 shows the relationships between the different design patterns pertaining to this pattern language.

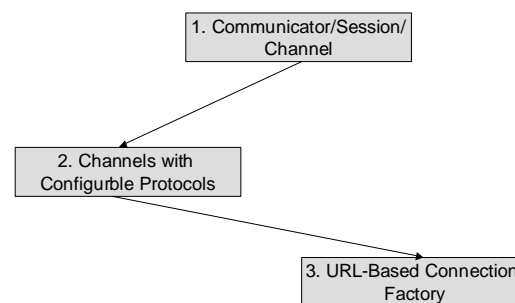


Figure 1 The pattern language for designing communication between end-points.

As a way to introduce this patterns, a metaphor based on everyday communication between people is presented: Consider a common case of direct communication between two friends that frequently communicate writing e-mails, letters, chatting lines, phone calls, and so on. They can maintain a conversation along the time using these different means. We can view each treated topic (vacations, health, work, etc.) as a “session”, and the state of each session depends on the development of the conversation about that topic. Each specific communication means is what we call “channel”, and a channel is required to develop a conversation. In this context we can imagine one of the friends telling the other one in a letter that is sick, then in the next communication (e.g a phone call) that friend will (probably) ask him about his illness (case of the channel change in a session). On the other hand, in the same phone call they can talk about other things, like politics, family, jokes, etc. (case of channel sharing between sessions).

This pattern language aims to provide a similar flexible communication model for software applications.

Pattern 1: The Communicator/Session/Channel

This pattern describes an architecture for the management of communication sessions between objects interacting through a (wireless) network. It hides to the application the complexity regarding networking protocols and the state of the communications.

Problem

Due to radio interferences and signal loosing, connections between some mobile systems over wireless networks are subject to interruptions and possible on-the-fly changes in the network configuration. In this way, a mobile system could try to restore a broken channel or, even more, could try to switch to different predefined channels (relying on a different communication technology) in order to reestablish the communication. If a sort of timeout is defined to retry a reconnection or an interrupted message transmission, the application doesn't need to be aware about brief interruptions or channel switching procedures that do not result in disconnectedness time or transmission delays longer than the allowed timeout.

The problem addressed by this pattern is: How to manage the continuity of communication over non-continuous network connections?

Context

You are developing a system that requires creating connections between two end-points using mainly wireless networking protocols. The system should support automatic mechanisms for connection reestablishment and/or vertical handoff [9].

Forces

- A system running on a mobile unit can be designed to handle a broken connection trying either to restore it or to switch to a different wireless network.
- An application can require that brief network interruptions -to handle reconnections or channel switching- do not affect the state of communication. In other words, the application should know which messages have already been sent and received in order to allow, after interruptions, the communication reestablishment and continuity.
- In some cases can be also convenient to share a communication channel between two or more processes running in the same systems.
- Programming languages and execution platforms usually offer low-level components, classes or routines to achieve connectivity across a network, which do not provide support for network interruptions and more advanced channel management.

Example

Consider a mobile robot that has to explore a certain area (Figure 2), communicating measurements obtained from its sensors (temperature, obstacles and their positions, intruders, etc.) to its remote monitoring station. The robot is enabled with two or more network interfaces -let's say Wireless LAN and GPRS- allowing it to select the most suitable under each circumstance, and to make the change even during a communication (due to the larger bandwidth and lower cost of the W-LAN compared to the GPRS, the robot will prefer the former any time is possible). This means that the robot needs to maintain accurate information about the state of the communication, that is, which messages have been sent and which ones have already been received, independently from the changes in the network used.

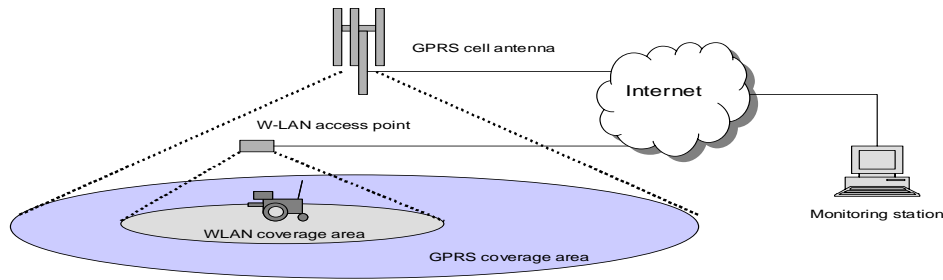


Figure 2. Mobile robot example.

Solution

This design pattern provides an architecture in which the message transmission (messages sent, received, etc.) is decoupled from the network channel, allowing the first one not to be affected by non-permanent interruptions or changes in the network access. This decoupling is based on the implementation of a session-level protocol, on top of the already existing protocols in the communication network.

Figure 3 presents the class diagram that describes this pattern, where the following participants are recognized:

a) Application

It is the system that requires maintaining communications with other remote systems. To establish a connection with a remote system, the application asks to the *Communicator* for the creation of a new *Session*. The application sends and receives messages through the network interacting directly with *Session* instances.

b) Communicator

It is the base class from which can be derived all classes with communication capabilities. It acts as a façade pattern hiding to the application all complexities related to networking. For the creation of new *Session* instances this class offers two methods: *createSessionAcceptor*, to create server-side connections; and *createSessionConnector*, to create client-side connections. In both cases parameters describing the connection (e.g. URLs) have to be given.

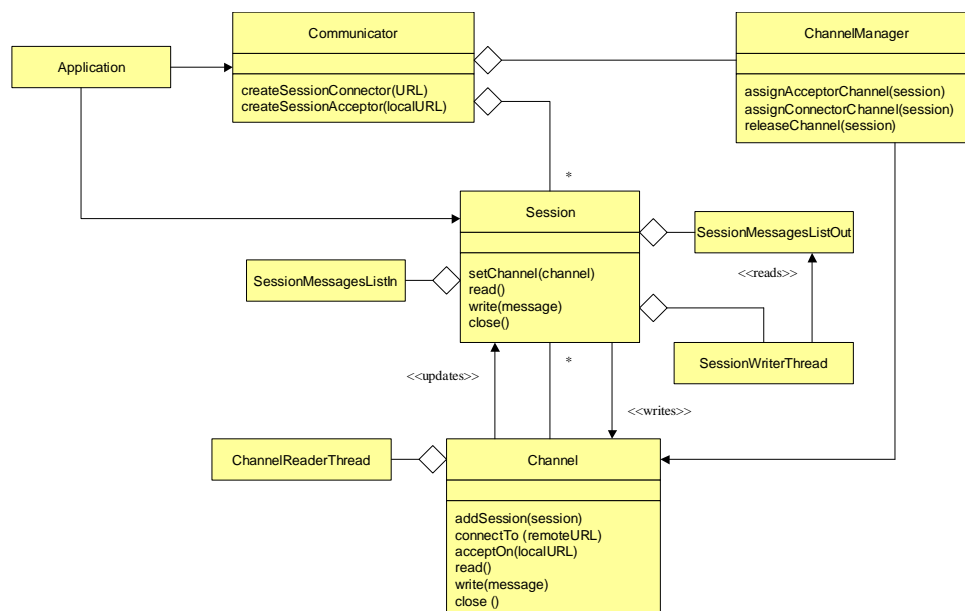


Figure 3. The Communicator class diagram.

b) Channel Manager

It is the class that encapsulates both, the connection reestablishment and the change of the channel assigned to a session. The criteria used to reestablish a broken channel or to switch among channels could be implemented as simple external requests, or as an intelligent autonomous decision that takes into consideration the state of communication and some quality of service metrics of the network.

c) Session

It is the class that maintains the state of transmitted messages between two end-points. Sessions receive from the Communicator (or directly from the application) messages to deliver across the network, which are first stored in the list of out coming messages, and then sent through the Channel. Sessions are prepared to stop writing to a Channel before initiating a change of Channel, and, after the change, to restore the state of connection. Even more, messages sent using one Channel could receive its acknowledgement through another Channel.

d) Session Messages Lists (In and Out)

Each Session holds two lists, the `SessionMessageListIn`, and the `SessionMessageListOut`. The former stores incoming messages, and the `ChannelReaderThread` updates it through the Session. The latter list manages out coming messages, and it is updated with new messages every time the write method of the Session is invoked. Messages are not removed from the list of out coming messages until acknowledgements are received or a request of closing the session is accepted. Correspondingly, incoming messages are not deleted from the In list until they are read by the application.

e) Session writer thread

It is the thread that runs in the Session. It takes the first message from the `SessionMessageListOut` and sends it to the receiver invoking the write method of the Channel.

f) Channel

It is the class in charge of sending the message across the network, representing one side of a connection between two end-points. It acts as an interface for the Session, providing methods to read from and write to the network. It encapsulates the stack of underlying network protocols needed for the communication, together with the control mechanism for data transmission. Decoupling the session state from the channel state allows Channel sharing among two or more Sessions, as it is shown in Figure 4.

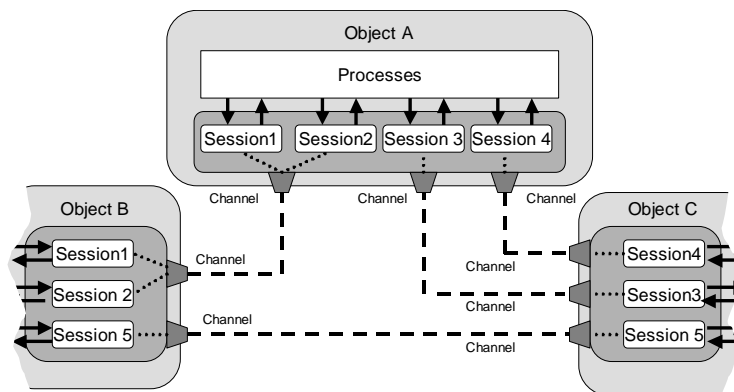


Figure 4 *A communication example*

g) Channel Reader Thread

It is the thread that runs inside the Channel. It receives messages from the network and stores them in the `SessionMessagesListIn`.

Notes:

Dynamics- In the first part of Figure 5 is shown the message sequence that describes the creation of a new session (in server mode). When the opening of a new session in server mode is requested, the `Communicator` creates a new `Session` instance, and then it asks the `ChannelManager` for the creation of a new `Channel` instance, which is further associated to the `Session`. Then the specific procedure for opening the Channel in the server mode is called through the method `waitForConnection`, which includes as parameter the configuration of the local protocol and port where the system must listen for an incoming connection. Opening a client connection is done in a similar way, the `Application` calls the `Communicator`'s `connectToRemote` method, which performs the same initialization of the `Session` and the `Channel`, but instead of calling in this the `waitForConnection` method, it is called the method `connectToRemote`. This method activates the low-level client side connection.

In the same figure is described the closing connection procedure, which is the same independently from the role (client/server) played by the party that initiates the procedure. Due to the fact that a `Channel` could be shared between different `Sessions`, the `Channel` is first released from the `Session` and it is closed when there are no other `Sessions` using it.

Architecture overhead- The reading and writing procedures rely in an asynchronous communication model, supported by the `ChannelReaderThread` and the `SessionWriterThread`. When the application has to communicate through synchronous connections, this approach incurs in an unnecessary overhead, as the input and output flows can be handled in the same thread. It is also important to notice that when a `Channel` is shared among two or more `Sessions`, an overhead exists for distinguishing which message goes to which `Session`. In practice this means that the head of each message has to include the target `Session` identifier, and the `Channel` has to provide a mechanism for sending the message to the corresponding destination `Session`.

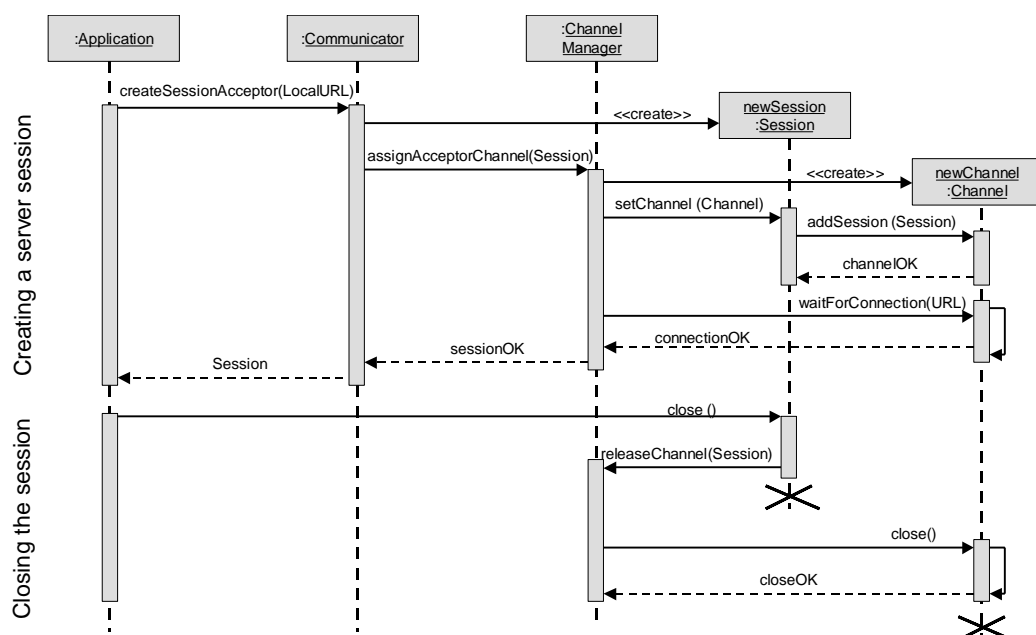


Figure 5 Message sequence diagram for session creation and closing using a new channel

Resulting context

Applications can manage communications over different networks, without awareness of either temporal interruptions or the network handover procedure.

Known uses

- In [6] is described the experience of an exploration robot Pioneer DX2 running a C++ control program that accepts commands from a remote operator through TCP sockets on a Wireless LAN, and HTTP/WAP over GPRS. The robot can switch between them accordingly with the kind of network signal received in its position.

Related patterns

- *Channel with Configurable Protocols*: The networking functions encapsulated by the class `Channel` can be implemented using the *Channel with Configurable Protocols* pattern, which provides access to the network selecting the adequate protocol.
- *Façade*: The `Communicator` class acts as a *Façade* pattern [3], providing to the application a simplified interface for the session creation and management.

Pattern 2: Channels with Configurable Protocols

It provides a way to handle message formats over different network protocols.

Problem

A distributed application usually requires the adoption of a common message protocol (e.g. XML, DAML+OIL, FIPA-SL, etc.) for its communication system. This message protocol has to be implemented on top of certain stacks of network protocols (e.g. HTTP over TCP-IP over IEEE802.11b, XML over SMSs, etc.), which vary depending on the kind of communicating device and their networking capabilities. If the whole system has a high degree of heterogeneity, then it could be necessary to implement the same message protocol over different network protocol stacks, increasing the complexity of the system.

This pattern solves the problem about how to combine message and networking protocols, making possible: (1) an efficient implementation of a common message protocol over different networking protocols; and (2) the reuse of network protocol management models for supporting different messaging protocols.

Context

You are developing a system that has to be prepared to interact with remote systems. These systems are running on a heterogeneous environment of devices and networks, and you have to provide message exchange formats over different networking protocols.

Forces

- Some execution platforms provide certain classes or functions to manage standard protocol stacks that can be used to format and send messages across the network. However, these stacks are not always shared by all the execution environments integrated by the communication system.
- Developing the own protocol stacks as single components can produce a simple architecture if just one stack is required for the entire system.
- Developing components for each level of the network protocol stack produces a more flexible but complex architecture.

- Most systems require a few combinations of protocol stacks.

Example

A team of in field technicians is in charge of measuring the pollution levels during an environmental emergency. The technicians are equipped with different instruments according to the polluting agents to quantify. These instruments are capable to transmit in real time their data to a central monitoring station that summarizes them and generates the entire map of pollution levels for the given area under emergency. The instruments run on different mobile platforms (e.g. J2ME, Windows CE, PALM-OS, etc.) and use different network stacks to transmit the information, combining different protocols and technologies (e.g. HTTP over Wireless LAN, SMSs over a GSM network, etc.) The central monitoring station has to be prepared to interact with these devices and new ones, without the need of further modifications at the application level.

Solution

This pattern divides the implementation of the stack of communication protocols into two interacting classes: a message protocol class that encapsulates the format (or language) of the message, and a network protocol class that encapsulates the underlying networking protocols. This decouples the message protocol from the network protocols, allowing the first one to rely on different configurations of the second one.

The participants involved in this pattern are (see Figure 6):

a) Application

It is the final user of the network services. It instantiates a `Channel` object, and asks it to create new connections for sending and receiving messages.

b) Channel

This class represents a high-level abstraction for a connection between two end-points, and acts as an interface for applications, providing methods to write to and read from the network. At the beginning of a new communication this class is responsible for the instantiation of the concrete `ChannelProtocol` class, according to the desired network protocol and message format. The `Channel` offers to the application two complementary ways to open new connections. The first, a server-side connection, is supported by the method `acceptOn` that waits for an incoming connection received in the local address specified as parameter of the method. The second, a client-side connection, is provided by the method `connectTo` that tries to open a session as a client connecting to the address received as parameter (establishing a connection requires one part acting as server and the other as client). The class also provides methods for reading (`read`) and writing (`write`) messages, and closing (`close`) the connection. All of these methods are mapped to the corresponding methods of the associated `ConcreteMessageProtocol` instance.

c) Channel Protocol

It is an abstract class that holds a reference to a stack composed by a message protocol and a network protocol. Its subclasses have to bind together a message-network protocol pair (instances of `MessageProtocol` and `NetworkProtocol` subclasses).

d) Concrete Channel Protocol

By extending the `ChannelProtocol` abstract class, the concrete channel protocol provides one instance of concrete message protocol with another instance of concrete network protocol. This allows a quick instantiation of a specific stack of protocols. The Concrete channel protocols depend on the configurations that are expected to be supported by the device or system.

e) Message Protocol

The pattern provides this abstract class from which concrete message protocols must be derived. The abstract class `MessageProtocol` holds a reference to a `NetworkProtocol` subclass instance. The class provides the implementation of the concrete methods `setNetworkProtocol`, which sets the network protocol to use; `acceptOn` and `connectTo`, which forward to the concrete network protocol the opening of a channel in server or client mode respectively, using the address provided as parameter; and the `close` method, which forwards the closing of the channel to the underlying concrete network protocol. This class also declares abstracts methods for writing (`write`) and reading (`read`) messages to/from the network protocol, which have to be implemented in its subclasses.

f) Concrete Message Protocol

The pattern delegates to subclasses of `MessageProtocol` the implementation of the message format and control logic of the protocol used to exchange messages between the communicating systems. Concrete message protocol classes must implement the `write` and `read` methods that perform encoding and decoding of messages. Both operations have to be implemented according to the selected message format (e.g. SQL, XML, SOAP, FIPA-SL, etc). A concrete message protocol can specify only a format for the messages (e.g. FIPA, KIF, or KQML formats) or can define a specific grammar or language to use for the messages (e.g. XML, DAML Prolog, SQL, etc). In the end, each concrete message protocol class can be considered as a specific parser for a certain kind of messages.

g) Network Protocol

This is an abstract class from which classes that encapsulate the stack of network protocols have to be derived. The abstract class `NetworkProtocol` holds a reference to a `MessageProtocol` subclass instance. It also implements the concrete method `setMessageProtocol`, which sets the Concrete Message Protocol instance. It also declares the abstract methods `acceptOn` and `connectTo` for the creation of connections as server or client, using the address received as parameter; the method `close` to close the opened channel; and the methods `write` and `read`, for data transmission.

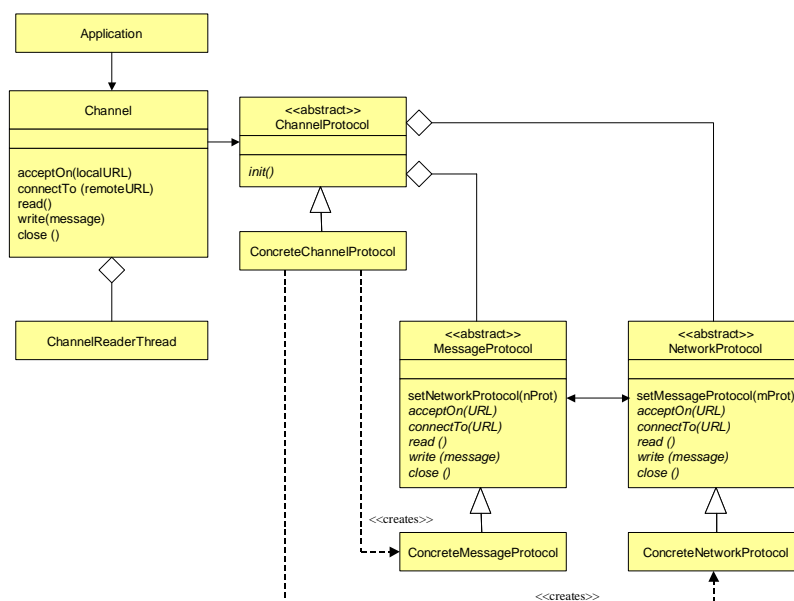


Figure 6 The Channel with Configurable Protocol.

h) Concrete Network Protocol

It extends the abstract `NetworkProtocol` class. It encapsulates a concrete stack of networking protocols, and implements the abstract methods declared in its base class for opening and closing connections, and for sending and receiving data through the network.

Figure 7 shows the message sequence diagram that describes the loading of protocols during the channel opening. Instances of concrete implementations of message protocol and network protocols are created when the `init` method of the concrete channel protocol is invoked. After initialization an `acceptOn` command is forwarded from the `Channel` instance to the concrete network protocol, which is responsible for the creation and activation of the specific objects that perform the low level communication. Closing the channel is performed by first closing the network protocol and then the message protocol. Further use of this stack requires a new invocation of the `init` method of the concrete channel protocol object.

Resulting context

When applying this pattern, message protocols (message format) and network protocols are defined separately and can be combined in runtime to create the desired network stacks.

Known uses

- In the world of Multi-Agent Systems, the Foundation for Intelligent Physical Agents (FIPA) has defined a format for the message exchange between agents [5]. In this way, a common format is used over different communication protocols.
- A similar approach is one followed in the Knowledge Sharing Effort (KSE) initiative in which they developed KQML [4], a message format and a set of interaction protocols for agents and platforms to adopt in its communications.

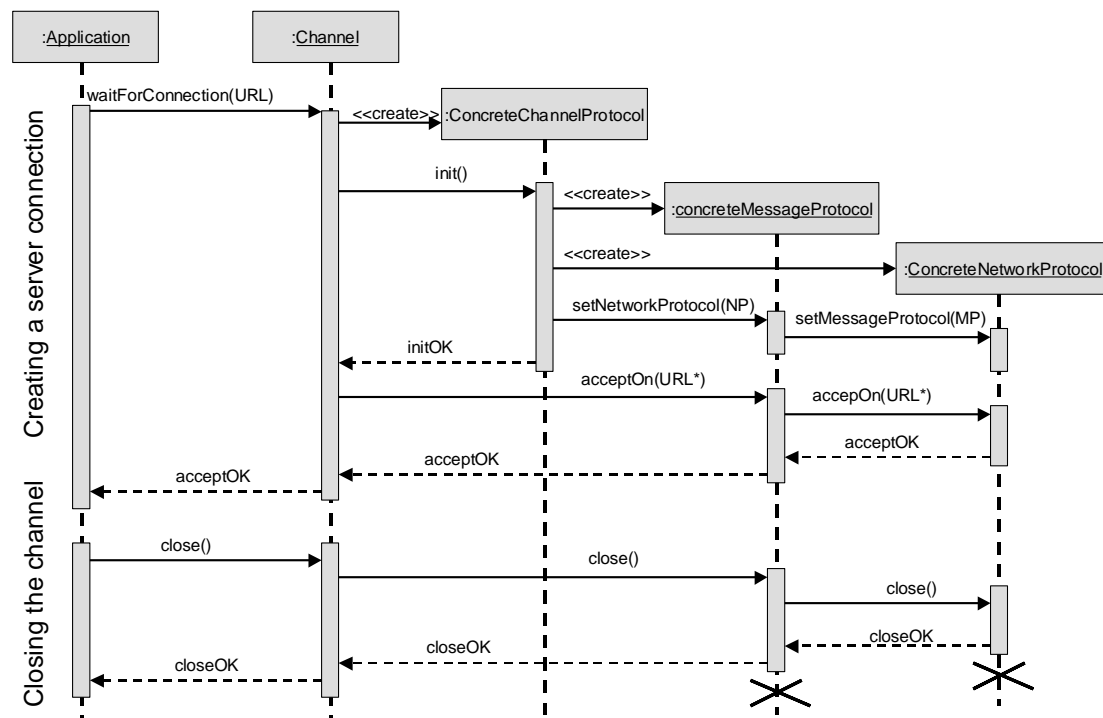


Figure 7 Opening and closing a Channel.

Related patterns

- URL-Based Connection factory: provides the mechanism for specification and instantiation of the desired concrete protocol object, using a common interface.
- Composite: The Channel class can use this pattern to define a composite message. Therefore it will be able to manage messages with different granularities, that is, of different length and type.
- Prototype: The Prototype pattern can be used to maintain the instances of all the Channel Protocols that have to be supported by an application.

Pattern 3: The URL-Based Connection Factory

It provides a single interface for opening communication connections using different protocol stacks.

Problem

Communication protocols evolve through time. Therefore communication applications should be flexible enough to adopt new protocols without incurring into modifications. Common applications, instead, include as part of their code the instantiation of classes associated to the protocols to use. As consequence, this simpler but more inflexible approach requires maintenance in order to deal with new protocols.

The problem is how to manage communication protocols dynamically, in such a way that they do not require the maintenance of the application.

Context

You are developing a system that has to interact with remote systems using different stacks of protocols. These protocols can be known or unknown at design time.

Forces

- Different stacks of communication protocols have specific sets of parameters that describe how a connection has to be done, but this requires managing different sets of parameters for the different communication protocols.
- The application should be able to manage new communication protocols, so it is desirable that this is done without incurring into further modifications.
- The GoF's Factory Method [3] defines in a class a common interface for delegating object instantiation to its subclasses.
- Internet resources can be accessed in different ways, but the use of Uniform Resource Locators provides a unique way to access them.

Example

An agent platform (such as JADE [11]) has to be able to connect to other platforms, agents and external resources using different protocols, such as IIOP/GIOP, RMI, HTTP, and so on. As soon as new protocols are decided to be incorporated, the platform should be able to adopt them without any changes in its existing code.

Solution

Objectify the connections and provide a factory class able to instantiate them with a single method. In addition, the protocol selection can be done by specifying a URL that describes the connection. The Figure 8 shows the class diagram of this pattern.

In this way, only the corresponding concrete connection class has to be created in face of a new protocol, implementing the common interface between the application and concrete connections. This common interface (Connection) together with the single method for connections instantiation enables the rest of the application to gain protocol independence.

The following participants compose the pattern:

a) Connection Factory

This class provides two methods to create instances of Connections: `connectTo`, which initiates a client-side connection to the address specified in the URL received as parameter; and `acceptOn`, for the creation of server-side connections that accepts as parameter the local URL, where the system will be listening to an incoming connection. For example, an application could ask for a TCP socket connection using the following method call (Java code):

```
Connection c;  
c = ConnectionFactory.connectTo("tcpsocket://lab.polito.it:8590");
```

This represents the request to open a TCP socket to the host called *lab.polito.it* in the remote port 8590. As a result of the invocation of each method, a concrete connection is returned.

b) Connection

It is the interface that concrete connections have to implement. Applications manage concrete connections through this interface. The interface specifies the methods that connections must manage. These methods depend on the concrete application under development. Usually operations to read and write messages, together with a method to close the connection are included.

c) Concrete Connection

This class represents a concrete connection that implements a specific stack of networking protocols (e.g. HTTP over TCP). It provides implementation for all the methods declared in the Connection interface. It encapsulates all the classes required to manage a connection (e.g. TCP socket related classes).

d) Application

Invokes the factory methods to get connections from the ConnectionFactory, and manages the returned concrete connection through the Connection interface.

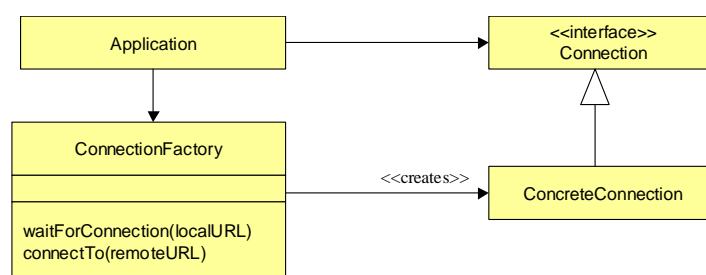


Figure 8 The Connection Factory.

Implementation

The implementation of the `ConnectionFactory` considers the adoption of a simple parser for the interpretation of the URL given as parameter in the methods `connectTo` and `acceptOn`. This parser has to identify the different parts of the URL. For example, in the case of an URL like `XXX://hostname:port` the parser should recognize `XXX` as the connection protocol, `hostname` as the target host, and `port` as the remote port to reach (other structures of URL should be also supported).

Creating the connection requires instantiation of the adequate concrete connection class, based on the protocol name. Languages that support reflection (e.g. Java) could take direct benefit of this capability using dynamic class resolution and loading. Consider the case of an implementation in Java, where classes to manage connections could follow the name structure `XXXConnection`, where `XXX` represents a string of characters of any length but equals to the name of the protocol part indicated in the URL. In this way, resolving the address “tcpsocket://lab.polito.it:8590” could be translated to the creation of an instance of “TCP SOCKETConnection”), and then the instantiation of the corresponding object can be done in this way:

```
1    connectionName = getProtocol(URL) + "Connection";
2    Class connCl = Class.forName(connectionName);
3    Constructor cnst = connCl.getConstructor(argsClass);
4    Connection con = (Connection) cnst.newInstance(instArgs);
```

In line 1 the method `getProtocol` extracts the protocol name part from the URL and returns it in uppercase (for simplicity). In line 2, the class corresponding to the name given in variable `connectionName` is loaded. In line 3 the variable `cnst` gets a reference to the constructor of the selected connection having the types of parameters indicated in array `argsClass`. Finally, in line 4 the selected connection with values of parameters indicated in array `instArgs` is instantiated.

The runtime instantiation of the concrete connections, in languages that do not support reflection, can be approached using the *Prototype* pattern described by Gamma et al [3].

Resulting context

A single interface is used to specify and instantiate different stacks of protocols.

Known uses

- The Sun's Generic Connection Framework [10] adopted this model to open communication from wireless mobile devices enabled with the Java 2 Micro Edition (J2ME). It was conceived to offer an extensible infrastructure for input/output and network operations.
- This pattern was also adopted in the Java Reflective Broker (JRB) [1], a distributed object model for the Java language developed by Telecom Italia Laboratory (formerly CSELT), when is required to get references from remote objects.
- A similar approach is also used by JADE [11], a software framework fully implemented in Java, designed to simplify the implementation of multi-agent systems. Its agents and the agent platform can establish different connections with the use of MTPs (Message Transport Protocol).

Related patterns

- *Channel with Configurable Protocols*: Both patterns can be integrated in order to provide a more flexible selection of channel protocols.

Acknowledgements

The authors would like to thank the VikingPLoP 2003 shepherd Michael Pont, and all the people in the VikingPLoP 2003 workshop for their useful comments and suggestions.

The work presented here has been supported by the Centro di Eccellenza per le Radio Comunicazioni (CERCOM) at the Politecnico di Torino.

References

- [1] Cselt (Telecom Italia Lab). Java Reflective Broker Specifications. Available on-line: <http://andromeda.cselt.it/users/g/grasso/bin/jrb.zip>
- [2] Brugali D. and Menga G. Architectural models for global automation systems. In *IEEE Transactions on Robotics and Automation*, Vol. 18, No. 4, pp 487-493, 2002.
- [3] Gamma E., Helm R., Johnson R., and Villisides J. Design Patterns: Elements of Reusable Object oriented Software. Addison-Wesley, NY, 1994
- [4] Finin T., Labrou Y. and Mayfield J. KQML as an Agent Communication Language. In: Software Agents, J.M. Bradshaw (Ed.), Menlo Park, Calif., AAAI Press, 1997, pages 291-316.
- [5] Foundation for Intelligent Physical Agents (FIPA). "FIPA Communicative Act Library Specification". Doc. No. SC00037J, 03/12/2002. Available at: <http://www.fipa.org/specs/fipa00037J/>
- [6] Lamanna M. Una Piattaforma di Automazione Globale per l'interazione fra unita mobile autonome, Politecnico di Torino, September 2003.
- [7] Menga G., Elia G., and Mancin M. G++: An Environment for Object Oriented design and Prototyping of Manufacturing systems. In W. Gruver and G. Bordeaux, eds., *Intelligent Manufacturing: Programming Environments for CIM*. NY: Springer-Verlag. 1993.
- [8] Huber A.J., Huber J.F.: UMTS and mobile computing. Artech House, Inc, Norwood (2002).
- [9] Stemm M., Katz R.H.: Vertical handoffs in wireless overlay networks. In *Mobile Networks and applications*, Special Issue: mobile networking in the Internet, Vol 3, Issue 4, (1999) 335-350.
- [10] Sun: J2ME Generic Connection Framework. Available on-line: <http://java.sun.com>.
- [11] TILAB (Telecom Italia Lab). JADE (Java Agent DEvelopment Framework). Available on-line: <http://sharon.cselt.it/projects/jade/>
- [12] Weiser M.: The Computer for the 21st Century. In *Scientific American*, September (1991) 94-100.

A System of Patterns for Concurrent Request Processing Servers

Bernhard Gröne, Peter Tabeling

Hasso-Plattner-Institute for Software Systems Engineering
P.O. Box 90 04 60, 14440 Potsdam, Germany

{bernhard.groene, peter.tabeling}@hpi.uni-potsdam.de

Abstract

This paper addresses architectures of concurrent request processing servers, which are typically implemented using multitasking capabilities of an underlying operating system. Request processing servers should respond quickly to concurrent requests of an open number of clients without wasting server resources. This paper describes a small system of patterns for request processing servers, covering a relative wide range of architectures. Certain types of dependencies between the patterns are identified which are important for understanding and selecting the patterns. As the patterns deal with the conceptual architecture, they are mostly independent from programming languages and paradigms. The examples presented in this paper show applications of typical pattern combinations which can be found in productive servers. The pattern language is completed by a simple pattern selection guideline. The systematical compilation of server patterns, together with the guideline, can help both choosing and evaluating a server architecture.

1 Introduction

1.1 Application Domain

The patterns discussed in this paper focus on *request processing* servers which offer services to an *open number* of clients simultaneously.

Requests and Sessions.

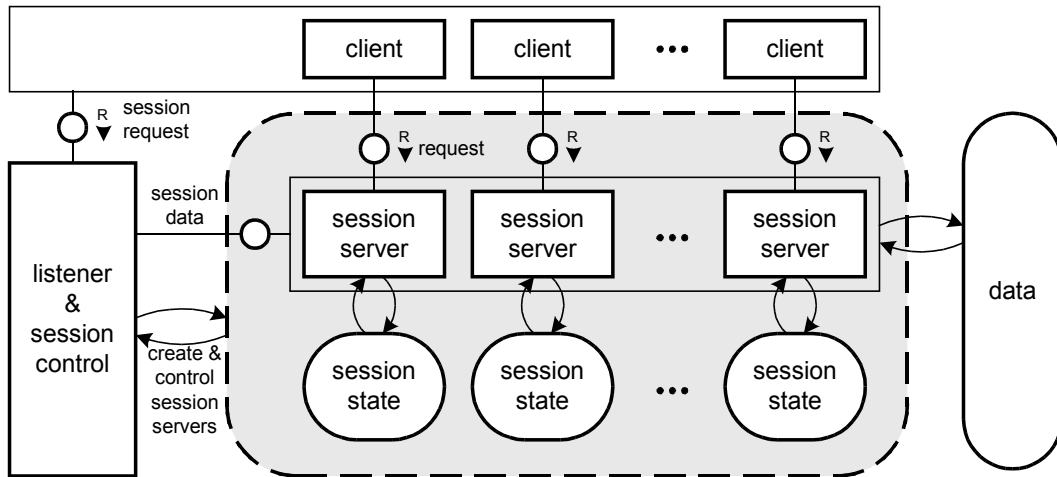


Figure 1 Clients and its corresponding servers (Block diagram legend: See figure 2)

Figure 1 presents an abstract conceptual model of the server type under consideration. For each client, it shows a dedicated *session server* inside the server. Each session server represents an abstract agent which exclusively handles requests of a corresponding client, holding the session-related state in a local storage. This abstract view leaves open if a session server is implemented by a task (process or thread) or not. A *session* starts with the first request of the client and repre-

sents the context of all further requests of the client until either the client or the server decides to finish it, which usually depends on the service and its protocol. Because each session server is only needed for the duration of the session it handles, session servers can be created and removed by the session controller on demand.

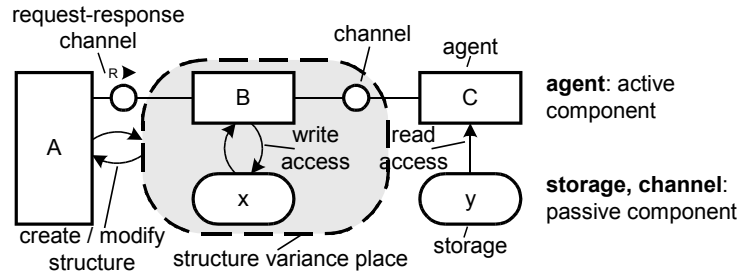


Figure 2 Legend of a block diagram

During a session, a client can send one or many requests to the server. For each request the server returns a response. In the following, we will exclusively focus on protocols where each client sends a sequence of requests and the server reacts with a response per request.

In this context, we have to distinguish between one-request-sessions and multiple-request-sessions. In case of *one-request-sessions*, the “session” is limited to the processing of only one single request, see figure 3. This is a typical feature of “stateless” protocols like HTTP. In contrast, a *multiple-request-session* like a FTP or a SMB session spans several requests, see figure 4. In this case, a session server repeatedly enters an idle state, keeping the session state for subsequent request processing until it is finally removed. If the client manages the session state, it sends the context to the server with every request, therefore the server can be simpler as it only has to manage single request sessions (The KEEP SESSION DATA IN THE CLIENT pattern in [Sore02]), but this solution has many drawbacks. In this paper, we will focus on the server's point of view of multiple request sessions which results in the KEEP SESSION DATA IN THE SERVER pattern [Sore02].

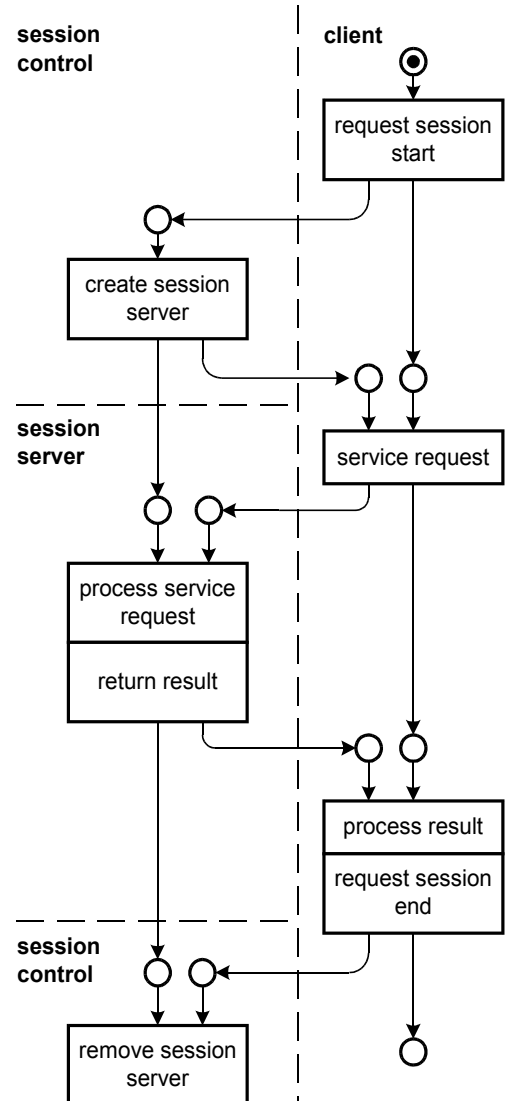


Figure 3 Single-request-session: Behavior of client, session server and session control (Petri Net)

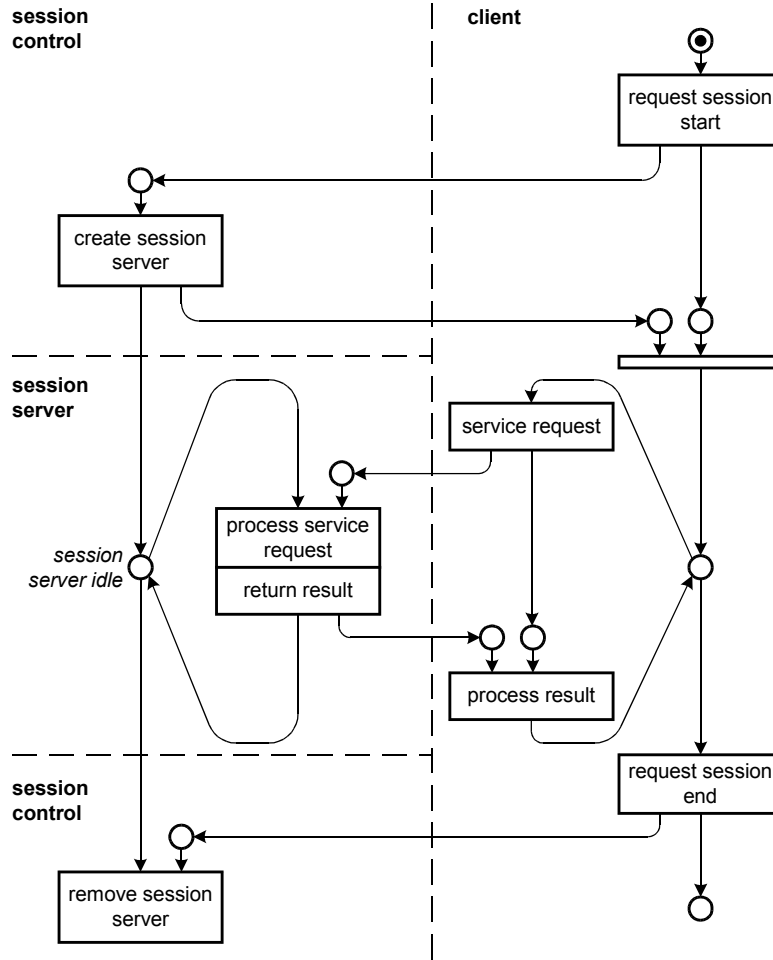


Figure 4 Multiple-request-session: Behavior of client, session server and session control (Petri Net)

Setting up Connections

If a dedicated connection between client and server (e.g. a TCP/IP connection) is used for communication during a session, the first request is a *connection request* sent by the client to the session controller. This request sets up the connection which can then be used to send *service requests* and return results. To set up the connection, the server has to accept the connection request. After establishing the connection, the TCP/IP service creates a new handle (socket) to access the new connection. The LISTENER / WORKER pattern shows this behavior in detail.

Multitasking

To serve multiple clients simultaneously, a server is usually implemented using processes or threads (tasks) of a given operating system. While the maximum number of concurrent clients is open in principle, it is actually constrained by the resource limitations of the server platform.

1.2 Forces

There are some forces which are common to all patterns described in this paper:

Response time. A network server should accept connection requests almost immediately and process requests as fast as possible. From the client's point of view, these time intervals matter:

1. Connect time (t_{conn}): The time from sending a connection request until the connection has been established.

2. First response time (t_{res1}): The time between sending the first request and receiving the response.
3. Next response times (t_{res2+}): The time between sending a subsequent request using an established connection and receiving a response.

The connect time t_{conn} usually should be short, especially for interactive systems. Minimizing the first response time t_{res1} is important for single-request sessions.

Limited Resources. Processes or threads, even when suspended, consume resources such as memory, task control blocks, database or network connections. Hence, it might be necessary to minimize the number of processes or threads.

Controlling the server. The administrator wants to shut down or restart the server without having to care for all processes or threads belonging to the server. Usually one of them is responsible for all other processes or threads of the server. This one receives the administrator's commands and may need some bookkeeping of the processes or threads belonging to the server.

1.3 Pattern Form

The pattern form used in this paper follows the Canonical Form. The graphical representations of architectural models are important parts of the solution's description. These diagrams follow the syntax and semantics of the Fundamental Modeling Concepts¹, while UML diagrams [UML] and code fragments are included as examples or hints for possible implementations. In order to reflect the cohesion within the pattern system, pattern dependencies (as discussed below) are explicitly identified.

1.4 Conceptual Focus of the Pattern Language

The patterns presented in this paper are not design patterns [GHJV94] in the narrow sense, i.e. they do not suggest a certain program structure such as a set of interfaces or classes. Instead, each pattern's solution is described as (part of) a *conceptual architecture*, i.e. as a dynamic system consisting of active and passive components without implying the actual implementation in terms of code elements written in some programming language. The resulting models mostly focus on the “conceptual view” or “execution view” according to [HNS99], but not the “module view” or “code view”. While this approach leaves some (design) burden for a developer, it allows presenting the patterns in their wide applicability – they are not limited to a certain programming paradigm, but in practice can be found in a variety of implementations, including non-object oriented systems.

Because the patterns are related to a common application domain, they form a system with strong dependencies between the patterns: A pattern language. In this paper, three basic dependency types are relevant:

Independent patterns share a common application domain but address different problems. Both can be applied within the same project.

Alternative patterns present different solutions for the *same* problem. Only one of the two patterns can be applied, depending on which of the force(s) comes out to be the *dominant* one(s). In case of such patterns, the dominant force(s) is/are identified as such and the alternative patterns are put in contrast.

Consecutive patterns: In this case, pattern B (the consecutive pattern) can only be applied if pattern A has already been applied/chosen, i.e. the resulting context of A is the (given) context of B. Such patterns are described in order of applicability (A first, then B) with B explicitly referencing to A as a pre-requisite. This aids sorting out “secondary” patterns which become relevant at later stages or won't be applicable at all.

¹ See [KTA+02],[KeWe03],[Tab02],[FMC]

2 A Pattern Language for Request Processing Servers

2.1 Overview

In the following, a system of seven patterns for request processing servers is presented. Table 1 and figure 5 give an overview. The Listener /Worker describes the separation of connection request and service request processing. Then there are two alternative task usage patterns, namely FORKING SERVER and WORKER POOL. For the latter, two alternative job transfer patterns are presented which are consecutive to the WORKER POOL pattern – JOB QUEUE and LEADER/FOLLOWER. As an additional, independent pattern, the SESSION CONTEXT MANAGER is discussed.

| Pattern Name | Problem | Solution |
|------------------------------------|--|---|
| LISTENER / WORKER | How do you use processes or threads to implement a server offering services for an open number of clients concurrently? | Provide different tasks for listening to and processing requests: One listener for connection requests and many workers for the service requests |
| FORKING SERVER | You need a simple implementation of the LISTENER / WORKER server. How can this server respond to an open number of concurrent requests in a simple way without using many resources? | Provide a master server listening to connection requests. After accepting and establishing a connection, the master server creates (forks) a child server which reads the request from this connection, processes the request, sends the response and terminates. In the meantime, the master server listens for connection requests again. |
| WORKER POOL | How can you implement a LISTENER / WORKER server providing a short response time? | Provide a pool of idle worker processes or threads ready to process a request. Use a mutex or another means to resume the next idle worker when the listener receives a request. A worker processes a request and becomes idle again, which means that his task is suspended. |
| WORKER POOL MANAGER | How do you manage and monitor the workers in a WORKER POOL? | Provide a WORKER POOL MANAGER who creates and terminates the workers and controls their status using shared worker pool management data. To save resources, the Worker Pool Manager can adapt the number of workers to the server load. |
| JOB QUEUE | How do you hand over connection data from listener to worker in a WORKER POOL server and keep the listener latency time low? | Provide a JOB QUEUE between the listener and the idle worker. The listener pushes a new job, the connection data, into the queue. The idle worker next in line fetches a job from the queue, reads the service request using the connection, processes it and sends a response back to the client. |
| LEADER / FOLLOWERS | How do you hand over connection data from listener to worker in a WORKER POOL server using operating system processes? How do you keep the handover time low? | Let the listener process the request himself by changing his role to "worker" after receiving a connection request. The idle worker next in line becomes the new listener while the old listener reads the request, processes it and sends a response back to the client. |
| SESSION CONTEXT MANAGER | How does a worker get the session context data for his current request if there are multiple-request-sessions and he just processed the request of another client? | Introduce a <i>session context manager</i> . Identify the session by the connection or by a session ID sent with the request. The session identifier is used by the session context manager to store and retrieve the session context as needed. |

Table 1: Pattern Thumbnails

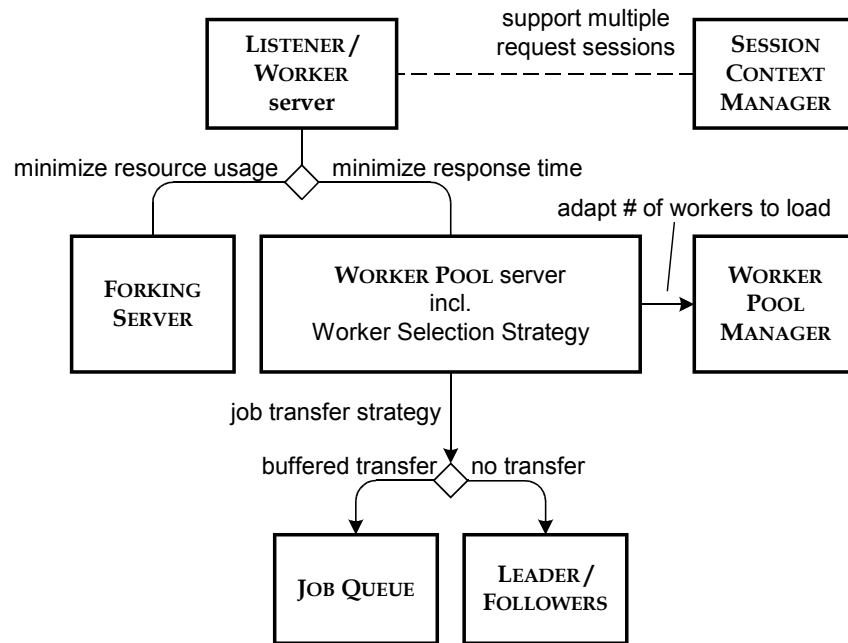


Figure 5 Overview of the Patterns in this document

2.2 Patterns for Request Processing Servers in Literature

For this domain, several patterns of this system have already been published in different forms. A good source is [POSA2] – in fact, most patterns described in this paper can be found in this book in some form. Some, like the LEADER/FOLLOWERS pattern, can be found directly, others only appear as variant (HALF-SYNC/HALF-REACTIVE) or are mentioned as part of one pattern although they could be considered as patterns of their own (like the THREAD POOL in LEADER / FOLLOWERS).

Apart from books, some pattern papers published for PLoP workshops cover aspects of request processing servers: The REQUEST HANDLER pattern [VKZ02] describes what client and server have to do to post and reply to requests in general. The POOLING [KiJa02a], LAZY ACQUISITION [KiJa02b] and EAGER ACQUISITION patterns describe aspects of the WORKER POOL mechanisms. The KEEP SESSION DATA IN SERVER and SESSION SCOPE Patterns [Sore02] are related to the SESSION CONTEXT MANAGER.

2.3 The Patterns

On the following pages, we will present the patterns as described above, each pattern starting on a new page. The guideline in section 2.4 gives hints when to choose which pattern.

Listener / Worker

Context

You want to offer services to an open number of clients using connection-oriented networking (for example TCP/IP). You use a multitasking operating system.

Problem

How do you use tasks (processes or threads) to implement a server offering services for an open number of clients concurrently?

Forces

- It is important to keep the time between connection request and establishing the connection small (connect time t_{conn}). No connection request should be refused as long as there is any computing capacity left on the server machine.
- For each server port there is only one network resource, a socket, available to all processes or threads. You have to make sure that only one task has access to a server port.

Solution

Provide different tasks for listening to connection requests and processing service requests:

- One *listener* listens to a server port and establishes a connection to the client after receiving a connection request.
- A *worker* uses the connection to the client to receive its service requests and process them. Many workers can work in parallel, each can process a request from a client.

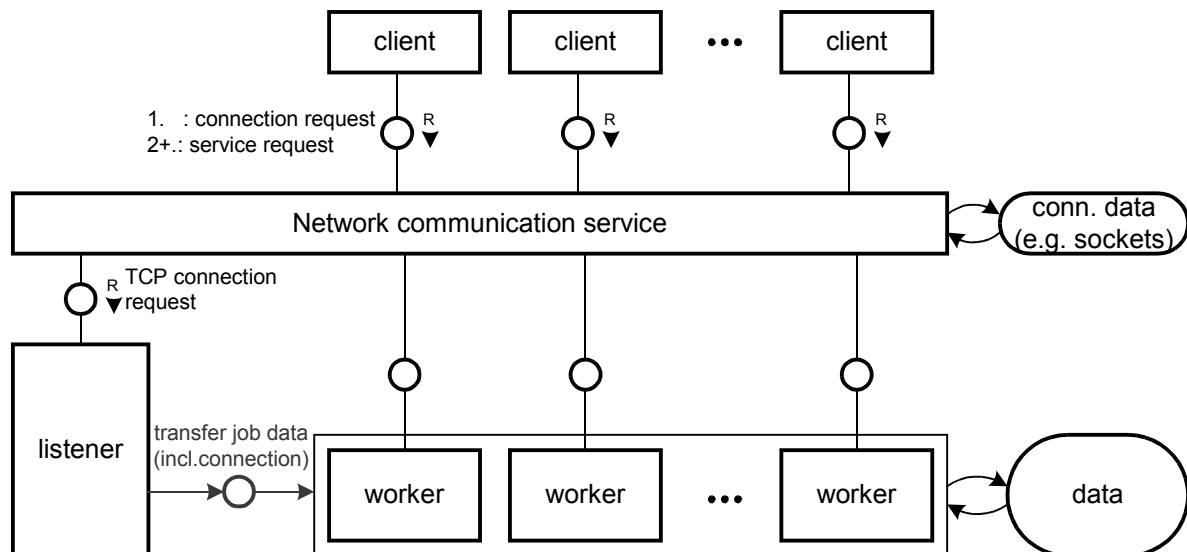


Figure 6 Listener / Worker pattern

Consequences

Benefits: As the listener's only task is to accept connection requests, the server is able to respond to connection requests quickly. Therefore it doesn't matter much that there is only one listener task per server port or even for all ports. A client sends its connection request and encounters

that the connection will be established quickly and that he is connected to a worker exclusively listening to his request.

Liabilities: Although it is obvious that the listener has to exist with the server's start, you can still decide when to create the workers. You can either choose the `FORKING SERVER` pattern where the listener creates the worker on demand, that is for every connection request. Or choose the `WORKER POOL` pattern and create the workers in advance.

Transferring the job data (in this case, just the connection to the client) from listener to worker must also be implemented. For this, the patterns `FORKING SERVER`, `JOB QUEUE` and `LEADER / FOLLOWERS` offer three different solutions.

Response time. There are 4 time intervals to be considered for a `LISTENER / WORKER Server`:

1. Listener response time (t_1): The time the listener needs after receiving a connection request until he establishes the connection.
2. Listener latency (t_2): The time the listener needs after establishing a connection until he is again ready to listen.
3. Connection handover (t_3): The time between the listener establishes the connection and the worker is ready to receive the service request using this connection.
4. Worker response time (t_4): How long it takes for the worker from receiving a request until sending the response.

The values of t_1 , t_2 and t_3 are only dependent from the multitasking strategy of the server, while t_4 is heavily dependent from the actual service request. All of them depend on the current server and network load, of course. Their effect on the response time intervals from the client's point of view (see section 1.2) is as follows: The listener needs t_1+t_2 for each connection request, so this sets his connection request rate and influences t_{conn} . The connection handover time t_3 is important for the first request on a new connection (t_{res1}); subsequent requests using this connection will be handled in t_4 .

Known Uses

Most request processing servers on multitasking operating system platforms use the `LISTENER / WORKER` pattern. Some examples: The `inetd`, `HTTP/FTP/SMB` servers, the `R/3` application server, database servers, etc.

Related Patterns

`FORKING SERVER` and `WORKER POOL` are two alternative consecutive patterns to address creation of the workers. `SESSION CONTEXT MANAGER` deals with the session data if workers should be able to alternately process requests of different sessions.

The `ACCEPTOR – CONNECTOR` pattern in [POSA2, p. 285] describes the separation of a server into acceptors and service handlers. The `REACTOR` pattern [POSA2, p. 179] is useful if one listener has to guard more than one port.

Forking Server

Context

You implement a `LISTENER / WORKER` server using tasks of the operating system.

Problem

You need a simple implementation of the `LISTENER / WORKER` server. How can this server respond to an open number of concurrent requests in a simple way without using many resources?

Forces

- Each operating system task (process or thread) consumes resources like memory and CPU cycles. Each unused and suspended task is a waste of resources.
- Transferring connection data (the newly established connection to the client) from listener to worker can be a problem if they are implemented with operating system processes.

Solution

Provide a master server listening to connection requests. After accepting and establishing a connection, the master server creates (forks) a child server which reads the service request from this connection, processes the request, sends the response and terminates. In the meantime, the master server listens for connection requests again. The forking server provides a worker task for each client's connection.

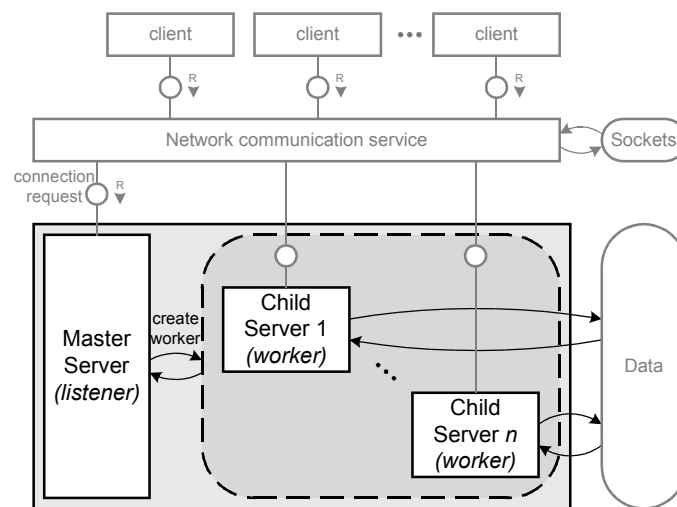


Figure 7 The forking server pattern

Figure 7 shows the runtime system structure of the `FORKING SERVER`. The structure variance area (inside the dashed line) indicates that the number of Child Servers varies and that the Master Server creates new Child Servers.

The master server task is the listener who receives connection requests from clients. He accepts a connection request, establishes a connection and then executes a “fork” system call which creates another task, a child server that also has access to the connection socket. While the listener returns to wait for the next connection request, the new child server uses the connection to receive service requests from the client. Figure 8 shows this behavior.

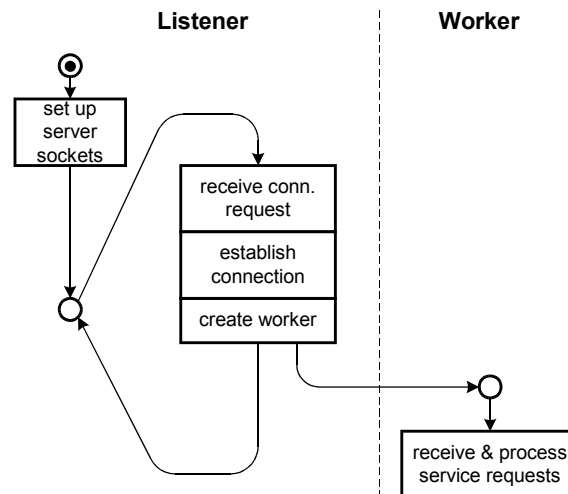


Figure 8 Behavior of the forking server

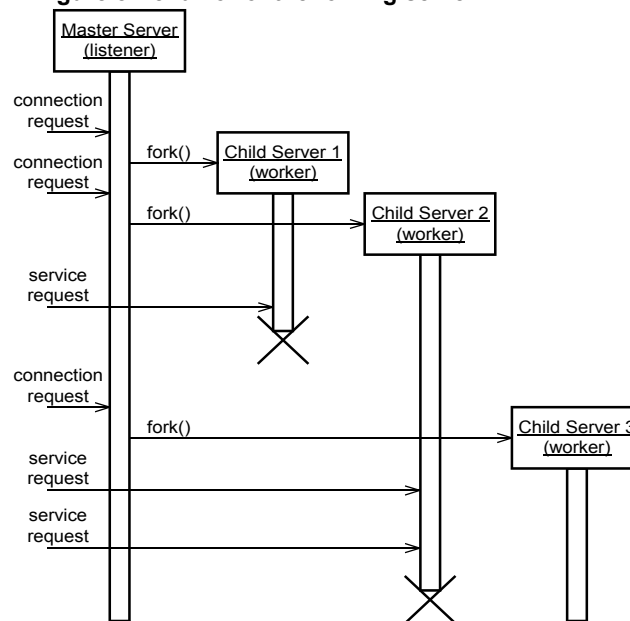


Figure 9 Example sequence of the forking server

Consequences

Benefits: The usage of server resources corresponds to the number of connected clients. A FORKING SERVER's implementation is simple:

- Connection handover: As `fork()` copies all process's data from parent to child process, this also includes the new connection to the client. If tasks are implemented with threads, it's even simpler because all threads of a process share connection handles.
- An idle FORKING SERVER needs very little system resources as it creates tasks on demand only.
- The Master Server only needs to know which workers are not terminated yet — this aids in limiting the total number of sessions (and therefore active workers) and makes shutting down the server quite simple.
- Handling multiple request sessions is easy because a worker can keep the session context and handle all service requests of a client exclusively until the session terminates.

Liabilities: A severe drawback of this kind of server is its response time. Creating a new task takes some time depending on the current server load. This will increase both the listener

latency time t_2 and the job handover time t_3 , which results in a bad connection response time and (first) request response time. If you need a more stable response time, use the WORKER POOL.

This also applies if you want to limit resource allocation and provide less workers than client connections. In this case you need a scheduler for the workers and a context management, for example the SESSION CONTEXT MANAGER.

Known uses

Internet Daemon. The Internet Daemon (inetd) is *the* prototype for the forking server which starts handlers for many different protocol types like FTP, Telnet, CVS — see section 3.1.

Samba smbd: Using the smbd, a Unix server provides Windows networking (file and printer shares) to clients. The Samba server forks a server process for every client.

Common Gateway Interface (CGI): An HTTP server which receives a request for a CGI program forks a new process executing the CGI program for every request.

Related Patterns

The WORKER POOL pattern offers an *alternative* solution, if a short response time is a critical issue. The SESSION CONTEXT MANAGER is an *independent* pattern which can be combined with FORKING SERVER, if a session spans multiple requests and it is not desirable to keep the according worker task for the complete session (for example, because the session lasts several hours or days).

The THREAD-PER-REQUEST pattern in [PeSo97] is very similar to the FORKING SERVER. The THREAD-PER-SESSION pattern in [PeSo97] describes the solution where session data is kept in the worker task instead of using a SESSION CONTEXT MANAGER.

Example Code

```
while (TRUE) {
    /* Wait for a connection request */
    newSocket = accept(ServerSocket, ...);
    pid = fork();
    if ( pid == 0 )
    {
        /* Child Process: worker */
        process_request(NewSocket);
        exit(0);
    }
    [...]
}
```

Worker Pool

Context

You implement a `LISTENER / WORKER` server using tasks of the operating system.

Problem

How can you implement a `LISTENER / WORKER` server providing a short response time?

Forces

- To minimize the listener latency time t_2 , you have to make sure that the listener is quickly ready to receive new connection requests after having established a connection. Any actions that could block the listener in this phase increase the listener latency time.
- Creating a new worker task after establishing the connection increases the connection handover time t_3 .

Solution

Provide a pool of idle worker tasks ready to process a request. Use a mutex or another means to resume the next idle worker when the listener receives a request. A worker processes a request and becomes idle again, which means that his task is suspended.

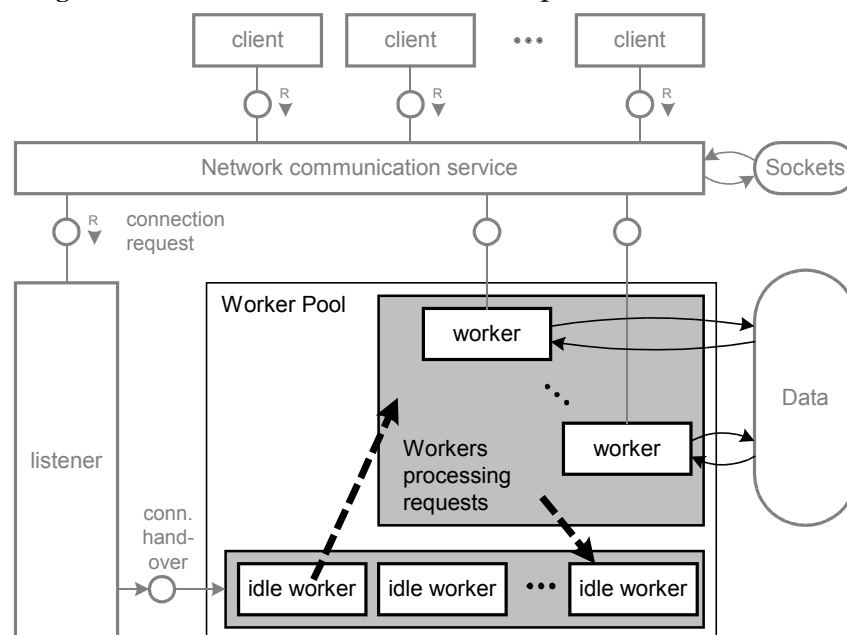


Figure 10 The Worker Pool pattern

Additionally, the strategy in choosing the next worker can be customized to gain better performance. The straightforward way is using a mutex. This usually results in a FIFO order or a random order, depending on the operating system resource in use. Another strategy could implement a LIFO order to avoid paging of task contexts.

The `WORKER POOL` pattern is a good solution if the server's *response time* should be minimized and if it can be afforded to keep processes or threads “alive” in a pool between requests. In contrast to the `FORKING SERVER`, this avoids creating a process or thread for every session or request, which increases the response time.

Consequences

Benefits:

- As the worker tasks are created in advance, the time to create a task doesn't affect the response time anymore.
- You can limit the usage of server resources in case of a high server load. It is even possible to provide less workers than clients.

Liabilities:

- You need a way to hand over the connection data from the listener to an existing worker. This strategy will affect the listener latency time t_2 and the connection handover time t_3 . The two alternatives to do so are the `JOB QUEUE` and the `LEADER / FOLLOWERS` patterns.
- A static number of workers in the pool might be a problem for a varying server load. To adapt the number of workers to the current server load, use a `WORKER POOL MANAGER`.

Known Uses

Apache Web Server. All variants of the Apache HTTP server use a `WORKER POOL`. Most of them use a `WORKER POOL MANAGER` to adapt the number of workers to the server load. See section 3.2 for further details.

SAP R/3. The application server architecture of SAP R/3 contains several so-called “work processes” which are created at a server's start-up and stay alive afterwards to process requests. Usually there are less work processes in the pool than clients. As R/3 sessions usually span multiple requests, the work processes use a `SESSION CONTEXT MANAGER`. See section 3.3 for a more detailed description.

Related Patterns

The `FORKING SERVER` pattern is an *alternative* pattern which minimizes resource consumption but increases response time. `WORKER POOL MANAGER` is a consecutive pattern which provides a manager to control the workers. The `SESSION CONTEXT MANAGER` is an *independent* pattern which can be combined with `WORKER POOL` if a session spans multiple requests. `JOB QUEUE` and `LEADER / FOLLOWERS` are consecutive patterns which deal with the transfer of job-related data from listener to worker.

A detailed description of a thread pool can be found in [SV96] and in the `LEADER / FOLLOWERS` pattern in [POSA2, p. 450ff]. It is also mentioned in a variant of the `ACTIVE OBJECT` pattern [POSA2, p. 393].

The `POOLING` Pattern [KiJa02a] describes more general how to manage resources in a pool. The creation of idle worker tasks at start-up is an example of `EAGER ACQUISITION` [KiJa02b].

Worker Pool Manager

Context

You have applied the WORKER POOL pattern.

Problem

How do you manage and monitor the workers in a WORKER POOL?

Forces

- At the server start, a certain number of worker tasks (processes or threads) have to be created before the first request is received.
- To shut down the server, only one task should receive the shutdown signal which will then tell the others to shutdown too.
- If a worker dies, he must be replaced by a new one.
- Workers consume resources, so there should be a strategy to adapt resource usage to the server load without reducing server response time.

Solution

Provide a WORKER POOL MANAGER who creates and terminates the workers and controls their status using shared worker pool management data.

To save resources, the Worker Pool Manager can adapt the number of workers to the server load: If the number of idle workers is too low or no idle worker available, he creates new workers. If the number of idle workers is too high, he terminates some idle workers.

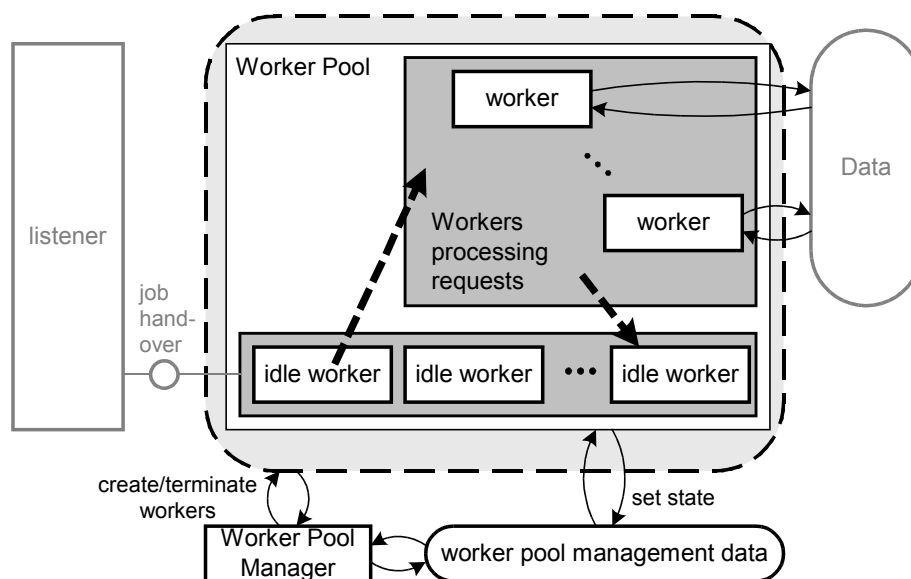


Figure 11 Worker Pool with manager

Figure 11 shows a WORKER POOL and its manager: The worker pool manager creates all tasks in the pool. For every task in the pool, it creates an entry in the worker pool management data. This storage is shared by the workers in the pool. Whenever an idle worker is activated or a worker becomes idle, it changes its state entry in the worker pool management data. The worker pool manager can count the number of idle and busy tasks and create new tasks or terminate

idle tasks, depending on the server load. Additionally it can observe the “sanity” of the workers. If one doesn't show any life signs anymore or terminates because of a crash, the manager can replace it with a new one.

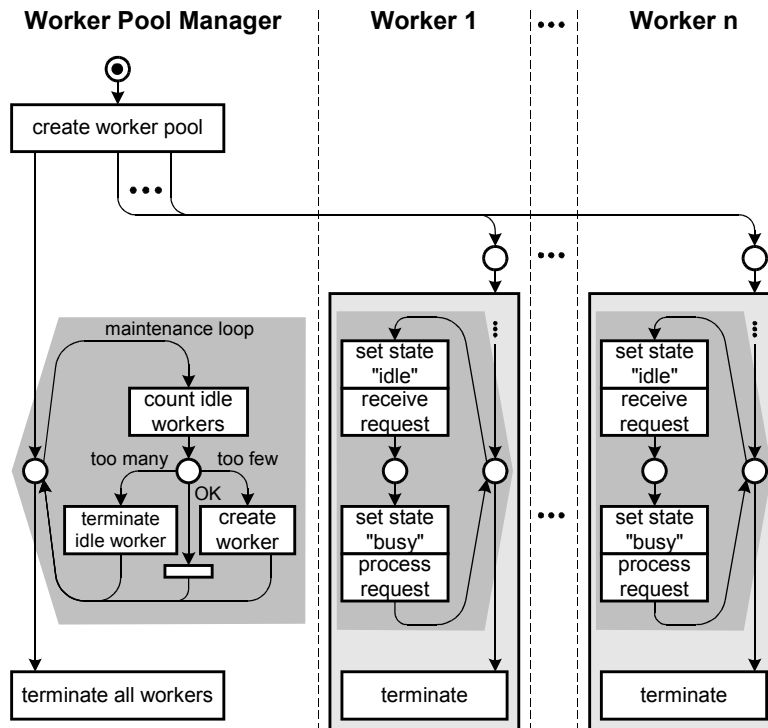


Figure 12 Interaction between Worker Pool Manager and Workers

Figure 12 shows the behavior of the worker pool manager and the workers: After creating the workers, the worker pool manager enters the maintenance loop where he counts the number of idle workers and terminates or creates workers, depending on the limits. The workers set their current state information in the worker pool management data.

Don't implement the WORKER POOL MANAGER in the same task as the listener, or you'll get the same problems as with the FORKING SERVER: Creating a new process may take some time which may increase the listener latency time t_2 dramatically.

Figure 13 shows an example sequence where the worker pool manager replaces a worker which terminated unexpectedly.

Consequences

Benefits: The Worker Pool Manager takes care of the workers and makes it easier to shutdown a server in a well – defined way. By constantly monitoring the status of the workers, he helps to increase the stability of the server. If he controls the number of workers depending on the current server load, he helps to react to sudden changes in the server load and still keeps resource usage low.

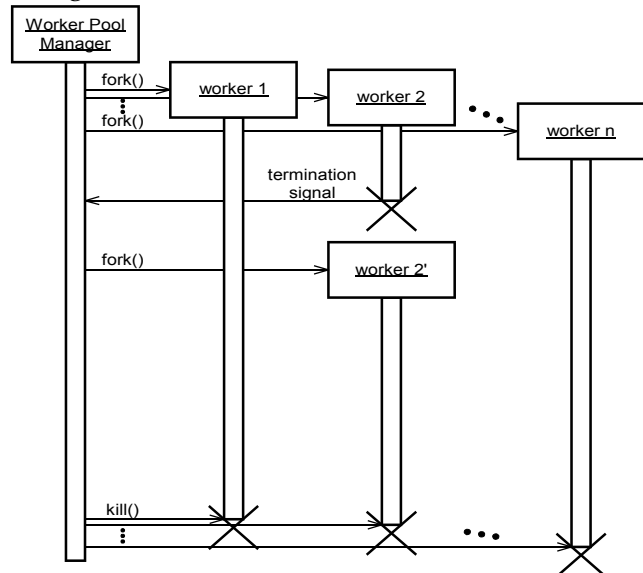


Figure 13 Example sequence: Worker Pool Manager creates, replaces and terminates workers

Liabilities: A worker now has to update its state entry in the worker pool management data storage whenever he enters another phase. This storage should be a shared memory. It is important to avoid blocking the workers while they update their state.

The Worker Pool Manager is an additional task which consumes resources as it has to run regularly.

Known Uses

Apache Web Server. All Apache MPMs have a dedicated process for worker pool management. Only the Windows version has a static worker pool while all other variants let the worker pool manager adapt the number of workers in the pool to the current server load.

SAP R/3 Dispatcher. The Dispatcher in an R/3 system manages the worker pool: He starts, shuts down and monitors the work processes and can even change their role in the system, depending on the current server load and the kind of pending requests.

Example Code

This example code has been taken from the Apache HTTP server (see section 3.2) and adapted to stress the main aspects of the pattern.

