vikingPLoP
Bergen 2007

Proceedings of the Sixth Nordic Conference on Pattern Languages of Programs

Edited by: Cecilia Haskins

Viking PLoP 2007, Proceedings of the Sixth Nordic Conference on Pattern Languages of Programs, Bergen, 29-30th September, 2007

Edited by Cecilia Haskins

Copyright © 2007. All rights reserved. Authors retain copyrights of their respective papers.

PLoP is a registered trademark of Hillside Group. The names of actual companies and products mentioned herein may be the trademarks of their respective owners.

For more information about VikingPLoP please visit www.vikingplop.org.

CONFERENCE PROCEEDINGS OF THE SIXTH NORDIC CONFERENCE ON PATTERN LANGUAGES OF PROGRAMS VIKINGPLOP

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 an annual pattern conference in Scandinavia, to encourage people in Scandinavia, who might not otherwise attend a PLoP® conference, to learn about patterns.

The first VikingPLoP was held in Højstrupgård, Denmark, in September, 2002. The conference was primarily structured around writers' workshops, supplemented by focus groups; one for upcoming shepherds of patterns.

Since then a VikingPLoP conference has been held in a Scandinavian country with rotating locations and conference chair duties.

2002: Helsingør, Denmark
2003: Bergen, Norway
2004: Uppsala, Sweden
2005: Helsinki, Finland
2006: Helsingør, Denmark
2007: Bergen, Norway

Conference Chairs

2002: Pavel Hruby and Kristian Elof 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 Elof 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

The Shepherding Award

Patterns are the essence of the PLoP® conferences. The shepherding process improves the quality of patterns submissions. Shepherds usually invest a lot of personal time and effort. And, while it usually is a rewarding process both for shepherd and author, it also has its challenges. An award was created to thank the many people who serve as shepherds. 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 have the right to nominate a shepherd. The award recipients are recognized here.

Recipients of The Neil Harrison Shepherding Award at VikingPLoP

2002: Linda Rising

2003: Alan O’Callaghan

2004: Cecilia Haskins

2005: Klaus Marquardt

2006: Andreas Rüping

2007: Allan Kelly

PATTERNS FROM VIKINGPLOP 2007

A Pattern Language for Sustainable Industrial Parks[1]

Proceedings for the 6th Annual Nordic PloP® Conference, Bergen, Norway, 28-30 September, 2007

Cecilia Haskins, Norwegian University of Science and Technology
cecilia.haskins@iot.ntnu.no

Dear reader,

This pattern language is written as part of the author's dissertation on applying systems engineering to the creation and maintenance of sustainable industrial parks. The patterns have emerged from, and are significantly substantiated by the extensive literature on this subject. The choice of insights in this pattern language derives from personal observations of Verdal, a small industrial Norwegian community, and from the many stories that the author has been told while working with the people there. At this time most of the pattern language exists as "patlets" that require refinement and maturity. The more mature patterns are submitted for this workshop.

Formatting conventions used in this work: pattern names that appear in SMALL CAPS belong to this pattern language; pattern names that appear in *Italics* are from a referenced source.

The patterns are structured in a framework as illustrated in Figure 1. The framework is consistent with that used by Daniel May in his Patterns for Building the Sustainable Organization.[2] Alexander recognized that people have a life cycle with eight stages (*Life Cycle*, Alexander, et. al., 1977). Stage seven is the Adult. The Adult is identified with a need for a *Work Community* with *Self-Governing Workshops and Offices*, and access to *Communal Eating.* The pattern *Industrial Ribbon* discusses geographic placement of an industrial park.

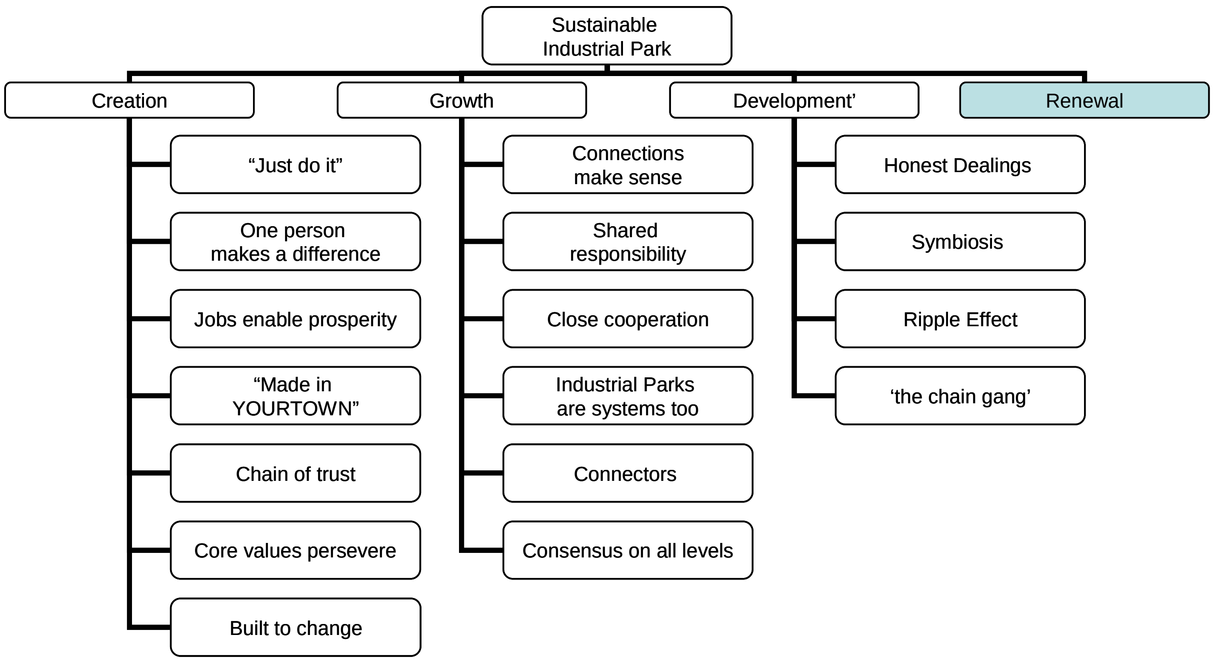


Figure 1 – Framework for Sustainable Industrial Parks Pattern Language

[1] © Haskins, 2007 – permission to use with attribution
[2] Daniel Chien-Meng May, 2001, Patterns for Building the Sustainable Organization, Australian Conference on Pattern Languages of Programs, Melbourne, Australia, http://www.thedanielmay.com/publication.php?id=23

Sustainable industrial park

Most readers are familiar with the industrial revolution. It is a concept that represents the social, political and economic changes that emerged as people left their farms and cottage industry (home crafts) to flock to the (urban) factories. Stories from the earliest days expound the horrors of child labor and sweat shops, where man was subservient to machine, and the factories spewed smoke and debris into the air and water that their workers would breath and drink. Gradually these conditions have improved, but there is still room for improvement in the way most companies interact with their people and the natural environment. For example, many nations have recognized the need to encourage the development of low-carbon or no-carbon technologies needed to wean their economies off fossil fuels. In theory, the wealth and the jobs created by these 'green' technologies should help to offset the costs of reducing carbon emissions.

To qualify as sustainable, an industrial park must focus individually and as a whole on the sustainable development triad – equity (in dealing with society), economic prosperity (for the company and the community), and stewardship for the natural environment (protecting the legacy of future generations). The principles of sustainability require that attention is paid to economic, environmental and social issues with equal weight. Many consider the concept of sustainable development to be an oxymoron – a desirable but unachievable ideal. Principles of industrial ecology, in which industry mimics nature, are often applied to the discussion of stewardship. Figure 2 illustrates the progression of change required to move to a complete industrial ecology.

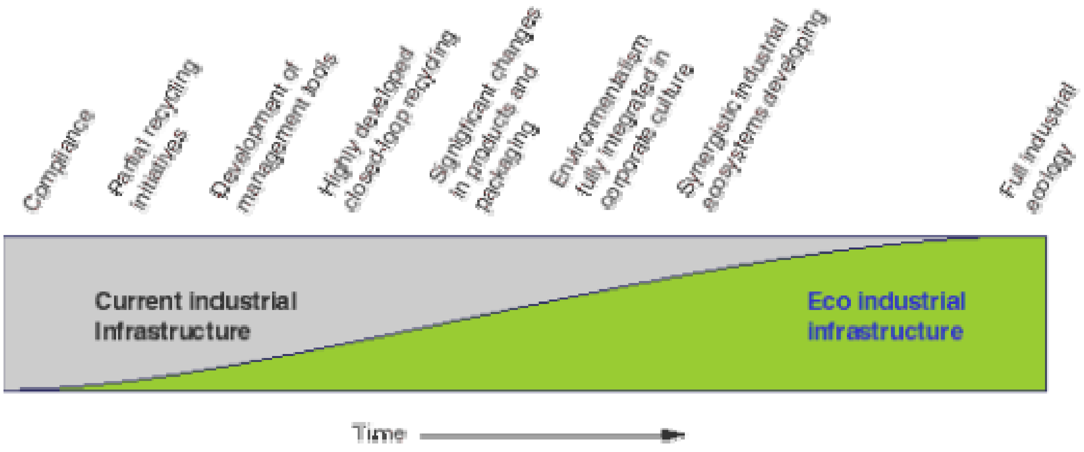

Compliance
Partial recycling initiatives
Development of management tools
Highly developed closed-loop recycling
Significant changes in products and packaging
Environmentalism fully integrated in corporate culture
Synergistic industrial ecosystems developing
Full industrial ecology
Current industrial Infrastructure
Eco industrial infrastructure
Time

Figure 2 – Stages of progress toward full industrial ecology (Little, 1994)

Industrial ecology employs the metaphor of natural ecologies to industrial (or human made) systems. Using this metaphor, the life cycle of an industrial park or other social community[3] can be viewed in the same way as the stages of a forest or other natural system. These four stages can be summarized as Creation, Growth, Development, and Renewal. Patterns for a sustainable community have been categorized, although many patterns are applicable throughout the entire life cycle, such as CHAIN OF TRUST.

[3] The term community here is used in many layers – from a group of people living in a city, to a region, country or even planetary scope.

During the creation stage, many ideas are generated; there is an abundance of energy, and a desire to make something happen (JUST DO IT). It is helpful during this phase to create a shared vision of how the industrial park will look in the future. Trial and error teaches what works, and this learning is applied during the growth stage. The participants begin to understand the role they play in the overall context (SHARED RESPONSIBILITY). The development stage is typified by more maturity, continued growth, and institutionalization of 'what works' such that success breeds success (RIPPLE EFFECT).

But success also breeds the conditions that constrain unlimited development. Eventually, the status quo is threatened by the need for change and renewal begins. In nature, a fire that destroys a section of the forest but allows dormant seeds to burst out when touched by the sun would be an example of this destruction and renewal phase.

At the end of a life, every entity experiences a form of creative destruction that eventually leads to renewal. McDonough[4] has coined the phrase 'Cradle to Cradle' to express this concept for product development. Alexander recommends encouraging a birth and death process for towns in his pattern '*The distribution of towns'* (Alexander, et. al., 1997). Time and experience have not yet given the author enough insights into this phase of the community life cycle, notwithstanding many real world examples. A related organizational pattern called *Transforming Institutions* can be found online.[5]

Patlets for the patterns that have emerged during the research are given in the tables that follow. The patterns have been allocated to a life cycle stage based on when they are most used or first needed. Pattern names in bold text are written and submitted to this workshop.

Acknowledgements

Sincere thanks to Linda Rising, whose many insights as shepherd have helped the author to improve the work; and to my advisor, Professor Annik Magerholm Fet, without whose encouragement this work might never have been conceived.

References

Christopher Alexander, Sara Ishikawa, Murray Silverstein. *A Pattern Language: Towns, Buildings, Construction*. New York: Oxford University Press, 1997.

Arthur D. Little, "Industrial Ecology: An Environmental Agenda for Industry," Industrial Ecology Workshop: Making Business More Competitive. Toronto: Ministry of Environment and Energy, February 1994.

[4] http://www.mcdonough.com/cradle_to_cradle.htm

[5] http://diac.cpsr.org/cgi-bin/diac02/pattern.cgi/public?pattern_id=398

Table 1 – Creation Patterns

"JUST DO IT"	What is the balance between theory and practice?
ONE PERSON MAKES A DIFFERENCE	Each person makes a contribution, but without a visionary, can progress be achieved? See also *Evangelist* from Fearless Change.[6]
JOBS ENABLE PROSPERITY	Employment is important to prosperity and well-being; in a non-agricultural setting, this means people need jobs provided by firms.
"MADE IN YOURTOWN"	Is there a definition of sustainable industrial park? See also *Beautiful Purpose* from Patterns for building a Beautiful Company.[7]
CHAIN OF TRUST	How reliable are the relationships in a business or social network? Who should be trusted?
CORE VALUES PERSEVERE	Longevity is desirable – what do firms need to do so they do not collapse before their time?
BUILT TO CHANGE	Survival is tightly linked to adaptation. See also *Genius of the AND* from Patterns for Building the Sustainable Organization[8]

Table 2 – Growth Patterns

CONNECTIONS MAKE SENSE	Will business or social networks exist under all conditions?
CLOSE COOPERATION*[9]	Can members of an industrial park share resources if all actors behave independently? See *It's a Relationship, not a Sale* from Customer Interaction Patterns.[10]
SHARED RESPONSIBILITY*	Can an industrial park achieve sustainability if each actor does not accept full responsibility for their contribution to the goals? See also *Culture that Grows* from Patterns for Building the Sustainable Organization[11]
INDUSTRIAL PARKS ARE SYSTEMS TOO	Can the principles of systems thinking contribute to our understanding of Industrial Parks? See also de Geus, The Living Company.[12]
CONSENSUS ON ALL LEVELS	Can a single industrial park maintain sustainability in isolation without local, regional and national cooperation? See also *Voices of the Unheard*[13]
CONNECTORS	Where do new relationships and partners come from? How are they identified?

Table 3 – Development Patterns

HONEST DEALINGS	How much are companies willing to disclose, and to whom?
SYMBIOSIS	In natural ecosystems, when two creatures depend on each other for survival this is called symbiosis.
RIPPLE EFFECT*	What positive influence do businesses and citizens have on each other?
THE CHAIN GANG	Are firms permanently shackled to prior commitments?

[6] http://www.cs.unca.edu/~manns/Summaries.pdf
[7] http://www.vikingplop.org/vikingplop2002/VikingPLoP2002_Proceedings.pdf
[8] http://www.thedanielmay.com/publication.php?id=23
[9] *suggested by Edwards, A. (2005). The Sustainability Revolution. Gabriola Island, Canada: New Society Publishers
[10] http://www.lindarising.org/
[11] http://www.thedanielmay.com/publication.php?id=23
[12] http://www.sfu.ca/~dmarques/federation/pdf/deGeus-TheLivingCompany.pdf
[13] http://trout.cpsr.org/program/sphere/patterns/pattern-form-display.php?pattern_id=76

CHAIN OF TRUST

... your firm belongs to many networks – professional, supply chain, community.

❖ ❖ ❖

How reliable are the relationships in a business network? Who can be trusted?

The interactions of a firm are seldom one person to one person. In supply chains, firm A contracts with firm B, who in turn contracts with firm C, creating a dependency between firms A and C, even if they have never had any direct dealings. Business contracts are made with an implicit understanding that the terms will be satisfied – deliveries will be punctual, accurate and for the agreed price. Follow-on business depends on reputation and reliability. Long-term contracts become a liability if any link in a supply chain fails to perform.

Therefore:

Build a chain of trust.

Observe the reliability of your vendors' vendors, and the integrity of your customers' customers. In business relationships firm A must not only trust firm B's ability to meet the terms of an agreement, but also to enter into equally reliable relationships that will not jeopardize the commitments made – thereby creating a chain of trust. Rising Customer Interaction Patterns contain a pattern *Build Trust* that also addresses ways to build trusted relationships. She also co-authored *Organizational Integrity*, a pattern about preserving integrity in the face of real-world situations.

Professional or community relationships may involve less risk because of the possibility for direct interpersonal contact in the event of a misunderstanding.

❖ ❖ ❖

The following are examples of CHAIN OF TRUST.

Wow-Media in Verdal operates as an advertising agency but is actually composed up of six separate small firms, each with their own specialization. They have joint contracts that determine the financial arrangements, but day-to-day assignments are handled in a more ad hoc manner, based on work load and other commitments. These six people work closely together, sharing office space and expenses, as well as revenue, and are beginning to gain a loyal customer base for the agency.

Williamson describes a form of agreement he calls ‘relational contracting’ in which two firms place more value in preserving their relationship than in enforcing punitive settlements in the event something goes wrong. Within such complex and long-term contracts, two or more partners will forge a chain of trust that supersedes the legal bond.

A recent USA TODAY/CNN/Gallup Poll asked the question, “Who do you trust?” People who own small businesses came in second with a 75% trust rating. On the other hand, right near the bottom of the list were chief executive officers of big corporations, with a mere 23% of respondents trusting them. The poll reflects the almost-daily revelations of corporate misdeeds, but there's more to it than headlines. Small-business owners better understand the importance of trust to the survival of their business. All of us who run small companies know that without trust, we wouldn't be in business long. Our customers don't do business with us because we have a brand name or even the lowest price; we have to connect with them in a more personal way. Most of us still do deals based on our word or a handshake; if we had to put everything in writing, our business would come to a standstill. In my 16 years in business, I've only had one client not pay their full bill. It comes down to this: If people don't trust you, you don't have a business.

Sometimes a company needs to work with their competitors. *The Dangerous Waterhole* pattern illustrates why competitors might want to cooperate in a process to develop a new technology or standard. Imagine a watering hole in a jungle. Animals eye the hole from a distance, but not one goes forward, lest it fall prey to another beast lurking behind the trees. But once an animal large enough or otherwise immune from sneak attacks descends on the watering hole, other animals soon follow suit. A similar situation might draw the world's premier widget manufacturers to the same watering hole. However, they are unlikely to initiate a common interface standard on their own; after all, why would competitors follow one company's proposal? Thus, little progress is made, until a neutral third party invites those companies to work together. A competitor that refuses to join runs the risk of having its own interests ignored.

❖ References ❖

Excerpt from Rhonda Abrams, How to build trust, USA Today. http://www.usatoday.com/money/smallbusiness/columnist/abrams/2002-07-19-trust_x.htm.

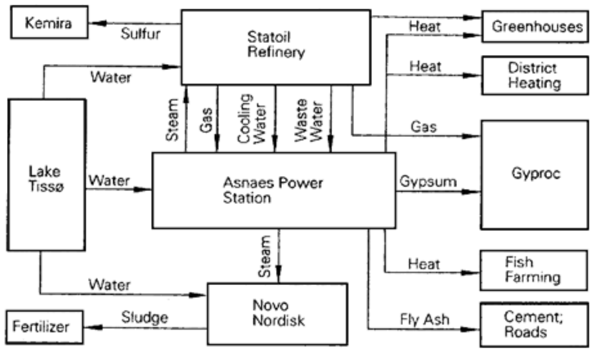
Gabriel, R. and Ron Goldman, (1998). The Jini Community Pattern Language, available online from www.dreamsongs.com/NewFiles/JiniCommunityPL.pdf.