The `child_main()` code is not shown here as it depends on the job transfer strategy (JOB QUEUE or LEADER / FOLLOWERS). Each worker updates his state in the scoreboard as shown in figure 12.

```
make_child(slot) {
[...]
    pid = fork();
    if ( pid == 0 )
    {
        /* Child Process: worker */
        scoreboard[slot].status = STARTING;
        child_main(slot);
        exit(0);
    }
[...]
    scoreboard[i].pid = pid;
}

manage_worker_pool() {
[...]
    scoreboard = create_scoreboard();
    /* create workers */
    for (i = 0 ; i < to_start ; ++i ) {
        make_child(i);
    }
    for (i = to_start ; i < limit ; ++i ) {
        scoreboard[i].status = DEAD;
    }

    /* Control Loop */
    while(!shutdown) {
        /* wait for termination signal or let some time pass */
        pid = wait_or_timeout();
        if ( pid != 0 ) {
            /* replace dead worker */
            slot = find_child_by_pid(pid);
            make_child(slot);
        }
        else {
            /* check number of workers */

```

```
idle_count = 0;
for (i = 0 ; i < limit ; ++i ) {
    if (scoreboard[i].status == IDLE)
    {
        idle_count++;
        to_kill = i;
    }
    if (scoreboard[i].status == DEAD)
        free_slot = i;
}
if (idle_count < min_idle) {
    make_child(free_slot);
}
if (idle_count > max_idle) {
    kill(scoreboard[to_kill].pid);
    scoreboard[to_kill].status = DEAD;
}
}
} /* while(!shutdown) */

/* Terminate server */
for (i = 0 ; i < limit ; ++i ) {
    if (scoreboard[i].status != DEAD)
        kill(scoreboard[i].pid);
}
}
```

Job Queue

Context

You have applied the WORKER POOL pattern to implement a LISTENER / WORKER server.

Problem

How do you hand over connection data from listener to worker in a WORKER POOL server and keep the listener latency time low?

Forces

- To decrease the listener latency time t_2 , the listener should not have to wait for a worker to fetch a new job. This happens when the listener has just established a connection to the client and needs to hand over the connection data to a worker.

Solution

Provide a JOB QUEUE between the listener and the idle worker. The listener pushes a new job, the connection data, into the queue. The idle worker next in line fetches a job from the queue, reads the service request using the connection, processes it and sends a response back to the client.

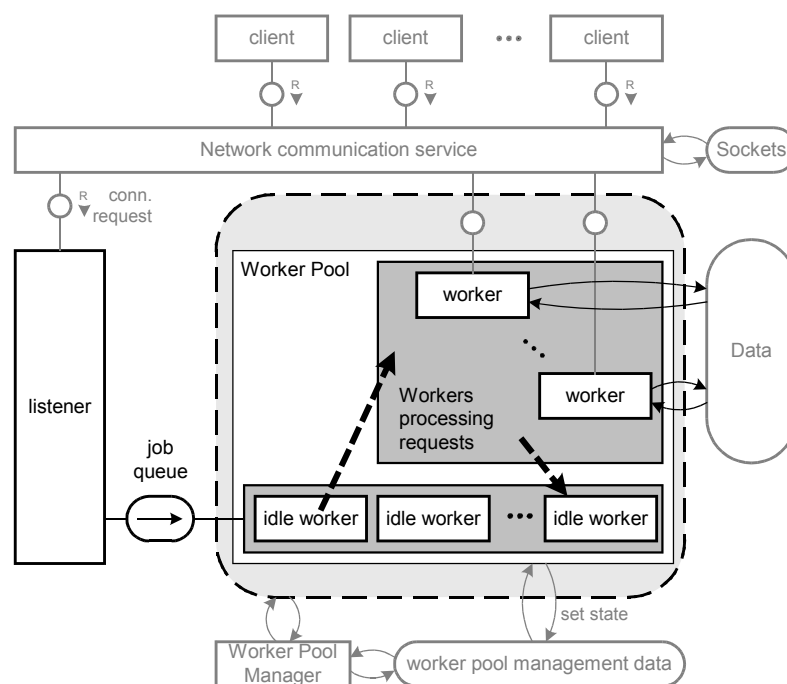


Figure 14 The job queue pattern

One or many listeners have access to the network sockets, either one listener per socket or one for all, as shown in figure 14. Instead of creating a worker task (like the Forking Server), he puts the connection data into the job queue and waits for the next connection request, see figure 15. All idle workers wait for a job in the queue. The first one to fetch a job waits for and processes service requests on that connection. After that, he becomes idle again and waits for his next job in the queue.

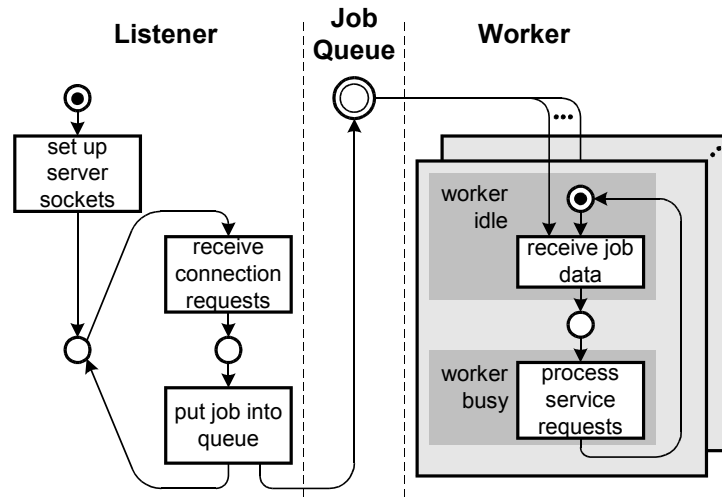


Figure 15 Behavior of Listener and Worker using a job queue

Consequences

Benefits: The listener just has to push a job into a queue and then return to listen again. The listener latency time t_2 is therefore low.

Liabilities: The job handover time t_3 is not optimal as the operating system has to switch tasks between listener, worker and queue mutex. When using operating system processes, the JOB QUEUE is not applicable, as there is no way to transfer a socket file descriptor (corresponding to the connection to the client) between two processes. In both cases, use the LEADER / FOLLOWERS pattern.

Known Uses

Apache Web Server. The Windows version (since Apache 1.3) and the WinNT MPM implement the JOB QUEUE with a fix number of worker threads. The worker MPM uses a JOB QUEUE on thread level (inside each process) while the processes concurrently apply for the server sockets using a mutex (section 3.2).

SAP R/3. On each application server of an R/3 system, requests are placed into a queue. These requests are removed from the queue by the “work proceses” for job processing, see section 3.3.

Related Patterns

JOB QUEUE is applicable as *consecutive* pattern to the WORKER POOL pattern. LEADER / FOLLOWERS is an *alternative* pattern which does not introduce a queue and avoids the transfer of job-related data between tasks.

The HALF-SYNC / HALF REACTIVE pattern in [POSA2, p.440] describes the mechanism to decouple the listener (asynchronous service, reacting to network events) from the workers (synchronous services) using a message queue combined with the thread pool variant of the ACTIVE OBJECT pattern [POSA2, p. 393]. A description of the THREAD POOL with JOB QUEUE can be found in [SV96], including an evaluation of some implementation variants (C, C++ and CORBA).

Example Code

This example only shows the code executed in the listener and worker threads. The creation of the threads and the queue is not shown here.

The *job queue* transports the file descriptor of the connection socket to the workers and servers as a means to select the next worker.

Listener Thread:

```
while(TRUE) {  
    [...]   
    /* wait for connection request */  
    NewSocket = accept(ServerSocket, ...);  
    /* put job into job queue */  
    queue_push(job_queue, NewSocket);  
}
```

Worker Thread:

```
while (TRUE) {  
    /* idle worker: wait for job */  
    scoreboard[myslot].status = IDLE;  
    [...]   
    ConnSocket = queue_pop(job_queue);  
  
    /* worker: process request */  
    scoreboard[myslot].status = BUSY;  
    process_request(ConnSocket);  
}
```


Leader / Followers

Context

You have applied the **WORKER POOL** pattern to implement a **LISTENER / WORKER** server.

Problem

How do you hand over connection data from listener to worker in a **WORKER POOL** server using operating system processes? How do you keep the handover time low?

Forces

- To access a new connection to the client, a task has to use a file descriptor which is bound to the process. It is not possible to transfer a file descriptor between processes.
- Switching tasks (processes or threads) between listener and worker increases the connection handover time t_3 .

Solution

Let the listener process the service request himself by changing his role to "worker" after receiving a connection request. The idle worker next in line becomes the new listener while the old listener reads the service request, processes it and sends a response back to the client.

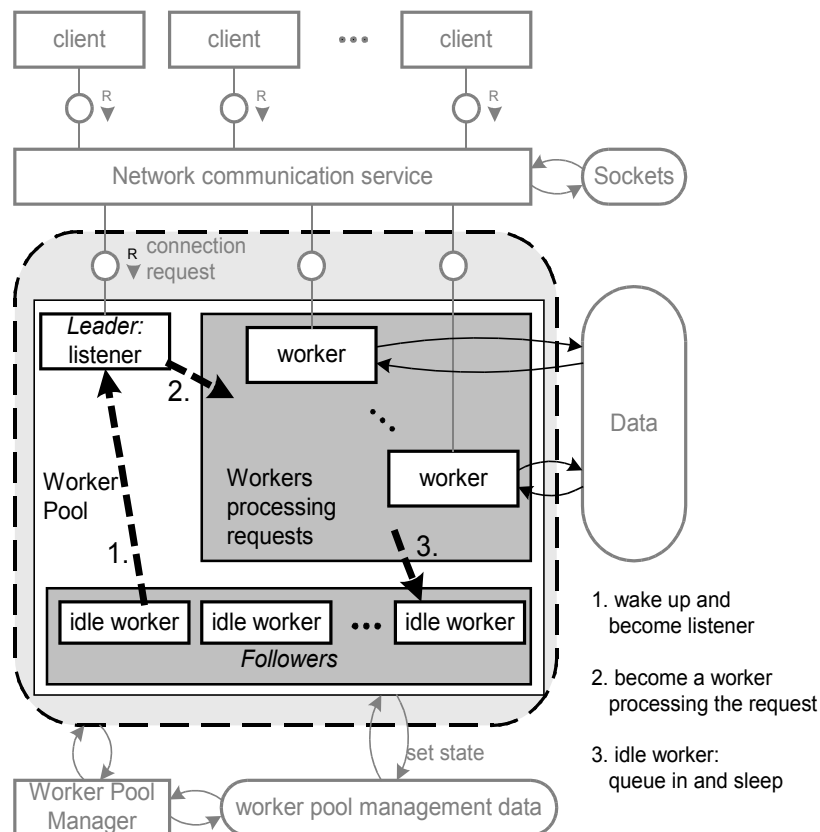


Figure 16 The Leader / Followers pattern

All tasks in the worker pool transmute from listener to worker to idle worker eliminating the need for a job queue: Using a mutex, the idle workers (the followers) try to become the listener (the leader). After receiving a connection request, the listener establishes the connection, releases

the mutex and becomes a worker processing the service request he then receives. Afterwards, he becomes an idle worker and tries to get the mutex to become the leader again. Hence, there is no need to transport information about the connection as the listener transmutes into a worker task keeping this information.

Figure 16 shows the structure: The processes or threads in the WORKER POOL have 3 different states: worker, idle worker and listener. The listener is the one to react to connection requests, while workers and idle workers process service requests or wait for new jobs, respectively.

The corresponding dynamics are shown in figure 17. Initially, all tasks of the pool are idle workers. Listener selection is done by a task acquiring the mutex which is released as soon as the listener changes his role to become a worker.

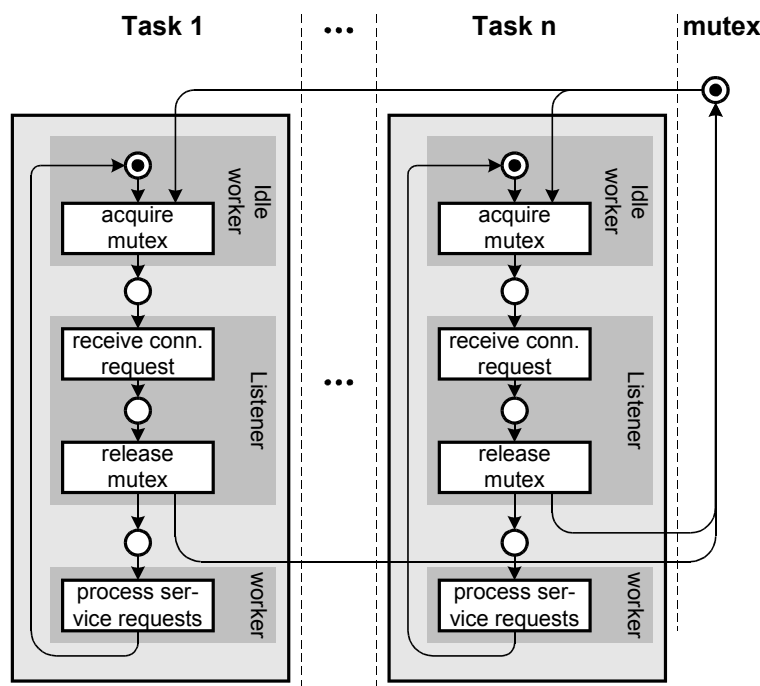


Figure 17 Behavior of the tasks in the Leader / Followers pattern

Consequences

Benefits: The listener changes his role by just executing the worker's code. This keeps the connection handover time t_3 very low as every information remains in this task. As file descriptors needed to get access to the connection to the clients don't leave the task, the LEADER / FOLLOWERS pattern enables the use of operating system processes to implement the WORKER POOL pattern.

Liabilities: The election of the next listener is handled via a mutex or a similar mechanism. This requires a task switch and leads to a non-optimal listener latency time t_2 . The LEADER / FOLLOWERS pattern avoids transferring job-related data, but introduces the problem of dynamically changing a process' or thread's role; for example, the server sockets must be accessible to all workers to allow each of them to be the listener. Both listener and worker functionality must be implemented inside one task.

Known Uses

Apache Web Server. A very prominent user of LEADER / FOLLOWERS pattern is the preforking variant of the Apache HTTP server (see section 3.2).

Call center. In a customer support center, an employee has to respond to support requests of customers. The customer's call will be received by the next available employee.

Taxi stands. Taxis form a queue at an airport or a train station. The taxi at the top of the queue gets the next customers while the others have to wait.

Related Patterns

LEADER / FOLLOWERS is applicable as *consecutive* pattern to the WORKER POOL pattern. JOB QUEUE is an *alternative* pattern which uses a queue for job transfer between tasks with static roles.

The LEADER / FOLLOWERS pattern has originally been described in [POSA2], p. 447.

Example code:

```
while (1) {
    /* Become idle worker */
    scoreboard[myslot].status = IDLE;
    [...]
    acquire_mutex(accept_mutex);
    /* Got mutex! Now I'm Listener! */
    scoreboard[myslot].status = LISTENING;
    newSocket = accept(ServerSocket, ...);
    [...]
    /* Become worker ... */
    release_mutex(accept_mutex);
    /* ... and process request */
    scoreboard[myslot].status = BUSY;
    process_request(NewSocket);
}
```

Session Context Manager

Context

You implement a `LISTENER / WORKER` server for multiple-request sessions.

Problem

If a worker processes service requests from different clients and a session can contain multiple requests, how does a worker get the session context data for his current service request?

Forces

Keeping session context data within a worker can be a problem:

- If a worker task is assigned exclusively for one session, it is unable to handle requests from other clients. This is usually a waste of resources and interferes with limiting the number of workers.
- You have to consider that the connection to the client may be interrupted during the session without terminating the session. The same applies to the worker task which can die unexpectedly. Therefore you might need to save and resume a session state.

Solution

Introduce a *session context manager*. Identify the session by the connection or by a session ID sent with the request. The session identifier is used by the session context manager to store and retrieve the session context as needed.

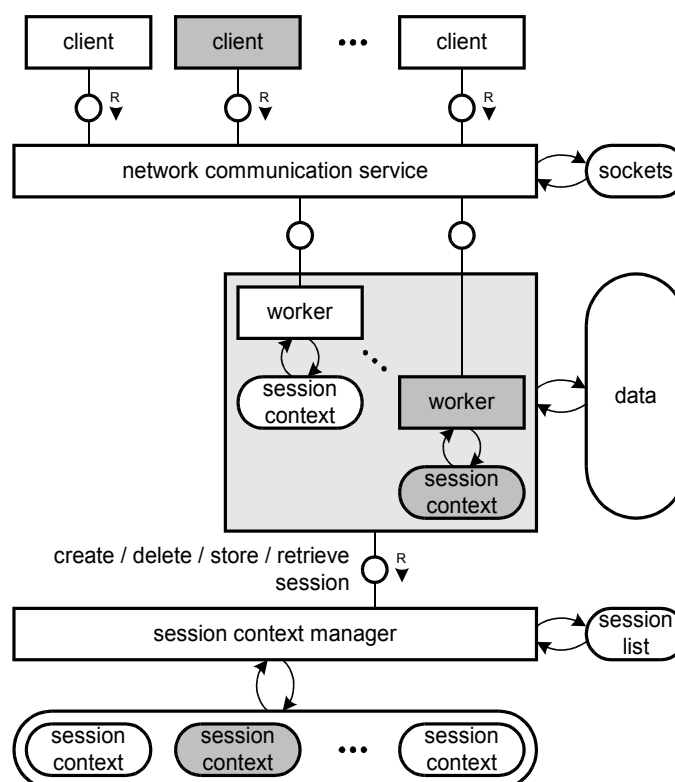


Figure 18 Session Context Manager (Central Manager Variant).

This is a specialized variant of the MEMENTO pattern [GHJV94]. Figure 18 shows a solution using a central session context manager: Each worker has a local storage for a session context. Before he processes a request, he asks the context manager to retrieve the session context corresponding to the client's session using the session ID extracted from the request. After processing the request, he asks the context manager to store the altered session context. In case of a new session, he has to create a new session context and assign a session ID to it. In case of the last request of a session, the worker has to delete the session context. The session context shaded grey in figure 18 belongs to the grey client. The grey worker currently processes a request of this client and works on a copy of its session context.

Figure 19 shows how to extend the behavior of the worker to support session context management. An example sequence is shown in figure 20.

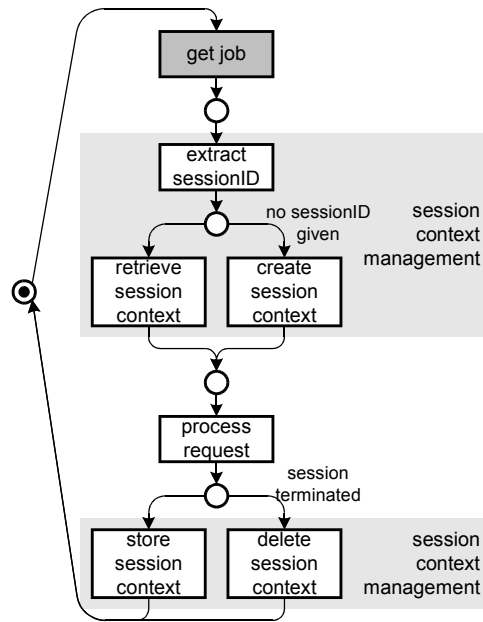


Figure 19 Session Context Management in the worker loop

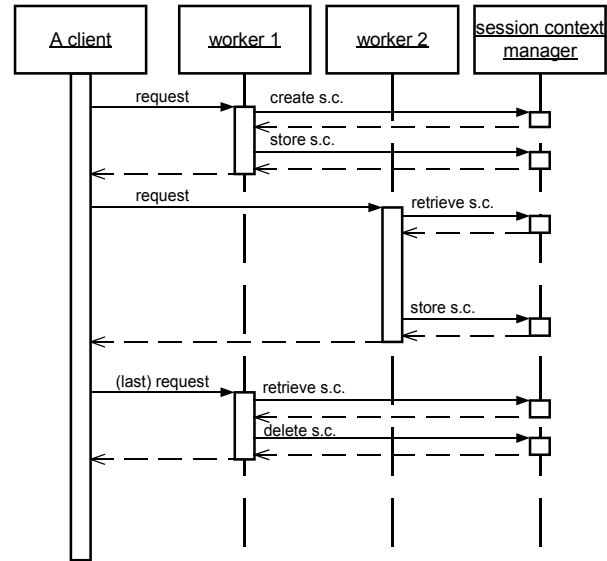


Figure 20 Example sequence for session context management

Variants. *Central Session Context Manager:* There is a single context manager for all tasks. If, for example, the session is bound to the connection, the listener not only reacts to connection requests but also to requests on open connections. The functionality of the session context manager can then be included in the listener.

Local Session Context Manager: Each worker manages the sessions and the session list. The functionality of the session context manager in figure 18 is then included in every worker task. The storage for the session contexts is shared between all workers.

Consequences

Benefits.

- Any worker can process a request in its session context. This enables an efficient usage of workers, especially in a WORKER POOL server.
- If the session ID is sent with every request, the connection can be interrupted during the session. This is useful for long sessions (from a few minutes to several days).
- Using a dedicated context manager helps separating the request-related and context-related aspects of request processing. For each request to be processed, session context management requires a sequence of (1) retrieving (creating) a session context, (2) job processing and (3) saving (deleting) the context.

Liabilities.

- A garbage collection strategy might be needed to remove session contexts of “orphan” sessions.
- Session context storage and retrieval increases the response time.

Known Uses

SAP R/3. Each application server of an SAP R/3 system contains workers, each of them having its own local session context manager (the so-called taskhandler, see section 3.3).

CORBA portable object adapter. CORBA-based servers can use objects (not tasks) as workers similar to a Worker Pool. In such configurations the so-called object adapters play the role of session context managers, see section 3.4.

CGI applications. An HTTP server starts a new CGI program for every request, like the `FORKING SERVER`. The CGI program extracts the session ID from the request (for example by reading a cookie) and then gets the session context from a file or database.

Related Patterns

The pattern is *consecutive* to `FORKING SERVER` or `WORKER POOL`. To realize access to session contexts of workers, the `MEMENTO` pattern [GHJV94] could be used. In case of local context management, `TEMPLATE METHOD` [GHJV94] could be applied to enforce the retrieve-process-save sequence for each request. The `KEEP SESSION DATA IN SERVER` and `SESSION SCOPE` Patterns [Sore02] describe session management in a more general context.

2.4 Guideline for Choosing an Architecture

In the following, a simple guideline for selecting patterns from the pattern system is presented. It helps deriving a server architecture by choosing a pattern combination appropriate for the intended server usage. The guideline presents the patterns according to their dependencies and fosters pattern selection by questions aiming at dominant forces.

1. Clarify the basic context for LISTENER / WORKER.
 - Is the server's type a *request processing server* or a different one, e.g. a streaming server? Should threads or processes provided by the operating system be used, including IPC mechanisms? If not, the pattern system might be not appropriate.
 - Does a session span *multiple requests*? Then consider 4.
2. Select the task usage pattern.

Is saving *resources* more important than minimizing the *response time*? If yes, choose a FORKING SERVER. If not, apply the WORKER POOL pattern instead.
3. When using a WORKER POOL,
 - Choose a Job Transfer pattern. Is *transferring job data* between tasks easier than *changing their role*? If yes, introduce a JOB QUEUE. If not, apply the LEADER/FOLLOWER pattern.
 - Does the number of concurrent requests vary in a wide range? Then use a dynamic pool instead of a static pool. In this case the WORKER POOL MANAGER dynamically adapts the number of workers to the server load.
 - Choose a *strategy* to select the next worker task from the idle worker set: FIFO, LIFO, priority-based, indetermined.
4. If a session spans multiple requests:
 - Does the number of concurrent sessions or their duration allow to keep a worker task exclusively for the session? If not, introduce a SESSION CONTEXT MANAGER.
 - Can the listener retrieve the session ID of a client's request? Then choose central context management, else local.

The following table shows that the possible pattern combinations yield six basic architecture types:

| | | <i>without</i> SESSION CONTEXT MANAGER | <i>with</i> SESSION CONTEXT MANAGER |
|----------------|-----------------|---|-------------------------------------|
| FORKING SERVER | | <i>Type 1</i> inetd, samba server, CGI | <i>Type 2</i> CGI applications |
| WORKER POOL | JOB QUEUE | <i>Type 3</i> Apache (Win32, Worker) | <i>Type 4</i> SAP R/3 |
| | LEADER/FOLLOWER | <i>Type 5</i> Apache (Preforking) | <i>Type 6</i> |

Table 2 Architectures covered by the pattern system

3 Example Applications

3.1 Internet Daemon

The Internet Daemon (`inetd`) is a typical representative of the FORKING SERVER without SESSION CONTEXT MANAGER (*Type 1*).

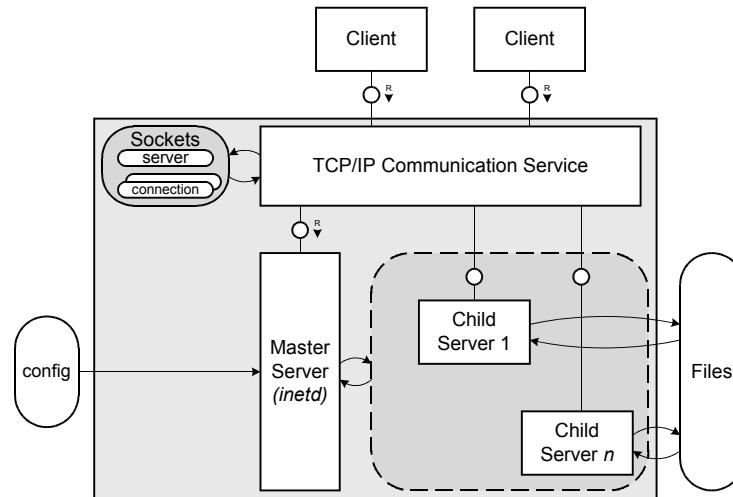


Figure 21 The `inetd` — a typical FORKING SERVER

Figure 21 shows the structure of the `inetd` server: It waits for requests on a set of TCP ports defined in the configuration file `/etc/inetd.conf`. Whenever it receives a connection request, it starts the server program defined for this port in the configuration file which handles the request(s) of this connection. The `inetd` starts a server program by the `fork()` - `exec()` sequence which creates a new process and then loads the server program into the process. The file descriptor table is the only data which won't be deleted by `exec()`. A server program to be started by the `inetd` must therefore use the first file descriptor entry (#0) to access the connection socket.

3.2 Apache HTTP Server

The Apache HTTP server [Apache] is a typical request processing server which has been ported to many platforms. The early versions use the *preforking* server strategy as described below. Since version 1.3, Apache supports the Windows™ platform using threads which forced another server strategy (JOB QUEUE).

Apache 2 now offers a variety of server strategies called MPMs (Multi-Processing Modules) which adapts Apache to the multitasking capabilities of the platform and may offer different server strategies on one platform. The most interesting MPMs are:

- Preforking (*Type 5*):
LEADER / FOLLOWERS using processes with dynamic worker pool management. The promotion of the followers is done with a mutex (results in a FIFO order).
- WinNT (*Type 3*):
JOB QUEUE using a static thread pool.
- Worker MPM (*Type 3* on thread level):
Each process provides a JOB QUEUE using a static thread pool. The process pool is dynamically adapted to the server load by a WORKER POOL MANAGER (Master Server). The listener threads of the processes use a mutex to become listener. (see section 3.2)

All MPMs use a WORKER POOL with processes or threads or even nest a thread pool in each process of a process pool. They separate the listener from the WORKER POOL MANAGER. A so-called

Scoreboard is used to note the state of each worker task. Most Worker Pool managers adapt the number of worker tasks to the current server load.

A detailed description of the Apache MPMs and of their implementation can be found in the Apache Modeling Project [AMP].

The Preforking MPM of Apache

Since its early versions in 1995, Apache uses the so-called Preforking strategy — a LEADER / FOLLOWERS pattern. A master server starts (forks) a set of child server processes doing the actual server tasks: listen for connection requests, process service requests. The master server is responsible for adjusting the number of child servers to the server load by assuring that the number of idle child servers will remain within a given interval.

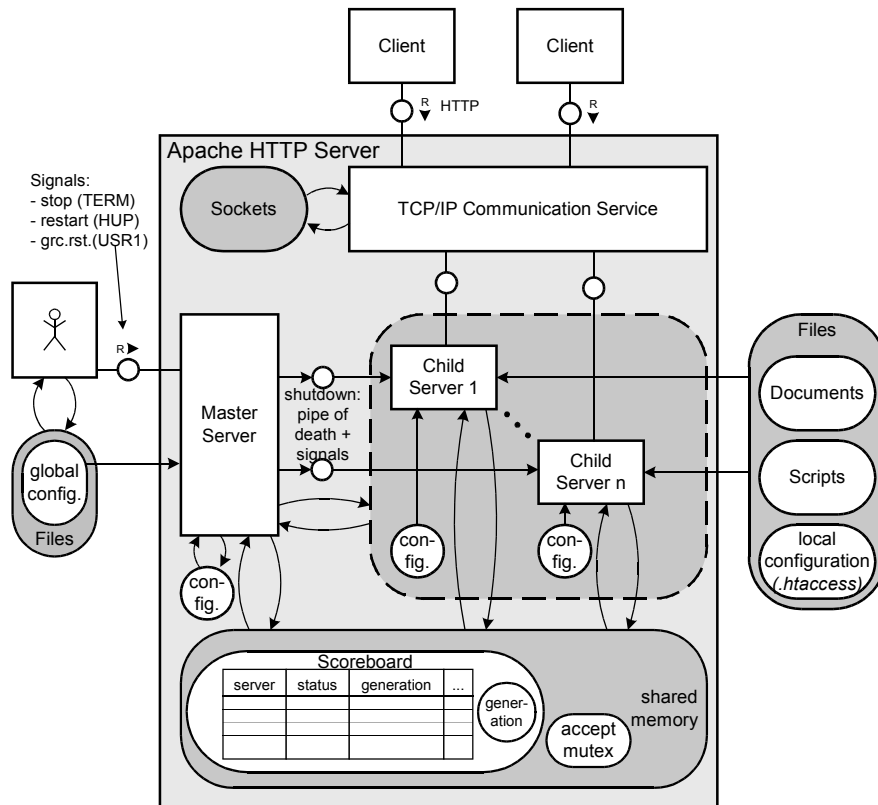


Figure 22 The Apache server using the Preforking MPM

Figure 22 shows the conceptual architecture of a preforking Apache server. At the top we see the clients, usually web browsers, sending HTTP requests via a TCP connection. HTTP is a stateless protocol, therefore there's no need to keep a session context. At the right-hand side we see the data to be served: Documents to be sent to the client or scripts to be executed which produce data to be sent to the client. The Scoreboard at the bottom keeps the Worker Pool management data as mentioned in the WORKER POOL pattern.

The master server is not involved in listening or request processing. Instead, he creates, controls and terminates the child server processes and reacts to the commands of the administrator (the agent at the left side). The master server also processes the configuration files and compiles the configuration data. Whenever he creates a child server process, the new process gets a copy of this configuration data. After the administrator has changed a configuration file, he has to advise the master server to re-read the configuration and replace the existing child servers with new ones including a copy of the new configuration data. It is possible to do this without interrupting busy child servers: The master server just terminates idle child servers and increments the generation number in the scoreboard. As every child server has an entry including its gene-

ration in the scoreboard, it checks after each request if its generation is equal to the current generation and terminates otherwise.

The child servers use a mutex to assign the next listener, according to the LEADER / FOLLOWERS pattern. In contrast to figure 16, the different roles of the workers in the pool are not shown.

The Worker MPM of Apache

Figure 23 shows the system structure of the Apache HTTP server using the worker MPM. The center shows multiple child server processes which all have an identical structure. They are the tasks of the WORKER POOL on process level, like in the preforking MPM. Inside each Child Server Process we find an extended JOB QUEUE structure: A listener thread waits for connection requests and supplies a job queue. (Idle) worker threads wait for new jobs. The idle worker queue signals to the listener if there is an idle worker ready to process the next job. If there is none, the listener doesn't apply to the accept mutex.

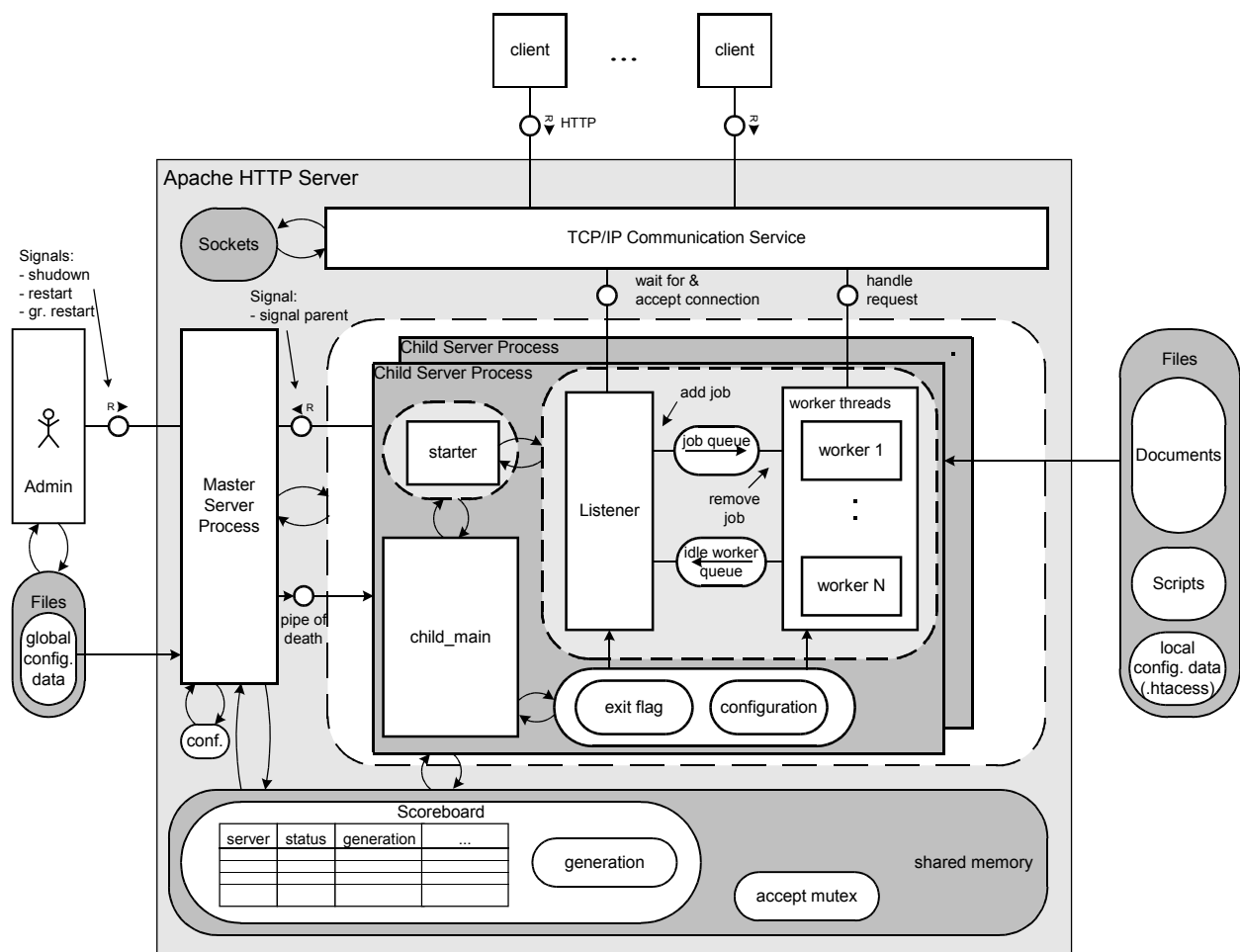


Figure 23 The Apache server using the Worker MPM

The child_main thread creates the listener and worker threads by creating the starter thread (which terminates after setting up listener, workers and the queues) and after that just waits for a termination token on the 'pipe of death'. This results in setting the "exit flag" which is being read by all threads.

Only one process' listener gets access to the server sockets. This is done by applying for the accept mutex. In contrast to the LEADER / FOLLOWERS pattern, the listener task doesn't change his role and processes the request. Instead, he checks if there is another idle worker waiting and applies for the accept mutex again.

3.3 SAP R/3

SAP's System R/3 [SAP94] is a scalable ERP system with an overall three tier architecture as shown in figure 24. The diagram also shows the inner architecture of an R/3 application server. (In practice, R/3 installations often include additional applications servers – these are omitted here for simplicity reasons.) The application server can be categorized as a *Type 4* server (cp. Table 2), because three of the patterns discussed above are actually applied.

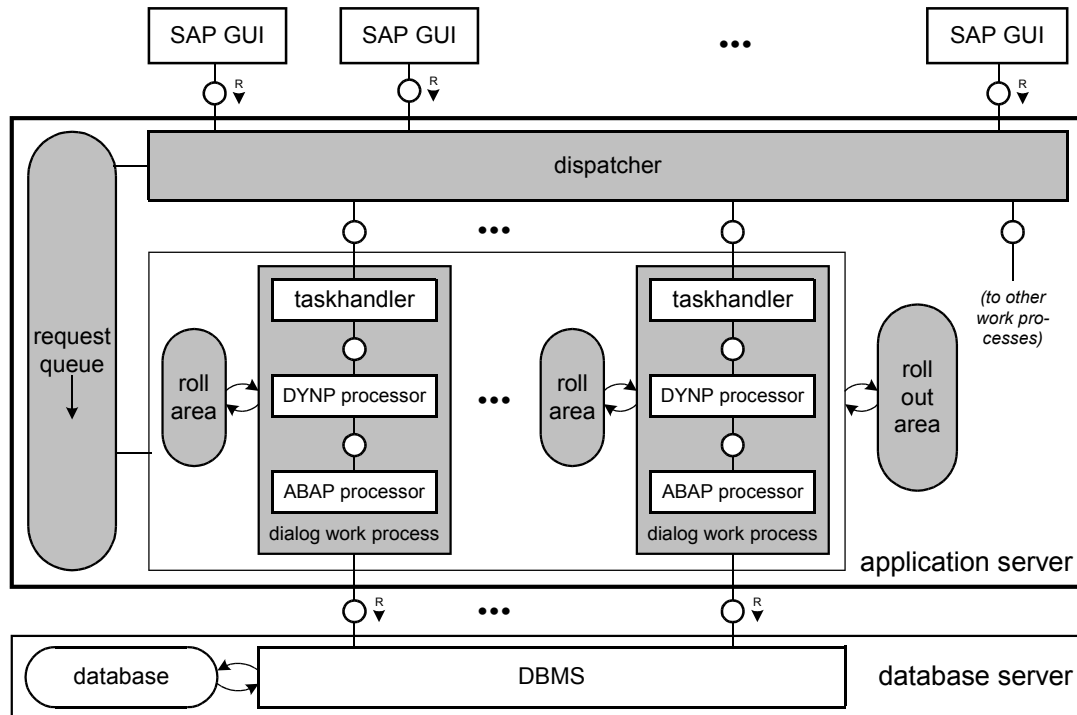


Figure 24 SAP R/3 Application Server Architecture

The server's basic architecture is designed after the *WORKER POOL* pattern. The so-called dispatcher (a process) plays the role of the listener and forwards requests (sent by the "SAP GUI" clients) to worker tasks called "dialog work processes". (There are further types of work processes, but these are of no interest here.)

Beside other purposes, the request queue is used for forwarding requests from the dispatcher to the work processes, i.e. it represents a *JOB QUEUE*. While the work processes read request data from the queue by themselves, initial worker selection is actually done by the dispatcher.

Because an R/3 session (called a *transaction* in SAP's terms) spans multiple requests, a local *SESSION CONTEXT MANAGER* called *taskhandler* is included in each *work process*. Before a request can be processed by the DYNP processor and the ABAP processor, the taskhandler retrieves the appropriate context from the *roll out area* (or creates a new one) and stores it in the work process' *roll area*. Afterwards, the taskhandler saves the context to the roll out area (at session end, it would be deleted).

3.4 Related Applications at Object Level

All of the above patterns have been discussed under the assumption of processes or threads being the building blocks for server implementations. However, some of the patterns are even applicable if we look at a system at the level of *objects* instead of tasks. When realizing a server's request processing capabilities by a set of objects, these objects can be "pooled" similar to a *WORKER POOL*. Instead of creating objects for the duration of a session or a single request, "worker objects" are kept in a pool and activated on demand. This helps controlling resource consumption and avoids (potentially) expensive operations for creating or deleting objects.

For example, this idea has been put into practice with Sun's J2EE server architecture [J2EE]. Here, the "stateless session beans" are kept in a pool while the so-called "container" plays the listener role, activating beans on demand and forwarding requests to them.

Another example are the so-called "servants" from the CORBA 3.0 portable object adapter specification [CORBA]. These are server-side worker objects which can also be kept in a pool managed by the "object adapter" (if the right "server retention policy" has been chosen, see the POA section in [CORBA]). The object adapter (in cooperation with an optional "servant manager") does not only play the listener role — it also acts as SESSION CONTEXT MANAGER for the servants.

4 Conclusion and Further Research

Design patterns in the narrow sense are often discussed in a pure object-oriented context. Hence, they often present object-oriented code structures as solution, typically classes, interfaces or fragments of methods. In contrast, the patterns presented in this paper are conceptual patterns which deliberately leave open most of the coding problem. This initially seems to be a drawback, but it also widens the applicability of a pattern and increases the possibility to identify a pattern within a given system. In fact, most of the industrial applications (known uses) examined in this paper are not implemented with an object-oriented language (although some OO concepts can be found). Furthermore, central ideas and topics (e.g. scheduling, task and session management) behind the patterns have already been described in the literature about transaction processing systems [GrRe93].

Designing a good code structure is often a secondary problem with additional forces such as given languages, frameworks or legacy code. In order to remain "paradigm-neutral", conceptual architecture patterns should be presented using appropriate notations like FMC. Object-Oriented implementation of conceptual architecture models is an important research topic in this context [TaGr03].

The integrated description of pattern systems has been developed together with the pattern system presented here. Further research is necessary to prove that this is applicable to pattern systems in general. This approach, backed by corresponding guidelines, may support software architects in applying patterns.

5 Acknowledgements

We wish to thank Uwe Zdun who gave us many valuable hints as our shepherd, and the participants in the writer's workshop, especially Neil Harrison, who gave us lots of very useful advice and suggestions for improvement.

References

- [Apache] Apache Group, *The Apache HTTP Server*, <http://httpd.apache.org>
- [AMP] Bernhard Gröne et al., *The Apache Modeling Project*, <http://apache.hpi.uni-potsdam.de>
- [CORBA] *The Common Object Request Broker Architecture*, version 3.0.2, The Object Management Group, 2002
- [FMC] Siegfried Wendt et al: *The Fundamental Modeling Concepts Home Page*, <http://fmc.hpi.uni-potsdam.de/>
- [GHJV94] Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1994
- [GrRe93] J. Gray, A. Reuter, *Transaction Processing: Concepts and Techniques*, Morgan Kaufmann, 1993
- [HNS99] Christine Hofmeister, Robert Nord and Dilip Soni, *Applied Software Architecture*, Addison Wesley, 1999
- [J2EE] *Java 2 Platform: Enterprise Edition*, Sun Microsystems Inc., <http://java.sun.com/j2ee>
- [KeWe03] Frank Keller, Siegfried Wendt, *FMC: An Approach Towards Architecture-Centric System Development*, IEEE Symposium and Workshop on Engineering of Computer Based Systems, Huntsville, 2003

- [KiJa02a] Michael Kircher and Prashant Jain, *Pooling*, EuroPloP 2002
- [KiJa02b] Michael Kircher and Prashant Jain, *Eager Acquisition*, EuroPloP 2002
- [KTA+02] Frank Keller, Peter Tabeling, Rémy Apfelbacher, Bernhard Gröne, Andreas Knöpfel Rudolf Kugel and Oliver Schmidt, *Improving Knowledge Transfer at the Architectural Level: Concepts and Notations*, International Conference on Software Engineering Research and Practice (SERP), Las Vegas, 2002
- [Lea97] D. Lea, *Concurrent Programming in Java*, Addison-Wesley, 1997
- [POSA1] F. Buschmann et al., *Pattern-Oriented Software Architecture: A System of Patterns*, Wiley, 1996
- [POSA2] D. Schmidt et al., *Pattern-Oriented Software Architecture: Patterns for Concurrent and Networked Objects*, Wiley, 2000
- [PeSo97] Dorina Petriu and Gurudas Somadder, *A Pattern Language For Improving the Capacity of Layered Client/Server Systems with Multi-Threaded Servers*, EuroPloP 1997
- [SAP94] *Introduction to Concepts of the R/3 Basis System*, SAP AG (Basis Modelling Dept.), 1994
- [Sore02] Kristian Elob Sørensen, *Session Patterns*, EuroPloP 2002
- [SV96] Douglas C. Schmidt and Steve Vinoski: *Object Interconnections: Comparing Alternative Programming Techniques for Multi-threaded Servers* (Column 5-7). SIGS C++ Report Magazine, Feb.-Aug. 1996.
- [Tabe02] Peter Tabeling, *Ein Metamodell zur architekturorientierten Beschreibung komplexer Systeme*, Modellierung 2002, GI-Edition - Lecture Notes in Informatics (LNI) - Proceedings, 2002
- [TaGr03] Peter Tabeling, Bernhard Gröne, *Mappings Between Object-oriented Technology and Architecture-based Models*, International Conference on Software Engineering Research and Practice, Las Vegas, 2003
- [UML] G. Booch et al., *The Unified Modeling Language User Guide*, Addison Wesley, 1998
- [VKZ02] Markus Völter, Michael Kircher, Uwe Zdun, *Object-Oriented Remoting – Basic Infrastructure Patterns*, VikingPloP 2002
- [VSW02] M. Völter, A. Schmid, E. Wolff, *Server Component Patterns*, Wiley, 2002

A Pattern Language for Standardization Work

Juha Pärssinen
juha.parssinen@vtt.fi

Abstract

This paper presents a pattern language for people who are going to participate in standardization work. In this paper, the technical details of communication protocols or other standardization artifacts are not the primary focus. Working on standardization means working in a heterogeneous group, and decisions that are made need to balance conflicting forces. The resulting solutions tend to be compromises from proposals made by one or more participants of the standardization effort. This pattern language gives advice to readers to reach the best possible compromise.

Introduction

This paper presents a pattern language for people who are participating in standardization work. It gives advice to readers who are not professional consensus makers (or politicians), but technical experts who participate in standardization work, and are the main responsible persons of their company.

There are four typical cases why an individual want to participate in standardization: to learn a standard as early as possible; to investigate what others have invented; to protect own work to avoid major changes (to protect investments to implementation); to delay or even stop the standardization work. This language doesn't consider the last case, which can be seen as a non-generative activity. Readers are advised to read Machiavelli [2] if they want to master also those aspects of politics.

The idea of this pattern language occurred to the author when he was writing with his co-author their previous pattern language for the specification of communication protocols [3].

Patterns in this language are presented in the sequence they are usually applied. *Compromises Anyway* is a starting-point for this language. It sets the context for all patterns. Other patterns are in two groups: patterns used before standardization meetings and patterns used during standardization meetings.

Compromises Anyway : Accept the fact that you (and hopefully other participants too) have to be ready to make compromises.

Before standardization meeting

Do Your Homework: Prepare well before meeting. Different phases of preparation are divided to three pattern: *Know Things*, *Know People* and *Know Yourself*.

Know Things: Start from pure technical values, and leave the rest for a moment. Study in advance all relevant technical aspects of standardization.

Know People: Study carefully the people who participate in standardization work in advance. In this pattern, people or participants can be considered as individuals and as the whole company.

Know Yourself: Know yourself well before you need to make any decisions. In this pattern “knowing yourself” means knowing not only you, but also the company or the organization you are working for.

Back-up team ready: You need to be able to alert your back-up team during the meetings.

During standardization meeting

Concepts over Names: If needed, ease your demands a little: keep the concept as it is, but accept a new name for it.

Power of Examples: Examples make things easier to understand.

Use Straw Poll: Straw poll can be used during meetings to see how things are going, and to avoid formal voting.

Pattern language

Compromises Anyway

The goal of standardization work is a standard that can be accepted by participants of a working group and at least majority of the voters of the concerned standardization body. If a standardization project is working in an area that is very far away from commercial products, or the area is so new that none has done anything, this is the rare case when the work can be based purely on solving emerging technical questions.

However, in the real world typically at least some of the participants have something important for them to push and/or protect during standardization work. They have already implemented first prototypes, and they want the standard to reflect what they have implemented to protect their investment and also benefit to be first in the market.

Therefore:

The only way to participate in standardization work and get it done is to accept the fact that you (and hopefully other participants too) have to be ready to make compromises. There are typically two ways to make a compromise: give-and-take and take-all. Of course, in real life standards, there can be sections that use give-and take and sections that use take-all.

In a give-and-take compromise every participant needs to make trade-offs, they work together towards a single standard, starting possibly from several conflicting initial proposals. Standards made using this approach are typically smaller, easier to understand, and more consistent than those made using take-all approach. The author strongly recommends give-and-take compromise for standardization work.

Unfortunately, it is not always possible to make trade-offs, and for this reason a take-all compromise is used. In a take-all compromise several conflicting initial proposals are put together, but they are not harmonized as in a give-and-take compromise. Conflicting parts can be hidden as alternative sections or as optional sections of the standard. A take-all compromise should be used only if the other choice is a long (possibly infinite) delay of the standard.

In this pattern language some advice is given to the reader for reaching the best give-and-take compromise possible. However, these patterns are useful if the take-all approach is taken. Patterns appear in two groups; those used before the meeting and those used during the meeting.

Before the Meeting

Do Your Homework

In standardization work, as in any other creative activity, people are in the middle of a swift information flow. They need to calculate the net effect of many conflicting forces of all technical aspects of concern. For most of the people, it is not possible to evaluate these things correctly when they are met the first time. Typically time and advice are needed from other experts to find a relevant solution.

Therefore:

Do your homework. Homework in this particular case is a plan or a procedure explaining what you want to achieve and avoid during standardization work. It contains information about particular points which are negotiable and which are not. To do your homework, collect information about all relevant things in advance, and spend time to analyze them with your colleagues. This includes also debriefing of the previous meetings. Homework has to be done well; this fact was written down by Sun Tzu, a general from ancient China:

“If you know the enemy and know yourself, you need not fear the result of a hundred battles. If you know yourself but not the enemy, for every victory gained you will also suffer a defeat. If you know neither the enemy nor yourself, you will succumb in every battle.”

Sun Tzu

As it was in elementary school, a written document is much better homework than any other.

“The strongest memory is no match to the palest ink.”

Chinese Proverb

You should spend time to make your plan, one made during the flight to the meeting place is usually made too late. You need time to read reference material and to talk with people in your home organization. You also need time to make things clear for yourself.

People who don't do their homework (properly) very often follow strategy: “If I don't say anything, I will appear to understand everything”. However, this is only a short term solution. Sooner or later there will be time to decide and then you won't have anything you can use to proceed wisely. Other people can guide and potentially mislead you if they want. If you like the concept of an anti-pattern, this could be one.

There are several aspects which you should sort out before participating in the meeting, these are considered one by one in the next patterns. These aspect are divided into three patterns following one of Sun Tzu's most quoted sentences:

“Hence the saying: If you know the enemy and know yourself, your victory will not stand in doubt; if you know Heaven and know Earth, you may make your victory complete.”

Sun Tzu

In the previous quotation, the enemy can be the other participants or people, yourself (obviously) is yourself, heaven (explained as a climate in [1]) as a techno-political situation, and earth as a technology in standardization area. Heaven (techno-politics) and earth (technology itself) together define domain for standardization.

In the following three patterns these four aspects are explained in the standardization context. *Know Things* pattern contains earth, *Know People* pattern contains enemy, and *Know Yourself* pattern contains yourself. These patterns will help you to do your homework. Heaven is divided between *Know People* and *Knowing Yourself*.

Know Things

It does not matter whether a standardization project is starting from a white sheet or there is a lot of existing prework, in any case you need to understand (and possibly solve) emerging technical questions during standardization work.

As explained in the *Do Your Homework* pattern, standardization domain contains two parts: heaven and earth. You see this clearly when you are evaluating any technical issue. There are always two sides to take into consideration: “engineering” values (earth) e.g. feasibility and performance, and “techno-political” values (heaven) e.g. who owns patents and who has invented technology. If you try to understand and evaluate both sides at once it can be overwhelming.

Therefore:

Start from pure technical values, and leave the rest for a moment. Study all relevant technical aspects of standardization in advance. You don't have to be an expert in everything, but you should know enough to understand the documents and to be able to participate in discussions. However, at the same time when you are studying technical information, collect also such information as who has invented things, who owns possibly patents, and which companies use those. That information is used in next patterns, *Know People* and *Know Yourself*.

A good example of this pattern is 3rd generation mobile phone standardization. There are two technologies strive to better utilize the radio spectrum by allowing multiple mobile phone users to share the same physical channel: TDMA (Time Division Multiple Access) and CDMA (Code Division Multiple Access). They have several fundamental technical differences, but also one (among others) interesting techno-political difference: one single company owns majority of CDMA patents. If you follow this pattern, you study TDMA and CDMA from engineering viewpoint. And make note for yourself about interesting patent issue.

Know People

During your studies of technical aspects in *Know Things* pattern, you have also collected related non-technical information. In this pattern, people or participants can be considered as an individual or as the whole company.

In a standardization body there are typically several participants from different interest groups which usually have conflicting goals. To do you homework properly, it is important to know other participant's goals as early as possible:

“By discovering the enemy's dispositions and remaining invisible ourselves, we can keep our forces concentrated, while the enemy's must be divided.”

Sun Tzu

You can estimate that result of work will be a compromise made by people, but you want to push it as much as possible in one direction and avoid other ones.

Therefore:

Study carefully the people who participate in standardization work. You should know participants' work history, e.g. what kind of education they have, what publications they have written, and have they participated in this kind of work before. Usually people are kind enough to tell all this in their WWW home page. If you have time skim through their publications. It is also important to know what kind of character the person is. The only reliable way to find out this is to get to know them, other people's opinions are not so trustworthy.

If the inventors of technical aspects in concern participate in work, they typically defend their work rigorously. People tend to think that what they have done is like their child, and nobody else than themselves are allowed to change it or really understand it. You should also know who are the real leaders of the work to know to whom you should talk to make any kind of progress.

Study also the participating companies, e.g. which are their core technology areas and which are their business segments. For example a manufacturer typically has a completely different kind of interests than a service provider. You can expect companies to try to lead the work in direction which is most important to them.

If we continue our TDMA vs. CDMA example from the previous pattern, now it is time for you to analyze effects of patents and other non-engineering values of each technology. In the previous pattern you make a note that one single company owns majority of CDMA patents. For this reason it can be expected that this company wants CDMA technology to be used in standard.

It is also important to know why people (or companies) participate in this particular project. There are four typical cases: they want to learn standard as early as possible; they want to investigate what others have invented; they want to protect their own work to avoid major changes (to protect their investments to implementation); they want to delay or even stop the standardization work because they have their own reasons not to have standard in this area. What to do with this information of course depends on your own goal.

Know Yourself

During your studies of technical aspects in *Know Things* pattern, you have also collected related non-technical information. In this pattern "knowing yourself" means knowing not only you, but also the company or the organization you are working for.

When you are preparing yourself for the meeting you are not only studying technical issues, you are also learning the strategy of your company. Information about technical issues is useless if you don't know motive for decisions and value of artifacts behind of them.

Therefore:

Read technical documents made in your own company and interview people who wrote them. Interview also people who are (or will be) implementors of prototypes, or end-user products. They typically have a lot of opinions, and quite often they are not documented anywhere.

Read also strategy documents and interview people who have wrote them. Strategy documents usually explain long-term goals of company, but do not explain why these are chosen.

An example of a typical case is that your company have a prototype ready, and as much as possible should be re-used in a final product. In this case you need to know enough details of it to understand what kind of changes in standard will render your prototype useless, and what kind of changes are only cosmetic. However, sometimes it is just better to throw the prototypes away. Unfortunately you are not typically the person to make that decision.

You should also carefully study motives of your company to participate in standardization effort. Four typical cases are mentioned at the end of pattern *Know People*.

When you know yourself it is obviously easier to set goals, and give to each of them preferences. However, you should not reveal any information about your list of preference before you really have to. Otherwise you will loose amount of important currency you can use in bargains, as written by Sun Tzu:

“By discovering the enemy's dispositions and remaining invisible ourselves, we can keep our forces concentrated, while the enemy's must be divided.”

Sun Tzu

Now it is time to continue TDMA vs. CDMA example. You have studied in previous pattern both of these technologies, and you have studied in this pattern the strategy of your company. Now it is time to choose your side based on this information.

Back-up Team Ready

No matter how well you prepare, there could always be a case for which you haven't prepared.

Therefore:

You should have a possibility to alert your back-up team during the meetings (at least virtually) anytime. Organize a team, which is available all the time during meeting days (and evenings, remember time-zones) for a quick teleconference, and who are ready to find and to send you those parts of reference material you have forgotten to take with you.

During Meeting

First Version is Important

Several fields of the modern technology have one common force: backward compatibility. Sometimes there are technical reasons for that, sometimes reasons are more political. In standardization work people tend to play an important role related to this force. Any kind of significant change is not possible in subsequent versions of standard because people who did the original work will not be pleased and will be strongly against any changes.

Therefore:

Work hard to get the first version as close to the right one for you as possible, later on it is difficult to make any significant change to it because it have to be “backward compatible”. This means that you should do your homework properly and when an issue is taken into discussion you should have your opinion ready. If you have accepted something, it will be very difficult to remove it later.

You should understand that anything added to standard will not just go away. The size of the standard always increases, never decreases. The only possibility to decrease the size of a standard is when people who put things in (inventors) eventually go away.

The well-known KISS (Keep It Simple and Straightforward) principle works also in standardization work.

Concepts over Names

You are working in the group whose aim is to create a standard. During the process new concepts are found and possibly added. However, participants of the work group are from different kind of interest groups and some of them have already those new things included to their goals, some may try to avoid to have them included. Typically this will wind up in a long and sometimes tense discussion.

You are in the situation that something you like to be added to standard is at stake. In the ideal situation new concepts added to standard are named as you like.

Therefore:

If there is strong resistance against something you want to be added to the standard, you can try to ease your demands a little: keep the concept as it is, but accept a new name for it. In some cases people are not against the thing itself, they are against how it is named in proposal. If the name of concept in concern is changed they could accept it without thinking twice. The reason behind this phenomena is that for many people names are most important things, and the first name given during standardization is the most important one. They potentially guide thoughts to particular direction or area, and some of them they want to avoid. For this reason they cannot even accept any names from that area. Of course this don't work if you are the one who want a particular name to the standard.

Power of Examples

You are working in the group whose aim is to create a standard. There are things included in the standard that are complicated to understand if there is only textual description of it.

Therefore:

Use examples to make things easier to understand. Do not forget that carefully chosen examples guide people understanding, and usually people stick to their first impression.

Use Straw Poll

Participants of the work group are from different interest groups which usually have conflicting goals. Often discussion about different issues will be long and sometimes tense. If people have taken a strong opinion they have difficulties to change it afterward. Especially an individual vote (even your own) is very difficult to change afterward.

Therefore:

During meetings use straw poll to see how things are going – try to reach “strong consensus”. People have easier to change their opinion if they can do it without losing their face. For this reason do not put anything to the vote if not absolutely necessary.

Acknowledgments

During the years many people have supported me and facilitated my work with patterns in protocol engineering with comments, ideas, encouragements, and resources. Especially I would like to thank my shepherds who helped me in this quest: Neil Harrison (VikingPLoP2003), Rick Dewar (EuroPLoP2002), Norm Kerth (EuroPLoP2001), and Michael Stall (PLoP2000). I would also like to thank my long time co-author, Markku Turunen, participating this (seems to be) never ending story.

This paper has been workshopped at VikingPLoP2003 in Bergen, Norway. I would like to thank especially Richard Gabriel for his excellent work as a moderator and to Linda Rising for her numerous comments on my paper.

For this pattern language I have interviewed experts who have experience from different standardization organizations: Morgan Björkander, Antti Huima, and Steve Randall.

References

- [1] Sun Tzu, *The Art of War*, Project Gutenberg Etext #132, 1994.
- [2] Niccolo Machiavelli, *The Prince*, Project Gutenberg Etext #1232, 1998.
- [3] J. Pärssinen, M. Turunen, *Pattern Language for Architecture of Protocol Systems*, a pattern workshop paper presented at EuroPLoP2001, 4 - 8 July 2001, Irsee, Germany.

Factory and Disposal Methods

A Complementary and Symmetric Pair of Patterns

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

May 2004

Abstract

complementary (of two or more different things) combining in such a way as to form a complete whole or to enhance or emphasize each other's qualities.

symmetry the quality of being made up of exactly similar parts facing each other or around an axis.

- correct or pleasing proportion of the parts of a thing.
- similarity of exact correspondence between different things.

The New Oxford Dictionary of English

Manual object creation may be in conflict with information hiding or instance-controlling requirements. The consequences of such separation and encapsulation can be addressed by the FACTORY METHOD pattern. Further control, economy, and symmetry may be found in the DISPOSAL METHOD pattern, in effect a mirror of FACTORY METHOD.

This paper revisits the classic FACTORY METHOD pattern, broadening the scope of this general pattern in line with the common usage of its name. Four specific variants are examined: PLAIN FACTORY METHOD, CLASS FACTORY METHOD, POLYMORPHIC FACTORY METHOD, and CLONING METHOD. FACTORY METHOD is accompanied by DISPOSAL METHOD, making the consideration of object lifecycle more clearly balanced. Two specific variants are examined: FACTORY DISPOSAL METHOD and SELF-DISPOSAL METHOD.

FACTORY METHOD and DISPOSAL METHOD are, in essence, quite high level whereas each of their variants is a more specific pattern. In the context of a specific pattern language or sequence it often makes more sense to zoom in on the specific variants rather than refer abstractly to the zoomed-out generalizations. This paper does not present a specific pattern language or a complete pattern sequence, more of a generative phrase or expression that can be incorporated and reified in a language or sequence.

Introduction

There is an inherent tension between data hiding and object creation. For example, if you hide object use behind an interface, how do you know which concrete class to use for creation? With any luck, if you are an experienced OO developer, you will now be sitting back in your seat, confident in the knowledge of at least one good answer. There is a good chance that this answer is FACTORY METHOD [Gamma+1995]:

Define an interface for creating an object, but let subclasses decide which class to instantiate. FACTORY METHOD lets a class defer instantiation to subclasses.

Well, you can lean forward now: this pattern deserves a revisit and revision to free it from a purely inheritance-centric view; it also warrants a counterpart to make it part of a greater design whole.

Both before and since the Gang of Four published FACTORY METHOD, the term *factory* has been used by programmers in a slightly broader sense, one not necessarily restricted to class hierarchies. Programmers will happily name a non-polymorphic method a *factory method*, so long as the obvious creational role indicated by a literal reading of the pattern name is followed. A factory is therefore generally a defined location with responsibility for encapsulating object creation.

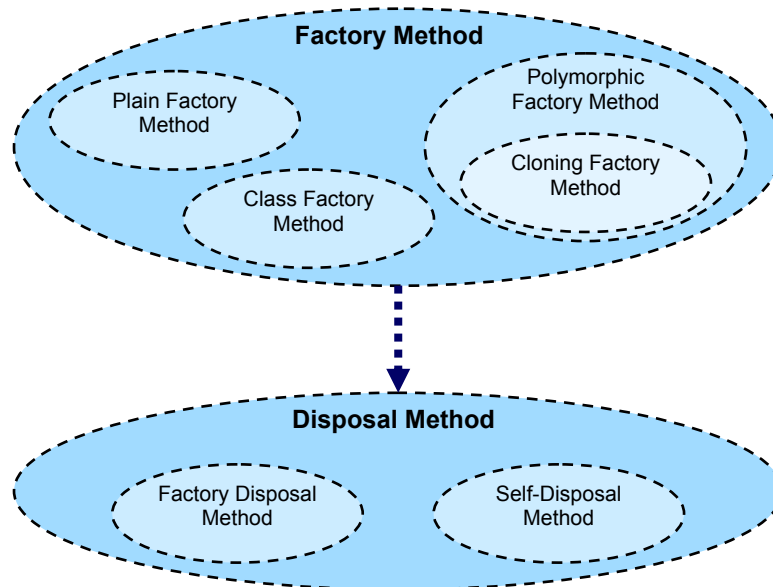
There is also something missing from the common discussion of object creation through factories: object disposal. Contemplating the sound of one hand clapping is a spiritual question not always well suited to the classically utilitarian materialism of objects. The absence of symmetry in the discussion of FACTORY METHOD suggests DISPOSAL METHOD. This relationship is not so much a tiny pattern language or short pattern sequence as a simple generative pattern phrase or subsequence, something that might be uttered in conversation in a language or included in a longer, domain-specific sequence. The symmetric pairing marries and mirrors FACTORY METHOD with DISPOSAL METHOD: one seeks closure in the other. As with any real mirror or marriage, the reflection is not perfect: in the detail of these patterns there is a great deal of independent variation that contrasts with the sketch-level symmetry.

Symmetry is a fundamental consideration [Alexander2002, Henney2003, Zhao+2003] that typically has the effect of simplifying a design, making it easier to comprehend and work with, not to mention more elegant and more whole. The simplification comes from the resulting regularity: something that is regular is easier to recall or second-guess than something that is not. Symmetry encourages consistency, becoming its own design map. This does not mean that designs should be globally and thoroughly symmetric but that, where transparency is not possible, a design should be predominantly and locally symmetric. A purely symmetric design is typically quite a dull one; a purely asymmetric one is unmemorable for different reasons.

A common question confronting both pattern authors and readers is that of specificity. Each occurrence of a pattern in a system is clearly highly specific, but at what level should the pattern itself be described? How specific should the problem be? What differentiates a variant of a pattern from the core pattern? Both the level and context of interest often dictate whether a pattern should be expressed at the most general level, e.g. a FACTORY METHOD is a method responsible for the creation of objects, or whether different flavors should be singled out and named, e.g. a POLYMORPHIC FACTORY METHOD defers the knowledge of exact creation type to be pushed down a class hierarchy. In the context of a specific pattern language it tends to make sense to focus on specific variants because the problems they address in the context of the language are similarly specific, e.g. while FACTORY METHOD offers a general heading for describing an approach to creational

encapsulation, a PLAIN FACTORY METHOD does not solve quite the same problem as a CLASS FACTORY METHOD, nor does it have quite the same consequences.

In this paper the focus is not a pattern language but on two patterns that, at a general level, form part of a vocabulary for object lifecycle design. PLAIN FACTORY METHOD, CLASS FACTORY METHOD, and POLYMORPHIC FACTORY METHOD are presented in the context of the more general FACTORY METHOD, with CLONING METHOD as a further flavor of POLYMORPHIC FACTORY METHOD. FACTORY DISPOSAL METHOD and SELF-DISPOSAL METHOD are presented in the context of DISPOSAL METHOD. The following diagram illustrates the relationships:



A low-ceremony pattern form is used. In general, the focus of the patterns is on the statically typed OO model of Java, C++, and C#. Specifically, code fragments are in Java.

FACTORY METHOD

Encapsulate the concrete details of object creation by providing a method for object creation instead of letting object users instantiate the concrete class themselves.

Problem: Code that depends on instances of a class or from a class hierarchy may need to create the objects itself. This may not be as easy as simply using a new expression:

- What if the creational logic cannot be contained easily inside a constructor? What if external validation is needed or object relationships must be established that might be considered beyond the scope of the object's immediate responsibility? For example, the constructor of a bank account object should not be responsible for allocating its account number, running a credit check, or ensuring that the instance is persisted by its associated bank.
- What if the class must be instance controlled, so that unconstrained use of new would be inappropriate? For example, neither ENUMERATION VALUES [Henney2000a, Henney2000b] nor SINGLETON objects [Gamma+1995] should be created manually or directly by their users.
- What if the appropriate concrete class is unknown to the user because the user manipulates an object only via a declared interface and not via its concrete class? For example, an object whose actual type depends on the actual type of another object should not cause the user to copy and paste repetitious type-dependent code. Cascaded if else if statements that hardwire instanceof, dynamic_cast, or is runtime type checks are a good way of obscuring a method's intent and reducing a class's openness and extensibility.
- A more specific example of needing object creation in the presence of hierarchical abstraction is the wish to take a proper copy of an object without knowing its concrete type.

These different scenarios are unified under a common pair of opposing forces:

- Objects are most simply and intuitively created using a new expression, specifying a concrete class and constructor arguments. This provides the user of a class with full control over instantiation.
- Direct object creation may inadvertently obfuscate and reduce the independence of the calling code if any of the necessary ingredients for correct object creation are not readily available. The concrete class, the full set of constructor arguments or the enforcement of other constraints may not be known at the point of call; to require them would increase the complexity of the calling code.

Solution: Provide a method for fully and correctly creating the appropriate object instead of relying on a new expression. The knowledge of creation is encapsulated within this *factory method*. The ability to create instances directly is hidden from the caller either by making constructors non-public or by pushing it down a class hierarchy.

However, unless created specifically for the purpose, including the role of *creator* in a class's repertoire can sometimes be considered an addition that dilutes its cohesiveness. The solution is certainly more encapsulated than the alternatives, but the cohesion can be considered a little lower than in a design where such creation was never needed.

There are three basic and one extended variant of FACTORY METHOD that determine how the different roles of *product* and *creator* (also known as the *factory*) are realized:

- PLAIN FACTORY METHOD: The *creator* is an object — not necessarily in a class hierarchy — and the type of the *product* either is fixed or varies only with environmental settings

or the arguments to the *factory method*. A PLAIN FACTORY METHOD implementation is normally just a case of providing an ordinary, possibly `final` or `sealed`, method that creates instances of another class, with no specific intent to be inherited or overridden.

- CLASS FACTORY METHOD: The *creator* is a class rather than an object, and so the *factory method* is static. The *creator* is often the same class as the *product* object type, which is not normally defined in a class hierarchy. Direct creation of *product* objects is often prevented by ensuring that instance constructors are non-public. CLASS FACTORY METHOD pattern is also known as STATIC FACTORY METHOD [Bloch2001, Haase2002].
- POLYMORPHIC FACTORY METHOD: The possible types of the *product* object are defined in a class hierarchy. Mirroring the hierarchy of what is created, an interface for *creator* objects is provided, offering the *factory method* abstractly, and the responsibility for creation is deferred to an implementing subclass. The knowledge of which type of *product* is required is contracted out to the *creator* hierarchy, removing the need for a closed and clumsy instanceof solution. This FACTORY METHOD variant is the classic Gang of Four version.
- CLONING METHOD: The *product* class is the same as the *creator* class. However, unlike a CLASS FACTORY METHOD the relationship is properly reflexive: the *creator* is an instance of the class, rather than the class, so that its result is another object of its own type. To be precise, the *product* is a proper copy of its *creator*. A CLONING METHOD is a specific kind of POLYMORPHIC FACTORY METHOD.

A PLAIN FACTORY METHOD is fairly straightforward in its common form. The *product* is normally concrete, and may have only non-public constructors:

```
public class ConcreteProduct
{
    ...
    private ConcreteProduct(...) ...
}
```

The *creator* is also normally concrete, and has sufficient access to the *product* type to create instances:

```
public class ConcreteCreator
{
    public ConcreteProduct create()
    {
        return new ConcreteProduct(...);
    }
    ...
}
```

For the bank account example, a bank object would adopt the role of *creator* and an account object would be a *product*. The bank would hide the details of creation and the account class would prevent general creation by users. The design is encapsulated between the two classes and need not involve any inheritance.

The form of a CLASS FACTORY METHOD is simple, with the class as *creator* and its instances as product:

```
public class ConcreteProduct
{
```

```
public static ConcreteProduct create()
{
    return new ConcreteProduct(...);
}
...
private ConcreteProduct(...) ...
}
```

For example, as an example of symmetry, a Java class that supports a meaningful `toString` override could consider providing a `fromString` or `valueOf` CLASS FACTORY METHOD in preference to a public `String` constructor. Using a CLASS FACTORY METHOD names the conversion concept explicitly. It sets string-based creation apart from other constructors to emphasize the inverse relationship with the common `toString` method.

The general POLYMORPHIC FACTORY METHOD has the most intricate detail, spanning two class hierarchies where the previous two variants typically address one or two classes on their own. There is the *product* hierarchy:

```
public interface Product
{
    ...
}
...
public class ConcreteProduct implements Product
{
    ...
}
```

And there is the *creator* hierarchy:

```
public interface Creator
{
    Product create();
    ...
}
...
public class ConcreteCreator implements Creator
{
    public Product create()
    {
        return new ConcreteProduct(...);
    }
    ...
}
```

Where the caller and the used class hierarchy are one and the same, TEMPLATE METHOD [Gamma+1995] is often used:

```
public abstract class ProductUser
{
    public void useNewProduct()
    {
        Product produce = create();
        ...
    }
}
```

```
    protected abstract Product create();
}
...
public class ConcreteProductUser implements ProductUser
{
    protected Product create()
    {
        return new ConcreteProduct(...);
    }
}
```

A degenerate arrangement of POLYMORPHIC FACTORY METHOD is CLONING METHOD (or VIRTUAL COPY CONSTRUCTOR or SELF-FACTORY METHOD), which is normally used in its own right to support polymorphic copying but can also be found in support of a PROTOTYPE approach to object creation [Gamma+1995, Coplien1992], with which it is often confused. In CLONING METHOD the types of the *product* and the *creator* are the same, and the *creator* instance provides itself as the model from which a new instance is built:

```
public class Product implements Cloneable
{
    public Object clone()
    {
        ... // cloning carried out and resulting object returned
    }
    ...
}
```

The cloning is instigated directly by the object user:

```
...
public void takeSnapshot(Product other)
{
    snapshot = (Product) other.clone();
}
...
```

In PROTOTYPE an object is held by a factory to be used as the prototypical instance from which new factory products are built. This may involve a CLONING METHOD:

```
public class ConcreteCreator implements Creator
{
    public Product create()
    {
        return (Product) prototype.clone();
    }
    ...
    private Product prototype;
}
```

Or not:

```
public class ConcreteCreator implements Creator
{
```

```
public Product create()
{
    return new Product(prototype.attributes());
}
...
private Product prototype;
}
```

In the second fragment the factory product is created using the attributes of the prototype object and explicitly constructing the object. In both cases the prototype is used as the instance on which factory products are based, but only in the first does the implementation mechanism qualify as a **FACTORY METHOD**.

DISPOSAL METHOD

Encapsulate the concrete details of object disposal by providing an explicit method for clean up instead of letting object users either abandon objects to the tender mercies of the garbage collector or terminate them with extreme prejudice and delete.

Problem: How should objects with significant clean-up behavior be disposed of after use? For garden-variety objects, the usual mechanism of the language for disposing of objects is normally sufficient. However, for some kinds of objects, such as resources, this may not be enough. Just as a **FACTORY METHOD** may hide details of an object's creation that cannot be handled fully by a constructor, details of an object's destruction may go further than can be adequately expressed by conventional finalization, whether a `finalize` method or destructor.

A resource can be defined by its use and context rather than in terms of its abstraction. A resource is any object that could easily become scarce in a system and whose scarcity would cause problems. Therefore, a resource can be defined liberally as anything that should be returned after acquiring and using it. For example, memory in C and C++ is a resource, but in a well-endowed Java or C# program it is typically not. However, in a smaller environment memory again becomes a resource. In the common application of a **FACTORY METHOD**, instance creation is controlled but object disposal is not. Because resource usage may need to be conserved and resources recycled, the user of a resource should have a clear contract for how a resource's usage lifetime is bounded.

In C++ an explicit `delete` by a factory-product user is asymmetric with the hidden `new` in the factory. A `delete` expression deterministically triggers the end of an object's life, which may be a somewhat more severe disposal than is wanted: it is difficult to recycle an object if it no longer exists. There is also no guarantee that an object was created using a plain `new`, hence a `delete` may be precisely the wrong action even to end an object's life. Memory that is acquired independently of construction would rely on a placement `new` expression for construction and an explicit destructor call for destruction; there is no corresponding `delete` expression.

Java and C# programmers can discard objects for later automatic collection by the garbage collector. It is, however, naïve to assume that a GC system solves all memory and resource management issues out of the box [Bloch2001]:

When you switch from a language with manual memory management, such as C or C++, to a garbage-collected language, your job as a programmer is made much easier by the fact that your objects are automatically reclaimed when you're through with them. It seems almost like magic when you first experience it. It can easily lead to the impression that you don't have to think about memory management, but this isn't quite true.

It is possible to further dilute confidence in totally transparent GC: your objects *may* be reclaimed automatically. There is little guarantee that they will be claimed in a timely manner, or even at all — although such low (or non-existent) quality-of-service would find favor with few developers. Frequent creation of fine-grained objects, such as iterators or value objects, can potentially lead to inefficient use of resources or even resource exhaustion.

With respect to resources, a specific and timely clean-up action may be required, but in the absence of explicit control over the tail end of an object's life this cannot be made implicit — and whatever the problem, Java's `finalize` is rarely the answer. GC addresses the issue of object collection to make memory resourcing transparent, but this does not apply to other resources.

Solution: Provide a method for explicit clean up and disposal of an object. Mirroring FACTORY METHOD, DISPOSAL METHOD answers the question of who is responsible for the clean up and disposal of an object by making the clean up an explicit operation for the user. Just as the user requested an object for use, they must also mark the end of its use.

DISPOSAL METHOD may be expressed as one of two basic variants:

- **FACTORY DISPOSAL METHOD:** Provide a method on the factory that originally created the object. The knowledge of an object's lifecycle is isolated in a single place, which allows transparent instance control, such as an object pool that caches and recycles objects.
- **SELF-DISPOSAL METHOD:** Provide a method on the object to be disposed of. This method either performs the clean up itself or, if a factory was involved in the object's creation, it returns of the object to its maker.

An obvious and negative consequence of this pattern is that the user must remember to both make the call and make the call exception safe. This situation is tedious and error prone, and can be ameliorated through additional encapsulation, such as a COMBINED METHOD [Henney2000c], EXECUTE-AROUND METHOD [Henney2001a], or a COUNTING HANDLE [Henney2001b]. Where instance control is neither about resource management nor in the hands of an object user, no disposal is required, so DISPOSAL METHOD is not necessarily appropriate.

FACTORY DISPOSAL METHOD is the natural complement of FACTORY METHOD, and its truest reflection:

```
public interface Creator
{
    Product create();
    void dispose(Product toDisposeOf);
    ...
}
```

In the bank account example closing an account would be a good example of a FACTORY DISPOSAL METHOD. The code that decides to dispose of a factory-created object must have access to both the *creator* and the *product*. This not only means that the lifetime of the product must be contained within that of its creator, but that the caller of the DISPOSAL METHOD is expected to co-ordinate the disposal correctly, i.e. it should ensure that it matches the right product with the right factory. This is often not a significant issue because factories and products are normally well matched in terms of types and scope usage. However, this slight increase in coupling can present a potential liability for some programs.

SELF-DISPOSAL METHOD is sometimes known as an EXPLICIT TERMINATION METHOD [Bloch2001]:

```
public interface Product
{
    void dispose();
    ...
}
```

Where a SELF-DISPOSAL METHOD is simply a forwarder to a FACTORY DISPOSAL METHOD, it clearly has to retain some kind of reference to the factory of origin. In such a case it

successfully encapsulates the knowledge of its origin and therefore the correct co-ordination of *product* to *creator*. The product user — or, to be precise, disposer — is freed from maintaining and using this extra reference. Although this offers a better encapsulation of the constraints governing the factory-product pairing, it can be seen as slightly less cohesive because responsibility for disposal is represented in the product interface, which would otherwise be focused purely on matters of product usage.

In C++ a DISPOSAL METHOD displaces the use of a public `delete` for the product type in question. The lifetime of an object is no longer subject to the operators in the language but to the higher-level interfaces and object lifecycle choices of a specific application. To ensure no clash between the use of a DISPOSAL METHOD and the common use of a `delete`, the destructor of the target object should not be public at the level of the interface. This restriction prevents any attempt to mix the `delete` and DISPOSAL METHOD models at compile time.

Acknowledgments

This paper is derived from a previous article [Henney2002].

I would like to thank Klaus Marquardt for his thorough and insightful shepherding of this paper for VikingPLoP 2003, Mark Radford for his additional comments both before and after the conference, and Neil Harrison for his comments following the conference. From the workshop at the conference I would like to thank Jacob Borella, Franco Guidi-Polanco, Alan O'Callaghan, and Titos Saridakis.

References

- [Alexander2002] Christopher Alexander, *The Nature of Order, Book 1: The Phenomenon of Life*, Center for Environmental Structure, 2002.
- [Bloch2001] Joshua Bloch, *Effective Java*, Addison-Wesley, 2001.
- [Coplien1992] James O Coplien, *Advanced C++: Programming Styles and Idioms*, Addison-Wesley, 1992.
- [Gamma+1995] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1995.
- [Haase2002] Arno Haase, "Idiom in Java", *JavaSPEKTRUM* 36, February 2002.
- [Henney2000a] Kevlin Henney, "Patterns of Value", *Java Report* 5(2), February 2000, available from <http://www.curbralan.com>.
- [Henney2000b] Kevlin Henney, "Value Added", *Java Report* 5(4), April 2000, available from <http://www.curbralan.com>.
- [Henney2000c] Kevlin Henney, "A Tale of Two Patterns", *Java Report* 5(12), December 2000, available from <http://www.curbralan.com>.
- [Henney2001a] Kevlin Henney, "Another Tale of Two Patterns", *Java Report* 6(3), March 2001, available from <http://www.curbralan.com>.
- [Henney2001b] Kevlin Henney, "C++ Patterns: Reference Accounting", *EuroPLoP 2001*, July 2001, available from <http://www.curbralan.com>.
- [Henney2002] Kevlin Henney, "Symmetrie in Java", *JavaSPEKTRUM*, July 2002, available in English as "The Importance of Symmetry" from <http://www.curbralan.com>.
- [Henney2003] Kevlin Henney, "Five Possible Things after Breakfast", *The Road to Code* blog at *Artima*, 23rd June 2003, <http://www.artima.com>.
- [Zhao+2003] Liping Zhao and James Coplien, "Understanding Symmetry in Object-Oriented Languages", *Journal of Object Technology* 2(5), September–October 2003, http://www.jot.fm/issues/issue_2003_09/article3.

The Good, the Bad, and the Koyaanisqatsi

Consideration of Some Patterns for Value Objects

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

May 2004

Abstract

koyaanisqatsi /ko.yaa.nis.katsi/ (from the Hopi language), noun:

1. crazy life.
2. life in turmoil.
3. life disintegrating.
4. life out of balance.
5. a state of life that calls for another way of living.

From the 1983 film of the same name, directed by Godfrey Reggio,
soundtrack composed by Philip Glass

This paper considers and combines some seemingly disparate ideas: symmetry within and across class interfaces; the notion of ineffective but recurring design styles as patterns, albeit dysfunctional ones; the role of balance within good patterns; value-based programming idioms in Java. Inappropriate symmetry in a faulty but common interface style is presented as a plausible pattern, which is then counterbalanced by a pair of patterns that break and recast the symmetry at a different level, resolving the design tension.

Pairing every get method with a set method offers an apparently simple and symmetric design style, but it is one fraught with problems. In the context of value-based programming in Java the balance may be better expressed across two classes: an IMMUTABLE VALUE and a MUTABLE COMPANION. These complementary patterns represent two sides of the same design coin, a symmetry and contrast at a different level to the per-method pairing of get with set. In contexts other than value-based programming in Java the design tension will be resolved in different ways. This is not to say that one should never write get and set methods, just that to characterize the practice as the GETTERS AND SETTERS pattern suggests that the use of get and set methods is a good solution to a specific problem rather than a cumulative consequence of other design decisions. Reasoning more carefully about GETTERS AND SETTERS as a pattern, with documented forces and consequences, exposes its unbalanced nature and its weakness as a design guideline.

Introduction

Symmetry is a notion grounded in our experience of the real world. We apply it, by metric and by metaphor, to other domains, both physical and abstract. It can be considered both formally and informally, with respect to a rigorously mathematical, mirror-perfect balance or as a looser, more tacit equilibrium of opposites. Symmetry is a form of consistency that is about balance and opposites.

Symmetry can be a useful, memorable, and harmonic local property in designs [Alexander2002, Coplien+2001, Zhao+2003]. It requires less effort to explain and recall, guiding expectation that when a particular feature is present, its logical counterpart will also be there. That where there is the capability for output there is also the capability for input. That where a resource can be acquired it can also be released. That where there is a commit there is also a rollback.

It is tempting to extrapolate this series of balanced pairings to get and set methods, i.e. where there is a get method there should also be a set method. However, this would be a step too far and a temptation that should be resisted. Such a guideline is simplistic rather than simple. It leads to code that is difficult to use correctly and, except in trivial cases, difficult to implement correctly. In spite of this, some company coding guidelines mandate a pairing of a set with every get, and even a get for every private field. Others have gone even further by arranging for a get and a set to be generated automatically for every field. A simple vowel shift might better describe such code: *uncapsulated*.

This style is sometimes favorably referred to as a pattern. Patterns can be considered either "good" or "bad". When programmers normally talk about patterns there is an implicit assumption that they are talking about patterns that improve the quality of their systems, an implicit assumption that documented patterns are, almost by definition, of the "good" variety, so that anything that is a "bad" pattern is not a pattern. From this perspective, automatic pairing of set with get methods cannot be considered a pattern, only a questionable and smelly [Fowler1999] coding guideline with a long footnote of problems. *Caveat setter*.

Any pattern describes a dialog with a design situation. It explores the context of a design problem, whether large scale or fine grained, enumerating the forces that drive and buffet the design. A pattern moves on to describe a solution. But, importantly, a pattern does not end there. The dialog continues by describing the consequences of applying the proposed solution, detailing the resulting context. What forces were balanced? What was left unresolved? What benefits have arisen? What liabilities have been incurred?

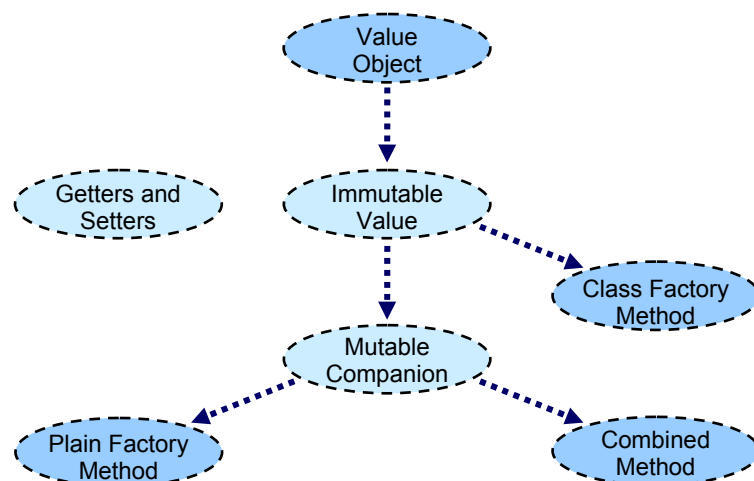
If "bad" patterns are also considered to be patterns — the case originally [Alexander1979] — we can understand their dysfunctionality by trying to consider each as an unfulfilled whole — a whole with holes. A "good" pattern is one whose forces are both genuine and well matched by its consequences. A "bad" pattern, on the other hand, is one whose forces are incomplete and whose consequences are out of balance.

Patterns, whether good or bad, have a recurring nature and recognisable form. By trying to understand GETTERS AND SETTERS as some kind of pattern we can see it more easily in code; we can better understand why it is not a good pattern; most importantly, we can see how we might better address the design problem — preferably with a solution rather than another problem. This is not to say that objects should never have paired get and set methods — if there is any guideline here, it is probably that any set methods should be accompanied by get methods, rather than the other way around [Henney2004] — just to point out that when useful, such a pairing is a consequence of other design decisions and less a decision in itself.

This paper takes the implementation of VALUE OBJECT types in Java [Fowler2003, Henney2000a, Henney2000b, Wiki] as its example for comparing "good" and "bad" patterns. GETTERS AND SETTERS can in principle apply to any kind of object in any language supporting objects, but the context of VALUE OBJECTS allows discussion of an alternative approach to be specific and concrete — in this case, the IMMUTABLE VALUE and MUTABLE COMPANION pairing. In another context, such as expressing the interface of an entity type or refactoring procedural code into object-oriented code, GETTERS AND SETTERS would be presented a little differently, taking into account differences in the context, forces, and consequences while retaining the similar recurring intention and construction that makes it some kind of pattern.

The complementary IMMUTABLE VALUE and MUTABLE COMPANION patterns are part of a larger pattern language, but are presented in this paper connected only to VALUE OBJECT, one another, and variants of FACTORY METHOD [Henney2003c]. GETTERS AND SETTERS is persuasive but deceptive: it is based on a slightly false identification of forces and a selective consideration of consequences. Closer inspection and reflection reveals that its liabilities outweigh its benefits. Although the pattern has an air of plausibility in its presentation, it is out of balance: any application of it can throw a design off centre. The complementary pattern pairing considers more authentic design criteria with forces that are relevant and an outcome that is more clearly balanced.

The following diagram shows the generative relationships (or lack of) between the patterns under discussion, highlighting the patterns described in this paper:



It is sometimes said that no pattern is an island, but, as the diagram shows, this is not always true. Although, in the documented form that follows, GETTERS AND SETTERS believes that it has VALUE OBJECT as its context, the relationship, from VALUE OBJECT's perspective, is an unconsummated one. A careful examination of GETTERS AND SETTERS' consequences indicates that what at first appears to be a beneficial pattern, is little more than a chimera. It has a false symmetry encouraged by assonance in its English form — a phony euphony. A *setter* does not in truth have the opposite effect to a *getter*. It is perhaps more informative and constructive to think in terms of *putter* or *modifier* and *enquirer* or *query*, evaluating the need for these roles specifically for each design. Such design symmetry as exists is local but not necessarily at the level of a single class interface: it can be found more convincingly broken and recast across two class roles, IMMUTABLE VALUE and MUTABLE COMPANION.

GETTERS AND SETTERS — A Dysfunctional Pattern

A VALUE OBJECT can be described in terms of one or more primitive values. A VALUE OBJECT provides a higher level means of describing information in the system than simply holding or passing around one or more primitive values. A VALUE OBJECT type describes a concept more precisely, including encapsulated operations that refer to the VALUE OBJECT as a whole. Each primitive value associated with a VALUE OBJECT is an attribute that has a specific role. However, exposing such attributes as public data is considered to be a poor practice.

Rather than communicate a concept, such as a date, in terms of primitive values, e.g. three int arguments, a VALUE OBJECT expresses the concept as a single object. This packaging is both easier to comprehend and simpler to manipulate. An object branded as a Date is clearly more meaningful than an ad hoc grouping of three otherwise unconstrained numbers.

However, if attributes, such as year, month, and day, are exposed as public fields, the strength of encapsulation is weakened:

```
public class Date
{
    public int year, month, day;
    ...
}
```

Constraint enforcement is lost: there is no 29th February 2003 and there is never a 32nd January. Public data advertises that the exposed fields have no relationships or rules governing them that the class author wishes to enforce.

Public data also commits the class to a single representation. Although it conveniently matches our intuition, representing a date as three integer values is not the most effective or efficient representation for most programs [Henney2003b]. A scalar epoch-based date, where the date is counted in days from a fixed date, is often simpler to work with and more compact in representation, e.g. a single integer counting the days since 1st January 1900. The year, month, and day attributes are still conceptually valid, but they would have to be calculated from the representation instead of actually being the representation. A similar case can be made for the day of the week and the week in the year: conceptually valid attributes that can be calculated, but not ones that should be stored and exposed publicly as fields along with other attributes.

Therefore, for each primitive value that can be considered an attribute of a VALUE OBJECT, provide a pair of methods that allow the attribute to be queried and set. An attribute will often correspond to a private field, but this need not be the case: an attribute may be a calculated value, derived from other fields.

The most common naming convention is to prefix the name of the conceptual attribute with a get and a set for each pair of methods:

```
public class Date
{
    public Date(int year, int month, int dayInMonth) ...
    public int getYear() ...
    public void setYear(int newYear) ...
    public int getMonth() ...
    public void setMonth(int newMonth) ...
    public int getDayInMonth() ...
}
```

```
public void setDayInMonth(int newDayInMonth) ...  
    ...  
}
```

However, this is not necessarily the clearest or cleanest option. GETTERS AND SETTERS can be implemented using slightly less pedestrian names, e.g. `year` instead of `getYear`.

Different implementations with different trade-offs can be expressed using a common interface:

```
public class Date  
{  
    ...  
    private int year, month, day;  
}
```

Or:

```
public class Date  
{  
    ...  
    private int daysSince1900;  
}
```

A problem with the fine granularity of the operations is that of invalid, intermediate states. Consider the following:

```
Date date = new Date(2003, 2, 28);  
date.setDayInMonth(30);  
date.setMonth(1);
```

The intent is that the object referred to by `date` is initialized to 28th February 2003 and then modified to 30th January 2003. The problem is that the ordering of the operations shown means that at one point the object would conceptually have to hold the date 30th February 2003, an invalid date. Either the operation must fail at this point or the validity enforcement of the class must be disabled, weakening its encapsulation. For attributes that are derived rather than stored directly, this can become even more of a challenge: 30th February 2003 could be interpreted as 2nd March 2003, which would lead to the final date being calculated as 2nd January 2003.

Another liability arises from sharing. A `Date` object that is shared by being passed as an argument or returned as a method result can be modified by one party in ways unexpected and unwanted by the other party. The only solution in this context is careful and defensive copying of arguments and results, so that each party holds a unique instance.

IMMUTABLE VALUE — A Better Pattern

VALUE OBJECTS are fine-grained, stateful objects used to express quantities and other simple information in a system. Object identity is not significant for a value, but its state is. VALUE OBJECTS form the principal currency of representation and communication between many kinds of objects, such as entities and services. As such, references to VALUE OBJECTS are commonly passed around and stored in fields. However, state changes caused by one object to a value can have unexpected and unwanted side effects for any other object sharing the same value instance.

How can you share VALUE OBJECTS and guarantee no side-effect problems? Defensive copying is a convention for using a modifiable value object to minimize aliasing issues. However, this practice is tedious and error prone, and may lead to excessive creation of small objects, especially where values are frequently queried or passed around.

With multi-threading the troublesome consequences of aliasing are multiplied. Synchronizing methods addresses the question of valid individual modifications, but does nothing for the general problem of sharing. Synchronization also incurs a performance cost.

Therefore, define a VALUE OBJECT type whose instances are immutable. The internal state of a VALUE OBJECT is set at construction and no subsequent modifications are allowed: only query methods and constructors are provided; no modifier methods are defined. A change of value becomes a change of VALUE OBJECT referenced.

The absence of any possible state changes means that there is no reason to synchronize. Not only does this make IMMUTABLE VALUES implicitly thread safe, but the absence of locking means that their use in threaded environments is also efficient. Sharing of IMMUTABLE VALUES is also safe and transparent in other circumstances, so there is no need to copy an IMMUTABLE VALUE, and thus no need to support cloning or copy construction.

Declaring the fields `final` ensures that the *no change* promise is honoured. This guarantee implies also that either the class itself must be `final` or its subclasses must also be IMMUTABLE VALUES:

```
public final class Date
{
    public Date(int year, int month, int dayInMonth) ...
    ... // other constructors
    public int year() ...
    public int month() ...
    public int dayInWeek() ...
    public int dayInMonth() ...
    public int dayInYear() ...
    ... // other query methods
    private final int daysSince1900;
}
```

References to — rather than the attributes of — an IMMUTABLE VALUE are changed to effect value change. The reference may be to an existing object or a new one may need to be created. There are complementary techniques for creating IMMUTABLE VALUES: provide a complete and intuitive set of constructors; provide a number of CLASS FACTORY METHODS [Henney2003c] if some aspect of the creation needs to be controlled or named, e.g. `Date.today()`; provide a MUTABLE COMPANION if values are used in complex expressions that could generate many IMMUTABLE VALUE instances. Values are not resources, so their FACTORY METHODS do not need to be mirrored by DISPOSAL METHODS [Henney2003c].

MUTABLE COMPANION — A Complementary Pattern

IMMUTABLE VALUE objects provide a simple and safe means of expressing and sharing values in a system. However, the construction of an **IMMUTABLE VALUE** is not always a simple matter of using a new expression or calling a **CLASS FACTORY METHOD**. Some values may be the outcome of complex or ongoing calculations.

Constructors for an **IMMUTABLE VALUE** type offer a way of creating instances from a fixed set of arguments, but they cannot accumulate changes or handle complex expressions without themselves becoming too complex or overly general. The need for frequent or complex change typically leads to expressions that create many temporary objects.

For example, concatenating multiple strings, stripping unwanted characters and tokens out of an input line before further processing, accumulating an invoice total from many items, and generating a sequence of dates approximately two weeks apart but not falling on public holidays or weekends are all tasks that would involve the creation — and rapid neglect — of many **IMMUTABLE VALUES**. What may, in isolation, be considered a relatively minor space and time overhead can hit a program's performance and memory recycling limits when frequent.

Therefore, provide a companion class for the IMMUTABLE VALUE type that supports modifier methods. A MUTABLE COMPANION instance acts as a factory for IMMUTABLE VALUE objects. For convenience this factory can stand not only as a separate class, but can also take on some of the roles and capabilities of the IMMUTABLE VALUE.

The modifier methods allow for cumulative or complex state changes. They should be synchronized **COMBINED METHODS** [Henney2000c] if shared use in a multi-threaded environment is intended, but un-synchronized otherwise. A **PLAIN FACTORY METHOD** [Henney2003c] allows users to get access to the resulting **IMMUTABLE VALUE**:

```
class DateManipulator
{
    ... // constructors, modifier, and other query methods
    public synchronized void set(int year, int month, int day) ...
    public synchronized Date toDate() ...
    ... // private representation
}
```

A **MUTABLE COMPANION** should not, however, have an inheritance relationship with its corresponding **IMMUTABLE VALUE**. A mutable object is not truly substitutable for a type whose usage contract is founded on its immutability. Such non-substitutable inheritance would reintroduce the sharing problems eliminated by using an **IMMUTABLE VALUE**. Because the role of a **MUTABLE COMPANION** is to create **IMMUTABLE VALUES**, rather than to be used as a value in its own right, neither cloning nor copy construction is a requirement.

The core Java `String` and `StringBuffer` classes are canonical examples of an **IMMUTABLE VALUE** type with a **MUTABLE COMPANION**. With the exception of a transparent caching optimization for its hash code, a `String` instance is immutable and freely shareable across threads. A `StringBuffer` has a synchronized method interface with the standard object-as-string query, `toString`, doubling as the **PLAIN FACTORY METHOD**. Both classes are `final` and neither has any familial relationship with any class but `Object`.

A **MUTABLE COMPANION** is not always necessary, and should be provided only when the cost or complexity of working with **IMMUTABLE VALUES** alone becomes inappropriate. Appropriateness is not so much subjective as relative: it depends on both the type that the **IMMUTABLE VALUE** models and the specific application of that type.

Acknowledgments

This paper is derived from a previous article [Henney2003a].

I would like to thank Joel Jones for his shepherding of this paper for VikingPLoP 2003 and Neil Harrison for his comments following the conference. From the workshop at the conference I would like to thank Jacob Borella, Franco Guidi-Polanco, Alan O'Callaghan, and Titos Saridakis. For their later comments on the conference draft I would also like to thank Phil Hibbs and Hubert Matthews.

References

- [Alexander1979] Christopher Alexander, *The Timeless Way of Building*, Oxford, 1979.
- [Alexander2002] Christopher Alexander, *The Nature of Order, Book 1: The Phenomenon of Life*, Center for Environmental Structure, 2002.
- [Coplien+2001] James Coplien and Liping Zhao, "Symmetry Breaking in Software Patterns", *Springer Lecture Notes in Computer Science*, October 2001, <http://www.bell-labs.com/user/cope/Patterns/Symmetry/Springer/SpringerSymmetry.html>.
- [Fowler1999] Martin Fowler, *Refactoring: Improving the Design of Existing Code*, Addison-Wesley, 1999.
- [Fowler2003] Martin Fowler, *Patterns of Enterprise Application Architecture*, Addison-Wesley, 2003.
- [Henney2000a] Kevlin Henney, "Patterns of Value", *Java Report* 5(2), February 2000, available from <http://www.curbralan.com>.
- [Henney2000b] Kevlin Henney, "Value Added", *Java Report* 5(4), April 2000, available from <http://www.curbralan.com>.
- [Henney2000c] Kevlin Henney, "A Tale of Two Patterns", *Java Report* 5(12), December 2000, available from <http://www.curbralan.com>.
- [Henney2003a] Kevlin Henney, "Unvollendete Symmetrie in Java", *JavaSPEKTRUM*, May 2003, available in English as "Unfinished Symmetry" from <http://www.curbralan.com>.
- [Henney2003b] Kevlin Henney, "The Taxation of Representation", *The Road to Code* blog at *Artima*, 30th July 2003, <http://www.artima.com>.
- [Henney2003c] Kevlin Henney, "Factory and Disposal Methods", *VikingPLoP* 2003.
- [Henney2004] Kevlin Henney, "Opposites Attract", *Application Development Advisor*, to be published.
- [Wiki] <http://c2.com/cgi/wiki?ValueObject>.
- [Zhao+2003] Liping Zhao and James Coplien, "Understanding Symmetry in Object-Oriented Languages", *Journal of Object Technology* 2(5), September–October 2003, http://www.jot.fm/issues/issue_2003_09/article3.

Factory and Disposal Methods

A Complementary and Symmetric Pair of Patterns

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

May 2004

Abstract

complementary (of two or more different things) combining in such a way as to form a complete whole or to enhance or emphasize each other's qualities.

symmetry the quality of being made up of exactly similar parts facing each other or around an axis.

- correct or pleasing proportion of the parts of a thing.
- similarity of exact correspondence between different things.

The New Oxford Dictionary of English

Manual object creation may be in conflict with information hiding or instance-controlling requirements. The consequences of such separation and encapsulation can be addressed by the FACTORY METHOD pattern. Further control, economy, and symmetry may be found in the DISPOSAL METHOD pattern, in effect a mirror of FACTORY METHOD.

This paper revisits the classic FACTORY METHOD pattern, broadening the scope of this general pattern in line with the common usage of its name. Four specific variants are examined: PLAIN FACTORY METHOD, CLASS FACTORY METHOD, POLYMORPHIC FACTORY METHOD, and CLONING METHOD. FACTORY METHOD is accompanied by DISPOSAL METHOD, making the consideration of object lifecycle more clearly balanced. Two specific variants are examined: FACTORY DISPOSAL METHOD and SELF-DISPOSAL METHOD.

FACTORY METHOD and DISPOSAL METHOD are, in essence, quite high level whereas each of their variants is a more specific pattern. In the context of a specific pattern language or sequence it often makes more sense to zoom in on the specific variants rather than refer abstractly to the zoomed-out generalizations. This paper does not present a specific pattern language or a complete pattern sequence, more of a generative phrase or expression that can be incorporated and reified in a language or sequence.

Introduction

There is an inherent tension between data hiding and object creation. For example, if you hide object use behind an interface, how do you know which concrete class to use for creation? With any luck, if you are an experienced OO developer, you will now be sitting back in your seat, confident in the knowledge of at least one good answer. There is a good chance that this answer is FACTORY METHOD [Gamma+1995]:

Define an interface for creating an object, but let subclasses decide which class to instantiate. FACTORY METHOD lets a class defer instantiation to subclasses.

Well, you can lean forward now: this pattern deserves a revisit and revision to free it from a purely inheritance-centric view; it also warrants a counterpart to make it part of a greater design whole.

Both before and since the Gang of Four published FACTORY METHOD, the term *factory* has been used by programmers in a slightly broader sense, one not necessarily restricted to class hierarchies. Programmers will happily name a non-polymorphic method a *factory method*, so long as the obvious creational role indicated by a literal reading of the pattern name is followed. A factory is therefore generally a defined location with responsibility for encapsulating object creation.

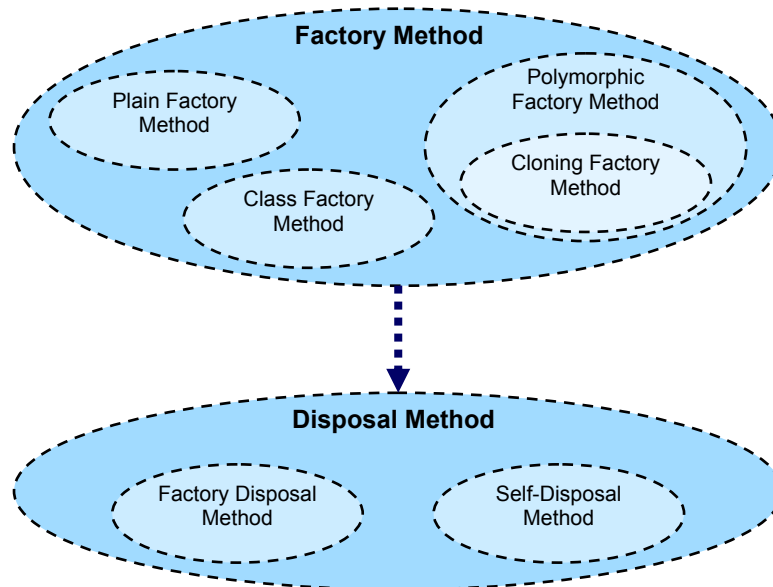
There is also something missing from the common discussion of object creation through factories: object disposal. Contemplating the sound of one hand clapping is a spiritual question not always well suited to the classically utilitarian materialism of objects. The absence of symmetry in the discussion of FACTORY METHOD suggests DISPOSAL METHOD. This relationship is not so much a tiny pattern language or short pattern sequence as a simple generative pattern phrase or subsequence, something that might be uttered in conversation in a language or included in a longer, domain-specific sequence. The symmetric pairing marries and mirrors FACTORY METHOD with DISPOSAL METHOD: one seeks closure in the other. As with any real mirror or marriage, the reflection is not perfect: in the detail of these patterns there is a great deal of independent variation that contrasts with the sketch-level symmetry.

Symmetry is a fundamental consideration [Alexander2002, Henney2003, Zhao+2003] that typically has the effect of simplifying a design, making it easier to comprehend and work with, not to mention more elegant and more whole. The simplification comes from the resulting regularity: something that is regular is easier to recall or second-guess than something that is not. Symmetry encourages consistency, becoming its own design map. This does not mean that designs should be globally and thoroughly symmetric but that, where transparency is not possible, a design should be predominantly and locally symmetric. A purely symmetric design is typically quite a dull one; a purely asymmetric one is unmemorable for different reasons.

A common question confronting both pattern authors and readers is that of specificity. Each occurrence of a pattern in a system is clearly highly specific, but at what level should the pattern itself be described? How specific should the problem be? What differentiates a variant of a pattern from the core pattern? Both the level and context of interest often dictate whether a pattern should be expressed at the most general level, e.g. a FACTORY METHOD is a method responsible for the creation of objects, or whether different flavors should be singled out and named, e.g. a POLYMORPHIC FACTORY METHOD defers the knowledge of exact creation type to be pushed down a class hierarchy. In the context of a specific pattern language it tends to make sense to focus on specific variants because the problems they address in the context of the language are similarly specific, e.g. while FACTORY METHOD offers a general heading for describing an approach to creational

encapsulation, a PLAIN FACTORY METHOD does not solve quite the same problem as a CLASS FACTORY METHOD, nor does it have quite the same consequences.

In this paper the focus is not a pattern language but on two patterns that, at a general level, form part of a vocabulary for object lifecycle design. PLAIN FACTORY METHOD, CLASS FACTORY METHOD, and POLYMORPHIC FACTORY METHOD are presented in the context of the more general FACTORY METHOD, with CLONING METHOD as a further flavor of POLYMORPHIC FACTORY METHOD. FACTORY DISPOSAL METHOD and SELF-DISPOSAL METHOD are presented in the context of DISPOSAL METHOD. The following diagram illustrates the relationships:



A low-ceremony pattern form is used. In general, the focus of the patterns is on the statically typed OO model of Java, C++, and C#. Specifically, code fragments are in Java.

FACTORY METHOD

Encapsulate the concrete details of object creation by providing a method for object creation instead of letting object users instantiate the concrete class themselves.

Problem: Code that depends on instances of a class or from a class hierarchy may need to create the objects itself. This may not be as easy as simply using a new expression:

- What if the creational logic cannot be contained easily inside a constructor? What if external validation is needed or object relationships must be established that might be considered beyond the scope of the object's immediate responsibility? For example, the constructor of a bank account object should not be responsible for allocating its account number, running a credit check, or ensuring that the instance is persisted by its associated bank.
- What if the class must be instance controlled, so that unconstrained use of new would be inappropriate? For example, neither ENUMERATION VALUES [Henney2000a, Henney2000b] nor SINGLETON objects [Gamma+1995] should be created manually or directly by their users.
- What if the appropriate concrete class is unknown to the user because the user manipulates an object only via a declared interface and not via its concrete class? For example, an object whose actual type depends on the actual type of another object should not cause the user to copy and paste repetitious type-dependent code. Cascaded if else if statements that hardwire instanceof, dynamic_cast, or is runtime type checks are a good way of obscuring a method's intent and reducing a class's openness and extensibility.
- A more specific example of needing object creation in the presence of hierarchical abstraction is the wish to take a proper copy of an object without knowing its concrete type.

These different scenarios are unified under a common pair of opposing forces:

- Objects are most simply and intuitively created using a new expression, specifying a concrete class and constructor arguments. This provides the user of a class with full control over instantiation.
- Direct object creation may inadvertently obfuscate and reduce the independence of the calling code if any of the necessary ingredients for correct object creation are not readily available. The concrete class, the full set of constructor arguments or the enforcement of other constraints may not be known at the point of call; to require them would increase the complexity of the calling code.

Solution: Provide a method for fully and correctly creating the appropriate object instead of relying on a new expression. The knowledge of creation is encapsulated within this *factory method*. The ability to create instances directly is hidden from the caller either by making constructors non-public or by pushing it down a class hierarchy.

However, unless created specifically for the purpose, including the role of *creator* in a class's repertoire can sometimes be considered an addition that dilutes its cohesiveness. The solution is certainly more encapsulated than the alternatives, but the cohesion can be considered a little lower than in a design where such creation was never needed.

There are three basic and one extended variant of FACTORY METHOD that determine how the different roles of *product* and *creator* (also known as the *factory*) are realized:

- PLAIN FACTORY METHOD: The *creator* is an object — not necessarily in a class hierarchy — and the type of the *product* either is fixed or varies only with environmental settings

or the arguments to the *factory method*. A PLAIN FACTORY METHOD implementation is normally just a case of providing an ordinary, possibly `final` or `sealed`, method that creates instances of another class, with no specific intent to be inherited or overridden.

- CLASS FACTORY METHOD: The *creator* is a class rather than an object, and so the *factory method* is static. The *creator* is often the same class as the *product* object type, which is not normally defined in a class hierarchy. Direct creation of *product* objects is often prevented by ensuring that instance constructors are non-public. CLASS FACTORY METHOD pattern is also known as STATIC FACTORY METHOD [Bloch2001, Haase2002].
- POLYMORPHIC FACTORY METHOD: The possible types of the *product* object are defined in a class hierarchy. Mirroring the hierarchy of what is created, an interface for *creator* objects is provided, offering the *factory method* abstractly, and the responsibility for creation is deferred to an implementing subclass. The knowledge of which type of *product* is required is contracted out to the *creator* hierarchy, removing the need for a closed and clumsy instanceof solution. This FACTORY METHOD variant is the classic Gang of Four version.
- CLONING METHOD: The *product* class is the same as the *creator* class. However, unlike a CLASS FACTORY METHOD the relationship is properly reflexive: the *creator* is an instance of the class, rather than the class, so that its result is another object of its own type. To be precise, the *product* is a proper copy of its *creator*. A CLONING METHOD is a specific kind of POLYMORPHIC FACTORY METHOD.

A PLAIN FACTORY METHOD is fairly straightforward in its common form. The *product* is normally concrete, and may have only non-public constructors:

```
public class ConcreteProduct
{
    ...
    private ConcreteProduct(...) ...
}
```

The *creator* is also normally concrete, and has sufficient access to the *product* type to create instances:

```
public class ConcreteCreator
{
    public ConcreteProduct create()
    {
        return new ConcreteProduct(...);
    }
    ...
}
```

For the bank account example, a bank object would adopt the role of *creator* and an account object would be a *product*. The bank would hide the details of creation and the account class would prevent general creation by users. The design is encapsulated between the two classes and need not involve any inheritance.

The form of a CLASS FACTORY METHOD is simple, with the class as *creator* and its instances as product:

```
public class ConcreteProduct
{
```

```
public static ConcreteProduct create()
{
    return new ConcreteProduct(...);
}
...
private ConcreteProduct(...) ...
}
```

For example, as an example of symmetry, a Java class that supports a meaningful `toString` override could consider providing a `fromString` or `valueOf` CLASS FACTORY METHOD in preference to a public `String` constructor. Using a CLASS FACTORY METHOD names the conversion concept explicitly. It sets string-based creation apart from other constructors to emphasize the inverse relationship with the common `toString` method.

The general POLYMORPHIC FACTORY METHOD has the most intricate detail, spanning two class hierarchies where the previous two variants typically address one or two classes on their own. There is the *product* hierarchy:

```
public interface Product
{
    ...
}
...
public class ConcreteProduct implements Product
{
    ...
}
```

And there is the *creator* hierarchy:

```
public interface Creator
{
    Product create();
    ...
}
...
public class ConcreteCreator implements Creator
{
    public Product create()
    {
        return new ConcreteProduct(...);
    }
    ...
}
```

Where the caller and the used class hierarchy are one and the same, TEMPLATE METHOD [Gamma+1995] is often used:

```
public abstract class ProductUser
{
    public void useNewProduct()
    {
        Product produce = create();
        ...
    }
}
```



```
    protected abstract Product create();
}
...
public class ConcreteProductUser implements ProductUser
{
    protected Product create()
    {
        return new ConcreteProduct(...);
    }
}
```

A degenerate arrangement of POLYMORPHIC FACTORY METHOD is CLONING METHOD (or VIRTUAL COPY CONSTRUCTOR or SELF-FACTORY METHOD), which is normally used in its own right to support polymorphic copying but can also be found in support of a PROTOTYPE approach to object creation [Gamma+1995, Coplien1992], with which it is often confused. In CLONING METHOD the types of the *product* and the *creator* are the same, and the *creator* instance provides itself as the model from which a new instance is built:

```
public class Product implements Cloneable
{
    public Object clone()
    {
        ... // cloning carried out and resulting object returned
    }
    ...
}
```

The cloning is instigated directly by the object user:

```
...
public void takeSnapshot(Product other)
{
    snapshot = (Product) other.clone();
}
...
```

In PROTOTYPE an object is held by a factory to be used as the prototypical instance from which new factory products are built. This may involve a CLONING METHOD:

```
public class ConcreteCreator implements Creator
{
    public Product create()
    {
        return (Product) prototype.clone();
    }
    ...
    private Product prototype;
}
```

Or not:

```
public class ConcreteCreator implements Creator
{
```

```
public Product create()
{
    return new Product(prototype.attributes());
}
...
private Product prototype;
}
```

In the second fragment the factory product is created using the attributes of the prototype object and explicitly constructing the object. In both cases the prototype is used as the instance on which factory products are based, but only in the first does the implementation mechanism qualify as a **FACTORY METHOD**.

DISPOSAL METHOD

Encapsulate the concrete details of object disposal by providing an explicit method for clean up instead of letting object users either abandon objects to the tender mercies of the garbage collector or terminate them with extreme prejudice and delete.

Problem: How should objects with significant clean-up behavior be disposed of after use? For garden-variety objects, the usual mechanism of the language for disposing of objects is normally sufficient. However, for some kinds of objects, such as resources, this may not be enough. Just as a **FACTORY METHOD** may hide details of an object's creation that cannot be handled fully by a constructor, details of an object's destruction may go further than can be adequately expressed by conventional finalization, whether a `finalize` method or destructor.

A resource can be defined by its use and context rather than in terms of its abstraction. A resource is any object that could easily become scarce in a system and whose scarcity would cause problems. Therefore, a resource can be defined liberally as anything that should be returned after acquiring and using it. For example, memory in C and C++ is a resource, but in a well-endowed Java or C# program it is typically not. However, in a smaller environment memory again becomes a resource. In the common application of a **FACTORY METHOD**, instance creation is controlled but object disposal is not. Because resource usage may need to be conserved and resources recycled, the user of a resource should have a clear contract for how a resource's usage lifetime is bounded.

In C++ an explicit `delete` by a factory-product user is asymmetric with the hidden `new` in the factory. A `delete` expression deterministically triggers the end of an object's life, which may be a somewhat more severe disposal than is wanted: it is difficult to recycle an object if it no longer exists. There is also no guarantee that an object was created using a plain `new`, hence a `delete` may be precisely the wrong action even to end an object's life. Memory that is acquired independently of construction would rely on a placement `new` expression for construction and an explicit destructor call for destruction; there is no corresponding `delete` expression.

Java and C# programmers can discard objects for later automatic collection by the garbage collector. It is, however, naïve to assume that a GC system solves all memory and resource management issues out of the box [Bloch2001]:

When you switch from a language with manual memory management, such as C or C++, to a garbage-collected language, your job as a programmer is made much easier by the fact that your objects are automatically reclaimed when you're through with them. It seems almost like magic when you first experience it. It can easily lead to the impression that you don't have to think about memory management, but this isn't quite true.

It is possible to further dilute confidence in totally transparent GC: your objects *may* be reclaimed automatically. There is little guarantee that they will be claimed in a timely manner, or even at all — although such low (or non-existent) quality-of-service would find favor with few developers. Frequent creation of fine-grained objects, such as iterators or value objects, can potentially lead to inefficient use of resources or even resource exhaustion.

With respect to resources, a specific and timely clean-up action may be required, but in the absence of explicit control over the tail end of an object's life this cannot be made implicit — and whatever the problem, Java's `finalize` is rarely the answer. GC addresses the issue of object collection to make memory resourcing transparent, but this does not apply to other resources.

Solution: Provide a method for explicit clean up and disposal of an object. Mirroring FACTORY METHOD, DISPOSAL METHOD answers the question of who is responsible for the clean up and disposal of an object by making the clean up an explicit operation for the user. Just as the user requested an object for use, they must also mark the end of its use.

DISPOSAL METHOD may be expressed as one of two basic variants:

- **FACTORY DISPOSAL METHOD:** Provide a method on the factory that originally created the object. The knowledge of an object's lifecycle is isolated in a single place, which allows transparent instance control, such as an object pool that caches and recycles objects.
- **SELF-DISPOSAL METHOD:** Provide a method on the object to be disposed of. This method either performs the clean up itself or, if a factory was involved in the object's creation, it returns of the object to its maker.

An obvious and negative consequence of this pattern is that the user must remember to both make the call and make the call exception safe. This situation is tedious and error prone, and can be ameliorated through additional encapsulation, such as a COMBINED METHOD [Henney2000c], EXECUTE-AROUND METHOD [Henney2001a], or a COUNTING HANDLE [Henney2001b]. Where instance control is neither about resource management nor in the hands of an object user, no disposal is required, so DISPOSAL METHOD is not necessarily appropriate.

FACTORY DISPOSAL METHOD is the natural complement of FACTORY METHOD, and its truest reflection:

```
public interface Creator
{
    Product create();
    void dispose(Product toDisposeOf);
    ...
}
```

In the bank account example closing an account would be a good example of a FACTORY DISPOSAL METHOD. The code that decides to dispose of a factory-created object must have access to both the *creator* and the *product*. This not only means that the lifetime of the product must be contained within that of its creator, but that the caller of the DISPOSAL METHOD is expected to co-ordinate the disposal correctly, i.e. it should ensure that it matches the right product with the right factory. This is often not a significant issue because factories and products are normally well matched in terms of types and scope usage. However, this slight increase in coupling can present a potential liability for some programs.

SELF-DISPOSAL METHOD is sometimes known as an EXPLICIT TERMINATION METHOD [Bloch2001]:

```
public interface Product
{
    void dispose();
    ...
}
```

Where a SELF-DISPOSAL METHOD is simply a forwarder to a FACTORY DISPOSAL METHOD, it clearly has to retain some kind of reference to the factory of origin. In such a case it

successfully encapsulates the knowledge of its origin and therefore the correct co-ordination of *product* to *creator*. The product user — or, to be precise, disposer — is freed from maintaining and using this extra reference. Although this offers a better encapsulation of the constraints governing the factory-product pairing, it can be seen as slightly less cohesive because responsibility for disposal is represented in the product interface, which would otherwise be focused purely on matters of product usage.

In C++ a DISPOSAL METHOD displaces the use of a public `delete` for the product type in question. The lifetime of an object is no longer subject to the operators in the language but to the higher-level interfaces and object lifecycle choices of a specific application. To ensure no clash between the use of a DISPOSAL METHOD and the common use of a `delete`, the destructor of the target object should not be public at the level of the interface. This restriction prevents any attempt to mix the `delete` and DISPOSAL METHOD models at compile time.

Acknowledgments

This paper is derived from a previous article [Henney2002].

I would like to thank Klaus Marquardt for his thorough and insightful shepherding of this paper for VikingPLoP 2003, Mark Radford for his additional comments both before and after the conference, and Neil Harrison for his comments following the conference. From the workshop at the conference I would like to thank Jacob Borella, Franco Guidi-Polanco, Alan O'Callaghan, and Titos Saridakis.

References

- [Alexander2002] Christopher Alexander, *The Nature of Order, Book 1: The Phenomenon of Life*, Center for Environmental Structure, 2002.
- [Bloch2001] Joshua Bloch, *Effective Java*, Addison-Wesley, 2001.
- [Coplien1992] James O Coplien, *Advanced C++: Programming Styles and Idioms*, Addison-Wesley, 1992.
- [Gamma+1995] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1995.
- [Haase2002] Arno Haase, "Idiome in Java", *JavaSPEKTRUM* 36, February 2002.
- [Henney2000a] Kevlin Henney, "Patterns of Value", *Java Report* 5(2), February 2000, available from <http://www.curbralan.com>.
- [Henney2000b] Kevlin Henney, "Value Added", *Java Report* 5(4), April 2000, available from <http://www.curbralan.com>.
- [Henney2000c] Kevlin Henney, "A Tale of Two Patterns", *Java Report* 5(12), December 2000, available from <http://www.curbralan.com>.
- [Henney2001a] Kevlin Henney, "Another Tale of Two Patterns", *Java Report* 6(3), March 2001, available from <http://www.curbralan.com>.
- [Henney2001b] Kevlin Henney, "C++ Patterns: Reference Accounting", *EuroPLoP 2001*, July 2001, available from <http://www.curbralan.com>.
- [Henney2002] Kevlin Henney, "Symmetrie in Java", *JavaSPEKTRUM*, July 2002, available in English as "The Importance of Symmetry" from <http://www.curbralan.com>.
- [Henney2003] Kevlin Henney, "Five Possible Things after Breakfast", *The Road to Code* blog at *Artima*, 23rd June 2003, <http://www.artima.com>.
- [Zhao+2003] Liping Zhao and James Coplien, "Understanding Symmetry in Object-Oriented Languages", *Journal of Object Technology* 2(5), September–October 2003, http://www.jot.fm/issues/issue_2003_09/article3.

Transformational Patterns for the Improvement of Safety Properties in Architectural Specification

Lars Grunske

Department of Software Engineering and Quality Management
Hasso-Plattner-Institute for Software Systems Engineering, University of Potsdam
Prof.-Dr.-Helmert-Straße 2-3, D-14482 Potsdam (Germany)
+49(0)3315509152
lars.grunske@hpi.uni-potsdam.de

Abstract

Over the past years, the functionality of technical systems has been increasingly implemented in software components. These software components have to fulfill requirements regarding non-functional properties (NFPs), such as safety, availability, reliability and temporal correctness. Along with the rising need for increased functionality, this led to an increased complexity of these systems. As a result, the architectural specification and its quality have become important for the success of the development process. For the non-functional requirements, the quality of the architecture can be determined by an architecture evaluation. If the system does not fulfill these requirements, the architecture must be modified to improve the non-functional properties. To support this process we present a set of transformational patterns, that help to restructure architectures to especially improve the safety properties of the systems under development.

1. INTRODUCTION

Embedded systems in the automotive, avionic, medicine or railway sectors have to fulfill requirements regarding non-functional properties, such as safety, security, availability, reliability, and performance. To prove that the system meets its non-functional requirements, analytic quality assurance techniques (i. e. fault tree analysis-FTA, failure mode and effect analysis-FMEA or performance analysis) are used. Due to economic reasons, this analysis should take place as early as possible in the development process of the system. This is obviously immediately after the construction of the software/hardware architecture because at this point we are able to assess non-functional system properties for the first time.

If such an assessment indicates that a system does not fulfill the non-functional requirements, the system architecture has to be changed to improve the corresponding non-functional properties. For this, the utilization of transformational patterns, which are similar to code level refactorings as proposed by [Folwer 99], is helpful. A transformational pattern is a refactoring at an architectural level [Grunske 2003], which only changes parts of the architectural model without the alteration of the provided external behavior of the system. Thus, transformational patterns can be considered recipes for improving the quality of the software architecture. This paper presents a set of transformational patterns, which especially address the improvement of safety aspects.

Before we come to the presentation of these patterns, we first introduce the underlying safety concepts to clarify the terms used in the patterns. Then, we introduce a notation for the description of transformational patterns.

1.1 The safety concept

The term safety can be defined for a system as freedom from unacceptable risks [EN 50126]. The term risk is defined as likelihood and severity of an accident (damage of life, property or environment) [Leveson 95]. The acceptance of risk depends on the society, the customer, and the laws as well on the likelihood and severity of the resulting accident. As an example, an accident in a nuclear power plant can cost the lives of many people. Therefore, the risk of an accident even with a very low likelihood is often not accepted.

To reduce the probability of an accident the preconditions under the control of the system designer must be known. These controllable preconditions are called hazards, which are states or conditions of a system that, together with the uncontrollable conditions in the environment, lead to an accident [Leveson 95]. An example of a hazard is a car with a defective air-bag

system. It depends on the environment, whether this hazard leads to an accident. If the driver does not crash the car, the defective air-bag system cannot lead to bodily injury of the driver.

A safety requirement is a formal description of a hazard combined with the tolerable probability of this hazard. The tolerable hazard probability must be determined in the risk analysis, in such a way that for all hazards of the system the combined risk is acceptable.

The hazard description can be classified with the following three categories [Levenson 95] [Avizienis et al. 01] and [Laprie 92]:

1. The system is not available
2. The system generates an incorrect output
3. The system misses a hard deadline

In the first category, the system can causes an accident, if the system fails to react to an event. Such a system, can be for example, a system without a safe state (airplane) or a medical system (cardiac pacemaker or artificial breathing system). In the second category, hazards are grouped where a system causes an accident, because it behaves incorrectly. An example is a hazard is a railway or traffic signal that signals green when it should signal red. The third category describes hazards caused by timing failures. In this category, systems reacting too early (cardiac pacemaker) or too late (air bag system) serve as examples.

1.2 Transformational pattern

A transformational pattern describes in an abstract way the structural changes of software architectures. These changes can be:

- the removal or the addition of architectural elements
- the redirection of a connection between the architectural elements

For the representation of a transformational pattern a graphical notation is used. This notation is denoted as T-notation, due to the fact it groups the architectural elements of a pattern into three parts, shaping a T. The semantic of this notation implies that all elements or connections on the bottom-left-side of the T must be removed from the architecture. All elements or connections on the bottom-right-side of the T must be added to the architecture. The elements above the T remain unaffected and serve as gluing points between the rest of the architecture and the new added elements. That is the reason why they are redundantly contained on the upper left and the upper right side.

An example for a transformational pattern in the T-notation is given in Figure 1-1. This abstract pattern describes that the components of the type A and B, the two ports with the type A and the connection between them must be removed. The ports (type A, B and C) above the T are remained unaffected and the component with the type C must be added to the software architecture. The application of this pattern is presented in Figure 1-2 for a concrete architecture.

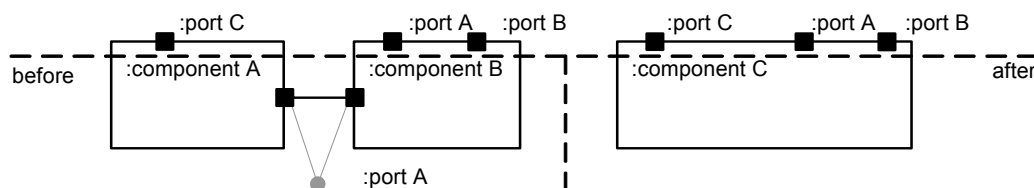


Figure 1-1 Example of a transformational pattern in the T-notation

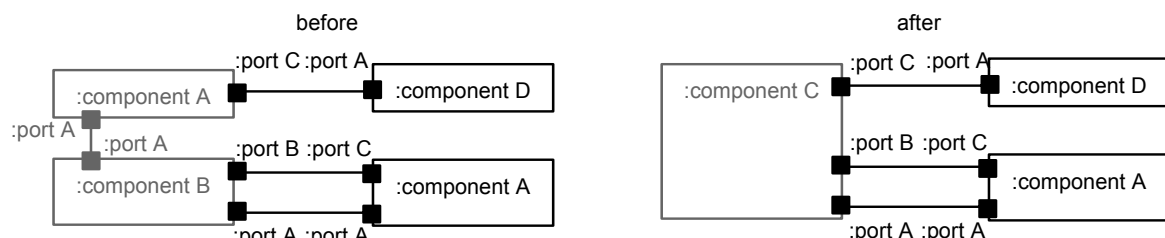


Figure 1-2 Example of the application of a transformational pattern

2. THE PATTERN LANGUAGE

2.1 Scope

The scope of this pattern language is to give a general overview of the concepts we utilize to increase the safety properties at the architectural level. Therefore, this pattern language contains a set of transformational patterns, which reduces the probability of a hazard. Together with the structure of the pattern language, this will guide software architects to improve the safety properties of architectures with safety critical focus.

Because of their usage in the architectural design phase, these patterns are very high level and present the basic concepts. Thus, they only serve as a guideline for the restructuring of the architecture. For detailed information about the implementation, we present several links for further reading, because this is not the aim of this pattern language. Further for some case studies and examples where the presented patterns are applied, we suggest [Douglass 02], [Huang, Kintala 93], [Pont 01] and [Leveson 95]

2.2 A Road Map

The patterns of this language are grouped into three different pattern types. These pattern types and the associated patterns are depicted in the following table.

| Pattern Type | Pattern |
|-------------------------------|--|
| Run-Time Fault Prevention | Protected Single Channel (2.3) Recovery Block (2.4) Multi Channel Redundancy with Voting (2.5) Two Channel Redundancy (2.6) |
| Design-Time Fault Prevention | Process Fusion (2.7) Hardware Platform Substitution (2.8) Hardware Platform Reassignment (2.9) |
| Errors and failures detection | Actuation-Monitor (2.10), Integrity Check (2.11) Watchdog (2.12) |
| Failure mitigation | Restart System Inform Operation Personal Goto Safe State |

The main patterns of this language are of the pattern type run-time fault prevention and design-time fault prevention. These patterns are used to restructure the architectural to improve the safety properties. Therefore, they present different solutions to reduce the probability of a safety critical hazard. Thus, the problem section is unique for all of these patterns.

The patterns of the type error and failure detection and failure mitigation are sub patterns. They build refinements of a pattern of the type architectural restructuring. All patterns and their relation are further illustrated in Figure 2-1.

For the selection of a suitable pattern and the application of a pattern, the following information, which can be determined by an architecture evaluation as presented in [Birolini 99],[Papadopoulos et al. 01],[Liggesmeyer 00],[McDermid, Pumfrey 95], [Fenelon et al. 94], is primarily necessary:

1. Which architectural elements cause a hazard with a higher probability than the tolerable hazard probability?
2. Which failure of a certain architectural element leads to a certain hazard?

Answers to the first question are necessary to find an occurrence in the architecture, where a pattern can be applied. This occurrence serves as a precondition for a pattern application. To choose an appropriate pattern furthermore answers to the second question are needed. For this the failures of an architectural element are further classified. This classification is similar to the failure classification at the system level:

1. An architectural element is not available
2. An architectural element generates an incorrect output
3. An architectural element misses a hard deadline

Based on this classification for each failure type a set of possible patterns is presented in the pattern language. Thereby, this failure types guide the software architect to select a suitable pattern. In addition to that, the development budget and the

technical feasibility serve as guidelines for pattern selection. The underlying economic and technical rationales are part of each pattern description.

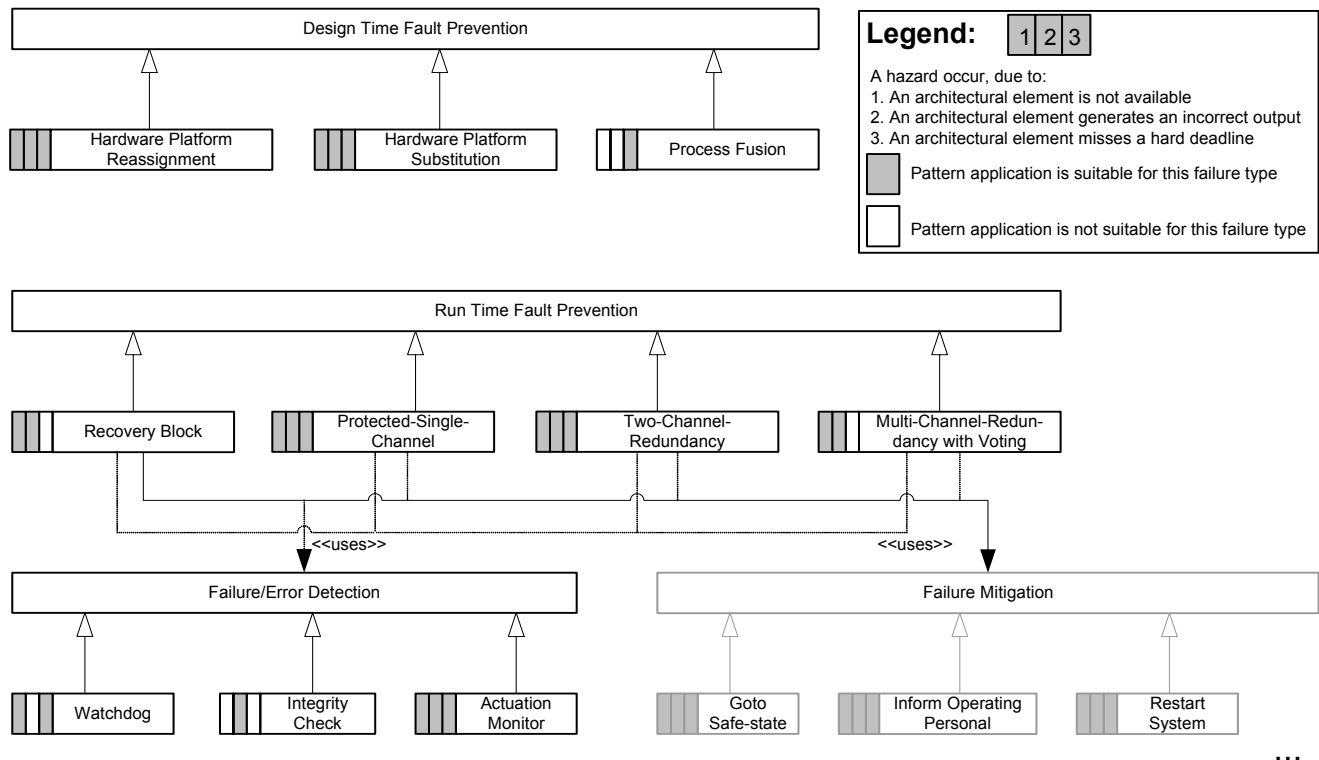


Figure 2-1 Pattern Language Structure

The selection of a suitable failure mitigation pattern also depends on the application area. As an example for the failure mitigation, a railway control system can achieve a safe state by stopping all trains. It is clear that this strategy would not work for a flying airplane, because it does not have a directly reachable safe state. Therefore, this pattern application is not appropriate in avionics. This observation leads to the conclusion that patterns for the failure mitigation type need to be extended with respect to the failure mitigation patterns for each relevant application field. This would enable the pattern language to be applicable to all types of embedded systems. This remains for future work.

2.3 Protected Single Channel

Aliases

Single Channel Protected Design [Douglass 02]

Problem

How can the probability of hazards be reduced that are caused by an erroneous behavior of a software component in the deployed system?

Context

This pattern can be applied if the erroneous behavior (failure) that causes the problematic hazard is precisely identified and is in one of the following types:

- An architectural element is not available.
- An architectural element generates an incorrect output.
- An architectural element misses a hard deadline.

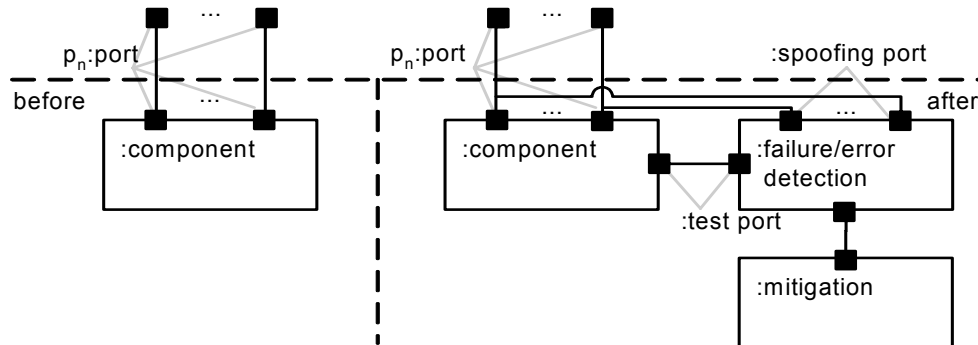
Furthermore, this failure must be detectable at the runtime of the system and mitigation strategies can be applied which reduce the probability of the failure propagation, reduce the severity of the losses, or eliminate the failure.

Forces

- Software components have become so complex that we cannot assume them to be error free.
- In safety critical applications, each failure of the software can lead to a harmful accident.
- If a failure can be detected, it can be handled.

Solution

To increase the safety properties the architecture is extended by a *failure/error detection* component. This component detects failures or errors that lead to a safety critical behavior. If this component detects the specific error or failure, a message is sent to the *mitigation* component that removes the error or mitigates the failure. The following architecture diagram illustrates the structure before and after the application:



This diagram shows that the *failure/error detection* component can access all relevant outputs of the observed component via the *spoofing ports*. Based on these ports the component detects the failures. The *test ports* of the *failure/error detection* component detect errors in the component which can cause a safety critical failure. There is a connection between the *failure/error detection* and the *mitigation* component to inform the *mitigation* component in case of an error or failure. If the *mitigation* component handles all identified safety critical failures, the probability of the hazard is reduced.

Rationales

- Economic rationales
 - o No further hardware is needed.
 - o With the *failure/error detection* and the *mitigation* component further software components or strategies must be developed.
- Technical rationales
 - o Only single point failures are detected.
 - o The failures must be understood and identified.

Resulting Context

The application of this pattern results in two other problems: how to detect concrete failures or/and errors and how to mitigate them. These problems must be solved to implement the failure/error detection and the mitigation component. For this the Actuation-Monitor (2.10), the Integrity Check (2.11) and the Watchdog (2.12) patterns are possible refinements for the failure/error detection component. For the mitigation, the mitigation patterns Restart System, Inform Operation Personnel and Goto Safe State are appropriate.

Related Patterns

Recovery Block (2.4), Multi-Channel-Redundancy with Voting (2.5) and Two-Channel-Redundancy (2.6)

2.4 Recovery Block

Aliases

Recovery Conversation, Roll Forward [Saridakis 02]

Problem

How can the probability of hazards be reduced that are caused by a systematic failure of software components in the deployed system, if the function of these components is critical for safety?

Context

It is possible to apply this pattern if the erroneous behavior (failure) that causes the problematic hazard is precisely identified and can be detected at runtime. A systematic error made in the development of a software component must be the reason for the failure.

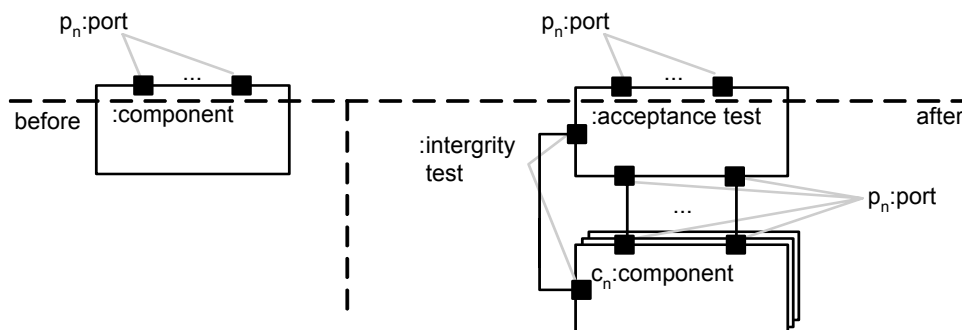
Forces

- For software components in a life critical system the availability and correctness of these components is fundamental for safety.
- Further, if a component is identified as faulty, it cannot be trusted anymore.

Solution

As a solution to this problem the recovery block concept was introduced by [Randel 75] and the further development is presented in [Randell, Xu 95]. The basic idea of this pattern is to use multiple heterogeneous developed components operating in parallel on a single hardware platform. All components are getting all information from the environment, but one of them is the primary component and the others are backup components. The primary components perform the desired operations that are checked by an *acceptance test* component. This *acceptance test* component can check the primary component itself to detect errors. If it detects an error or a failure, the primary component becomes one of the backup components and the next component in the pool of backup components becomes the primary component.

To get protection from failures caused by systematic errors the heterogeneous components must fail independently. For this, these components should be heterogeneous [Avizienis 85]. In order to realize this different teams have to be responsible for their implementation. The reduction of a systematic error is achieved by the diverse implementation of the software components [Mitra et al. 99]. However, [Knight, Leveson 86] point out that a diverse implementation does not detect all systematic errors. Different development-teams made similar faults and therefore the different versions do not fail independently.



Rationales

- Economic rationales
 - o No necessity for additional hardware.
 - o Additional and diverse implementations of the software-components are needed.
- Technical rationales
 - o Different development teams must implement diverse components that fail independently.
 - o The correctness of the *acceptance test* component is essential for the safety properties after the application of this pattern.

Resulting Context

The application of this pattern leads to the problem which strategy should be applied to detect failures and errors in the *acceptance test* component. To solve this problem the Actuation Monitor (2.10), Integrity Check (2.11) and Watchdog (2.12) patterns are possible refinements.

Related Patterns

Protected-Single-Channel (2.3), Two-Channel-Redundancy (2.6), Multi-Channel-Redundancy with Voting (2.5)

2.5 Multi-Channel-Redundancy with Voting

Aliases

Homogeneous redundancy, Heterogeneous Redundancy and Triple Modular Redundancy [Douglass 02], Fail-Stop Processor [Saridakis 02]

Problem

How can the probability of hazards be reduced that are caused by random or wear-out failures of the hardware platform on which a safety critical component is executed?

Context

The pattern application is possible, if non-known failures of software components can lead to the problematic hazard. These failures can be of one of the following types:

- An architectural element is not available.
- An architectural element generates an incorrect output.

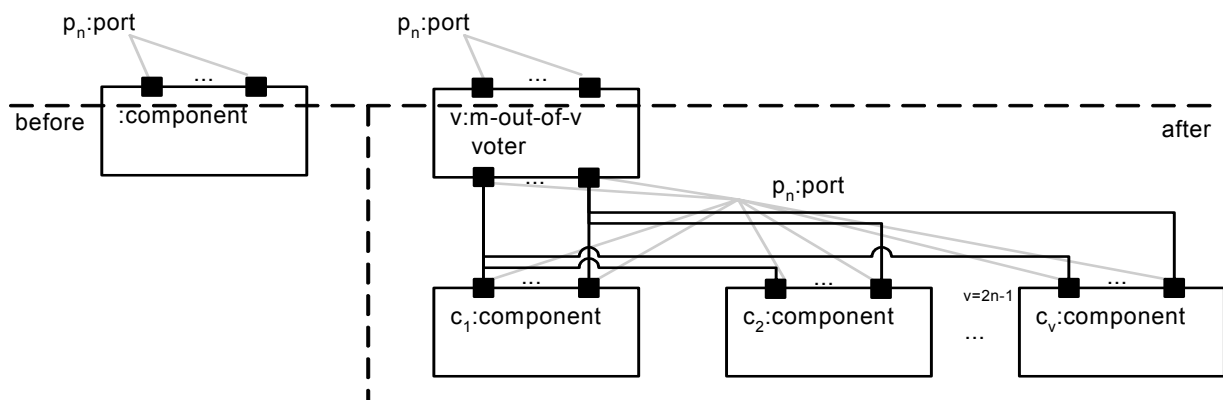
The causes of these failures of the software component are random or wear-out failures of the hardware platform on which the component is processed.

Forces

- Random or wear-out failures of the hardware platform can influence all software components that are executed on this hardware platform.
- Some safety critical systems cannot shut down, because they have high availability requirements or they do not have a safe state.
- Due to these high availability requirements the hardware platforms must be maintained at the runtime of the system.

Solution

The component that causes the safety critical behavior is substituted by multiple components on different hardware platforms and a comparator component (*m-out-of-v voter*) that gets the inputs from the environment and generates multiple messages for the components. Based on these messages the components compute the results and send them back to the comparator component, that chooses the message to be sent to the environment by a majority voting. For the realization of this pattern, often 3 components and a 2-out-of-3 voting are used [Douglass 02].



The pattern application improves the reliability of the system, if the comparators hardware platform and the communication channel are more reliable than the components hardware platforms.

Extension

This pattern can provide protection against systematic failures. Therefore, the redundant components must be heterogeneous implemented by different development teams. Note, that a diverse development does not provide fail independent components as stated in [Knight, Leveson 86].

Rationales

- Economic rationales
 - o Additional hardware is needed
 - o If the pattern is applied to protect the system against systematic errors in the development process additional and diverse software-components are needed

Resulting Context

If the comparator component detects a failure, a mitigation strategy can be implemented furthermore. For this a failure mitigation pattern Restart System, Inform Operation Personal and Goto Safe State can be used.

Related Patterns

Protected-Single-Channel (2.3), Two-Channel-Redundancy (2.6), Recovery Block (2.4)

2.6 Two-Channel-Redundancy

Aliases

Switch to back up

Problem

How can the probability of hazards be reduced that are caused by random or wear-out failures of the hardware platform, on which a safety critical component is executed?

Context

Similar to the pattern Protected-Single-Channel (2.3), the application of this pattern is possible, if it is possible to precisely identify the erroneous behavior (failure) that causes the problematic hazard. The failure must be detectable at runtime and may be one of the following: First, a random or wear-out error of the hardware platform may be the reason for it. Second, a systematic error made in the development of a software component is the cause of the failure.

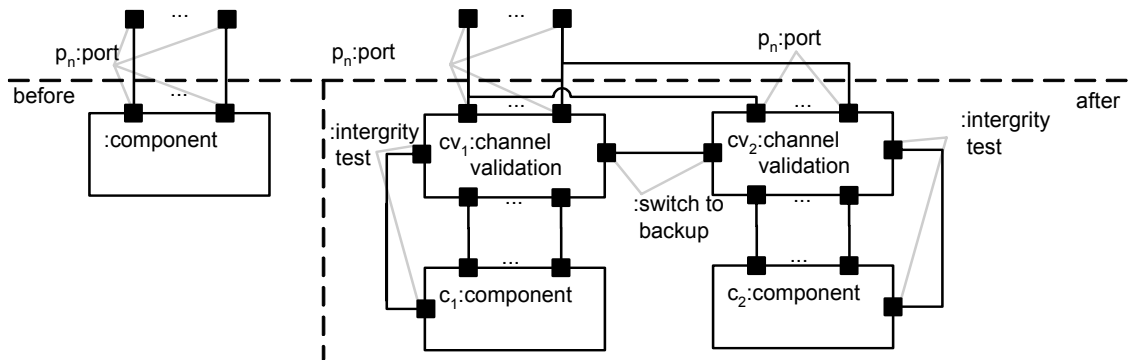
Forces

The forces are identical to the pattern Multi-Channel-Redundancy with Voting (2.5). However, the difference is:

- The utilization of multiple hardware platforms as they are used in the pattern Multi-Channel-Redundancy with Voting (2.5) can often not be applied due to high cost
- Further the utilization of multiple hardware platforms is not possible, due to space or weight limitations

Solution

Use two components operating in parallel on different hardware platforms. Both components are getting all information from the environment, one active and one passive. The active component checks its operation with a *channel validation* component. The results of this validation are sent to the *channel validation* of the passive channel via the connection between the *switch to back up* ports. If a failure occurs, an error is detected, or the active channel omits to send the results, the passive channel becomes active. In this case, the former active channel must be informed. To check the correct operation of one channel the utilization of several strategies is possible. These are similar to the failure/error detection mechanisms used in the Protected-Single-Channel (2.3) pattern. Thus, it is possible to substitute the *channel validation* with a failure detection pattern, which realizes a switch to back up strategy, if a failure is detected.



By the application of this pattern, one component c_1 or c_2 is still available in case of random or wear-out failures of the hardware platform of the other component.

Extension

This pattern can also be used to provide protection against systematic failures. Therefore, the redundant components must be heterogeneous implemented by different development teams. Note, that a diverse development does not provide fail independent components as stated in [Knight, Leveson 86].

Rationales

- Economic rationales
 - o Additional hardware is needed.
 - o For a protection against systematic errors: an additional diverse implementation of the software-component.
- Technical rationales
 - o Necessity for communication mechanisms between the two channels.

Resulting Context

The application of this pattern leads to the demand for a failure or error detection strategy. It is necessary to address this problem in order to implement or refine the failure/error detection component. Therefore, the Actuation Monitor (2.10), Integrity Check (2.11) and Watchdog (2.12) pattern can be put into practice.

After the application of this pattern, it is clear that the active channel becomes passive and the passive channel becomes active in case of a critical error (switch to backup). However, in this case another failure mitigation strategy can be applied.

Related Patterns

Protected-Single-Channel (2.3), Recovery Block (2.4), Multi-Channel-Redundancy with Voting (2.5)

2.7 Process Fusion

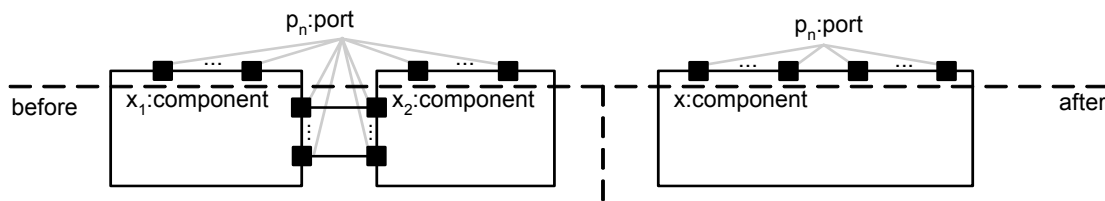
Problem and Context

In architectures with a large number of active software components (tasks or processes), which are processed and scheduled on a single hardware platform, the time to switch between these processes (redirect the program counter, save and restore the registers, swap the cache pages) is high. This can have a negative effect on the temporal correctness. Therefore, the pattern application is possible, in case a safety critical situation occurs, if:

- An architectural element misses a hard deadline

Solution

To reduce the number of small active processes, combine two processes to one new process, which acts as the two processes did before. This will lead to a scheduling, where task switching does not occur as often as before. As an effect, the resulting components will have a better chance to meet their deadlines.



Rationales

- Technical and economic rationales
 - o Reducing the coupling between the processes and increase the cohesion of the processes this will influence the development and maintainability effort
 - o Reduce the encapsulation, because one component has now access to the data of the other component
 - o The scheduling plan and the priorities of the components must be redesigned

2.8 Hardware Platform Substitution

Problem

How can the probability of a hazard be reduced that is caused by an improper hardware platform?