Rising, L., (1998). Customer Interaction Patterns, available online from www.lindarising.org.

Rising, L., et. al, (2002). *Organizational Integrity* in Beautiful Company Pattern Language, available online from www.lindarising.org.

Williamson, O. E., (1985). The Economic Institutions of Capitalism. New York: The Free Press.

CONNECTIONS MAKE SENSE



Schematic of industrial symbiosis in Kalundborg, DK

... a firm is building relationships within an industrial business network.

❖ ❖ ❖

How does a firm decide whether to pursue a relationship?

Building a relationship takes time and energy; time, to evaluate trustworthiness, and energy, to follow up on negotiations and joint activities. Not every potential relationship will deliver the same value – not every relationship makes sense.

But relationships are important to the survival of organizations. A business network can help individual firms share risk, reduce costs, and achieve flexibility by providing access to special skills or smoothing demand curves. New sales can be realized from a network of extended contacts.

Therefore:

Focus time and energy on connections that derive value – make sense.
Some relationships are mandated, such as connections with governmental and regulatory organizations. Others are a matter of choice, such as suppliers, subcontractors and partners. Good relationships are based on win-win negotiations which display a balance between effort expended and benefits realized. These benefits can take the form of tangible economic value to the firm or intangible values such as happy employees or clean air, which share the benefits with the stakeholders of the firm. Connections may result in long-term contracts or mergers. See the discussion of *Dangerous Waterhole* above for a description of situations where working with the competition is what makes sense.

Additional discussion and examples are provided in the Patterns for Building the Sustainable Organization – see *Constant Exploration* and *Genius of the AND.*

❖ ❖ ❖

The following are examples of CONNECTIONS MAKE SENSE.

Erling Pedersen, CEO of the Industrial Development Council in the Kalundborg region wrote in 1999, “There was no original joint management … the network did not evolve with any academic knowledge of scientific environmental network theories, but as good economical management practices. All projects required investments and resulted in revenues or savings for the parties involved.” The employees of these firms were looking for innovative ways to reduce the cost of waste treatment and disposal, to find cheaper input materials and energy, and to generate income from their own production by-products [Desrochers, 2001].

In 1999, in the face of imminent layoffs that would have devastated the local community, the management of Aker Verdal initiated a series of fissions that divided the company from a large entity into many small companies. As a result, there are more external relationships to manage, but the employees are buffered from fluctuations in the offshore oil industry by working for smaller firms that provide services to Aker Verdal, as well as other customers. [Kvarsvik, 2002] However, these actions averted more adverse consequences of an economic downward spiral.

"Wharton management professor Harbir Singh, who has done extensive research on mergers, says that the crucial distinguishing factor between success and failure in a merger is a sense of objectivity on the part of executives -- a "realistic outlook" that needs to be maintained from the initial transaction through the entire integration process. The danger, it seems, is when executives "fall in love" with the idea of the acquisition, wanting it to work no matter what the cost.” [Sikor, 2005] This excerpt from the article offers an insight into the danger of pursuing a relationship that does not make sense – which happens often enough that in the USA, “a cottage industry of sorts has emerged to help companies navigate the rough terrain of integration -- and especially to help them overcome the internal inertia that comes with facing change.” [ibid]

❖ References ❖

P. Desrochers, Cities and Industrial Symbiosis: Some Historical Perspectives and Policy Implications, Journal of Industrial Ecology, Fall 2001, pp. 29-44.

A. Kvarsvik, Omstilling ved Aker Verdal: fra en stor til mange små. Hovedfagsoppgave, Geografisk Institutt, NTNU, Trondheim, Mai 2002

Daniel Chien-Meng May, 2001, Patterns for Building the Sustainable Organization, Australian Conference on Pattern Languages of Programs, Melbourne, Australia, available from http://www.thedanielmay.com/publication.php?id=23

M. Sikora, Why do so many mergers fail? Knowledge@Wharton, March 30, 2005, http://knowledge.wharton.upenn.edu/article.cfm?articleid=1137

CONNECTORS

... a firm or community are ready to grow.

❖ ❖ ❖

Where do new relationships and partners come from? How are they identified?

There are many criteria for entering a business relationship as indicated by CHAIN OF TRUST and CONNECTIONS MAKE SENSE. Often the skills that make a good business leader are not the same as those that make a good marketer, or finder of new opportunities. Firms and industrial communities need help to find new suitable partners.

Therefore:

Identify a 'connector' – a person with a strong personal network and the ability to persuade. This person will attract potential firms to the industrial park.

This pattern is identified and well defined by Manns and Rising in *Fearless Change*. Connectors are people who have relationships that make them effective in reaching out to other people. This phenomenon is also extensively discussed in Gladwell's *The Tipping Point*. He describes Connectors as, "… people with a special gift for bringing the world together." The best way to find a connector is to observe your acquaintances and how many different communities they move in comfortably. People who gravitate to sales or similar jobs are often good connectors, or know people who are connectors.

Two other patterns also offer additional insight into CONNECTORS: *It's a Small World*, from the Pattern Language for Building a Beautiful Company, and *Gatekeeper* from Organizational Patterns for Agile Software Development.

❖ ❖ ❖

The following are examples of CONNECTORS.

When it was clear that 400 jobs would be obsolete, the CEO of Aker Kværner Verdal tapped Pål H. to start up an incubator. He understood that Pål had both the business acumen and the personality to identify and mentor potential entrepreneurs. Pål has built an organization that includes other connectors who are equally effective at bringing in new contracts and new tenants for the Verdal Industrial Park and managing those relationships.

Gladwell tells a story about meeting business tycoon Roger Horchow, a Connector. He found that while his many connections had been helpful in his professional endeavors, collecting people was not part of a business strategy, it was just who he was. Roger simply likes people and has “an instinctive and natural gift for making social connections.”

In an article on networking, Keith Ferrazzi discusses the research of Mark Granovetter on social networks. Granovetter found that professionals rely primarily on their set of personal contacts to get information about job-change opportunities over using a search firm or reading the newspaper (or internet in today’s society). Fifty-six (56) percent of those surveyed found their current job through a personal connection. Only 19 percent used what we consider traditional job-searching routes, like newspaper job listings and executive recruiters. Roughly 10 percent applied directly to an employer and obtained the job.

As a result of the study, Granovetter immortalized the phrase "the strength of weak ties" by showing persuasively that when it comes to finding out about new jobs-or, for that matter, new information or new ideas-"weak ties" are generally more important than those you consider strong.

According to Ferrazzi, it pays to talk to strangers. “Many of your closest friends and contacts go to the same parties, generally do the same work, and exist in roughly the same world as you do. That's why they seldom know information that you don't already know. Your weak ties, on the other hand, generally occupy a very different world than you do. They're hanging out with different people, often in different worlds, with access to a whole inventory of knowledge and information unavailable to you and your close friends.”

❖ References ❖

Connecting with Connectors by Keith Ferrazzi, downloaded from http://www.fastcompany.com/resources/networking/ferrazzi/032105.html

Coplien, J. O., and Harrison, N., (2004). Organizational Patterns of Agile Software Development. Upper Saddle River, NJ: Prentice Hall.

Gladwell, M., (2000). The Tipping Point: how little things can make a big difference. New York: Little, Brown and Company.

Granovetter, Mark. 1974. Getting a Job: A Study of Contacts and Careers. Chicago: University of Chicago Press.

Manns, M. L. and L. Rising, (2004). Fearless Change: patterns for introducing new ideas. Boston: Addison-Wesley.

Rising, L., et. al., (2002). Patterns for Building a Beautiful Company, downloaded from http://www.lindarising.org/ and http://www.thedanielmay.com/publication.php?id=27

JUST DO IT!

... it is time to take action – but what action?

❖ ❖ ❖

What is the balance between preparation and practice?

Decision-making is an important business activity. Decision-making takes time – time that could be used doing other work. A common scenario is frequent iterations of data collection, planning and brain storming, and planning and brain storming, and …… Then after deciding on a course of action, thinking of all the reasons why something won't work, and start the cycle again.

In addition, sometimes it does not make sense to wait to achieve a 100% agreement through CONSENSUS ON ALL LEVELS. Occasionally the timing of a decision may mean the difference between success and failure.

Therefore:

Follow the advice of one of the greatest marketing campaigns in corporate history and "Just do it!"

Time spent agonizing over decisions is time lost. There is a middle ground between reckless abandon and falling prey to the paralysis of analysis. Theory is important, but it should inform practice, not replace it.

Fearless Change contains a pattern of the same name with additional discussion; the name of the pattern has been reused with permission. Daniel May has written Patterns for Building the Sustainable Organization. The pattern *Start from a Grain* contains similar advice with suggestions for start-up activities and additional examples where this pattern has been applied. A similarly named pattern, *Get on with It*, describes this situation for software development projects.

In situations where the information is fuzzy or the issues are not defined well enough to support a decision, Cockburn recommends, in the pattern *Clear the Fog*, taking some action and then learn from the feedback or consequences what the real issues are before taking action.

❖ ❖ ❖

The following are examples of JUST DO IT!

The best illustration of this pattern is found in the comparison of accomplishments in each of two Norwegian communities after four of six years of special financial support from the state. Both communities are in the same region, within 30 km of each other, and faced significant job-losses at the end of the 1990's. Community A used the time to develop fancy templates for applications for funding and documents filled with theoretical advice. After the first four years, Verdal had seen numerous small business start-ups – both successes and failures – and every laid-off employee had found a new employer, if they were not actually the boss.

This online advice is offered to small business owners: "*All too often when speaking with business owners who are worried about the success of their business, I find the answer to their problems is: Just do it! Many entrepreneurs believe they need things to be perfect before they can implement a marketing campaign. Instead they should test the campaign and perfect it based on the results of the test. To test, you have to take action and just do it... A survey of top entrepreneurs asked, 'What is the biggest regret you have in starting and running your business?' The answer 97% of the time had to do with something they didn't take action on rather than something they did take action on. This tells us that even if we take action and make a mistake, it's better than not doing anything at all.*"

Eric Stolterman offers some reflections on life in a small Swedish village. "*The information system that makes this community work is very simple. There is a list of all inhabitants, divided into working groups. This list is sent out once a year. We get a note in our mailbox maybe four times a year with announcements of the time and place for activities. These notes are sent out by the responsible working group. If we measure the success of our community by what we accomplish I think most people would rate it as an active and successful community. If the activity was measured by the amount of information flows – the community would probably be rated as almost non-existing.*" This quote illustrates the importance of doing over talking about doing.

❖ References ❖

Cockburn, A., (1998). Surviving Object-Oriented Projects: A Manager's Guide. Reading, MA: Addison-Wesley.

Coplien, J. O., and Harrison, N., (2004). Organizational Patterns of Agile Software Development. Upper Saddle River, NJ: Prentice Hall.

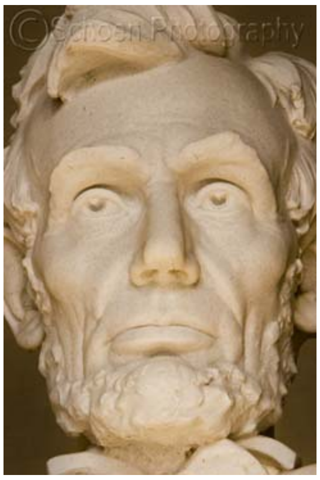
Manns, M. L. and L. Rising, (2004). Fearless Change: patterns for introducing new ideas. Boston: Addison-Wesley.

Daniel Chien-Meng May, 2001, Patterns for Building the Sustainable Organization, Australian Conference on Pattern Languages of Programs, Melbourne, Australia, downloaded from http://www.thedanielmay.com/publication.php?id=23

Just do it! Advice to the small business community. Downloaded from http://www.smallbusinessconsulting.com

Stolterman, E. (1999). Activity and Community some Personal reflections , (Accepted for presentation at the workshop at ECSCW in Copenhagen, September, 1999.), downloaded from http://www.scn.org/tech/the_network/Proj/ws99/stolterman-pp.html

HONEST DEALINGS



... your firm operates in a CHAIN OF TRUST.

❖ ❖ ❖

How much information sharing is necessary to maintain a business network?

Decision-making is complicated – multiple criteria, multiple stakeholders, uncertain consequences. The difficulty is further exacerbated when a decision spans a business network, across multiple organizations / firm boundaries. Imperfect information is always a trade-off between the time and cost of collecting more data against the urgency of the decision. Often information sits in many individual heads and is not unknown. The challenge is to bring all relevant information together when it is needed.

Therefore:

In a business network, no information relevant to meeting commitments should be kept secret.

Organizational literature is filled with buzzwords such as business intelligence and knowledge management. Somewhere within these concepts, and enabled by increased interconnectivity afforded by the information age, a technological solution may be formed. Alternatively, more human centric approaches such as geographically co-located teams are also feasible.

The issue that is at the heart of this matter is how comfortable is each firm with sharing information that in the past may have been perceived as proprietary, confidential or competitive advantage.

❖ ❖ ❖

The following are examples of the importance of HONEST DEALINGS.

The CEO of Aker Verdal recognized that the business needed to change direction and become leaner to remain competitive on the international market. To keep the facility open in the long run meant that 400 people needed to leave the company, and many of those who remained needed retraining. Rather than keep this information to himself, he began preparing the employees and the local community to deal with this change. As a result of his honest mode of dealing with the trade unions, and the local government, everyone took an initiative to contribute to mitigating this necessary layoff. As a result, within two years, all 400 employees had found new employment – some as owners of their own company – and additional new jobs were created in the process.

BROWNSVILLE, Texas - This border community has long shared knowledge and cultural ties across the border with its sister city of Matamoros, and vice versa. So it should be no surprise that negotiations are now taking place on both sides of the border to develop the relatively new concept of eco-industrial parks. One Mexican company - Quimica Flor - produces gypsum as a by-product of its industrial production of hyfluoric acid, potentially an ideal candidate for reuse and recycling in other applications, including road building materials and plaster. However, before any business arrangements can be made, the composition of the gypsum as well as the industrial processes needs to be studied in depth. "We are coordinating our efforts," Lockett said. "We have to. Information that can benefit the border environment and economy has to be shared bi-nationally."

From the Chinese Guidelines for eco-industry park planning
"An eco industrial park has the following main characteristics: ... (3) Ensuring the steady and sustainable development of the industrial park through the application of modern administration, policy and new technology such as sharing information, saving water and energy, re-circulation and reuse, environment monitoring and sustainable transportation technique." Sharing information has been an important focus of the Chinese Circular Economy initiatives.

❖ References ❖

Mader, R. (1995). Eco-Industrial Park. Downloaded from http://www.planeta.com/ecotravel/border/0095industry.html

The guidelines were formulated by China's State Environmental Protection Administration for guidance and evaluation of EIP projects. The document is dated 2003-12-31 and is available at http://www.indigodev.com/sepa_eip

❖ Photograph ❖

Legends are often untrue, but Abraham Lincoln was honest. During his years as a lawyer, there were hundreds of documented examples of his honesty and decency. He was known at times to convince his clients to settle their issue out of court, saving them a lot of money, and earning himself nothing.

According to one account, an impoverished old widow of a Revolutionary soldier was charged $200 for getting her $400 pension. Lincoln sued the pension agent and won the case but he didn't charge her for his services and, in fact, paid her hotel bill and gave her money to buy a ticket home!

Photo downloaded from: http://www.pbase.com/kjschoen/favorites

Data Compatibility in Dynamic Environments with Extension Points

Birthe Böhm, Norbert Gewald
Corporate Technology, Siemens AG
Günther-Scharowsky-Str. 1, 91050 Erlangen, Germany
{birthe.boehm, norbert.gewald}@siemens.com

Gerold Herold
Medical Solutions, Siemens AG
Hartmannstr. 16, 91050 Erlangen, Germany
gerold.herold@siemens.com

Dieter Wißmann
Department of Electrical Engineering and Computer Sciences
University of Applied Sciences Coburg
Friedrich-Streib-Str. 2, 96450 Coburg, Germany
wissmann@fh-coburg.de

Intent

With this pattern it is possible to extend a data model whilst staying compatible with the original data model and leaving all the related software components intact and in conformance with previous versions or standards.

That is, it is possible to adapt a data model to future requirements without the need to adapt the whole system to the changed data model as a consequence.

Example: Consider an automotive company that has a data model for the customization and production process of a car in which a data type CarFeaturesT is included. CarFeaturesT describes a customer's selections when a new car is ordered and gives information for the production process. This means that data based on this type is handled by a number of different systems such as the order system used by salespersons, but also in systems that are optimizing the finishing or giving instructions to the workers in the factory and so on. All of these systems are developed based on the current definition of the data type CarFeaturesT. In this example, there are only two options for the customer and therefore two members in the data type CarFeaturesT: "color" and "number of doors". After a while new features, such as navigation systems, become available for the car. How should CarFeaturesT be extended to include this new feature? Is it necessary that all participating systems are adapted to a new version of the CarFeaturesT data type or can the changes be limited to the systems that need to have this new information? Can both the old and new features be validated successfully by the new and old systems and therefore make sure that they got some valid data?

© Siemens AG 2007. All rights reserved. Permission granted to make copies for the purpose of VikingPLoP 2007.

Context

This pattern is useful if a system relies on a data model that needs to be extended, or may need to be extended in the future, and you want to limit the effort for adapting the system software in case of such a planned or even unforeseen change. It is expected that during the design of the data model either the location of these extensions can be anticipated or changes to the data model are expected at standard locations in general. Persistently stored data that is compliant to the previous version of the data model or standard data model remains unaffected and can be understood by the system as well.

The term "data model" is used as a description of data on a meta level in general. For example, a data model can relate to a relational database model, to an object model or structs that are used by programming languages such as C.

Problem

If you have a common data model or data standard with multiple software applications using it, you would like to have this model as stable as possible. Extensions normally lead to expensive changes of the participating software applications, even though the application may not be interested in the extended part. Therefore we need a way of extending the data model or standard without getting incompatible with the previous version of it.

Forces

To address these issues the following forces must be resolved:

- Requirements to the system and also to the data do change – in general because of new functionalities or because the system is extended.
- Changing of data models requires much effort, particularly when a data standard should be defined by different parties but also in systems with many stakeholders. This effort should be kept as low as possible.
- Depending on the frequency of standardization activities, there can be a significant time lapse until a modification of a data model is agreed by all participants. During this time, the data objects cannot be extended because a modified data object would not be valid for the data model that is still in use. This can lead to an innovation barrier. Temporary extensions of a data model can bridge the gap between subsequent versions of a data model.
- Changes to a data model may be temporary and need not to be incorporated into a widely used data model.
- In general, future modifications to a data model are not known in advance but should be possible when needed.
- A data model with insufficient description of its elements and their values might be misinterpreted. Therefore a data model should capture its elements as precisely as possible although it might be more difficult to change it afterwards. A balance between a strict data model that allows no additions in the data objects but can be validated and a very flexible but highly interpretable data model has to be found. "Validation" in this context means the possibility to

check the syntax and semantics of data against the data model to which the data should be conformant.

- Data that was made persistent and that conforms to the data model standard shall be valid as long as the data model is in use, even if the data model changes over time.

Solution

During data model design, identify the data types included in a data model that are likely to be extended in the future. Add empty placeholders, called extension points, to these data types that are used further on to cover any necessary extensions due to future requirements. If extensions to these data types are required then use the defined extension points for this purpose and make sure that all affected system components are adapted. The other system components can still work with the previous data model and ignore the extensions. These extensions can be introduced into a new version of a data model later on and adopted by the whole system if required.

Consequences

Benefits:

- Extensions to a data model are possible without adapting all system components that deal with the data. That means forward compatibility of system components related to the data is achieved.
- Standard data models can be adapted to actual needs without the necessity to change the standard. Innovations can be implemented also if the data model is affected. Later on, the extensions can be incorporated in the next version of the data model.
- The definition of a standard data model will be easier due to its inherent extensibility.
- All extensions can be understood and validated by the components for which they are intended, whilst ignored by the rest of the system.
- Extension points can be also pre-structured as it is discussed in the Implementation and Variants sections. A defined structure of the extensible part of the object prohibits uncontrolled definitions of extensions.
- Persistently stored data that conforms to the standard data model is still compliant to an extended data model as long as all system components regard the extensions as optional. That means that system components adapted to a newer version of the data model are able to deal with this older data. That is, backward compatibility related to the data is achieved.

Liabilities:

- Extensions to a data model must be anticipated or standard extensions must be planned for all relevant data model parts. Only then are extensions possible later on.

- Validation of the extension data is not possible for components adapted to a previous version of the data model, unless some specific rules are observed – see section Variants for details.
- The possibility for dynamic extension of data models might be misused to avoid new versions of data models. It should only be applied if data is to be extended and if the extensions are not used by all system components. Extension points should normally be a temporary solution until the extensions are integrated into a new version or a standard (e.g. while new components are tested). A stricter data model means in general better control of the semantics and syntax of the data and this is exactly what is neglected in case of extension points. There may be circumstances where it makes sense to use the extension permanently, for example when a format cannot be changed after its creation, e.g. the Internet Protocol header definition cannot be changed easily due to its huge impact.
- The concept requires that all system components that are dealing with the extended data are able to handle the additional data placed in the extension point object. Either the extended data has to be ignored but kept by the system components or it can be interpreted based on the new data model. All other exchanged data, however, must follow the original data model.
- If extension points are pre-structured generically (see Implementation and Variants section) then the data types used inside the data extension should be declared in the data model in order to be able to validate them.

Implementation

The first step for the implementation is to anticipate where the future extensions may happen in the data model. Pick these data types and add extension points to these data types.

An extension point is basically an empty container that is added to a data model during the design of a data model wherever extensions are likely to occur in the future. These extension points can carry data later on that were not considered at the design stage. They therefore encapsulate all extensions and make sure that the rest of the data model can be used as originally intended.

Figure 1 shows the diagram of a typical object type with some specific object content summarized by a container of type ObjectContentT and an extension point of type ExtraDataContainerT. The minimum demands on the additional data can be set with the definition of the type ExtraDataContainerT, as it is explained later on.

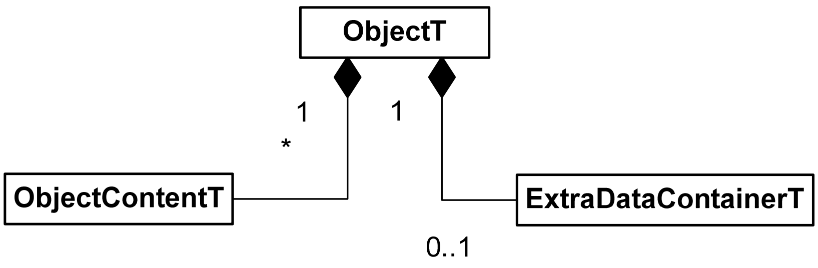


Figure 1: Object type with extension point

When the extension of the data model is made at the anticipated location, redefine the content of the empty extension point, i.e. of container type ExtraDataContainerT in Figure 1. In the following make sure that all affected system components are adapted. That is, design a new data model that redefines the content of the required extension points and adapt the affected system components so that they can access and work with the data in the extensions. This makes sure that the syntax and semantics of the extensions are properly defined for all affected system components.

Other system components, which are not interested in the extension, do not have to be adapted to the new model, but can still work with the previous data model. But they have to ignore completely and keep all data extensions they are not familiar with, that is, all extension points in the data objects.

Example: Consider again an automotive company. The following example shows how a data type CarFeaturesT that gives details of some car features could look like based on type ObjectT as defined above, see Figure 2.

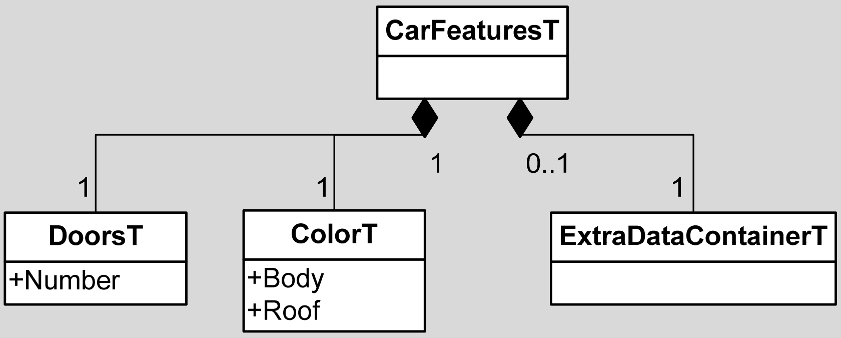


Figure 2: An example data model

The data type CarFeaturesT contains sub-elements for describing the number of doors and the color of the car but also an optional container for possible future extensions: an element of type ExtraDataContainerT. This element is intended for future use only and is not used in a typical runtime object at first as it is shown in Figure 3.

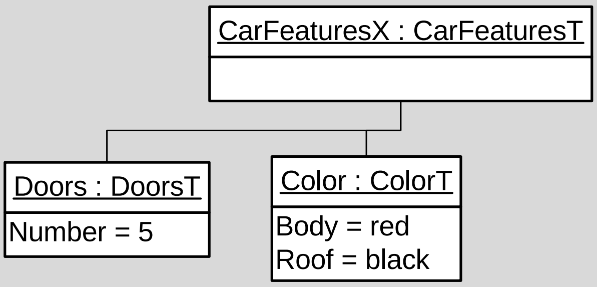


Figure 3: An example runtime data object

When new features, such as navigation systems, become available they can be added to the data using the extension point by directly redefining the data type ExtraDataContainerT. In Figure 4 the new container used for holding the navigation system data is displayed: NavigationSystemT extends ExtraDataContainerT with an additional element DisplayType which might e.g. specify whether a color or black-and-white display is chosen. It is not absolutely necessary that the data modeling language supports inheritance as Figure 4 might suggest. This feature facilitates the implementation of this pattern and the validation of the data using extension points but it can be substituted by other concepts as well.

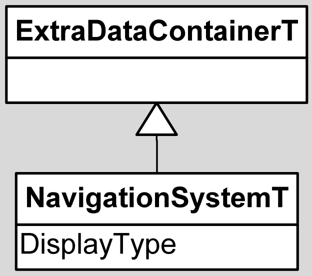


Figure 4: The new container NavigationSystemT

The resulting data object of CarFeaturesT given in Figure 5 can be processed by both kinds of systems: The participating systems that need to interpret the data describing the navigation system have to be adapted to the new data model. All other systems can just ignore the additional data and rely on the previous data model.

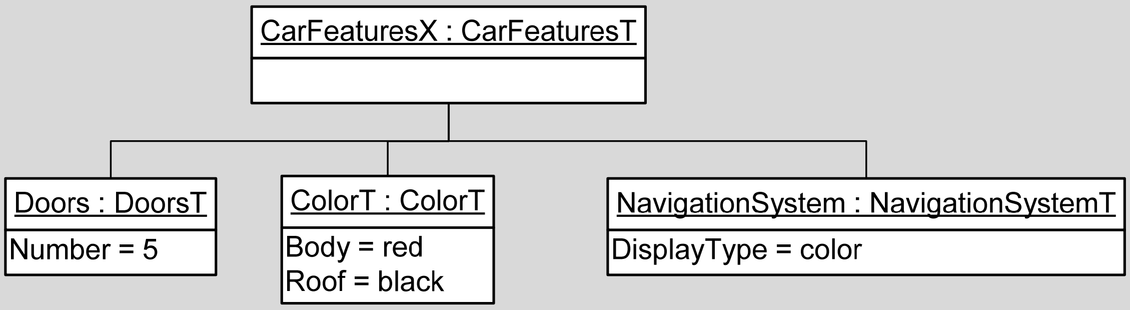


Figure 5: An extended data object

In our example of the automotive company, only the production station where the navigation system is incorporated into the car is affected by the data model change while all other production stations can ignore the extension since the additional information about the navigation system is not relevant to them.

Variants

The first two variants given in this section differ in the level of sub-structuring of the extension points while the last two variants deal with different approaches for embedding extension points in data models, see Figure 6. These variants are all enhancements of the basic pattern.

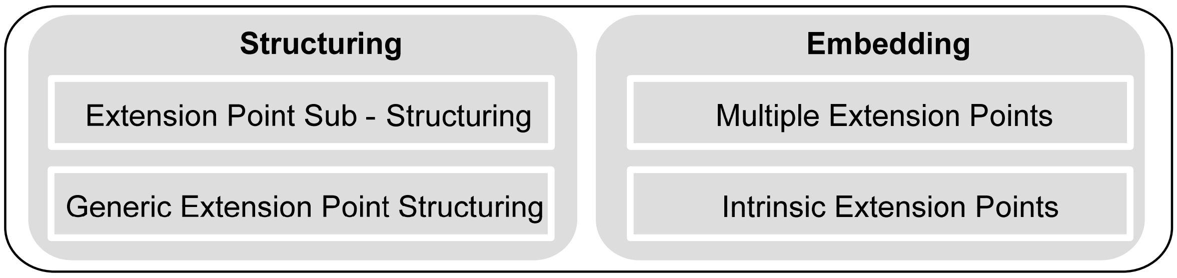

Structuring
Extension Point Sub - Structuring
Generic Extension Point Structuring
Embedding
Multiple Extension Points
Intrinsic Extension Points

Figure 6: The variants of the basic pattern

Extension Point Sub-Structuring

Extension points can be pre-structured to enable easy multiple extensions of the data model and also provide a basic guideline for extensions. As it is shown in Figure 7, the ExtraDataContainerT type that was introduced in Figure 1 as extension point may contain an arbitrary number of containers of type ContainerT. These containers can be built up in a hierarchy to reflect the structure of the data being modeled, again shown in Figure 7. The container of type ExtraDataContainerT, though, is available only once.

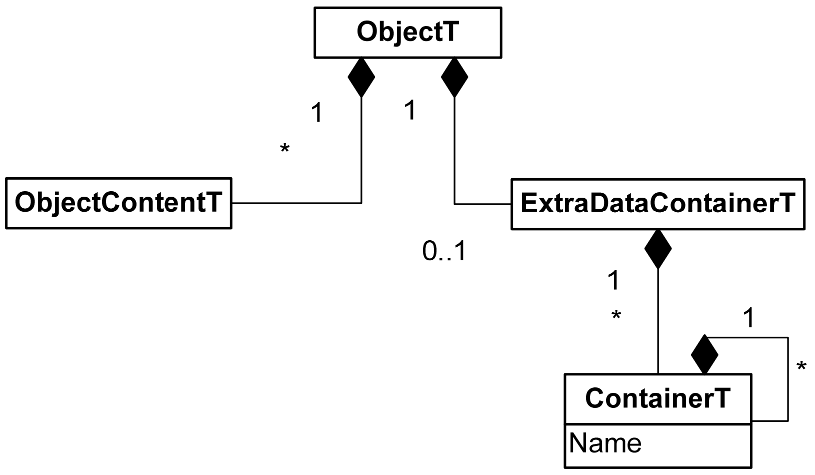


Figure 7: A sub-structured ExtraDataContainerT type

Basically, the sub-containers of ExtraDataContainerT are used for extensions. That is, containers that conform to the definition of ContainerT can be added to an element of type ExtraDataContainerT in this data model. These containers are used to substitute objects of type ContainerT – they do not need to be real sub-classes in the sense of object orientation but it is just necessary that they are acknowledged as compatible replacements. As it is shown in Figure 8, specializations of ContainerT, e.g. ExtendedContainer1T, are used for adding the additional data.

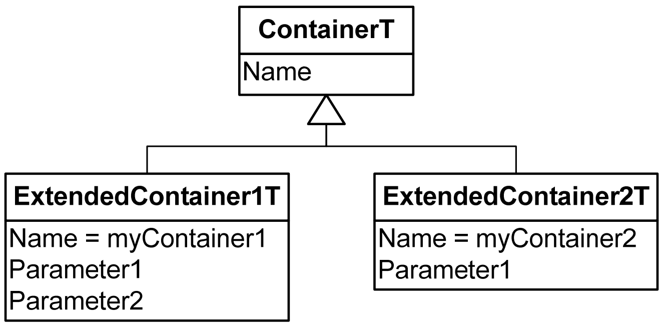


Figure 8: Specialized containers for extensions

Since there might be several sub-containers added by different system components, all ContainerT elements must have a name attribute to distinguish them, i.e. the name must be unique inside the ExtraDataContainerT element. Rules should be established which define how to generate a name for a container or a parameter object since it must be possible to identify each object uniquely.

The pre-structuring of the extension point enables a better structuring of data inside the extension point and also makes it possible that the data model is extended independently by different system participants. In addition, system components that are not affected by the extensions and therefore not adapted can still identify the extension point containers and ignore only their content. Limited adaptation is also possible with this variant: A system component can use selected extension point containers but ignore other containers it has no knowledge about. A drawback of this variant is, that it is necessary to synchronize the extensions regarding the container names that are assigned to specific extensions. In addition, the extensions can be validated only by system components adapted to the extensions.

Generic Extension Point Structuring

The basic pattern and the previous variant require always the redefinition of the extension point (or a part of the extension point) in a future data model. Instead of changing the data model it is also possible to employ a generic version of the extension point in the data model instead: This implies again a more detailed pre-structuring of the extension point. In Figure 9 we see that a ContainerT element contains either other ContainerT or ParameterT elements. Such a parameter can be named and has a value which is the real data to be stored. The parameter names need to be unique inside the ContainerT element in order to make them distinguishable.

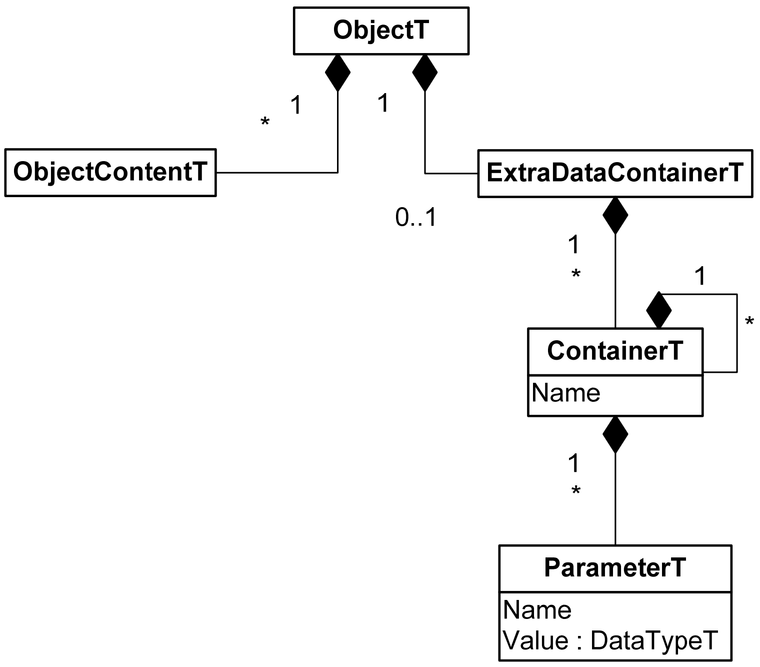


Figure 9: Object with an extension point structured as generic container hierarchy

With this concept, it is even possible to add additional data to an ExtraContainerT object without the need to define new data types at all: It is sufficient to add a container of the base type ContainerT and fill it with arbitrary parameters and their values. That is, the element of type ContainerT and its elements of type ParameterT or ContainerT are used directly without any changes.

For validation purposes it is recommended to define different parameter data types in advance, i.e. when the initial data model is created. These parameter types should be sub-types of the generic DataTypeT as used by the type ParameterT and could be, for example, standard data types like strings or integers. Then, ParameterT objects can contain a name-string value pair or a name-number value pair. Using this concept, objects of type ExtraDataContainerT can be validated and understood completely. Only the interpretation of the name attributes in the ContainerT and ParameterT objects as well as the content of the Value element itself are not defined.

With the recursive ContainerT concept it is possible to define large tree hierarchies. If the data model is small, the recursive definition of the ContainerT type can be removed. This reduction will result in a model which can be understood easier. But be careful with a large list of parameters. Related parameters should be better grouped in one container to express this relationship properly and to enhance understanding of the model. Therefore the recursive definition of ContainerT is very helpful.

In contrast to the previously described variant, the data model is not changed in case of extensions but the built-in generic mechanism is used directly for extensions. Therefore it is always possible to validate the data according to this data model and even process the extension point data without any knowledge of new data. On the other hand this is also a drawback since the semantics of the extension point data is not defined at all and might be misinterpreted easily.

Multiple Extension Points

In the basic pattern and the variants described so far we have used a single extension point for each object type. Variants are possible, e.g. using multiple extension points in object types.

Multiple extension points are useful if an object is very large and if extensions are made only to specific parts. In this case at each of the possible extension locations an extension point can be set up – so that the extensions can be made at the logically best positions in the object type. The number of extension points should be limited, though, otherwise the resulting data model is difficult to comprehend.

Intrinsic Extension Points

The extension point concept can also be integrated in a base data model. All domain specific objects shall be derived from this base data model and therefore contain the ability to be extended without changing the data model. Figure 10 shows an object type ObjectT that could be used as the basis for all object types in a data model and would be a typical element of such a base data model.

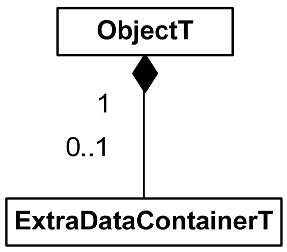


Figure 10: A basic object type ObjectT with an integrated extension point

ObjectT already contains an optional element of type ExtraDataContainerT and therefore all derived object types would also support such an extension point as a standard feature. Again, inheritance is no prerequisite but all data types of the data model can support such an extension point independently.