Context

The architecture misses its safety requirement due to a set of software components executed on a hardware platform with a low quality. This low quality may result in reliability and/or in performance problems. In case of reliability problems, the system can cause a hazard due to the following reasons:

- A software component is not available
- A software component generates an incorrect output

In case of performance problems, the system can cause a hazard because:

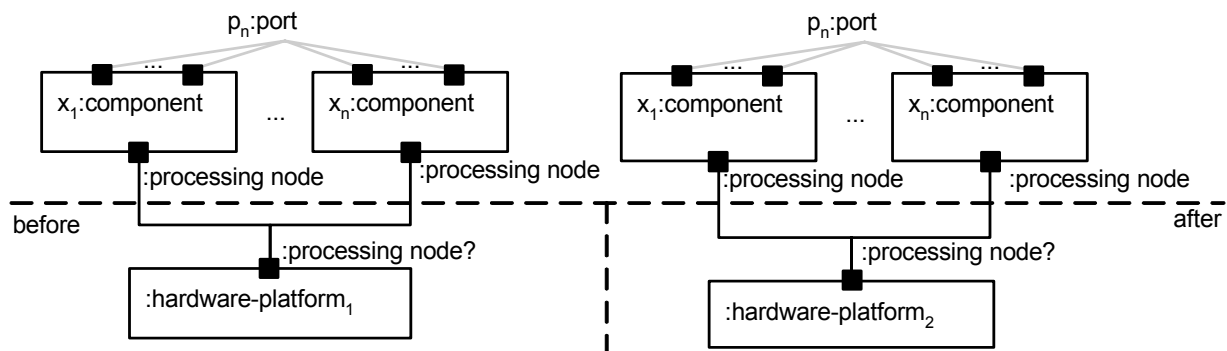
- A software component misses a hard deadline

Forces

- The reliability and availability of the hardware platform influences the reliability and availability of software components, which are executed on this hardware platform
- The performance of the hardware platform influences the performance of software components, which are executed on this hardware platform

Solution

Integrate a newer and better hardware platform in the architecture. All software components that are processed on the old hardware platform are now processed on this new hardware platform. This implies that if the new hardware platform have better reliability properties, the probability of the safety critical failures due to unavailability and incorrectness are reduced. In case of a higher execution speed of the new hardware platform, the components of the resulting software architecture will have a better chance to meet their hard deadlines.



Rationales

- Economic rationales
 - o A hardware platform with a better quality may be more expensive
- Technical rationales
 - o Existing software components which are developed for the old hardware platform are incompatible to the new one
 - o For simple and common hardware platforms, often safety cases with proved information about failure rates exist. These safety cases must be generated for newer hardware platforms.

Related Patterns

Hardware Platform Reassignment (2.9)

2.9 Hardware Platform Reassignment

Problem

How to reduce the probability of a hazard that is caused by a component that is executed on an improper hardware platform

Context

The erroneous behavior depends on the processing of a software element on a specific hardware platform. This can be in one case, if the same hardware platform processes too many processes and it becomes impossible that this platform schedules a software element with respect to its deadline. Thus it can be possible that:

- A software component misses a hard deadline

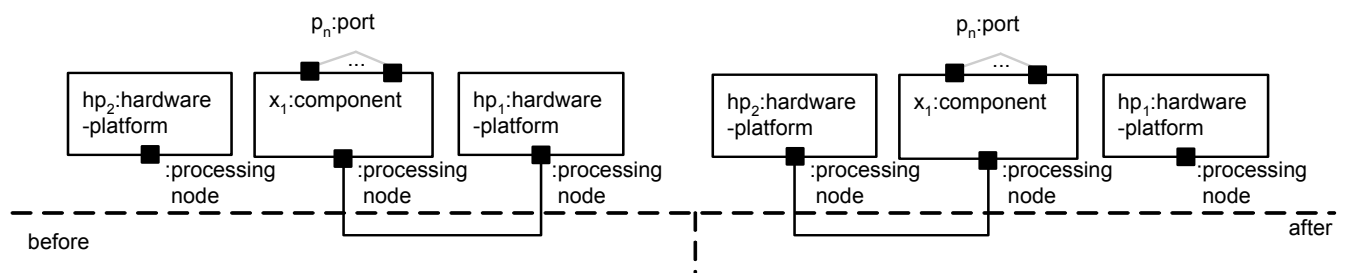
In the other case the processing of a software element on a specific hardware platform can lead to a common cause failure [Mauri 00] [Pumfrey 99], which increases the probability of a hazard. Such common cause failures can simply be a failure of the processing unit or the power supplies of the hardware platform. In this case, the consequences may be:

- A software component is not available
- A software component generates an incorrect output

Note that the problem is not due to the quality of the hardware platform as stated in the Hardware Platform Substitution (2.8) pattern.

Solution

The software element can be assigned to another hardware platform, where the software element can be scheduled without any problem or the probability of common cause failures are reduced.



Rationales

- Technical rationales
 - o The software component and the new hardware must be compatible
 - o The communication overhead must be considered between the reassigned software component and software components on the former hardware platform
 - o The influence of common cause failures will be changed

Related Patterns

Hardware Platform Substitution (2.8)

2.10 Actuation Monitor

Aliases

Monitor-Actuator [Douglass 99]

Problem

Find an appropriate mechanism to detect failures or errors that can lead to known hazards.

Context

This pattern is applicable to refine the *failure/error detection* component of the Protected-Single-Channel (2.3), the Two-Channel-Redundancy (2.6) or the Recovery Block (2.4) pattern to detect a critical incorrect behavior. Therefore, the identification of the set of all critical incorrect outputs should have taken place during the hazard analysis.

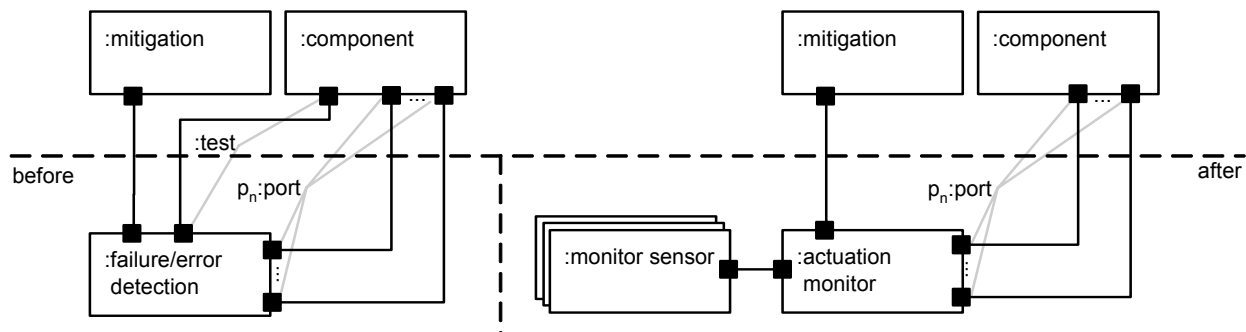
Forces

- The set of relevant hazards is often known for a specific application domain.
- Further, before the system development risk and hazard analysis are accomplished that identified the remaining hazards.

Solution

This pattern introduces an *actuation monitor* component that identifies failures of the component (*actuator*) allowing for the utilization of an appropriate fault-handling mechanism in the *mitigation* component. For this reason, the *actuation monitor* component receives a copy of all messages sent by the *actuator* and proves them for conformance to the safety requirement. For this the *actuation monitor* component needs information about the current state of the environment. Thus, the actuation monitor pattern introduces a set of redundant sensors denoted as *monitor sensors*.

Due to their nature, safety requirements are often specified with formal methods like temporal logic formulas. This allows for the automatic generation of the *actuation monitor* component from this safety specification. For further implementation specific details of the Actuation Monitor we point to [Douglass 99,02] and [Lala, Harper 94].



Rationales

- Economic rationales:
 - o With the *actuation monitor* an additional component must be developed
 - o With the *monitor sensor* an additional hardware is needed

Example

A level crossing can cause a hazard, if a train is in the area of the level crossing and the gates are open. To detect this hazard with the actuation monitor pattern two sensors are needed: one to detect that the train is in the critical section of the level crossing and one to get the current state of the gates. If the signals of these sensors point out that the system is in a hazard state, the mitigation component should stop the train.

Related Patterns

Watchdog (2.12), Integrity Check (2.11)

2.11 Integrity Check

Aliases

Data Integrity Test, Code Integrity Test, Built in Test, Information redundancy

Problem

How can be failure or error identified that are caused by an alteration of the software component.

Context

It is possible to implement this pattern as a refinement to the *failure/error detection* component of the Protected-Single-Channel (2.3), the Two-Channel-Redundancy (2.6) or the Recovery Block (2.4) pattern, to detect a critical incorrect behavior, because:

- An architectural element generates an incorrect output

In this case the unauthorized alteration of the architectural element after the deployment of the system is the reason. As an example, the external influences like electromagnetic impulses or fields that can negate a bit in the memory of the code or date the segment of a component.

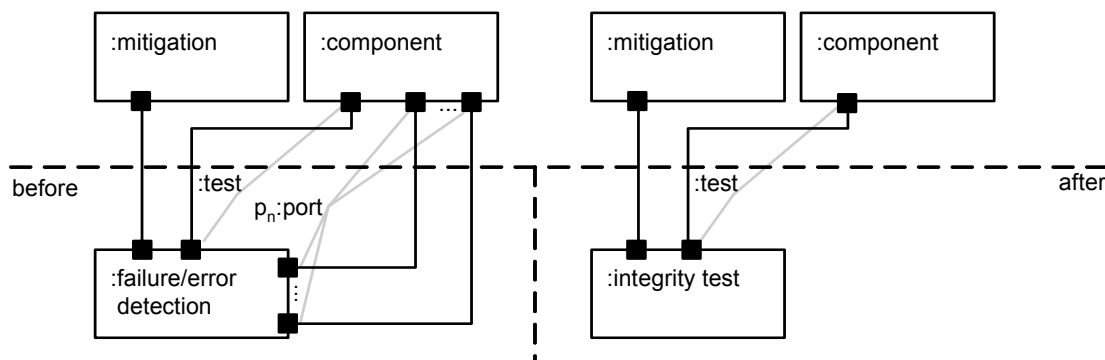
Force

- Software components in many areas of safety critical systems must be permanently available over large time period. Thus, the probability that the component might be altered is very high and it is necessary to detect this failures.
- An alteration of a safety critical software component can lead to an incorrect behavior of the software component that can lead to a hazard of the system.

Solution

Refine the general *failure/error detection* component of the Protected-Single-Channel (2.3), the Two-Channel-Redundancy (2.6) or the Recovery Block (2.4) pattern with a component called *integrity test* that checks the integrity of the safety sensitive component. Add redundant information to the data and code to determine, if something or someone has injected errors. In the implementation, to distinguish between the code and data segment, because the code segment is static and the data segment is changing by an operation of the component. Thus, it is possible to extend the code segment with static redundant information, like proof bits (CRC, checksums or parity). Based on these static bits a detection of an alteration of the code segment of the software component is possible. The integrity check of the data segment is more difficult. Therefore generally two methods exist. The first method is heavyweight and requires from each operation that changes the data segment to change the additional redundant bits. The second method is lightweight and utilizes range checks of the used data and no redundant bits. If an alteration of the code and/or data segment is detected than the mitigation component can use a restart strategy, if the error is not persistent.

For further readings about integrity tests, CRC, checksums [Stone et al 98] is appropriate.



Rationales

- Technical rationales:
 - o The memory usage of the component increases due to redundant information that must be saved
 - o The performance of the components is influenced negatively due to the overhead to store and to check the additional redundant information

Related Patterns

Actuation-Monitor (2.10), Watchdog (2.12)

2.12 Watchdog

Aliases

Watchdog Timer [Mahmood, McCluskey 88], Watchdog Processor, Heartbeat, I am Alive [Saridakis 02], Are You Alive [Saridakis 02]

Problem

Find an appropriate failure or error detection mechanism that identifies timing problems of a software component.

Context

This pattern may serve as a refinement to the *failure/error detection* component of Protected-Single-Channel (2.3), the Two-Channel-Redundancy (2.6) or the Recovery Block (2.4) pattern to detect a critical incorrect behavior because:

- An architectural element misses a hard deadline
- An architectural element is not available

The first failure type can be denoted as the architectural element reacting too late and the second failure type can be denoted as the architectural element reacting infinitely too late.

Forces

- The identification of a timing failure is problematic
- An unavailability of a software component is a timing failure. It means the software component reacts infinite to late
- Furthermore an unavailable component cannot tell that they is unavailable (A dead man cannot tell you that he is dead)

Solution

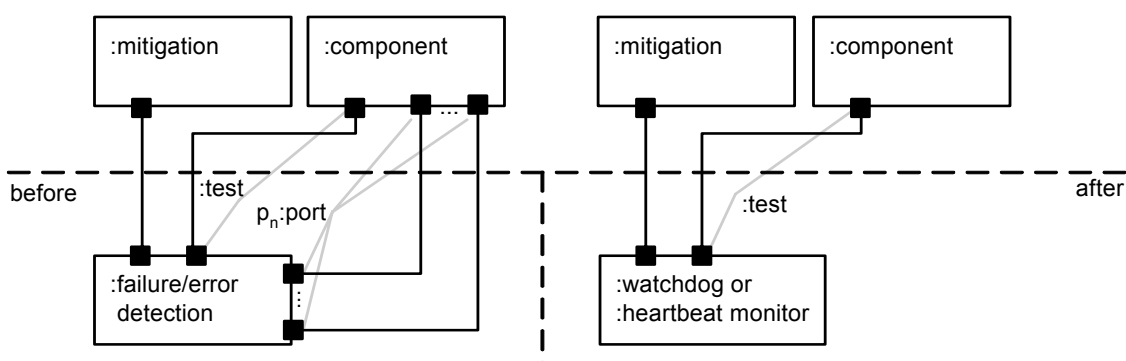
For the realization of this pattern with the watchdog and the heartbeat, two alternative versions exist. We describe these versions in one pattern because they share too many aspects and implementation details.

Watchdog version:

In the watchdog version substitute the *failure/error detection* component with a *watchdog* component. This component implements an independent timer. This timer is preset with an initial value, which is often the deadline of the monitored component. After the start of the timer the component being watched must reset it before it expires; otherwise, the watched component is assumed to have missed a deadline. For the realization of the watchdog timers it is either possible to use hardware or software. For further readings, we recommend [Mahmood, McCluskey 88], [Saridakis 02] and [Pont, Ong 02].

Heartbeat version:

The heartbeat version substitutes the *failure/error detection* component with a *heartbeat monitor* component. This component sends periodically a message (a heartbeat) to a monitored components and waits for a reply. If the monitored component does not respond within a predefined timeout interval with an acknowledgment message, it is declared as not available.



Rationales

- Technical and economic rationales:
 - o An additional hardware is required to realize an independent timer.
 - o The deadline of the component must be known.

- The availability of the process communication between the test ports influences the result of the failure/error detection.

Example

As an example for the application of the watchdog pattern, we chose an airbag control system. The corresponding controller must be available, if a car crash occurs. To ensure that this is the case, it is necessary to reset a watch component after every polling of the crash sensors. If the controller misses to reset the watchdog, the system should inform the car driver for example with an alarm lamp, or, in a rigorous version, reduce the speed of the car.

Related Patterns

Actuation-Monitor (2.10), Integrity Check (2.11)

3. CONCLUSION AND FUTURE WORK

In this paper, we addressed the increasing demand for the rigorous fulfillment of non-functional requirements at the architectural level by introducing a pattern language to especially improve the safety properties. This language serves as a guide for software architects to choose a suitable safety pattern with respect to the problem and the economic and technical context.

Besides, this work is not complete. For future work, we will present a set of failure mitigation patterns for each relevant application field.

REFERENCES

[Avizienis 85]

Avizienis A., The N-Version Approach to Fault-Tolerant Software, IEEE Transactions on Software Engineering, vol. SE-11, no. 12, Dec. 1985, pp.1491-1501.

[Avizienis et al. 01]

Avizienis, J.-C. Laprie, Randell B., Fundamental Concepts of Dependability, Research Report N01145, LAAS-CNRS, April 2001

[Birolini 99]

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

[Buschmann et al. 96]

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

[Douglass 99]

Douglas B. P., Doing Hard Time, Addison Wesley, Reading, Massachusetts, 1999

[Douglass 02]

Douglas B. P., Real Time Design Patterns, Addison Wesley Reading, Massachusetts, 2002

[EN 50126]

CENELEC, Railway applications The specification and demonstration of dependability, reliability, availability, maintainability and safety (RAMS), European Committee for Electrotechnical Standardisation, Brussels, Standard EN 50126-29, November 1995

[Fenelon et al. 94]

Fenelon P., McDermid J.A., Nicholson M., and Pumfrey D. J., Towards Integrated Safety Analysis and Design, in: ACM Applied Computing Review, August 1994

[Fowler 99]

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

[Gamma et al. 95]

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

[Grunske 2003]

Grunske L., Automated Software Architecture Evolution with Hypergraph Transformation, in

- Proceedings of the 7th International IASTED on Conference Software Engineering and Application (SEA 03), Marina del Ray, Nov. 3-5, 2003 to appear
- [Hofmeister et al. 99]
Hofmeister C., Nord R. and Soni D., Applied Software Architecture, Reading, MA: Addison Wesley Longman 1999
- [Huang, Kintala 93]
Huang, Y. Kintala, C. Software Implemented Fault Tolerance: Technologies and Experience. Proceedings of the 23rd Fault-Tolerant Computing Symposium, pp. 2-9. June 1993.
- [Knight, Leveson 86]
Knight, J. C. and Leveson, N. G., An Experimental Evaluation of the Assumption of Independence in Multiversion Programming, IEEE Transactions on Software Engineering, Volume 12, Number 1, January 1986, pp. 96--109.
- [Lala, Harper 94]
Lala J. H. and Harper R. E., Architectural Principles for Safety-Critical Real-Time Applications, Proc. of the IEEE, vol. 82, no. 1, Jan. 1994, pp. 25-40
- [Laprie 92]
Laprie, J.C. (ed.), Dependability: Basic Concepts and Associated Terminology. Vol. 5, Dependable Computing and Fault-Tolerant Systems Series, Vienna: Springer 1992
- [Leveson 95]
Leveson N. G., Safeware: System Safety and Computers. Addison-Wesley, 1995.
- [Liggesmeyer 00]
Liggesmeyer P., Qualitätssicherung softwareintensiver technischer Systeme, Heidelberg: Spektrum-Akademischer-Verlag 2000
- [Mauri 01]
Mauri G., Integrating Safety Analysis Techniques, Supporting Identification of Common Cause Failures, Dissertation, University of York, YCST-2001-02, York 2001
- [Mahmood, McCluskey 88]
Mahmood A., McCluskey E. J., Concurrent error detection using watchdog processors a survey, IEEE Transactions on Computers, vol. 37, February 1988, pp. 160--174
- [McDermid, Pumfrey 95]
McDermid J.A., Pumfrey D.J., A Development of Hazard Analysis to aid Software Design in Proceedings of the Ninth Annual Conference on Computer Assurance COMPASS '94, Gaithersburg, pp. 17-25, IEEE 1995
- [Mitra et al. 99]
Mitra, S., Saxena N.R., and McCluskey E. J., Design Diversity for Redundant Systems, 29th International Symposium on Fault-Tolerant Computing (FTCS-29) Fast Abstracts, pp. 33-34, Madison, WI, June 15-18, 1999
- [Papadopoulos et al. 01]
Papadopoulos Y., McDermid J. A., Sasse R., Heiner G., Analysis and Synthesis of the Behaviour of Complex Programmable Electronic Systems in Conditions of Failure, Reliability Engineering and System Safety, 71(3):229-247, Elsevier Science 2001
- [Pont 01]
Pont M.J., Patterns for time-triggered embedded systems: Building reliable applications with the 8051 family of microcontrollers, ACM Press / Addison-Wesley. 2001
- [Pont, Ong 02]
Pont, M.J. and Ong, H.L.R., Using watchdog timers to improve the reliability of TTCS embedded systems, in Hruby, P. and Soressen, K. E. Proceedings of the First Nordic Conference on Pattern Languages of Programs, September, 2002, pp.159-200
- [Pumfrey 99]
Pumfrey, D.J., The Principled Design of Computer System Safety Analyses, Dissertation, University of York 1999
- [Randell, Xu 95]
Randell B. and Xu J., The Evolution of the Recovery Block Concept, in Software Fault Tolerance, Michael R. Lyu, editor, Wiley, 1995, pp. 1 – 21.
- [Randell 75]
Randell B., System structure for software fault tolerance, IEEE Transactions on Software Engineering, No. 2, 1975, pp. 220-232.
- [Saridakis 02]
Saridakis T., A System of Patterns for Fault Tolerance, in Proceedings of the EuroPlop 2002.
- [Saridakis 03]
Saridakis T., Design Patterns for Fault Containment, Proceedings of the EuroPlop 2003.
- [Stone et al 98]
Stone J., M. Greenwald M., C. Partridge C., and Hughes J., Performance of Checksums and CRCs over Real Data, pp. 529-543, IEEE/ACM Trans. on Networking, Vol. 6, No. 5, October 1998.

Two sets of Patterns about Group Communication and Dynamics

Ofra Homsy
Tel-Aviv, Israel
tngt@netvision.net.il

Introduction

This paper contains two sets of patterns, each comprised of three patterns. Each set refers to a different aspect in working with groups of people.

One set of patterns relates the angle of an outsider to the group, someone who is about to face a group of people for the first time, with a specific agenda: passing on information.

The other set of patterns relates the inside angle of a group. Someone familiar with the personalities comprising the group, the group practices and preferences, who finds it can be hard to act within the group either because of personal traits or because of the group norms.

Thanks

I want to express warm appreciation to my shepherd, Peter Sommerlad, for a wonderful shepherding experience. His insights, his questions, his patience and sense of humor were great aids in the process of cultivating these patterns of mine.

A big Thank-You to the Organizing Committee of the VikingPLoP. Without their support, patience and encouragement I would not be able to present these patterns.

And to my long time friend, Shlomit Baruchi, a group-leader and tour-guide.

The **first set of patterns** relates to a problem many speakers encounter when they prepare a lesson, a lecture or any other type of a presentation: ‘will the target audience learn what I want to teach by the end of my presentation’.

These patterns are intended for new teachers, students, new guides, and other people who are relatively inexperienced in giving lectures or presenting information of any type for a lecture, a course (long or short), a demonstration, or an industrial presentation. They may benefit also people with some experience who wish to improve their preparation techniques.

The first pattern, **Does Mom Understand**, proposes a way to evaluate how successfully we have modulated our presentation for a layman to understand.

The second pattern, **Modular Presentation**, refers to the times we are asked to prepare a presentation but we cannot or do not have enough time to find information about the audience’s level of knowledge.

The third pattern, **Aim Your Presentation**, suggests a way to tailor the presentation to the needs of the intended audience.

These last two patterns combine: One is up-front static modular preparation; the other is the dynamic adjustment to the situation at hand.

Does Mom Understand?

Context

You are preparing a presentation to an audience unfamiliar with a subject matter of your expertise. The presentation can be verbal (a lecture), in writing (a paper), assisted by posters to illustrate your topic or by computer.

Problem

You are well familiar with your subject matter, and so are your colleagues. You worry that your explanations may contain or be based on prior knowledge or assumptions that are obvious to you but not to someone new to the subject. These may fail the understanding or learning of the new subject.

Forces

- Knowledge pertaining to material you work with becomes a part of your thinking and vocabulary.
- Lacking basic information to understand a new presentation can frustrate and hinder understanding in your audience.
- Going over too many basic topics takes up time and tries the audience's patience.

Solution

Show your work to a layman, someone who is not familiar with the subject matter. Check what they understood, and whether that is what you wanted them to understand. Check what did they not understand and you may need to add. Verify that you didn't simplify your explanations too much.

Make corrections to your presentation so you don't need to explain anything beyond the material you present. Include a picture that clarifies a point, omit jargon if it's not necessary, or add an explanation for a jargon word if you are going to use it.

Iterate this process: after you made changes, show your work to a new layman. You may want to repeat this process with a few people (provided you have enough time, and enough good friends) until you are satisfied with your presentation.

It may be useful to try to match the layman with your target audience if possible, to get a close representation of results.

Precautions

Layman is a relative term – obviously there does have to be some common ground. For example if the topic is 'Concepts in Ancient Mandarin', it would be too frustrating for the layman not to have basic knowledge in Mandarin.

Known Uses

* In the pattern community the writing process is aimed at and measured by what is understood from a written pattern. A writers-workshop gives the author a view of what people understood of the pattern, and if this understanding is not what was meant, the author has a chance to make changes to the pattern.

* From High School through University, I would ask my mother to read or listen to papers I was preparing. As our fields of interest are very different, she would invariably protest that she doesn't know the first thing about these subjects, and I always said that was the best test for me: if she got my message, then so too will my audience. If there were issues of prior knowledge, we would put them aside for a while, and still try to see if the message comes through (something like "if I knew what this term means, then I would have no problem understanding this whole part")

* In my work place there's a custom that close colleagues often ask each other to read an email message they composed, to make sure what they want to say is also what can be understood from the reading. We have two methods of doing this: either he first tells me what it is that he wants to explain, and then I read and see if the message matches the purpose, or he asks me to read and then asks what was my understanding of what he is trying to say.

* In Fagan Inspection the role of the 'Reader' is to present the reviewers with his/her understanding of the code or document under review.

Resulting context

If what a layman learns from your presentation is not what you meant they learn, or maybe it is over simplified, you can go and change your material so it is better understood.

Checking your work with a layman, learning what they understood and what was not clear helped you correct and change your presentation.

You may want to create a **Modular Presentation**, so you can be prepared in case your audience knows more then you expected (in which case you'll be able to skip ahead) or have extra information ready in case your audience proves to know less then you anticipated. You may want to better **Aim Your Presentation**, to avoid or at least reduce the need for these measures.

Modular Presentation

Context

You are preparing a presentation to an audience unfamiliar with the subject matter, but you are not sure of the actual learning needs of your audience.

Problem

You didn't have time or opportunity to **Aim Your Presentation**, which can be anything - a lecture, a lesson or any kind of industrial presentation.

Or, you may want to present this topic to multiple audiences that may vary in their level of knowledge and understanding of this field.

You don't want to present too few explanations, because this may encumber understanding or learning of the new subject. On the other hand you don't want to bore your audience with too much information they are already familiar with.

Forces

- Too much or too little basic information to understand a new presentation can frustrate or hinder understanding in your audience.
- There is limitation on time or resources when presenting the new idea.
- Going over basics takes up time.
- While teaching basic terms or information may be imperative to understanding of your presentation, getting into too much detail can make it boring.

Solution

Prepare a modular presentation: Include more material than you plan to actually present. Prepare your presentation so you could either skip ahead without losing the context or expand where the audience requires. Prepare skip-able posters (or slides) so if you see your audience is familiar with an issue you can skip them without leaving pertinent information out. Prepare a reserve of intro slides on the more important subject, so if you notice your audience needs expansion on some issue you can turn to them and add to your presentation. Prepare a picture or a table to clarify an important subject should it turn out hard to understand during your lecture. Bring examples from different domains to help you audience to better relate and understand your point.

Drawbacks

This modus operandi creates extra work in preparation, and may not fit a presenter with a tight timetable.

Known Uses

* Seasoned presenters prepare their material in modules in advance. This way they can change the schedule for the following days if they learn that their audience is either too unskilled or too advanced for their original plan on day one, or advance their lecture if they find upon beginning that their audience is well versed in the topic.

* **Skippable Sections** pattern [1] is an example of writing with the purpose of allowing readers to skip parts without losing pertinent information.

* When guiding a tour, the more experienced tour-guides carry with them drawings and maps of places the group may pass on the way even if they don't plan a stop there. They do this so they will be prepared for two scenarios: either questions from more curious group members wishing to learn more, or for cases of unexpected changes in the plan and a need to visit a replacement site.

Resulting context

Having prepared for different optional scenarios of your presentation, you have had to work longer in preparations, but you gained maneuverability: during your presentation you can expand more if your audience requires, or you can skip ahead if you see your audience is getting impatient.

You also gain the ability to present your material to different audiences or to re-use some of your modules for different presentations.

Modular presentation can allow the author/presenter to establish a common baseline of knowledge with the audience and to avoid too elevated parts without crushing the whole presentation.

You may want to check **Does Mom Understand** your presentation if cut or extended, and correct accordingly.

You may also wish to reduce the unknown factor, or if you don't have enough time to prepare extra material, you may want to **Aim Your Presentation** to your audience needs, as shown in the following pattern.

Aim Your Presentation

Context

You are preparing a presentation to an audience unfamiliar with the subject matter of your expertise, and you are not sure of the actual learning needs of your audience.

Problem

You don't want to present too few explanations, because this may hinder the understanding or learning of the new subject. You also don't want to bore your audience with too much information they are already familiar with.

Forces

- Not having basic information to understand a new presentation can frustrate and hinder understanding in your audience.
- There is limitation on time or resources when presenting the new idea.
- Going over basics takes up time.
- While building basic terms or information may be imperative to understanding of your presentation, getting into too much detail can make it boring.

Solution

Ask for information about your target audience before you start planning your presentation. Ask the agency that invited you to talk, or someone who worked with this audience in the past or in the present. Maybe even get in touch with a representative of the audience whom you can ask your questions (even be your layman as shown in the pattern **Does Mom Understand?**)

Spend some time learning about the group of people you are about to meet, and pay attention. Make sure you understand the import of the information you get. Not all the information can be gleaned by direct questions, some anecdotes sometimes convey significant insight or provide ideas you may wish to incorporate into your lecture. If you didn't have time to do this in advance, you can also ask your audience at the beginning of your presentation. This is a less favorable solution, as it leaves you with very little time for adjustments.

Known Uses

* Applicant forms to courses include questions about prior learning, knowledge and experience, for the very reason of adjusting the course to the level of the candidates. As the material for a course is prepared in advance, it is possible to change it according to information gained from the application forms.

* Peter Sommerlad, when introducing OO in the past in a short talk, often did a short survey about the roles of the people in the audience (developer, manager, other), the knowledge of programming languages (Cobol, C, FORTRAN, Basic, Pascal, etc.) and adjusted his on-the-fly examples.

* I once took a group of Australian tourists to a visit in an Alpaka ranch in Israel. The place had its home guide, and just before she started talking to the group, she asked for information about the group. Unfortunately, it seems she missed one vital piece of information, and though we hoped to learn about the South-American animals, she extended the lecture to some other interesting species the ranch held, focusing essentially on the few Wallabies in their care.

Resulting context

Having spent time learning the background, needs, abilities and expectations of your presentation audience, you can prepare your presentation accordingly. It can build on information already known to the audience, expand on topics less familiar to them, you may even incorporate jokes that relate to their world or use examples from your audience's melee.

You may want to check **Does Mom Understand** your presentation, and, if you have time, you can consider creating a **Modular Presentation** so you can adjust or compensate for cases you have misjudged your audience.

The **second set of patterns** refers to some group dynamics, trying to capture good practices for overcoming obstacles in intra-group communication, either bringing input from less prominent group members or enabling group discussion where it may be a little difficult to create.

Inequality exists in every society. Even in the most egalitarian groups some members gain more prominence than others. [2] And every cohesive group defines its norms and practices, underlining the social interaction and conduct in the group.

These patterns try to give people who work with a group or in a group, such as team leaders, members of groups, managers, moderators, guides, ideas about resolving situations where they may find it hard to introduce new ideas into the group.

The first pattern, **Safe Discussion**, looks at group culture and tries to capture a way for group members to discuss a problem or an issue in dispute in the group.

The next two patterns connect two sides of the same issue: a way for capturing ideas from less prominent group members.

The second pattern, **Spokesperson**, Presents the first half of a concept. It presents reasons or conditions in which it may be useful for one group member will volunteer to speak up for the more timid group member.

The third pattern, **Be Thy Mouth**, presents the other side of this concept. It presents reasons or circumstances in which it may be helpful for a more timid group member to seek out the help of another, more extrovert, or more comfortably situated group member, to bring up an idea before the group.

Spokesperson and **Be Thy Mouth** are complementary patterns that occur together. They are two sides of the same coin, it just depends on who is the active part.

Safe Discussion

Context

In every group some members become more predominant, gain the respect of group members through hard work or special contribution they put into the success of the group, or even because of their personality. There are of course less prominent group members.[2] Every group large or small, every society, defines its boundaries: what is an acceptable behavior, what are acceptable topics, what are the acceptable routes of public discussion.

The social structure, social connections between group members, the social interaction within a group may sometimes hinder members from bringing up problems, suggestions or complaints, either because they feel inadequate or the issue may be associated with prominent members of the group.

Problem

You become aware of a delicate issue to discuss within your group. The topic or the fact of presenting it may not align with the group established practices and norms, but you feel this needs to be addressed as soon as possible, as this is an important issue or an important suggestion that may impact the group. However, bringing it directly to formal discussion may be too loaded. It may step on toes or intrude on “internal politics” and sensitivities.

Forces

- People who never knew of the subject may not care to discuss it at all or have no information to allow them to create an opinion, leaving out valuable contribution.
- The longer an issue is familiar the more comfortable people become talking about it.
- Issues not discussed by the whole group miss out involving the whole group in their consequences.
- Some people may find it difficult to address their criticism, objections, or arguments directly to other people.
- There are issues and problems that if not discussed on time, may aggravate over time.

Solution

Introduce the issue within different settings using a non-formal discussion arena, in a non-threatening, public way:

- Call in an outside moderator to lead the discussion.
- Use recognized unofficial channels of communication (internal newsletter).
- Locate the discussion in a neutral location with informal settings.

This way you bring the issue into awareness, challenging but not directly confronting neither members nor norms of the group. You aim to create unofficial talks, let people know of the issue, discuss it, create an opinion about it.

Eventually you can bring the issue up in a formal discussion. For example call a meeting, put a proposal of legislation, and say ‘We have an issue to discuss’.

The primary goal is getting the issue to public discussion, which was your original intent.

Precautions

This pattern may be cultural dependent, as it requires a measure of agreement on free discussion. Not all groups or organizations are built this way. [3]

Known Uses

* Commercial Advertisement that may not be fit for a wide campaign, such as on TV, may still be released through web commercial or on Email that gets sent and re-sent by individuals. Thus it may become a center of discussion in the community even if it doesn't completely adhere to social conventions.

* Keeping with the tradition of PLoP conferences, EuroPLoP supports an open, egalitarian atmosphere. Still, since it is a community, and involves many people, sometimes from different cultures, participants may be reluctant about raising an issue for discussion. This may be either because they feel new to the community, or not as experienced as some of the other participants etc. At EuroPLoP an issue can first be raised through the internal newsletter (Kloster Hearsay), where the settings are less formal and sometimes humorous. If more participants are interested the issue it can then be followed by a 'Birds of a Feather' (BoF) meeting, or be raised as an issue in the community at large such as mailing lists, EuropHillside meeting.

* An advise given by many institutions teaching management skills, is to listen to discussions going on in the coffee corner of the office. This informal meeting place allows for free conversation, revealing people's true views on issues. [4]

Resulting context

When bringing an issue to a non-direct discussion, it may still cause some discomfort, either to the group or to individuals within the group, but it will not be directed at known individual(s).

This way gives all group members time to get familiar with the topic, and develop opinions they can bring up when a direct, formal discussion is finally open.

This may also help in preventing a problem from aggravating, or enhance your overall group's success if you managed to air an internal issue that everyone were reluctant to open for fear of 'stirring up the muddy water'.

Spokesperson

“And he shall be thy spokesman unto the people; and it shall come to pass, that he shall be to thee a mouth, and thou shalt be to him in God's stead.”

Exodus 4:16

Context

Even cohesive groups are comprised of people with different personalities and tendencies. Though familiarity tends to improve and facilitate the work and the social relations in a unified group, not everyone in a group feels comfortable facing the entire group.

Problem

Through friendship, or by chance, you hear a more timid member of the group expressing a concern, an idea, an objection or an opinion you think has value to the group. You feel this can be an important contribution to the group and you don't want this insight to be lost.

However it may happen that this person is unable to speak up for whatever personal reason, or it may be important not to identify the initiator of a discussion issue.

Forces

- Not everyone in a group feels able to talk up, either because of personality trait or because of cultural norms.
- Social standings in a group may influence an individual's decision about expressing themselves, addressing the group.
- Sometimes the opinion developed during discussion is influenced by the source or the way that the issue is brought up.
- Stepping in to represent someone else's ideas may be interpreted as patronizing.
- In every group there are people who feel comfortable speaking and expressing ideas, even if they encounter disagreement.

Solution

Be the one to introduce the idea or to voice the opinion, instead of the more timid person. Since you feel more confident speaking to the group (maybe because you have more experience, or you feel secure in your social standings in the group, or you are more of an extrovert than the person who has an issue to broach).

Discuss the reason they avoid speaking, then suggest, privately and respectfully, to that person to speak for him/her, bring their objection before the group.

You can suggest that when you'll address the group you will say you are talking on behalf of someone, or if the issue is too delicate, or that person is too shy, you can decide together to let things stand, just lending your sponsorship, voicing the opinion or idea.

Precautions

This needs to be conducted with care.

This intervention can be interpreted as patronizing if not handled with empathy, good faith, respect, compassion and discretion. It cannot be done without permission of the owner of the idea.

Also, being the one speaking up, you may be considered the owner of the idea, and if there are bad feelings related you might get the backlash as they might be directed at the spokesperson.

You should also get well familiarized with the topic and nuances of the opinion you will be representing, so you can discuss it well when you bring it up.

Known Uses

* Peter Sommerlad, a member of the pattern community, has seen pairs of people work great, with one extrovert and a more introvert person, which clearly performed very well together, because the environment accepted the different roles taken.

* Shlomit Baruchi, a group-leader and tour-guide, often looks out for someone in the group saying something quietly or expressing an idea only in a side discussion. If she finds this may be a valuable contribution to the group, she will talk to this person privately, trying to encourage them to speak up. And if they are too timid, she suggests to them to be the one to voice their opinion or idea.

Resulting context

Having more experience, or better social standings in the group, you get less scared when introducing an idea or suggestion, even when it creates opposition.

By giving voice to an idea or an opinion of a more timid group member, you get your group to consider it more objectively or more willingly than if a less prominent member brought it up.

This way the group may gain from an input that might have been lost to it.

You may also help the more timid person to gain confidence, as they see their idea gets merit, being discussed within the group. Even if the suggestion is rejected at the end of the discussion, they can see nothing bad has happened.

Be Thy Mouth

“And he shall be thy spokesman unto the people; and it shall come to pass, that he shall be to thee a mouth, and thou shalt be to him in God's stead.”

Exodus 4:16

Context

In a group there are always people who feel more comfortable facing the entire group, presenting ideas, suggestions or objections in discussions. Either because of a personality trait or because they are new to the group, some people feel less able to speak out.

On another venue, group dynamics tend to create roles, which certain group members fill, (such as the initiator, the critic, the mediator). Over time, group member may tend to find it hard to go out of their role, even if they have some contribution to make.

Problem

You have an idea that you think can contribute to your group. Or, you want to express an opinion in a debate going on in the group. But either because you are shy, or new to the group, or knowing your group practices, you think if the idea came from you it will not get proper attention or would not be considered, you feel inadequate to speak up. You feel very strongly that your input can have an impact on the group, but you feel unable to speak up.

Forces

- Group members may tend to judge the merit of opinions or suggestions according to the person bringing them up.
- Some people are naturally shy, or being new to a group may be reluctant to speak up, or because of character trait or position feel less able to speak up.
- Group members may feel ‘stuck’ in a role, being unable to suggest something new because “it’s somebody else’s role to bring up ideas for the group”.
- In every group there are people who feel comfortable speaking and expressing ideas, even if they encounter disagreement.

Solution

Find a member of the group whom you trust and feel comfortable with, and that you think will receive better attention from group members. **Tell that person about your idea and ask her/him to speak for you** – to be the one to present the idea or opinion to the group in your stead.

Explain your reason for asking: if it is because of a personal difficulty or because of other consideration. The reason may influence your chosen spokesperson’s decision. Be prepared for a refusal, listen to the opinion and consider whether you can speak out yourself, or you may want to try to find another spokesperson.

Precautions

You need to be careful not to over-use the good will of your ‘sponsor’.

Also, don’t become Sirano de Berzherak - even if you find it too hard to speak in front of the group, make sure you do get the credit for your ideas.

Make sure to go over the topic with the spokesperson until s/he is fully versed in the information and in the nuances of your opinion on the issue or your idea. S/he needs to be able to represent you well when s/he brings it up for discussion.

Known Uses

* Lobbying: a group trying to oppose legislation or invoke it tries to find a known figure that will support and represent it’s cause. They will try to choose a person that has authority, good reputation and ability to represent their idea to the best effect. The group will approach this person, explain their idea and ask that person to speak for them.

* As an introvert who learned to extrovert, in the past, I used this method in groups I was member of, and felt too shy to speak out. I would ask for the sponsorship of another group member, have them introduce my idea to the group. (This indeed played part in my learning to be more extroverted, as I saw my ideas and suggestions being carried out).

Resulting context

Having recruited a ‘sponsor’ to voice your idea for you, you may see an idea of yours that may have been ignored, given better consideration by the group.

This may even help the more introverted people develop confidence as they see their idea given merit by the group members, if only to discuss it, even if they don’t accept it. Thus maybe the next time they will be less shy about speaking out their suggestions. (More ideas for moving from being an introvert person to becoming a more extroverted person you can see in Joe Bergin’s pattern: **Introvert-Extrovert**, [5]).

1 A Pattern Language for Pattern Writing, Swati Gupta, 3rd April 2003

<http://www.cs.pitt.edu/~chang/231/8pattern/LANGUAGE/>

2 Cultures and Organizations, Software of the mind. Geert Hofsted, McGraw-Hill Companies, 1991, p23

3 Meetings from a social psychological view: disturbance recovery, Harko Verhagen, the FEEL project, Royal Institute of Technology, Stockholm, Sweden. <http://www.dsv.su.se/feel/DSV/feeldeliverable1.pdf>

4 An example for such advice can be found at <http://www.gotobiz.co.uk/progress/reports.htm>

5 Extrovert-Introvert, Josef Bergin, Proceedings of EuroPLoP 2002, p 323

Analysis Patterns Specifications: Filling the Gaps

Marta Pantoquilha¹, Ricardo Raminhos², João Araujo³

Departamento de Informática
Faculdade de Ciências e Tecnologia
Universidade Nova de Lisboa
Quinta da Torre, 2829-516 Caparica
Portugal

E-mail: ¹ mbp@netvisao.pt, ² rfr@netvisao.pt, ³ ja@di.fct.unl.pt

Abstract. Patterns present solutions for recurrent problems in software engineering. They are applicable at different stages of the software development process. This paper focuses on patterns at requirements and analysis level. Although the term “requirements patterns” has appeared in the requirements engineering community, the name “analysis patterns” is more established in the patterns community. Here we briefly discuss the existing approaches and identify their limitations. The primary goal of this paper is to propose a new template to fill some gaps concerning the specification of analysis patterns.

1 Introduction

The traditional software development lifecycle includes the following phases: requirements elicitation, analysis, design, implementation and test. Each phase creates a more detailed image of the system than the previous one. Nevertheless, to be effective, software development must consider reuse from early stages. Patterns are considered a successful technique to help reusing previous specifications and solutions.

Software patterns are classified depending on various factors including their application in the software development phases (see Figure 1). The most common patterns are analysis and design patterns. Anti-patterns are a kind of pattern that embraces all the development phases (including the test phase), as well as the project management area.

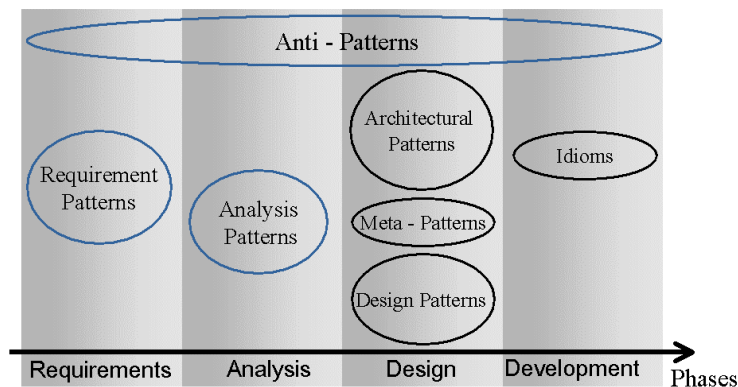


Figure 1 - Project lifeline with the corresponding patterns in each phase.

The term requirements patterns appeared in the requirements engineering community [Robertson, 1996] [Konrad and Cheng, 2002], but it is not widely used. Requirements patterns document user needs and specify generic system behaviour at a high level of abstraction. Requirements patterns also describe generic actions that developers can take to improve non-functional requirements, such as performance, security, reliability, maintainability, and accuracy. These actions are related to client-system interaction or operator-system relations.

The purpose of analysis patterns is to build an analysis model, which will focus on business processes instead of software implementation. The main concerns of these patterns are the conceptual models and the flexibility and reuse of the resulting systems. The conceptual models are represented by a static structure, entity relations (e.g. objects or functions), and data transformations.

The primary goal of this paper is to propose a new template to specify analysis patterns. We also discuss and compare some of the existing approaches and clarify the difference between analysis and requirements patterns.

This paper is organised as follows. Section 2 describes and compares some previous work on requirements and analysis patterns, and identifies their problems. Section 3 proposes a template to describe analysis patterns. Section 4 illustrates the template with an example. Finally, Section 5 draws some conclusions and discusses some future work.

2 Problems with requirements and analysis patterns approaches

In this section we describe the state-of-the-art of requirements and analysis patterns approaches. Afterwards, we establish a comparison between these approaches and conclude with the identification of the most significant problems that arise from the application of those approaches to pattern specification.

2.1 Requirements patterns

In [Whitenack, 1994], a pattern language is described for requirements elicitation. The elicitation process is performed using twenty “easy patterns” that should be applied sequentially (e.g. Customer Expectations, Problem Domain Analysis, Prototypes, Requirements Validation). Through this pattern language the author expects “to guide analysts and product developers to most appropriately apply a set of techniques and methods so as to produce a more thorough analysis and understanding of the problem area” and also “to provide a framework upon which to define and capture requirements”. The 20-pattern appliance is not mandatory, and the user is responsible for choosing the pattern combination that suits him better. Associated to each pattern a set of “Direct Deliverables” is also suggested, providing structure to the modelled information at all stages.

S. Robertson [Robertson, 1996] uses an event/use case approach and employs a very simple template for pattern description with only 4 fields: name, context, solution, and related patterns. Robertson suggests that events and use cases should be used to divide the system into small chunks. These chunks can then be structured into a pattern. Patterns are, therefore, catalogued, based on the name of the use case to which they refer. In her paper, Robertson shows how a particular problem can be abstracted at different levels in order to become a pattern used in different problems.

S. Konrad and B. Cheng [Konrad and Cheng, 2002] focused on requirements patterns for embedded systems. They use a UML approach (class, use case and sequence diagrams) for the pattern definition. Also, they explain the pattern context using problem frames [Jackson, 2000]. A very extensive and detailed template is used to describe patterns (13 fields), based upon that suggested for design patterns [Gamma et al, 1995].

We notice that the term requirements patterns does not differentiate from analysis patterns described as follows.

2.2 Analysis Patterns

M. Fowler [Fowler, 1997], initially proposed the concept of analysis patterns for the representation of conceptual models for commercial processes (accountability, commercial trades and organizational relations). Refinement patterns (design, architectural, etc) are never suggested, and the solution is mostly conceptual. The author presents each pattern through an informal / technical discussion without any kind of structured template.

E. B. Fernandez and X. Yuan present the Semantic Analysis Pattern (SAP) approach [Fernandez and Yuan, 2000]. SAP is “a pattern that describes a small set of coherent use cases that together describe a basic generic application”. The selection of use cases is realised carefully to maximise reusability.

The work by A. Geyer-Schulz and M. Hahsler [Geyer-Schulz and Hahsler, 2001] introduces some structure to analysis patterns. They focus on the cooperative work domain and collaboration between applications.

Analysis patterns proposals include patterns for oil refineries [Zhen and Shao, 2002], the order and shipment of a product [Fernandez et al., 2000], the repair of an entity [Fernandez and Yuan, 2001], negotiation [Hamza and Fayad, 2003] and course management [Yuan and Fernandez, 2003]. Also, in [Hamza and Fayad, 2002] a pattern language is proposed to achieve stability while specifying analysis patterns.

2.3 Comparison between Requirements and Analysis patterns

Here we present a short comparison between requirements and analysis patterns, depicted in table 1. In this comparison, we point out the main characteristics of the approaches of both kinds of patterns and also what we consider to be their limitations.

Table 1 - Comparison between Requirements and Analysis patterns.

| | Characteristics | Limitations |
|------------------------------|--|---|
| Requirements Patterns | <ul style="list-style-type: none"> • They capture in detail functional and non-functional requirements. • They can be extended by design or architectural patterns. • They allow a smooth transition to the implementation phase, due to the pattern detailed description. • They are a more directed form to the programmer understanding. | <ul style="list-style-type: none"> • Little research in this field. • High commitment to the solution domain, due to decisions expressed in the pattern. • The existence of a variety of templates for the different approaches. • The existing approaches do not seem to justify the term requirements patterns as they use similar principles as analysis patterns. |
| Analysis Patterns | <ul style="list-style-type: none"> • They are suitable for the description of conceptual problems. • They present low commitment to the solution domain allowing a high level of freedom for implementation due to their sparse specification details. • Due to the high abstraction level, there is a huge gap between the patterns specification and implementation. • They provide a more directed form to the architect understanding. • In order to migrate to the implementation level, an extra iteration is needed. This extra step could be the transformation of an analysis pattern into a requirement pattern. We would be passing from a low level to a high-level implementation detail. • Lots of work in this field. | <ul style="list-style-type: none"> • The presentation form (degenerate template) used by M. Fowler [Fowler, 1997], is low in specification detail and information about the pattern description. • The existence of a variety of templates for the different approaches. |

This comparison is important to highlight the fact that what defines requirements patterns is not significantly different from analysis patterns, from the approaches studied. This is a result of their proximity, i.e., they are closely related and share a similar level of abstraction. To avoid confusion with these terms, we suggest that patterns at this level of abstraction be called only analysis patterns, by the requirements engineering and patterns community.

2.4 Requirements and Analysis Patterns: Identifying the Problems

In general we believe that the current approaches to specify requirements and analysis patterns are not sufficient to be used by requirements engineers, especially when they start using patterns, because they lack detailed information. Including more details in the descriptions of patterns would facilitate the requirements engineer's work, as this would help them to take the right decisions to use the patterns, efficiently and successfully. This information should include, for example, functional and non-functional requirements, some possible static and dynamic models, and even related anti-patterns.

Additionally, from table 1, we realise that not having a consensus on how these patterns should be specified prevents them from being accepted widely. Hence, standardization helps the use of patterns in the requirements engineering community.

As a solution to these problems, we propose a template with elements that are common to these approaches and new elements to fill some gaps that we consider are missing. In the next section we will present such template. Note that this template is only for analysis patterns, since it still is not clear what requirements really are as their objectives. However, the template also comprises requirements patterns' aspects.

3 A Template for Analysis Patterns

In this section we present a template to specify analysis patterns. Table 2 shows the attributes of the template and their respective descriptions. The attributes which were not a part of any previous template are ticked in the final column. The proposed template is based upon the one described in the POSA approach [Buschmann et al., 96] [Schmidt et al., 2000].

Table 2 - Template for analysis patterns.

| Attributes | | | Short Description | New |
|-------------------------|----------------------|---------------------|---|-----|
| Name* | | | Pattern identifier. | |
| Also Known as | | | Additional names that can also identify this pattern. | |
| Evolution* | | | Chronological register of all previous versions of this pattern. The following notation should be used: {Date, Author, Reason, Changes}. To be used by developers who have already used the pattern to check its changes. | ✓ |
| Structural Adjustments* | | | Presentation of field extensions and omissions to the pattern template. | ✓ |
| Problem* | | | A short description of the problem that this pattern solves. | |
| Motivation* | | | Description of a problematic situation intended to motivate the use of the pattern. | |
| Context* | | | Precise description of the environment in which the problem and solution recur and for which the solution is desirable. | |
| Applicability* | | | Description of the conditions wherein the pattern can be applied. | |
| Requirements* | Functional* | | List of all functional requirements organised through use cases. | ✓ |
| | Non-Functional* | | List of all non-functional requirements. | ✓ |
| | Dependencies* | | Identification of dependencies for requirements. This could be represented through a graph. | ✓ |
| | Priorities* | | Definition of priorities among the requirements. This could be represented by a hierarchical structure. | ✓ |
| | Conflict Resolution* | | Explanation for requirements interaction and conflict resolution. | ✓ |
| | Participants* | | Identification and description of the actors that interact with the system. | ✓ |
| Modelling* | Structure* | Class Diagram* | Structure of the elements of the pattern. | ✓ |
| | | Object description* | Objects description and their responsibilities. | ✓ |

| | | | | |
|---------------------|------------|------------------------------------|---|---|
| | Behaviour* | Collaboration or Sequence Diagrams | Suitable for scenarios description. | ✓ |
| | | Activity Diagrams | Suitable for scenarios and overall description. | ✓ |
| | | State Diagrams | Suitable for scenarios and overall description. | ✓ |
| | Variants | | Description of alternative solutions. | |
| Resulting Context* | | | System configuration after the pattern application. | |
| Consequences* | | | Advantages and disadvantages of the pattern application | |
| Anti-Patterns Traps | | | Most common pitfalls that can occur from the pattern application | ✓ |
| Examples* | | | One or more application examples that illustrate: initial context, how the pattern was applied and all transformations necessary to the initial context so that it could be applied | |
| Related Patterns | | | List of associated patterns (describing connected problems and solutions) | |
| Refinement Patterns | | | Design or architectural patterns that can be used for further refinement | ✓ |
| Implementation | | | Advice on how the solution should be implemented (without specific details e.g. code) | |
| Known Uses* | | | Describes known pattern occurrences and applications in existing systems. This should include at least three different systems | |

* - Required field

Below we discuss the newly introduced attributes.

- **Evolution:** This explains all the transformations the pattern suffered. With the addition of the evolution field we can track the pattern's progress: from current state to the original version. This helps developers that have already used the pattern identifying what changes have taken place. This makes it easier for the developers to adapt to the new version of the pattern. Additionally, if they want to propose modifications to the pattern, they should know what has been done before. This can help validate the new modifications. In the construction of the original pattern this field should only contain information about the Date and Author. In Figure 2 we illustrate the chronological evolution for an abstract pattern.

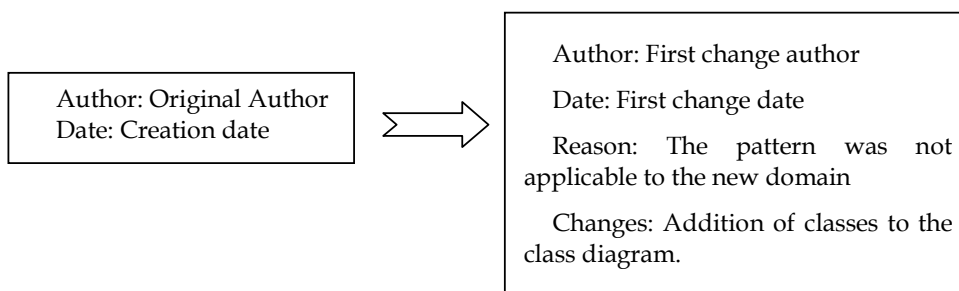


Figure 2 - Evolution tracking system.

- **Structural Adjustments:** This field explains the structure adopted to describe the pattern. It should include all additional extensions, all omitted fields, and the reasons for those decisions. With this information the reader can easily understand the used structure. We suggest the use of a layout similar to the one presented in Table 3.

Table 3: Suggested structural adjustments layout.

| Attribute | Extension | Omission | Reason |
|----------------|-----------|----------|--------------------------|
| Implementation | | ✓ | Reason for the omission |
| New attribute | ✓ | | Reason for the extension |

- **Requirements:** This field (divided into six sub-fields) contains a description of all requirements that must be addressed to solve the problem (how they interact and are balanced). A highly detailed problem specification is gained through the addition of this attribute. With the proposed division, it becomes simpler to understand of the requirements involved, their type (functional, non functional), their dependencies and priorities and how they are solved. We can also extract a use case diagram from the requirements, which shows the services used by the actors (participants).

Considering non-functional requirements early in the analysis stage has a significant impact in problem understanding and its modelling. Naturally, this is also valid when describing analysis patterns. Non-functional requirements provide a more complete description for the analysis pattern and also influence its structure and behaviour. Choosing one model instead of another can have, as a decision factor, the fulfilment of one or more non-functional requirements. Moreover, the prioritization and establishment of dependencies among functional and non-functional requirements will allow grasping the meaning of the problem with more detail and accuracy. Therefore, the inclusion of non-functional requirements in an analysis pattern will help the pattern user to identify and evaluate the applicability of the pattern to his/her specific problem. Some approaches can be used if a higher detail on non-functional requirements relations is required or when complementing our approach. One such example is the approach discussed by Araujo and Weiss [Araujo and Weiss, 2002]. They use the NFR framework [Chung et al, 2000] for describing a design context and also a set of related patterns. The outcome is that several design issues related to system architecture may be addressed by the integration of various patterns. Taking all this into consideration, it is our understanding that the presence of non-functional requirements in the pattern specification can benefit the problem resolution, and consequently the pattern applicability.

For the *Priorities* field in the template we suggest the use of a hierarchical structure as depicted in Figure 3.

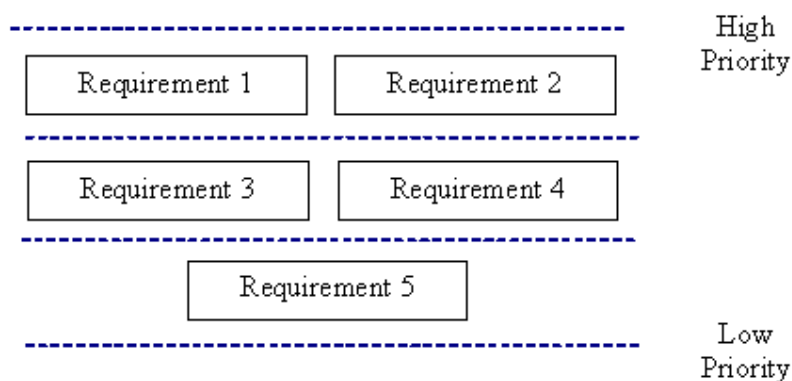


Figure 3: Suggested hierarchical diagram.

For the *Dependencies* field we suggest the use of a dependencies graph. One example is presented in Figure 4, where it is stated that Requirement 1 depends on Requirement 2 and Requirement 3, and so on.

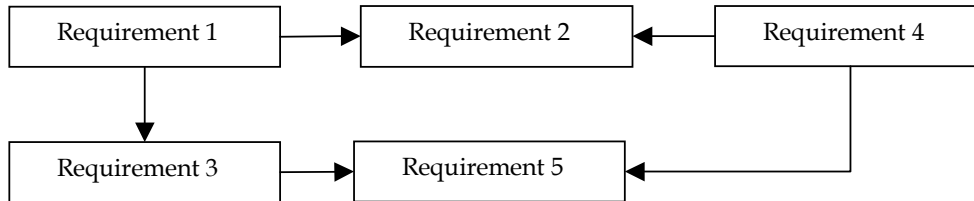


Figure 4: Suggested dependency graph.

In prior template versions, the requirements were addressed in the field *forces*. This was an unstructured field that contained a mixture of functional and non-functional requirements. With our approach, we add structure to the requirements identification and documentation. We also add information about their relationship - dependencies and priorities. To ease the pattern understanding we propose the use of illustrative diagrams: a hierarchical diagram for the requirements priorities specification and a dependency graph for the dependencies establishment. With the inclusion of the participants attribute, we can describe the actors that will manipulate the system. These actors are the ones that will interact with the use cases identified in the *Functional* field.

- **Modelling:** In this section are presented several models that illustrate the solution. This solution is divided in two main groups: behaviour and structure.
 - **Structure:** This group represents the solution's static structural aspects using a UML class diagram. More detailed specification can be obtained in the attribute *Object Description*. This field describes all objects that are present in the class diagram. Note that the class diagram can be represented in different levels of abstraction.
 - **Behaviour:** Offers an illustrative set of scenarios, and also describes the overall pattern behaviour. The pattern should contain at least one scenario example, in an abstract level, and one overall description. Two distinct levels are focused: scenarios examples that show only part of the system, and the overall system behaviour, which illustrates the system's functioning as a whole. There is a great freedom on the diagrams choice that illustrates this field. At least an activity diagram showing the overall system behaviour should be included. In other templates this section only contained examples of part of the system behaviour. Although important, this is not sufficient, because the user does not have a global vision of the system functionality.

The modelling description is presented in previous pattern templates under the name *Solution*. Although some fields are commonly used in both *Modelling* and *Solution*, the modelling approach offers a more detailed, structured and visual (with the addition of several diagrams) understanding.

- **Anti-patterns traps:** With this field we try to avoid common errors in this pattern application by presenting the most common negative results. This field should contain a list of anti-patterns names and a short description of each. To recover from a negative solution the user, using the anti-pattern name as lookup key, should refer to William Brown in [Brown, 1998].
- **Refinement patterns:** This attribute is used to suggest or identify suitable patterns (e.g. design or architectural) that can be applied to the implementation of this pattern.

4 Example

To illustrate the concepts described in the previous section, we present an example that uses the template to specify the analysis pattern *Party* from [Fowler, 1997] (see Table 4).

Table 4: Applying the approach to *Party*.

| Attributes | | Short Description |
|------------------------|------------|--|
| Name | | Party |
| Evolution | | <p>{ date: 1997, author: Martin Fowler }</p> <p>{date: 2003, authors: M. Pantoquilho, R. Raminhos and J. Araújo, reasons: to add more information Changes: Adaptation of the degenerative form to a structured template.}</p> |
| Structural Adjustments | | None. |
| Problem | | To model an address book that contains information about people and companies. |
| Motivation | | “Take a look through your address book, and what do you see? You will see a lot of addresses, telephone numbers, the odd email address ... all linked to something. Often that something is a person, however the odd company shows up.” [Fowler, 1997] |
| Context | | Persons and organizations are present in almost every system that deals with people. The address book is just an example. Persons and Organizations share a common behaviour: they access a common set of objects (telephone numbers, email addresses ...) and operations over them. |
| Applicability | | When you have people and organizations in your model and you see common behaviour. |
| Requirements | Functional | <p>R1: Each Person or Organization has none or more telephone numbers. R2: Each Person or Organization has none or more addresses. R3: Each Person or Organization has none or more e-mail addresses. R4: Person and Organization share some descriptive data. R5: Each Person or Organization may obtain, add or modify information about other persons or organizations.</p> <p>Use case model:</p> <pre> graph TD Person --> Party Organization --> Party Party --> AddContact([Add Contact]) Party --> UpdateContact([Update Contact]) Party --> QueryE-Mail([Query E-Mail Address]) Party --> QueryTelNum([Query Telephone Number]) Party --> QueryAddress([Query Address]) </pre> |

| | | | |
|-----------|---------------------|---|--|
| | Non-Functional | <p>R6: Person / Organization must be authorized (security).</p> <p>R7: Information must be obtained in a short period of time (performance).</p> <p>R8: Information must be correct (accuracy).</p> | |
| | Dependencies | | |
| | Priorities | | |
| | Conflict Resolution | <p>The non functional requirements of performance (R7) and security (R6) may conflict. However this conflict is solved by the assignment of priorities to the different requirements.</p> | |
| | Participants | <p>Person and Organization</p> | |
| Modelling | Structure | Class Diagram | |
| | | Object Description | <p>Person: Defines a person.</p> <p>Organization: Defines an organization.</p> <p>Party: Super type defining a Person or Organization. Contains all functionalities common to both of them.</p> <p>Telephone Number: Describes a telephone number.</p> <p>Address: Defines an address.</p> <p>E-mail Address: Defines an e-mail address.</p> <p>The <i>Person</i> and <i>Organization</i> classes contain the specific data to each entity that they describe. The common attributes are held at the <i>Party</i> class.</p> <p>The <i>Party</i> is defined through the relations with the classes <i>Telephone number</i>, <i>Address</i>, and <i>E-mail Address</i>.</p> |

| | | | |
|--|-------------------|------------------------------------|--|
| | | | Sequence diagram for the use case Query Address |
| | Behaviour | Collaboration or Sequence Diagrams | <pre>sequenceDiagram actor Person as : Person participant Address Person->>Address: Get Address Address-->>Person: (Address)</pre> |
| | | Activity Diagrams | <pre>graph TD Start(()) --> Authorize([Authorize user]) Authorize --> Decision{ } Decision --> Add([Add Information]) Decision --> Get([Get Information]) Add --> Confirm([Confirm]) Get --> Return([Return Information]) Confirm --> End((())) Return --> End</pre> |
| | Variants | | <p>Another possible solution is presented in [Fowler, 1997] (below). Although this variant also solves the problem, it does so by adding a lot of redundancy (inheritance is eliminated and associations are doubled). Therefore, the solution presented in "Class Diagram" is a better one, for the reasons described in the "Consequences" field.</p> <pre>classDiagram class Person class Organization class TelephoneNumber[Telephone Number] class Address class EmailAddress[E-mail Address] Person "0..1" -- "0..n" TelephoneNumber Person "0..1" -- "0..n" Address Person "0..1" -- "0..n" EmailAddress Organization "0..1" -- "0..n" Address Organization "0..1" -- "0..n" EmailAddress</pre> |
| | Resultant Context | | <p>“Party is defined as the super type of person and organization. This allows me to have addresses and phone numbers for departments within companies, or even informal teams.” [Fowler, 1997].</p> |
| | Consequences | | <p>Advantages:</p> <ol style="list-style-type: none">1. Elimination of data and code duplication. |

| | |
|---------------------|--|
| Anti-Patterns Traps | Golden Hammer – Persons and Organizations can be modelled with other patterns than <i>Party</i> . Refer to <i>Related Patterns</i> , for other pattern information. The Blob - <i>Party</i> must only contain the common attributes to Person and Organization. You should resist the temptation to incorporate all data and operations in <i>Party</i> . |
| Examples | “In the UK National Health Service, the following would be parties: Dr. Tom Cairns, the renal unit at St. Mary’s Hospital, St. Mary’s Hospital, Parkside District Health Authority, and the Royal College of Physicians.” [Fowler, 1997]. |
| Related Patterns | <i>Role Object</i> |
| Refinement Patterns | Unknown. |
| Implementation | Use an object-oriented language to the implement this pattern. Although this is a very simple system, you should resist the temptation to implement it in a single class. (avoid the Blob anti-pattern) |
| Known Uses | Not specified. |

To illustrate the behaviour specification of this pattern, we show sequence and activity diagrams. The sequence diagram for *QueryAddress* is described in an abstract form, without including interface and control objects, as these are normally added in the design stage. The activity diagram is used here to specify the global pattern behaviour.

The original *Party* template uses the “Degenerative Form”, which is a very unstructured template form. Other templates for requirements and analysis patterns have a better structure and a high level of detail. In [Fernandez et al., 2000] and [Fernandez and Yuan, 2000] patterns are presented in a systematic way, through context, problem, forces, solution, consequences, known uses and related patterns. The solution part is described by class, state transition, sequence and activity diagrams. Patterns described in [Hamza and Fayad, 2002] adopt a similar strategy.

Our approach also uses those diagrams, but organised in a different way. Moreover, we include, more explicitly, some aspects that we found important, e.g., evolution, dependency, priorities and conflict resolution of requirements, and anti-patterns.

To demonstrate the template utility and adequacy we filled the gaps in the original *Party* pattern description (but note that we did not include all the sequence diagrams, for space reasons). The additional information makes the pattern more complete and easier to understand.

We believe that this template will contribute to provide more useful and organised descriptions of analysis patterns.

5 Conclusions

This paper discussed and compared the main characteristics and limitations of some approaches for requirements and analysis patterns, and the appropriateness of the term “requirements patterns”. Moreover, we outlined the main problems concerning the specification and adoption of requirements analysis patterns by the requirements engineering community. Afterwards, we presented a solution to these problems by defining a template that gathered elements from existing approaches and incorporated new ones that we considered that were missing. The approach was illustrated by applying it to the *Party* pattern described in [Fowler, 1997]. We believe that the approach presented here will improve the specification of analysis patterns with more detailed information.

For future work we intend to apply the template to different patterns as a rigorous way of validation, and define a process model for requirements that incorporates the approach described here.

Acknowledgements

We want to thank our shepherd, Eduardo Fernandez, for his priceless support and all the colleagues of our group discussion at the workshop for their helpful suggestions.

References

- [Araujo and Weiss, 2002] I. Araujo and M. Weiss (2002). "Linking Patterns and Non-Functional Requirements", *PLoP 2002*, Allerton Park, Monticello, Illinois, USA.
- [Buschmann et al., 1996] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal (1996). *Pattern-Oriented Software Architecture: A System of Patterns*, John Wiley & Sons.
- [Brown et al, 1998] W. Brown, R. Malveau, H. McCormick, T. Mowbray (1998). *Anti-Patterns: Refactoring Software, Architectures and Projects in Crisis*, John Wiley & Sons.
- [Chung et al., 2000] L. Chung, B. Nixon, E. Yu, and J. Mylopoulos (2000), *Non-Functional Requirements in Software Engineering*, Kluwer.
- [Fernandez and Yuan, 2000] E.B. Fernandez, X. Yuan (2000), "Semantic Analysis Patterns", *19th International Conference on Conceptual Modeling, ER2000*, Salt Lake City, UT, USA, pp. 183-195.
- [Fernandez et al., 2000] E. B. Fernandez, X. Yuan, S. Brey (2000). "Analysis Patterns for the Order and Shipment of a Product", *PLoP 2000*, Allerton Park, Monticello, Illinois, USA.
- [Fernandez and Yuan, 2001] E. B. Fernandez, X. Yuan, "An Analysis Pattern for the Repair of an Entity", *PLoP 2001*, Allerton Park, Monticello, Illinois, USA.
- [Fowler, 1997] M. Fowler (1997). *Analysis Patterns - Reusable Object Models*, Addison Wesley.
- [Gamma et al., 1995] E. Gamma, R. Helm, R. Johnson, J. Vlissides (1995). *Design Patterns: Elements of Reusable Object-Oriented Software*, New York, Addison-Wesley.
- [Geyer-Schulz and Hahsler, 2001] A. Geyer-Schulz, M. Hahsler (2001). "Software Engineering with Analysis Patterns", *Technical Report 01/2001*, Institut für Informationsverarbeitung und -wirtschaft, Wien, Austria.
- [Hamza and Fayad, 2002] H. Hamza, M. Fayad (2002). "A Pattern Language for Building Stable Analysis Patterns", *PLoP 2002*, Allerton Park, Monticello, Illinois, USA.
- [Hamza and Fayad, 2003] H. Hamza, M. Fayad (2003). "The Negotiation Analysis Pattern", *EuroPLoP 2003*, Irsee, Germany.
- [Jackson, 2000] M. Jackson, *Problem Frames: Analyzing and Structuring Software Development Problems*, Addison Wesley, 2000.
- [Konrad and Cheng, 2002] S. Konrad, B. Cheng (2002). Requirements Patterns for Embebed Systems. *IEEE Joint International Conference on Requirements Engineering*, Essen, Germany, 2002.
- [Robertson, 1996] S. Robertson (1996). "Requirements Patterns Via Events / Use Cases", *The Atlantic Systems Guild*. http://www.systemsguild.com/GuildSite/SQR/Requirements_Patterns.html
- [Schmidt et al., 2000] D. Schmidt, M. Stal, H. Rohnert, F. Buschmann (2000). *Pattern-Oriented Software Architecture. Volume 2: Patterns for concurrent and network objects*, John Wiley & Sons.
- [Whitenack, 1994] B. G. Whitenack, Jr. (1994). "RAPPeL: A Requirements Analysis Pattern Language for Object Oriented Development", *Knowledge Systems Corp*, <http://www1.bell-labs.com/user/cope/Patterns/Process/RAPPeL/rapel.html>
- [Yuan and Fernandez, 2003] X.Yuan, E.B.Fernandez (2003). "An Analysis Pattern for Course Management", *EuroPLoP 2003*, Irsee, Germany.
- [Zhen and Shao, 2002] L. Zhen, G. Shao (2002). "Analysis patterns for oil refineries", *PLoP 2002*, Allerton Park, Monticello, Illinois, USA.

VikingPLoP 2004

A Pattern Language for Participants of Standardization Work

Juha Pärssinen
juha.parssinen@vtt.fi

Abstract

This paper presents a pattern language for people who are going to participate in standardization work. In this paper, the technical details of communication protocols or other standardization artifacts are not the primary focus. Working on standardization means working in a heterogeneous group, and decisions that are made need to balance conflicting forces. The resulting solutions tend to be compromises from proposals made by one or more participants of the standardization effort. This pattern language gives advice to readers to reach the best possible compromise.

Introduction

This paper presents a pattern language for people who are participating in standardization work. It gives advice to readers who are not professional consensus makers (or politicians), but technical experts who participate in standardization work, and are the main responsible persons of their company.

There are four typical cases why an individual want to participate in standardization: to learn a standard as early as possible; to investigate what others have invented; to protect own work to avoid major changes (to protect investments in their own implementation); to delay or even stop the standardization work. This language doesn't consider the last case, which can be seen as a non-generative activity. Readers are advised to read Machiavelli [2] and Adams [5] if they want to master also those aspects of politics. However, in this paper some advice are given to defend against power politics.

Patterns in this language are presented in the sequence they are usually applied. *Compromises Anyway* is a starting-point for this language. It sets the context for all patterns. Other patterns are in two groups: patterns used before standardization meetings and patterns used during standardization meetings.

Compromises Anyway: Accept the fact that you (and hopefully other participants too) have to be ready to make compromises.

Before standardization meeting

Do Your Homework: Prepare well before meetings. Different phases of preparation are divided to three patterns: *Know Things*, *Know People* and *Know Yourself*.

Know Things: Start from pure technical values, and leave the rest for a moment. Study in advance all relevant technical aspects of standardization.

Know People: Study carefully the people who participate in standardization work in advance. In this pattern, people or participants can be considered as individuals and as the whole company.

Know Yourself: Know yourself well before you need to make any decisions. In this pattern “knowing yourself” means knowing not only you, but also the company or the organization you are working for.

Back-up Team Ready: You need to organize in advance a back-up team, which you are able to alert during the meetings.

During standardization meeting

Decisions will stay. Typically everything decided is difficult to change afterwards.

Documents define the Truth. Everything written down can and will be used afterwards.

Concepts over Names: If needed, ease your demands a little: keep the concept as it is, but accept a new name for it.

Use an Example: Examples make things easier to understand.

Use Straw Poll: Straw poll can be used during meetings to see how things are going, and to avoid formal voting.

Be Cool as a Cucumber. You can participate in negotiations more efficient, if your feelings are not running high.

Create a Network. Individuals have typically less knowledge and power than groups.

Pattern language

Compromises Anyway

The goal of standardization work is a standard that can be accepted by participants of a working group and at least majority of the voters of the concerned standardization body. If a standardization project is working in an area that is very far away from commercial products, or the area is so new that none has done anything, this is the rare case when the work can be based purely on solving emerging technical questions.

However, in the real world typically at least some of the participants have something important for them to push and/or protect during standardization work. They have already implemented first prototypes, and they want the standard to reflect what they have implemented to protect their investment and also benefit to be first in the market. This typically means that at least some participants have conflicting proposals and goals.

Therefore:

The only way to participate in standardization work and get it done is to accept the fact that you (and hopefully other participants too) have to be ready to make compromises. You should not think like: “If I have to make compromise I will lose”. If you will eventually have a standard that is available early enough and it contains a feasible set of compromises to participants, it is a win-win situation for everyone participating. If the whole work fails totally, it is a lose-lose situation to everyone who need to have a standard in concern.

There are typically two ways to make a compromise: give-and-take and take-all. Of course, in real life standards, there can be sections that use give-and-take and sections that use take-all.

In a give-and-take compromise every participant needs to make trade-offs, they work together towards a single standard, starting possibly from several conflicting initial proposals. Standards made using this approach are typically smaller, easier to understand, and more consistent than those made using take-all approach. The author strongly recommends give-and-take compromise for standardization work.