This variant is useful whenever it is not possible to foresee where extensions will occur in the future and the effort for changing the data model is very high. Since these widespread extension points allow uncontrolled extensions, this freedom is also a drawback of this variant.

See Also

Related patterns are the handle/body patterns as summarized by Coplien (1998), e.g. the decorator pattern from Gamma, Helm, Johnson and Vlissides (1995). Like this pattern, the decorator pattern makes it possible to extend an object. But the decorator pattern does this by wrapping dynamic objects while this pattern identifies extension points and prepares an extension for data objects.

Known Uses

- XML@PROFIBUS standard for defining engineering data formats: This pattern is used inside this standard as a basis for all complex data types including the sub-structuring of the extension point.
- TCP (Transmission Control Protocol) Header: This data structure defines an element “Options” which can be used for various data to be interchanged between communication partners. It therefore is a simple implementation of the extension point concept.
- In Siemens internal data formats for the automation and building technology sector this concept is also applied. All complex data types offer an extension point which can be used for future extensions. The data formats cover also the sub-structuring of the extension points.

Acknowledgment

We would like to thank Michael Kircher for his early and very valuable review of this pattern and Kristian Elof Sørensen for his great coaching during the shepherding for the VikingPLoP 2007 conference. Many thanks also for all the valuable contributions made during the conference.

Literature

B. Böhm, N. Gewald, G. Herold, D. Wißmann: *Patterns in Data Modeling and the Dynamic Extension Pattern as Example*, in: Proceedings of the IADIS International Conference e-society 2006, Dublin, Ireland

F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, M. Stal: *Pattern Oriented Software Architecture Volume 1: A System of Patterns*, Wiley, 1996

J. Coplien: *C++ Idioms*, in: J. Coldewey, P. Dyson (eds.): Proceedings of the 3rd European Conference on Pattern Languages of Programming and Computing (EuroPLoP ’98), Universitätsverlag Konstanz, 1998

E. Gamma, R. Helm, R Johnson and J Vlissides: *Design Patterns: Elements of Reusable Design*, Addison-Wesley, 1995

M. Fowler: *Analysis Patterns: Reusable Object Models*, Addison-Wesley, 1997

M. Fowler et al: *Patterns of Enterprise Application Architecture*, Addison-Wesley, 2003

M. Kircher, Prashant Jain: *Pattern Oriented Software Architecture Volume 3: Patterns for Resource Management*, Wiley, 2004

D. Schmidt, M. Stal, H. Rohnert, F Buschmann: *Pattern Oriented Software Architecture Volume 2: Patterns for Concurrent and Networked Objects*, Wiley, 2000

Refactoring State Machines

Matthias Rieger, Bart Van Rompaey, Serge Demeyer
Lab On Re-Engineering
University of Antwerp, Belgium
{matthias.rieger,bart.vanrompaey2,serge.demeyer}@ua.ac.be

Abstract

State machines are a common way of representing a variety of problems in computer science. For certain implementations of state machines, e.g. `switch`–statements, unattended growth can lead to overly complex and unmaintainable code. This paper presents a small, code–level pattern language which assists a maintainer in the different tasks of identifying, recovering, and refactoring of state machine implementations with the goal of applying some of the well known idioms and patterns that have been developed for the procedural and the object–oriented paradigm.

1 Introduction

Code that grows over the course of multiple evolutionary steps tends to get more complex than what can be handled by someone not familiar with the code. When the increasing cost of adding another feature and the rising risk of introducing errors are too large to be carried, a dedicated effort to refactor first becomes necessary.

In such a legacy system one is typically confronted with code fragments that are too complex to directly integrate some required changes. To reduce the complexity via refactoring, you first have to understand the nature of the code. When identified as a state machine implementation, i.e. a set of states with transitions triggered by external input, specific solutions exist to make such fragment maintainable again. Martin argues that finite state automata are among the most useful abstractions in the software arsenal, as they (i) provide a simple way to define the behavior of a complex system; (ii) provide a powerful implementation strategy; and (iii) are almost universally applicable [6]. As such, approaching the change as an extension of a state machine may ease the maintenance process.

Figure 1 contains an overview of a small pattern language for refactoring state machines that we present in this paper. Having some clues that the given code fragment encompasses some state machine implementation triggers the entrance point to the language, helping the maintainer to (i) recognize the state machines; (ii) extract its essence in terms of states and transitions; and (iii) subsequently transform them into a modular, well structured implementation under a variety of implementation choices.

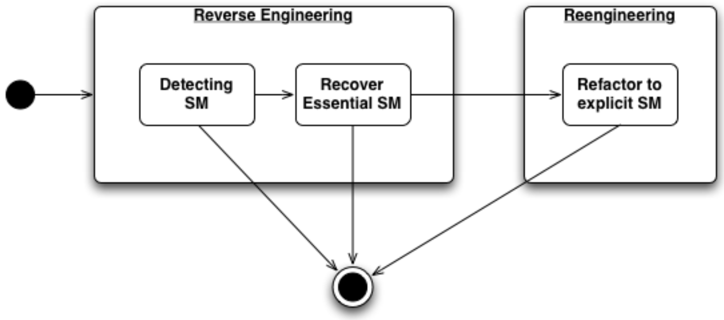


Figure 1: Pattern Language for Refactoring State Machines (SM).

The first two patterns of the pattern language constitute the *reverse engineering* part and are described in Section 2.

A *state machine* is a well known representation of a specific class of problems. There exist multiple idioms and patterns which describe how to implement a state machine. Knowing that a given code fragment implements a state machine therefore gives us a clear goal towards which we can guide our refactoring effort. Therefore, in the first pattern, named DETECTING AN IMPLICIT STATE MACHINE, we need to gather evidence that helps to decide whether the given code fragment implements a state machine. A positive answer gives rise to applying the follow-up patterns.

The next pattern, RECOVER THE ESSENTIAL STATE MACHINE seeks to guide the developer in identifying the essential composition of the state machine in terms of states, transitions and action code. Having recovered this state machine model out of the source code, two more questions pop up. First, the developer needs to decide upon a new go/no go decision about an eventual refactoring. Secondly, the most favorable implementation for the case needs to be chosen.

Once we have extracted the structure of a state machine in terms of states and transitions, we can go over to the next step of rebuilding it using one of the well known idioms and patterns for state machine implementations. The intent is in all cases the same: reimplement an existing state machine so that its states and transitions become *explicit*. The prefered implementation technique depends largely upon the contextual programming paradigm, the desired level of abstractions and the representation and amount of states. In Section 3 we describe a refactoring pattern, REFACTOR TO EXPLICIT STATE MACHINE IMPLEMENTATION, to decide upon and refactor towards one of two typical state machine implementations. The procedural state machine implementation uses a master function that dispatches control to state-representing functions. The object-oriented approach describes the refactoring of a deteriorated state machine implementation towards the object–oriented state pattern by Gamma et al [4].

The pattern language in this paper is written for software developers working on evolving legacy systems, having to decide between quick hacks and from scratch implementations when coping with new requirements. The patterns here help to explore and carry out a path in the middle: refactoring the existing implementation first, facilitating future additions. In that sense, this work is similar to the reengineering patterns of [2] and the refactoring guidelines of [5]. Furthermore, a small appendix contains some basic information about state machine representations (Appendix A).

2 Reverse Engineering State Machines

Suppose a complex piece of code, containing lots of branches with complex conditions that you are required to understand and which you suspect of implementing a state machine. In order to perform a successful refactoring towards a clean implementation, we first need to decide if the code fragment in question really concerns a state machine. Secondly, we have to reverse engineer the state machine and extract all states, transitions, and associate the action code within the state machine correctly with the given states and transitions.

Forces. The following forces play a role in determining the difficulty, required effort, scope and support required to proceed with reverse engineering the code fragment:
Reverse Engineering Goal. Will the concerned code be extended with new requirements? Are you tracking a bug, possibly involving test writing? Maybe you plan a rewrite or port of the code. In all cases you need to understand what the code is doing, although the level and scope of the desired understanding differs.
Familiarity with the code. Which artifacts are available except for the source code? The documentation may mention the presence of a state machine or even presents a (possibly outdated) specification. Are the original developers still around, or, if not, how experienced are the current maintainers with the code? Moreover, what is your own knowledge of state machines?
Impact and involvement. What is the role of this code in the system? What is the current change rate? Checking the involvement of other developers (possibly via the versioning system) as well as how the system relies on this piece through its interface gives a feeling about how difficult and important it is to reverse engineer.
Quality of the code. Several lightweight metrics give a first impression of the code's quality. The readability can be verified via size metrics and identifier names. The number of bugs in recent months tell something about the reliability. Determine changeability via versioning system queries as well as by asking the developers (how many of your colleagues avoid this piece of code at all cost?). The outcome helps to estimate how much effort is required.

Pattern: Detecting an Implicit State Machine

Intent: Determine whether a complex code fragment implements a State Machine using a code review checklist.

Problem

How can you identify a State Machine implementation in source code?
This problem is difficult because:

- There exist many guidelines and patterns about how to implement a State Machine [7, p. 202]. Some implementations are strongly localized in a lengthy construct, others rely more on abstractions and dispatching. Clearly, each of such implementations bears its own characteristics, adding to an eventual checklist. Moreover, a State Machine can just as well be implemented in an arbitrary combination of conditions and abstractions.

Yet, solving this problem is feasible because:

- Developers typically use naming conventions to denote State Machine parts such as states or transitions. As a reviewer, you can make use of such indicators.
- Given system documentation, you may have encountered state diagrams or requirements that are likely to translate into a State Machine.

In order to perform a successful refactoring to a clean state machine, we first need to decide if the code fragment in question really implements a state machine. Secondly, we have to reverse engineer the state machine and extract all states and transitions, and associate the action code within the state machine correctly with the given states and transitions.

Solution

Detecting the Structure of a State Machine

A state machine is a closed program section that repeats itself, getting input from outside, and selecting on possible alternatives among a set, until it has reached an end-state, upon which the state machine is exited.
The basic structure of a state machine is therefore (in pseudo code):

```
state = initialState;
while(moreInput() && state != endState) {
    <select action code based upon state and input>
    <execute action code with input>
    <transition to new state>
}
```

Any code fragment that conforms to this schema can be viewed as a state machine. The selection of the action code happens via a `switch`–statement or an `else–if` chain. In many case the state is represented by a numerical type. Therefore, a `switch` condition is able to dispatch control flow to the appropriate state code. The following code exhibits a canonical example for such an implementation:

```
switch($state) {
 case 1:
   $return .= $utf7[ord($char) >> 2];
   $residue = (ord($char) & 0x03) << 4;
   $state = 2;
   break;
 case 2:
   $return .= $utf7[$residue | (ord($char) >> 4)];
   $residue = (ord($char) & 0x0F) << 2;
   $state = 3;
   break;
 case 3:
   $return .= $utf7[$residue | (ord($char) >> 6)];
   $return .= $utf7[ord($char) & 0x3F];
   $state = 1;
   break;
}
```

When the state is encoded in a string, however, an state machine implementation must use an `else–if` chain, as can be seen in the following example:

```
if(pos == root_struct) {
   status = "body";
   root_struct_ns = msg["namespace"];
} elseif(name == "Body") {
   status = "envelope";
} elseif(name == "Header") {
   status = "envelope";
} elseif(name == "Envelope") {
   // nothing to do
}
```

It is also possible that states are encoded in the values, or ranges of values, of multiple variables. The following example shows how three states are defined as ranges of the variable `count`:

```
if (count == 0)
    ...           // action for empty collection
else if (0 < count && count < maxEntries)
    ...           // action for normal usage
else if (maxEntries <= count)
    ...          // action for exhausted capacity
```

Finally, the purpose of the code can be an indication that it is implementing a state machine. State machines are preferred solutions for a number of common tasks such as, for example, (stateful) protocols, lexical analyzers, (GUI) event handlers, etc. Oftentimes programmers note that the present code implements a state machine in the accompanying comments.

What's Next?

Applying this pattern has given some evidence whether the complex pieces of code is a state machine implementation. In case of a positive answer, you can proceed to the next pattern, RECOVER THE ESSENTIAL STATE MACHINE , to recognize the state machine components in the code and to reverse engineer the implementation to a state diagram. If there is not enough evidence, the state machine path should be left. Other patterns dealing with complexity may apply, such as EXTRACT METHOD [3], REFACTOR TO UNDERSTAND or TRANSFORM CONDITIONALS INTO REGISTRATION [2].

Pattern: Recover the Essential State Machine

Intent: Recover the composition of a State Machine from the implementation.

Problem

How can you extract states, transitions and actions from a State Machine implementation?
This problem is difficult because:

- A State Machine may not have been implemented consistent with a particular implementation pattern. States or transitions may have been omitted.

Yet, solving this problem is feasible because:

- You can exploit the check list heuristics that made you decide that we are dealing with a State Machine implemenation.

In order to perform a successful refactoring to a clean state machine, we need first to decide if the code fragment in question really implements a state machine. Secondly, we have to reverse engineer the state machine and extract all states, transitions, and associate the action code within the state machine correctly with the given states and transitions.

Solution

Once we know that the code fragment in questions implements a state machine, we can extract its individual constituents, e.g. its *states*, *transitions*, and the actions which are tied to them. With the information about states, transitions and actions you can draw a representation of the state machine, e.g. a state diagram (see Appendix A.1). This serves as the basis for refactoring the state machine.
To be able to identify the states and transitions of the state machine better, consider a first round of refactorings[1]. We have, for example, observed that separate `cases` in a complex `switch`–statement often contain duplicated pieces of code. Be aware, however, that moving code around before you have identified the states and transitions may make reverse engineering more difficult if the code contains statements pertaining to these concepts.

[1]The *Refactor to Understand*-Pattern (see `http://st-www.cs.uiuc.edu/users/brant/Refactory/SS99/sld056.htm`) gives hints on what to do.

Extracting the State Variables

A state machine has a finite number of states, which are represented by either one, or a group of variables. The *state variable* (`sv`) holds the input values that trigger the transitions. Three hints can lead you here:

Name: Names ending in `status`, or `state`, are good indicators for the `sv`. Be aware, however, if the `input` variable comes from another state machine, for example from the settings of a radio button GUI element, it might still be called `state`. The name `status` is also often used for the resulting value of an invocation.

Usage: The `sv` will be assigned to whenever a state transition occurs. This normally happens towards the end of a case. The `input` variable, on the other hand, will only be read from, since the assignment of new input values will happen outside of the state machine.

Definition: The state variable must be defined globally, whereas the input variable is a most likely a parameter to the function containing the `switch`–statement.

After having identified the state and the input variables, list all potential values that they can have (find all assignments to `sv` or, if `sv` is of an enumeration type, find its definition). The `switch`–statement should provide one case for each of the states and/or each of the input values explicitly named.

Single or multiple state variables: In the simplest and most common case, the current state is represented by the value of a single variable. But there are cases where the state is represented by a set of variables.
For example, the state here is defined by a set of flags, each one being true in the exclusion of the others.

```
bool inTeX      = false ;
bool inComment  = false ;
bool inString   = false ;
bool inClause   = false ;
```

There might also be flags which encode state in additional to the main state variable, e.g.

```
    bool isTransitionToFirstActivation;
    bool isCreateAndGo;
```

These flags may indicate an additional state that is not represented by a potential value of `sv`. If you find secondary state variables consider extending the range of the primary state variable (adding new primary states) and getting rid of the secondary ones, thus making the state machine more explicit.

Extracting the Transitions

A state machine, in the course of its execution, transitions between its states. For each state, determine the transitions that leave it. A transition is characterized by an assignment to (one of) the state variable(s) `sv`.
The value of `sv` at this position indicates the new state of the state machine. If `sv` was not assigned to before the transition is concluded, a self-transition has occurred, leaving the state machine in the same state.
A `return` from the function containing the `switch`–statement or `else–if` chain may indicate a transition to the end-state of the state machine. If there are such direct returns from within the state machine, consider creating a separate 'STOP' case and transition to this state instead of returning directly. At the cost of an extra iteration through the machine you will have a more explicit model, and you will be able to more easily handle refactorings like extracting action code into separate functions. The action code of the 'STOP' case will also be the right place for cleanup actions that must be performed when the state machine exits.

Extracting Action and Transition Code

A state machine usually attaches some actions to either the states, the transitions, or both.
For each state determine precisely which code belongs to it. This code is the action code and it will also contain the transition code. The following example presents the difference between state code, which is executed each time the state is entered, and transition code, which is only executed for a specific transition.

```
case INIT:
  lnp = doread_line(&curproc);    // state action
  if (INSTR(lnp) == ps_pro) {
    state = AFTERPRO;             // transition 1
  } else {
    state = NORMAL;               // transition 2

    headl = lnp;                  // transitional action
    lp = &lnp->l_next;
  }
  break;
```

Some `switch`–statements use the *fall-through* mechanism, e.g.

```
switch(state) {
    case stateA:
    case stateB:  actionCode1();
                  if (status == stateA) { break }
    case stateC:  actionCode2();          break;
}
```

A fall-through, especially a conditional fall-through, is usually harder to understand than if all cases conclude with a `break`. The solution below is therefore preferable:

```
switch(state) {
    case stateA:  actionCode1();                break;
    case stateB:  actionCode1(); actionCode2(); break;
    case stateC:                 actionCode2(); break;
}
```

This solution also shows the advantage of encapsulating the action code into a function: it can be invoked from multiple states.

What's Next?

At this point, you should end up with a model of the state machine (such as a state diagram). This model specifies the state machine as it was implemented. Compared against given specifications, it tells you (i) how it compares to the expected behaviour; and (ii) how it compares to the required extensions.
In case the specification is close enough to the recovered model, or it is obvious enough how to proceed to closing the gap, you can start to REFACTOR TO EXPLICIT STATE MACHINE IMPLEMENTATION.

Some reasons why you would not want to continue with this implementation include:

- Lacking confidence in the model, because you are not convinced that the obtained model is correct and complete. The implementation may have been too complex to accurately extract all states, transitions and actions.
- The model may be too far away from the desired implementation; the required changes may have too much impact on the model.

As such, a reimplementation from scratch may be the preferrable solution. Binder provides an overview of state machine implementation alternatives [7].

3 Refactor a State Machine

Forces. Multiple design guidelines exist for implementing a state machine. We identified the following forces to impact the decision for a particular solution:
Future changes. How will the state machine change in the foreseable future? How will that impact the states, transitions or actions of the state machine. Solutions differ in the effort that is required to make such changes and extent to which they scale.
Time of changes. When are changes to be made, and how much time will be foreseen for the update? If high availability of the system is important, dynamically updating the state machine may be a requirement. A solution where states and transitions are easy to update and extend at runtime, such as a table-driven implementation, then becomes more favorable.
Experience. The experience of the developers in either procedural or object–oriented techniques influences the decision in the sense that a learning curve has to be taken into account if the developers are not accustomed to the chosen solution.
From the set of solutions, we present two patterns discussing refactoring a state machine implementation to the common solutions of using the object–oriented state pattern and a procedural, function-based approach.

Pattern: Refactor to Explicit State Machine Implementation

Intent: Refactor a complex state machine to a state machine implementation with explicit representations of state, action and transition.

Problem

Many designs for state machines have been promoted in literature. The object–oriented *state pattern* proposes the introduction of a subclass hierarchy representing the various states. In a procedural design, a set of functions captures the various states. What target design suits best given an implicit state machine implementation?

Solution

In the *procedural implementation* of a state machine, states are modeled by functions instead of classes. For each state we have one function which associates the actions with the given inputs. We also need an initial dispatch function, which, based on a variable that holds the current state, dispatches the control flow to the function that represents this state. The following example shows a procedural implementation with two states:

```
void initialize() {
  currentState = idle;
}

void dispatch(int inputEvent) {
   // dispatch control flow to current state
   switch(currentState) {
     case idle:  idleState(inputEvent); break;
     case busy:  busyState(inputEvent); break;
   };
}

// handle 'idle' state
void idleState(int inputEvent) {
   if(inputEvent == START) currentState = busy;
}

// handle 'busy' state
void busyState(int inputEvent) {
  if(inputEvent == STOP) currentState = idle;
}
```

The object–oriented *state pattern* is a well known design pattern by Gamma et al. [4]. The state pattern (see Figure 2) represents a state machine by the so–called *context* object which acts as the interface to the clients of the state machine. The states of the machine are all implemented in separate classes, each one inheriting from an abstract STATE class. The CONTEXT class holds a pointer to the current state and delegates handling of the input event to the state. When a transition occurs, an instance of the new state is created and replaces the old state in the context class reference.

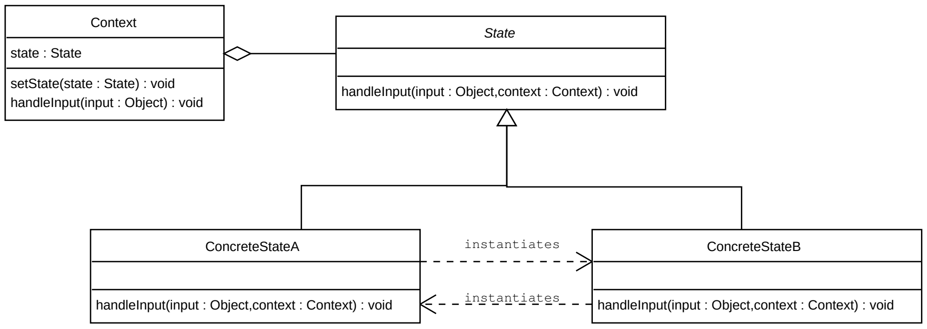

Context
state : State
setState(state : State) : void
handleInput(input : Object) : void
State
handleInput(input : Object,context : Context) : void
ConcreteStateA
handleInput(input : Object,context : Context) : void
instantiates
instantiates
ConcreteStateB
handleInput(input : Object,context : Context) : void

Figure 2: State Design Pattern.

Trade-offs

The following tradeoffs play a role in preferring a particular alternative solution (see also the *Consequences* section of the state pattern in [4]):

- **Programming Paradigm.** Mechanisms like polymorphism are only available in an object–oriented language. Even with a suitable programming language, the successful application of object–oriented design principles depends on the familiarity of the developers with the paradigm. The procedural solution can be applied in any programming language, i.e. it does not rely on object–oriented features.

- **State machine Isolation.** Depending on the size of the state machine, an implementation using multiple classes can introduce too much boilerplate code, as the number of states corresponds to an equally large number of classes to be created. If there is only a small number of input events that are recognized in a certain state, the overhead of creating a new class for each state makes this solution inefficient. Additional classes also tend to pollute the namespace and design diagrams. As such, in the case of a state machine of limited size, a single switch–statement may be the right level of abstraction to make the code understandable. The procedural way provides a more compact solution. We are assuming here that the state machine is only a small part of the system and don't want to put too much emphasis on it, which a design with multiple classes—as required by the state design pattern—would do. The entire state machine can be implemented in a single source file which reduces clutter in the source tree. As an example for the understandability of smaller switch–statements, see the following code which implements a comment remover for C/C++ programs.

```
switch(state) {
  case INCODE:

   switch (c) {
    case '/':  c=peek();
               if       (c=='/') { nextline();        }
               else if  (c=='*') { append(' ');
                                   advance();
                                   state=INCOMMENT; }
               else              { copy_advance();   } break;
    case '"':  copy_advance();     state=INSTRING;     break;
    default:   copy_advance();
   }
   break;

  case INSTRING:

   switch (c) {
    case '\\':  copy_advance();    copy_advance();     break;
    case '"':   copy_advance();    state=INCODE;       break;
    default:    copy_advance();
   }
   break;

  case INCOMMENT:

   switch (c) {
    case '*':   if (peek()=='/') { advance();
                                   state=INCODE; }
                advance();                            break;
    default:    advance();
  }
}
```

This state machine consists of three states and four transitions. Its implementation counts about 25 lines of code (disregarding the helper functions). Since we can keep the entire state machine on as single page, the procedural implementation is preferable over the OO solution.

- **Facilitating Evolution.** In the OO solution, a `switch`–statement growing with the number of state is not needed since dispatch is done via polymorphism. If there are many states, getting rid of an unwieldily `switch`–statement improves the understandability. Moreover, polymorphism removes one level of `switch`–statement, that is needed in the procedural implementation, makes the code more maintainable in light of potential changes in the number of states.

  The number of states determines the size of one `switch`–statement of the procedural solution. Moreover, the procedural implementation requires at least two `switch`–statements are needed to dispatch the control flow to the correct location. We are however free to change the order of the two dispatches needed. If there are many more inputs than states we just dispatch on the input first, and then on the state. This results in more but smaller methods. Since we must write at least one dispatch based on the value of the state variable `sv`, having a large number of states will make procedural code difficult to understand.

- **State Representation.** In the OO implementation, states are made explicit by means of a dedicated class: a CONCRETESTATE class contains only code pertaining to a single state. This focusses the attention of the person reading the code and makes the code more understandable. The procedural approach using a dispatch function which does not contain any action code has the advantage that it is self-documenting in that it gives a good overview over all the existing states.

- **Multi-variable State.** If the state must be represented by the values of multiple variables, the danger of state changes that leave an inconsistent state arises in the OO approach, as changes need to be made in two different classes. Encapsulating all state variables in an object hides these variables to the outside and makes state changes atomic again. Since from the the viewpoint of the context object, a state transition is only a single assignment, state transitions are then always atomic.

Refactoring Steps

Once a decision has been made for either the object–oriented or the procedural implementation, the refactoring steps are similar for both cases.

1. Define a blank canvas for the new implementation. This can be a fresh function or a new class. The code will be moved gradually from the old to the new location. At the end, the old location is an empty shell which delegates to the new place.

2. Encapsulate all action code in separate methods. This makes the state-handling function more transparent, showing all different inputs that are handled at a glance. If possible, try to separate the action code from the transition code which again improves understandability.

3. Use naming conventions for the methods and variables. For example:

   - The method which dispatches on the current state has a name such as `dispatch()`, since it is the entry point into the state machine.
   - In the procedural case, every method representing a state should end in `State`.
   - The variable storing the state value should end in `State` or `Status` as well.
   - The name of the variable containing the input value that determines the transition to the next state should contain the string `command` or `action`.

   Naming conventions let you separate *dispatch*–methods from methods implementing the actions of the state machine. Separating these entitites lets the maintainer selectively improve his understanding of what the system does. Moreover, it will isolate the impact of future changes (e.g. introduction of additional action code).

4. Test the state machine as before the refactoring. Since the behavior of the state machine is not changed, all existing tests should work without changing.

Related Patterns

- Factor Out State [2].
- *Embedded State Machine Implementation*[2].
- *Replace State-Altering Conditionals with State* [5].

Known Uses

- Bucknall discusses how to refactor a parser to recognize a floating-point number in a string. Originally implemented as a large switch statement representing a state machine, he proposes to refactor to an object model, for ease of implementing (a closer match with the actual state machine), understanding and maintenance [1].

A State Machines and their Representations

A finite state machine (FSM) or finite automaton is a model of behavior composed of states, transitions and actions. A state stores information about the past, i.e. it reflects the input changes from the system start to the present moment. A transition indicates a state change and is described by a condition that needs to be fulfilled to enable the transition. An action is a description of an activity that is to be performed at a given moment. A transition is triggered by an input action or event. A sophisticated state machine may execute an action when a specific state is entered, and another action when a state is left. A simple state machine associates an action with each transition between a source and a target state.
To determine which action code to execute next, a state machine works on two parameters, the current state and the input. The normal view of a state machine as a certain state acting on the input leads to the idiom of first selecting the state and then take the input event to decide on the action to be executed.

A.1 State Machine Representations

State machines have two common representations: the *state transition table* and the *state diagram*.

[2]`http://www.embedded.com/2000/0012/0012feat1.htm`

State Transition Table

A state transition table is a way to present the transitions of a state machine in a clearly arranged manner. A commonly used layout shows the states on the vertical axis, and the input events on the horizontal axis. Each cell contains the state into which the machine transitions, given the state on the horizontal and the input on the vertical axis (an empty cell means that a state does not handle a given input).

Events **States**	**1**	**0**
S$_1$	S_1	S_2
S$_2$	S_2	S_1

Table 1: State Transition Table. State S_1 transitions to S_2 if input **0** is recieved.

A table driven approach of designing a state machine does a good job in giving an oversight of all possible states, inputs, and transitions. It's however difficult to include into the display the actions that have to be taken for each transition.

State Diagram

A graphic way to represent state machines is a diagram made out of bubbles and arrows. Each circle represents a state, and arrows represent transitions from state to state. The transitions are annotated with the input by which they are triggered.

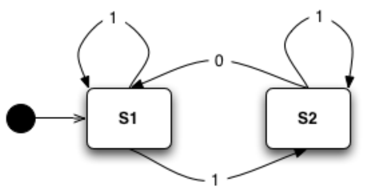


Figure 3: State Machine Diagram for the state machine of Table 1. The initial state is S_1 as indicated by the double circle.

State diagrams are a good way to design a state machine, or to reverse engineer a state machine from the source code, because they can be drawn freely on a piece of paper. A new transition that is discovered during the reverse engineering phase can be added more easily to a free–form drawing that to the rigid structure of a table.

Acknowledgment

This work was executed in the context of the ITEA project if04032 entitled *Software Evolution, Refactoring, Improvement of Operational&Usable Systems* (SERIOUS) and has been sponsored by IWT, Flanders.

References

[1] J. Bucknall. Object-oriented state machines. *The Delphi Magazine*, (115), March 2005.

[2] S. Demeyer, S. Ducasse, and O. Nierstrasz. *Object-Oriented Reengineering Patterns*. Morgan Kaufmann, 2002.

[3] M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 1999.

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

[5] J. Kerievsky. *Refactoring to Patterns*. Addison Wesley, Reading, Mass., 2005.

[6] R. Martin. *Agile Software Development, Principles, Patterns, and Practices*. Prentice Hall, October 2002.

[7] R.Binder. *Testing Object-Oriented Systems*, chapter 7: State Machines. Addison-Wesley, 2000.

Patterns in the Story-based Software Development Method

Jan Reher, Systematic Software Engineering A/S*

December 2007

1 Introduction

The Story-based Software Development Method aims to make the everyday life of software developers simpler and more structured so that they can concentrate on the task of delivering useful, working software.

The method's key characteristic is that implementation of a feature in a system occurs incrementally in a number of small work packages that share a common structure. These work packages are called stories. This makes for a method that is disciplined, manageable, mature, agile, productive, and programmer-friendly.

Planning the implementation of a feature consists mainly of breaking the feature into stories according to three strict rules: 1) each story should be of manageable size, 2) it should implement a valuable contribution towards the complete feature and 3) it should leave no loose ends. Quality assurance of code and ancillary work products produced during the execution of a story happens while the story is being executed.

This paper presents six patterns that will help the team find, shape, organise, and execute the stories needed for implementing a complete feature. But first it is necessary to explain the origins and values of the Story-based Method, and to define a few key concepts.

*Copyright © 2008 Jan Reher and Systematic Software Engineering A/S. Contact the author at electro-magnetic mail address jar@systematic.dk.

1.1 Origins of the Method

The Story-based Software Development Method was invented by the author at the company where I work as Senior Developer. The company is called Systematic Software Engineering A/S and has its headquarters in the city of Århus in Denmark. Systematic is accredited with a level 5 maturity rating according to the Capability Maturity Model [Chrissis+06], and is currently working on further improving its ways of working by incorporating agile and lean practices.

The Story-based Method has been in use at Systematic since September 2005, and is currently being executed in 15 software projects ranging in size from one-year projects with just three developers to multi-year projects with 50 developers supported by testers, user interaction designers and other expert roles.

Story-based Software Development is a lean method because it embodies and realises the priciples of Lean Software Development [Poppendieck+03]. These include: communication and learning, the elimination of waste, building quality into the product, deferring commitment, delivering early and frequently, respecting and empowering the development team, and optimising the whole.

The method is also compliant with the requirements of the Capability Maturity Model, though using the method is far from enough in itself to earn an organisation a CMMI certificate.

The method is inspired by the works of several methodologists:

Alistair Cockburn For insisting that methods should be habitable by people [Cockburn04]. The Story-based Method was conceived among developers who refused (and still refuse) to use methods that they feel are getting in their way. And for explaining why and how to use use cases for analysing and documenting requirements [Cockburn00]. The Story-based Method recommends use cases.

Ivar Jacobson, Grady Booch, and James Rumbaugh For the completexness of their Unified Method framework [Jacobson+99]. The Story-based Method is by no means a complete method but the Unified Method framework showed how to define a method with a well-defined border and how to provide clear connections to other methods and techniques.

Mary and Tom Poppendieck For stating the Lean principles which is one possible and meaningful definition of what constitutes a “good” method [Poppendieck+03]. The Story-based Method is based on the Lean principles.

Watts Humphrey et al. For the rigour demanded by the Capability Maturity Model [Chrissis+06]. The Story-based Method is CMMI-compliant.

Kent Beck For Extreme Programming [Beck04] which shows that it is acceptable to state hard rules that may not be broken, provided that you explain why. Like XP, the Story-based Method is fairly dogmatic at times. And for [Beck03] which states that before you go about solving a problem by writing code, you should state the problem as a (failing) test. The Story-based Method encourages defect containment and early testing.

The Story-based Method is not a universal method, neither in terms of applicability nor completeness. It focuses on the core task of transforming a requested and fairly well-defined feature into complete, tested, and releasable software. It draws upon the strengths of a number of best practices and patterns from many disciplines of software engineering without being particularly dependent on them, notably the use of unit tests for testing [xUnit], use cases for specifying requirements [Cockburn00], early testing for reducing risks and increasing efficiency [Poppendieck+03] [Beck03], continuous integration [Fowler06] for bringing the system's components together, and an iterative and incremental approach to project management, for example Scrum [Schwaber+01].

1.2 The Feature

A key concept of the Story-based Method is the feature. It is defined thus

> **Feature**: A large-granularity lump of coherent functionality. A feature can be described in detail by a set of requirements, preferably expressed mainly as use cases (See section 1.4). Whether the development team needs a detailed description before the implementation of a feature can begin, or the team can make do with an overall description, depends on the nature of the project and the contract, and on the team's relationship with the customer.

The Story-based Method tells how to define, scope, estimate, design, code, document, and test a feature. The patterns in this paper focus on planning and executing the work needed to move the feature from a fairly precise description to deliverable software.

In the context of this method, implementing a feature means moving it from concept to reality. Implementation therefore involves both writing code and producing ancillary work products, and ensuring the quality of them all.

The Story-based Method is primarily a feature-driven approach to software developments. It is not, for example, primarily architecture-driven or primarily test-driven. This means that the question one should always ask first is *"how does what the team is doing right now contribute to the delivery of the current feature?"*, and secondly, *"how does the team maintain and extend the architecture?"* and *"how does the team test this?"*.

1.3 An Example Feature

An example feature will illustrate the patterns presented in this paper. The feature is not far from an actual feature that was implemented in a real system using the Story-based Method.

The example concerns a system that stores and manages some sort of data, and helps users manipulate the data. The data could be stocks, ships and their cargo, email messages, documents, or something else. The example feature is a *monitoring window* that pulls interesting data out of the system and presents the data to the end user.

This feature presents two main challenges to the development team: First, the list of potentially relevant data that users will want to see is practically endless, and it is difficult to prioritise what data presentations to implement and deliver first. Second, the team cannot be sure how the data should be presented without trying the window out. Both problems are best solved by getting some feedback from real use, or at least comprehensive tests. Thus this feature is fairly easy to describe in overall terms but it should not be analysed and designed in depth up front.

To get the implementation work started, the team should probably at first establish a preliminary architecture for the subsystems that will realise the feature, and make a preliminary design of the user interface.

The preliminary architecture could be fairly simple. Drawing from the body of useful design patterns, assume that the team decides on a layered architecture: At the bottom the team puts a layer acting as an adapter between the various subsystems that provide data to our monitoring window and the window itself. On top of the adapter is an internal data representation, and on top of that is the GUI itself.

The best way to make a preliminary design of the user interface probably involves some amount of prototyping. Let's assume that we decide on a resizable and tabular data presentation. The end-user can use the window as it is, or he can do his own configuration of it.

This concludes the definition, scoping, and preliminary design of the feature. The next step is to find and describe the stories that will make up the implementation work. Each pattern in this paper includes a continuation of this example.

1.4 Concerning Use Cases

The Story-based Method recommends use cases as defined by Alistair Cockburn [Cockburn00] as a technique for finding and documenting requirements, and for testing and validating the implemented system. Here, I will briefly outline the technique:

A use case is a piece of structured prose describing an interaction between an *actor*—usually a person—that is seeking some *goal*, and a *system*—usually a software system—that is meant to deliver that goal. An actor may follow different paths while seeking the goal, depending on circumstances. These paths are called *scenarios*.

The *main success scenario* of a use case is the straightforward path through the use case where both the system and the actor act as appropriate as imaginable, and nothing goes wrong. It is usually the path that is used the most and provides the most value. Other scenarios are termed *extension scenarios* and constitute other—often more convoluted—paths through the use case. Some of these reach the goal, while some fail to reach the goal.

Use cases usually have *preconditions*, things that must be in place before the use case can commence. Use cases usually provide guarantees in the form of *postconditions* that are guaranteed to hold when the use case ends.

The key advantage of specifying requirements as use cases is that is focuses on delivering value in the form of goals to the end-user. Other advantages are that the use case form is a robust and inspiring template for analysing business processes and specifying requirements, and that most stakeholders in a software project can understand and relate to use cases.

In the following I will assume that features are specified using use cases, and that you, gentle reader, are familiar with use cases. Please refer to [Cockburn00] for a full treatment.