Unfortunately, it is not always possible to make trade-offs, and for this reason a take-all compromise is used. In a take-all compromise several conflicting initial proposals are put together, but they are not harmonized as in a give-and-take compromise. Conflicting parts can be hidden as alternative sections or as optional sections of the standard. This will usually lead to an ambiguous standard, which also breaks the well-known KISS principle: Keep It Simple Stupid! A take-all compromise should be used only if the other choice is a long (possibly infinite) delay of the standard.

There can be another kind of problem if standardization is started from scratch: the standard might propose impossible things. For this reason in some standardization organizations work is based only on working implementations.

In this pattern language some advice is given to the reader for reaching the best give-and-take compromise possible. However, these patterns are useful if the take-all approach is taken. Patterns appear in two groups; those used before the meeting and those used during the meeting.

Before the Meeting

Do Your Homework

In standardization work, as in any other creative activity, people are in the middle of a swift information flow. They need to calculate the net effect of many conflicting forces of all technical aspects of concern. For most of the people it is not possible to evaluate these things instantly. Typically time and advice are needed from other experts to find a relevant solution.

Therefore:

Do your homework. Homework in this particular case is a plan or a procedure explaining what you want to achieve and avoid during standardization work. It contains information about particular points which are negotiable and which are not. To do your homework, collect information about all relevant things in advance, and spend time to analyze them with your colleagues. This includes also debriefing of the previous meetings.

You cannot know exactly in advance how much effort and time you need to spend for your homework to be efficient in the meeting, but in the end, you have to do your homework. This fact was written down also by Sun Tzu, a general from ancient China. In this text the words enemy, yourself, heaven, and earth are underlined, when they refer to concepts introduced in quotations of this paper. These concepts should not be taken literally, e.g. enemy in this paper is your opposing side, not your hereditary enemy.

“If you know the enemy and know yourself, you need not fear the result of a hundred battles. If you know yourself but not the enemy, for every victory gained you will also suffer a defeat. If you know neither the enemy nor yourself, you will succumb in every battle.”

Sun Tzu

You should spend time to make your plan, one made during the flight to the meeting place is usually made too late. You need time to read reference material and to talk with people in your home organization. You also need time to make things clear for yourself.

“My intelligence does not stop at my skin.”

Howard Gardner

Integral parts of the intelligence of an individual are also his or her tools, i.e. Papers, books, and computer with documents and databases, and his or her network of colleagues [3].

People who don't do their homework (properly) very often follow strategy: “If I don't say anything, I will appear as if I understand everything”. However, this is only a short term “weasel” solution (see [5] for more complete definition of weasels and their behaviour in workplaces). Sooner or later there will be time to decide and then you won't have anything you can use to proceed wisely. Other people can guide and potentially mislead you if they want. If you like the concept of an anti-pattern, this could be one.

There are several aspects which you should sort out before participating in the meeting, these are considered one by one in the next patterns. These aspect are divided into three patterns following one of Sun Tzu's most quoted sentences:

“Hence the saying: If you know the enemy and know yourself, your victory will not stand in doubt; if you know heaven and know earth, you may make your victory complete.”

Sun Tzu

In the previous quotation, the enemy can be understood as the other participants or people, yourself can be understood as (obviously) yourself, heaven (explained as a climate in [1]) can be understood as a techno-political situation including emotions and feelings, and earth can be understood as a technology, i.e. hard facts, in standardization area. Heaven (techno-politics) and earth (technology itself) together define domain for standardization.

In the following three patterns these four aspects are explained in the standardization context. *Know Things* pattern contains earth, *Know People* pattern contains enemy, and *Know Yourself* pattern contains yourself. These patterns will help you to do your homework. Heaven is divided between *Know People* and *Knowing Yourself*.

Know Things

It does not matter whether a standardization project is starting from a white sheet or there is a lot of existing prework, in any case you need to understand (and possibly solve) emerging technical questions during standardization work.

As explained in the *Do Your Homework* pattern, standardization domain contains two parts: heaven and earth. You see this clearly when you are evaluating any technical issue. There are always two sides to take into consideration: “engineering” values (earth) e.g. feasibility and performance, and “techno-political” values (heaven) e.g. who owns patents and who has invented technology. If you try to understand and evaluate both sides at once it can be overwhelming.

Therefore:

Start to analyse pure technical facts, and leave politics and potential emotions for a moment. It is much easier to judge the hard facts first, and later enhance your judgement by politics and potential emotions. This enhancement will be done in patterns *Know People* and *Know Yourself*.

Study all relevant technical aspects of standardization in advance. You don't have to be an expert in everything, but you should know enough to understand the documents and to be able to participate in discussions. However, at the same time when you are studying technical information, collect also information related to politics and emotions, including the following: who has invented things, who owns possibly patents, and which companies use those. That information is used in the next patterns, *Know People* and *Know Yourself*.

A good example of this pattern is 3rd generation mobile phone standardization. There are two technologies strive to better utilize the radio spectrum by allowing multiple mobile phone users to share the same physical channel: TDMA (Time Division Multiple Access) and CDMA (Code Division Multiple Access). They have several fundamental technical differences which are completely out of the scope of this paper (see [6] if you want more information), but also one (among others) interesting technological difference: one single company owns majority of CDMA patents. If you follow this pattern, you study TDMA and CDMA from engineering viewpoint. And make a note for yourself about interesting patent issue.

Know People

During your studies of technical aspects in *Know Things* pattern, you have also collected related non-technical information. In this pattern, people or participants can be considered as an individual or as the whole company.

In a standardization body there are typically several participants from different interest groups which usually have conflicting goals. To do your homework properly, it is important to know other participant's goals as early as possible:

“By discovering the enemy's dispositions and remaining invisible ourselves, we can keep our forces concentrated, while the enemy's must be divided.”

Sun Tzu

You can estimate that the result of work will be a compromise made by people, but you want to push it as much as possible your preferred direction and avoid other ones.

Therefore:

Study carefully the people who participate in standardization work. You should know participants' work history, e.g. what kind of education they have, what publications they have written, and have they participated in this kind of work before. Usually people are kind enough to tell all this in their WWW home page. If you have time, skim through their publications. It is also important to know what kind of character the person is. The only reliable way to find out this is to get to know them, other people's opinions are not so trustworthy. All this information of participants is also useful to you when you are building your network, see *Create a Network* pattern.

If the inventors of technical aspects in concern participate in work, they typically defend their work rigorously. People tend to think that what they have done is like their child, and nobody else than themselves are allowed to change it or really understand it. You

should also know who are the real leaders of the work to know to whom you should talk to make any kind of progress. Remember, that position of people in the organization and their role as leaders are not always comparable [3].

Study also the participating companies, e.g. which are their core technology areas and which are their business segments. For example a manufacturer typically has a completely different kind of interests than a service provider. You can expect companies to try to lead the work in the direction which is most important to them.

If we continue our TDMA vs. CDMA example from the previous pattern, now it is time for you to analyze the effects of patents and other non-engineering values of each technology. In the previous pattern you make a note that one single company owns majority of CDMA patents. For this reason it can be expected that this company wants CDMA technology to be used in forthcoming standard.

It is also important to know why people (or companies) participate in this particular project. There are four typical cases: they want to learn standard as early as possible; they want to investigate what others have invented; they want to protect their own work to avoid major changes (to protect their investments to implementation); they want to delay or even stop the standardization work because they have their own reasons not to have standard in this area. The first two groups, i.e. learners and investigators, are usually mostly harmless, but the other two will eventually cause troubles. It is important to know are people either trying to protect their own work or trying to stop the project. If they want to protect their work, they are usually willing to make technical compromises. But if they just want to stop work, there are nothing to negotiate. In that case there are only political ways to solve the situation.

Know Yourself

During your studies of technical aspects in *Know Things* pattern, you have collected not only hard technical facts, but also some amount of related non-technical information. In this pattern “knowing yourself” means knowing not only you, but also the company or the organization you are working for.

When you are preparing yourself for the meeting you are not only studying technical issues, you are also learning the strategy of your company. Information about technical issues is useless if you don't know motive for decisions and value of artifacts behind them.

Therefore:

Read technical documents made in your own company and interview people who wrote them. Interview also people who are (or will be) implementors of prototypes, or enduser products. They typically have a lot of opinions, and quite often they are not documented anywhere, i.e. they have *tacit knowledge* [4].

Read also strategy documents and interview people who have written them. Strategy documents usually explain long-term goals of a company, but do not explain why these are chosen [4].

An example of a typical case is that your company has a prototype ready, and as much as possible should be re-used in a final product. In this case you need to know enough details about it to understand what kind of changes in the standard will render your prototype useless, and what kind of changes are only cosmetic. However, sometimes it is just better to throw the prototypes away. Unfortunately you are not typically the person to make that decision.

You should also carefully study the motives of your company to participate in standardization effort. Four typical cases are mentioned at the end of *Know People* pattern.

During this process of reading documents and interviewing your company's people you should also build your network of colleagues (see *Create a Network* pattern), and find candidates for your back-up team (see *Back-up Team Ready* pattern).

When you know yourself it is obviously easier to set goals, and give to each of them preferences. However, you should not reveal any information about your list of preference before you really have to. Otherwise you will lose important currency you can use in bargains, as written by Sun Tzu:

"By discovering the enemy's dispositions and remaining invisible ourselves, we can keep our forces concentrated, while the enemy's must be divided."

Sun Tzu

Real life examples of this can be found from [5]:

"... you should never do: tell a repairman how much you think something will cost before you get the estimate."

Scott Adams

A typical way to make a budget in companies is to ask for more than you actually need to have eventually enough funding. Or as can be found from [5]:

"Toss in lots of ridiculous clauses so you have plenty to negotiate away later"

Scott Adams

Obviously this doesn't work if your opponent knows your goal and preferences in detail.

Now it is time to continue the TDMA vs. CDMA example. You have studied in the previous pattern both of these technologies, and you have studied in this pattern the strategy of your company and how these patents will affect to your company, e.g. what kind of business relations you have to the patent owner. Now it is time to choose your side based on this information.

Back-up Team Ready

In previous patterns you have done your homework, i.e. studied the technology (*Know Things*), other participants and their organization (*Know People*), and also yourself (*Know Yourself*). However, no matter how well you prepare, there could always be a case for which you haven't prepared. Everything can change during standardization project, i.e. new participant can join to technical group or new relevant technology can emerge.

Therefore:

You should have a possibility to alert your back-up team during the meetings (at least virtually) anytime. Organize a team, which is available all the time during meeting days. You might have found good candidates for your team when you previously use *Know Yourself* pattern. Your network of colleagues (see *Create a Network* pattern) could also be useful.

If it is possible, take your team with you to meeting. If this is not possible e.g. for financial reasons, make sure you have possibility for a quick teleconference whenever you need (remember time-zones). Remember that if you have a big group of people with you, it is more difficult to say: “I have to negotiate with my colleagues at office” and buy time in this way. However, a big group have also some advantages, e.g. from [5] can be found discussion about “meeting weight” of each participant.

You should also have a small team located to your office which is ready to find and to send to you those parts of reference material you have forgotten to take with you.

During Meeting

Decisions will stay

Several fields of the modern technology have one common force: backward compatibility. Sometimes there are technical reasons for that, sometimes reasons are more political. In standardization work people tend to play an important role related to this force. Any kind of significant change is not possible in subsequent versions of standard because people who did the original work will not be pleased and will be strongly against any changes.

Therefore:

Work hard to get every decision as close as possible to the right one for you. Later on it is difficult to make any significant change to it because it will have to be “backward compatible”. This means that you should *Do Your Homework* properly and when an issue is taken into discussion you should have your opinion ready. If you have accepted something, it will be very difficult to remove it later.

You should understand that anything added to standard will not just go away. The size of the standard always increases, never decreases. The only possibility to decrease the size of a standard is when people who put things in (inventors) eventually go away.

Documents define the Truth

You and the whole standardization group have worked hard, and reached eventually some kind of consensus. When time goes by, decisions made during meetings will not be so easy to remember, and people often tend to remember same things differently.

Therefore:

Make sure that every decision, even the smallest ones, are written down and documented in meeting minutes and/or draft documents. Make sure that the whole document history is recorded, and documents are distributed and kept so that no-one can change them afterwards unnoticeably. As time goes by, the only things which matter will be written documents.

“The strongest memory is no match to the palest ink.”

Chinese Proverb

When work will be continued in future, be also sure that documents which are used as a base of work are the same as they were after previous meeting. Some examples of this misuse of this pattern can be found from [5], like “Insane Forgetting” on pp. 148.

Concepts over Names

You are working in the group whose aim is to create a standard. During the process new concepts are found and possibly added. However, participants of the work group are from different kind of interest groups and some of them have already those new things included to their goals, some may try to avoid having them included. Typically this will wind up in a long and sometimes tense discussion.

In the ideal situation new concepts added to standard are named as you like. However, you might end up in the situation that something you like to be added to standard is at stake.

Therefore:

If there is strong resistance against something you want to be added to the standard, you can try to ease your demands a little: keep the concept as it is, but accept a new name for it. In some cases people are not against the thing itself, they are against how it is named in proposal. If the name of concept in concern is changed they could accept it without thinking twice. The reason behind this phenomena is that for many people names are most important things, and the first name given during standardization is the most important one. They potentially guide thoughts to particular direction or area, and some of them they want to avoid. For this reason they cannot even accept any names from that area.

Of course this pattern doesn't work if you are the one who want a particular name to be in the standard. However, especially in this case you should also investigate if this is tried to be used against you. It is possible that someone is trying to introduce names which they know you are not willing to accept, and in this way the whole work is tried to delay or even ruin completely.

One remaining issue is that you should recognize if two participants are proposing same underlying concepts, but are using different name for it. In some case it might take long time before they understand this situation. In this case you should investigate if they really want that name. If not, the case can be closed quickly.

Use an Example

You are working in the group whose aim is to create a standard. There are things included in the standard that are complicated to understand if there is only a formal or semiformal description of it. If things are not understood, participants will have difficulties to decide if they are for or against them.

Therefore:

Use informal examples to make things easier to understand. In standards examples are usually considered as informal, i.e. they do not define a standard. Do not forget that carefully selected examples guide people understanding, and usually people stick to their first impression. Remember that this works also for yourself when you are reading documents.

Use Straw Poll

Participants of the work group are from different interest groups which usually have conflicting goals. Often discussion about different issues will be long and sometimes tense. If people have taken a strong opinion they have difficulties to change it afterward. Especially an individual vote (even your own) is very difficult to change afterward.

Therefore:

During meetings use straw poll to see how things are going – try to reach “strong consensus” and test the water from time to time. People have easier to change their opinion if they can do it without losing their face. For this reason do not put anything to the formal vote if not absolutely necessary. People will not easily vote differently in the future, even if they have changed their mind.

Be Cool as a Cucumber

Often in a standardization group there are conflicting views, and sometimes long and possibly tense discussions cannot be avoided. One weakness of a group of talented people is that they spent too much time in competitive debate, and the debating becomes an unending session of academic showmanship [3].

Emotions also spread easily, which used to be an alarm signal of danger, e.g. approaching predator, in prehistory of man. Nowadays emotions can spread quickly in a working group whose participants are tired, and people can start to provoke to each other.

Therefore:

Be cool as a cucumber. When feelings run high, it is almost impossible to think and to behave rationally. People, who can take things calmly, can be the most efficient in the negotiations. [3]

You can try the following when feelings are starting to run high: calm down, listen your own feelings, and recognize them. Show to other participants that you want to discuss and solve problems – not fight. Tell your opinion calmly, and try to find a reasonable compromise [3]. It is also possible to have short break, and give time for people to cool down. Unformal discussion during break can also sometimes solve problems.

Remember, that you can understand your opponent, and be in a spirit of compromise, but you don't have to be of the same mind and be ready to accept everything [3].

Create a Network

Single participants of a standardization group typically don't have much influence during meetings, usually the principle “one man, one vote” is used. If you are trying to propose a new thing, it is interesting to have an estimation how many will probably vote for it and how many will probably vote against it. You like also to gather support for your proposal before a meeting. Often many things are actually decided long before they are brought to official meetings. How can you increase your influence and reduce your uncertain factors?

Therefore:

Create a network of colleagues. Part of the expertise of a successful standardization participant is the ability to create alliances and collect necessary information between official work meetings. You should also use the opportunity to build your network when you use *Know People* and *Know Yourself* patterns.

Creating a network is a continuous activity. You should be able to participate lunches, breaks, and evening activities together with other participants. Most of the alliances are collected at “off-duty”. You should also be able to discuss other subjects than ongoing work, people and also yourself need breaks during long meetings. It obviously is also easier to make compromises with people you know than completely strangers. However, quite often people spend all time outside meetings (and sometimes even during meetings) to do their other duties.

If you are in charge of hosting a standardization event you can also plan extra time for networking activities, i.e. social events, including restaurants, excursions, or sport activities. For example in ETSI they have organized once a week an unofficial beach volley ball playing session for visiting experts.

Acknowledgments

I wish to thank my shepherd, Peter Sommerlad, for his valueable comments during shepherding for VikingPLoP2004, and also for the idea to add weasels [5] to this paper.

The previous version of this pattern language has been workshopped at VikingPLoP2003 in Bergen, Norway. I would like to thank especially Richard Gabriel for his excellent work as a moderator, Linda Rising for her numerous comments on my pattern language and Neil Harrison's guidance as a shepherd of this paper for VikingPLoP2003.

I also would like to thank all VikingPLoP2003 and VikingPLoP2004 workshop participants about their comments and encouragement.

For this pattern language I have interviewed experts who have experience from different standardization organizations: Morgan Björkander, Antti Huima, and Steve Randall. Without them I would not have been able to write this pattern language.

References

- [1] Sun Tzu, *The Art of War*, Project Gutenberg Etext #132, 1994.
- [2] Niccolo Machiavelli, *The Prince*, Project Gutenberg Etext #1232, 1998.
- [3] Daniel Goleman, *Working with Emotional Intelligence*, Bloomsbury, 1998.
- [4] Edgar H. Schein, *The Corporate Culture Survival Guide*, Jossey-Bass, 1999.
- [5] Scott Adams, *Dilbert and the Way of the Weasel*, Bantam, 2002.
- [6] CDMA vs. TDMA, <http://www.arcx.com/sites/CDMAvsTDMA.htm>, 2004.

VikingPLoP 2005

Patterns for Documenting Frameworks – Part I

Authors Ademar Aguiar, Gabriel David

FEUP & INESC Porto, Universidade do Porto

E-mail: ademar.aguiar@fe.up.pt, gtd@fe.up.pt

Good design and implementation are necessary but not sufficient pre-requisites for the successful reuse of object-oriented frameworks. Although not always recognized, good documentation is crucial for effective framework reuse, and comes with many issues. Defining and writing good quality documentation for a framework is often hard, costly, and tiresome, especially when not aware of its key problems and good solutions for them. This document contributes two patterns to the work in progress of writing a pattern language that describe proven solutions to recurrent problems of documenting object-oriented frameworks. The patterns aim at helping non-experts on cost-effectively documenting object-oriented frameworks. The two initial patterns here presented address the problems of *guiding the readers on the documentation* and *introducing the framework*, respectively the patterns “DOCUMENTATION ROADMAP” and “FRAMEWORK OVERVIEW”.

Introduction Object-oriented frameworks are a powerful technique for large-scale reuse capable of delivering high levels of design and code reuse. As software systems evolve in complexity, object-oriented frameworks are increasingly becoming more important in many kinds of applications, new domains, and different contexts: industry, academia, and single organizations.

Although frameworks promise higher development productivity, shorter time-to-market, and higher quality, these benefits are only gained over time and require up-front investments. Before being able to use a framework successfully, users usually need to spend a lot of effort on understanding its underlying architecture and design principles, and on learning how to customize it. This usually requires a steep learning curve that can be significantly reduced with good documentation and training material.

This paper contributes with two patterns to the work in progress of writing a pattern language addressing problems of documenting frameworks, some of the several technical, organizational, and managerial issues that must be well managed in order to employ frameworks effectively.

Pattern language The pattern language comprises a set of interdependent patterns that aim at helping developers on becoming aware of the typical problems they will face when documenting object-oriented frameworks. The patterns were mined from existing literature, lessons learned, and expertise on documenting frameworks, based on a previous compilation about framework documentation [1].

The pattern language describes a path commonly followed when documenting a framework, a path not necessarily required to follow strictly from start to end to

achieve effective results. In fact, many frameworks are not documented as extensively as suggested by the patterns, due to different kinds of usage (white-box or black-box) and different balancing of tradeoffs between cost, quality, detail, and complexity. One of the goals of these patterns is precisely to expose such tradeoffs in each pattern, and to provide practical guidelines on how to balance them to find the best combination of documents to the specific context at hands.

According to the nature of the problems addressed, the patterns are organized in *process patterns* related with the process of cost-effectively documenting frameworks (*how to do it? which activities, roles and tools are needed?*) and *artefact patterns* (*which kinds of documents to produce? what should they include? how to relate them?*), to which belong the patterns here documented.

Artefact patterns Artefact patterns address problems related with the documentation itself, here seen as an autonomous and tangible product independent of the process used to create it. They provide guidance on choosing the kinds of documents to produce, how to relate them, and what to include.

Similarly to other technical documentation, the overall quality of framework documentation is complex to determine and assess, and this is perhaps the first issue. Documentation must have quality: it must be easy to find, easy to understand, and easy to use [2]. Task-orientation, organization, accuracy, and visual effectiveness are among all documentation quality attributes, the most difficult ones to achieve on framework documentation [1].

From the reader's point of view, the most important issues are on providing accurate task-oriented information, well-organized, understandable, and easy to retrieve with search and query facilities. From the writer's point of view, the key issues are on selecting the contents to include, on choosing the best representation for the contents, and on organizing the contents adequately, so that the documentation results of good quality, easy to produce and maintain.

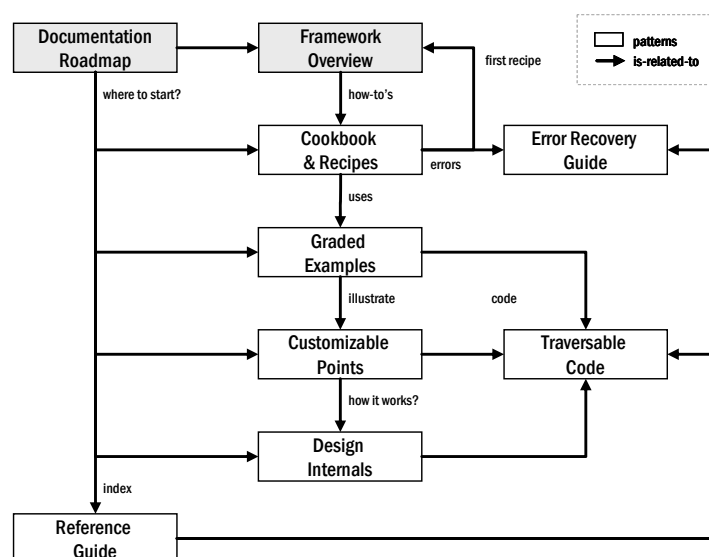


Figure 1 Documentation artefact patterns and their relationships.

Patterns overview To describe the patterns, we have adopted the Christopher Alexander's pattern form: *Name-Context-Problem-Solution-Example* [3]. Before going to the detail of each pattern, we will overview the pattern language with a brief summary of each pattern's intent. For contextual purposes, all the artefact patterns are overviewed below and depicted in Figure 1 highlighting the two patterns described in this paper.

Documentation Roadmap helps on deciding what to include in the documentation overview to provide readers of different audiences with useful and effective hints on what to read to acquire the knowledge they are looking for.

Framework Overview suggests providing introductory information, in the form of an overview that briefly describes the domain, the scope of the framework, and the flexibility offered, i.e. contextual information about the framework, the first kind of information that a framework (re)user looks for.

Cookbook & Recipes explains how to provide readers with information that explain how-to-use the framework to solve specific problems of application development, and how to combine this prescriptive information with small amounts of descriptive information to help users on understanding what they are doing.

Graded Examples describes how to provide and organize example applications constructed with the framework and how to cross-reference them with the other kinds of artefacts (cookbooks, patterns, and source code).

Customizable Points describes how to provide readers with task-oriented information with more precision and more design detail than cookbooks and recipes, so that readers can quickly identify the customizable points of the framework (hot-spots) and to understand how they are supported (hooks).

Design Internals explains how to provide detailed information about what can be adapted and how the adaptation is supported, by referring the patterns that are used in its implementation and where they are instantiated.

Reference Guide suggests what to include as reference information and how to structure the documentation to make it the most complete and detailed as possible to assist advanced users when looking for descriptive information about the artefacts and constructs of the framework.

Traversable Code provides hints on how to organize and present source code, both of the examples and the framework itself, when desired, to make it easy to browse and navigate from and to other software artefacts included in the overall documentation, namely models and documents.

Error Recovery Guide explains how to help users on understanding and solving the errors they encountered when using the framework.

Pattern **Documentation Roadmap**

To completely satisfy all audiences and requirements, the documentation of a framework typically includes a lot of information, organized in different types of documents (framework overviews, examples of applications, cookbooks and recipes, design patterns, and reference manuals), that provides multiple views (static, dynamic, external, internal) at different levels of abstraction (architecture, design, implementation).

Problem The complex web of documents and contents provided with a framework must be organized and presented in a simple manner to the different audiences, so that the readers don't become overwhelmed or lost when using the documentation. In other words, the overall documentation must be easy to use by all kinds of readers, so that each individual reader can quickly acquire the strict degree of understanding she needs to accomplish her particular engineering task (reuse, maintenance or evolution).

How to help readers on quickly finding in the overall documentation their way to the information they need?

Forces **Different audiences.** Readers of different audiences have different needs and interests that must be addressed by the documentation.

Different kinds of reuse. A framework can be reused in different ways, each requiring different levels of knowledge about the structure and behaviour of the framework, and therefore pose different demands on the documentation.

Easy-to-use. Despite its inherent potential complexity, the documentation must result easy-to-use.

Solution Start by providing a roadmap for the overall documentation, one that reveals its organization, how the pieces of information fit together, and that elucidates readers of different audiences about the main entry points and the paths in the documentation that may drive them quickly to the information they are looking for, especially at their first contact.

The roadmap would help both when navigating top-down, from a main entry point of the documentation to concrete topics and subtopics, and when navigating bottom-up from a small piece of information to a bigger one to try to identify it as part of a whole, still unknown in a first contact.

To be effective, a documentation roadmap for a framework should be written in a task-oriented manner and to include the following aids:

- topics organized by audience, kind of task, and order of use, to help readers of different audiences quickly retrieval of the information they need to accomplish the tasks they have in mind;

- emphasis of the main entry points and subordination of the secondary ones, to improve visual effectiveness;
- overview of topics conveying how subtopics are related, to support non-linear readings;
- transitions and links to topics, and from them to the roadmap again, to improve the overall navigability around the roadmap.

Although all the above mentioned aids are important to include, in fact, the optimal level of importance assigned to each one depends on several factors: the *framework* being documented, which *audiences* are you willing to address in the overall documentation, which *kinds of reuse* to support explicitly in the documentation, and which *tasks* to emphasize and subordinate.

Independently of how these factors are balanced, the roadmap should be easy to use, easy to understand, well-organized, and visually effective, a set of quality characteristics that suggest not to include everything in the roadmap, but only the entry points and topics more relevant for the most important tasks of the target audiences.

Examples **JUnit.** The JUnit framework [4] provides a very simple documentation roadmap, where the audiences, kinds of reuse and tasks are implicitly mentioned in the names of the entry points in the documentation and their respective descriptions (Figure 2).

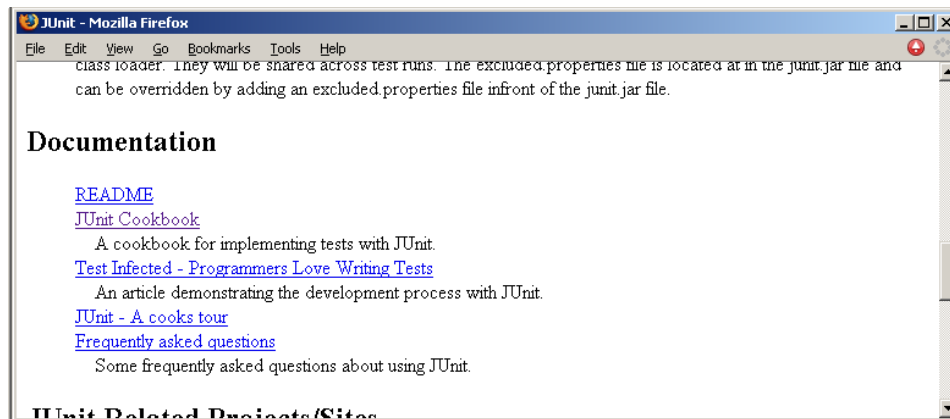


Figure 2 Example of a very simple documentation roadmap delivered with JUnit.

The audiences addressed can be implicitly identified as the following:

- *first-time users*, which will probably be attracted by the “JUnit Cookbook” entry;
- *selectors* and *common users*, by the “Test Infected - Programmers Love Writing Tests” entry;
- and *framework developers*, by the “JUnit - A cooks tour” entry.

The kinds of reuse that are clearly expressed in this roadmap are:

- *instantiating*, by the “JUnit Cookbook” entry;
- *selecting* and *instantiating*, by the “Test Infected - Programmers Love Writing Tests” entry;
- and *mining*, by the “JUnit - A cooks tour” entry.

Finally, the tasks mentioned are simply *how to implement tests*.

This roadmap reflects the simplicity of the JUnit’s framework itself, the intent of the writers on making the documentation simple and short, by documenting the main usage task of JUnit: to write tests.

Swing. The Sun’s JFC/Swing framework [5] is a popular example of a large framework for which exists a lot of documentation.

In part due to its large dimension and vast diversity of possible tasks when reusing Swing, be it black-box or white-box reuse, the roadmap is split along several documents, according to the kind of audience and reuse tasks.

The screenshot shows the Sun Developer Network website. The main heading is "The Swing Connection - Index by Content". Below this, there are three main sections, each with a table of articles:

| About Swing | | |
|----------------------------|--|---------|
| Component Gallery | Graphical overview of the Swing components | |
| Getting Started With Swing | Overview of the Swing/Java Foundation Classes. A good place to start | |
| Swing Architecture | A more in-depth look at the architecture and design of Swing | 4/98 |
| The Java Look & Feel | About the design of the Java look and feel | |
| What's coming in JDK1.4 | Article about APIs introduced in the JDK1.4 release | 11/2/00 |

| Working with Swing Components | | |
|---|--|----------|
| Swing Borders | How to use Borders | 5/98 |
| Containers | Explaining the hierarchy of Swing windows | |
| JList | Advanced JList Programming | |
| JTree | Understanding the Tree Model | |
| TableLayout | Introduction to and design of the TableLayout manager | 2/24/02 |
| High Performance Frequently-Updated JTables | Techniques for using a JTable which needs to frequently update its data. | 12/02/02 |
| Tree Table, part 1 | Building a Tree Table, part 1 | |
| Tree Table, part 2 | Building a Tree Table, part 2 | |
| Tree Table, part 3 | Building a Tree Table, part 3 | |

Note: for Swing Text articles, see below.

| Putting it all together | | |
|----------------------------------|---|---------|
| Accessibility | Not just a nice thing to do — it's now the law. | |
| Card Panel | An alternative to the AWT Card Layout | 1/4/00 |
| Component Orientation | How JFC Components support bi-directional text. | |
| Drag & Drop | DnD basics for AWT and Swing | |
| Dynamic Action Listeners | Generating Action Listeners Dynamically | |
| Dynamic Action Listeners, part 2 | Updated article on Generating Action Listeners Dynamically | 1/21/00 |
| Java Help | How to download and use the Java Help package | |
| Using Java 2D with Swing | Demonstrates the ease of creating beautiful user interfaces with Swing and the Java 2D API. | 2/7/02 |

Figure 3 Example of the complex documentation roadmap for JFC/Swing framework.

Figure 3 presents “The Swing Connection - Index by Content”, which is the document of Swing that mostly resembles a framework documentation roadmap.

.NET. The Microsoft’s .NET framework is another popular example of a very large framework for which also exists a lot of documentation.

For this framework there exists a documentation roadmap that clearly addresses different audiences and different kinds of reuse, from where the reader is driven to other documents more specific to the kind of reuse selected with more detailed about the tasks possible with the framework.

Figure 4 presents the “Getting Started” document, which is intended to first-time (re)users, a secondary document accessible from the right-hand side menu that contains the main entry points to the documentation, organized by kind of audience.

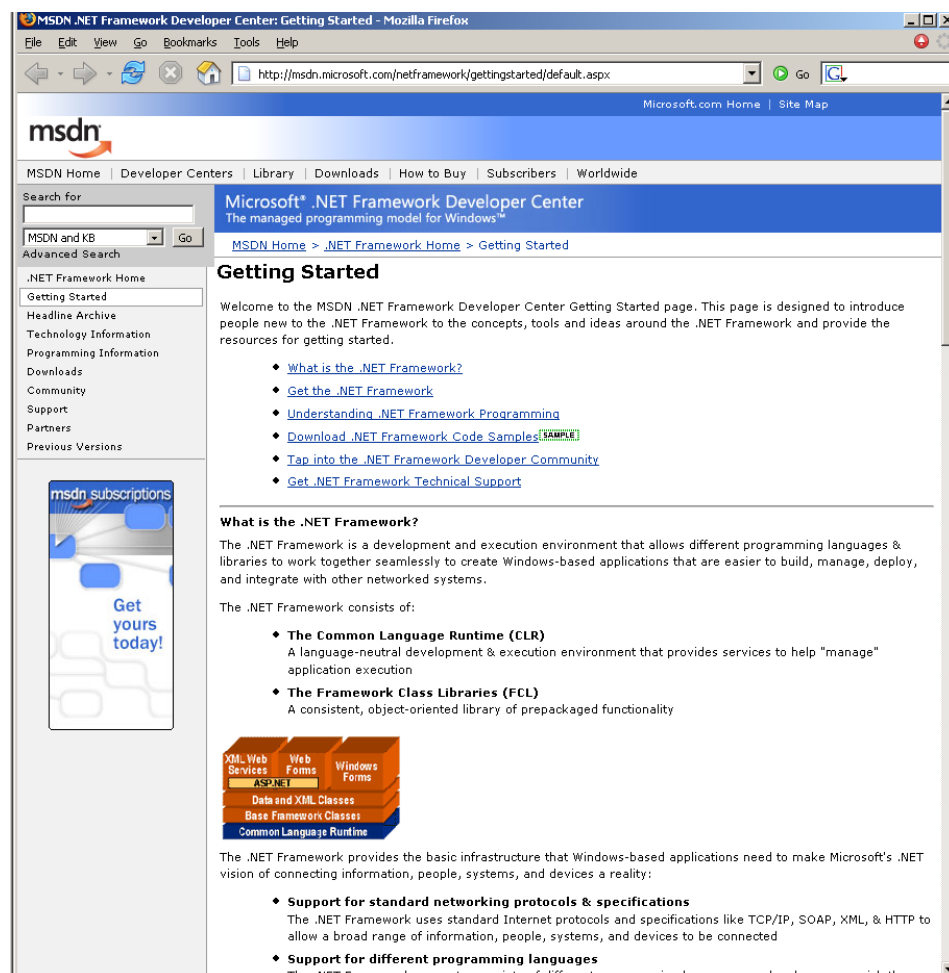


Figure 4 Example of the documentation roadmap for the .NET framework.

Pattern **Framework Overview**

To be effective, the documentation of a framework must include information that explains the purpose of the framework, how to use it, and how it is designed and implemented [4][6].

Problem In addition to different purposes, the documentation of a framework must also meet the needs of different categories of software engineers involved in framework-based application development, playing different roles (framework selectors, application developers, framework developers, framework maintainers, and developers of other frameworks [7]), having varying levels of experience, and therefore requiring different kinds of information.

How to help readers on getting a quick, but precise, first impression of a new framework?

Forces **Different audiences.** In a first contact, the most important kinds of readers to consider are framework selectors, someone who is responsible for deciding which frameworks to use in a project, and new framework users, developers without previous knowledge or experience with the framework [1]. The first kind of information that a new framework user looks for is contextual information. Framework selectors will look for a short description of the framework's purpose, the domain covered, and an explanation of its most important features, ideally illustrated with examples.

Completeness. The readers appreciate complete information, i.e. that all of the required information is available. Completeness implies that all of the relevant information is covered in enough detail, but only the necessary, and that all the promised information is included. But completeness depends on the reader's point of view, and therefore requires knowing the audience and the tasks the documentation should support [2].

Easy-to-understand. Information that is clear, concrete, and written using an appropriate style is usually easy to understand the first time. Clarity (especially conciseness) often conflicts with completeness (especially relevance and too much information), requiring a good knowledge about *what* readers need to know and *when* they need to know it [2]. Concrete examples help readers on understanding what they are learning because they map abstract concepts to concrete things, which readers can see or manipulate.

Solution Provide an introductory document, in the form of a framework overview, that describes the domain covered by the framework in a clear way, i.e. the application domain and the range of solutions for which the framework was designed and is applicable.

In addition, a framework overview usually defines the common vocabulary of the problem domain of the framework. It clearly delineates what is covered by the framework and what is not, as well as, what is fixed and what is flexible in the framework.

An effective way of communicating this information consists on presenting the basic vocabulary of the problem domain illustrated with a rich set of concrete application examples.

This information is of great value for potential users specially during the selection phase because it helps them to evaluate the appropriateness of the framework for the application at hands, and thus fundamentals the selection, or rejection.

In a framework overview, it is common practice to refer or review examples, from simple to complex ones (pattern “GRADED EXAMPLES”), and to refer or include an overview of all the documentation (pattern “DOCUMENTATION ROADMAP”). A framework overview is often the first recipe in a cookbook (pattern “COOKBOOK & RECIPES”).

Examples **JUnit.** The framework overview that accompanies JUnit is represented in Figure 5. It was extracted from the document “Frequently Asked Questions (FAQ)” [8], which, from all the documents delivered with the JUnit’s framework, is the one that most clearly presents the information typical of a framework overview, despite its placement in a FAQ not being evident in a first contact with the documentation.

Although not being a good exemplar of a framework overview, it contains its most basic ingredients (the domain covered, the features, and an overview of the documentation), and thus it reasonably fulfils the requirements of a framework overview.

The biggest problem with the JUnit’s framework overview is that it is not easy to find in a first look at the documentation and is not complete. It is however, easy to understand, what is also very important.

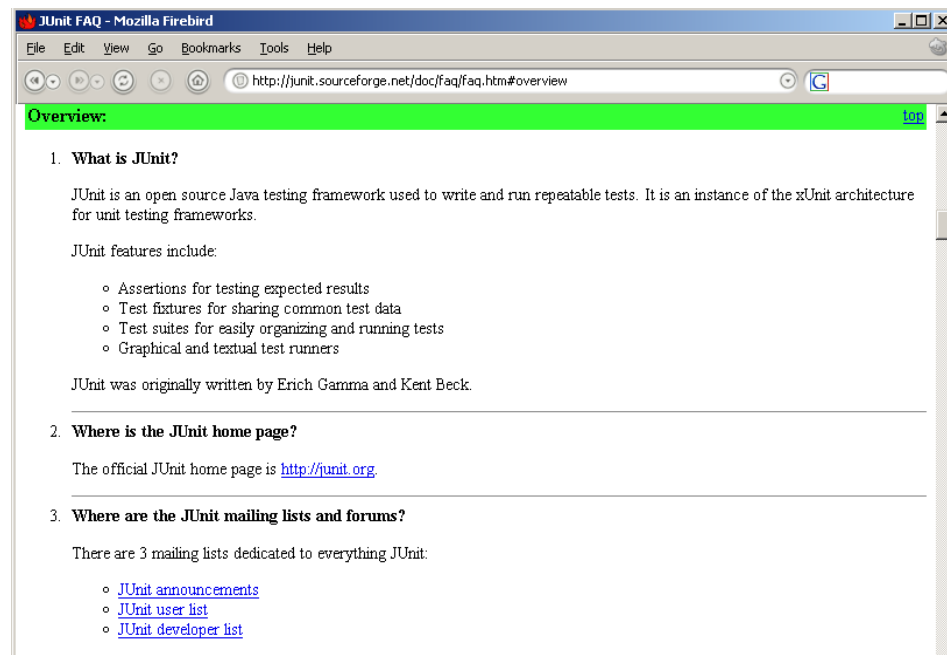


Figure 5 Example of the framework overview delivered with JUnit.

Swing. The framework overview of JFC/Swing is provided in the Sun’s tutorial “Creating a GUI with JFC/Swing”, lesson “Getting Started with Swing“, topic “About the JFC and Swing” (Figure 6).

This framework overview clearly describes the domain covered by the JFC/Swing and its main features. Although, the customizations possible with the framework are described textually in the overview, they are not linked to the concrete examples of applications included in the documentation in other lessons of the tutorial, both visual and source code examples.

Being the JFC/Swing framework so well-known, this placement of the overview so deep in the documentation hierarchy, instead at the top, to be easy to find in a first contact with the documentation, is acceptable and possibly even more convenient, considering that only a small minority of readers are expected to need to learn what JFC/Swing is about. What most readers would need to learn is *what* and *how* they can customize in JFC/Swing to create the GUI they have in mind.

The contents and organization of this framework overview reflects the importance of knowing well the audience and the tasks more likely to need support from the documentation.

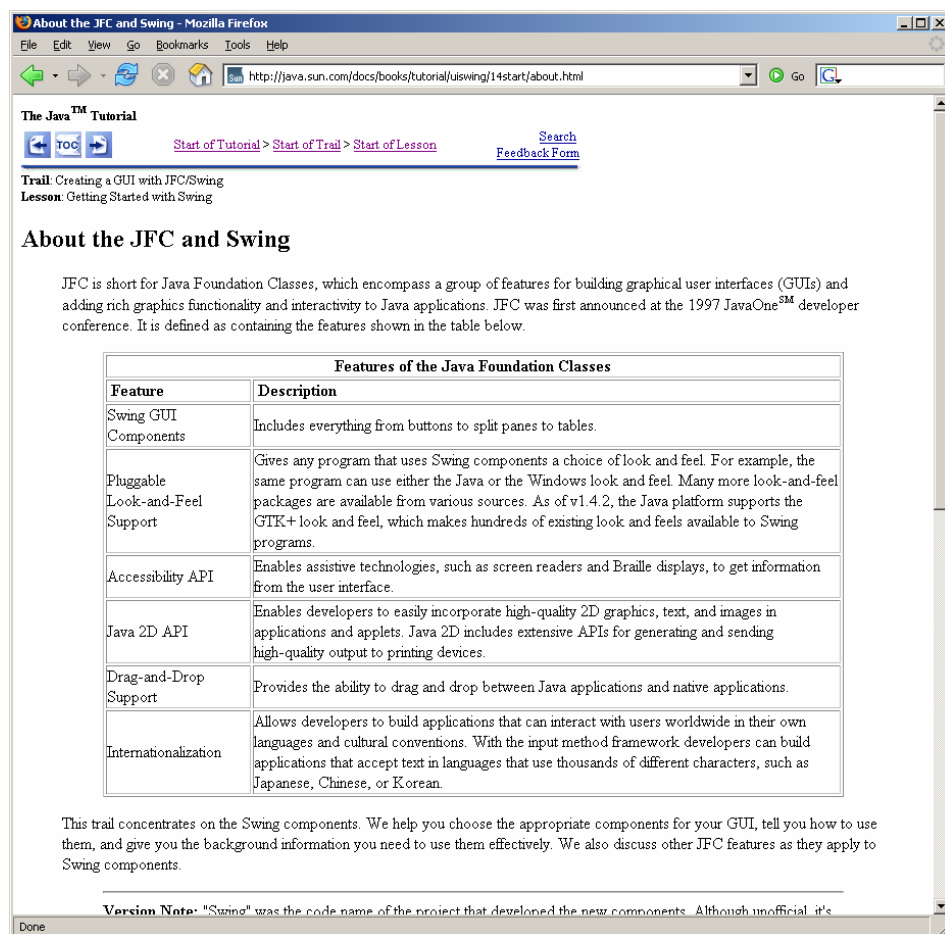


Figure 6 Example of the framework overview provided for JFC/Swing.

.NET. The overview of the .NET framework is provided in the “Getting Started” document (Figure 4), topic “What is the .NET Framework?”. A more detailed technical overview is provided in the “Technology Overview” document (Figure 6).

This framework overview briefly describes the purpose of the framework, its application domain and its main components, and refers other documents containing more specific information about the framework.

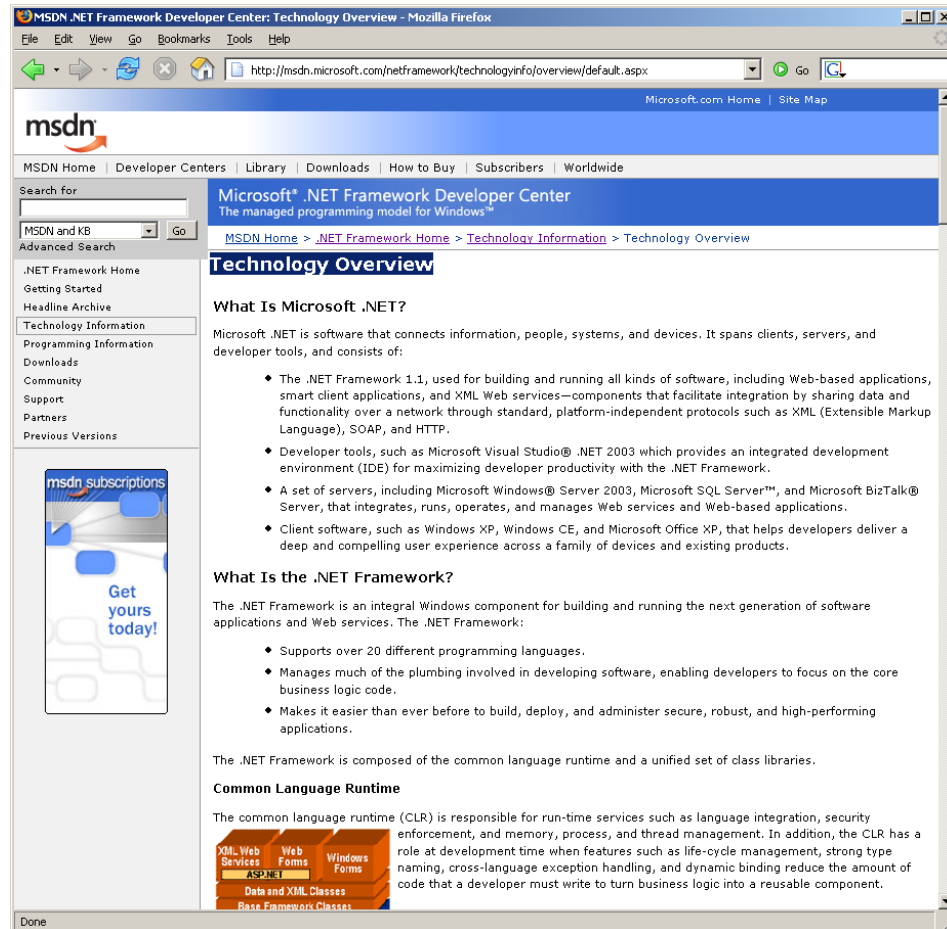


Figure 7 Example of the framework overview provided for the .NET framework.

EMF. The overview document of the Eclipse Modeling Framework (EMF), “Eclipse Modeling Framework Overview”, is another good example (Figure 8). It is very complete and provides a brief presentation of the framework and its key features illustrated with the help of concrete examples.

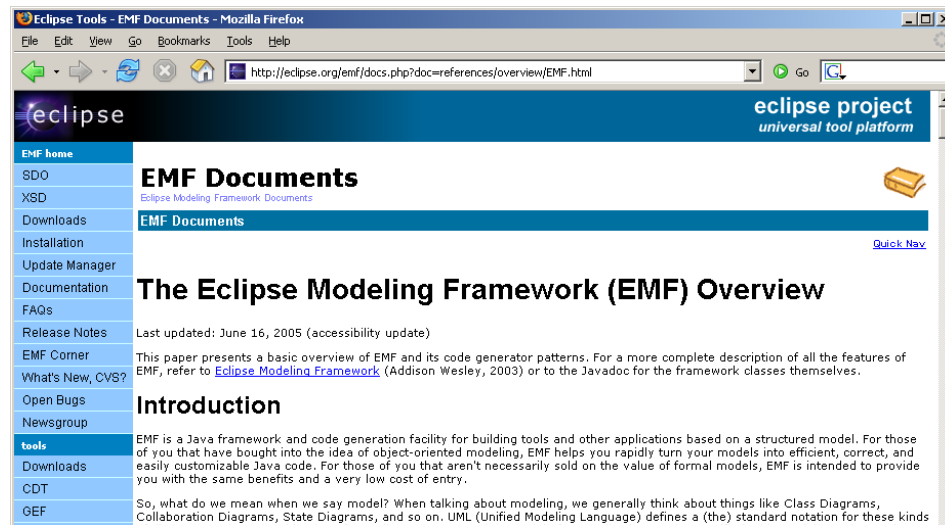


Figure 8 Example of the overview provided with the Eclipse Modeling Framework (EMF).

Credits The authors would like to thank Neil Harrison, for having pushed forward this pattern language to start being workshopped at VikingPLoP'2005, and, most importantly, for the clear and valuable feedback provided on the previous versions of this document.

We would like to thank also Juha Parssinen and Sami Lehtonen for their collaboration during all the phases of this document, Eduardo Fernandez, Kevlin Henney, Klaus Marquardt, Sergiy Alpaev, Uwe Zdun, and all the other participants of the writer's workshop at VikingPLoP'2005, for their motivation, comments and suggestions for improvement.

References

- [1] Aguiar, A. (2003). A minimalist approach to framework documentation. PhD thesis, Faculdade de Engenharia da Universidade do Porto.
- [2] Hargis, G. (2004). Developing quality technical information. Prentice-Hall, 2nd edition.
- [3] Alexander, C., Ishikawa, S., and Silverstein, M. (1977). A Pattern Language. Oxford University Press.
- [4] Beck, K. and Gamma, E. (1997). JUnit homepage. Available from <http://www.junit.org>.
- [5] Eckstein, R., Loy, M., and Wood, D. (1998). Java Swing. O'Reilly & Associates, Inc.
- [6] Butler, G., Keller, R. K., and Mili, H. (2000). A framework for framework documentation. ACM Comput. Surv., 32(1es):15.
- [7] Butler, G. (1997). A reuse case perspective on documenting frameworks. Available from <http://www.cs.concordia.ca/faculty/gregb>.
- [8] Clark, M. (2003). JUnit: FAQ - frequently asked questions. Available from <http://junit.sourceforge.net/doc/faq/faq.htm>.

Patterns of Argument Passing

Uwe Zdun

Department of Information Systems
Vienna University of Economics, Austria
zdun@acm.org

Argument passing means passing values along with an invocation. Most programming languages provide positional arguments as their ordinary argument passing mechanism. Sometimes ordinary argument passing is not enough, for instance, because the number of arguments or their types differ from invocation to invocation, or optional arguments are needed, or the same arguments are passed through a chain of multiple receivers and must vary flexibly. These issues can be resolved using ordinary argument passing mechanisms, but the solutions are usually cumbersome. In many systems, such as programming languages, programming environments, frameworks, and middleware systems, advanced argument passing solutions are provided to better address these issues. In this paper we present four patterns applied in these advanced argument passing solutions: `VARIABLE ARGUMENT LISTS` allow an operation to receive arbitrary numbers of arguments, `OPTIONAL ARGUMENTS` let operations have arguments which can either be provided in an invocation or not, `NON-POSITIONAL ARGUMENTS` allow arguments to be passed in any order as name/value pairs, and `CONTEXT OBJECTS` are special types used for the purpose of argument passing.

Introduction

Argument passing

Argument passing is an integral part of all forms of invocations, for instance, performed in object-oriented and procedural systems. All systems that provide facilities for performing invocations thus must provide some way to pass arguments (also called parameters) to operations. As a first example for such systems one might think of programming language implementations, such as interpreters, compilers, and virtual machines. But argument passing is also relevant for any other kind of system that performs invocations on top of the facilities offered by a programming language, such as middleware systems, aspect-oriented composition frameworks, component frameworks, interactive shells of operating systems or programming languages, enterprise integration frameworks, and so forth.

Intended audience

In this paper we present some patterns that provide advanced argument passing solutions. These patterns are important for developers of the systems named above, when they want to provide some argument passing mechanism in a language or framework. In addition to that, the patterns are also relevant for developers using these systems, because in some situations the ordinary (i.e. positional) argument passing mechanisms offered by the language or framework do not cope well with a particular design problem. Then it is advisable to write a little argument passing framework on top of the language or framework that supports the respective pattern.

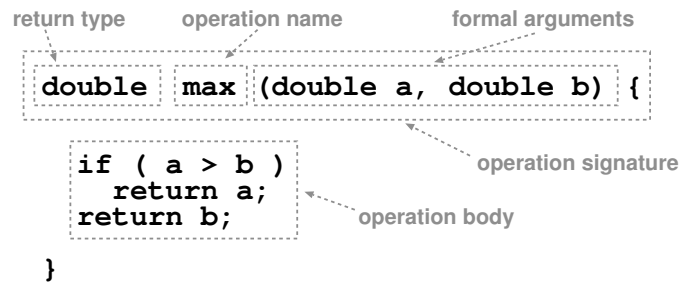


Figure 1: Operation definition

Terminology

To explain the patterns, we use the following terminology applicable to all kinds of languages and frameworks mentioned above: any kind of function, procedure, method, etc., be it local or remote, is called an *operation*. Each operation has an *operation signature*. The signature contains at least an *operation name* and a list of *arguments*. In typed environments the signature also contains types for each argument and a return type. An operation is “called” or “invoked” using an *invocation*.

In the text below, we sometimes refer to “ordinary arguments”. With this term we mean the typical, positional arguments offered by almost any procedural or object-oriented programming language, such as C, C++, Java, Tcl, etc.

The arguments in the signature are called *formal arguments* because they act as placeholders for argument values provided by the invocation. The concrete argument values provided by an invocation are called *actual arguments*. Each *actual argument* is mapped to the *formal argument* at the same position.

When the invocation takes place, each formal argument is filled with the value of the respective actual argument. This can be done by copying the value of the actual argument into the storage space of the operation (call-by-value), as opposed to providing only the address of the storage space of the actual argument to the operation (call-by-reference). Another scheme of argument passing is call-by-name, which refers to passing the (unevaluated) code of the argument expression to the operation, and this code is evaluated each time the argument is accessed in the operation. In dynamic languages call-by-name can be applied at runtime (examples are arguments evaluated using `eval` in Lisp or Tcl), or it can be performed statically by compilers or preprocessors (e.g. C macros). There are a number of other, less popular parameter passing schemes, such as call-by-value-return in Fortran, or copy-a-value-in and copy-it-back.

The terms are illustrated using a Java method and invocation as an example in Figures 1 and 2.

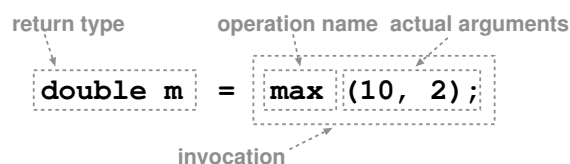


Figure 2: Operation invocation

Pattern Language Outline

The following patterns are presented in this paper:

- A **VARIABLE ARGUMENT LIST** provides an argument passing mechanism with a special syntax, which allows the client to invoke the operation using any number of arguments for this last argument. The actual arguments are put as a list into the last formal argument.
- **OPTIONAL ARGUMENTS** are an argument passing mechanism that uses a special syntax to denote that one of the formal arguments is optional. A default value is provided, which is used in case the client does not provide the **OPTIONAL ARGUMENT** in an invocation.
- **NON-POSITIONAL ARGUMENTS** are an argument passing mechanism that allows clients to provide arguments in an invocation as name/value pairs. Because each argument is named, the arguments can be provided in any order, and the argument passing mechanism automatically matches the correct actual arguments to the formal arguments.
- A **CONTEXT OBJECT** is a special object type that is used for argument passing. This object type is used as an argument of the operation (often it is the only argument), and the arguments to be passed to the operation are encapsulated in the **CONTEXT OBJECT** (e.g. as instance variables).

Related Patterns

There are a number of related patterns, documented elsewhere, that play an important role for the patterns described in this paper. We want to explain some of these patterns briefly:

- An **AUTOMATIC TYPE CONVERTER** [Zdu04b] converts types at runtime from one type to another. There are two main variants of the pattern: one-to-one converters between all supported types and converters utilizing a canonical format to/from which all supported types can be converted. An **AUTOMATIC TYPE CONVERTER** is primarily used by the patterns presented below for realizing type conversions.
- In a **REFLECTION** [BMR⁺96] architecture all structural and behavioral aspects of a system are stored into meta-objects and separated from the application logic components. The latter can query the former in order to get information about the system structure. In an argument passing architecture, **REFLECTION** is especially used to introspect ordinary operation signatures to perform a mapping between the arguments passed using the patterns and ordinary invocations.
- The **CONTEXT OBJECT** pattern is a general pattern for passing arguments using a special object type. Various special-purpose variants of this pattern have been described in the literature before: **INVOCATION CONTEXTS** [VKZ04] are **CONTEXT OBJECTS** used in distributed invocations; **MESSAGE CONTEXTS** [Zdu03, Zdu04a] are **CONTEXT OBJECTS** used in aspect-oriented composition frameworks and interceptor architectures; **ARGUMENTS OBJECT** [Nob97] is an object that contains all elements of an operation signature, for instance, as variables; **ANYTHING** [SR98] describes a generic data container; **ENCAPSULATE CONTEXTS** [Kel03] are **CONTEXT OBJECTS** used to encapsulate common data used throughout the system; **OPEN ARGUMENTS** [PL03] are **CONTEXT OBJECTS** used to support a dynamic set of arguments.
- A **PROPERTY LIST** [SR98] is a data structure that allows developers to associate names with arbitrary values and objects. This structure is needed to represent a simple list of

NON-POSITIONAL ARGUMENTS. The PROPERTY LIST pattern thus can be used to internally implement the NON-POSITIONAL ARGUMENTS pattern.

Figure 3 shows an overview of the patterns described in this paper and the relationships explained above. The patterns described in this paper are rendered in black, the related patterns in grey.

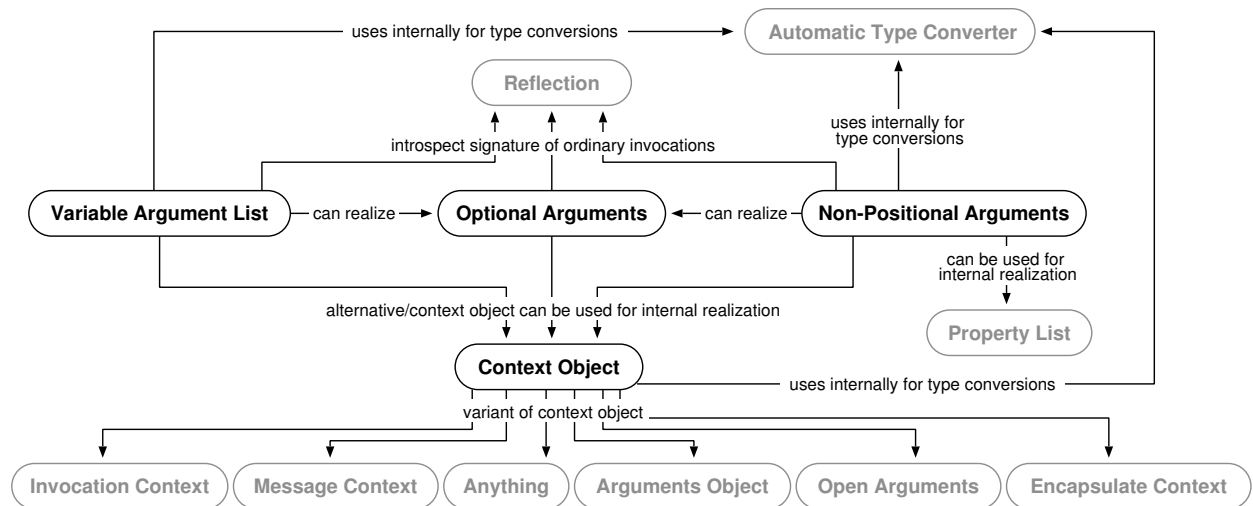


Figure 3: Pattern map: Most important relationships of the patterns

Variable Argument List

| | |
|-----------------|---|
| Context | You are writing operations to be invoked for instance in a programming language or in a distributed system. |
| Problem | A particular operation needs to receive a varying number of arguments, and you do not know in advance how many arguments will be received. You only know that the arguments to be received are all of the same type, and they can be treated in a uniform way. Ordinary operation signatures, however, cannot retrieve arbitrary numbers of arguments. Thus you have to apply tedious and error-prone workarounds for this situation, such as abusing polymorphism (e.g. overloading) or passing the arguments in a helper data structure. |
| Forces | <p>Consider you want to process a list of objects, but do not know in advance how many arguments are in the list. For small numbers of objects, you can use overloading to be able to invoke the operation with a varying number of arguments:</p> <pre>void processList(Object o1) { //... } void processList(Object o1, Object o2) { //... } void processList(Object o1, Object o2, Object o2) { //... }</pre> <p>Besides the problem that you have to write a huge number of unnecessary operations, you face the problem that this approach does not scale well for possibly larger numbers of arguments: consider you might receive lists with up to 1000 arguments. You would have to write 999 unnecessary operations.</p> <p>An alternative solution for this problem is to bundle the arguments in a collection data structure (such as a list or an array). But this solution is quite complex because in each such operation you have to process the arguments in the list, and for each invocation you have to fill the data structure before you can perform the invocation.</p> <p>A collection data structure requires the caller to put the appropriate arguments into the data structure, which make the caller more complex. Note that the overloading solution sketched above, in contrast, makes the callee more complex and error prone.</p> <p>Another problem of using a collection data structure is that two kinds of invocations exist in the system. Rather it would desirable that all invocations look the same.</p> |
| Solution | Provide a special syntax for the VARIABLE ARGUMENT LISTS that might be added as the last argument, or the only argument, to the argument lists of operations. Each argument in the VARIABLE ARGUMENT LIST is of the same type, which might be a generic type such as string or void pointer. The language implementation (e.g. the compiler or interpreter) or the framework implementation (e.g. the distributed object framework) provides a |

functionality to process the VARIABLE ARGUMENT LISTS. Thus, from the developers perspective, all invocations of VARIABLE ARGUMENT LIST operations look just like ordinary invocations, except that they vary in their length. Also, provide an API to make the arguments passed through a VARIABLE ARGUMENT LIST accessible from within the operation.

Discussion

The arguments of VARIABLE ARGUMENT LIST must be distinguishable in the invocation from other arguments. That's the reason why VARIABLE ARGUMENT LIST are usually realized as the last argument in the operation. An alternative is to delimit them in the invocation, for instance using a special character. But this would violate the goal that invocations with VARIABLE ARGUMENT LIST should look the same as ordinary invocations.

In principle it is also possible to have VARIABLE ARGUMENT LISTS be placed in the middle of ordinary arguments. This, however, is not advisable because in this case it is easy that bugs, such as wrong number of arguments, are not detected.

Similarly, a simple, working solution is to allow only for one VARIABLE ARGUMENT LIST per operation. In principle it is also possible to have more VARIABLE ARGUMENT LISTS in one operation, if the arguments can be distinguished by their type. Again, this might lead to bugs that are hard to find, for instance, when one of the argument types can be automatically converted to another one.

In type-safe environments, type-safety is an issue when using VARIABLE ARGUMENT LISTS. The typical solution is to let all arguments in the VARIABLE ARGUMENT LIST be of the same type. Otherwise, it would be necessary to define how to handle the different types and maybe delimit them, meaning that VARIABLE ARGUMENT LISTS would have a pretty different appearance in the signature than ordinary arguments (which is usually not wanted). If different types are needed in a VARIABLE ARGUMENT LIST, a super-type of these types can be used for defining the VARIABLE ARGUMENT LIST or, if this is not possible, a generic type, such as `string`, `void*`, or `Object`. A VARIABLE ARGUMENT LIST in an untyped environment is equivalent to using a generic type in typed environments.

In summary, in most cases it is advisable to allow for only one VARIABLE ARGUMENT LIST per operation signature and enforce that this VARIABLE ARGUMENT LIST is the last argument of the operation signature. All arguments of the VARIABLE ARGUMENT LIST are passed as the same type.

Note that we require a way to retrieve the arguments in the VARIABLE ARGUMENT LIST from within the operation. Here, VARIABLE ARGUMENT LIST arguments must be a bit different than ordinary arguments, because in the one operation signature element that represents the VARIABLE ARGUMENT LIST n arguments are hidden. Usually, an API or special syntax is provided, which provides a way to (a) retrieve the list of variable arguments (e.g. as a list data structure) and (b) find out how many variable arguments are passed through. Using this information, the VARIABLE ARGUMENT LIST can be processed using the operations of the list data structure.

Consequences

VARIABLE ARGUMENT LISTS solve a prevalent concern in writing generic and reusable operations. They are an elegant solution because they are applied automatically and do not look much different to ordinary invocations. Only the operation implementation must be written in a slightly different style.

If VARIABLE ARGUMENT LIST are not language-supported or framework-supported, some effort

to provide an implementation is required. A simple emulation (e.g. using a collection data structure) is not much work, but one also has to write a little program generator to convert the invocation to the VARIABLE ARGUMENT LIST format.

A much simpler, but slower solution is to use strings (or other generic types) for argument passing and an AUTOMATIC TYPE CONVERTER to convert the invocations back and forth. An invocation:

```
processList(3, 1, 2, 3);
```

would then become:

```
processList("3, 1, 2, 3");
```

This is not very desirable in the context of many programming language because again we would end up with two different styles of invocations. Moreover, the solution is rather slow because back and forth conversion to strings is required. But there are situations where this solution is highly applicable. For instance, in string-based programming languages (such as most scripting languages) there is no difference in the invocation styles. Or, in middleware implementations the invocations are sent as a byte-array over the wire anyway. Thus, again, there is no difference to all other invocations.

VARIABLE ARGUMENT LISTS can make overloading resolution more complex, ambiguous, or, in some situations, even impossible. Thus usually it is advisable not to use overloading for an argument that is realized as a VARIABLE ARGUMENT LIST, or at least introduce an unambiguous rule for overloading VARIABLE ARGUMENT LISTS.

Type-safety might be compromised, depending on the VARIABLE ARGUMENT LISTS implementation (compare the C/C++ and Java known uses below).

Known Uses Some known uses of the pattern are:

- In C and C++ VARIABLE ARGUMENT LISTS are language-supported. In place of the last argument you should place an ellipsis (“...”). C and C++ provide an API to process the VARIABLE ARGUMENT LIST (starting with `va_`) as in the following example:

```
void processList(int n, ...) {
    va_list ap;
    va_start(ap, n);
    printf("count = %d: ", n);
    while (n-- > 0) {
        int i = va_arg(ap, int);
        printf("%d ", i);
    }
    printf("\n");
    va_end(ap);
}
```

This operation can be used like any other operation:

```
processList(1, 1);  
processList(3, 1, 2, 3);
```

Please note that functions that take a variable number of arguments (“varargs”) are generally discouraged in C/C++ style guides (see e.g. [CEK⁺00]) because there is no truly portable way to do varargs in C/C++. If varargs are needed, it is advisable to use the library macros for declaring functions with `VARIANT ARGUMENT LISTS`.

- In the scripting language Tcl (similar to other scripting languages) a special argument `args` can be provided to an operation as the last argument. In this case, all of the actual arguments starting at the one that would be assigned to `args` are combined into a list. This combined value is assigned to the local variable `args`, which is an ordinary Tcl list.
- Leela [Zdu04c] is a Web services framework that uses `VARIABLE ARGUMENT LISTS` for generic argument passing between Web services. A Leela service is bound to a SOAP endpoint, and this endpoint offers a string-based interface. This interface is mapped to the Web service operation using `REFLECTION` (see also the pattern `INTROSPECTION OPTIONS` [Zdu03]).
- In Java, starting with version 5.0, Var-Args are provided. Java’s solution is similar to the C solution. A major difference is that it is type-safe. For instance, we can specify an operation for processing a `String` list:

```
public static void processList(String... args) {  
    for (String a : args) {  
        System.out.println(a + " ");  
    }  
}
```

Java’s Var-Args can receive any argument type by using a more generic type, such as Java’s `Object`, for instance.

- Many programming languages provide a `VARIABLE ARGUMENT LIST` mechanism to receive arguments from the command line. This design is due to the argument passing interface of command shells, especially UNIX shells, which led to C/C++’s “`int main(int argc, char *argv[])`” interface to programs. Most contemporary programming languages support a similar interface, for instance, in Java, command line arguments are mapped to a special `String` array that is the argument of the operation “`static void main(String[] args)`”.

Optional Arguments

Context You are writing operations to be invoked for instance in a programming language or in a distributed system.

Problem Sometimes one operation can be defined for a varying number of arguments. This situation can in principle be solved using VARIABLE ARGUMENT LISTS. But consider the situation is slightly different to the problem solved by VARIABLE ARGUMENT LISTS: you know the possible arguments in advance, and the number of arguments is manageable. The arguments might be of different types (or kinds in untyped languages); that is, they cannot or should not be treated uniformly.

Forces Constructors are operations that should be able to receive differing numbers of arguments because different clients want to configure different values. All unspecified values should be filled with default values. Consider the following Java code as an example:

```
class Person {
    Name name;
    Address homeAddress;
    Address workAddress;
    ...
    Person(Name _name, Address _homeAddress, Address _workAddress) {
        name = _name;
        homeAddress = _homeAddress;
        workAddress = _workAddress;
    }
    Person(Name _name) {
        this(_name, null, null);
    }
    ...
}
```

In this example, the variables `homeAddress` and `workAddress` are optional and have `null` as a default value. To realize this concern, the `Person` constructor needs to be defined twice, just to pass the default values to the operation that really does the work. Usually, there are more such constructors, and we need to provide similar forwarders in subclasses as well. For instance, to provide the option that the work address is optional, another constructor has to be added.

The solution in the example uses Java's method overloading which works by realizing a concern using multiple operations with different signatures and possibly chaining them with invocations among each other (as in the example above). This is a heavy-weight solution for the simple problem of realizing an optional default value. For each optional argument, and each possible combination of optional arguments, we need to provide one additional operation. The result is a lot of unnecessary lines of code, reducing the readability of the program.

Another problem is that we cannot provide all possible combinations of arguments because Java's overloading mechanism selects methods only on basis of the signature of the operation.

Sometimes the types of arguments conflict, for instance, in the above example we cannot provide default values for both `homeAddress` and `workAddress`, because the two operation signatures:

```
Person(Name _name, Address _homeAddress);
```

and:

```
Person(Name _name, Address _workAddress);
```

are conflicting. The compiler cannot distinguish between them because they have the same types in their signature.

Note that it is not elegant to use `VARIABLE ARGUMENT LISTS` in this and similar examples. The arguments of constructors are named and typed. With a `VARIABLE ARGUMENT LIST` you would have to pass all the arguments using a generic type, and then obtain the individual arguments using their position in the `VARIABLE ARGUMENT LIST`. This approach makes it hard to handle changes in argument lists.

Solution

Introduce a special syntax for operation signatures to mark some arguments as `OPTIONAL ARGUMENTS`. For each `OPTIONAL ARGUMENT` provide a default value. Provide a language-support or framework-support for selecting or passing arguments to operations who have `OPTIONAL ARGUMENTS`. It is important that there are no syntactic ambiguities which actual argument belongs to which formal argument.

Discussion

`OPTIONAL ARGUMENTS` require default values because without them it would be undefined how to handle an invocation in which an `OPTIONAL ARGUMENT` is omitted. Default values can be provided in different fashions:

- They can simply be provided in the operation signature, where the optional argument is defined.
- They can be looked up at runtime and added to the actual invocation by the language implementation or framework. To use this variant is advisable if the default values should be modifiable after the program has been compiled or started. For example, the default values can be defined in a configuration file or an external repository.
- They can be defined programmatically: some code handles the situation when an `OPTIONAL ARGUMENT` is not provided by an invocation.
- They can be implicitly defined, for instance, by some convention. For example, if there is an “empty” value or system-wide default value, this value can be chosen by the language or framework if no value for the `OPTIONAL ARGUMENT` is given. If there is an old value (e.g. from previous invocations), also the old value can be used as the default value.

The `OPTIONAL ARGUMENTS` pattern is often combined with other patterns. A `VARIABLE ARGUMENT LIST` is implicitly an `OPTIONAL ARGUMENT` that defaults to “empty”. When the `OPTIONAL`

ARGUMENTS pattern is combined with VARIABLE ARGUMENT LISTS, it is important that the order of the two patterns in argument lists is clearly defined, so that there are no ambiguities. NON-POSITIONAL ARGUMENTS are often OPTIONAL ARGUMENTS, meaning that an omitted NON-POSITIONAL ARGUMENT is treated as being optional. A CONTEXT OBJECT implementation might also provide support for OPTIONAL ARGUMENTS.

Consequences OPTIONAL ARGUMENTS provide a look and feel similar to ordinary invocations. They can be applied automatically. In the operation signature, a special syntax is required for defining an argument as being optional and for defining or retrieving the default value. Usually invocation and operation bodies do not have to be adapted to be used with OPTIONAL ARGUMENTS.

In compiled languages, the default values cannot be changed at runtime. For a change of a default value a recompilation is necessary.

Known Uses Some known uses of the pattern are:

- In a C++ operation definition, the trailing formal arguments can have a default value (denoted using “=”). The default value is usually a constant. An example is the following operation signature, which receives two `int` arguments, the second one being optional with the default value 5:

```
void foo(int i, int j = 5);
```

- Many scripting languages support OPTIONAL ARGUMENT for operations. In Tcl [Ous94], for instance, OPTIONAL ARGUMENTS can be defined as pairs of argument name and default value. OPTIONAL ARGUMENTS need not be specified in an operation invocation. However, there must be enough actual arguments for all the formal arguments are not OPTIONAL ARGUMENTS, and there must not be any extra actual arguments. For instance, the following `log` procedure has an optional argument `out_channel`, which is per default configured to the standard output:

```
proc log {log_msg {out_channel stdout}} {
    ...
}
```

If the last formal argument has the name `args`, it is treated as a VARIABLE ARGUMENT LIST. In this case, all of the actual arguments starting at the one that would be assigned to `args` are passed as a VARIABLE ARGUMENT LIST. That is, it is not possible that there are ambiguities between the OPTIONAL ARGUMENTS and the arguments for the VARIABLE ARGUMENT LIST.

- In the GUI toolkit TK [Ous94], constructors of widgets provide access to the widget options, such as background, width, colors, texts, etc., as OPTIONAL ARGUMENTS, which represent either empty values (like an empty text) or values that are often chosen (e.g. the color of the surrounding widget). A TK widget can therefore be initiated with only a very few lines of code because only those options that differ from the defaults must be provided. For example, the following code instantiates a button widget and configures it with the label “Hello” and a callback command that prints “Hello, World!” to the standard output:

```
button .hello -text Hello -command {puts stdout "Hello, World!"}
```

The operation `configure` allows TK programs to access the widget options. Thus `configure` is an operation with NON-POSITIONAL ARGUMENTS in which each argument is an OPTIONAL ARGUMENT and its value defaults to the current setting of the widget. This way only those options of a widget to be changed must be specified in a `configure` invocation. For example, we can configure a red background for the button widget:

```
.hello configure -background red
```

- The GNU Program Argument Syntax Conventions [GNU05] recommend guidelines for command line argument passing. To specify an argument as an OPTIONAL ARGUMENT, a so-called long option, it is written as `--name=value`. This syntax enables a long option to accept an argument that is itself optional. Many UNIX tools and `configure` scripts follow this convention. For example, many `configure` scripts offer a number of options, such as `--prefix` and `--exec-prefix` (those are used for configuring the installation path). These arguments can optionally be appended to `configure` invocations:

```
./configure --prefix=/usr --exec-prefix=/usr
```

If the options are omitted, they have a default value, such as `/usr/local`.