1.5 Concerning the Importance of Frequent Closure

A key value of the Story-based Method is to reach frequent closure. I claim that any software development method should provide this. Frequent closure must be achieved in several areas:

- **Functionality** is what provides value to the end-user. Frequent closure of functionality enables the development team to frequently deliver new (parts of) features to the customer. This facilitates communication, feedback, and learning as well as providing value to the customer. In the Story-based Method the preferred atom of functionality is a complete use case scenario.

  Without frequent closure of functionality, the development team and the customer are unable to learn from actual usage of the system. They are therefore forced to work according to a plan and not according to changing needs and circumstances [Poppendieck+03].

- **Quality** is what makes functionality useful and valuable. With frequent closure of quality (what [Poppendieck+03] calls a zero-defect policy) new functionality will be useful, whereas without it, new functionality will merely be interesting.

  Without frequent closure of quality, the customer looses confidence in the project, while the development team pushes an ever growing heap of problems into the future.

- **Architecture** is what makes the system internally sound and able to grow and change. [Bass+03] calls this *architectural qualities not discernible at runtime*, which encompass coherence and low coupling, clarity, etc.

  With frequent closure of architecture, the development team retains control over the growth of the system as more and more features are implemented, and remain able to add new features.

  Without frequent closure of architecture, the architecture degenerates. New functionality can only be addded through excessive labour, and quality is likely to suffer.

- **Work products** means any internal or external artefact that the development team must produce [Chrissis+06]. This includes the system's software and hardware components as well as ancillary material: user manuals, test specifications, design documentations, teaching material, installation and maintenance software etc. With frequent closure of work products, the overall set of deliverables stays coherent and useful.

Without frequent closure of work products, the system will be harder to test, install, use and understand.

- **Plan** means that any work package which the development team is responsible for should be small and comprehensible, and that at any given time any person in the development team should be working full-time on only one work package. So at any given time, the number of work packages in progress is limited, and their expected completion is in the near future.

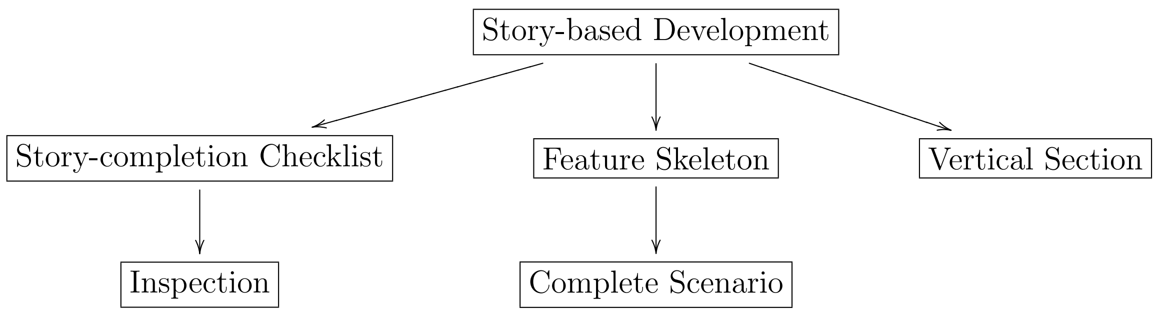
  With frequent closure of plan, the project team is able to accurately state their progress to date and expected future progress. This in turn makes it possible to re-prioritise future work packages and re-plan remaining work.

  Without frequent closure of plan, it is difficult to say how much work has actually been completed, and hence how much remains. This in turn makes it impossible to re-prioritise and re-plan, and missed milestones are likely.

The pattern descriptions that follow will discuss how each pattern helps to achieve frequent closure in these araes.

2 Patterns

This section describes the patterns currently identified in the Story-based Method. The patterns are arranged in a pattern language as shown in the diagram below.



2.1 Story-based Development

All implementation work should be done according to the Story-based Method.

. . . the team needs to create or modify a software system to add a feature, reorganise parts of the system's architecture, or fix a non-trivial bug.

◇ ◇ ◇

How should work be divided into work packages so that each work package is of manageable size and shape, provides value, and forms a closure?

Consider each work package that work may be divided into.

The work package should be of *manageable size* as expressed by the number of hours a team of developers need to complete it. What this is depends on the team's situation and skills. The size limit may be set at 20, 30, or even 50 hours. This limit means that the people responsible for a work package should be able to complete it within one or two work-weeks. This timeboxing ensures that delays are spotted easily and early so that corrective actions can be taken.

But simply time-boxing the work is not enough. A work package should also provide *value* to a stakeholder, e.g. the customer, the end-user, or fellow team members, by implementing a part of the feature being worked on. If the team can achieve this, then the completion of each work package will be an opportunity for presenting completed work to stakeholders and getting feedback on it. This is necessary for steering towards providing the most value to the customer and end-users.

Furthermore, the work package should be a *closure* that leaves no loose ends dangling, e.g. tests that do not pass or design documentation that does not correspond to the code. Just writing the code is not enough. Reaching closure is very satisfying in itself, and presents an opportunity for pausing and reconsidering options. Section 1.5 has more to say on the importance of reaching frequent closure.

Work packages should also be ordered so that those that carry a high risk can be treated with special care, using special tools and resources, and can be executed early.

One possible way of decomposing the work is to use a phase-based approach. The phases are traditionally something like:

1. Analyse the requirements, document them, and do quality assurance of the requirements specification.

2. Design a solution for the requirements, document the design, and do quality assurance of the design description.
3. Write the code according to the design, document the code, and do quality assurance of the code.
4. Test that the code fulfills the requirements.
5. Write the users' manual.
6. Release the system.

The merits and drawbacks of a phase-based approach have been discussed by numerous authors [Jacobson+99] [Poppendieck+03] and will not be repeated here. Suffice it to say that this approach is inadequate. Specificallly, it does not balance the forces described above.

Therefore:

Use stories as the unit of work.

The word *story* is hopelessly overloaded and ambiguous, and present different connotations to different people, but my colleagues and I have been unable to invent a better name. It has a precise definition within the scope of the Story-based Method, and I will go to some lenghts to explain it.

The term "story" is taken from XP's analogous "user story" concept. XP in turn borrowed the term from Alistair Cockburn [Cockburn00], and twisted its meaning quite a bit.

A story is defined thus:

> **Story**: A tiny software development project executed by one or a few developers with the goal of implementing a tiny, incremental bit of the complete desired feature. The story includes all the activities usually done in software projects, but is limited to an effort of about 40 hours.

An important point is that a story is a unit of work, not a unit of functionality. This distinguises the Story-based Method from for example Extreme Programming where a "user story" means both a small unit of functionality requested by the customer, and the work required to implement that functionality. In the scope of the Story-based Method, a story is just the unit of work. A story may implement coherent functionality in the form of scenarios in a use case; indeed, the pattern COMPLETE SCENARIO encourages the team to execute stories that do precisely that.

So not only must the team write the code necessary to implement the story's contribution to the feature, the team must evolve and document the design behind the code, it must design and implement necessary manual and automated tests, it must write the required user's documentation, it must integrate code, and it must perform quality assurance of all the work products touched. This will ensure that the system stays stable and coherent, and retains its required qualities as the team works on it.

This hasty enumeration of activities is not enough. Some structure is needed. Please see the pattern STORY COMPLETION CHECKLIST.

Stories can be ordered according to value, dependencies, and risks. In the terms of [Schwaber+01], stories are items on the sprint backlog. More generally, the pile of stories form a WORK QUEUE [Coplien+05].

One or a few developers are responsible for each story. Assigning two developers to a story yields the advantages of DEVELOPING IN PAIRS [Coplien+05]. More people working on a story are likely to get in each other's way. Stories are sometimes linearly dependent on each other so developers should work on a single story at a time. Sometimes a batch of stories can be executed in parallel and this should be exploited where appropriate. The stories that implement a given feature should be handled by a dedicated group of developers. This gives the advantages of FEATURE ASSIGNMENT [Coplien+05].

Depending on the team's needs and level of rigour, it will be able to confidently release the system and its supporting work products at the completion of every single story. At the very least, a completed story presents an opportunity to ENGAGE CUSTOMERS [Coplien+05].

Experience indicates that the total effort needed to execute the activities in the story should not exceed 40 hours.

◇ ◇ ◇

This pattern is the starting pattern of this paper, indeed of the Story-based Method, in the sense that all other patterns in the paper build on the context established by STORY-BASED DEVELOPMENT. Thus, he pattern STORY COMPLETION CHECKLIST explains how each story should be structured, while the pattern INSPECTION explain to to do quality assurance on the activities performed in a story. The patterns FEATURE SKELETON and COMPLETE SCENARIO provide advice as to the goal of each story.

Stories provide a common template for work packages. The template is useful when designing, estimating, planning, and executing implementation work. Estimating work by decomposing it into stories is no more difficult per se than with any other work breakdown structure, but it takes a particular mindset and it takes practice.

Story-based Development results in closure of work products, quality, and plan (section 1.5) at the end of the execution of each story, which happens several times a week in most project teams:

- Work products: A story is complete only when all relevant work products are completed.
- Quality: A story is complete only when the inspector is satisfied with the quality of the work done. See Inspection.
- Plan: A story is of limited scope and duration, so scope or effort estimate overruns are detected early.

Not all the work that is necessary to implement a feature should or must be done as stories. Notable exceptions include:

- Analysing users' needs and requirements: This activity produces no executable code.
- Building simple proof-of-concept prototypes for studying or validating candidate technologies, techniques, or designs: Prototyping work often focus on key aspects of a complete solution and does not need the full quality assurance of the Story-based Method.
- Finalising documentation.
- Fixing trivial bugs.
- Refactoring code.

How to perform such work is outside the scope of this pattern, and of this paper.

As a rule of thumb, any work that substantially affects the production code of the system should be performed as stories. Other work should probably not be performed as stories. Beware of overusing this pattern by shoehorning all work packages into the story template.

Example:
The work of implementing the Monitoring Window described above will be organised as a number of stories. As stated, it is difficult to determine the exact requirements for the Monitoring Window except through daily use. The team will therefore plan and execute a limited number of stories to implement a simple first version of the window, and release it to the users as frequently as possible to get feedback from them. The team is able to do this because each story is a closure of quality, work products, and plan.

2.2 Story Completion Checklist

A checklist provides structure to the activities necessary to execute a story.

…the team is doing Story-Based Development.

◇ ◇ ◇

How should the activities that must be performed in a story be structured?

Software development involves many activities, e.g. analysing user needs, designing and executing tests, inventing a design, writing code, and writing design documentation. These activities need to be structured to some degree. This is true for the development project as a whole, and true for the individual story, which can be regarded as a miniature software development project.

The team wants to impose enough structure on this work to help the developers responsible for the story remember what they need to do, in which order, and to what extent. But the structure must not be too rigid. Good developers are intelligent people, and they will most likely rebel against enforced and unwarranted structuring.

The structure should express best practices for software development. Currently at my company, these include techniques like refactoring, developing the design while you write the code to increase feedback and learning, and early testing by specifying and running automated unit tests while you write the code. However, individual teams should be free to explore new ways of working and not be tied to any specific technique or practice.

The structure must also only impose a light ceremonial overhead on real work.

Finally, the structure must support frequent closure as discussed in section 1.5 and defect containment by insisting that a work package is really complete before progressing to the next one ("done-done" in the sense of [Poppendieck+03]).

Therefore:

Use a checklist to drive the execution of a story.

The checklist should be brief enough to print on a single sheet of paper. The checklist's items should state what activities must be performed. Detailed but generic descriptions of the activities can and should be written elsewhere. These descriptions form the project's or the organisation's body of knowledge about software development and should be consulted and updated by developers when appropriate. But the story completion checklist expresses the essence of that knowledge, and is used for driving day-to-day work in stories.

In my experience a development team on a given project needs to execute more or less the same activities for each story they work on. Thus, all stories on the project can be driven by the same checklist. The checklist can and should be tailored to the project's particulars.

Below I present a fairly generic story completion checklist that is based on my organisation's best practices as discussed above. It should provide a good starting point for most projects. The list is linearly laid out, but though progressing from top to bottom is indeed the preferred route through the checklist, developers are supposed to interleave and iterate the activities as appropriate:

Feature: ______________
Story: ______________
Developers: ______________
Inspector: ______________

Activity	**Done**	**Inspected**
Reconsider the scope and estimate of the story	☐	☐
Analyse needs and requirements	☐	☐
Draft the user interface	☐	☐
Draft the user documentation	☐	☐
Identify existing manual tests that must be run	☐	☐
Draft new tests (manual and automated)	☐	☐
Draft the design	☐	☐
Draft the code	☐	☐
Write new manual tests	☐	☐
Write new automated tests	☐	☐
Perform refactorings	☐	☐
Complete the code	☐	☐
Execute existing and new manual tests	☐	☐
Complete the design documentation	☐	☐
Complete the user documentation	☐	☐
Integrate the story	☐	☐

Story complete: ______________________________________
Date Developers Inspector

When beginning a story, the developers fill in the fields at top. Each story should have a name that states its goal in brief. The developers then cross out activites that do not apply to this particular story, and add extra necessary ones. As the developers complete activities, they mark this on the checklist, and request an inspection of their work (See the pattern INSPECTION). When the inspector approves an activity on the checklist, the developers may proceed. Thus the checklist connects implementation to inspection, and the developers to their inspector.

The story is done when all activities are done, and the inspector has inspected and approved all activities.

At the end of the story, the developers and the inspector sign the checklist. By putting their signature on the story completion checklist, the people involved signify their responsibility for the scope and quality of the work performed.

◇ ◇ ◇

No two stories performed according to the checklist will be the same, yet they will share a common structure. That is, they will express the same pattern.

The checklist helps developers reach closure in these areas:

- **Quality**: The checklist encourages developers to consider design, coding, and test as interrelated aspects of the implementation work. This helps ensure that new functionality implemented in the story is of a sufficient quality. Other developers can help with quality assurance by doing INSPECTION on the story.

- **Work Products**: The checklist ensures that the story is not done when the code is complete, but only when all ancillary work products are done too.

- **Plan**: Each item on the checklist marked "Done" signifies measurable progress. This includes integrating the code and other work products produced in the story with the team's collected work products. Thus the checklist fosters INCREMENTAL INTEGRATION [Coplien+05].

EXAMPLE:
Implementation of the Monitoring Window feature should start with a FEATURE SKELETON (section 2.4). This story should be executed according to the story completion checklist, with the following modifications:

- The user interface is not terribly important for this story. Also, there are probably no existing manual tests for the Monitoring Window. Therefore activities that concern these aspects of implementation should be omitted, i.e. crossed out on the printed checklist.

- This story is primarily concerned with demonstrating the validity (or invalidity!) of the proposed architecture for the Monitoring Window. Therefore the team should add an activity near the end of the checklist that evaluates the produced design and code against the proposed architecture. Simply add this activity in handwriting on the printed checklist.

2.3 Inspection

Quality assurance of work done in stories is performed on the spot by an inspector role.

. . . the team is doing Story-Based Development. Work in stories is structured according to a Story Completion Checklist.

◇ ◇ ◇

How should the quality of work performed in stories be assured?

Quality assurance of developers' work is important. One of the most effective ways to do this is to write automated tests [xUnit] and the Story Completion Checklist lists several activities concerned with automated tests. Another very effective technique is to have someone else review your work and comment on it. The traditional way to do this is to hold review meetings where e.g. the design, code, or user interfaces of the system are scrutinized by experts with the purpose of finding defects and improvement opportunities.

Reviews also present an opportunity for knowledge sharing, and for mentoring of less-experiences developers.

However, quality assurance through review meetings is too heavy and cumbersome to work in the context of the Story-based Method: Because the amount of work done in a story is quite small, the team cannot afford the administrative overhead imposed by conducting a review meeting with several people involved after each completed activity in the Story Completion Checklist, or even after each story. An alternative is to postpone reviews until the team has collected a sufficiently large pile of completed stories, and then review them all together. But this means putting work on the shelf, which according to [Poppendieck+03] is waste, and it means that errors or improvement opportunities are discovered late.

These ill effects are worsened if the team also wants the reviews to provide knowledge sharing, and mentoring, both of which must be timely to provide the most benefit.

Therefore:

For each story, assign the role of *inspector* to an experienced developer, and have the inspector assure the quality of developers' work as the story progresses.

The inspector should be well acquainted with the feature being implemented, preferably by being the one responsible for the realisation of the complete feature. The inspector must be on call for immediate inspection.

When the developers think they have finished an activity, they call for the inspector. He then reviews the work products produced by the developers, interviews them about their work, and provides mentoring and advice. The inspector should adopt the role of WISE FOOL [Coplien+05], and should make a habit of asking stupid and lateral questions.

Any defects found (and the inspector has the final say as to what constitutes a defect) must be corrected immediately. When he is satisfied with the quality of the work, the developers may proceed with the story.

The inspector should adjust his level of interference to the maturity of the developers responsible for the story. Inexperienced developers may benefit from frequent inspection. In this manner, inspection is a way to implement APPRENTICESHIP [Coplien+05].

The most important work product of a story is the code. Therefore, the inspector must be a developer. The inspector must also have a thorough understanding of disciplines such as user interaction design, test, and architecture to be able to evaluate these parts of the developers' work. He should, however, be aware of the limits of his own knowledge and involve experts in these disciplines when necessary. If the ARCHITECT ALSO IMPLEMENTS [Coplien+05], he can credibly inspect developers' code. Inspection will then contribute to achieving ARCHITECT CONTROLS PRODUCT [Coplien+05].

Thus quality assurance by inspection is an integrated and ongoing activity in the Story-based Method, and most defects are corrected minutes or hours—not days or weeks—after they are introduced.

◇ ◇ ◇

Together with the STORY COMPLETION CHECKLIST, inspection helps ensure closure of quality and work products.

Inspection does not eliminate the need for testing. Different kinds of errors are best found using diferent techniques. Run-time errors, such as failure to comply with a requirement or plain bugs, are best found using tests, be they manual or automated. Other errors, such as misuse of programming language, architectural rot, or inconsistent documentation, are best found by inspection. Thus inspection and testing are complementary.

A potential danger of this patterns is that inspection can sometimes focus too much on the details of the individual story at the expence of the overall quality of the feature's implementation. The developers and their inspector should discuss this risk. Depending on the project's quality goals, it may be a good idea to execute additional quality assurance activities addressing an entire feature or component, such as reviews of code, design documentation, test specifications, or user manuals.

An inspector cannot be expected to inspect more than about ten developers at the time, and then he will have most of his time full. This of course means that he cannot do much implementation work himself. If the inspector focuses too much on his own implementation work, he will become a bottleneck.

A variant solution that remedies this is to let developers be inspected by peer developers. Compared to the primary solution of this pattern, it frees the experienced developer to do other work than planning and inspection. On the other hand, it prevents the experienced developer from sharing his experience by doing inspections. Peer-to-peer inspection has the additional advantage that it fosters knowledge sharing among developers, and trains future inspectors. The team must decide how to balance these forces.

(Note: This variant solution might be considered a separate pattern: Peer-to-peer inspection.)

Example:
For the Monitoring Window, assign the role of inspector to the person who described its initial requirements and designed the initial architecture. This person should also be involved in the actual implementation of the feature, i.e. he should also execute stories. Some other developer will have to do the inspection of these stories. This could be an inspector working on another feature, or a developer working on the Monitoring Window feature, who can thus be trained in the role as inspector.

2.4 Feature Skeleton

The first story should implement a skeletal version of the feature.

…the team is doing Story-based Development and needs to add a feature to a system.

◇ ◇ ◇

There are many possible places to start implementing the feature. What should be done first?

Frequently, many aspects of a feature are not clear. These could be the feature's priority, the customer's real needs, algorithmic issues, risks, the user interface design, the domain object model, or third party tools or technologies to use.

The team needs to address these unknowns, and the sooner the better. The team does not want to spend too much time merely contemplating problems. The team needs to get started on implementing its best guess so that there is a sense of progress, and so that both the team and other stakeholders can get real feedback from the system. If the team does not get rapid feedback on its best guess, it is likely to be off the mark, and valuable time will have been wasted.

One way of getting rapid feedback is through prototypes. Prototyping work should focus on resolving just one aspect of a problem, and should disregard other aspects and qualities. This means that the development of a prototype is not a closure, neither with regard to functionality, quality, nor work products. The prototype cannot be released to the users. In essence, you cannot trust a prototype.

Therefore:

Start with a story that implements a skeletal version of the complete feature.

The story need not deliver enough of the complete feature to be of any real end-user value. Focus on exploring one or a few of the unknowns, on reducing risks, and on providing the architectural foundation (hence the "skeleton") for the feature. This foundation is best discovered by considering the feature as a whole. Also, domain knowledge is very important. Thus, doing a feature skeleton is one concrete way to Get On With it [Coplien+05].

A skeleton should have all its bones in place. This means that the story should be a Vertical Section that implements something in all layers of the architecture.

Sometimes developers can actually accomplish quite a lot within the scope of a feature skeleton story. They can after all aim close to the 40-hour ceiling, and include a bit of "meat" on the skeleton. But it is much better to focus on getting the skeleton done in this story, and then execute other stories afterwards that gradually add more and more meat. So if a skeleton story's estimate turns out to be close to the ceiling, remove some "meat". This will reduce the story's scope, risk and estimate accordingly.

A FEATURE SKELETON is frequently the right place to start implementing a feature. Once the skeleton story is done, execute stories that implement COMPLETE SCENARIOS.

But skip the FEATURE SKELETON if there are no significant unknowns to consider. Go straight to stories that implement COMPLETE SCENARIOS.

Note that the implementation of a FEATURE SKELETON is *not* a prototyping session. The developers must execute all the usual activities on the STORY COMPLETION CHECKLIST, and inspection must still be done. This ensures that the story remains a closure.

If the team needs to build a prototype to prove feasibility or explore possible solutions, it should do so but be conscious of that that is what the team are doing. Once the team has resolved the problem, toss the prototype. See BUILD PROTOTYPES [Coplien+05].

◇ ◇ ◇

A FEATURE SKELETON will provide the development team and other stakeholders with a sense of early success and of being well on the way. It also helps in planning and prioritising the work ahead.

The story is likely to implement a small bit of useful functionality, which subsequently can be shown to stakeholders to receive some early feedback.

The customer might even want to start using the system as it is. Since a story is a closure of quality as defined in section 1.5, it is safe to do so.

This pattern is analogous to the architecture pattern WALKING SKELETON, and also provides EARLY SUCCESS [Cockburn04]. It addresses some of the same forces as the XP pattern BOOTSTRAP STORY [Andrea01] but is able to present a different resolution of the forces because the principles of the Story-based Method are different from those of Extreme Programming.

EXAMPLE:
Start work on the Monitoring Window feature by executing a story that implements a FEATURE SKELETON. The story will also be a VERTICAL SECTION through all three layers of the intended architecture: The adapter

layer, the model layer, and the presentation layer. Some aspects of the feature can be factored out to trim the story down to a true Feature Skeleton:

Simple Data: The story will produce a system that can monitor just one piece of data. Pick one that is fairly easy to dig out of the system and fairly high on the customer's prioritised list of data to monitor.

Simple User Interface: The presentation layer will be a simple window displaying just the one piece of data. What constitutes a simple window depends on the available GUI toolkit. It could be simply a modal message box.

Simple Algorithm: Eventually the monitor must periodically sweep the system and update itself but to further reduce scope this story will implement a monitor that simply performs one sweep of the system when it starts.

2.5 Vertical Section

A story should touch several layers in the architecture of the system or sub-system at hand.

. . . the team is doing Story-based Development, and needs to add functionality to a system, or to enhance a quality of a system.

◇ ◇ ◇

As the team extends the system, the developers need to consider both functionality (the feature) and form (the system's architecture). How does the team balance these?

Any system or sub-system has an architecture, whether it is intentional or accidental. Well-designed systems tend to have a component-based architecture with a clear layered structure [Bass+03]. A given feature does not reside in any single layer, nor in any single component. Rather, a feature is an emergent property of the collaboration of several components in a system.

Picturing the architecture of a system as a stack of horizontal layers, it makes sense to say that a feature is orthogonal to the architecture; the feature emerges from a set of vertical sections through the system.

One possible way of structuring the work of implementing the feature is to create a work package for each layer or component that must be modified in order to implement the feature. This approach has a number of drawbacks:

- It requires that the team first construct a fairly detailed design of the modifications that must be made to the system. This design must be made without writing any code to validate it, because if developers did write the code, they would be executing the work which they are actually only trying to estimate and plan.

- The team cannot validate the completed solution until all or most of the work packages are complete. Each component can to some degree be tested in isolation while it is being implemented but verification and validation of the interaction between the components and their emerging properties (i.e. the feature) will not be possible before the end. Thus each work package is not a closure, and feedback and learning after each work package is not optimal. Wasted effort is likely.

- Risks tend to reside in the interaction between components rather than inside components. These risks cannot be handled early.

- Component-based development tend to proceed either top-down or bottom-up. Neither approach delivers a feature that is even partly useful to a customer before the very end.

Therefore:

Design each story so that it changes the design and code of several layers of the architecture.

When looking at a system with a layered architecture, the parts of the system changed during the execution of the story should form a vertical section through the architecture.

One way of doing this is as follows:

1. First, execute a story that builds a Feature Skeleton. Since this story focuses on architecture, it is likely to be a Vertical Section, too.
2. Then, execute a number of stories that each are Vertical Sections, and incrementally adds functionality on the side (so to speak) of the result of the previous stories. Repeat this until the feature is completed.
3. Frequently evaluate the result of the stories with special attendance to the balancing of functionality and architecture, get feedback, reconsider and re-plan as necessary.

◇ ◇ ◇

A story that forms a Vertical Section provides architectural closure because the developers executing the story are forced to understand, consider and maintain all layers of the system's architecture as they add new functionality. This supports a continuous and controlled enhancement of the architecture. In [Poppendieck+03] terms: The developers are forced to see the whole because they need to touch the whole.

Sometimes it is not possible to touch all layers of the architecture, for instance because some layers are being developed by another organisation, or because they have already been developed and are now fixed. In this case, merely extend the vertical section as far as it can go.

Example:
Stories for the Monitoring Window feature should form a vertical section through all three layers of the intended architecture: The adapter layer, the model layer, and the presentation layer.

2.6 Complete Scenario

A story should implement a complete scenario of a use case.

. . . the team is doing Story-based Development. The feature to be implemented is expressed as use cases or in some similar form that focuses on interaction scenarios between an actor and a system delivering the feature.

◇ ◇ ◇

Some use cases in the feature are too large to be implemented in one story, so the effort must be split into several stories. What goal should each story have?

The team wants to first implement those parts of the use cases that provide the most value to the customer. There are two possible reasons for this: First, the time available for implementation may be limited, either by a deadline or a budget, or both, and the team must use that time to provide the most value. Second, implemented parts can be delivered to the customer, and this will provide user happiness, or cash flow, or both.

A use case has two facets: Scenarios define the overall flow of the use case, while business rules and pre- and postconditions define the details. A use case scenario is completely implemented only when both the flow and the details are in place.

But it can be difficult to implement both flow and details for a whole scenario within the limited timeframe of one story. And frequently the details cannot be clearly defined until the system has been used in real work situations.

Splitting the scenario into two or more consecutive parts for implementation in consecutive stories is no solution since a partly implemented scenario provides no value.

Therefore:

Design stories that implement one or several complete scenarios.

If story estimates get too high, disregard complicated business rules, pre- and postconditions, invariants, and other complications. Frequently these can only be fully analysed and defined after the system has seen real use anyway. This work belongs in separate stories.

Stories that implement Complete Scenarios should be executed in sequence so that those that have high development risk or high customer value are executed first.

Such a story should have the name of the scenario that it implements. This will help communicating the story's goal to stakeholders, and aid in prioritisation of the story. See IMPLIED REQUIREMENTS [Coplien+05].

◇ ◇ ◇

With the completion of each story that implements a COMPLETE SCENARIO the team has achieved closure of functionality by adding a piece of useful and coherent functionality to the system. There may be some details missing which anyway are best discovered by getting feedback from real use of the system in the customer's environment. And this requires that complete scenarios are there for the customer to use.

EXAMPLE:
Continuing with the monitor window example, assume that the feature includes a use case where the user can have a snapshot of the data displayed in the window dumped to a file. The user must be able to select which format to dump to (e.g. Excel, comma-separated values, or XML), the name of the file, whether to overwrite an existing dump file of that name, and which data to include in the dump. For the sake of the example, assume that the complete use case is too much work to fit into a single story. Domain experts tell the team that the most likely scenario involves selecting XML format, the filename "dump.xml", and to dump whatever the monitoring window currently displays.

This is a good COMPLETE SCENARIO, and the team should execute a story that implements it. This will result in a useful bit of functionality, and also provide a foundation for subsequent stories that tackle the other scenarios in the use case, and add other details.

3 Related Work

3.1 Extreme Programming

The Story-based Method shares many characteristics with Extreme Programming [Beck04]. Notably, both methods are fairly dogmatic in the sense that they state (different) sets of mutually supporting rules that are broken at the development team's own peril. For example, XP insists that all production code be written while pair programming, and the Story-based Method insists that all work products be verified through inspection.

The values that underlie the two methods are very similar and roughly defined by the buzzword "Lean". However, the methods achieve these values in different ways, and a team executing the Story-based Method does not look and behave like a team executing Extreme Programming.

Gerard Meszaros' paper *Using Storyotypes to Split Bloated XP Stories* [Meszaros04] and Jennitta Andrea's paper *Managing the Bootstrap Story in an XP Project* [Andrea01] describe patterns for managing Extreme Programming user stories. The papers discuss their patterns in the context of XP with an emphasis on use cases. The patterns presented in this paper can be seen as extensions and generalisations of Meszaro's and Andrea's patterns, stated from the viewpoint of the Story-based Method.

3.2 Patterns for Effective Use Cases

The book *Patterns for Effective Use Cases* [Adolph+02] builds on the work of Alistair Cockburn [Cockburn00] by presenting a number of patterns that can be used to identify, size, and shape the use cases that describe a feature. The patterns are somewhat similar to the patterns presented in this paper: Both are most readily applicable in a software development context where features (requirements) are described and understood as use cases, and some of the patterns presented in this paper have direct analogies in [Adolph+02]. The table below lists some:

This paper	**Adolph et al.**
Story-based Development	The book's basic argument why use cases should be used at all is not represented as a pattern
Vertical Section	No analogies
Complete Scenario	Complete Single Goal
Construct a Feature Skeleton, then repeatedly implement a Complete Scenario	Breadth Before Depth, Spiral Development, and Ever Unfolding Story

However, the two sets of patterns have different areas of applicability: The patterns in [Adolph+02] will help writing use cases that describe a feature, while the patterns presented in this paper will help organising the implementation effort necessary to bring those use cases to life.

4 Summary and Conclusions

This paper has presented 6 patterns embodying advice on how to size, shape and execute stories.

- First and foremost, do Story-based Development.
- Execute each story by following the Story Completion Checklist.
- Perform quality assurance through Inspection.
- Strive to make each story work on a Vertical Section through the system's architecture.
- Start with a Feature Skeleton and then repeatedly execute stories that implement a Complete Scenario.

Section 1.5 described the importance of frequent closure in a number of areas. It is time to revisit these areas and conclude to what extent the Story-based Method and the patterns support this quality:

- **Functionality** Stories that implement a Complete Scenario implement useful functionality which can be delivered to the end-users. This facilitates communication, feedback, and learning.

- **Quality** By structuring stories according to the Story Completion Checklist and doing Inspection each story will not only deliver new code that realises new functionality, but will also do design and execute tests necessary to verify and validate the new code. This means that the functionality realised in the story can not just be demonstrated to the customer, but can safely be used for real work in the end-users' environment, thereby providing value to the end-users.

- **Architecture** By executing stories that form Vertical Sections through the architecture, the development team is forced to constantly consider the system's architecture, and to maintain and develop it as they add new functionality. Therefore, the architecture remains sound at the end of each story.

- **Work products** By structuring stories according to the Story Completion Checklist and doing Inspection, each story will not only produce and test new code, but will also produce ancillary work products like installation and user manuals. This ensures that the complete set of project deliverables are coherent and complete at the end of each story, and that the functionality implemented in a story can be transitioned into the end-user's environment.

- **Plan** When doing Story-based Development, the story is the development team's unit of planning. Because there are well-defined rules—Story Completion Checklist and Inspection—for deciding whether a story has been completed or not, there is never any doubt as to which work packages are completed and which are not, and the development team's progress to date and expected future progress can be readily determined.

The patterns presented in this paper constitute a first attempt at establishing a pattern language for planning and executing stories. No doubt further experience with the Story-based Method will give new insights into the nature of good and faulty stories, and reveal new patterns. Whether or not these patterns coalesce into a true pattern language remains to be seen.

5 Acknowledgments

My thanks go to my VikingPLoP 2007 shepherd Michael Weiss for his detailed reviews of the paper and for inspiring questions and comments.

In like measure, thanks to Met-Mari Nielsen for suggesting that I write this paper, for introducing me to the PLoP way, and for review.

Also thanks to the participants of the VikingPLoP 2007 conference in Bergen, Norway for three delightful, challenging, and encouraging days. May we meet again.

Thanks to my colleagues at Systematic for time, patience, enthusiasm and reality checks.

Final thanks to Michael Holm, owner and managing director of Systematic, for creating the space.

References

[Adolph+02] Steve Adolph and Paul Bramble: *Patterns for Effective Use Cases*. Addison-Wesley 2002, ISBN 0201721848.

[Andrea01] Jennitta Andrea: *Managing the Bootstrap Story in an XP Project*. Retrieved from www.jennittaandrea.com.

[Bass+03] Len Bass, Paul Clements, and Rick Kazman: *Software Architecture in Practice, Second Edition*. Addison-Wesley 2003. ISBN 0321154959.

[Beck03] Kent Beck: *Test-driven Development: By Example*. Addison-Wesley 2003. ISBN 0321146530.

[Beck04] Kent Beck: *Extreme Programming Explained: Embrace Change*. Addison-Wesley 2004. ISBN 0321278658.

[Chrissis+06] Mary Beth Chrissis, Mike Konrad, and Sandy Shrum: *CMMI: Guidelines for Process Integration and Product Improvement, 2nd edition*. Addison-Wesley 2006. ISBN 0321279670.

[Cockburn00] Alistair Cockburn: *Writing Effective Use Cases*. Addison-Wesley 2000. ISBN 0201702258.

[Cockburn04] Alistair Cockburn: *Crystal Clear: A Human-Powered Methodology for Small Teams*. Addison-Wesley 2004. ISBN 0201699478.

[Coplien+05] James O. Coplien & Neil B. Harrison: *Organizational Patterns of Agile Software Development*. Pearson Prentice-Hall 2005. ISBN 0131467409.

[Fowler06]	Martin Fowler: *Continuous Integration*. Retrieved from www.martinfowler.com.
[Jacobson+99]	Ivar Jacobson, Grady Booch, and James Rumbaugh: *The Unified Software Development Process*. Addison-Wesley 1999. ISBN 0201571692.
[Meszaros04]	Gerard Meszaros: *Using Storyotypes to Split Bloated XP Stories*. Retrieved from www.clrstream.com.
[Poppendieck+03]	Mary and Tom Poppendieck: *Lean Software Development: An Agile Toolkit*. Addison-Wesley 2003. ISBN 0321150783.
[Schwaber+01]	Ken Schwaber and Mike Beedle: *Agile Software Development with SCRUM*. Prentice Hall 2001. ISBN 0130676349.
[xUnit]	www.junit.org

Design Patterns in the Context of Multi-modal Interaction

Andreas Ratzka
Institute for Media, Information and Cultural Studies
University of Regensburg
D-93040 Regensburg, Germany
Andreas.Ratzka@sprachlit.uni-regensburg.de

1 Abstract

Multi-modal interaction aims at more flexible, more robust, more performant and more natural interaction than can be achieved with traditional unimodal interactive systems. In order to achieve this, the developer needs some design support in order to select appropriate modalities, to find appropriate modality combinations and to implement promising modality adaptation strategies. This paper presents a first sketch of an emerging pattern language for multi-modal interaction and focuses on the sublanguage "fast input". This work is part of a thesis project on pattern based usability engineering for multi-modal interaction.

2 Introduction

Multi-modal interaction means interaction via several interaction-channels such as speech, pointing device, graphics and the like. According to Oviatt & Kuhn (1998) the goal of multi-modal interaction is

- to provide flexibility and adaptability of the system with respect to users and context of use,
- to provide higher robustness of interaction due to mutual disambiguation of input sources,
- to gain more interaction performance because of better integration into the work situation and
- to provide more natural interaction.

Current design support is usually restricted to very general rules, indicating the appropriateness of certain modalities. In this context one can allude modality theory (Bernsen 2001) and the modality properties derived from this theory (Bernsen 1999). Some other important issues are covered by the works of Bürgy (2002) and Calvary et al. (2005), which categorise aspects of task, environment, application data, user, device in order to provide appropriate advice towards modality usage.

These approaches provide only relatively general design advice but don't give more concrete suggestions. One assumption of this thesis project is that, although multi-modal interaction is a relatively new field with very little market penetration, there exists already a corpus of well founded research results and successful system implementations in which recurring patterns can be found.

This pattern collection is based on a thorough literature review on multi-modal interaction in industrial and research projects. Following questions helped to find an adequate categorisation of question-solution pairs and thus a basis for pattern mining:

- When to use a certain interaction modality?
- How to combine multiple interaction modalities?
- How to adapt modality usage according to the context of use (user, environment, situation)?