Non-Positional Arguments

| | |
|----------------|---|
| Alias | <i>Named Actual Arguments, Named Parameters</i> |
| Context | You perform invocations, for instance, in a programming language or in a distributed system. |
| Problem | You need to pass arguments along with an invocation. You are faced with one of the following two problems: firstly, at the time when you design the operation which receives the arguments, you do not know how many arguments need to be passed. Different invocations of the operation require a different number of arguments. Secondly, some invocations require a large number of arguments. These invocations are hard to read because one must remember the meaning of each argument in order to understand the meaning of the whole invocation. Matters become even worse when both problems occur together, i.e. there is a large number of arguments and some of them are OPTIONAL ARGUMENTS. |
| Forces | <p>Consider the following invocation:</p> <pre>ship.move(12, 23, 40);</pre> <p>This very simple invocation can only be understood with the specification of the operation <code>move</code> in the back of the mind. Developers usually have to deal with a lot of such operations at the same time, and thus it is impossible to remember the meaning of all arguments of all operations. To understand a program, one has to continuously look at the operation specifications.</p> <p>This example illustrates the problem of readability that many ordinary invocations might have, once a certain number of arguments is exceeded.</p> <p>Another important problem is that of extensibility. Programming languages like Java offer overloading to extend operations, such as <code>move</code> in the example above. This way we can overload an operation, and provide multiple realization of the operation. For instance, we can provide one <code>move</code> implementation that receives three integers as arguments (as above), and one <code>move</code> implementation that receives an object of type <code>ThrustVector</code>. But as overloading depends on the type system, we can only define overloaded operations with a different number of arguments and/or different types of arguments. We are not able to provide a second realization of <code>move</code> that also receives three integers.</p> <p>From time to time, we make little semantic mistakes when invoking such operations with ordinary operations. For instance, we might twist two arguments in an invocation. Most of the time the compiler finds such mistakes because different types are needed, or our application code complains because the values provided are not meaningful. But sometimes such mistakes stay undetected because the twisted arguments are of the same type and the provided values are meaningful. For instance, in the above example, a little mistake like:</p> <pre>ship.move(40, 12, 23);</pre> <p>might stay undetected. Such mistakes might produce hard to find bugs.</p> |

The pattern VALUE OBJECT [Hen00a, Hen00b] provides a possible solution to this problem. A VALUE OBJECT is realized by a lightweight class that has value semantics, and is typically, but not always, immutable. If we make all values of the example operation VALUE OBJECTS, we could write the invocation as follows:

```
ship.move(Left(12), Right(23), Thrust(40));
```

Together with overloading, VALUE OBJECTS provide a well defined interface for ship movements, which is typed and supports multiple combinations of arguments. Using VALUE OBJECTS, however, requires us to change the operation signature. This might not be possible for third-party code, and thus we need to write a VALUE OBJECT wrapper for each extended third-party operation. The VALUE OBJECT solution does only work well for small numbers of possible arguments, because operation overloading means writing additional operations for each possible combination of arguments. Also, types can only be used to distinguish arguments as long as they are different (consider two Thrust arguments, for instance).

Solution **Provide an interface to pass NON-POSITIONAL ARGUMENTS along with an invocation. Each NON-POSITIONAL ARGUMENT consists of an argument name plus an argument value. The argument name can be matched to the respective arguments of the operation. Thus it is no longer necessary to provide the arguments in a strict order, but any order is applicable. Usually NON-POSITIONAL ARGUMENTS are OPTIONAL ARGUMENTS.**

Discussion Non-positional arguments provide each argument as a name/value pair. We need some syntax to distinguish names from values. For instance, we can start each argument name with a dash “-”. Then we can write the above invocation example in a form like:

```
ship.move -left 12 -right 23 -thrust 40;
```

Of course, this form does not conform to the syntax of ordinary arguments of the programming language anymore. Thus we must implement some support for dealing with NON-POSITIONAL ARGUMENT invocations:

- We can provide program generator (preprocessor) which parses the program text, finds the NON-POSITIONAL ARGUMENT invocations, and checks that they conform to the arguments required by the operation. The preprocessor substitutes the NON-POSITIONAL ARGUMENT invocations with ordinary argument invocations.
- A more simple way to realize NON-POSITIONAL ARGUMENT invocations is to use a string-based syntax. That is, all operations receiving NON-POSITIONAL ARGUMENTS receive only one string as an argument in which the NON-POSITIONAL ARGUMENTS are encoded, such as:

```
ship.move("-left 12 -right 23 -thrust 40");
```

This syntax is simple, but we need to parse the string, type-convert the arguments (using an AUTOMATIC TYPE CONVERTER), and map them to the ordinary arguments. Runtime string parsing is slower than invocations injected by a program generator.

- We can provide a special kind of CONTEXT OBJECT which holds NON-POSITIONAL ARGUMENTS. That is, the CONTEXT OBJECT must be able to store a dynamic length table or list of name/value pairs, and the values must be of a generic type. Thus type conversion might get more simple than in the string-based variant, and the solution is more efficient than string parsing. The CONTEXT OBJECT, however, requires a different syntax than ordinary invocations. Thus in most cases CONTEXT OBJECTS should rather be used internally to implement NON-POSITIONAL ARGUMENTS and stay hidden from the developer.
- Finally, it is also possible that a programming language provides support for NON-POSITIONAL ARGUMENTS. Alternatively, some programming languages can be extended with support for NON-POSITIONAL ARGUMENTS. All other variants, described before, required a NON-POSITIONAL ARGUMENT framework – on top of a positional arguments implementation – for supporting the pattern.

When NON-POSITIONAL ARGUMENTS are implemented on top of positional arguments, we need some converter that is invoked between the invocation and the execution of the operation. The converter must transform the NON-POSITIONAL ARGUMENTS into positional arguments. That is, it needs to map the named actual arguments to names of the formal, positional arguments. To do so, the converter must know about the name and type of each positional argument, so that it can map the NON-POSITIONAL ARGUMENTS in the correct way. This knowledge can either be provided to the converter (e.g. at compile time or load time), or REFLECTION can be used by the converter to acquire the information at runtime.

The converter is also responsible for applying type conversions if they are necessary (e.g. using the AUTOMATIC TYPE CONVERTER pattern), and must raise exceptions in case of type violations. Note that the converter must also check for overloaded operations and other kinds of operation polymorphism, if supported by the programming language, and decide on basis of the provided NON-POSITIONAL ARGUMENTS which operation implementation needs to be invoked.

In the “programming language support” variant, the language implementation (compiler, interpreter, virtual machine, etc.) realizes the converter. In the “program generator/preprocessor” variant, the generator generates the conversion code. In the other variants, “string-based syntax” and “CONTEXT OBJECT”, the developer might have to manually trigger the converter. For instance, the first lines in the invoked operation might query the arguments, or the invoking code must trigger conversion, such as:

```
system.invokeWithNonPosArgs("ship.move -left 12 -right 23 -thrust 40");
```

The converter internally needs to hold and perhaps pass around the name-value pairs. The pattern PROPERTY LIST [SR98] provides a data structure that allows names to be associated with arbitrary other values or objects. It is thus ideally suited as an implementation technique to internally represent the NON-POSITIONAL ARGUMENTS before they are mapped to the invocation. A hash table data structure is an (efficient) means to implement the PROPERTY LIST data structure (this solution is used by many scripting languages such as Perl or Tcl).

The pattern ANYTHING [SR98] is an alternative for PROPERTY LIST, where PROPERTY LIST is not sufficient. It is a generic data container for one (primitive) value of any kind or an associative

array of these values. The pattern thus can also be used to implement NON-POSITIONAL ARGUMENTS. Finally, the CONTEXT OBJECT pattern can be used (only internally) to hold and pass around the NON-POSITIONAL ARGUMENTS.

All NON-POSITIONAL ARGUMENTS for which we can assume a default value are usually OPTIONAL ARGUMENTS. For instance, in the example we might want to move the ship without changing the course, or just change the course, or just change the course in one direction. Using NON-POSITIONAL ARGUMENTS with OPTIONAL ARGUMENTS we can assume the old value as default for all values not specified and then do invocations like:

```
ship.move -thrust 30;  
...  
ship.move -left 15 -right 23;  
...  
ship.move -thrust 10 -right 15;
```

Consequences The biggest advantage of NON-POSITIONAL ARGUMENTS is that they enhance readability and understandability of long argument lists. They can also be used on top of positional arguments, meaning that they can be used to enhance the documentation of invocations in a framework, without having to change the positional signature of a (given) target operation (see the SOAP example below for an example of distributed invocations).

Extensibility is also enhanced because overloading extensions of an operation can be based on the selection with an identifier (the argument name) and not only using the type of the argument. The combination with the OPTIONAL ARGUMENT pattern supports the extensibility of NON-POSITIONAL ARGUMENTS and enhances the changeability, when extensions are introduced: developers can define default values for extensions to a given operation. That is, invocations using the old version without the extension are still valid and do not need to be changed.

NON-POSITIONAL ARGUMENTS reduce the risk of mistakes during argument passing because the developer has to name the argument to which a value belongs.

A drawback of NON-POSITIONAL ARGUMENTS is that they are more verbose than positional arguments. That is, a program with NON-POSITIONAL ARGUMENTS has more lines of code. This drawback does not necessarily occur, when OPTIONAL ARGUMENTS are used together with NON-POSITIONAL ARGUMENTS and default values can be used. Consider an operation with 20 options, and you want to change only one of them. NON-POSITIONAL ARGUMENTS with OPTIONAL ARGUMENTS allow you to specify only the name and the value of that one argument: the result is an invocation with two extra words for arguments. An operation with positional arguments that changes all 20 options would require 18 words more than that (plus invocations to query the old values of the arguments not to be changed).

If the system or language does not yet support NON-POSITIONAL ARGUMENTS and you want to introduce them, depending on your solution, different changes to the system need to be made. For instance, you might have to introduce a converter, and the converter introduces a slight performance decrease. In some solutions, discussed above, the signature or implementation of the operations must be changed. Other solutions (like the language-based or generator-based variants) require more efforts for implementing them. The efforts and drawbacks of the individual solutions need to be compared to the benefits of NON-POSITIONAL ARGUMENTS.

If positional arguments are supported as well, two styles of invocation are present. The syntax

for both variants should be distinctive, so that developers can see at first glance, which kind of invocation is used or required by an operation.

Known Uses Some known uses of the pattern are:

- The SOAP protocol [BEK⁺00], used for Web services communication, uses NON-POSITIONAL ARGUMENTS. For instance, an invocation of an operation `GetPrice` with one argument `Item` might look as follows:

```
<soap:Body>
  <m:GetPrice xmlns:m="http://www.w3schools.com/prices">
    <m:Item>Apples</m:Item>
  </m:GetPrice>
</soap:Body>
```

In SOAP the response message also contains NON-POSITIONAL ARGUMENTS.

Web services frameworks implemented in languages that do not support NON-POSITIONAL ARGUMENTS must map SOAP's NON-POSITIONAL ARGUMENTS to the positional arguments of the programming language. For instance, the Web services framework Apache Axis [Apa04] contains an AUTOMATIC TYPE CONVERTER which maps the NON-POSITIONAL ARGUMENTS delivered in SOAP messages to Java invocations, and vice versa.

- The GUI toolkit TK provides NON-POSITIONAL ARGUMENTS for configuring the TK widgets. Each widget has a huge number of options. Most of the time it is enough for a Widget instance to configure only a few of these options. For both readability and extensibility reasons, it is not a good choice to perform the configuration of the widget options using operations and operation overloading, as used by many other GUI toolkits. For instance, a button widget with NON-POSITIONAL ARGUMENTS can be created like this:

```
button .b -text Hello! -command {puts hello}
```

We can also send any of the possible widget arguments as NON-POSITIONAL ARGUMENTS to the widget for reconfiguration. For instance, we can change the font like this:

```
.b configure -font {Times 16}
```

The advantage of NON-POSITIONAL ARGUMENTS are that we can choose any of the 32 widget options in TK 8.4 for a button in any order and that we can directly see which option is configured in which way. TK constantly evolves. For instance, in TK 8.0 the button widget had only 28 options. Nevertheless TK 8.0 scripts usually work without changes, compatibility operations, or other measures, because the NON-POSITIONAL ARGUMENTS are combined with OPTIONAL ARGUMENTS.

- OpenACS [GA99] is a toolkit for building scalable, community-oriented Web applications on top of a Web server. It uses the Tcl scripting language as a means for developers to add user-defined operations and call them from Web pages (or Web page

templates). One means to support flexible operations are so-called `ad_proc` operations. These operations can be declared with regular positional arguments, or with NON-POSITIONAL ARGUMENTS. In addition, when NON-POSITIONAL ARGUMENTS are used, it is possible to specify which ones are required, optional, and boolean. Optional arguments require a default value. They are an implementation of the OPTIONAL ARGUMENTS pattern. An example is:

```
ad_proc -public auth::authenticate {  
    {-username:required}  
    {-domain ""}  
    {-password:required}  
} {...} {...}
```

In this operation signature, the arguments `username` and `password` are required, the `domain` argument is an OPTIONAL ARGUMENT, which defaults to an empty string.

- XOTcl [NZ00] is an object-oriented Tcl variant which supports NON-POSITIONAL ARGUMENTS for all its operations. Its model is similar to that of the OpenACS framework.

Context Object

| | |
|-------------------|--|
| Context | You are invoking operations, for instance, in a programming language or distributed object system. |
| Problem | You want to deliver complex or varying information to an operation. For instance, there is a huge number of arguments, the number of arguments varies from invocation to invocation, or there are OPTIONAL ARGUMENTS. So ordinary, positional arguments are not really working well here. In addition to passing the information to the operation, you need to process the information in some way. For instance, you might have to transform them into a different format (e.g. marshal them to transport them over a network). Or the same information is passed through multiple operations and each one can add or remove some information. The arguments might be of different types (or kinds) and thus cannot be treated uniformly. So the patterns VARIABLE ARGUMENT LIST or NON-POSITIONAL ARGUMENTS do not resolve all concerns either. |
| Forces | <p>Consider information that is passed through multiple operations, and each operation can add or remove arbitrary information. For instance, this situation is typical for realizations of the patterns CHAIN OF RESPONSIBILITY [GHJV94], INVOCATION INTERCEPTOR [VKZ04], and PIPES AND FILTERS [BMR⁺96, SG96]. Using ordinary, positional arguments is cumbersome here because each operation would have to know the signature of its successor to be able to invoke it. Thus the modifiability of this architecture would be limited: the operations could not be assembled in arbitrary order.</p> <p>A VARIABLE ARGUMENT LIST could help to avoid this problem because all operations would just receive the VARIABLE ARGUMENT LIST and thus have the same signature. The operations could be assembled in any order. But, as a drawback, each operation would have to process the VARIABLE ARGUMENT LIST before the arguments could be accessed or changed. This means a slight performance overhead. Also, it should be possible to reuse the code for processing the list in different operations because likely most of them will process the list in more or less the same way – which is not supported directly by the VARIABLE ARGUMENT LIST pattern. VARIABLE ARGUMENT LIST only support arguments of the same kind. If there are different types, for instance, conversion to and from a generic format would be necessary.</p> <p>If just a variable number of named arguments is needed, NON-POSITIONAL ARGUMENTS might resolve the problem. If the arguments or the processing requirements are more complex, however, this won't work well either, because NON-POSITIONAL ARGUMENTS do not support complex processing instructions.</p> |
| Solution | Pass the arguments in a special object type, a CONTEXT OBJECT. This object provides all arguments as instance variables. The class of the object defines the structure of the CONTEXT OBJECT and the operations required to process the arguments. Using ordinary inheritance, more special CONTEXT OBJECTS can be derived. |
| Discussion | <p>A CONTEXT OBJECT must be instantiated and filled with values (e.g. with the actual arguments to be passed to an operation). A typical example looks as follows:</p> <pre>Context c = new Context();</pre> |

```
c.setValue("left", new Integer(12));  
c.setValue("right", new Integer(23));  
c.setValue("thrust", new Integer(40));  
o1.invokeOperation(c);
```

In the operation receiving the invocation the arguments must be accessed via the `CONTEXT OBJECT`'s API. For instance, an access to a value might look as follows:

```
Integer left = (Integer) c.getValue("left");
```

The API in this example uses key/value-pairs. This, however, is just an example, `CONTEXT OBJECTS` can use any kind of data structure. For instance, the same example could be realized using a special ship `CONTEXT OBJECT` that receives the values as instance variables, such as:

```
ShipContext sc = new ShipContext();  
sc.left = 12;  
sc.right = 23;  
sc.thrust = 40;  
o1.invokeOperation(sc);
```

A `CONTEXT OBJECT` is an alternative to the patterns `VARIABLE ARGUMENT LIST`, `OPTIONAL ARGUMENTS`, and `NON-POSITIONAL ARGUMENTS`. `CONTEXT OBJECT` is more generic than these patterns. This is because each of the other pattern's solutions can be realized using a `CONTEXT OBJECT`. However, in the concrete solutions applied by these patterns, the patterns provide more support than a solution using a generic `CONTEXT OBJECT`. In particular, the patterns usually allow for invocations and argument access that looks no different to ordinary invocations and argument access.

There are some style guides that advise the use of `CONTEXT OBJECTS`. For instance, an "old" C programming guide says: "if a function takes more than four parameters, pack them together in a struct and pass the struct instead". `CONTEXT OBJECTS` can be seen as the object-oriented successor of this guideline. In general, however, `CONTEXT OBJECTS` are especially used in infrastructure software. That is, they are often not visible to the developer, but only used internally. Examples are:

- Implementations of the patterns `VARIABLE ARGUMENT LIST`, `OPTIONAL ARGUMENTS`, and `NON-POSITIONAL ARGUMENTS` in interpreters, compilers, or program generators can pass the arguments within their implementation using `CONTEXT OBJECTS`.
- Distributed object systems need to pass the distributed invocation through the layers of the distributed object system using a `CONTEXT OBJECT`. At the client side, an invocation gets step-by-step transformed into a byte-array to be sent over the wire. At the server side, the invocation of the remote object is created step-by-step from the incoming message. Again the invocation needs to be passed through multiple entities. Besides invocation information, extra context information must be transmitted, such as security information (like passwords) or transaction contexts. This special variant of the `CONTEXT OBJECT` pattern for distributed object frameworks is called `INVOCATION CONTEXT` and is described in the book *Remoting Patterns* [VKZ04].

- Aspect-oriented software composition framework need to intercept invocations and indirect them to aspect implementations. This is most often done with CONTEXT OBJECTS (see [Zdu04a]). The pattern language in [Zdu03] describes a pattern for such MESSAGE CONTEXTS¹. The pattern is a special variant of CONTEXT OBJECTS.

In all three examples the CONTEXT OBJECTS are hidden from developers and are used inside of a framework used by the developer. An AUTOMATIC TYPE CONVERTER is usually applied by the CONTEXT OBJECT implementation to transparently convert generic types, used for instance in a distributed message, to the specific types defined by the user operation, so that the use does not have to care for type conversion.

Besides the two variants of CONTEXT OBJECTS mentioned above, INVOCATION CONTEXT [VKZ04] and MESSAGE CONTEXT [Zdu03, Zdu04a], there are more CONTEXT OBJECT variants documented in the pattern literature:

- ARGUMENTS OBJECT [Nob97] is an object that contains all elements of an operation signature, for instance, as variables. The ARGUMENTS OBJECT is passed to the operations with that signature instead of the arguments. The pattern is used in object-oriented languages as well as in procedural languages (e.g. a C struct can be used to encapsulate argument variables). ARGUMENTS OBJECT is a very simple variant to realize a CONTEXT OBJECT. It is advisable to use this variant, if fixed operation signatures should be simplified (or unified).
- ANYTHING [SR98] is a generic data structure that can hold any predefined primitive type, as well as associative arrays of the primitive types. Using these associative arrays, complex CONTEXT OBJECTS can be built (the arrays can contain arrays). Note that the CONTEXT OBJECT implementation in the ANYTHING pattern is scattered among multiple implementation objects. It is advisable to use this variant, if a generic data container that can be packed with arbitrary fields and values to be passed along a call chain (e.g. as in the patterns CHAIN OF RESPONSIBILITY, INVOCATION INTERCEPTOR, and PIPES AND FILTERS) is needed.
- ENCAPSULATE CONTEXT [Kel03] is a CONTEXT OBJECT that encapsulates common data used throughout the system. This pattern is used to pass the execution context for a component or a number of components as an object. The execution context can, for instance, contain external configuration data. Thus the ENCAPSULATE CONTEXT pattern describes one particular use case of the CONTEXT OBJECT pattern. Henney presents a pattern language for realizing ENCAPSULATE CONTEXT, consisting of four patterns: ENCAPSULATED CONTEXT OBJECT, DECOUPLED CONTEXT INTERFACE, ROLE-PARTITIONED CONTEXT, and ROLE-SPECIFIC CONTEXT OBJECT (see [Hen05]). These patterns are generally useful to implement CONTEXT OBJECTS.
- OPEN ARGUMENTS [PL03] are CONTEXT OBJECTS that support a dynamic set of arguments.

CONTEXT OBJECT provides a generalization of these individual patterns.

Consequences The CONTEXT OBJECT encapsulates the arguments of an operation in an object and makes them

¹In [Zdu03, Zdu04a] this pattern is called INVOCATION CONTEXT. To avoid confusion with the same-named pattern from the book *Remoting Patterns*, we henceforth use the pattern name MESSAGE CONTEXT.

exchangeable. If a number of operations are invoked using the same CONTEXT OBJECT, data copying can be avoided: all consecutive operations work using the same CONTEXT OBJECT and pass it on to the next operation after they have finished their work. The CONTEXT OBJECT couples the data structures (arguments) and the operations that are needed to process these arguments. CONTEXT OBJECTS are extensible using ordinary object-oriented inheritance.

The downside of using CONTEXT OBJECTS is that invocations do not look like ordinary invocations, but much more code for instantiating and filling the CONTEXT OBJECTS is needed. This reduces the readability of the code. Thus CONTEXT OBJECTS are not transparent to the developer using them. Also, the operation receiving the CONTEXT OBJECTS is different to an operation using ordinary arguments. For these reasons, CONTEXT OBJECTS are often used in infrastructure software, where they are hidden from the developer.

Note that there are some situations, where a global, well known space is a simple alternative to CONTEXT OBJECTS, which avoids passing the CONTEXT OBJECTS through the whole application (an example is the environment provided to CGI programs by a web server). This alternative can easily be abused. Likewise, a danger of using CONTEXT OBJECTS is that they can be abused for tasks that are similar to those of SINGLETONS [GHJV94]. They should only be used for modular chains of invocations. A CONTEXT OBJECT that references all elements of a system and is used like a global data structure is dangerous because it strongly couples different architectural elements, meaning that the individual architectural elements cannot be understood, loaded, or tested on their own anymore.

Known Uses Some known uses of the pattern are:

- An aspect-oriented composition framework intercepts specific events in the control flow, called joinpoints. The aspect is applied to these joinpoints. Thus the operation that applies the aspect must be informed about the details of the joinpoint. Many aspect-oriented composition framework use CONTEXT OBJECTS to convey this information. For instance, AspectJ [KHH⁺01] realizes its joinpoints using the JoinPoint interface. From an aspect the current joinpoint can be accessed using the variable `thisJoinPoint`. The aspect framework automatically instantiates the JoinPoint instances and fills it with values. For instance, the following instruction in an AspectJ advice prints the name of the signature of the currently intercepted joinpoint:

```
System.err.println(thisJoinPoint.getSignature().getName());
```

- In the Web services framework Apache Axis [Apa04], when a client performs an invocation or when the remote object sends a result, a CONTEXT OBJECT, called the `MessageContext`, is created before a Web services message is processed. Both on client and server side, each message gets processed by multiple handlers, which realize the different message processing tasks, such as marshaling, security handling, logging, transaction handling, sending, invoking, etc. Using the `MessageContext` different handlers can retrieve the data of the message and can potentially manipulate it.
- In CORBA the Service Context is used as a CONTEXT OBJECT which can contain any value including binary data. Service context information can be modified via CORBA's Portable Interceptors.

- In .NET `CallContexts` are used as `CONTEXT OBJECT`. They are used to transport information from a client to a remote object (and back) that is not part of the invocation data. Examples include security credentials, transaction information, or session IDs. The data is an associative array that contains name/value pairs:

```
CallContext.setData("itemName", someData);
```

- `CONTEXT OBJECTS` are often used when objects are simulated on top of procedural APIs. The first argument is a `CONTEXT OBJECT` which bundles all the data about the current object. For instance, the Redland API [Bec04] simulates objects using this scheme. Each Redland class has a constructor. For instance, the class `librdf_model` can be created using the `librdf_new_model` operation. A `CONTEXT OBJECT` of the type `librdf_model` is returned. A pointer to this `CONTEXT OBJECT` type is used in all operations of the `librdf_model` type. For instance, the “add” operation looks as follows:

```
int librdf_model_add (librdf_model* model, librdf_node* subject,  
                    librdf_node* predicate, librdf_node* object);
```

Conclusion

In our earlier pattern collection *Some Patterns of Component and Language Integration* [Zdu04b] we provided the starting point for a pattern language on the topic of software integration, namely component and language integration. Argument passing is an important issue in the realm of component and language integration because the argument passing styles of two systems to be integrated must be aligned. Thus, in this paper, we have supplemented our earlier patterns for component and language integration with some additional patterns. These patterns can be used by developers to realize argument passing architectures which provide more sophisticated argument passing solutions than ordinary invocations. Of course, these patterns can be applied for other tasks than component and language integration as well. As future work, we plan to further document patterns from the component and language integration domain, and integrate them into a coherent pattern language.

Acknowledgments

Many thanks to my VikingPloP 2005 shepherd Peter Sommerlad, who provided excellent comments which helped me to significantly improve the paper.

References

- [Apa04] Apache Software Foundation. Apache Axis. <http://ws.apache.org/axis/>, 2004.
- [Bec04] Dave Beckett. Redland RDF application framework. <http://www.redland.opensource.ac.uk/>, 2004.
- [BEK⁺00] D. Box, D. Ehnebuske, G. Kakivaya, A. Layman, N. Mendelsohn, H. F. Nielsen, S. Thatte, and D. Winer. Simple object access protocol (SOAP) 1.1. <http://www.w3.org/TR/SOAP/>, 2000.
- [BMR⁺96] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal. *Pattern-oriented Software Architecture - A System of Patterns*. J. Wiley and Sons Ltd., 1996.
- [CEK⁺00] L. Cannon, R. Elliott, L. Kirchhoff, J. Miller, J. Milner, R. Mitze, R. Schan, E. Whittington, N. Spencer, H. Keppel, D. Brader, and M. Brader. Recommended c style and coding standards, 2000.
- [GA99] P. Greenspun and E. Andersson. Using the ArsDigita community system. *ArsDigita Systems Journal*, Feb 1999.
- [GHJV94] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.
- [GNU05] GNU. Program argument syntax conventions. http://www.gnu.org/software/libc/manual% slashhtml_node/Argument-Syntax.html, 2005.

- [Hen00a] K. Henney. Patterns in Java: Patterns of value. *Java Report*, (2), February 2000.
- [Hen00b] K. Henney. Patterns in Java: Value added. *Java Report*, (4), April 2000.
- [Hen05] K. Henney. Context encapsulation – three stories, a language, and some sequences. In *Proceedings of EuroPlop 2005*, Irsee, Germany, July 2005.
- [Kel03] A. Kelly. Encapsulate context. In *Proceedings of EuroPlop 2003*, Irsee, Germany, June 2003.
- [KHH⁺01] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. Getting started with AspectJ. *Communications of the ACM*, October 2001.
- [Nob97] J. Noble. Arguments and results. In *Proceedings of Plop 1997*, Monticello, Illinois, USA, September 1997.
- [NZ00] G. Neumann and U. Zdun. XOTcl, an object-oriented scripting language. In *Proceedings of Tcl2k: The 7th USENIX Tcl/Tk Conference*, Austin, Texas, USA, February 2000.
- [Ous94] J. K. Ousterhout. *Tcl and Tk*. Addison-Wesley, 1994.
- [PL03] G. Patow and F. Lyardet. Open arguments. In *Proceedings of EuroPlop 2003*, Irsee, Germany, June 2003.
- [SG96] M. Shaw and D. Garlan. *Software Architecture: Perspectives on an Emerging Discipline*. Addison-Wesley, 1996.
- [SR98] Peter Sommerlad and Marcel Rüedi. Do-it-yourself reflection. In *Proceedings of Third European Conference on Pattern Languages of Programming and Computing (EuroPlop 1998)*, Irsee, Germany, July 1998.
- [VKZ04] M. Voelter, M. Kircher, and U. Zdun. *Remoting Patterns – Foundations of Enterprise, Internet, and Realtime Distributed Object Middleware*. Wiley Series in Software Design Patterns. October 2004.
- [Zdu03] U. Zdun. Patterns of tracing software structures and dependencies. In *Proceedings of EuroPlop 2003*, Irsee, Germany, June 2003.
- [Zdu04a] U. Zdun. Pattern language for the design of aspect languages and aspect composition frameworks. *IEE Proceedings Software*, 151(2):67–83, April 2004.
- [Zdu04b] U. Zdun. Some patterns of component and language integration. In *Proceedings of 9th European Conference on Pattern Languages of Programs (EuroPlop 2004)*, Irsee, Germany, July 2004.
- [Zdu04c] Uwe Zdun. Loosely coupled web services in remote object federations. In *Proceedings of the Fourth International Conference on Web Engineering (ICWE'04)*, Munich, Germany, July 2004.

Business strategy patterns for sustainable knowledge based comp

Allan Kelly - <http://www.allankelly.net>

“According to leading management thinkers, the manufacturing, service, and information sectors will be based on knowledge in the coming age, and business organizations will evolve into knowledge creators in many ways.

According to [Peter Drucker] we are entering ‘the knowledge society,’ in which ‘the basic resource’ is no longer capital, or natural resources, or labour, but ‘is and will be knowledge’ ...”

1 Abstract

In the knowledge economy individuals and firms derive their competitive advantage from their knowledge and ability to act on this knowledge. Before knowledge can be traded for money it must be packaged and delivered. Knowledge may be packaged as and delivered by way of a service, e.g. a motor-mechanic or a management consultant sell their knowledge as a service; or knowledge may be packaged into products which are sold, e.g. Honda packaged knowledge of car an engine design into cars, motorbikes, lawn mowers, etc.

To best exploit knowledge a firm must know when to sell products, when to sell services, when to switch from one to the other and when, and how, to use one to complement the other. This paper presents several business strategy patterns for dealing with this decision:

Start-up Services for Products

Continuing Services for Product

Complementor, Not Competitor

Services Trump Products

Services Before Product

2 Audience

These patterns are intended to codify several common business practices in a pattern language so they may be communicated and studied more clearly.

The patterns given here are intended for those interested in how corporate strategies may be applied. This group includes both students of the subject and new managers.

The author is interested in the applicability of the pattern form to business domain; whether the form works, what insights it can offer and what value it offers in codifying and communicating business practice.

3 Background

3.1 *Patterns and business strategy*

"What is strategy? There is no single, universally accepted definition. Various authors and managers use the term differently"

It is beyond the scope of this paper to answer this question. However most authors agree that strategy is something beyond reacting to day-to-day offence, strategy implies a conscious decision, in effect a design decision.

"The patterns cover every range of scale in our surroundings: the largest patterns cover aspects of regional structure, middle range patterns cover the shape and activity of buildings, and the smallest patterns deal with the actual physical materials and structures out of which the buildings must be made."

The patterns presented here document re-occurring business strategies brought about by conscious business design decisions. They take as their starting point the idea that the intellectual capital of the business is its greatest asset.

Although these patterns are to be presented at a conference primarily concerned with software patterns they differ in one important way from software patterns.

Software patterns are usually written by those who had a hand in the creation of the software, in part because only these people know the inside of the software. In contrast many of Alexander's patterns come from observation and critique of existing buildings. (Richard Gabriel discusses this in more detail in <http://c2.com/cgi/wiki?WhereDoPatternsComeFrom>.)

The patterns presented here are closer to Alexander's patterns drawing on observation and critique of existing literature and practise. These patterns draw on publicly available sources and the observations of the author.

3.2 *Packaging expertise*

The Knowledge Economy is based on *what you know* not *what you make*. Individuals and firms with specialist knowledge are able to sell their knowledge. The more extensive and exclusive the knowledge the greater the price they can command. However, knowledge cannot be traded in the same way we trade steel, coffee beans or soap-powder, it is not possible to sell "one unit of knowledge."

In order to exploit their knowledge as a commercial product it becomes necessary for sellers to somehow package their expertise in the field. The most obvious way of doing this is to sell consultancy services. Thus, if I require expertise in a field I have no expertise in, say, logistics, I can hire an expert in logistics; knowledge is provided by way of a service.

Still, the consultant must decide how they are to sell their services. For example, the consultant may offer to do the work I require for me, they may do this for a fixed price or on a time-and-materials basis. Alternatively, they may decide to sell training in the domain so I can do the work. So, when knowledge is sold as a service it may be packaged and delivered in a number of different forms.

Alternatively, rather than sell their knowledge as consultancy the domain experts may choose to sell it as an outsourced service. Rather than handle my own logistics I can sub-contract with TNT, Federal Express or other firms who specialise in this field.

Another way in which the knowledge holder may choose to exploit their expertise is to develop an actual product that uses the knowledge and sell that instead. This is common in the software world where companies like SAP embed knowledge in software, so, for example, I could solve my logistics problems by buying a logistics package.

The more knowledge intensive the work the greater the value of the knowledge:

“In today’s knowledge-based economy, superior knowledge is likely to be the most valuable resource of all. Knowledge is valuable precisely because it is hard to manage and hard to trade. ... Knowledge resides inside the heads of lower ranking staff, not the files of top management.”

Carried to an extreme all firms are knowledge enterprises. For example, a car manufacturer has knowledge how to build a factory, knowledge of how to manage a production line, knowledge of car design, knowledge of car marketing, and so on.

The patterns in this paper look at how organizations can exploit their knowledge, either as service-products or actual products, and how they can move between these two types of product.

Sometimes this knowledge is concerned with operations (how we manage our day-to-day business), sometimes with organisation and structures (how we set up our business units) and sometimes with the market we are in (customer tastes and competitor's products).

3.3 Sources

“patterns do not come only from the work of architects and planners”

The patterns documented here draw on personal experience and observation together with publicly available sources. A conscious decision has been made to use public sources so the reader can enquire further themselves.

3.4 Value chain for a complex knowledge based product

For a customer it is sometimes easy to realise the value of product, for example, we buy a new television set, we take it home and plug it in and watch television, the value is easily extracted. But for a complex product, as

is often the case of knowledge-based products, it is not so easy to extract the value.

Some complex products are relatively easy to use, for example, if we buy a new laser printer. We simply plug it in and install the driver software. But other products require costly installation and configuration, and if we are to extract the full value of them the product must be maintained during its lifetime.

Some products, the real value comes when used by expert users, a CAD (Computer Aided Design) system is of little use to the layman but in the hands of an architect or trained designer. It not only increase their productivity will allow them to create designs they otherwise could not.

Other products are valuable, because they can be used by the layman, they may enable a production line worker to magnify their productivity. Or by using advanced products the amount of training required can be reduced. For example, computer aided milling machines can work directly from CAD files, removing the need for a highly trained operator.

Source of customer value for a complex product

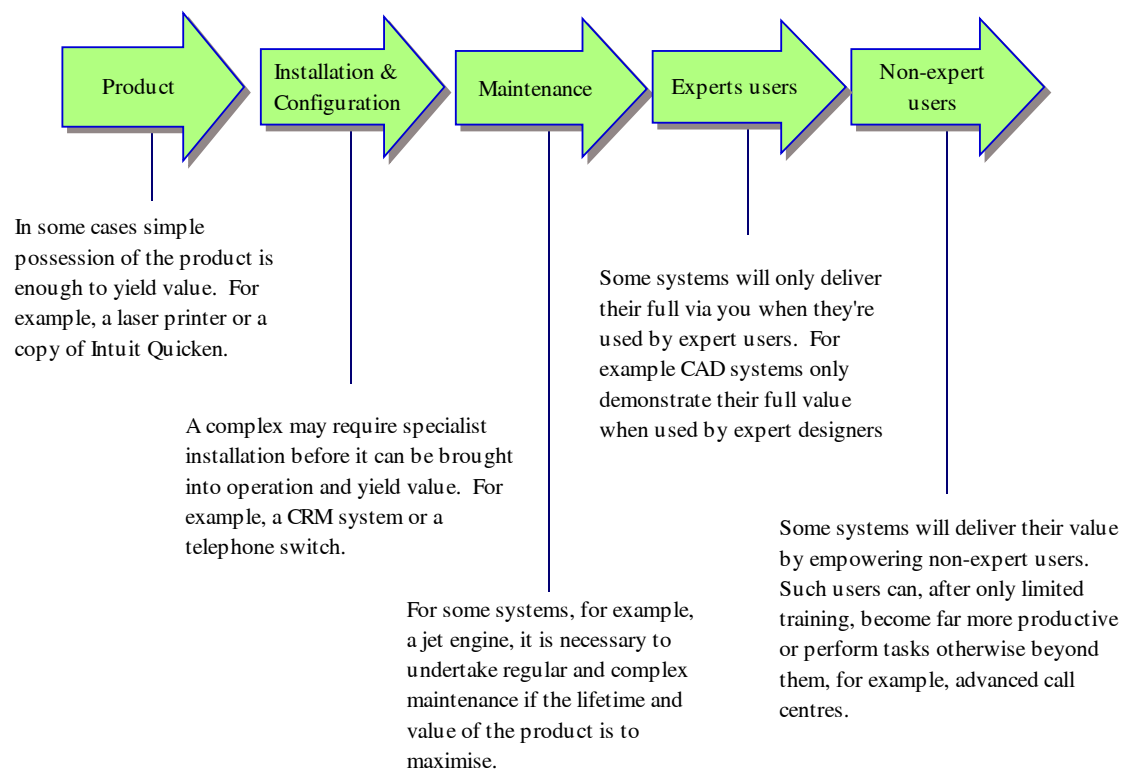


Figure 1 - Value chain for a complex product

Figure 1 shows that customer's value chain for a complex product. Not every step is required for every product. For example, most of the value from a laser printer comes from the product itself, installation and configuration on minimal, most printers require little maintenance and all users receive similar value.

Yet for another product, for example, it telephone switch, value is created, both by the product itself and by the particular installation and configuration options chosen. Indeed, without the correct installation and configuration. The telephone switch is of little use on its own.

There is knowledge embedded in these complex products themselves. And there is also knowledge, surrounding them in these follow-on activities. Configuring the telephone switch is a skilled task, performing maintenance on a jet engine is a skilled task, and the use of a CAD system is as already noted, a skilled task.

Some customers will perform these activities themselves, many others would prefer to have these activities performed by someone else. This opens the opportunity to services, from many of these advanced products and knowledge of how to work with them. To configure them maintain them all use them is as essential as the product itself. Without the product the service may not exist without the service. The value of the product cannot be realised.

The patterns contained within this paper deal with the interplay between products and services. Common to both is the role of knowledge, without knowledge product cannot be created and without knowledge we cannot recognise the value of the product. Product is not necessarily better than service and services are not necessarily better than product. Ideally, the two are complimentary, but if the relationship is not understood the two can come into conflict.

Sometimes the product and the service are provided by the same organisation, and sometimes by different organisations. This too creates the opportunity of conflict, were there should be concord.

4 The Patterns

4.1 Pattern map

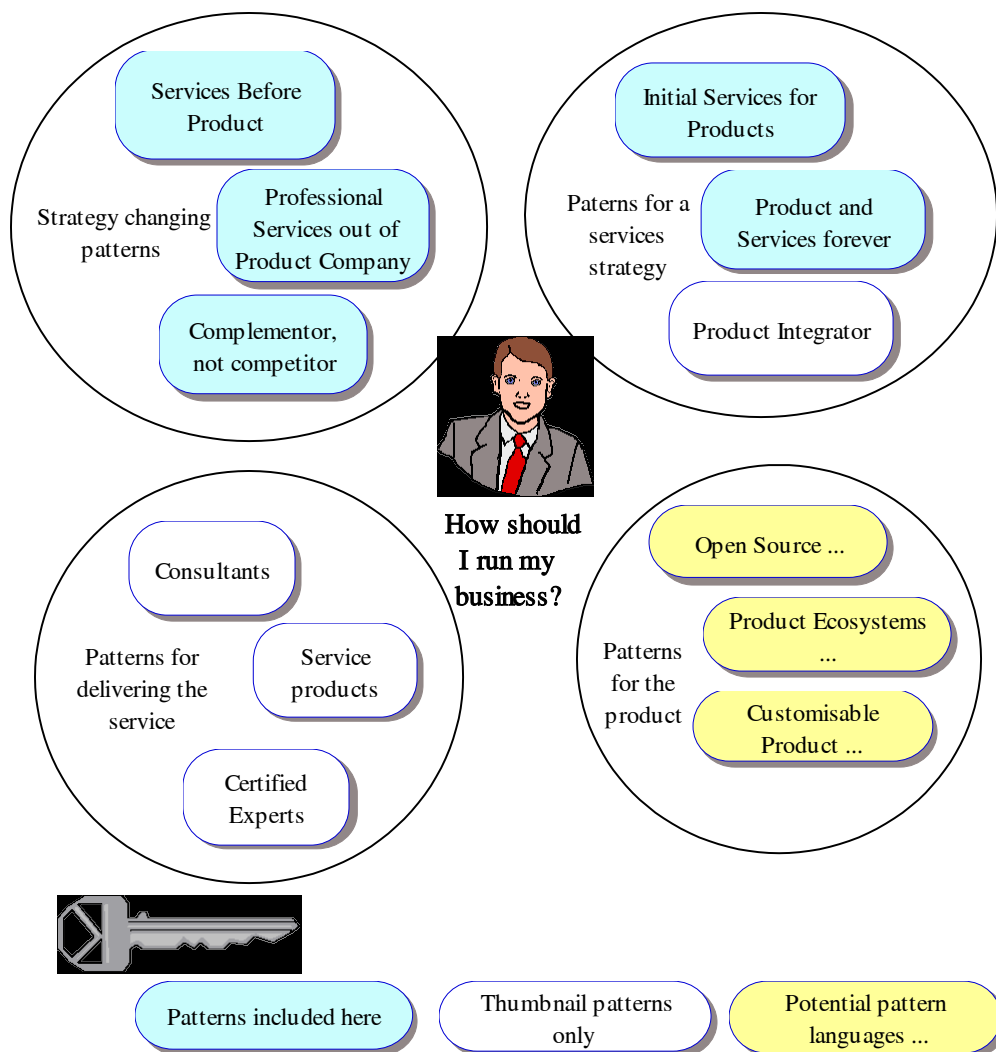


Figure 2 - Pattern relationships

4.2 Thumbnails

| | |
|---|--|
| <p><i>Start-up Services for Products</i></p> <p>Page 9.</p> | <p>Your product serves a complicated market, consequently your product is complicated. Customers need help to get the most from the product. Therefore, create a professional services group within your organization and sell consultancy services to help the introduction of your product.</p> |
| <p><i>Services Before Product</i></p> <p>Page 24.</p> | <p>You are creating a start-up company but you are short of money and/or need a better understanding of the market. In order to get a better understanding of the market you need to get into the market. Therefore, sell consultancy services to start with, you will generate money and get a feel for the market before you start work on your product.</p> |
| <p><i>Product Company from Services Company*</i></p> | <p>Your established company sells professional services but you find you keep re-inventing the wheel. Therefore develop a product that embeds your knowledge, you will be able to reach more customers and grow the market.</p> |
| <p><i>Local Prophets*</i></p> | <p>When attempting to reposition your organization to sell a new product of service it can be difficult to know what to do. Therefore, try to find groups inside the organization who are doing this already and build on them.</p> |
| <p><i>Continuing Services for Product</i></p> <p>Page 12.</p> | <p>Complex products often require ongoing maintenance and support. The company that makes the product already knows a lot about the product so well positioned to do this activity too. By sharing knowledge between services and products operations both can be improved.</p> |
| <p><i>Services Trump Products</i></p> <p>Page 21</p> | <p>Your company has been successful selling products but you are running of growth, you may already be losing money. Therefore, use your knowledge of the products to move up the value chain and sell services instead of or in addition to products.</p> |

| | |
|---|---|
| <i>Complementor, Not Competitor</i> Page 18. | Choosing to compete in multiple product categories against multiple competitors' means you sometimes compete against companies who could help sell your other products. Therefore, withdraw weaker and less strategic products, you can now complement your former competitors and increase sales of your leading products. |
| <i>Product Integrator*</i> | In a market where products from several suppliers must be brought together and made to work together some customers will be willing to pay a third party to do this work. Product Integrators specialise in this type of work. |
| <i>Certified Expert**</i> | Your product is complicated and needs experts users. You don't want or can't satisfy the need for these expert from your own resources. Therefore, create a certification scheme to endorse expert users. |
| <i>Consultants**</i> | Your company sells knowledge by providing consultancy services. This knowledge is communicated and applied by individual practitioners. To sell more services you need more and more people. Therefore, build an organization that can find, train and manage individuals. Each individual is a consultant. |
| <i>Product Services**</i> | Your company has specialist knowledge and expertise in a particular service. Customers would like to use your service but there are many different options. Therefore, pre-define your services as products; limit the number of options to "productize" your service. Customers can now buy your service "off the self." |
| <i>Customisable Product***</i> | There are many ways to customise a product - tool bar settings, configuration files, scripting. Different customisations are applicable for different applications. |
| <i>Product Ecosystems***</i> | Products like Palm pilots and iPods become platforms for which third party companies develop products. |

*Draft only (not included)

**Currently these patterns only exist in thumbnail.

*** Proposed pattern languages in this field.

4.3 Start-up Services for Products

The author's employer sells software tools to mobile telephone operators. The product is very powerful and can be customised in a number of ways. Some of these customisations are simple and can be done by users. Others require a high degree of product knowledge. The company provides professional services consultants who can tailor the product to customer's requirements and embed the product in the customer's methods of working.

| | |
|----------------|---|
| Context | <p>Your company makes money by selling a technically advanced product to other companies. Such products can be difficult to install, integrate and use. Customers may be put off buying your product because of these difficulties, or they never realise the full value of the products.</p> <p>Such complications can lead to dissatisfied customers or deter potential customers from buying your product.</p> |
| Problem | <p>How do you help customers get past initial barriers so they can see the full value of your product?</p> <p>Your product might be difficult to install, or it might need complicated configuration, customers might need training before they can use it, or the customer organisation might need to change the way it does things as a result of the product.</p> |
| Forces | <p>Your business strategy is to make money from selling products. But, customers find it hard to use your product out-of-the-box.</p> <p>The product is difficult to install and requires specialist knowledge to get it working, but once installed it provides worthwhile benefits.</p> <p>Some people might consider these difficulties to be the customer's problem, but if it stops the customer from using your product and potentially buying more of your product then it is your problem.</p> <p>You have tried selling the customer the product and letting them install and configure it, but customers find it difficult to install and configure the product. This has deterred some potential customers from even buying the product. Other customers have not realised the full potential of the product.</p> <p>Customising the product is complicated but without customisation the full value of the product cannot be realised, in fact, without customisation and integration the product may be useless.</p> <p>Unless customers can realise the full potential of product it is difficult to charge a high price to the product. Consequently, you may not recognise the full potential revenue from your product.</p> <p>Customers can use the product themselves but they may require specialist training. Since the product is proprietary to your company</p> |

so it is difficult to find suitable trainers and costly to hire.

Therefore...

Solution

Provide Professional Services to help customers integrate and customise your product. Your staff may do all of the work or just assist customers in doing it themselves.

For a complicated and expensive product it is worth holding the customer's hand for a while. You can also provide training services to educate customers and users in how to use the product and how to get the most from it.

In a small company the services may be provided by development staff, as your company grows you will want to create a dedicated group of consultants who work with customers to integrate and customise the product.

Development staff may not be the best people to provide consultancy services; temperament and personal objectives often differ between back-room and customer-facing staff. Such staff may have deliberately chosen a backroom position to avoid customer contact.

Consequences

Your consultants can get customers over the initial hurdles to using your product. They can install, customise and integrate the product.

Once customers are over the initial blocks they can start to realise some of the benefits of the product. The specialist consultants can now move on to customising the product to achieve maximum value for the customer.

When a product is installed, integrated and customised to a specific environment there will be a greater acceptance of the product. Customers are less likely to use a competitors product and more likely to buy from you again.

By having your people work with the customer, and by tailoring your product to the customers knees you will create a closer relationship with the customer, this is good for both sides.

Consultancy is not limited to installation, integration and customisation. Other services such as training can be offered to help customers - although different services may be delivered by different individuals all working for you professional services group.

Your consultants are experts not just in your product but in customer's businesses, and integration issues. Consultants who are regularly seeing customers and dealing with their problems are a valuable source of information when it comes to deciding what to develop next.

Providing consultants is not cheap, in addition, there are often travel and accommodation costs incurred when working with customers. These costs must be covered somehow. If the costs are included in

the product price, then the overall prize will be higher, and the sale more difficult. Alternatively, these costs may be billed separately, in which case the customer must be persuaded to pay them in addition to the product costs. This too will make the sale more difficult.

Customers may be put off by the costs of customisation and training. Products such as SAP are reputed to cost a lot in terms of time and money to install, customise and train users.

You will need to hire and retain consultants. Finding people with particular mix of skills required may be difficult. Once found these consultants will need to be paid regardless of whether they work, or not.

Companies - especially small ones - that use their development engineers to provide services to customers will find that their product development is hindered.

Examples

Smith Communications (a pseudonym) had one product, an advanced e-mailing system. The company made its money from selling the product but it needed to be integrated with a customers other systems such as a database and accounts system. So the company set up a professional services group that worked with customers directly to integrate the product.

Professional services were sold near cost, sales staff would even throw in several days consultancy to sweeten a sale, consequently the professional services group never made a profit. When the company hit financial trouble and layoffs were required the group was the first to be cut.

However, the product still needed installation and configuration, so some of the staff found themselves hired back on ad hoc contracts.

Related patterns

Selling services provides for a second revenue stream but you should be clear where your competitive advantage lies. If your value added lies with the consultants not the product it might be time to consider Continuing Services for Product or Services Trump Products - you may also consider Open Source options. Some companies become overly dependent on selling services rather than product without realising it.

Customisable Product describes how a product may be made more configurable.

Certified Expert provides another route for solving this problem, it may be used in combination with or as an alternative to this pattern.

Sources & Known Uses

The author has worked in several companies that have employed this model, these cover the telecoms, office automation and financial sectors.

See also *Secrets of Software Success*

4.4 Continuing Services for Product

Jack Welch tells how during the 1990's General Electric (GE) entered the service business in a big way. What had been thought of as "after sales service" became a major contributor to profits. For example, at the start of the 1990's the firm was already a leading maker of aircraft engines, by 2000 over 60% of engine revenue came from servicing such engines - both GE engines and those of competitors like Rolls-Royce.

Context

You are successful in your market and are looking for growth opportunities.

Your product has a long life span; the sale is only the beginning. Over its lifetime the product requires on-going service and maintenance. These requirements go beyond Start-up Services for Products.

Problem

How do you improve your product, your customer's experience of your product and grow your company at the same time?

Forces

You know a lot about the lifetime management of your products but you consider yourself a designer and manufacturer, indeed you may be one of other leading designers or manufacturers; but that is where your responsibility ends, customers are expected to arrange and pay for ongoing service themselves.

You have traditionally been concerned with the sale price of your product but customers are, perhaps increasingly, concerned with the total cost of ownership over the lifetime of the product. You can better serve your customers, if you can align your objectives with their objectives.

Customers understand that complex products need to be maintained but they don't necessarily want to do this themselves. In fact, they may view such activities as distractions from their core business.

In a competitive market you need to undercut your competitors to make a sale, but, a low price may not be profitable.

You are ready, one of the leading companies in the market, but this means that growth opportunities are limited. Being number one in the market is great, but shareholders still expect growth.

Therefore...

Solution

Use your product, industry and market knowledge to compete at additional points in the value chain (Figure 2) and use what you learn to improve the product. Taking a wider perspective on the consumer experience will provide new opportunities for innovation, revenue growth and improved customer satisfaction.

Customers continue to consume your product long after the sale is

closed. As the developer and manufacturer of a complex product you have specialist knowledge of the product. This knowledge can be used throughout the product life cycle.

Knowledge gained in design and production may be applicable - later in the product's life, particularly in maintenance . Similarly, knowledge gained later in the product life may be fed back to design and production phase. This allows you to create a virtuous circle of learning.

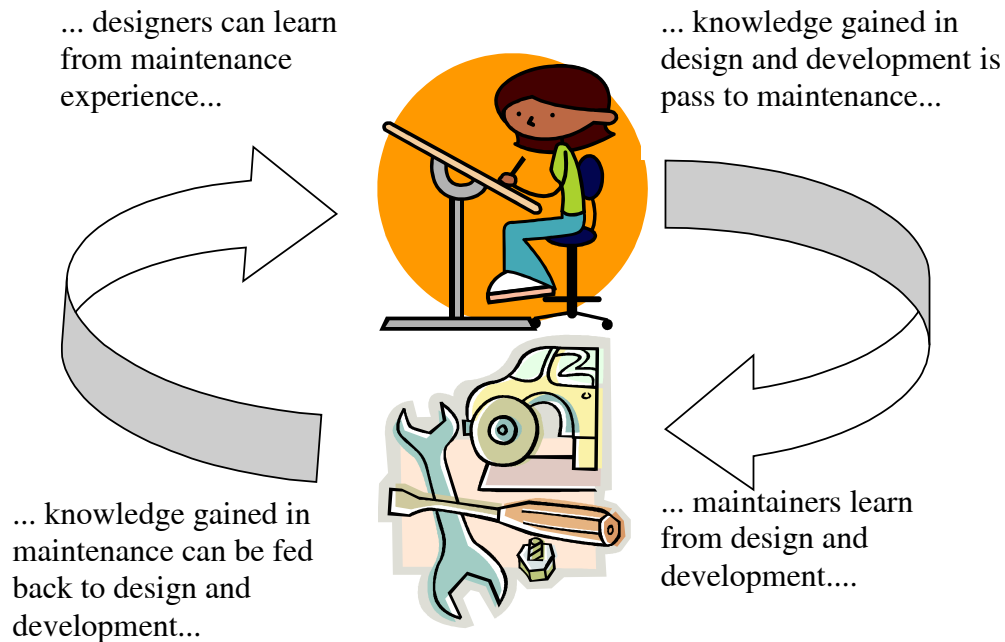


Figure 3 - Create a virtuous learning circle between development and maintenance teams

Having people on site with a customer provides good opportunities to understand what the customer wants from the product, how they use it and what features they need and value. This can be of benefit to both supplier and customer.

Obviously you know your own products best yet competing products will not be too dissimilar and these will need servicing to. You may choose to offer service work with these products to this will increase the size of your market, provide a fuller service to customers and help you understand the competition better.

Customers who buy your product for its innovative features may be willing to pay for ongoing support and maintenance charges and tolerate glitches.

Remember to balance your product development operations against your service operations. Product development generates little revenue by itself and costs are incurred long before revenue is

generated. When revenue does arrive there is a simple moment with product is exchanged for cash.

In service operations the cash will arrive shortly after the service is performed. However services continue to be delivered so cash continues to arrive the business. Managers must seek to smooth the flow of work in cash to ensure a steady flow of work and high utilisation of resources.

Yours service operations should add value to the customer; it is no use shipping a low quality product in the hope of charging them for service. Customers may feel stung if they receive a poor product and hefty service bill. Table 1 illustrates this. It is important to understand the incentives created on both sides by your sales model.

| Physical product | Service product | |
|-------------------------------------|-------------------------------------|--|
| Sell at profit | Sell at profit | Good position for seller - be careful not to over charge the customers or they may find an alternative supplier. (Continuing Services for Product is being used.) |
| Sell at profit | Sell at loss, or Sell at cost | Only sustainable in the short run; incentive to create a product that does not need service so this is good for the buyer. Free service may be seen as a type of quality guarantee. |
| Sell at cost | Sell at loss, or Sell at cost | Not sustainable. If you can't make money for either product or service sales you should quit the market. |
| Sell at cost, or Sell at loss | Sell at profit | Sustainable if both sides understand the model and incentives are aligned. (Continuing Services for Product is being used.) If <i>Software Ltd</i> agrees to write a special software application for <i>Mega Corp</i> at a fixed price but charge for each bug-fix then it will be in <i>Software's</i> interest to delivery a low-quality product and make profits on bug fixes. If on the other hand <i>Software Ltd</i> agrees to a fixed price contract with a fixed annual service fee it will be in their interest to delivery a high quality product to start with, this ensures ongoing service costs are minimised. Of course <i>Software</i> may still deliver a low |

| | | |
|--|--|---|
| | | quality product, and in the short term this may save them money, but in a long-term this isn't not in their interest. |
|--|--|---|

Table 1 - Source of profits determines incentives when using Continuing Services for Product

Consequences Using this pattern the sale and installation are only the start of a commercial relationship between buyer and seller, the initial revenues are only a small part of the overall revenue stream. For such a product the ongoing service revenue, over a number of years, may be far greater than the initial sale price.

As long as your service offering is priced to make a enough profit you can afford to reduce your initial selling price in the hope that you will make a profit on future services. You may even sell your product at a loss provided you are sure can be recover the money on services.

Buying your product solves one problem for the customer but creates another. As the seller you know this first and best, you can solve the customers problem, make your product more attractive and generate a revenue stream for yourself.

Gaining an ongoing revenue stream may allow you to reduce your sales price to gain a sale. It also means the company's future income is in place and is less dependent on *the next sale*.

(Competitors may still win the business if they can show that the total cost of ownership of your product is higher than the total cost of ownership of theirs. However such figures are difficult to calculate and some customers may be prepared to pay a higher price overall if they can reduce their initial capital outlay.)

Using this model to expand your business you must be able to sell more consultants and service technicians, thus you need to hire more people. It takes time to recruit and train new people in technically complex so your growth rate may be constrained. Retaining current staff can also be a problem, especially when they are highly trained and experienced.

Apportioning costs and benefits becomes very difficult - as we see with quality. In the short run it may be more profitable to hold back on new features and quality improvements to reduce costs and boost fees. However, in the longer term this risks the product becoming dated and uncompetitive.

There are opportunities created by having design, production and maintenance with in one organisation. For example, GE has designed software into its aero engines to provide additional information from maintenance services. Increased development costs will be offset by reduced maintenance costs.

If you choose to service competitor products in addition to your own products you must respect your competitor's intellectual property rights. You may become aware of innovations in rival products that you could incorporate into your own products. However, in doing so, you may break, intellectual property law.

Examples

This pattern is itself a variation on the classic razor sales technique - sell the razor cheap and make the money on the blades. Other examples include games console manufacturers (e.g. Sony, Nintendo) who make their money on the games, and printer manufactures (e.g. HP, Lexmark) that make money on ink cartridges. However, in this case, it is a knowledge-based service that is being sold rather than consumables.

Related patterns

Its a Relationship, Not a Sale advises us to look the beyond the current transaction and consider the long-term relationship during which time there may be multiple sales.

Start-up Services for Products discussed the need to provide services to get a product installed and working correctly. Continuing Services for Product goes beyond this model and shows how ongoing services can be sold in addition to the initial product.

Services for competing products may include support for Open Source products.

Sources and Known examples

The author has seen this pattern applied in both its functional and dysfunctional versions.

Welch describes GE's entry into the jet engine services market.

An aside: Dysfunctional Product with Continuing Services

There is a dysfunctional version of Continuing Services for Product. It occurs when the incentives for seller are misaligned. This is illustrated by the story of Ball Group (a pseudonym).

Ball Group sold software to manage banks' treasury operations. The company had little capital and found the revenue from selling consultancy services useful.

After a while the company had more consultants than developers and made more revenue from selling services than product so the product was sold cheaply. The incentive for Ball Group was to cut corners, spending less on quality and documentation made customers more dependent on consultancy services. This might appear as a win-win: lower costs and higher revenues but it was not a win-win for customers.

For a while Ball Group flourished and grew, staff numbers increased and new product development began. However the model was unsustainable and after several rounds of redundancies the company was acquired.

While Ball Group's product added value to the customers operations the subsequent services did not, work was merely displaced from pre-sale to post-sale. The need to buy services detracted from value of the product.

Customers were paying twice, once for the product, and again for consultants and fixes, this might look good on the balance sheet of Ball Group -who made the customers pay for work which should have been done before the product ships, thereby reducing their costs and increasing the revenue. Eventually customers would realise the true cost of the product and the low initial quality.

Neither was about being developed for long-term, the only work that occurred was to address the immediate problems or to implement customer requested features. Meanwhile competitors were developing products with a longer-term view.

Moral: Continuing Services for Products can be Win-Win or Lose-Lose

If implemented well Continuing Services for Product can be win-win for supplier and customer. The supplier wins, because they get to increase their revenues and learn more about the hall product lifecycle, which in turn allows them to improve the product.

The customer wins, because they get better value for money from their supplier and service organisation, and overtime they get an improved product because the supplier is able to use what they have learned in servicing the product to improve the original product.

However, if implemented badly Continuing Services for Product can be lose-lose: the supplier can ship, inadequate product knowing they can hide the defects behind their service contracts, over time these contracts and the product will become more expensive.

The customer loses, because they get an inferior product and an increased total cost. Eventually, they will change supplier.

4.5 Complementor, Not Competitor

“Three years, a lot of activity, and a few billion dollars later, we still weren’t [application software] leaders...

However, one thing we were doing exceptionally well was irritating the heck out of the leading application providers - companies like SAP, PeopleSoft and JD Edwards. These companies were in a great position to generate a lot of business for us ...

What we said to them was ‘We are going to leave this market to you; we are going to be your partners rather than your competitor’.”

| | |
|---------------------|---|
| Context | Within your product portfolio you sell two products that are complementary and usually sold together, say Widgets and Foobars. It doesn’t make sense to buy one without the other. You are not the only company selling such products but most of your competitors sell just Widgets or Foobars. Indeed, some people buy your Widgets and use them with other people’s Foobars. (The reverse seldom happens even if it possible.) |
| Problem | How do you arrange your product and services portfolio so you maximise your profits and don’t loose money on products? |
| Forces | <p>Customers look to buy a total solution of Widgets and Foobars, and this is what you have traditionally sold. But, while your Widgets are very good there are better Foobars on the market. Customers may choose to buy Foobars elsewhere and your competitors are unlikely to recommend your Widgets to go with their products.</p> <p>Developing both Widgets and Foobars has allowed you to innovate in the past but your competitors are focused on innovation in Widgets or in Foobars, having both does not offer a lot more opportunities for innovation.</p> <p>By selling these products together you make a bigger sale so the revenue is greater but when you look at it in detail you are making most of your profit from the Widgets.</p> <p>Both Widgets and Foobars are expensive to develop, Widgets make money but Foobars are less profitable and may be losing money.</p> <p>Traditionally both Widgets and Foobars have fitted with your <i>core competencies</i> - the production of both was important in your business. But your business strategy and core competencies have changed, Widgets, but Foobars are no longer core to your business. Even if you still make money on the sale of Foobars they may not fit to be a long-term goals.</p> |
| Therefore... | |
| Solution | Concentrate your activities on the most profitable part of the solution, discontinue the less profitable parts and replace the missing |

pieces with ones from other producers. Rather than competing with everyone seek to complement those who can help you sell more of your most profitable products.

Customers will want to buy Widgets and Foobars, now you no longer compete with the Foobar makers they can be a source of customers. Work closely with Foobar sellers, get to know their products and form strategic partnerships to ensure their products work well with your Widgets.

Discontinuing a loss making product should immediately help your balance sheet. By partnering with others you can increase sales revenue for your profitable products - a classic *win-win* situation.

You need to prove to your new partners that you are committed to this strategy. Move fast and decisively to show that you are now a friend not an enemy. Foobar manufacturers could still recommend one of the other Widget manufacturers so work together to be the best Widget for their Foobars.

Consequences You no longer supply a total solution with your own products, you sell your most profitable product and complement it with third party products to offer a total solution.

Opportunities for innovation between Widgets and Foobars are more difficult to find and exploit, however, you can be more focused on innovation in Widgets.

Sales of Widgets only may be smaller but they will be more profitable, plus you are hoping to sell more Widgets by working together with the Foobar manufacturers. You may look to make up revenue from consultancy services too (see other patterns in this paper).

You have saved the cost of developing an expensive product. Product development costs are lower, the cash may be used elsewhere, say, in new products or services.

However, if you will need to ensure that your Widgets are compatible with the various Foobars available elsewhere. Ensuring compatibility can in itself be a timely and costly endeavour.

Customers are no longer locked into your products; they now choose your products because you have the best solution to their needs, not because they have no choice.

New partners may seek to lock you into their product; if you become dependent on one Foobar maker you will be in a weak position if they ask for special consideration and price cuts. Work with several Foobar partners so you have the choice to walk away from a deal if a partner asks too much.

Examples Games console manufactures usually lose money on each console

sold while making large profits on the games for the consoles. After selling over 6 million Dreamcast consoles and losing \$500 million, Sega left the market in 2001 and chose to focus on producing software for Sony, Microsoft and Nintendo consoles. In 2004 Sega merged with Sammy and made healthy profits in 2005.

Related patterns

Contrast with Continuing Services for Product where the sale of one product - possibly at a loss - allows you to make money from a second.

You still know a lot about Widgets and Foobars so you may also be in a position to sell integration services – see *Product Integrator*.

Sources & Known Examples

The Economist details the Sega story, while Louis Gerstner tells the IBM story at length.

Aside: Competitor not Complementor

There is a mirror image to this pattern. Sometimes a company which could be a complementor in a market decides to enter the market as a competitor. For example, at around the time, Sega decided to leave the games console market Microsoft entered the market with their Xbox console.

The firm spent millions of dollars researching and developing the new console to enter the same highly competitive market Sega was leaving. Microsoft could have chosen to play the role of complementor and develop games for Sony and Nintendo consoles but instead chose become a player in the market themselves.

Microsoft reasoned that despite the cost of entering the games console market. It was a market they had to be in as part of their overall strategy. To enter the market the company leveraged their existing core competency of software development but had to develop new competencies in hardware development and console marketing.

4.6 Services Trump Products

By the early 1990's IBM was in trouble, companies where buying PC's not mainframes. The company realised that customers wanted services and results rather than products. As the IT industry became more complicated customers would rather pay someone else to provide IT. So, IBM re-invented itself as an IT services company.

| | |
|---------------------|--|
| Context | <p>You have high brand awareness with customers and your products have a good reputation. Yet sales growth is slowing or declining.</p> <p>Your market may be nearing saturation, or, maybe competitors are producing better (or cheaper) products, maybe your market has changed - new products don't have the margins of old products.</p> |
| Problem | How do you grow your business when selling more products doesn't work any more? |
| Forces | <p>Customers want your product not for its innate qualities but for the capabilities it provides, e.g. they aren't interested in buying a computer for its technical specification, they are interested in running a stock-control system.</p> <p>The products you sell are increasingly commodities; customers can buy similar products from competitors. Consequently, you're forced to compete on price, but you do not have a price advantage, your company is not designed to be a low cost producer, neither do you believe you can become a low-cost producer, any time soon.</p> <p>Your business is centred on the first stage of the value chain (the product part of Figure 2), growth and profits are increasingly difficult to get in this stage, but the later stages of the value chain still offer opportunities. Even as the products become increasingly commoditized, the opportunities to add value, and later in the chain increase.</p> <p>Your product is just one part of a bigger solution. There are a host of activities that occur around your product. But, it is these activities that add value to the customers not your product itself.</p> <p>Since these activities add more value there is more profit to be had from supplying these services than there is from supplying the product.</p> |
| Therefore... | |
| Solution | <p>Use your experience from selling products to sell services in the same industry. This allows you to move to a more profitable part of the value chain.</p> <p>Selling services may help you to sell even more products (see Start-up Services for Products for examples) or it may mean you have to drop products so you can work with different suppliers - see</p> |

Complementor, Not Competitor.

Unlike a one off product sale, a service contract represents an ongoing commitment by both buyer and seller so the revenue stream will continue. You may earn less from a sale on day one but overtime, your total income will be greater

Providing a service is different from selling products, sales are based on relationships not short transactions. It is important to ensure compensation schemes are aligned to support the new goal not the old one..

To justify customer trust and costs you must show customers that the end result is noticeably better, this involves a number of intangible factors. Although these intangibles are difficult to get right and quantify they are the important in beating competitors.

Scaling up a services business can be more difficult than expanding a product company. Service delivery is inherently dependent on the people delivering the service, finding the right people, motivating and retaining them rather. Growth is no longer simply about manufacturing and selling more products, it is about hiring the right people and helping them work in the most effective manner possible.

Try to find some *Local Prophets* and use these people and groups to help create your new organization, e.g. IBM built on the experience of the ISSC group .

Even if the company is retreating from product activities and laying off staff you may need to hire staff in your services business. Similarly, you may find the need to buy in more services expertise to provide more skills and experience in your new strategy.

This pattern is not just about retreating from one sector and expanding in another more profitable sector. It is about building on what you already know and serving your customers better.

Consequences You no longer sell a commodity product; you sell capabilities as a value adding service. Rather than sell the computer you sell the results of the computer. This is reflected in a improved profit and growth.

The sale focuses on the final product rather than the individual pieces and activities that go to make up the results. You address the customers needs directly rather than showing how your product allows others to address their needs.

If your analysis is right the services you supply will carry a higher profit margin than products. This is especially true where products are becoming a commodity - and products become a commodity faster when common standards are in place.

Examples IBM is one of the best known examples of a product company that

has converted itself into a services company. There are many other examples in different industries, e.g. Home Depot in the US home improvement market and Rolls-Royce aero-engines.

Another, less successful, example is the UK based *Boots the Chemist*. In the late 1990's supermarkets started to undercut Boots prices, its brand name offered little protection.

Boots responded with "Wellness" centres. The company had long offered optician services through Boots the Opticians, now it added dental, aromatherapy and other "medical" services. However, the company found it difficult to produce profit from these operations and in autumn 2004 the company sold the dental and laser eye treatment operations.

Related patterns

Complementor, Not Competitor can be used word you wish to offer services for products produced by (former) competitors.

Compensate Success discusses the need to ensure that your employees are rewarded in line the company's goals.

This pattern has similarities to Continuing Services for Product, both are about product companies that move into services. However, in Continuing Services for Product the company expands into the sector while in Services Trump Products the company retreats from product.

Sources

Home Depot is described in the Financial Times of 9 May 2004 .

Lou Gerstner describes IBM's change of strategy in *Who says Elephants can't dance?*

Boot strategy is described in *The Economist* and the companies own press releases on its website.

4.7 Services Before Product

The author once worked for a British company that pioneered handheld PC's. The founders had left a previous company and undertaken consultancy projects before raising enough capital to develop a 8088 based pocket PC.

| | |
|---------------------|---|
| Context | You have an idea for a product; you have some expertise and a little money but not enough of either. |
| Problem | How do you leverage your existing expertise and money to get closer to your goal of creating a product ? |
| Forces | <p>You need money to develop products but those with money (e.g. business angels and venture capitalists) expect to see some business plans. Worse still, there may not be anybody willing to risk money in your field.</p> <p>In order to develop business plans you need get into the market, but you can't get into the market without something to sell.</p> <p>Founding a company requires good timing. Founders need to be ready to quit their jobs and join the new company, business plans should be laid out, funding should be in place, the sales pipeline should be ready to go. But, getting all factors to coincide can be difficult and delay the endeavours. Delays may lose customers (who find other suppliers) or founders may be offered attractive positions elsewhere.</p> |
| Therefore... | |
| Solution | <p>Fund your new company by providing professional consultancy services in the same field as your envisioned product. Increase your expertise and knowledge of the field and, improve your cash position.</p> <p>Selling expertise through consultancy services requires less preparation than developing and manufacturing a product. Cash can be produced relatively quickly thus allowing you and other founders to eat while you develop products and business plans.</p> <p>(This is not to say, consultancy provides for a free lunch, creating a consultancy will require expenditure. However, cash can be generated more quickly via consultancy than through product development.)</p> <p>Being in the market allows you to identify potential customers, their needs and where existing products fail. You will also be able to meet potential competitors and complementors.</p> <p>When your knowledge and cash reserves are good you can start to develop products for sale.</p> <p>There are risks in changing your business model. Changing from a</p> |

services based to a product based company is not simply a case of “flipping the switch.” Culture, people and financing are all different.

Consultants and product developers are different. People hired to work as consultants may not want to work on product development when the time comes to switch, similarly, product developers might not like the idea of spending time with customers and in hotels until the cash is available for product development.

Consequences Being in the market increases your credibility when you go looking for funding. You can refine your product ideas, your marketing, your strategy and expand contacts. Even if you have not been able to start developing product you will be able to make a better case for the product.

You have entered the market by selling yourself. As a short-term move this gets you exposure to the market and cash, you can enhance your technical knowledge and understanding of the market.

This move also buys time to get organised - the founders don't have to all join on day one. As you develop the sales pipeline you can bring more people onboard. However, while you are organizing and improving your knowledge the market is moving forward, you have also provided time for potential rivals to enter the market too.

Consultancy work can provide a steady, lucrative, cash flow. This can be addictive. Using your staff for product development will cut cash flow until the product is ready. Product launch can be expensive and risky; the promise of cash today may be more appealing than cash tomorrow.

Indeed, provided you are successful at offering services you should consider the need to change your business plan altogether. Just because you originally envisaged producing products does not mean you have to produce products someday. If there is good money made in services than there is no reason to stop.

Examples John is CEO of a small software company in central England. The company makes money by developing software for other businesses. Profits are used to help develop the company's own products. The time has been used to focus the company on markets where its technology is useful.

Also known as Bootstrap

Related patterns Contrast this pattern with Services Trump Products. In both patterns the company provides services, however, in one the company is moving from products to services and in the other the company is moving from services to products.

Early versions of the product may provide an opportunity to follow Continuing Services for Product, later, when the product is better developed consider Complementor, Not Competitor, other firms may

supply the services and allow you to concentrate on the product.

It is important that you whole know which pattern you are following. Problems can occur when people are pursuing different objectives.

**Sources &
Known
Examples**

Authors observations

Martek Marine, Financial Times - undertook ship services work to finance development of maritime safety technology.

Acknowledgements

The author would like to thank Klaus Marquardt for his valuable advice and suggests during shepherding for VikingPLoP 2005 - and for keeping me working until the last minute!

In addition, this paper was greatly enhanced by comments and feedback from the business patterns workshop group at VikingPLoP - in particular Linda Rising, Cecilia Haskins and Florian Humplik - who was also good enough to give a second set of comments on the revised paper.

History

| Date | Event |
|---------------|---|
| April 2005 | Submitted to VikingPLoP 2005 conference. |
| August 2005 | Shepherded version submitted for VikingPLoP. |
| November 2005 | Incorporated VikingPLoP comments and feedback. |
| December 2005 | Additional editing and comments from Florian Humplik. |

Bibliography

Load Balancing and High Availability Patterns

Kai Wei and Antti Ylä-Jääski

Telecommunications Software and Multimedia Laboratory

Helsinki University of Technology

kaiwei@cc.hut.fi

Abstract

To resolve the limited power of a single computing machine, clusters are widely used in the Internet world. As a result of using clusters great attention has been paid to load balancing. In this paper, from the cluster point of view, various load balancing and high availability related patterns are introduced. Software developers can utilize the patterns presented in this paper to develop load balancing and high availability applications.

1 Introduction

Load balancing and high availability solutions have been implemented in critical equipment for a long time. Their importance has been further highlighted as the Internet evolves to be an essential part of people's life. More and more efforts have been spent on the development of load balancing and high availability solutions. This paper presents patterns of commonly used load balancing and high availability solutions.

Load balancing provides benefits to the Internet in a number of ways: it improves efficiency for network bandwidth and server utilization, increases the reliability of services to clients and enhances scalability of services. Identifying the need and choosing the pattern accordingly may help to design a load balancing solution. In addition to load balancing, high availability of a service is usually required to make the service available all the time. Thus, the high availability solution is designed along with the load balancing solution.

The relationships between the patterns to be introduced in this paper are presented in Figure 1. From the figure, we can see that both load balancing with pre-defined algorithms pattern and high availability of server in cluster pattern can support the pattern of server cluster with one unique virtual access point. This means that a load balancing pattern can work with high availability pattern within the scope of a cluster environment.

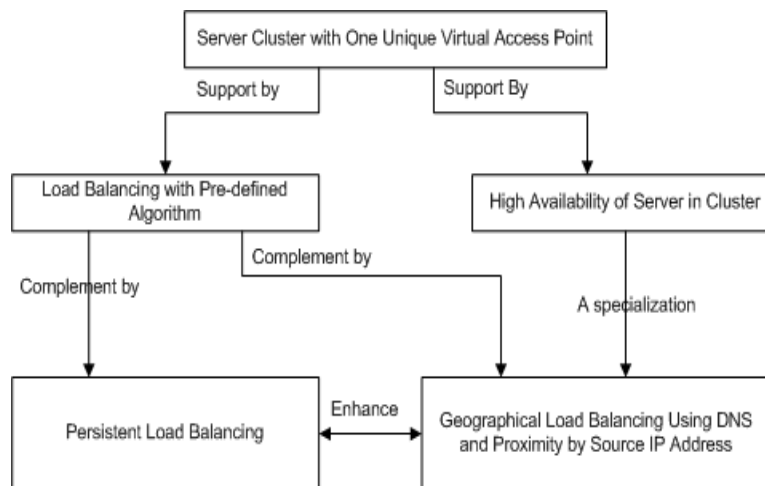


Figure 1: Pattern Relationships

The persistent load balancing with pre-defined algorithms pattern and geographical load balancing using DNS and proximity by source IP address pattern complement the pattern of load balancing with pre-defined algorithms. Meanwhile, it is possible for them to enhance each other. A load balancing pattern could be chosen according to the requirements, such as distance between servers, the persistence of traffic, etc. In general, several patterns can work simultaneously without conflicting with each other.

In addition to providing the function of load balancing, geographical load balancing using DNS and proximity by source IP address is a specialization of high availability of server in cluster. Geographical high availability may be the best name to describe this function. It can balance the load according to the availability of sites. In other words, the DNS server only returns the IP address of the available real server when it is enquired.

The terms used in the patterns are explained at the end of the paper.

2 Patterns

2.1 Pattern: Server Cluster with One Unique Virtual Access Point

Context: In a network, each server is usually dedicated to provide a specified amount of unique services. A certain service in one of these servers might be in high demand.

Problem: When the server is over utilized, it responds the clients' request slowly or does not respond at all. This in turn can reduce the performance of the entire network, as requests of clients are rejected by the server and then resubmitted by the clients.

Forces: The server in which one service runs has limited capability, adding more hardware or upgrading hardware results in a more complex application and an increase in

cost.

If one specific service runs over only on one server, the service is not available when it needs maintenance or OS upgrade.

Under the condition that requires the availability of 24X7, such as in production line or B2B web, interruption of a service is not allowed.

If the service is critical in the Internet, such as an authentication service, its unavailability affects the entire network even though other services are available.

Bugs in application program could cause an interruption of service. They may result in financial loss. For example, a one-hour interruption of an online shop may lead to big losses.

Solution: One unique virtual access point is adopted for handling the requests from clients. Identical services run over multiple servers that connect to all the other servers directly or via a back-end network. These servers are named real servers. The number of servers expands with the increasing requests. One unique virtual access point is exposed to the client side so that all requests are sent to this virtual access point, followed by further deliveries to real servers.

A dedicated piece of equipment, in the form of specialized software running on a PC or a specialized hardware, can be put in-between the real servers and clients. On the real server side of this dedicated equipment, all real servers connect to it directly or via a back-end network. On the client side, a virtual IP address is assigned to a pool of real servers in which an identical service is running. The requests from the clients are first sent to the virtual IP address, and then the dedicated equipment distributes them to the available real servers. The responses from real servers will be returned to the dedicated equipment, then forward back to the client. The dedicated equipment works like a relay, taking care of collecting and distributing the requests, and forwarding the responses.

Resulting Context: When a server cluster is used, no more hardware is required to be added to each server, the possibility of complicating the service is avoided.

The real servers behind the virtual access point are connected directly or via a back-end network. This makes dynamic expansion possible.

When one server is not available for some reasons, the rest of the servers can take over its load. However, an interruption of service will happen when all the real servers go down. For example, if all the real servers are put in the same physical place and there is a power cut, the real servers will lose their power supply and it will result in the interruption of the service. The high availability of server in cluster pattern can provide a solution to overcome this problem.

Even when multiple real servers are available to provide an identical service, it is likely that the service will not always be available for certain clients if the load to them is balanced unevenly. The load balancing pattern that will provide a solution for this will be described later.

For some of traffic, especially ones based on TCP or UDP, persistence is required. Namely, requests over one connection must be handled by one real server all the time. This issue will be discussed in the persistent load balancing section.

2.2 Pattern: Load Balancing With Pre-defined Algorithms

Context: There is a server cluster. The real servers in it have either similar or different capabilities.

Problem: If the virtual access point in a server cluster virtual access point cannot balance traffic according to the capability of each real server, real servers with low capability might be overwhelmed while real servers with high capability will have redundant capability.

Forces: The performance of real servers should not be sacrificed when implementing the load balancing solutions. Thus, it should be implemented via the virtual access point.

In case of real servers in cluster having the same capabilities and their capabilities remaining equal during the handling of requests, evenly balancing the load can result in maximum benefit.

In some cases, even though the servers in cluster have the same capabilities, varied resources required by each request may result in the real-time capability of each server being different at any time. To balance the load according to the actual capability of servers can lead to good performance.

When traffic load rockets and the capabilities of existing real servers are saturated, new real servers should be added to the pool transparently without interrupting access to the service.

Solution: The load of traffic can be controlled in virtual access point by using various algorithms. Using these algorithms, the load can be balanced intelligently and the capability of whole cluster can be fully utilized. The most popular algorithms are roundrobin, least connection, response time, dynamic load and weight balancing. These algorithms can be used separately, as well as combined together, e.g., the weight algorithm can work with any other algorithm at the same time.

With the roundrobin algorithm, new connections are distributed to each real server in turn. Namely, the first connection goes to the first real server in the server cluster, the next connection goes to the second real server, followed by the third connection, and so on. When all the real servers in this cluster have received at least one connection, the distributing process starts all over again.

With the least connection algorithm, the number of connections currently distributed to each real server is measured by the virtual access point in real time. The real server with the fewest connections is chosen for distributing the next client connection request.

The response time algorithm uses the response time of real servers to distribute new connections to real servers. The virtual access point monitors the response time in real time and distributes new connection inversely proportioned to response time. For example, a real server with half the response time as another real server will receive twice as many connections. This algorithm is the most self-regulating. The fastest real servers typically can get the most connections over time. As a matter of fact, the response time algorithm is one of the dynamic load balancing algorithms. Besides it, more statistics about real servers can be collected at the virtual access point in real time, such as CPU usage, memory usage, the number of connections, etc. These statistics are analyzed by the virtual access point and the actual capability of each server is determined according to the analysis result. Then the traffic load is distributed according to the calculated actual capability of each real server.

With the weight algorithm, weights are assigned to each real server permanently according to their design capabilities. For example, there are two real servers in a server cluster. If the capability of server1 is half of that of server2, the weight of server1 is 1, and the weight of server2 will be 2. The more weight the real server gets, the more new connections the virtual access point distributes to it. For instance, using the same example, the first two connections go to server2, and the third connection goes to server1. In this fashion, the connections are distributed proportionally to the weight of each real server.

Taking advantage of load balancing algorithms, new real servers can be added to the pool of servers transparently without interrupting access to service.

Resulting Context: The new real server can be added to the cluster seamlessly when the maximum capability of the cluster has been reached. When the new server is ready to accept the request, virtual access point starts distributing the load to it. The virtual access point distributes new connection depending on the adopted algorithm. When using the roundrobin or response time or dynamic algorithms, the new real servers are treated as existing real servers. When using the leastconn or weight algorithms, new connections will be distributed to new real servers until the connections that they carry match the rule of algorithm, namely proportional to the connections carried by old real servers.

When real servers in the cluster have the same capabilities and their capabilities remain equal during the handling of requests, the roundrobin or least connection algorithm can be adopted.

By using the response time or dynamic load balancing algorithms one can provide real time load balancing. They are suitable for real servers whose capabilities change from time to time.

When real servers in cluster have different capabilities, the weight algorithm is the best choice.

The location of real servers can be far away from each other. In some cases, the real servers may be located in difference countries. Thus, the algorithms mentioned in this pattern are not suitable anymore, because the real servers cannot be connected to each other directly or via a back-end network. Instead, wide area network technology is

adopted between real servers. This issue will be discussed in the pattern of geographical load balancing using DNS and proximity by source IP address section.

2.3 Pattern: Persistent Load Balancing

Context: The traffic over socket and connection oriented protocol can be exchanged only after the connection has been established, when balancing it, in addition to using the algorithms in load balancing with pre-defined algorithms pattern, persistency must be guaranteed.

Problem: Any packet over one TCP connection should be forwarded to one real server that initially handled the request, otherwise the connection will be aborted and the client cannot be served. It is also true for some traffic based on UDP. How is persistency achieved?

Forces: When load balancing algorithms, such as roundrobin or least connection are used, the packets from one client may be distributed to different real servers. The TCP connection should be established in advance, thereafter requests and responses are exchanged over this established connection. If any server receives a TCP request or response without a connection being established in advance, it would discard the request or response silently. In this case, no client can be severed.

Even though UDP is a stateless protocol, some applications developed over it also have the requirement of connection persistence. It is called protocol specific persistence. For example, to take advantage of simplicity of the UDP, Wireless Application Protocol (WAP) has been invented over UDP. The WAP message can be in connection mode or connectionless mode. For the connection mode WAP message, UDP connection persistency is obligatory for keeping it working. With load balancing algorithms such as the round robin algorithm, UDP packets from the same source socket may be forwarded to different real servers without taking established connection to the account. Real servers will drop any packet that cannot match an established connection.

In any authenticated web-based application, it is necessary to provide a persistent connection between a client and the web server to which it is connected. Because HTTP does not carry any state information for these applications, it is important for the browser to be mapped to the same real server for each HTTP request until the transaction is complete. This ensures that the client traffic is not load balanced mid-session to a different real server, forcing the user to restart the entire transaction.

In e-commerce web sites, a real server may have data associated with a specific user that is not dynamically shared with other real servers at the site. If the request from this client is not forwarded to the real server with associated data, the transaction cannot go on and the trade will be interrupted.

Solution: The traditional way to achieve TCP or UDP session persistency is to use the source IP address or source socket as the key identifier to do load balancing. The packets

from the same IP address or same sockets are always forwarded to a certain real server within a period of time.

Cookies are strings passed via HTTP from servers to browsers. The cookies can be inserted in HTTP packets and it can be queried to identify the client. The virtual access point can query each HTTP packet and ensure the packet with the same cookie or certain content residing in the cookie is distributed to a certain real server, hence the connection persistency can be ensured.

The SSL session ID is fixed when SSL transactions are running. Persistency based on SSL Session ID can be ensured until the transaction is complete.

Resulting Context: The IP address and port used by client are not changed after connection is established to a virtual access point. Thus, if the virtual access point always distributes the packets from certain socket to a certain real server, the persistency can be guaranteed.

SSL is a set of protocols built on top of TCP/IP that allows an application server and client to communicate over an encrypted HTTP session, providing authentication, non-repudiation, and security. The SSL is adopted by e-commerce web site. The SSL session id is assigned to each client when the request is initialized, and is unchanged during entire transaction. Using SSL session ID, a virtual access point forwards the requests from clients to the same real server to which it was bound during the last session.

2.4 Pattern: Geographical Load Balancing Using DNS and Proximity by Source IP Address

Context: High content availability is achieved through mirroring content at all sites. If one site fails, the others take over the load.

The real servers can be reached through many routers or exchange equipment. In some cases, the real servers may be located in different countries.

Domain name resolution is required before clients sending requests to application servers.

Problem: When the algorithms mentioned in the load balancing with pre-defined algorithms pattern are used, it not only results in latency, but also under utilization of the best performing site. How can this be avoided?

Forces: The real servers are located in multiple physical sites. It is not guaranteed that the request will be sent to proximal real server. For example, there are two real servers; one in Helsinki, the other is in New York. When the roundrobin algorithm is adopted, requests from the client in Helsinki may be forwarded to New York. Obviously it will lead to huge latency comparing with sending this request to the real server in Helsinki. In addition to country, the border of sites can be defined by less bandwidth within the network topology, political restriction, service provider and security measure.

The best performing sites should receive the majority of traffic over a given period of time but they should not be overwhelmed. The performances of sites vary according to their capability and load on them. The performance indicators can be collected over a given time period to balance the traffic, for example, every minute. The period should be adjustable, because if performance indicators are collected frequently, it will increase unnecessary load; if they are collected rarely, it cannot reflect the true situation of each site.

It is possible to configure global load balancing by using complex system topologies involving routers, protocols, and so forth. However, due to the complexity, it makes the configuration difficult to be designed and implemented and hard to maintain.

Solution: Utilize Domain Name System (DNS) and proximity by source IP address to balance traffic.

The domain name, not the IP address is used in the application and all the IP addresses of real servers are recorded in DNS server. When application queries DNS for name resolution, the DNS first checks the source IP address of DNS request, and picks up one IP address of a real server, which is closest to the location of this IP address[1].

In addition to providing to DNS resolution, the DNS can utilize tools to monitor the status of real servers periodically.

Resulting Context: When a client initiates a DNS query, the DNS server can return the IP address of real server closest to the client. With this mechanism, the client can be served by the proximal real server, and there will be no latency during client connection set up[1].

Tools on the DNS server monitor the status of the real servers so that the IP address of real server with lighter load can be returned by the DNS. Thus the best performing sites receive the majority of traffic over a given period of time but are not overwhelmed

Because the DNS server plays an important role in the geographical load balancing, the routers and protocols are not required for load balancing purpose. Even though routers and protocols could be used in this topology, they exist to aim other targets. Thus the topology can be simplified.

Due to the fact that the DNS is only involved in the DNS resolving phase, not in later connection establishment, tear down and transaction processing, the pattern of load balancing with pre-defined algorithms and persistent load balancing are compatible with it.

2.5 Pattern: High Availability of Server In Cluster

Context: In a cluster, when one real server or service on that server is not available for some reason, but the virtual access point still distributes requests to the failed real server or failed service, the client cannot be served.

Problem: How can it be ensured that requests from clients are distributed to running server or service, not to failed one, so that clients can be served properly?

Forces: In a high demand network topology, no device can create a single point of failure for the network or force a single point of failure to any other part of the network. This means that your network remains in service despite the failure of any single device.

In the geographical load balancing situation, if the DNS server returns the IP address of failed real server to client, the further client request to failed real server cannot be handled.

If just one virtual access point exists, the cluster cannot work if it fails.

Solution: The virtual access point monitors the status of real server periodically. It is called a health check. When the real server is healthy, traffic is distributed to it. Otherwise traffic is not sent to it until it is detected to be healthy again. When multiple services are running over one real server, the status of each service is monitored. Once a service on a real server is unhealthy, traffic is merely not sent to this service. This does not affect the traffic to other services in this real server.

To guarantee high availability, put two or more virtual access points in-between clients and real servers. Multiple virtual access points monitor the healthy status among each other.

The DNS monitors the status of real server periodically. When the real server is not available, the DNS server will remove its IP address from the DNS records.

Resulting Context: As one kind of health check, the virtual access point sends requests to all real servers or services. When the expected response comes back, the virtual access point continues to distribute traffic to it. Otherwise the virtual access point stops forwarding traffic. However, when a real server fails between two consecutive health checks, it is likely for a virtual access point to still distribute traffic to failed real server. In this case, the interval between two health checks can be adjusted to improve the above situation. Bear in mind that too small interval may result in performance downgrade of the virtual access point, because virtual access point has to utilize certain resource to carry on the task of health checking.

When geographical load balancing is adopted, the DNS removes IP address of failed server from the DNS record. Thus it is impossible for DNS to return it to client upon DNS query request. Consequently the client request is not forwarded to a failed real server.

Multiple virtual access points could be in the mode of either active-standby or active-active. When using active-standby mode, one virtual access point holds the active flag. The clients only send traffic to the one with the active flag. The other virtual access points monitor the status of active one periodically. When the standby ones detect the failure of the active point, one of them takes over the active flag and starts to handle traffic. When using active-active mode, all virtual access points have activated their own flags. When one of them fails, one of the working access points takes over the active flag from the

failed one and starts to handle traffic on behalf of it.

When multiple virtual access points are adopted, they check the healthy status of each other periodically. The virtual access point may fail between two health checks from other virtual access points. So the interval of health check should be adjusted to avoid a long delay in detecting failure, but also avoid affecting the performance of virtual access point too much. In addition, when one virtual access point fails, the failover happens. Usually the network topology has to be reconstructed. It may take time for events such as reconstructing the network with the spanning tree protocol enabled; this can take up to 45 seconds. During the network reconstruction, no traffic can go through. To minimize reconstruction time, the network topology and protocol used in network should be selected with consideration.

3 Related Patterns

The patterns presented in the paper extend the performance and reliability patterns [2]. The key improvement is that geographical load balancing and high availability are covered by this paper. They are not introduced in the earlier performance and reliability patterns. Additionally, in the solutions of load balancing with pre-defined algorithms pattern and persistent load balancing pattern, more methods and concrete implementations are presented in this paper.

4 Glossary

Cluster: a group of servers or services that act like a single system. In the external world, the clients experience only a unique machine or a unique access point.

Client: the application or user which is using services in a client/server relationship. For instance, when using a browser to surf the Internet, the browser is the client. The computer, which handles requests of a browser and respond HTML pages, is called the server.

Persistency: all of transactions from a certain client are always distributed to a certain real server, to avoid the service interruption.

Real Server: a computer or computer program that physically provides services to clients. It is opposite to a virtual access point.

Session: a series of interactions between two communication end points that occur during the span of a single connection. Typically, one end point requests a connection with another specified end point and if that end point replies agreeing to the connection, the end points take turns exchanging commands and data. The session begins when the connection is established at both ends and terminates when the connection is ended. [4]

Transaction: a sequence of packets exchanging between client and server to satisfy a request so that the integrity of request can be ensured.

Virtual Access Point: the access point for each collection of services. Clients send requests to a virtual access point and are not aware of real server cluster behind the virtual access point.

5 Acknowledgements

We would like to thank Dietmar Schuetz for his excellent shepherding and his patience, and Juha Pärssinen for his valuable comments.

References

- [1] Coyote Point System Inc, Establishing Geographically-Distributed, High-Availability Internet Presence with Coyote Point Envoy, 14.02.2000 [Referred 20.03.2005] <<http://www.coyotepoint.com/pdfs/cpenvoywp.pdf>>
- [2] Microsoft, Performance and Reliability Patterns, 2005 [Referred 20.05.2005] <<http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnpatterns/html/DesLoadBalancedCluster.asp>>
- [3] Mockapetris P., RFC 1035 - Domain names - implementation and specification, 11.1987 [Referred 13.04.2005] <<http://www.faqs.org/rfcs/rfc1035.html>>
- [4] whatis.com, session, 30.06.2004 [Referred 11.03.2005] <http://searchwebservices.techtarget.com/sDefinition/0,290660,sid26_gci541649,00.html>
- [5] Zeus Technology, Seven Myths of Traffic Management, 2004 [Referred 14.03.2005] <http://www.zeus.com/library/white_papers/7_myths.pdf>

Applied MVC Patterns

A pattern language

© 2005 Sergiy Alpaev

mailto: s_alpaev@acm.org

+380 050 342 49 21

Ukraine, Dnepropetrovsk, Mironova str 15

Permission is hereby granted to copy and distribute this
paper for the purposes of the VikingPLoP '2005 conference.

Version 1.0

Abstract

How to get advantages of MVC model without making applications unnecessarily complex? The full-featured MVC implementation is on the top end of ladder of complexity. The other end is meant for simple cases that do not call for such complex designs, however still in need of the advantages of MVC patterns, such as ability to change the look-and-feel. This paper presents patterns of MVC implementation that help to benefit from the paradigm and keep the right balance between flexibility and implementation complexity.

1. Introduction

We state that full-featured MVC implementation as described in [POSA] can be considered quite complex in certain cases or it may lack solutions for some issues in other contexts. For example, distributed applications have some specifics related to handling latency issues, network connection errors, which affect the way we design interactions with user. Such issues are typically beyond the scope of MVC papers. These issues are important part of the context in which we apply MVC pattern.

The statement that classical MVC is hard to apply in some applications today is partially proved by the fact that there is a huge set of various patterns which implement more general paradigm of separation between data, presentation and interaction logic, for example, see Document-View pattern [DOCVIEW], Hierarchical MVC [HMVC], Model-View-Presenter [MVP].

This paper contains several MVC implementation options, which we identified applying MVC in different contexts, such as interacting with remote Services layer, implementing complex Presentation layer for the simple interaction scenarios and others.

2. Scope

This paper covers implementation of traditional MVC as described in [POSA] in the context of information systems. We do not cover implementation of MVC in the context of technologies like J2EE or others although the particular examples of pattern usage may refer to certain frameworks or platforms.

In this paper we do not include the MVC variations, which omit one of the parts of the traditional MVC triad concentrating its responsibilities in some other member of (former) triad such as Document-View pattern (see [DOCVIEW]).

We also do not attempt to analyze all known variations of MVC although we mention quite a lot of them in the pattern descriptions where appropriate. We focus on patterns, which are rather specializations of general MVC paradigm, so the context in which MVC is applied is also an implied part of the context for all patterns listed here.

3. Roadmap

This chapter is a kind of informal table of contents for the paper, which may help reader to jump right to the most interesting pattern combination bypassing the rest of the paper.

The full-featured Model-View-Controller (MVC) triad is composed from a set of design patterns, making it possible to handle quite complex scenarios of user interactions. Nevertheless not all the scenarios actually need the full MVC complexity. We can arrange various MVC implementation options into a kind of a virtual ladder of complexity from the simplest scenarios to the most complex ones, as shown below. All examples and pattern descriptions are written with the assumption that the application is built according to the architecture described in Reference Architecture chapter (see chapter 5).

3.1 Simple data model and one View

Simple data models and lack of multiple views in the application is what often makes people think that applying MVC in this case is overkill. However, in long run keeping strict separation of View and Model has many benefits.

PASSIVE View and CLOSED MODEL patterns may help to keep the balance between implementation complexities of today's use cases and needs for future evolution.

3.2 Applications with large number of similar interaction patterns

To maximize reusing of code that implements similar interaction patterns is the primary design goal for such applications. We need to make the Controller part as common as possible to reuse it in the scenarios where user interactions are common.

ACTIVE VIEW pattern frees the Controller from responsibility of filling the View with data. MODEL AS SERVICES FAÇADE pattern frees the Controller from knowing where the data are taken from. Both patterns used together allow extracting common Controller, which can be reused to handle common workflow.

3.3 Applications with complex interactions with remote Services layer

Certain application requirements may make impossible providing communication with remote services in transparent manner to the user ([NOTEDC]). For example, if application is supposed to communicate over slow connection then latency becomes a serious usability factor. Another issue, which typically affects the way we design interactions with user is network connection errors. In case of connection interruption we often cannot do anything else than suggest the user to repeat the operation later, so speaking MVC language we have to introduce special logic in the Controller which handles interactions with user to resolve network connection problems. DISCONNECTED MODEL pattern addresses separation of responsibilities of interacting with remote Services layer between parts of the MVC triad.

3.4 Applications with complex Presentation layer

The complexity of some applications is mostly driven by the way the domain objects are presented on the screen. If we drop the user interface details from the use case descriptions, and extract high-level abstract scenarios from the use cases, these scenarios will be quite simple. The likely direction of evolution for such applications is increasing Presentation Layer complexity and adding new extensions to the use cases, being related to the way domain objects are presented to the user. To provide grounds for the smooth evolution of the Presentation layer we add direct connection from the View to the Model, see ACTIVE VIEW pattern.

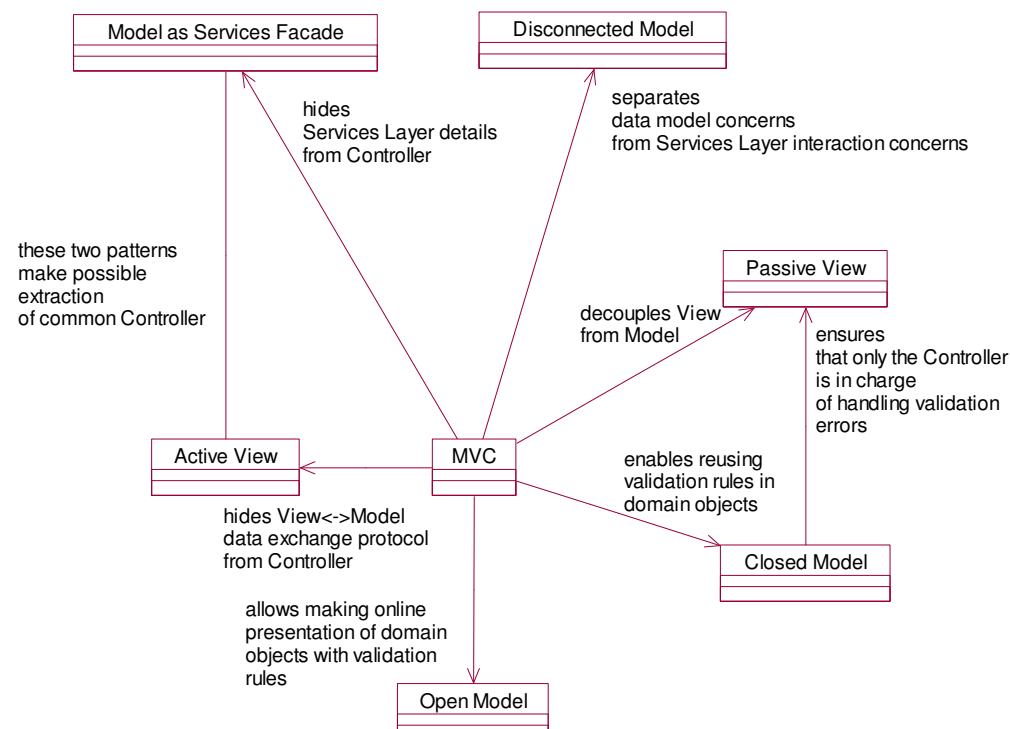
Other patterns that may help in implementation of such applications are USE CASE CONTROLLER (see [UCC]) and MODEL AS SERVICES FAÇADE.

3.5 Applications with complex validation rules and requirements for online viewing of application data

OPEN MODEL pattern relaxes the requirement to keep data in the Model conforming to business rules of an application all the time. This helps to implement quite complex interaction scenarios, for example, having many View instances for the same domain object that show the object data in online mode, and checking complex validation rules on the fly, as the data are being edited.

3.6 Pattern relation map

The diagram below shows relations between patterns presented in this paper. Names near the connections reflect the value, which the patterns bring to each other when implemented together. For example, we may allow extracting of common Controller component for the set of similar interaction patterns using MODEL AS SERVICES FAÇADE and ACTIVE VIEW (see bidirectional connection between MODEL AS SERVICES FAÇADE and ACTIVE VIEW patterns).



4. Considerations

The following are common considerations that were taken into account analyzing the patterns included in this paper.

Separation of responsibilities. The patterns in the paper differ in the way they distribute responsibilities between parts of the MVC triad. Two examples of different ways to distribute responsibilities of interacting with Services layer are DISCONNECTED MODEL and MODEL AS SERVICES FAÇADE;

Automated testing. In certain cases, we may simplify writing automated tests for the parts of the triad by using some of the patterns. The typical examples are

PASSIVE View and DISCONNECTED MODEL;

Common code isolation. Several patterns facilitate extraction and reusing of the common code which implements typical interaction patterns with user, for example: “Do you want to save changes, Yes/No/Cancel”, “Abort/Retry/Ignore” idioms and applications with common “Open/Save changes/Close” user actions;

GUI dependencies. All patterns take into account the need to keep dependencies of application code from GUI framework as thin and as isolated in the View as possible.

5. Reference Architecture

The patterns described in this paper are applied in the context of specific layering scheme. The short description of the layers is given below:

- Presentation layer contains classes, which interpret user actions and present information to the user.
- Services layer defines an application's boundary and its set of available operations from the perspective of interfacing client layers. It encapsulates the application's business logic, controlling transactions and coordinating responses in the implementation of its operations [FOW]. Some patterns in the paper assume that Services layer is physically placed in remote components and accessed through some sort of façade (for example, Business Delegate, see [J2EECORE]).
- Domain Model is an object model, which implements business rules and defines object-oriented abstractions of problem domain [FOW].
- Data access layer makes domain objects persistent.

6. MVC Patterns

Model View Controller

There are so many books and papers, which describe MVC that it is impossible to list them all. Instead, we decided to select description of the pattern given in “Pattern-Oriented Software Architecture. A System of Patterns” (see [POSA] for detailed reference), which, in our opinion, is quite comprehensive for the needs of this paper on the one hand and widely known on the other hand to be considered as commonly accepted description of the

pattern. There are other great sources, which give description of MVC, one of which is GoF book (see [GOF]).

For convenience of reader we include small excerpt of the pattern from POSA book:

| | |
|-----------------|--|
| Context | Interactive applications with a flexible human-computer interface |
| Problem | ... building a system with the required flexibility is expensive and error-prone if the user interface is tightly interwoven with the functional core. This can result in the need to develop and maintain several substantially different software systems, one for each user interface implementation. |
| Forces | <p>The same information is presented differently in different windows, for example, in a bar and pie chart</p> <p>The display and behavior of the application must reflect data manipulations immediately</p> <p>Changes to the user interface should be easy and even possible at run-time</p> <p>Supporting different “look and feel” standards should not affect code in the core of application</p> |
| Solution | <p>The Model component encapsulates core data and functionality. The Model is independent of specific output representations or input behavior.</p> <p>The View components display information to the user. A View obtains data from the Model.</p> <p>Each View has associated Controller component. Controllers receive input, usually as events that encode mouse movements ... or keyboard input ...</p> <p>Events are translated to service requests for the Model or the View.</p> |

Passive View

| | |
|----------------|--|
| Context | <p>Data model of the use case is very simple and is not likely to evolve.</p> <p>View does not have any internal MVC triads that might require accessing Model data bypassing the Controller.</p> <p>The MVC triad is not going to be extended by adding new View types.</p> <p>The mapping between Model domain and View domain is very simple. View does not need to interpret Model data in own way to present them on the screen or this interpretation is common and is not part of application specific logic.</p> |
| Problem | <p>Full-featured MVC triad assumes that View has knowledge about Model.</p> <p>That couples View to Model and unnecessary complicates View</p> |

| | |
|--------------------------|--|
| Forces | <p>Isolation of components from each other. The more components are isolated from each other the easier the application is to maintain.</p> <p>Reusing View. The same view needs to be reused to present different types of data. For example, GUI Widgets that were enhanced for the one application might be useful in other applications. Common dialogs and forms like Microsoft Windows Common Print Dialog are good to reuse too.</p> |
| Solution | <p>Make the View unaware of the Model. Make the Controller responsible for synchronizing the View state with the Model state.</p> |
| Rationale | <p>The View becomes simple translator of Controller calls to calls to the GUI framework. The View also gets completely decoupled from Model and as a result the View gets freedom of speaking own domain language in its programmatic interface; in extreme case the View can be just some GUI control reused from the framework as is.</p> <p>The lack of need to extend the application later with new View types and simple data model are prerequisites for using this pattern. If these prerequisites are not met, we do not get the benefits promised by the pattern because the design is not simplified so much comparing to full-featured MVC when this pattern is applied.</p> <p>When the data model of application is complex and will likely evolve over time, then using the PASSIVE VIEW pattern complicates the design. The Controller is involved in the process of data exchange between View and Model so we will have to touch Controller whenever the data model evolves (for example new fields are added or the structure of entities is changed).</p> <p>When the application has several types of View and is going to be extended with more View types, the fact that the Controller is included in the chain of data exchange between View and Model also plays its negative role. The Controller has to know about every particular type of View to be able to feed it with data (assuming that the views show different types of application data). This makes the task of adding new types of views more complex. The better solution in this case is ACTIVE VIEW pattern that moves the Controller out of the chain of data exchange.</p> |
| Resulting Context | <p>Automated testing</p> <p>Provided that actual View implementation is hidden behind abstract interface to isolate GUI framework specifics from the Controller both Model and Controller can be subjects for automated testing. To do that we will need to provide mock View implementation.</p> <p>Separation of responsibilities</p> <p>View is not responsible for contacting Model to get data anymore, and the only responsibility it gets is to maintain image on the screen. The View and the Model get simple and isolated from each other.</p> |

The Controller becomes less reusable since it is coupled with the View in this pattern. This is typical for classical MVC so we do not lose anything here comparing to MVC.

In extreme cases when the View type is just a class from GUI Widget without any wrapper over it, the Controller becomes coupled to GUI toolkit.

Example Suppose we are designing new GUI control, for example, new fancy edit box which supports entering data by mask. We assume that the design is done in the way that mapping of data entered by user to what is expected by application (ZIP code, for example) is done by application, which is out of the scope of the control. For the control the data are just string conforming to the mask. The data model of the control is as simple as it could ever be.

The logic of interpreting user keystrokes and evaluating them against the mask is placed in the Controller and this is what makes our edit box unique among other edit boxes. This is highly unlikely that we will need to reuse the same Controller with other View types other than edit box (or those, which cannot be implemented somehow as an edit box).

Thanks to all listed above we can simplify our MVC triad eliminating connection between View and Model and making the Controller a bus of all interactions between the two.

That modification simplifies View effectively decoupling it from the Model interface. Evolution of the Model does not affect the View.

With that modification we can use some third-party object as a View instance, for example the View may be implemented as a thin wrapper over Swing JTextField. Since the View is decoupled from the Model, it does not have to implement string representation of edit box content as a member variable. Actually, our View does not need even to speak the language of application, so it does not need method `setString` (unless we want it to be designed that way). For this particular example, the View probably will need something like `setCharAt(int pos, char ch)` method, since the mask will define where the next character will appear and if it will be preceded by symbol from the mask (like ‘)’ for phone number). That makes the View responsibility very narrow and focused (the responsibility is maintaining image on the screen).

Closed Model

Context The user input may violate domain validation rules.

View has own cache of data for presentation and does not request model updates on every keystroke (this is true for most designs of dialogs and forms based on today’s GUI frameworks).

The application use cases do not require showing same data online in multiple views when the views are updated immediately when something is changed in the Model.

Problem Wrong or inconsistent data break integrity of system if not validated prior to

using inside the system.

Delegating validation to Services layer may decrease performance.

Forces **Reusing domain model classes.** It is desirable to reuse domain model classes in the Model.

Reusing of domain model validation rules. Classes that represent domain model of application already have reach validation rules, which are checked in their mutator methods. It is highly desired to reuse this functionality.

Solution Make the Model responsible for validation of the data by triggering the validation in all mutator methods. Treat the validation rules as the invariants of the Model object. One of the options may be to keep an instance of domain object in the Model and delegate all validation to that object making it responsible for keeping itself in a consistent state.

Resulting Context The Model contains valid and complete data that are safe to use by anybody around.

The user cannot enter everything at once so data visible on the screen are typically inconsistent until the last moment (when the last field is filled). The Model cannot do validation until the data form something meaningful and compose something, which can be validated. Typically, the data are ready for validation when user requests application to commit changes. To make sure the data are not passed to the Model until they compose meaningful block the View has to have own cache for the data and keep them until the Controller requests updating the Model. This requirement for the View to have internal cache for the data is a restriction for the pattern. See OPEN MODEL pattern if that restriction makes the pattern inapplicable.

Disconnected Model

Context Services layer introduces new concerns to the application. One of typical concerns is dealing with remote nature of calls to the components, which are typically hidden by Services Layer facade. Other issues are handling concurrency exceptions in client-server environment, timeouts in communications, etc. For interactive applications these issues are important usability factor.

Problem Allocating responsibility of getting data from Services layer to the Model looks natural since the Model owns data in MVC, so it knows best what data are needed at which moments of time (Model is the Expert according to GRASP patterns, see [GRASP]).

However, this makes the Model responsible for two things – keeping data consistent and dealing with Services Layer specifics. This makes the Model code less manageable and complicates support.

Forces **Separation of concerns.** We do not want to mix the code that provides an abstraction of data to display with details of how reading of that data from the Services layer is handled.

Providing safe access to Model accessor methods for the View. We want to make sure that Model does not report any exceptional situations, which might require involvement of user to Views according to the convention described in chapter 7.1, Handling exceptions thrown by Model.

Solution Place the responsibility of interacting with Services layer on the Controller effectively disconnecting the Model from Services layer. Controller gets responsibility of feeding the Model with data.

Rationale You've got an excellent idea and switched back to your mail client application to share the idea with a friend and but application hangs. You are waiting and getting nervous... It wakes up but the idea is gone... One of the reasons why it may happen is that there is the code somewhere in the email client, which does some network operations whenever you have an excellent idea. That code probably has even an evil comment, something like "This code is loading Contacts when user opens window for new message. Since the process of loading contacts from remote server is slow we do not load them until user starts composing new message (see LazyLoad pattern)". This example illustrates that network operations rarely can be designed in the way that they are happening behind the scenes without involvement of user. Even if application does loading of contacts in separated thread, that loading still can end up with network connectivity errors, which may need user attention (or that loading may not finish on time and user has to wait anyway).

The given example shows that while the Model can be considered an Expert (see [GRASP]) in the way how data should be managed it rarely can be good at managing how these data are obtained, since the process of getting data (or saving) in client-server application typically involves interaction with user. The Controller is our Expert in interactions with user. Putting responsibility of interacting with remote services to get data to the Controller is beneficial since this way the Controller naturally becomes a handler of all user interactions related to handling network issues. For our example above, our mail application might give to the user some control over loading contacts in background mode. One of the options might be to load them in asynchronous mode keeping UI responsive (Controller decides if to start loading and instructs Service layer adapter to fill the Model in background) and present some indicator in the UI of the pending loading process (Controller would watch the progress and instruct View to update indicator).

Note that the Controller interacts with remote services by means of some façade so it does not have to deal with any network specifics. The job of the Controller in this case is just interaction with user to resolve problems and handling latency issues from user interactions point of view (for example, instructing Presentation layer to show progress bars).

Resulting context

Separation of responsibilities

Model is responsible solely for mapping domain objects to meet what Views expect to see.

In most applications, this is quite simple a task so the Model also gets very simple.

In addition to standard responsibilities implied by MVC the Controller gets the responsibility of talking to Services layer (to façade over Services layer). That makes Controller more complex and may lead to mixing of concerns because the same component gets responsibility of managing high-level workflow of interaction and workflow of interaction with Services layer. That may be solved by delegating responsibility of handing remote interactions to some other Controller placed on top of Services layer.

Automated testing

Model becomes an autonomous object that does not have any dependencies to other objects. This makes the task of writing automated tests for it quite easy.

If OPEN MODEL is used with DISCONNECTED MODEL (this way, Views do not use domain objects directly) we may need to make some precautions in order to prevent invalid data from being given to Services layer without proper validation. The Model (which knows when data are valid and when they are not) in OPEN MODEL pattern is allowed to have invalid data and it cannot guarantee that Controller takes them after proper validation.

One of the options may be drawing a clear boundary between interfaces of the Model designed for Controller and those designed for Views. Model methods of the Controller interface should perform required validation before returning data to the Controller. In its turn, the Controller should expect validation errors reported from the methods of this interface and be ready to handle them.

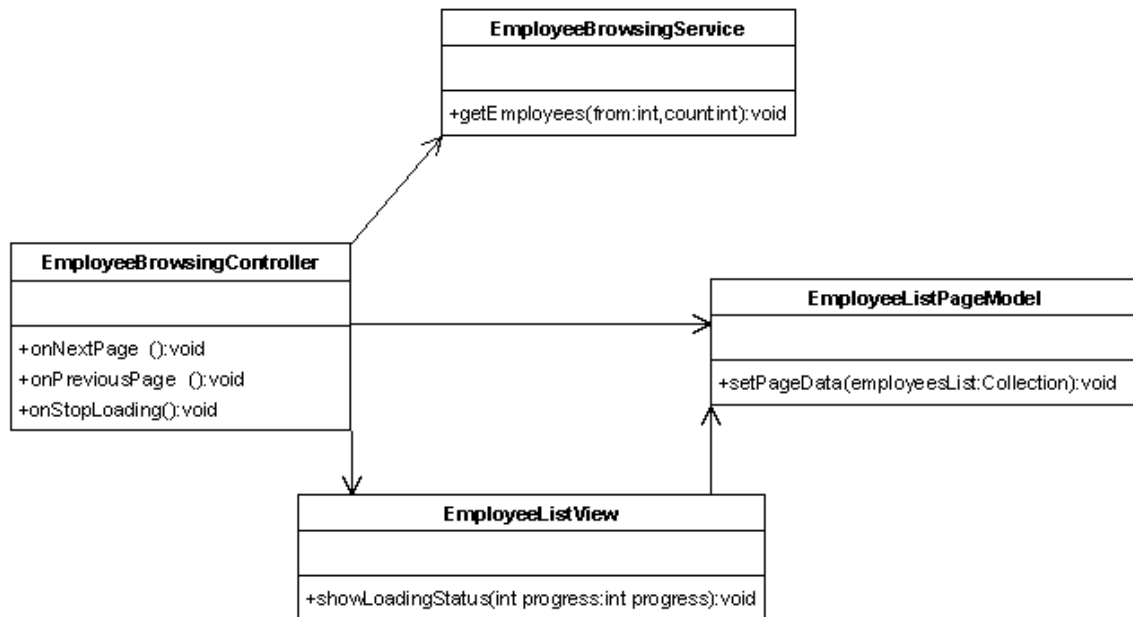
Example

The dialog that shows some large sets of data in page-by-page manner may be a good candidate for applying the DISCONNECTED MODEL pattern.

As an example, let's use a scenario for the application: the dialog that shows a long list of employees expecting user to select ones for a project in some Human Resource (HR) Management system.

Since the data set is large, we usually do not want to keep everything in memory. Instead, the data are read on demand. As we noted, reading data on demand can be rarely considered a private implementation detail of Model object since it involves interactions with user.

The solution with DISCONNECTED MODEL pattern is presented on the following figure.



Controller keeps track of what was already read and calls Services layer (represented by `EmployeeBrowsingService` object) when required.

Model as Services façade

- Context** Use cases of application have many standard interaction patterns, which differ only in the types of the objects that are manipulated by user. For these standard interaction patterns the logic of handling exceptional situations, which require involvement of the user, is also standard and the same for every use case.
- Interaction with remote Services layer does not require dealing usability issues caused by remote nature of communication such as overcoming network delays by using asynchronous loading of data.
- Problem** Allocating responsibility of interacting with Services layer to Controller has some advantages, for example, the Controller is naturally becomes handler of exceptional situations caused by Services Layer if those exceptional situations require involvement of user. However, such a distribution of responsibilities makes all three parts of the triad dependent on types of domain objects being manipulated by the triad and prevents reusing the Controller for all typical interaction patterns listed in the Context.
- Forces** **Reusing Controller.** We want to reuse the Controller in all MVC triads that implement the same scenarios and differ only in types of objects being manipulated (Model) and the way they are presented (View).
- Solution** Make the Model responsible for fetching data from Services layer. Apply ACTIVE VIEW pattern to isolate Controller from knowledge about the types of objects being manipulated by the triad.

| | |
|---|--|
| Rationale | <p>To maximize code reuse we need to hide variances between different MVC triads that handle typical interaction patterns so that common code can be isolated and parameterized to handle a particular use case.</p> <p>According to Context of the pattern, the only difference between interaction scenarios of an application use cases is the types of domain objects, which are handled in use case steps. We hide these variances behind the Model, making it responsible to know the type of domain object and Services layer interface used to obtain it. This way we concentrate variable part of the use case scenario in two components: the Model (which knows type of object and source to get it from) and View (which knows how to present the object to the user). As a result, the Controller has only one responsibility, and namely that one of handling common interaction patterns; it can be reused with other Model/View pairs.</p> <p>The Model is the Expert (from GRASP patterns, see [GRASP]) in knowledge which data to get since it holds the data.</p> <p>Note that the Model becomes a Façade over Services layer only in context of given MVC triad meaning that the Model is the one among other members of the triad who takes over the responsibility of contacting Services layer and isolates other members from knowing specifics of Services layer. That does not mean however that the Model is the only façade for Services layer in the context of whole application. The Model does not have to contact Services directly; it can (and should) be done through application level Services façade.</p> |
| Related Patterns Resulting Context | <p>This pattern is used together with ACTIVE VIEW to achieve isolation of Controller from domain model.</p> <p>Common code isolation</p> <p>The Controller becomes independent from the data types exchanged between Views and Model. This makes it possible to reuse Controller in order to handle all these similar use cases. For every use case MVC triad is composed from common Controller and View/Model pair unique to the use case. To make it work all the Views and Models have to conform to common interfaces for View and Model respectively.</p> <p>Separation of responsibilities</p> <p>Model gets responsibility to contact Services to get data when needed. As it is noted in chapter 7.1 (Handling exceptions thrown by Model) it is quite important to make sure that Views do not get exceptions reported by Services to Model on behalf of View calls. To allow this the Controller might need to make sure that the Model loads all required data prior to receiving any call from View.</p> |

Controller has responsibility to handle exceptions thrown by Model when the Model contacts Services. If the requirements to handle exceptions differ from scenario to scenario, the task of making reusable Controller gets more complex. We suggest applying this pattern in simple scenarios where all interactions with user are similar (see Context chapter of the pattern) including exceptions handling.

All knowledge about types of the data needed to serve the Views, and how the data are retrieved is encapsulated in one object (Model). That simplifies other parts of the triad.

Automated testing

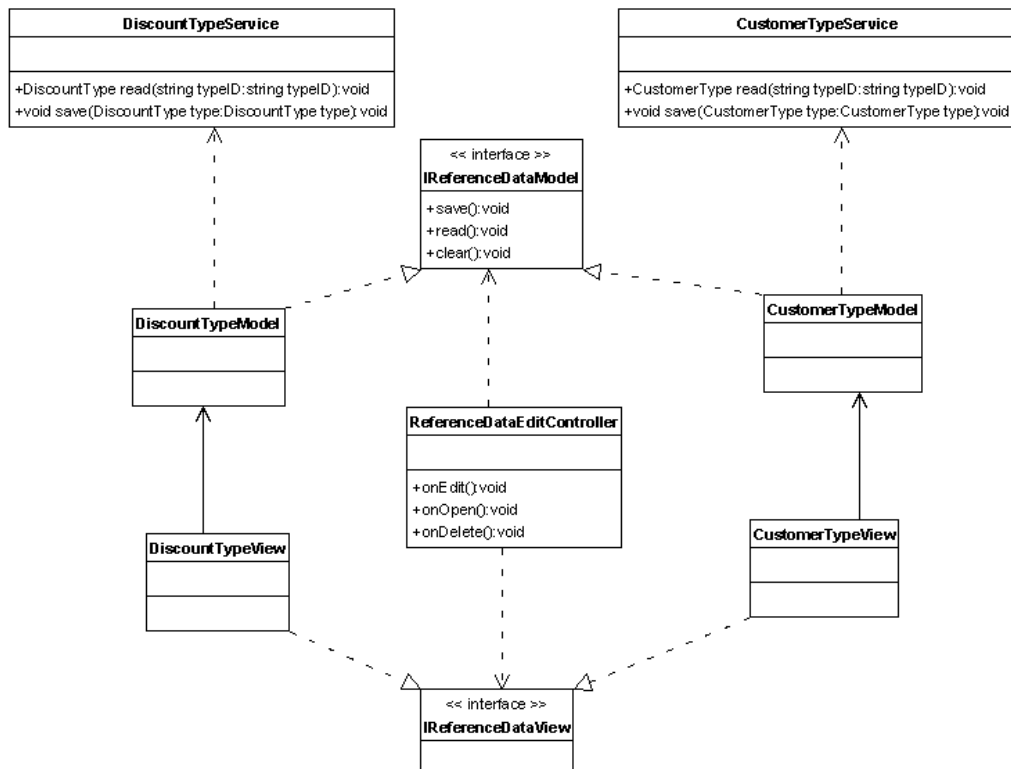
To test Controller we will need to provide mock View and Model.

The pattern implies that we have abstract interfaces for Model and for View so introducing mock View and Model does not require changes in application code to extract interfaces.

Example

Information systems typically work with two types of data: operational data such as orders and dictionary data such as descriptions of customer types, types of discounts available etc. These rarely changing data define initial setup of the system for particular enterprise. Typically, the workflow of editing this kind of data is common for all types of reference data. It may look like simple sequence of steps such as the following: open the entity, make changes, save the entity. In large information system we may easily have up to hundred types of reference data types.

The solution with Model as Services façade for a simple application that manages types of discounts and types of customers is shown below. Note that one type of Controller is used to serve both types of reference data.



The Model and the Views take the knowledge about the particular type of entity out from Controller making the Controller common for all entities.

Active View

| | |
|------------------|---|
| Context | <p>Application manipulates complex domain objects but high-level workflow of that manipulation is simple. There are many extensions of main simple use case related to the way the domain object is presented on the screen.</p> <p>The primary driver of complexity of application use case is the complexity of domain objects and the way they are presented to the user. Most likely, the future changes in application will affect the way the complex domain objects are presented to the user or the domain model itself while the high-level use case scenarios remain stable.</p> <p>Complexity of presentation will be handled by internal Presentation layer controllers, which may need to access the Model.</p> |
| Problem | <p>Traditionally MVC assumes that the Controller is involved in interaction between Model and View. If that interaction is designed in the way that Controller becomes dependent on the domain model of application then the evolution of the domain model affects the Controller. Changes in the domain model may break the implementation of high-level use case workflow concentrated in the Controller.</p> |
| Forces | <p>Separation of concerns. We want to keep implementation of basic use case workflow separated from complexity of the Presentation layer so the two may evolve independently.</p> <p>Reusing Controller. We want to reuse main use case Controller with other Model/View pairs</p> |
| Solution | <p>Allow Views to talk directly to the Model to fetch data they need to display hiding this way the knowledge about domain model from the Controller and concentrate it in the Model and View. Delegate the responsibility of transforming model data into data used for presentation to the Views.</p> |
| Rationale | <p>The complexity of Presentation layer in some applications may require adding internal MVC triads to top-level View objects handle presentation options of complex domain objects. For example, some parts of these objects may be initially hidden to show later by user request. Interaction scenarios may include quite complex steps such as drag&drop to adjust some characteristics of domain objects, or launching wizards to perform complex tasks.</p> <p>Internal MVC triads that implement these extensions of main workflow require access to the Model.</p> <p>Allowing main MVC View object to access the Model directly simplifies future evolution of View object. In this case adding new internal MVC triads will not affect the main Controller.</p> |

Related Patterns

ACTIVE VIEW decouples Controller from domain model if used together with MODEL AS SERVICES FAÇADE.

However, it can also be used on its own to enable smooth evolution of Presentation layer if reusing the Controller is not important.

Other patterns which enable smooth evolution of Presentation layer and/or reusing common controllers are following:

USE CASE CONTROLLER pattern suggests implementing the Controller very close to the use case abstraction level greatly simplifying maintenance and evolution of application, see [UCC].

HIERARCHICAL MODEL VIEW CONTROLLER pattern breaks application into many MVC triads that have their controllers linked together in hierarchical manner. See [HMVC].

Resulting Context

Views are free to evolve as dictated by new requirements for Presentation layer without affecting high-level workflow of the use case.

Extensions of main use case pertaining to the presentation options are handled at a proper layer of abstraction.

Views become coupled to the Model.

Automated testing

Interactions between parts of the triad are more complicated which in turn complicates the task of writing automated tests. This is the price to pay for the benefits described in sections Separation of responsibilities and Common code isolation section below.

Separation of responsibilities

View gets responsibility to get data for display from the Model and to convert it into form suitable for presentation. As a result, View and Model completely hide the protocol used to feed the View with data and to deliver data entered by user in the View back to the Model from the Controller.

Common code isolation

The way in which responsibilities are distributed between parts makes Controller fully independent from the types of domain objects that are manipulated by the use case scenario. This allows extracting Controller classes that are common for many interaction patterns.

Example

For one of the examples of using ACTIVE VIEW to satisfy needs of reusing common Controller please refer to MODEL AS SERVICES FAÇADE pattern description.

The following example shows how we can enable smooth evolution of domain model and implementation of low-level use case extensions related to presentation of domain objects by separation of high-level use case workflow using ACTIVE VIEW pattern.

Suppose we have use case for our new Human Resource (HR) Management system given below:

Assign an employee to the project

| Actor action | System response |
|---|--|
| HR Manager specifies a skill set necessary for a project and requests a matching employee from the system. | The system presents employees who match the specified skill set and not engaged in other projects. System presents for every employee the following information: Temperament, Leadership skills, Communication skills. |
| HR Manager makes a decision to assign an employee to the project. HR Manager considers temperament, team playing abilities, leadership skills, and communication skills. HR Manager specifies a project name and confirms the assignment. | The system confirms that the employee has been assigned to the project. |

Let's assume that we have that use case already implemented and our system allows basic selection and assigning employees to the projects. Our HR department continuously works on improving the process of team building, so we feel that in the future the system will have to provide more sophisticated scenarios of assigning people to the projects and that would be the most likely direction of the system evolution.

Let's simulate this kind of evolution by stating new requirement for the system; for example, let's assume that in new version the Project Manager has to be able to select employees using history their past achievements in addition to factors listed in the use case.

Note that the basic workflow of employee assignment is the same sequence of specifying the criteria for selection of an employee, making the decision and confirming the assignment. This high-level scenario is not changed. What was changed is the way how "HR Manager makes a decision". New version of system affects the way HR Manager makes a decision by providing more information.

New requirement causes changes in domain classes (very likely) and in the way it is presented on the screen, (new fields are added).

The high-level workflow described above remains constant. Therefore, it is beneficial to have this constant workflow segregated in separated component, which is the Controller as required by MVC.

To achieve that separation we need to hide the part, which is subject for future changes from the common controller. In our case, the variable part is domain model and presentation logic of that model. To concentrate that variable part in the Model and View we delegate to the View responsibility of contacting Model to get data for display making Controller free from knowledge about data exchange protocol between Model and View. Model and View implement PROTECTED VARIATIONS pattern (see [GRASP]). Controller is responsible only for sending messages to the View and Model that these two may consider worth to start doing data exchange. However, the particular reaction on these messages is implementation detail of View and Model.

By moving Controller out of data exchange protocol between View and Model Active View pattern enables evolution of domain model and Presentation layer without touching high-level use case implementation.

Open Model

| | |
|--------------------------|---|
| Context | <p>The user input may violate domain validation rules.</p> <p>Application use cases require having several windows or views showing the same data in different forms in online mode when changes in one view are immediately seen in other views.</p> <p>User cannot enter everything at once; therefore, the data will be incomplete at some points of time and probably violate validation rules.</p> <p>MVC assumes that Views take data from the Model; that requires keeping incomplete data which potentially violate validation rules in the Model</p> |
| Problem | <p>Wrong or inconsistent data break integrity of system if not validated prior to using inside the system.</p> <p>Delegating validation to Services layer may decrease performance.</p> |
| Forces | <p>Eliminating data duplication. We would like to avoid creating copies of data in View to be able to show partially entered data (while keeping old valid copy in the Model). In this case showing the same data online in several View instances requires complex synchronization between View instances.</p> |
| Solution | <p>Do not trigger validation from mutator methods of the Model. Postpone the validation of the Model data until Controller explicitly requests it.</p> |
| Resulting Context | <p>Model becomes a snapshot of data shown in the Views and therefore may contain incomplete or partially entered data. Views display the data taken from the model as is.</p> <p>Since the Model can contain invalid data (invalid from business logic point of view) the domain objects cannot be used as Model data exposed to the Views because domain objects are not allowed to violate validation rules. Instead, the Model has to have some intermediate data structure that represents projection of domain object to the Views. That data structure is not required to conform to validation rules all the time.</p> |

Separation of responsibilities

With OPEN MODEL we relax requirements for the data to always conform to validation rules of an application. Therefore it is important to make sure the invalid data are not propagated outside the triad and do not cause errors. Who requests validation from the Model and when it is done depends on the pattern of interaction with Services layer.

If DISCONNECTED MODEL is used then Controller should request validation of the data from the Model prior to feeding them to the Services layer. Alternatively, the Model may expose special interface for the Controller that always triggers validation before returning the data.

If MODEL AS SERVICES FAÇADE is used then the Model is responsible for making sure the data are valid prior to giving them to the Services layer.

Example

Microsoft Excel is a good candidate for implementation of this pattern.

When we enter some invalid value in the cell, lets say, incorrect formula like “=()” Excel shows a message box describing the problem. When you press OK button, Excel selects the incorrect text to let you correct the problem. If we have two Excel windows open for the same sheet (you can use Window/New Window menu item) Excel shows illegal content (“=()”) in both windows. This way you can use either window to correct the mistake. This may be useful with long data sheets that do not fit one screen. For example, user may edit formula in one window while using the other one to locate the dependent values required to correct the mistake.

One of the ways to implement the Model component for such scenario is to store cell values in the Model. The Model should be able to store illegal cell values. Otherwise, it will not be possible to show the same (illegal) value in two windows, showing the same sheet.

Functionality of “Window/New Window” menu item makes Excel eligible for OPEN MODEL pattern since this feature allows having several views to show the same content; and that content may not always be correct.

7. Cross-cutting concerns

7.1 MVC triad level of abstraction

The way MVC is implemented is significantly transformed when we consider MVC applied within the context of different application layers.

For example, implementing fancy GUI controls often require non-trivial ways of using the GUI framework. For example, to add drag&drop features to .Net DataGrid control columns Microsoft recommends overriding painting method of the control and use native calls to take screen shot of a column (see MSDN library, article “Dragging and Dropping DataGrid Columns” [D&DDGRID]). If we used MVC pattern to implement all that we would need our Controller to intercept all mouse movements, which is quite low-level intrusion into GUI framework code. The View needs to be done also on a quite low-level, since truly impressive effects often are achieved by overriding low-level things such as calling native

code. The View part of MVC in GUI controls is usually coupled to GUI framework since this is prerequisite for using low-level APIs and GUI framework internal backdoors such as mentioned native calls.

However, the Controller that implements workflow of order submission in airline tickets reservation system looks quite different. At this point, all low-level mouse movements and native code calls are already encapsulated by low-level controls, so this Controller can be implemented on a pretty high level of abstraction. This Controller talks the language of application domain (see [UCC] for example); it can freely use quite abstract language in the commands to Views, such as `switchToReadOnlyMode`, which may hide quite complex logic in the implementation part. The View in this triad is high-level component, which may even be fully isolated from GUI framework by set of wrappers if needed.

Although the two MVC triads shown above are implemented according to the same paradigm of separating between data, presentation and control parts, they are very different in nature. The first one, which handles DataGrid column drag&drop functionality, is implemented on a low-level; the second one can be considered a high-level implementation of a single application use case.

It is important not to mix levels of abstractions in the same MVC triad. The Controller, which implements high-level use case workflow and handles low-level mouse movements to interpret user gestures at the same time, is hard to support since these two things rarely change together. It is better to delegate interpretation of mouse movements to low-level MVC triad composing a hierarchy of controllers.

There are several strategies of separating MVC triads into some sort of hierarchy. The following is short overview of two of them:

HIERARCHICAL MODEL-VIEW-CONTROLLER suggests chaining MVC triads making every triad responsible for one single aspect of application, for example handling one View instance (see [HMVC]). The pattern suggests that every MVC triad corresponds to one View instance and application is broken according to hierarchy of View instances nested into each other.

USE CASE CONTROLLER pattern (see [UCC]) suggests making Controller responsible for handling use case workflow. The Controller in this case is done on a high-level and its implementation is directly traced to use case description. This way the controllers in the application are linked together according to use case extension and inclusion relationships. The relationships between View instances do not have any direct effect on links between controllers.

7.2 Handling exceptions

This chapter describes a problem, which is important to consider when we implement MVC-based design.

The Model may throw exceptions from the methods. Handling of some of these exceptions requires involvement of user to make a decision what to do in the situation (exceptions that do not involve user in handling are not considered here.). Two typical reasons for this kind of exceptions are validation errors, occurring when clients of the Model try to modify data in the Model in the way that violates business rules and exceptions reported by Services layer, if the Model contacts Services Layer to implement some of its responsibilities.

It is important to allocate responsibility of handling exceptions reported by Model to proper part of the triad if the logic of handling exceptions requires interaction with user.

The problem appears when the exception is thrown by the Model as a result of a call made by View. The View cannot handle this exception since it does not have the required knowledge how to interact with user to solve the problem (the Controller has it).

View might simply delegate the handling to Controller but it requires dealing with the problem of code duplication between many types of View. Even if we have an elegant solution for that, (we may extract common code into some base class for example) this solution introduces cyclic dependency between View and Controller, which further complicates the design.

Ideally, our MVC design should simplify the task of making the Controller responsible for handling exceptions as much as possible. The following sections recommend some solutions depending on a pattern selected.

Model as Services façade, Open Model, Closed Model patterns

One of the options is not to throw any exceptions that are supposed to be handled by user from the methods designated for Views. One of the ways to achieve that is to avoid contacts with Services Layer from the methods that are called by Views. In this case, the Model should have all the data ready before the Views can contact the Model. Separation of interfaces for the Controller and for the Views may clear up the code in this case.

Active View

ACTIVE VIEW pattern also requires some solution for this problem because the View contacts the Model directly by design in this pattern. One of the options is to disallow View to call mutator methods on the Model. Instead, whenever the user makes any request to do data modifications the View should forward this call to the Controller in form of event using OBSERVER pattern. This way the contract of the View with clients (what we usually call an interface) consists of two components – interface of the View class and set of events it fires.

Disconnected Model

DISCONNECTED MODEL pattern eliminates the problem completely delegating the task of talking to Services layer to the Controller.

Passive View

PASSIVE VIEW pattern also does not have this problem because all the interactions between View and Model are happening through the Controller.

8. Acknowledgements

We would like to express my special thanks to Nelly Delessy for shepherding the paper and providing excellent and valuable comments.

Our special thanks to Viktor Sergienko for valuable comments, which helped making the first reshaping the paper contents from cover to cover.

In addition, we would like to thank the audience of Design Patterns seminar sponsored by Intel, Russia, for raising issues and questions during the seminar, which helped refining the ideas described in the paper later on.

9. References

- [UCC] Ademar Aguiar, Alexandre Sousa, Alexandre Pinto, Use Case Controller, <http://hillside.net/patterns/EuroPLoP2001>, EuroPLoP 2001
- [FLA] Four Layer Architecture <http://c2.com/cgi/wiki?FourLayerArchitecture>, <http://c2.com/cgi/wiki?FourLayerArchitectureDiscussion>, 2004
- [FOW] Martin Fowler, Patterns of Enterprise Application Architecture, Addison-Wesley Professional, ISBN: 0321127420, 2002
- [POSA] Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P., and Stal, M. 1996 Pattern-oriented software architecture: a system of patterns. John Wiley & Sons, Inc.
- [GRASP] Larman, C. 2001 Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and the Unified Process (2nd Edition). Prentice Hall PTR, ISBN:0130925691
- [HMVC] Jason Cai, Ranjit Kapila, and Gaurav Pal, HMVC: The layered pattern for developing strong client tiers, JavaWorld, http://www.javaworld.com/javaworld/jw-07-2000/jw-0721-hmvc_p.html#resources
- [DOCVIEW] MSDN Library, http://msdn.microsoft.com/library/default.asp?url=/library/en-us/vccore/html/_core_Document.2f.View_Architecture_Topics.asp, 2005
- [MVP] Potel, M., MVP: Model-View-Presenter The Taligent Programming Model for C++ and Java, IBM developerWorks, <http://www-128.ibm.com/developerworks/java/library/j-mvp.html>
- [GOF] Gamma, E., Helm, R., Johnson, R., Vlissides, J., 1995 Design Patterns: Elements of Reusable Object-Oriented Software, Addison-Wesley Professional, ISBN: 0201633612
- [NOTEDC] Kendall, S., Waldo, J., Wollrath, A., Wyant, G., 1994 A Note on Distributed Computing, <http://research.sun.com/techrep/1994/abstract-29.html>
- [AGGMVC] <http://c2.com/cgi/wiki?ModelViewControllerAsAnAggregateDesignPattern>
- [C2ONMVC] <http://c2.com/cgi/wiki?ModelViewController>
- [D&DDGRID] MSDN Library, http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnwinforms/html/dragdrop_datagrid.asp, 2005
- [DDD] Evans, E. 2003 Domain-Driven Design: Tackling Complexity in the Heart of Software, Addison Wesley Professional, ISBN: 0321125215

Patterns for ERP-Landscapes

Florian Humplik, Peter Leitner, Wolfgang Zuser, Thomas Grechenig

Industrial Software, Vienna University of Technology

{firstname.lastname}@inso.tuwien.ac.at

Abstract

This paper presents a pattern language with the objective to design, implement and maintain interdependent IT-systems from different vendors in the business context (so called ERP-Landscapes).

Figure 1 summarizes the patterns presented within the paper and their interdependencies. They share the need to cope with heterogeneity. Heterogeneity in ERP-Landscapes has been induced by the on-going specialisation of applications in the business context (applications specializing in diverse fields like personal accounting, legal consolidation, inventory management, cost accounting and controlling).

Sooner or later every business application has to interface with the input or output of other applications. From that moment you have to deal with the problems associated with an ERP-Landscape. Considering such a setting at design time will improve the adaptability of your applications. This paper characterizes different issues, which have to be considered when developing business applications in the context of an ERP-Landscape. The pattern language reflects a holistic approach showing the need to integrate both business and IT domain knowledge. Emphasizing on one aspect will decrease the over-all quality of the solution.

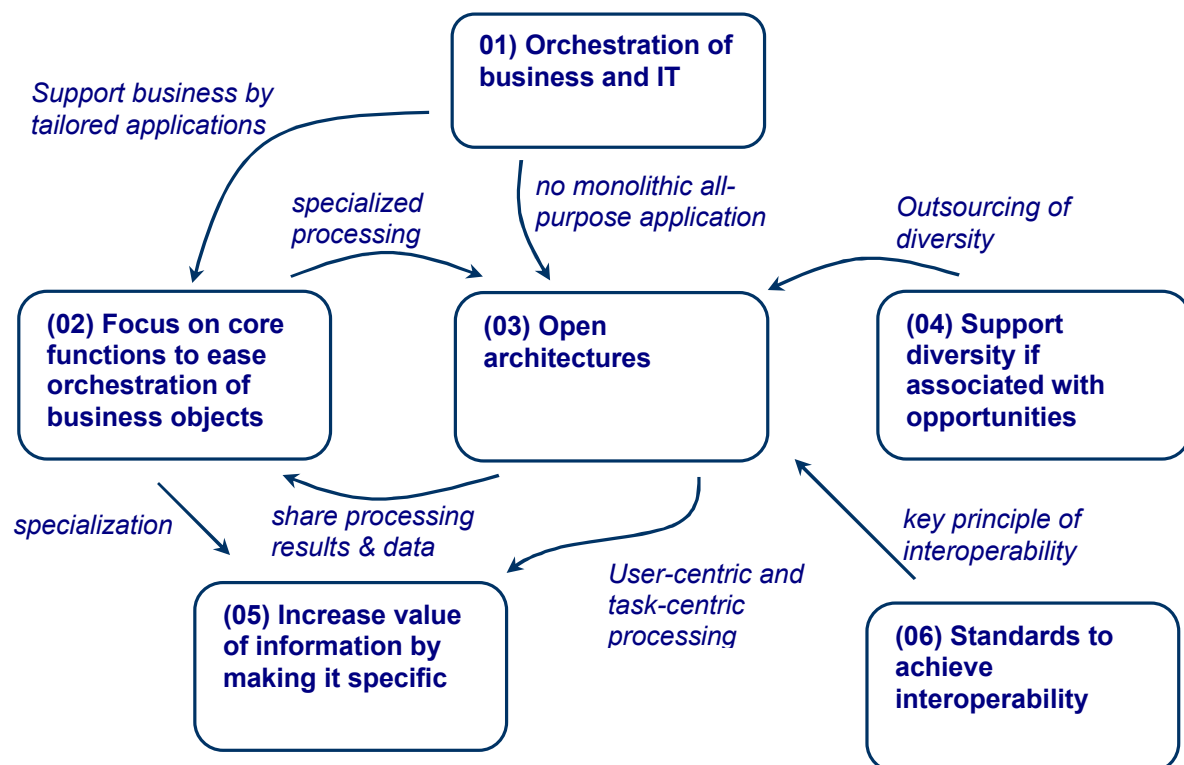


Figure 1: Pattern Map

Audience

The intended audience are people who are involved in the design and implementation of IT-systems within the business context. Both business people and technicians can benefit from the pattern language. Reading of the paper requires basic knowledge of both the business and the IT-domain. The whole paper is guided by the idea to increase understanding for people belonging to the 'other' domain.

Content

| | |
|---|-----------|
| PATTERNS FOR ERP-LANDSCAPES..... | 1 |
| Abstract..... | 1 |
| Audience | 2 |
| Content..... | 2 |
| Introduction..... | 3 |
| (01) Orchestration of business and IT | 5 |
| (02) Focus on core functions to ease orchestration of business objects..... | 8 |
| (03) Open architectures | 11 |
| (04) Support diversity only if associated with opportunities | 14 |
| (05) Standards to achieve interoperability | 16 |
| (06) Increasing the value of information by making it specific | 18 |
| ACKNOWLEDGEMENTS | 20 |
| REFERENCES | 21 |

Introduction

This paper wants to improve the process of integrating heterogeneous ICT-systems (information and telecommunication systems) in the business context. The intended audience are people concerned with the development and integration of software systems in the business context. The work has been motivated by the different types of heterogeneity we have encountered within ERP-landscapes such as:

- heterogeneity in data providing the basis for business processes
- heterogeneity in business processes covered by the systems
- heterogeneity among the stakeholders and their information requirements
- heterogeneity in opinions and assumptions of systems users
- technical heterogeneity among server and client applications

A major competitive factor in today's economy is optimised business processes. A prerequisite is efficient support by ICT-systems. Thus, many companies are required either to introduce or renew ERP-systems (enterprise resource planning systems). But first we have to clarify the terms ERP-system and ERP-landscapes:

- ERP-system: Any software that can be perceived as a model for the enterprise or at least for some of its aspects (processes and resources). Such software is designed to utilize or administrate a company's resources.

We are considering software and solutions in the following fields: finance, accounting, enterprise planning and control inventory systems, supply chain management, CRM (customer relation management) solutions, timekeeping systems, production management, etc. Of course this is a broad range of applications. Hence, the definition covers almost any kind of software used in the business context.

- ERP-Landscapes: The applicability of (business) off-the-shelf software is limited. As a result companies find themselves running and maintaining multiple applications. This creates the necessity to integrate the various software systems (either with automation or manual processes). We refer to interdependent ICT-systems within a company as an ERP-landscape emphasising the underlying principle of heterogeneity.

The inherent complexity in business processes is closely related to considerable costs. Installing an ERP-system generally is a large budget item. However, successfully establishing an ERP-system may increase a company's competitiveness significantly. In contrast, failure means lost investment. Vital, competitive companies are characterized by steady growth. The long lifecycle of an ERP-system demands thorough consideration of post-introduction adaptations, extensions, and modifications.

In the following, we describe representative patterns in ERP-landscapes. The patterns serve as a guideline to deal with some of the most common and most serious problems during the design phase of ERP-systems.

Most of the considerations shaping the design of ERP-landscapes deal with some sort of system extensibility. The extensibility of an IT-system is determined by a lot of decisions at design time. Therefore the paper clearly focuses on architectural issues at design time.

Figure 1 illustrates the situation software engineers typically encounter. The small red box (striped) represents the ERP-system that more or less reflects the complexity of the socio-technical system of an enterprise. The illustration consists of the following elements:

- Circles represent various systems. Intersection means both systems share elements.
- The distance to the circle “enterprise” illustrates how close the relation is and might be considered as an indicator for the interdependency of any two systems.
- The thickness of a line connecting any two systems approximates frequent interaction.

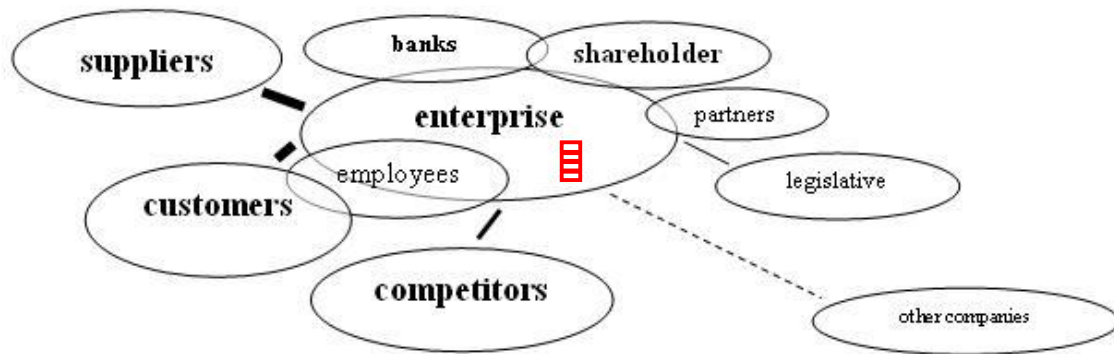


Figure 2: The enterprise as a socio-technical system

Technical problems are part of day-to-day business. However, the real challenges usually result from interactions with the various stakeholders and environments. The patterns presented in this paper address problems in a context illustrated in Figure 1.

The pattern language has been developed based on an XML-document (using XSL and CSS). The intention is to transform the original document into multiple formats such as HTML and PDF. We have employed a pattern form that has been inspired by multiple authors:

- The chronology and basic structure is based on the form used by Alexander in ‘A Pattern Language’.
- Apart we have introduced a set of headings to ease reading and provide more structure which has been inspired by the work of Jim Coplien.
- The section ‘related patterns’ has been inspired by the GoF-form. That section is intended to account for hierarchical dependencies among the patterns.

(01) Orchestration of business and IT

Alternative name:

- Align business with IT and vice versa

Companies introduce or renew ERP-systems to achieve some kind of improvement. But the transformation of real-world business processes is not obvious. At the start of the project the actual processes are documented and provide the basis for all further steps.

Context

At the beginning of a project, there are three sources of information concerning business processes:

- The normal experience of day-to-day business operations
- The project team member's experiences, e.g. from previous projects
- The reference-models referenced within literature

The requirements defined by the customers usually reflect the current business processes.



Problem

How to discover the requirements of running efficient business processes in order to realize the benefits of modern ICT-systems?

Forces

- **Expertise in business administration vs. expertise in ICT**

Companies lacking expertise in the field of IT usually focus on their day-to-day 'normal' experiences during the requirements phase. Therefore the full range of benefits offered by modern ICT-systems is not considered when generating requirements and solutions, e.g. new types of communication, sophisticated processing, support of business processes and user interaction, etc. The same is true in reverse. Even well established processes have to be aligned with the requirements of ICT-systems to work efficiently.

- **Resistance to change vs. flexibility and dynamic organisation principles**

The changes associated with the introduction of ICT-systems affect the company's employees. Therefore their opinions and feelings (e.g. fears) have to be taken into account. Keeping things as they are avoids possible conflicts.

- **Established and proven processes vs. optimized but newly introduced processes**

Changing well established processes is usually difficult to achieve. It is almost impossible to question well established processes without questioning the people in charge. It is easier to support change if the introduction of an ICT-system provides a rational explanation.

- **Established and proven processes vs. mature IT**

Room for improvement results from deficits in IT-experience or complex chains of responsibility rooted in a company's organisational structure. IT-projects provide the opportunity of transferring know-how and thus, enable the persons in charge to see things from a different perspective. This know-how represents the key factor to use the full advantages of IT and thus, enables a successful redesign of business processes.

- **Usage of pre-defined processes vs. cost-intensive renewal**

It is crucial for IT-experts to become familiar with a company's specific business processes. Common practice relies on simple records of customer needs and subsequent transfer to an appropriate software solution. The orchestration of customer problems and the technological opportunities are usually not fully exploited.

- **Short term gain vs. long term value**

Today, almost any business is short term-oriented, e.g. on quarterly reports. Success or failure is determined by the change in figures (either comparing two subsequent periods or the same period in two subsequent years). However, the positive effects rooted in process reengineering take some time to be implemented and are likely to be discovered only after some time lags. Well-engineered processes can be expected to be more effective than historically grown processes.

Solution

Focus on business process (re-)engineering (BPR) during the requirements phase of the project.

The considerable costs associated with the introduction of an ERP-system justify additional budgets during the analysis and design phase. Spending extra time in the beginning is proven to generate additional value for the company. Questioning the processes underlying an ERP-system may improve overall process quality and at the same time provide an opportunity to tailor processes to the needs of IT-systems. The idea of such an approach is to question information presented by the customer to prevent choosing suboptimal design decisions.

Resulting Context

The proposed approach ensures plurality both in ideas and involvement. Questioning customer requirements also means taking more responsibility and therefore increases workload for ICT-professionals. BPR is not without cost and usually does not achieve short-term cost effectiveness. Therefore such an approach requires both additional budgets and support of upper management.

Known uses

- A real estate company was accustomed to manually maintain reminder lists to keep track of landowners' payments. Each list held one entry per property and month. If a payment was received then the corresponding entry had to be marked with a green checkmark.

After a new IT system had been introduced the employees kept on maintaining those lists manually. The only improvement: the lists were generated by a computer. This improvement saved a lot of time and costs.

However the software could have easily provided the final list of open payments, thus making the time consuming manual processing redundant. In the end the use of automatically generated reminders and payment forms immediately improved the overall performance of the business considerably.

- In the early times of book-keeping, each payment had to be manually recorded in the books. Today, banks offer services to facilitate that kind of processing, e.g. import of data using e-banking or other data devices.

These services add information to digital paying-in slips, e.g. information about the customer, invoice number, date, etc., enabling IT systems to automatically record the payments. On the one hand the usage of such services led to more efficient business processes. On the other hand those benefits require various modifications and adoptions both in business organisation and thinking.

Figure 3 contains information about the customer in the last line (in the lower left corner marked with a circle: the customer number).

The image shows a P.S.K. payment slip with the following details:

- Top Left:** P.S.K. logo, EURO INLAND
- Top Right:** EUR 378,22
- Middle Left:** Kontoausweis Empfänger: 00096061060, GLZ Empfängerbank: 60000, Empfänger: UNIVERSITÄT WIEN
- Middle Right:** Verwendungszweck: Studienbeitrag f. WS 2004
- Bottom Left:** Auftraggeber/Erstschlichter - Name und Anschrift: Florian Humplik, LEOPOLDSDG. 29/14, 1020 WIEN
- Bottom Right:** 693510831004, Matr. Nr. 9351063
- Bottom Center:** 93510831004 (circled in red), 00096061060, 61060000, 00000037822, 42+

Figure 3: Add information to facilitate automatic processing



Related Patterns

down:

- [ERP-Landscapes] (03) Open architectures
- [ERP-Landscapes] (02) Focus on core functions to ease orchestration of business objects

horizontal:

- [RAPPeL] (1) Building the right things
- [RAPPeL] (32) Envisioning
- [RAPPeL] (30) Behavioral Requirements

(02) Focus on core functions to ease orchestration of business objects

Alternative names:

- Butterfly effect
- Domino effect

An ERP-system consists of a sequence of more or less trivial processing steps, e.g. administrating inventory and customer data, billing etc. Coordination between these processes is less straight-forward. It is not the number of elements (the number of processing primitives) but the number of interdependencies that determine the complexity of such a system.

To give you an example: The Marketing department demands sales reports based on projects. Bad news: reporting is indeed a top level system component. Therefore all ascending processing steps will be affected, such as input of data from now on has to include some link to a project.

So any change request bears the risk of a complete re-design of the project. Good examples for such 'minor' changes are: more differentiation of information (more detailed data), support of multiple languages, security requirements. Either of these examples is associated with shortcoming during the requirements engineering phase.

Context

The customer provides requirements including business objects. IT-experts are responsible for the orchestration of those business objects. Complexity is usually not apparent at first sight particularly for the customer. Slight changes in the specification of a single business object may add up considerably more complexity in the orchestration of the business objects.



Problem

How does the ICT-expert cope with inherent complexity of ERP-systems?

Forces

- **Simplicity vs. complexity**

Keeping an ERP-system simple requires balancing usability issues (for example efficient interfacing, quick processing) with support for the full range of an enterprises' business processes.

Supporting only a subset of business processes may be the key factor to support a business processes efficiently. The underlying principle is to perform some pre-processing tasks outside the system and integrate those results (for example resource planning in a spreadsheet document). Therefore you can focus on the core functions of the system.

- **Abstraction and generic solutions vs. reduced adaptability and/or extensibility**

Focusing on core functions is closely associated with more abstract solutions. The more abstract a solution the more widely applicable it is. The considerably high development expenses, however, must be paid in advance. Furthermore, implementation areas where abstraction is most promising and reasonable have to be identified, often based on limited experience.

- **Internal abstraction vs. external abstraction**

Abstraction on the code level facilitates reuse and thus, improves the quality of the code (internal abstraction). Abstraction becomes critical when it is obvious to the user. Software is more likely to be accepted by the end-user if it provides look-and-feel of real-world objects. Uniform and standardized user interfaces (external abstraction) conflict – to a certain extend - with such end-user needs.

Internal abstraction facilitates maintenance of the system for the, whereas external abstraction provides is likely to makes the system more difficult to use.

Solution

Focus on core functions to make abstraction techniques more likely to be applicable. This holds true for any phase of the project and in any component of the system.

Have the customer agree on the top-most project objective “keep it simple”. This will shorten time to market and provide both the customer and the ICT-experts with fast end-user feedback. Therefore further development will be shaped by the experience gained from the development of the core system. Carrying out a project according to that recommendation definitely puts more effort on the requirements phase.

Resulting Context

The benefit of abstraction is the possible application in a multitude of situations. Therefore the ICT-expert encourages organisational learning, thus steadily improving the components quality. Building generic solutions both requires considerably more human resources on the senior level and considerably increases the overall complexity of the project. The higher development costs have to be paid in advance therefore putting even more pressure on the organisation (for example, the need to follow-up projects increases the project's over-all risk).

Implementation

Business objects and their persistence requirements have to be defined. Those objects can be used as the basis for a MVC (Model view controller) approach as building blocks for reports and user interfaces.

Using abstraction techniques, such as XML, can turn possible threats resulting from interdependencies into opportunities. System properties re-used across many components may favour the application of generator frameworks or pre-processors. At this point, we emphasise that some efforts will be suitable for prototypical applications only, e.g. skeletons and stubs, hence the need for the integration of code is evident. The precondition for such an approach is a clear separation between information processing and information presentation at architectural level.

Known uses

- Report generators are a good example of how to separate a system's core functions from the presentation of information, by separating the output from the processing core. There are a lot of systems available on the market such as MIS OnVision. Tools like that support both read-only presentation of information and the input of data into databases of different vendors (even combining different database systems within one application).
- Connectors to other systems provide the basis for separating a system's core by delegating non-core tasks to other applications. Almost all of the major ERP vendors provide such connectivity. For example SAP provides connectors based on Java and .Net. Products from ERP-vendors provide connectivity on a low architectural level. Therefore SAP-consultants like SAPPHIR provide their own middleware products (for example SAPPHIR SITWare) to ease integration with third party products. By employing the full range of SAP computations it is possible to build a streamlined enterprise planning solution on top of SAP.
- Make use of a multidimensional database to carry out analysis and focus on the core financial management functions within the ERP-system.
- VXML: This standard provides a language to define the workflow with a voice recognition application. So the core functions of the voice recognition system remain the same. VXML introduces a new layer responsible for handling the workflow.



Related Patterns

horizontal:

- [ERP-Landscapes] (03) Open architectures
- [RAPPeL] (1) Building the right things
- [RAPPeL] (32) Envisioning
- [RAPPeL] (30) Behavioral Requirements

down:

- [ERP-Landscapes] (05) Increased value of information by making it specific

(03) Open architectures

Alternative names:

- Open design

Applications cannot focus on all the aspects of a business, thus IT vendors have specialized in particular processes, such as human resources, accounting, controlling, CRM.

Context

Such heterogeneous ICT-systems reduce a company's dependency on ICT-suppliers at the cost of various problems associated with synchronization and consistency issues. Therefore the need for integration arises. In heterogeneous ICT-landscapes defining interfaces becomes an even more critical issue.



Problem

How can long-lasting ERP-landscape architectures be designed?

Forces

- **applications specializing in certain tasks (without considering other applications needs) vs. applications following overall-corporate needs**

This problem reflects the well known problem of local optimization. In the context of a single task, a tailored solution will be more attractive than a generic solution. But in the corporate-wide context corporate-wide standards and homogenous interfaces may take higher priority.

- **Tightly integrated and optimized products from single vendors vs. open architecture**

Problems may arise when depending on a single vendor. If the supplier has to file for bankruptcy, technical support or product support are at risk. Ways to retain some independence are to require the vendor to share both documentation and to gain access to source code. Another approach is to define clear interfaces to replace an existing implementation with one of another vendor (open architecture).

- **An application's lifecycle vs. changing requirements**

Usually, ERP-system projects are long running projects with an extensive maintenance and evolution phase. Time-consuming or long-lasting projects continuously have to adapt to changing requirements.

- **Proprietary features vs. openness**

Some of these challenges can be solved at database level. But as many problems rely on specific objectives, e.g. fast database access without synchronization, or usage of proprietary database functions, the problem is usually solved in the application layer.

Solution

Use an open architecture.

Define the core functions of your application and consider both user interfaces to enter data within your application and interfaces to import existing data sources. Within heterogeneous ICT-systems interfaces define the bandwidth of possible integration. The higher the degree of integration the more likely a company can reduce the number of redundant data management tasks.

Therefore, insist that ICT-vendors implement import and export tools for wide-spread formats such as XML; have the database schemata documented and published; have the most important processing parameters encapsulated within the database (or ensure access to them in any other way), and motivate ICT-vendors to provide well documented APIs to set up real-time interaction between applications.

Open design makes various IT core-tasks even more difficult (for example consistent data across different subsystems and data sources and handling of security and synchronization issues).

Resulting Context

An open architecture is intended to interface with other applications. Ensuring proper and efficient interfacing requires inside knowledge about the applications to interface with. Therefore some kind of monitoring of third party vendors has to be established to cope with different version and new features in an efficient way.

Known uses

- Open Source projects like the Perl-based SQL-Ledger and the java-based Compiere offer an API and publish the underlying database model. To interface with the application you have to pay for the documentation of the API which is part of both applications business model.
- SAP supports database systems of different vendors. But it is difficult to interface on the database level (amazing abundance of tables and relations). Therefore other third party vendors (for example MIS ImportMaster and the SAP middleware SITWare by SAPPHIR) provide interfacing solutions.
- I have learned about a company that bought the SAP BW solution just to ease access to the underlying data.
- For example loading the SAP R3 data into SAP BW on a daily basis might not be feasible because the whole ITL processing might exceed the timeslots available. Because enhancing the hardware was associated with higher costs the company opted for a redundant ITL process to load their MOLAP (multidimensional online analytical processing) application on a daily basis.



Related Patterns

up:

- [ERP-Landscapes] (01) Orchestration of business and IT

horizontal:

- [ERP-Landscapes] (02) Focus on core functions to ease orchestration of business objects
- [ERP-Landscapes] (04) Support diversity if associated with opportunities

down:

- [ERP-Landscapes] (05) Increase value of information by making it specific
- [ERP-Landscapes] (06) Standards to achieve interoperability

(04) Support diversity only if associated with opportunities

IT systems represent an abstract model of a real-world business organization. Therefore they should reflect organisational structures (for example legal entities or departments). The need for organizational differentiation arises from several causes, for example, the need to create easily manageable organization unit, and mergers. Some issues related to mergers or acquisitions are differences between the joining entities in pricing, invoicing and information design.

Setting up segregated IT systems for each organisational unit is not appropriate if the same task has to be performed in multiple systems (for example data import and export). It is vital to keep data consistent.

Context

The majority of large companies consist of small subunits on different organizational levels, e.g. vestiges of mergers and acquisitions. Subunits may differ in their objective, business processes, and organizational structure.



Problem

How can the diversity of organizational subunits be incorporated into a single system?

Forces

- **Corporate identity vs. independent organizational subunits**

Managers prefer homogenous organizational structures. They are easier to compare and less complex. However, the organisational structure must reflect the most important characteristics of each subunit to provide a reasonable basis for analysis.

- **Historical reflection vs. present business state**

In a real-world situation business data has to be interpreted within the context of time to analyse time series. If the organisational structure changes there exist two time series. One based on the historical organisational structures the other reflecting the current organizational situation.

In this case it helps to think of the problem in terms of two layers. On the one layer there are real-world objects associated with costs (such as employees, machines and accounts) and there exists more than one way of grouping them. Such logical groupings are introduced by organisational structures such as teams, projects, departments and legal entities.

Usually a single all-purpose logical structure (mapping) does not exist. Therefore the grouping has to be done on a dynamic basis. ERP-systems have to consider diverse mappings at different points in time, and diverse mappings due to different requirements across different tasks (e.g. less detail grouping in the context of budgeting).

- **The winner takes it all**

Usually, people prefer to accomplish tasks the way they have been used to by using tools they are familiar with. However mergers or acquisitions usually follow the rule that the less powerful company simply has to adapt.

Solution

Build parallel hierarchies and de-couple objects from logical groups.

Separate the objects to be grouped from their logical grouping. Economic processes are inherently dynamic. Therefore parallel hierarchies simultaneously provide different logical groupings.

Resulting Context

Application of parallel hierarchy techniques produce overlapping hierarchies that can easily lead to multiple computations. The most imminent drawback is increased complexity.

Implementation

The database must allow for storing hierarchical structures representing the logical groups of arbitrary building blocks. The final business logic operates on the hierarchical structures. Attributes may for interpreting elements differently according to various contexts.

Known uses

- The principle of separating between hierarchies and objects is the underlying concept of multidimensional databases, such as the Alea MOLAP-database of MIS AG. Imagine the field of accounting. Financial statements are composed of accounts which have to reflect different accounting standards (such as local GAAP and US-GAAP). Therefore maintain two parallel hierarchies one for local GAAP and one for US-GAAP. As both hierarchies are basically composed of the same base elements you need some method to account for adjustments. Figure 4 gives some basic understanding of how to bridge between local GAAP and US-GAAP.

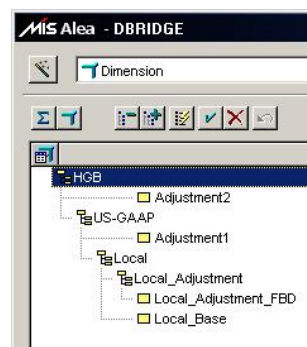


Figure 4: Parallel hierarchies and bridging between different GAAPs



Related Patterns

horizontal:

- [ERP-Landscapes] (03) Open architectures

(05) Standards to achieve interoperability

Heterogeneous ICT-landscapes requires interfaces for interconnection, import and export.

Context

The market share of multinational IT companies has introduced the phenomenon of a high number of de-facto standards, such as PDF, RTF, XLS. They exist in parallel to the official standards like those issued by ANSI. It is crucial to select the appropriate (de-facto) standards.



Problem

Which interfaces should be used and provided?

Forces

- **Full support for formats and full connectivity vs. selected support and economic development**

Full connectivity becomes more difficult when the products of multiple ICT-suppliers are maintained. The number of supported vendors imposes enormous maintenance costs.

- **Open and extensible architecture vs. proprietary products**

An open architecture grants access to your implementations. Therefore your competitors are entitled to develop extensions to your products or even develop substitutes for (parts of) your application. An open architecture can be integrated in landscapes, whereas proprietary systems require adaptations of the landscape itself.

- **Standards might become obsolete**

Supporting widespread applications (for example Lotus 123) is risky. If new technologies emerge, then it will be difficult to identify tomorrow's standards. Additional challenges will arise if you have to support new technologies in the very early stage of their life-cycle.

- **People issues vs. best practice**

To set up a standard industry proponents have to agree on rules and to adapt their own products according to that rules which is a costly issue. Therefore and due to strategic issues it is difficult to get supported by all the industry-leading companies.

- **Version 1.x vs. version 1.y**

There is no support of a certain format. Usually only a certain range of versions are supported. Therefore interoperability is closely related to the management of ICT-infrastructure.

Solution

Use and provide interfaces which implement standards or de-facto standards.

Many of today's development environments offer off-the-shelf tools or controls that support the transformation of databases or data into a multitude of formats. Special attention has to be paid on topics like restricted scope of applicability of certain formats, limited availability of a suitable formats, scalability issues or difficulties due to different versions of formats.

Resulting Context

If you have decided to support certain standards then certain managerial tasks are ahead. Updates of applications within heterogeneous ICT-Landscapes become an even more crucial topic the orchestration of different application has to cover versions of import and export formats.

Implementation

Keep in mind different domains require different standards. For example within the domain finance and controlling MS Excel plays a dominant role. It is common practice for major vendors to facilitate migration into and out of their competitor's products. For example MS Excel supports the import of Lotus 1-2-3 documents and vice versa. So by supporting either MS Excel format or Lotus 1-2-3 you support both formats because both worlds provide import facilities for the competitor's product. Therefore clever selection among predominant formats allows for a wide coverage, due to various import facilities provided by market leaders' products.

Known uses

- TurboCASH provides for the import of a proprietary format, a wide spreadsheet format and XML-format. Apart reports can be exported using various formats, such as TXT, RFT, CSV, XLS, WMF, and HTML. Different formats are used according to the context.
- NOLA provides for CSV export. CSV provides easy import and export supported by a wide range of applications.



Related Patterns

up:

- [ERP-Landscapes] (02) Focus on core functions to ease orchestration of business objects
- [ERP-Landscapes] (03) Open architectures

(06) Increasing the value of information by making it specific

Alternative name:

- Tailored fits best

An ERP-system is usually built upon at least one database system. But the available information can be combined in a million ways. Apart from that there another million of compression levels available (for example grouping of information by summing it up).

Context

The definition of system requirements reveals different user groups with diverse information needs. The typically long lifetime of an ERP-system may also cause future changes in information needs.



Problem

How to cope with information needs of different stakeholders?

Forces

- **Information flood vs. information deficiency**

IT can easily flood the user with information. Compacting information reduces the amount of data, yet may ignore some necessary details for specific users. It is somehow an art to determine the appropriate detail-level of information to solve specific tasks.

- **Predefined selection vs. user-defined selection**

Selecting predefined information for user groups and task routines increases the efficiency in solving the tasks, yet decreases individual search possibilities for solving related tasks. User-defined selection of detail level data empowers the user but may also increase the overall search effort (time consuming and therefore costly task).

Solution

Put the customer in charge and design access to information.

By employing report engines with a drag-and-drop like interface the customer is empowered to adapt information presentation to changing needs. The first step to manage diverse information needs is to identify the different groups. Next, balance the company security policies with the information needs of any single group. As a general rule, access to information should be denied. Only after a request has been approved should access be granted. Moreover the access should only be granted at a certain detail level.

Resulting Context

Administrating information access is likely to be underestimated. Putting the customer in charge is a nice thing but only works if accompanied with training. Sometimes vendors do not prefer such an approach because customers become more independent and former revenues for adapting the system disappear.

It remains difficult to manage role-centric information needs. Job profiles (roles) provide a good basis for restricted access. For example the finance department requires a compressed view for external communication but detailed access for internal reporting. Therefore simple job-based restrictions are not flexible enough. The requirement for the user to interactively switch between different aggregation levels on the application layer is obvious.

Implementation

Parameterisation has to provide for the appropriate tools to meet the requirements. The objective may also be realised on an architectural level. Providing separate applications specialized in specific information aspects, e.g. personnel accounting, thus provides detailed information and exports the aggregated information into the main system.

Typical information systems use a single database and basically rely on a single type of processing. In consequence, the need for creation of a superior layer arises which uses the same information but presents it in a user-centric way.

Known uses

- Compiere: Offers a wide range of possibilities to the customize reports (such as fading in and fading out of columns or grouping) as seen in Figure 5 Customizing Reports.



Figure 5: Customizing Reports

- In Europe the salary of employees is usually considered to be critical information. Therefore this information is usually presented on a highly aggregated level.
- Multidimensional databases are best suited for any sort of reporting on an aggregated level. Such systems are based on dimensions and attach hierarchical structures with information. For example the dimension salary may introduce aggregate levels like total, total per department or per employee. Systems like MIS Alea provide access control based on hierarchy levels. So you can build one single report (using a reporting tool like MIS OnVision) and manage the appropriate aggregation level of information by setting the appropriate access right based on the dimension's hierarchy.



Related Patterns

up:

- [ERP-Landscapes] (03) Open architectures

Acknowledgements

Special thanks to our shepard Kristian Elov Sørensen. In addition, the authors received many valuable inputs during the workshops at VikingPLOP2005 - in particular Linda Rising, Cecilia Haskins and Allan Kelly. The paper has been shaped by the inputs of those people. Special thanks for the deep involvement of Cecilia Haskins who spent a lot of time discussing various topics and supported the final editing of the paper.

REFERENCES

- [01] Timeless Way of Building; Alexander, C.; 1979; [Pattern roots]
- [02] A Pattern Language: Towns, Buildings, Construction; Alexander C., Ishikawa S., Silverstein M.; 1977; [Pattern roots]
- [03] Software Patterns; Coplien J.; 2000; Bell Laboratories, The Hillside Group;
<http://users.rcn.com/jcoplien/Patterns/WhitePaper/> [overview patterns]
- [04] Design Patterns; Gamma, Helm, Johnson, Vlissides; 1995; Addison-Wesley;
[GOF, object orientation and patterns]
- [05] Pattern in der Praxis; Keller W.; 1999; VAA-Fachtagung im März 1999 in Köln;
www.gdv-online.de/vaa/vaafe_html/tagung1/pattprax.pdf [patterns]
- [06] Using Pattern Languages for Object-Oriented Programs; 1987; Beck K.,
Cunningham W.; OOPSLA'87;
<http://uk.builder.com/whitepapers/0,39026692,60006877p-39000929q,00.htm>;
[patterns, OO]
- [07] RAPPeL: A Requirements Analysis Process Pattern Language for Object
Oriented Development; Whitenack B. G.; 1994; [patterns for requirements
analysis]
- [08] Pattern-basierte Modellierung von Geschäftsprozessen; Förster A.; 2002;
Universität Paderborn Fachbereich Informatik, Diplomarbeit 2002;
www.upb.de/cs/ag-engels/Papers/2002/Diplomarbeit_Foerster2002.pdf;
[patterns for business processes according to ISO 9000]
- [09] Objektorientierte Geschäftsprozessmodellierung mit der UML; 2003;
Oesterreich B., Weiss C., Schroder C., Weilkiens T., Lernhard A.; 1. Aufl.,
Verlag dpunkt-Verl., Heidelberg [modelling of business processes, UML]
- [10] Geschäftsprozessanalyse - ereignisgesteuerte Prozessketten und
objektorientierte Geschäftsprozessmodellierung für betriebswirtschaftliche
Standardsoftware; Staud J. L.; 2001; 2., überarb. und erw. Aufl., Verlag Springer
[EPK (event-driven process chains), SAP]

- [11] Business Process Reengineering – Strategien zur Produktivitätssteigerung – Konzepte und praktische Erfahrungen; 1995; Zeitschrift für Betriebswirtschaft, Ergänzungsheft 2/95Schriftleitung: Albach H.; [production planning, productivity]
- [12] The Core Competence of the Corporation; Prahalad, C., K., Hamel, G.; 1990; Harvard Business Review, Vol. 90, no. 3, May-June (1990), pp. 79-91; [core competence]
- [13] Fearless Change; Rising, L., Manna, L., 2004; Addison-Wesley; [patterns how to cope with change]
- [14] The Patterns Handbook: Techniques, Strategies, and Applications; Rising, L. (ed.); 1998; Cambridge University Press; [introduction patterns]

Patterns of THE MATURE CUSTOMER

Author:

Susanne Hørby Christensen
Buddinge Hovedgade 159,1
DK-2860 Søborg
Denmark
Tlf: +45 3966 7484
E-mail: susanne@itu.dk

Student at
IT-University of Copenhagen
Rued Langgaardsvej 7
DK-2300 København S

Abstract

Software development organizations are urged to be more mature in their work routines. But to accomplish success with software development the customer should also accomplish some level of maturity. I present some patterns that facilitate maturity for customers of software systems.

Content

| | |
|---------------------------|---|
| Interfaces between Roles | Define clear communication interfaces between you, your colleagues and the supplier organization. |
| Prioritized Requirements | Make visual summaries of your requirements and be active and realistic in prioritizing your requirements together with your supplier. |
| One Place for Information | Make communication transparent to all stake holders so that the progress of the project visible to everybody. |
| User Narratives | Be attentive to rumors and stories from all levels in your organization, use them properly and do not forget to give feedback. |
| Incremental Releases | Let the future users examine the releases continuously at regularly intervals. |
| Change Request Control | Allowing change is necessary - but it is important to map these change requests to the priorities. |

Introduction

Background

Ideally two organizations signing an IT-project contract should have the same goal and vision for their future work and cooperation. This is however often not the case and consequently mistakes and misunderstandings will occur between the involved parties.

The customer differs from the supplier by having interests in many other things than IT-systems. The customer focuses on completely different products, and uses IT as a tool - and tools are expected to be user friendly and run without errors.

The reasons for procuring an IT system are similar whether the customer belongs to the public sector, the food industry or designs wear. The organization might be aware of the importance of this new system, but except for the problem-solving part, they are not interested at all. In Denmark there is growing demand for more mature customers, as it has been found in projects conducted by the Ministry of Science, which work with these issues. [7]

The supplier on the other hand, knows the technology, but might not know very much about the customer business area or the processes used in the customer's organization. Therefore it is very important that the customer is able to provide relevant information about their business area and their process to the supplier.

To prevent problems and misunderstandings happening in IT-projects, we want mature customer and supplier organizations. But what characterizes maturity? Patterns of the

Mature Customer aim to give a description of some of the important characteristics for the customer and hopefully these patterns will evolve further for years to come.

The Patterns

The patterns aim at customers procuring an IT system, which involves development. It is expected that the project involves several persons in both organizations, and the person addressed "you" in the patterns is the project manager at the customer site.

The patterns should provide a toolbox for software customers. Do not expect a complete language, but some central patterns for how a mature customer will act.

The structure of the patterns is based on the Alexandrian form [2]: Name, context, delimiter, problem, forces, solution, more about the solution, delimiter, resulting context and potentially diagrams.

Sources for the patterns

Inspiration for the patterns comes from interviews and from literature.

Interviews

I am not a senior project manager, but I interact with a wide variety of experienced people, and from these conversations I have conducted the pattern mining for my Patterns of The Mature Customer.

I held a workshop (15th March 2005), with 6 participants: 2 suppliers and 4 customers. The suppliers were Gertrud Bjørnvig (TietoEnator) who works with patterns at a daily basis and Thomas Christensen (Nordija) who have had several customers in the public sector. The customers were Lotte Mossin (former DSB Informatic now TietoEnator), who was the project manager of IT-procurements, Jens Chr. Hauge (Agency for Governmental Management) who has been involved as Principal Consultant in integrating several governmental IT-systems, Jens Petersen (Ringsted Municipality) IT-Manager and Sten Mogensen (Ministry of Science, Technology and Innovation) who currently is involved in a project about defining the mature customer in IT-projects. The workshop provided great input for the development of the patterns and was followed up by another workshop in August, where most of the same people took the opportunity to provide me with their feedback. [7].

Literature

For my patterns I found Neil Harrison and James Coplien's "Organizational Patterns of Agile Software Development" [3] a good source of inspiration. The book gives a good description of how to build a supplier-organization - i.e. a software developing organization. They also describe the relationship with the customer, but always from the supplier's point of view. I also made use of Mary Lynn Mann and Linda Rising's book: "Fearless Change - Patterns for Introducing New Ideas" [5]. The latter book describes

the introduction of new ideas in a broader sense. Apart from these two newer books, I derived great inspiration from reading Alexander's books when I started to uncover the ideas of patterns [1] [2].

When trying to map and summarise the ideas from these books from a customer's point of view, I came across a couple of missing links, e.g. with respect to meeting the demand set by the discussion about the customer's responsibility to ensure success of procuring IT systems.

I believe that "Organizational Patterns of Agile System Development" [3] is a powerful resource for an organization - and it also suggests some ideas for how two organizations merge. But a customer/supplier relationship might only last for a short period of time where the management team in one organization does not have the power to lead a change management project including both organizations.

Interfaces between Roles

... you are at the beginning of an IT project. You know your own organization, but is the communication path clear to everybody?

Communication might get fuzzy when an IT-project starts to evolve.

You are responsible for the outcome of the project. On the one hand you would like to be informed about everything that is going on - on the other hand you want your staff to work independently and determined.

If the communication path is not clear to everybody, questions concerning the project might circulate for too long before they finally make their way to the right person. If people can not get answers to their questions they start guessing, and you will lose track of your own organization as well as the supplier's organization.

Therefore:

Define clear communication interfaces between you, your colleagues and the supplier organization.

It is important that everybody with a question finds the answer with no run-around! The roles for a project should be well defined, so that e.g. a development team in the supplier organization can easily find the technical coordinator in your organization in case of technical questions.

When people across the organizations get the answers they need when they need them, they will feel appreciated and they will be more self-confident, which will show in their daily work.

The communication interfaces could consist of several persons, e.g. the project manager (you), the technical coordinator, the requirements coordinator, the quality coordinator and the steering committee. The size of the roles depends on the size of the project, i.e. in small projects one person might have more than one role and in larger projects there might be several persons attached to one role (with one primary responsible).

The communication flow could be supported by a web-application as well as professional communicators who could review documents and transform them such that they become readable to all stakeholders.

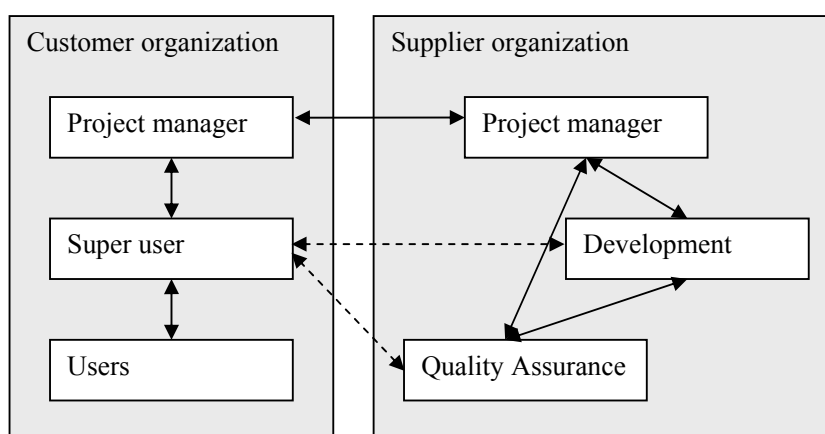
When people throughout the organization know who are responsible for different issues, it will calm down tensions. But be aware that if the communication chart is too rigid, it could lead to a stifling bureaucracy where those who want to appear important will make sure that as much info as possible will go through their office irrespective of whether this is sensible or not.

It can be perfectly fine if the developers and quality assurance staff contacts super users in your organization directly and vice versa, but be attentive to the possibility that trouble in the communication might occur. E.g. if there is a lot of communication between customer super users and suppliers developers, you might lose track of what is happening, and if misunderstandings occur it will cause difficulties for you. Therefore it is important to keep contact to the entire project.

Of course it should be possible for super users in your organization to maintain a dialogue with development and quality assurance in the supplier organization, but be aware that the communication is controlled in such a way that you are not exposed to misunderstandings. This is helped by ONE PLACE FOR COMMUNICATION (to be found later in this paper).

This pattern is closely related to the organization patterns: COMMUNITY OF TRUST, MERCENARY ANALYST, ENGAGE CUSTOMERS, GATEKEEPER AND SHAPING CIRCULATION REALMS [3].

The diagram below shows the idea. Be aware that the number of roles/shareholders might vary according to the size and type of the project.



Prioritized Requirements

... you have applied INTERFACES BETWEEN ROLES and you are about to plan the project with the supplier.

When the development of your system starts, you risk the supplier starting to implement features that block more important features or that features are implemented in a completely different way than expected.

As a customer you want to have the best solution for the amount of money/resources you have dedicated.

Often you have an idea of what the future system should be like, but it is not always the way it turns out to be.

Sometimes systems focus on features of minor importance and sometimes systems turn out to be even better than expected by the customer.

On the hand you should tell the supplier all about your expectations and expect to have a solution fulfilling these. On the other hand you should be open-minded to the supplier's point of view - they might see other constellations that you cannot see because you are too deeply involved in the old systems.

It is crucial that you are in control of the processes in your organization and that you trust, that the supplier is control of their processes.

Therefore:

Make visual summaries of your requirements.

When you are active from the beginning, it will be easier to take a though decision later on. Be open to ideas but keep track of the core features. It is a good idea to make visual summaries of the requirements. Such summaries are very valuable later in the process, where changes of the request/suggestion might occur. These summaries help the developers to understand exactly what they should implement and the dialog with the supplier help you to acquire realistic expectations, e.g. there is a Danish research project showing remarkable increase in understanding for the user's situation and thereby better user interfaces, when the developers have user scenarios in addition to technical descriptions [6].

All the way through the project it is important to keep in mind that the system is influenced by any business problems you might have and that - you cannot expect a system to deal with your business problems if your business is a mess.

When applying PRIORITIZED REQUIREMENTS you demonstrate your determination towards being a mature customer who is open, active and responsible. You will get a deeper understanding of the development aspects of the project, and your attitude would probably make the flow of communication easier. You should however be aware of the integrity of keeping everybody involved, but make sure you do not step over the suppliers limit by being too deeply involved. This may cause offense and challenge the important COMMUNITY OF TRUST [3].

This pattern could be supported by a supplier using the organizational pattern IMPLIED REQUIREMENTS that help your understanding of the requirements. IMPLIED REQUIREMENTS gives chunks of functionality a common name that makes more sense than traditional requirements. [3]

One Place for Information

... system development has started and people from your organization talk about the project and some even talk with the supplier organization.

E-mails, printed documents, verbal communication may result in misunderstandings that can destroy the faith to build trust between organizations.

It is important to have a constant flow of information between organizations as well as within organizations. But when a developer suddenly starts working from the description in an old document or rumors spread far and wide, people become insecure and dissatisfied about the direction.

Therefore:

Make communication transparent to all stake holders so that the progress of the project visible to everybody by having a central official place for project information.

When there is one place, where all information is collected and readily available, you have a better chance of avoiding misunderstandings and rumors. This place should be easily accessed by stake holders in the project - both in the customer and the supplier organization.

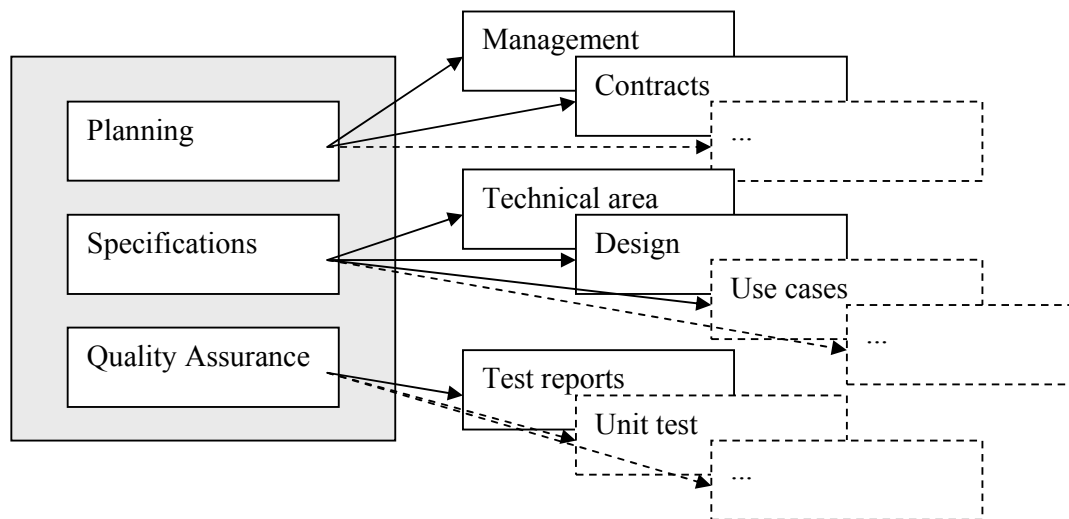
Preferably it should be a place on the web or intranet where all documents are available in all versions, the newest one being the default. Of course it can be difficult to maintain a system with all details, but at least it should be possible to find all documentation, decisions made and the plans for the future. This could be coupled with an awarding system encouraging people to pay attention to details.

The system could be provided by the supplier and therefore be included when PRIORITIZED REQUIREMENTS is implemented.

This pattern could be helped by a GATEKEEPER [3], who is the information agent on the supplier's side.

The diagram illustrates a customer interface to ONE PLACE FOR INFORMATION, where the gray area could be the entrance, and the rest is subfolders. The dotted lines indicate

which subfolders might be hidden to the customer. All relevant information should of course be visible, but too much information could stifle the intent.



User Narratives

... you are in the beginning of an IT project and you hear stories and rumors from the future users

Missing important contributions of personal knowledge from the users can result in a bad base for your system.

Your employees/colleagues will be the future users of the system. Often a new system should help them do their job in a smarter/faster way and include some new features that the users have been missing for a while. Expectations of the system will often have been discussed prior to the implementation of the system.

These discussions are crucial for you. They tell you how the new system will be perceived - are the users frightened that it will reduce the number of employees or do they expect the system to help them do a good job, so that the company can expand and perhaps hire more employees? There will be rumors to kill and nuggets of gold to be discovered. The issues will be discussed in the canteen and similar places, but can be hard to hold of for the management team.

You will hear all sorts of comments and stories. Some of these stories have great importance for the project but it is difficult to conserve them.

You might hear something from a future user that seems unimportant but at a later stage, it could be crucial to you (or somebody else within the project) to be able to find this particular story. The opposite could also be the case, where a story at first seems extremely important but later turns out to be irrelevant.

Therefore:

Be attentive to rumors and stories from all levels in your organization, use them properly and do not forget to give feedback.

Meet the future users in face-to-face interviews or workshops, where these stories are encouraged. And be aware of stories that pop up during your day-to-day work.

Make sure that you are open towards these stories when they arise - otherwise your employees will soon stop telling you which things pop up in their minds.

The information flow from the future users towards the developers depends on the number persons involved.

When you ask for stories you should be clear about the purpose. You might engage some employees and it is crucial that you give them feedback. Tell them how you used their ideas or explain why you did not, and make sure that the people are aware of the possibility that their ideas might not be used.

If it is a small system, with just a handful of end users, interviews and direct communication between end users and developers will be logical. But in huge systems with many end users it would be necessary to make a plan for this communication. It would be important to implement ONE PLACE FOR INFORMATION to ensure that the people involved can follow their ideas throughout the entire project. The USER NARRATIVES help you to avoid risks seen by your employees, but also to see germinating troubles and thereby be able to avoid these rumors before they become reality.

It is important to evaluate input and give feedback continuously, otherwise skepticism will grow and ruin the COMMUNITY OF TRUST build within your organization.

Incremental Releases

... the supplier has started implementing the solution for you.

The project might miss the path. Features that are necessary can be difficult and expensive to fit in afterwards.

There are many stories about customers who order a system and unfortunately get exactly what they asked for - perhaps several months/years after signing the contract. Fortunately this is not seen as often as in the past, but still expensive surprises can occur. You want to follow the project closely but you do not want to waste resources by being too involved.

Therefore:

Let the future users examine the releases continuously at regularly intervals.

To prevent disastrous results it is very important that the future users get their hands on the product and try it while the developers observe them.

Such sessions can enlighten the developers about how the users will use the system in practice and it will be possible to discover if the developers misunderstood the intent of the requirements.

These releases should be planned and incorporated as milestones and be demanded in correlation to release of money. It is crucial that the customer has PRIORITIZED REQUIREMENTS.

This is an aspect from the agile world, but it is perfectly possible to fit it in to other types of projects as well.

Sometimes it might be difficult to test a single feature because it depends on other features which are not yet fully implemented. Instead of delaying the entire examination, it could be a good idea to let the future users comment on mock-ups of planned user interfaces. This could catch some of the problems early.

This pattern is closely related to ENGAGE CUSTOMERS and BUILD PROTOTYPES, as well as with INCREMENTAL INTEGRATION, ENGAGE QUALITY ASSURANCE AND APPLICATION DESIGN IS BOUNDED BY TEST DESIGN [3]

It is important that the persons responsible for accept of such releases should be included in INTERFACES BETWEEN ROLES.

The illustration shows a mock-up which might be OK from a technical point of view, but logically the user might want a slightly different structure - and maybe some additional fields and buttons.

The illustration shows two side-by-side mock-up forms for adding two people. The first form is titled "Add person 1 of 2" and has fields for Name, E-mail, Address, and Country. The second form is titled "Add person 2 of 2" and has fields for Zip, Web, Phone, and City. Both forms have "Exit" and "Menu" buttons at the bottom.

| Add person 1 of 2 | |
|----------------------|----------------------|
| Name: | <input type="text"/> |
| E-mail: | <input type="text"/> |
| Address: | <input type="text"/> |
| Country: | <input type="text"/> |
| <div>Exit Menu</div> | |

| Add person 2 of 2 | |
|----------------------|----------------------|
| Zip: | <input type="text"/> |
| Web: | <input type="text"/> |
| Phone: | <input type="text"/> |
| City: | <input type="text"/> |
| <div>Exit Menu</div> | |

Change Request Control

... there have been some suggested changes in the original requirements

For several reasons changes to the original requirements occur. This process can be fatal if not properly managed.

Change requests can come from the supplier organization as well as from your steering committee or as a suggestion from the future users. It could also be the result of a new law that influences your business.

Major changes can disturb the entire project if you simply accept it without careful planning of what to do now. Such disturbances will be fatal for the spirit within the organizations.

Therefore:

Allowing change is necessary - but it is important to map these change requests to the priorities.

You should set up a change process in partnership with the supplier organization. Every time a change suggestion occurs, it should be examined by users, technical staff, management as well as the supplier's ditto. This should lead to a risk analysis in order to determine the risk of including or excluding the change, and be followed by a decision; to reject the change, to implement it as quickly as possible or to fit it into an update later on. The risk includes impacts towards cost, schedule as well as quality.

You would compare this against your PRIORITIZED REQUIREMENTS and see if it has the substance to bring it to the supplier.

This is very similar to FEEDBACK BEFORE CHANGE, an unpublished pattern by Cecilia Haskins that considers the importance of feedback from the supplier prior to acceptance of change requests. [4].

And like PRIORITIZED REQUIREMENTS it is closely related to the common names achieved by IMPLIED REQUIREMENTS [3]

Acknowledgement

I am grateful to Neil Harrison for shepherding these patterns in spring 2005 - it was a hard but good learning process.

Thanks to the people in Business Workshop at Viking PLoP 2005 for their careful reviews and valuable ideas: Allan Kelly, Cecilia Haskins, Florian Humplik, Jesper Christensen, Juha Pärssinen, Kristian Sørensen, Linda Rising, Mauri Myllyaho, Michael van Hilst, Pavel Hruby, and Rebecca Rikner.

Thanks to the participants of the workshops held in Denmark, where I got many ideas for central patterns and interesting feedback: Gertrud Bjørnvig, Lotte Mossin, Jens Chr. Hauge, Jens Petersen, Sten Mogensen, and Thomas Christensen.

Finally I want to thank my friend, Anne Scott, for reviewing the document.

References:

- [1] Alexander, Christopher. (1979). *The Timeless Way of Building*. New York, Oxford University Press.
- [2] Alexander, Christopher. (1977). *A Pattern Language*. New York, Oxford University Press.
- [3] Coplien, James & Neil Harrison. (2004). *Organizational Patterns of Agile Software Development*. Pearson Prentice Hall. ISBN: 0-13-146740-9.
- [4] Haskin, Cecilia.
- [5] Manns, Mary Lynn & Linda Rising. 2004. *Fearless Change - Patterns for Introducing New Ideas*. Addison-Wesley. ISBN: 0201741571
- [6] Strøm, Georg (2005) 'Undgå misforståelser i softwareudvikling', *Computerworld*, Copenhagen, 2 December, p. 18.
- [7] Two workshops. One held at TietoEnator A/S 15th March 2005, with focus on customer-supplier relationship. Participants were: Gertrud Bjørnvig (TietoEnator), Lotte Mossin (TietoEnator), Thomas Christensen (Nordija), Jens Petersen (Ringsted Kommune), Sten Mogensen (Ministry of Science, Technology and Innovation), Jens Chr. Hauge (Agency for Governmental Management) and I. The second one held at IT-University of Copenhagen 23rd August 2005, where the draft patterns were discussed among other subjects. Participants were Thomas Christensen (Nordija), Jens Petersen (Ringsted Kommune), Sten Mogensen (Ministry of Science, Technology and Innovation), Jens Chr. Hauge (Agency for Governmental Management) and I. So in both workshops both suppliers and customers were present, to ensure that the parameters were discussed in a proper way.

Privilege Separation

A Security Pattern

Dan Forsberg, <dan.forsberg@hut.fi>
Helsinki University of Technology;
Nokia Research Center

Abstract

When *Privilege Separation* pattern is used it divides one functional element into smaller functional elements with different privileges and restricted interfaces. The intent is to separate privileges of functional entities and thus restrict the area where the functional entity has rights to act. When properly and wisely applied makes the system more secure, modular, and easier to analyze by dividing an entity into multiple entities.

CONTEXT

Server software programming in many times requires special privileges for doing certain operations like binding the server into a service ports, accessing files with confidential information like passwords, and managing cryptographical operations like data signing and encryption with confidential session keys.

PROBLEM

The problem that this pattern solves is *how to minimize the effects of vulnerable exploited code* (like buffer overflows).

FORCES

There exists a system with multiple functionalities and valuable information and only part of the system functionalities need to access this information. Unintended leakage of the information must not happen between functional elements.

A system needs to access secure information (continuously), which means that some kind of access control to the information must be implemented. This pattern becomes useful if the whole system does not need the information as it is, but a derivation of it (like authentication result, handle to a socket/file descriptor, etc.).

On the other hand if this pattern is not used at all the system modularity does not exist and probability for security vulnerabilities increases. A bad example would be a server that needs high privileges for a small amount of time to do a small task, but the whole server runs all its lifetime with high privileges.

When dividing the system into smaller systems, the complexity of the combined system increases. As a result of the division into multiple entities the interfaces between the entities must be specified. The more interfaces, the more work is needed. In both of these cases a balance between complexity and sound security need to be found.

SOLUTION

Normally modularity is based on functional aspects. Privilege separation pattern brings additional modularity based on different privileges. In addition to using this pattern the least privilege security principle should be applied to the resulted divided entities for providing a full solution to the problem.

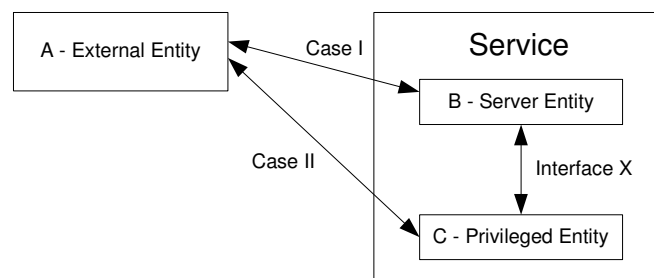


Figure 1 An example of privilege separation pattern structure

Figure 1 illustrates a simple structure of the privilege separation pattern. External Entity (A) uses a service which consists of a combination of the Server Entity (B) and the Privileged Entity (C). Server Entity and the Privileged Entity co-operate together via interface X for serving the External Entity.

Privileged entity has access privileges to sensitive or valuable resources. When applying this pattern the service entity is divided into two entities with separated privileges: server entity and privileged entity. The privileged entity holds privileges to access valuable resources and the server entity communicates with it.

The privileged entity derives information and/or resources for the usage of the server entity. In Case I this information can be a result of processing between the External Entity and the Privileged Entity, like authentication result from SIM card or a file descriptor from operating system to a user space program. In Case II the external entity communicates with the server entity. The privileged entity provides information like checking if the supplied password was correct according to a passwd file in the system.

Depending on the resources that the privileged entity provides, the interface to the service consumer entity can be through the server entity (Case I, see Figure 2) or through the privileged entity (Case II, see Figure 3).

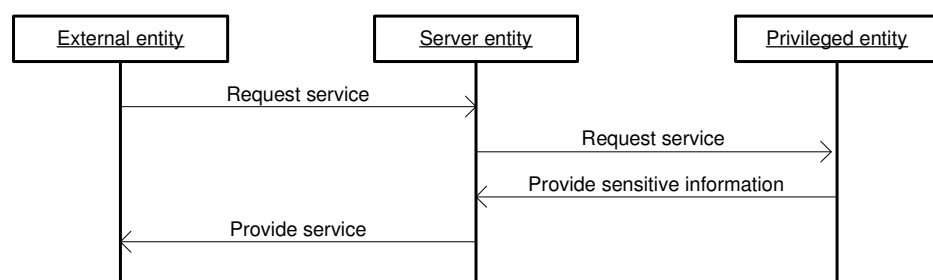


Figure 2 Case I

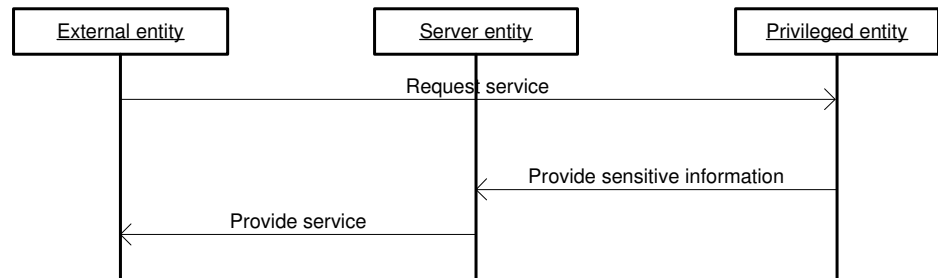


Figure 3 Case II

If the pattern is recursively (see Figure 4) applied too many times, the system becomes cumbersome and complicated to manage. An example would be a system that implements different access control lists (ACL) for each and every file, application, and resources. Using the same access rights for multiple files (access rights group) is thus used to make the system more manageable.

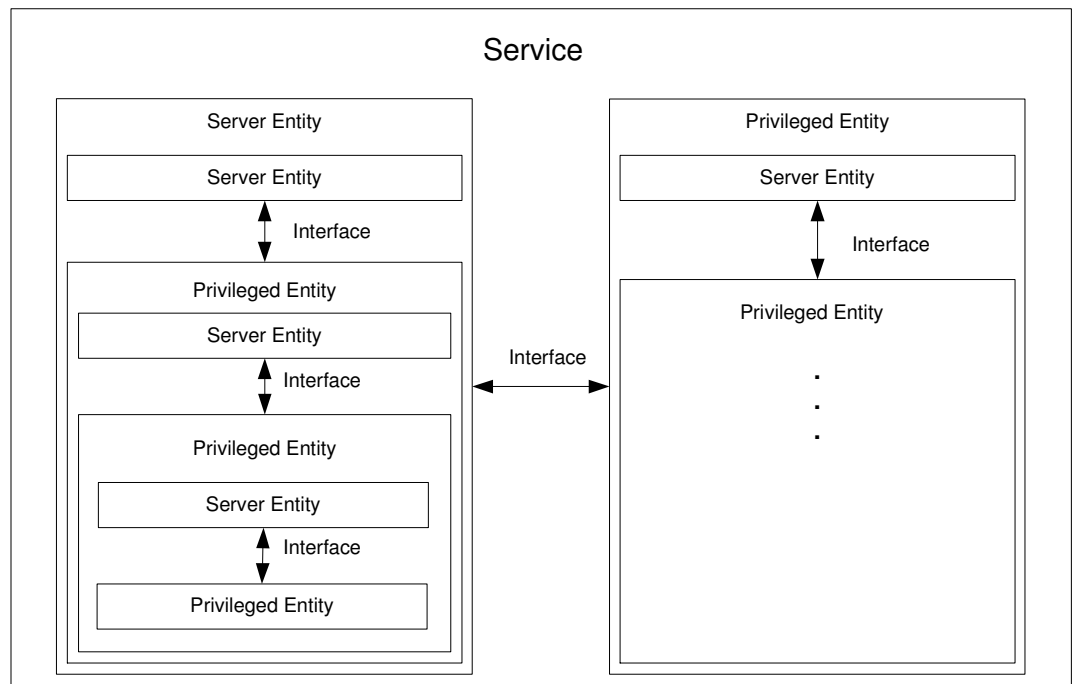


Figure 4 Simple example of the recursive nature of the *Privilege Separation* pattern

RESULTING CONTEXT

As a result the systems privileges are divided and only part of the system is permitted to access privileged information. If wisely applied vulnerable code in one entity does not break the security of the other entities or at least makes it harder to exploit the code vulnerabilities.

KNOWN USES

- OpenSSH privilege separation [1], [2] is an example where the SSH server was divided into two functional entities for better security. Also `vsftpd` uses privilege separation to limit the effect of programming errors [3].
- When designing network architecture isolating long-term security credentials into separate servers in the network architecture to better protect them. This separate server would then have an interface to other selected network interfaces that are eligible to contact the server. Examples include isolated databases with access control like `passwd` file and programs/libraries that can access it and a HLR register in telecommunications systems architectures.
- When managing and using security credentials. Caging security credentials with hardware approaches, like SIM cards in mobile phones.
- It is important to restrict the rights of an executing process in an Operating System. Patterns like *File Access Control*, *Controlled Virtual Address Space*, and *Controlled Execution Environment* [4] are tools when providing privilege separation, which helps minimizing the scope of security threats for network servers. An extreme example of privilege separation is `chroot(1)` (“change system root”) system capability in Unix systems. It provides strong process isolation and actually implements also the patterns listed above.
- Applying this pattern for the software design and development tools, especially to the graphical user interfaces, could mean that the developer would be able to separate privileges by painting areas of code with mouse or selecting files. Then the compiler together with the target system could provide different privileges to these areas. This seems to be a novel idea. A target system could be an operating system for example. Separation could be a combination of processes and files. On the other hand the compiler and OS kernel could support privilege changing for a process or thread by inserting code in the compilation phase into the executable binary. This inserted code would then automatically change the privileges of the process/thread. How to select the privileges is out of the scope of this paper. A configuration file could be used for example.

Related Security Patterns

Single Access Point security pattern creates a single interface for communication with external entities. After our privilege separation pattern has been used *Single Access Point* security pattern can be used for the communication between resulting entities. However, the divided entity may have other interfaces towards the external entities, which thus breaks the *Single Access Point* security pattern model.

Applying *Layered Security* pattern makes the system to have multiple levels of security checks. When *Privilege Separation* pattern is applied it may provide or create another security layer, which fulfills the goal of the *Layered Security* pattern. On the other hand the same layer may be used multiple times (for example file access rights), even if the *Privilege Separation* pattern is used.

Reference Monitor security pattern [4] can be applied to the privileged entity on our pattern. It defines a process that intercepts all requests for resources and checks if

the requests are authorized or not. When applying *Privilege Separation* pattern the privileged entity becomes a reference monitor for the valuable information, in our example case the server entity (see Figure 1). The *Authenticator* security pattern [5] can also be applied to the privileged entity if the origin of the request needs to be authenticated.

Related Principles

Least privileges security principle should be applied to the resulting entities after the *Privilege Separation* pattern has been applied. *Privilege Separation* pattern supports the *defense in depth* security principle, since it creates new entities and separates their privileges.

Acknowledgements

We would like to thank Eduardo Fernandez for kindly sheperding this paper through and VikingPLoP'05 conference participants for giving very valuable feedback for a pattern writer newbie.

References

- [1] Niels Provos, Peter Honeyman; "Preventing Privilege Escalation"; 12th USENIX Security Symposium Proceedings, 2003; URL: http://www.usenix.org/publications/library/proceedings/sec03/tech/provos_et_al.html
- [2] David Brumley and Dawn Song; "Privtrans: Automatically Partitioning Programs for Privilege Separation"; 13th USENIX Security Symposium Proceedings 2004; URL: <http://www.usenix.org/publications/library/proceedings/sec04/tech/brumley.html>
- [3] Chris Evans, "Probably the most secure and fastest FTP server for UNIX-like systems", URL: <http://vsftpd.beasts.org/>
- [4] E.B.Fernandez, "Patterns for operating systems access control", Procs. of PLoP 2002, <http://jerry.cs.uiuc.edu/~plop/plop2002/proceedings.html>
- [5] E.B.Fernandez and J.C.Sinibaldi, "More patterns for operating systems access control", Procs. EuroPLoP'03, 381-398, <http://hillside.net/europlop/europlop2003/>

Patterns for Software Release Versioning

Klaus Marquardt
Email: pattern@kmarquardt.de

How to version software releases may be an afterthought during development, but they have all the potential to make your life miserable once the software is in production.

This paper covers techniques to identify a particular version, policies to determine version compatibility, and release update strategies. It aims to help the project participants responsible for releases. The affected roles are software architect, release manager, project lead, and product manager. In small projects these roles may be covered by one or two persons.

Introduction

While the software is planned and designed, you think of plans and processes, specifications and delivery dates, architecture and middleware, test and integration, deployment and installation. Versioning releases is often treated as an afterthought. Sure, software is versioned, so what is the point?

There are two points: complexity in size, and in time. All but the smallest systems are a combination of distinct programs or libraries, software developed by different teams and eventually running on different machines. The interoperation of all this code needs to be assured. Over time, the installed base of a software may become significant, and each installation needs to be maintained. Bugs are detected and fixed, new features developed, and the installed base becomes inhomogeneous.

You now enter the domain of not just version identification, but of version interoperation and release management. Which programs can be installed together to function properly, and how is the compatibility checked? And some time more upstream, what is the right granule and time for software items to release?

Luckily, most software systems can survive without deep thoughts on versioning. Implicit and tacit knowledge can bring you so far, but adding just a little more complexity can break your system and requires urgent and careful action. This collection of patterns aims to make the versioning issues explicit, prepare you for the foreseeable, and help you decide what amount of thought to spend when.

Roadmap

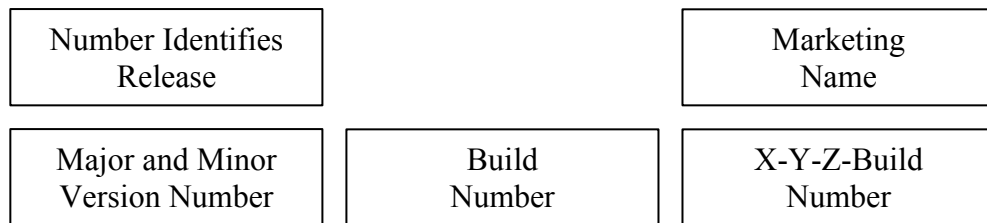
The patterns in this collection have relations to the outside world. Release versioning is virtually pointless without the ability to re-construct any version of the installed base. Configuration management patterns [*Berczuk*] describe some of the essential development practices and processes.

Especially for patch releases and partial releases, many more policies and practices are documented, e.g. in [*Hohmann*]. However, these are based on combinations of patterns from this collection.

The kind and purpose of the software release is covered by the alternative patterns FUNCTIONAL RELEASE, PATCH RELEASE, and TEST RELEASE.



Different policies of version numbers all base on NUMBER IDENTIFIES RELEASE: MAJOR AND MINOR VERSION NUMBER, BUILD NUMBER, and their combination X-Y-Z-BUILD NUMBER. For the public image, numbers are often replaced or complemented by a MARKETING NAME.



Release versions are also used for compatibility checking, either explicitly or implicitly. The common policies are MAJOR VERSION COMPATIBILITY, WHITE LIST CHECK, and BLACK LIST CHECK.



Patterns for finding the right granularity for individually versioned items are thumbnailed in an appendix.

Further patterns in the domain of distribution and installation need to link with these release versioning patterns. Distribution comprises shrinkware, downloads, and auto installation via internet or radio or digital TV. Installation may require user confirmation and interaction, or take place silently.

Acknowledgements

Thanks to Neil Harrison for shepherded this pattern collection to VikingPloP, to all workshop participants at VikingPloP 2005 for their feedback, and to the VikingPloP community, especially to Juha and Sami, for a great venue.

References

- Berczuk* Stephen Berczuk, Software Configuration Management Patterns: Effective Teamwork, Practical Integration. Addison-Wesley 2002
- Hohmann* Luke Hohmann, Beyond Software Architecture: Creating and Sustaining Winning Solutions. Addison-Wesley 2003
- Kelly* Allan Kelly, Business Strategy Patterns for Selling Knowledge with Products and Services. To appear in: Proceedings of VikingPloP 2005

Functional Release

Consider a company developing a software product.

What status of the software should be delivered to the customer?

Forces

Preparing a working software baseline for release requires effort in testing, packaging, marketing, and distribution, but releasing software is the key business of most software companies.

Premature shipping distracts users and may risk a vendors business, but late shipment diminishes the value for the customer and the return on investment.

Solution

Therefore, deliver a new software release when a major functional gain has been achieved.

Finding the balance between the costs associated to a new release, and the revenues at stake when deferring a release and thus costs, is an art in itself. It requires a combination of market knowledge, and financial controlling. The development team members contribute their knowledge of the time and effort it takes to finalize a release for shipment.

Improved

Major achievements become available to the customer.

Premature releases that would increase the cost of ownership over the revenues expected, can be avoided by a cost analysis.

Consider

Functional releases need identification. When a combination of numbers is used, increase the numbers to express the amount of novelty. An increase of a major version number indicates not only major improvements, but typically implies incompatibility to past releases.

For marketing purposes, consider using a Marketing Name to improve the mental identification of the new release.

The business model of the company may combine software releases with further opportunities to sell services or consultancy [*Kelly*].

Service Pack also known as: Patch Release

Consider a versioned product that applies `MARKETING NAME`.

How do you distribute updates that solve problems of a current software release, and identify the distributed versions?

Forces You need to distribute corrections, fixes, and features that are overdue, but marketing fixes is different from marketing new products.

Associations with a software release shall be positive, but admitting bugs in a previous version is painful.

You need a way of identification that is quickly graspable, but that is not confused with new key versions and products.

Solution **Therefore**, bundle several updates into a Service Pack that can be installed with the previous version of the software. Identify each Service Pack by a number, as there might be several of them necessary over the software's lifetime.

The (minor) version numbers of the updated software items need not be visible to external customers. However, document the Service Packs so that the development team can make the mapping and reconstruct the entire source on demand.

Service packs are often bundles that comprise the updated functionality of more than one minor version. Subsequent Service Packs should not require one another, but each should include all updates of the previously released Service Packs.

Since you have an interest that your customers install the Service Packs, make the entire process of distribution and installation as painless as possible. Give the Service Pack for free, using multiple distribution channels such as CD or internet download. Do not acquire consumer data that they might not want to give you. Announce each new Service Pack visibly through all channels of customer relationship management.

Improved Service Packs are an elegant way to fix problems on the client's side.

Your reputation can change for the better, as you actually do care for the installed base.

Consider You need to maintain distinct branches of development to separate fixes (Service Packs) from new features for future releases.

To avoid unnecessary installation, the software release should visibly display not only its version but also the latest installed Service Pack.

Patch Releases and Service Packs follow the same considerations and processes, but can have varying granularity. For a terminology, typical Service Packs are large and contain several different patches.

Test Release

Consider a company developing a software product.

How can you ensure that the product meets the user's expectations?

Forces Developing software is expensive, but shipping software that does not meet the markets needs means spending even more money.

Releasing software is expensive, but only released software has the potential to be evaluated for usability and appropriate functionality.

Solution **Therefore**, prepare a small number of subsequent test releases to a limited number of test users.

Ensure that a test release, just as any other release, can be identified. It is common to use a zero as number, or to name the releases as alpha or beta.

For early test releases, you may want to track where they are tested and used, and prevent unnoticed installation.

Collect feedback for each test release. Only publish a subsequent test release after the feedback has been included into the software development. Make sure that a part of the test users stay in the test release distribution list, so that you have a consistent feedback on your ability to react.

Improved Functional releases for a broad audience are thoroughly tested.

Market acceptance and financial success is more likely.

Consider Even test releases are actual releases that are visible to some of your key customers. They are worth some marketing as they contribute to the public image of the final software product.

For a numbering scheme, consider using `X-Y-Z-BUILD NUMBER` with a `Z` component of zero. Complement this with a `MARKETING NAME` such as “Alpha 2” to adjust expectations about stability – and the influence that feedback can have.

Number Identifies Release

Consider a software product or component release.

All users need a means to refer to a particular release. How do you identify that release?

Forces

Selecting names can be cool, just as your project name is, but a sense of humor seldom scales for different (team) cultures.

Each release is created at a particular date, but exposing the date may induce an impression of staleness, even if an unchanged version means quality.

Any number may do, but you need a sense of which version is newer than another.

Solution

Therefore, identify the release by a unique version number. Increase the number with every release.

Make sure that each release gets its own number, and that numbers are not reused. Do not slip even in exceptional circumstances, like one time creation for your very special customer, or a trade show presentation. Also ensure that even test and trial releases are versioned – every release that might possibly leave the privacy of the development team.

Make this software version number visible on the shipping media and in the software itself. The software should support an API function to retrieve its version number.

Improved

Each release is uniquely identified without confusion.

Each release found in some place can be referenced and reconstructed in source code.

The version information does not exhibit purpose or contents, thus diminishing the potential that users take offense.

Consider

From knowing the official release number, your development environment should enable you to re-create the entire sources. Patterns on software configuration management are available for your support [*Berczuk*].

Plain numbers are boring. They cannot serve for marketing purposes.

Combine this with `MARKETING NAMES` and create a one-to-one relation between the version number, and the combination of `MARKETING NAME` and `SERVICE PACK`.

`BUILD NUMBERS` are a way to distinguish releases for tests and trials from those releases available to the public.

Major and Minor Version Number

Consider a software product or component release.

A single number to identify a release version can serve as a reference and identification. However, how could you convey hints on key features or compatibility?

Forces You need a way of identification that is quickly graspable, but that still conveys a hint of information.

Any number may do, but you need a sense of which version is newer than another.

You want to signal major achievements, but the product still remains the same, covering the same users' needs.

You want to signal minor advances and corrections, but this shall have a different scope than major achievements.

Solution **Therefore**, identify all versioned items with a major and a minor version number. Use positive integers for both.

Increase the minor version number with each release, except when you increase the major number in which case the minor number starts with 0.¹ Increase the major number with major achievements, or to indicate incompatibility.

The major and minor version number strategy is a tricky beast despite its popularity. It evokes associations with respect to compatibility and advances that may not hold. It is never totally clear when to change a major version number, and what implications this has.

When using this numbering scheme, it is tempting to abuse it as a marketing vehicle. Mixing these will cause confusion, so decide whether you focus on marketing aspects, or on compatibility. Technically, it is best applied when you are serious about compatibility and do not need to market the software directly, such as in embedded systems where the software is just among other ingredients.

Improved The numbers you use are more expressive now.

Consider Using the major and minor version numbering scheme is mostly linked with the MAJOR NUMBER COMPATIBILITY. Beware that compatibility needs to be checked, and that the amount of possible combinations grows with the square of minor version numbers.

The assumed expression is by convention only, and cannot be extrapolated into the future.

For alternatives to express technical and marketing aspects, see X-Y-Z-BUILD NUMBER and MARKETING NAME.

¹ Hence the proverb: “never buy a version dot-0!”

Build Number

Consider a versioned product that applies `MARKETING NAME`.

How do you reference the status of the internal development?

- Forces**
- You need to identify a software version, but that identification is not related to features or their marketing.
 - You distribute versions of the software for testing and trials, but only a few of these will become official releases.
 - The version identification shall be visible to the end user for reference, but that identification shall not transport any expectation whatsoever.

Solution **Therefore**, maintain a numerical build number and store this number within the software itself. This Build Number is kept in addition to a `MARKETING NAME` or a `MAJOR AND MINOR VERSION NUMBER`. In a system consisting of multiple versioned software items, each has its own Build Number.

Make this build number meaningless and visible. Meaningless implies that you will not consider this information in a compatibility check, though it will appear in a Bill Of Material [Berczuk]. Visible means that any user of the software can see it and reference it in a mail or during a phone call.

Test and trial versions may have a life beyond your expectation, and distribute themselves in unexpected places. Maintain a database with the Build Numbers that left the development team, and document their release status.

Improved Build Numbers reference a snapshot of the development process, without any implications beyond that the software has been build.

This reference can be exhibited and exchanged in any situation.

Consider Creating a build number adds to the complexity of your tool suite.

`BUILD NUMBERS` are most effective when created automatically. The build process should include incrementing that number and linking or packaging it with the software. A new number can be given anytime and should not be based on quality criteria beyond that the build is a complete one. To ensure it does not transport expectations, use a number that is not related to a date.

X-Y-Z-Build Number

This is a variant to and combination of MAJOR AND MINOR VERSION NUMBER, BUILD NUMBER, and MARKETING NAME.

A major-minor version number evokes associations about features and compatibility. However, how could you actually convey this information?

Forces

You need a way of identification that is quickly graspable, but that still conveys a hint of information.

Any number may do, but you need a sense of which version is newer than another.

You want to signal major achievements, but the product still remains the same, covering the same users' needs.

You want to signal minor advances and corrections, but this shall have a different scope than major achievements.

Solution

Therefore, identify all versioned items with a major and a minor version number, plus a patch level and a build number. Use positive integers for all four parts.

Increase the minor version number with each release that enhances functionality, except when you increase the major number in which case the minor number starts with 0. Increase the major number with major achievements, or to indicate incompatibility.

Increase the patch level when you indicate full compatibility and identical functionality, but a change due to bug fixes. Use the build number as a technical reference to your configuration management system.

Improved

The numbers you use are more expressive now.

X-Y-Z-BUILD NUMBER can support both marketing and technical needs, and thus replace or complement a MARKETING NAME.

Consider

Using the major and minor version numbering scheme is mostly linked with the MAJOR NUMBER COMPATIBILITY. Beware that compatibility needs to be checked, and that the amount of possible combinations grows with the square of minor version numbers.

The assumed expression is by convention only, and cannot be extrapolated into the future.

Marketing Name

Consider a versioned product.

A release with a mere version number is boring, it may even evoke negative associations. How can you brand a release for marketing and give positive hints to potential customers?

Forces

You need a way of identification that is quickly graspable, but that still conveys a hint of information.

Associations with a software release shall be positive, but a major-minor combination is not creating trust.

You want to signal major achievements, but marketing is not interested to signal minor advances or corrections.

Solution

Therefore, use a marketing name for shippable versions. This can be an integer number indicating your long history (“10g”), or relating to your intended shipping date (“Windows 1995”). Make sure that each marketing name has a match into an actual software version.

Finding brilliant marketing names is an art which software engineers are typically not good at. Continuity, novelty, or advancements are not expressed in the same wording and domain as base line numbers and compatibility concerns.

To match one view into the other, a simple technical number is sufficient, such as a BUILD NUMBER. The match should not use a MAJOR AND MINOR VERSION NUMBER as this is more complex than necessary and transports information that somebody needs to put in – and neither marketing nor development is interested.

Improved

Names can be more expressive than numbers.

The domains of technical issues and marketing issues are separated.

Consider

You need a mapping between both domains.

Mixing a marketing name with a numerical identification can create a mess, when users expect features or compatibility based on the numbers, that the actual software does not provide.

Combine the MARKETING NAME for major advances with a different strategy to release bug fixes and corrections, like SERVICE PACKS.

Note that the MARKETING NAME does not need to match the project name, even though the project name may be known to the public prior to the shipping date.

Major Version Compatibility

Consider a system of several components.

Which combination of versioned items can you consider compatible?

Forces

You cannot assume that any combination of software interoperates smoothly, but users implicitly expect compatibility of updates and new versions.

Similar versions can cooperate, but major differences in features will limit the compatibility.

You can determine the compatibility during development, but a run time check increases the chances to identify conflicts.

The compatibility of different software items better be stated explicitly, but the number of possible combinations grows non linear.

You want to be able to incrementally update particular software portions, but you need to test all combinations that can occur.

Solution

Therefore, use the MAJOR AND MINOR VERSION NUMBERS for compatibility checking. Assume all versions of the same major version number compatible, regardless of their minor version.

While this assumption is easy to check by the installed software components, it is hard to achieve during development. All possible combinations must be tested, at least in a pair wise approach. Except where legally required, you do not need to test triangle and more complex settings, as their failure probability is very low.

Improved

The compatibility check is technically easy and can happen at run time.

The decision on compatibility is easy to understand and matches the users' expectations.

Compatibility matches to similar features, bug fix releases are considered compatible.

Consider

There is no guarantee that only thoroughly tested combinations of software become operational.

The decision about compatibility is typically not based on thorough testing of all combinations, but on implicit assumption.

The MAJOR VERSION COMPATIBILITY strategy does not scale for very many minor releases, especially when a large number of items is involved. It is often combined with a strategy to limit the number of minor releases to less than a hand full.

White List Check also known as: Positive Check

Consider a system of several components

How can you check for version compatibility, when you cannot rely on the implicit understanding of a MAJOR VERSION COMPATIBILITY?

Forces You cannot assume that any combination of software interoperates smoothly, but users implicitly expect compatibility of updates and new versions.

Similar versions can cooperate, but major differences in features will limit the compatibility.

You can determine the compatibility during development, but a run time check increases the chances to identify conflicts.

The compatibility of different software items better be stated explicitly, but the number of possible combinations grows non linear.

You want to be able to incrementally update particular software portions, but you need to test all combinations that can occur.

Solution **Therefore**, maintain a list that includes tested combinations of different items. Check whether the actual versions are listed as compatible. Reject interoperation when the combination is not listed.

The POSITIVE CHECK strategy comes with the understanding that every combination that is not explicitly mentioned as compatible, is assumed to be incompatible. This emphasizes the importance of testing and verification and prevents unintended slips in the release procedure.

The information which versions cooperate can be maintained outside the software and have independent distribution channels. In systems where some parts are movable or exchangeable, the allowance information might not include the actual combinations although they have been tested already. Thus, an update channel for the allowance data is as important as the distribution of the software itself. If fraud prevention is essential, the allowance information needs to be encoded.

Improved The compatibility check is technically easy and can happen at run time.

Only thoroughly tested combinations of software can become operational.

The release process is not bound to the development but to QA.

Consider Installations done by end users have a higher risk to refuse operation.

A second type of data needs to be distributed.

Black List Check also known as: Negative Check

Consider a system of several components

How do you conveniently check for version compatibility, when you are still in your testing phase?

Forces

You can assume that most combination of software interoperate, but you need a way to exclude those combinations that you know to be incompatible.

You can determine the compatibility during development, but a run time check increases the chances to identify conflicts.

You want to be able to incrementally update particular software portions, but you need to test all combinations that can occur.

The compatibility of different software items better be stated explicitly, but for just trying something in your own environment you avoid significant administrative overhead.

Solution

Therefore, assume compatibility, and define those cases where you know that versions do not interoperate. List these failure cases in a negative list, and check for their appearance.

The `NEGATIVE CHECK` strategy comes with the understanding that every combination that is not explicitly mentioned as incompatible, is assumed to be compatible. This reduces the importance of formal testing and verification and enables an informal release procedure by the developers.

Improved

The compatibility check is technically easy and can happen at run time.

Software to be tested can quickly become operational.

Software known to be erroneous in combination, can be excluded from operation.

Consider

There is no guarantee that only thoroughly tested combinations of software become operational.

Installations done by end users have a higher risk to malfunction, possibly even unnoticed.

As the `NEGATIVE CHECK` strategy can lead to unexpected behavior when not carefully controlled, depending on the usage in the field it needs to be replaced by a more strict checking strategy.

Appendix: Patterns for the Granularity of Versioned Items

Package Version

How do you avoid to drown in a large number of versions, and enable effective checking?

Version each package as a whole. Limit the number of visible and checkable version numbers to about a dozen or less.

This pattern is common practice with components and plug-ins.

Data Version

Consider a system containing many different components that interoperate via data exchange. How do you enable effective compatibility checking?

Version the data structures that are shared between different software portions. Assume that all relevant behavioral aspects are expressed in the data itself.

This pattern is common practice in database centric host systems, and XML based enterprise systems.

Item Version

Consider many different devices from different vendors and of different versions. All these devices need to interoperate at a basic level. How do they know what to expect from each other?

Version each single item of exchange individually.

This pattern is common practice in widely used networks, such as messages in telecommunication protocols.