There is not one universal approach of classification, there are rather several categorisation alternatives. The categorisation I propose divides this pattern language into several, partially overlapping, sublanguages according to the goals of multi-modality in interaction design as described in Oviatt & Kuhn (1998, view previous section):

- Fast Multi-modal Input
- Multimodal Accessibility
- Robust Interaction
- Natural Task Support

This paper focuses on the sublanguage *Fast Multi-modal Input.*

2.1 Organisation of the Sublanguage "Fast Input"

The successful solution of the problems described in these patterns can be facilitated by the application of multi-modal interaction techniques which lead us to *abstract multi-modal interaction patterns*:

- Content-appropriate Input
- Context-appropriate Modality Usage
- Maximum Communication Bandwidth

At a more concrete level *concrete multi-modal interaction patterns* have been identified:

- Voice-based Interaction Shortcut
- Speech-enabled Form
- Speech-enabled Palette
- Gesture-enhanced Natural Speech
- Composed Gesture

2.2 Relationship to other Pattern Languages

As this work does not attempt to reinvent user interface design, but rather to enrich and combine traditional graphical and speech-based user interfaces (GUIs and SUIs), patterns from those domains are relevant as well such as the GUI-patterns in Tidwell (1999, 2005), van Welie & Trætteberg (2000) and Sinnig et al. (2004) and the SUI-patterns in Schnelle et al. (2005) and Schnelle & Lyardet (2006). Some of those patterns are modality independent such as *Favourites* and *Preferences* (van Welie & Trætteberg 2000). Others can, although described from the point of view of traditional interaction styles, extended to multi-modal interaction such as *Warning* (van Welie & Trætteberg 2000) or *Persona* (Schnelle & Lyardet 2006). Furthermore there exist close relationships between muldimodal interaction patterns and traditional interaction patterns. Following table illustrates some of the patterns referenced by this work:

Name, Reference	**Problem**	**Solution**
Helping Hands (van Welie 2001)	"Users need to enter many different types of [graphical] objects".	"Use one hand to enter the data while the other hand is used to switch modes", to select the appropriate tool from the palette.
Form (Tidwell 1999; Sinnig et al. 2004), (www.welie.com)	"The user must provide structural textual information to the application. The data to be provided is logically related"	"Provide users with a form containing the necessary elements. Forms contain basically a set of input interaction elements and are a means of collecting information [...]".
Auto-completion (Tidwell 2005)	"The user types something predictable, such as a URL, the user's own name or address, today's date, or a filename [...]".	"With each additional character that the user types, the software quietly forms a list of the possible completions to that partially entered string [...]".
Drop-down Chooser (Tidwell 2005)	"The user needs to supply input that is a choice from a set [...], a date or time, a number, or anything other than free text typed at a keyboard. [...]".	"For the Dropdown Chooser control's 'closed' state, show the current value of the control in either a button or a text field. To its right, put a down arrow. [...] A click on the arrow (or the whole control) brings up the chooser panel, and a second click closes it again [...]".
Composed Command (Tidwell 1999)	How can the artifact best present the actions that the user may take?	Provide a way for the user to directly enter the command, such as by speech or by typing it in.
Form Filling (Schnelle & Lyardet 2006)	"How to collect structured information from the user [in the context of speech-based applications]?"	"Identify a short description or label of the field to be filled in and prepare a variable to store the entered information. [...] Present the label to the user, followed with an optional input prompt and silence to let the user enter the data, as if she were filling out a form [...]".

3 Patterns for Fast Multi-modal Interaction

This paper focusses on the subcollection *fast input.* Along with the pattern descriptions, relations among these patterns are outlined, as well as relations between these patterns and other pattern collections, such as the GUI-centric patterns found in Tidwell (1999, 2005), van Welie & Trætteberg (2000) and Sinnig et al. (2004), and the voice user interface patterns described in Schnelle et al. (2005) and Schnelle & Lyardet (2006).

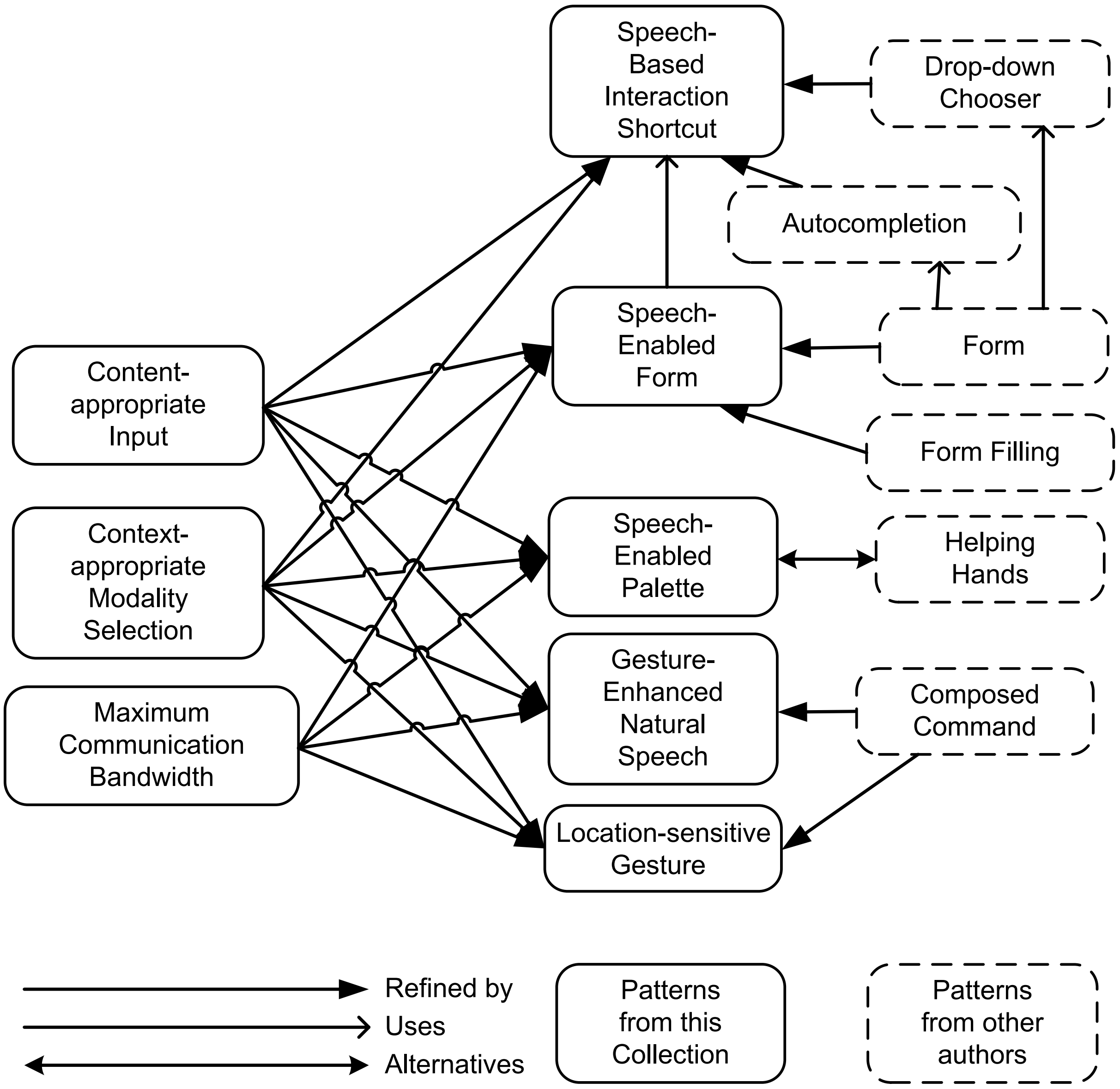

Speech-Based Interaction Shortcut
Drop-down Chooser
Autocompletion
Speech-Enabled Form
Form
Form Filling
Content-appropriate Input
Context-appropriate Modality Selection
Maximum Communication Bandwidth
Speech-Enabled Palette
Helping Hands
Gesture-Enhanced Natural Speech
Composed Command
Location-sensitive Gesture
Refined by
Uses
Alternatives
Patterns from this Collection
Patterns from other authors

Figure 1: Pattern Map

3.1 Abstract Multi-modal Interaction Patterns

3.1.1 Content-appropriate Input

Context Interactive systems allow the input and output of different kinds of data such as images, text, sounds, videos.

Problem There is no uniform way to input or present data of several specific formats and types.

Forces

- Information such as files, directories, locations on a map can be input via typing. But users memorise rather the location of those ones than their exact spelling.
- The selection of an item from a small (and thus completely visible) set is easily done via pointing and clicking, but the larger a set is growing additional navigation efforts are necessary such as scrolling or moving through hierarchical menus.
- Shortcut keys are a valid alternative for selection of menu items, interaction objects or tools on a palette. However, it is difficult to assign consistently intuitive shortcut-characters to a wide range of commands or menu options. Thus some shortcut-keys seem arbitrary and require a certain learning effort.
- Items can be selected via speech input from larger sets which reduces navigation efforts, but speech recognition errors might lead to additional error correcting steps which slows down interaction even more.
- Items that have well known and pronouncable names can be selected via speech input but cryptic filenames or e-mail addresses are likely to be misrecognised.

Solution Analyse tasks, workflow and the data to be exchanged between user and sytsem and choose interaction modalities accordingly.

Use a natural encoding of input and output data. Spacial data, proportions etc. can be most appropriately encoded spatially whereas precise conceptual data are to be encoded as spoken or written text.

"Natural encoding" should not be misunderstood too literally. In multimedia editing applications (video and sound editing) encodings have to be transformed: You need e.g. timeline visualisations in order to select the video or audio snippets you want to edit.

Consider furthermore confidental data. Private information should not be read out loudly, as it is nothing to bystanders or office mates.

Rationale Users prefer speech input to input descriptive data, or to select objects among large or invisible sets (Grasso & Finin 1997; Grasso et al. 1998; Oviatt et al. 2000). Pointing devices are preferred for inputting spacial or sketch-based data.

Modality Theory (Bernsen 2001, 1994a, b; Bernsen & Verjans 1995; Luz & Bernsen 2001) points out the plausibility of the complementary usage of analogous graphics and linguistic text in HCI.

Refining Patterns

- *Voice-based Interaction Shortcut* uses speech input to select items from a large set in order to reduce navigation overhead.
- *Speech-enabled Form* combines pointing and speech input. Pointing is used to select a field on an interaction form whereas speech input is used to input data into the selected field.
- *Speech-enabled Palette* combines speech input and pointing for palette based applications. Speech is used to select a tool on the palette whereas pointing is used to perform the graphic manipulation task. This way the mouse pointer can stay in the manipulation area and need not be moved repeatedly to the palette and back.
- *Gesture-enhanced Natural Speech* combines speech input with gestures or pointing actions. Speech input is used to specify the desired action (or query) along with some easily pronouncable parameters whereas pointing is used to select the interaction object (or query parameter), the action is to be performed on, as well as further parameters (such as the destination of a copying action).
- *Location-sensitive Gesture* combines iconic gestures with an implicit pointing action. Pointing is used to select an interaction object, the iconic gesture expresses an action that is to be performed on this object (such as a cross for deleting this object). Pointing is implicit in this case because the pointing location is deduced from onset and offset location of the drawing action.

Consequences

- Pointing does not require to memorise cryptic names.
- Selecting via speaking does not require all items to be displayed on screen.
- Selecting menu options via speech makes arbitrary key mappings superfluous.

3.1.2 Context-appropriate Modality Usage

Context There are different people using different kinds of interactive devices in differing environments and situations.

Problem Differing contexts of use lead to differing requirements which cannot be satisfied with one general uniform design.

Forces

- Typing is powerful for a lot of tasks but if the target user group includes typing-unskilled or even illiterate people other alternatives have to be used.
- In order to maximise mobility small devices have only tiny keyboards or only virtual on-screen keyboards. Using them for string input is annoying and slow.
- Speech input as promising text input and item selection alternative is likely to fail in loud environments. The same is true for speech output which is overheared in loud environments.
- Pointing is good for selecting objects but requires the selectable objects to be presented graphically. Small devices offer only little display space such that additional scrolling and navigation might be needed.
- Graphic output and feedback is useful in a lot of situations but cannot be perceived in bad lighting conditions or by blind people.
- Graphical output can be easily scanned by users but small devices have only little space for data to be displayed.
- Speech output and input can be more comfortable in mobile interaction but highly confident data must not be read out loudly in public environments.
- Environmental factors can be controlled via special installations such as specially mounted lamps, phone booths, directional speakers, earphones, or view shields but these measures are not viable in every case such as mobile interaction.

Solution Analyse task and workflow, target user group and interaction scenarios in order to determine appropriate interaction modalities. Offer more than one alternative interaction modality to the user so that he can choose the most appropriate one.

Modality theory (Bernsen 2001) and modality properties (Bernsen 1999) describe characteristics of different interaction modalities and are a basis for selecting adequate interaction styles.

Refining Patterns

- *Voice-based Interaction Shortcut* uses speech input to select items from a set which is too large to be displayed completely on screen. Small devices which offer only little space to display menu items or list-choosers encourage the usage of this pattern.

- *Speech-enabled Form* combines pointing for selecting a form field and speech input for filling in the field. Devices which lack a keyboard or situations where the user can use only one hand or a single finger for interacting with the system encourage the usage of this kind of multi-modal form.

- *Speech-enabled Palette* combines speech input and pointing for palette based applications. Speech is used to select a tool on the palette whereas pointing is used to perform the graphic manipulation task. Occasional users are more likely to remember a meaningful name of the palette-tool to be selected than an arbitrary shortcut key.

- *Gesture-enhanced Natural Speech* combines speech input with gestures or pointing actions for inputting composed commands. This allows the user to input composed action commands even with devices that do not provide a keyboard. Additional menus or buttons that would clutter small displays are not needed, either.

- *Location-sensitive Gesture* combines iconic gestures with an implicit pointing action for inputting composed commands. This allows the user to input complex action commands even with devices that do not provide a keyboard. Additional menus or buttons that would clutter small displays are not needed, either.

Consequences

- Typing-unskilled an illiterate people can interact comfortably via speaking or pointing.

- Speaking accelerates string input in mobile computing.

- In loud or dark environments the user can sidestep to alternative input modalities and switch output modalities.

- Design complexity is likely to increase as functionality has to be made accessible via several alternative interaction channels. Thorough testing of the system is necessary as with complexity sources of programming and design errors increase.

3.1.3 Maximum Communication Bandwidth

Context Tasks a user is performing with an interaction device are sometimes quite complex and require input and output of a lot of information, selecting tools (using a meta-tool called palette) and using additional interactive devices (such as car driver assistants).

Problem Additional devices – be they real or virtual – require the user to switch (visual) attention.

Forces

- Selecting of visible tools on a palette can be easily done via pointing, but in graphic manipulation tasks this leads to repeatedly moving of the mouse pointer between palette and canvas which is very time consuming.
- Typing text, commands or command keys is a powerful means of interaction. But in combination with pointing and graphic manipulation the user is slowed down as he needs to move his right hand from mouse to keyboard and back again.
- Reading text is preferred to hearing spoken text because users can determine pace themselves, but when graphical text is used in combination with graphical presentations or visualisation of complex processes the user has to switch his attention repeatedly between two locations. The same holds for situations where the user has to pay attention to the visual scene such as during driving.

Solution Analyse the task, data, target user groups and usage scenarios and determine appropriate interaction modalities as suggested by the patterns *Content-appropriate Modality Input* and *Context-appropriate Modality Usage.* Distribute parallel subtasks to different interaction modalities such as pointing and speaking, graphical visualisation and spoken text. This way attention switches can be avoided or minimised and interaction gets more efficient.

Rationale *Multiple Resource Theory* (Wickens 1980; Wickens et al. 1984; Wickens 1992) postulates that task interference gets minimized when different tasks are allocated to separated cognitive resources.

Baddeley (1986, 2003) postulates two subsystems of working memory, the *phonological loop* and the *visuo-spacial sketchpad.* These subsystems possess to some extent independent cognitive ressources.

Dual-task experiments conducted by Wickens et al. (1984) indicate that least task interferences occur, when the user performs the visual-spacial task using his hands and eyes and the verbal task with the vocal and auditory channel.

According to Srinivasan & Jovanis (1997), users of a car navigation systems performed better in driving when they were given spoken instructions than when they got visual instructions projected into the windscreen.

Studies conducted by Ren et al. (2000) have revealed that the combination of pointing devices such as pen or mouse with speech input is fruitful in both CAD systems and map-based interfaces. This way, interaction performance can be increased.

Cohen et al. (2000) compared standard direct-manipulation with the QuickSet pen/voice multimodal interface. Multi-modal interaction was significantly faster.

Mayer & Moreno (1998) have shown that learning can be more effective when different types of data are combined in a multi-modal way.

Elting et al. (2002) have shown that the combination of speech output and graphic pictures improve recall performance especially in the case of interacting with a PDA.

Refining Patterns for Variant I: Maximised Input Bandwidth

- *Speech-enabled Form*, *Speech-enhanced Action Space* and *Gesture-enhanced Natural Speech* combine all three pointing and speech input. No extra time is necessary to change hands between keyboard and mouse or to move the mouse cursor between different screen areas.
- *Gesture-enhanced Natural Speech* and *Location-sensitive Gesture* allow to input complex data efficiently without having to scroll around or open any extra menu.

Refining Patterns for Variant II: Maximised Perception Bandwidth

Following patterns are not subject of this paper but will be described in forthcoming ones.

- *Audio-visual Presentation* combines spoken text and graphical output. There is no need to jump with the eyes between reading area and visualisation and find again where you have stopped reading. Spoken text can be listened at the same time as the user is looking at the visualisation.
- *Workspace-integrated Information Display* enhances the user's working environment with audio messages and enriches when approppriate the visual field with augmented reality features using headup or even head-mounted displays.

Consequences

- Users have to change mouse and hand position less frequently.
- Overlapping and parallel input of speech and pointing gestures can speed up interaction.
- Parallel audio-visual output of related content can be perceived faster than written text and graphics.
- Even if speech or gesture recognition can accelerate interaction recognition errors might compromise this advantage. Clarification and error corroboration dialogs have to be designed with care. This is especially true when natural speech input is supported. The system should find a stable way of error corroboration without discouraging the user to make use of efficient interaction styles in future.
- System complexity is likely to increase as additional components (speech recognizers, gesture recognisers) have to be integrated into the program. These components increase storage and CPU requirements.

3.2 Concrete Patterns for Fast Multi-modal Input

3.2.1 Voice-based Interaction Shortcut

Context The user has to select items from a large set. Consider selecting an action from a menu or selecting a list item from a drop-down chooser.

Either the number of choices is quite large or screen size is scarse such that the items cannot be displayed all at once.

The interaction device is supporting speech input.

Problem Which interaction style allows the user to quickly select the desired item?

Forces

- Selection via pointing is very intuitive. But if the selection options are numerous and cannot be presented simultaneously on screen they have to be arranged into scrolling lists or hierarchical menu structures. In this case the advantage of intuitivity is compromised by lengthy scrolling or clicking through menus.
- Shortcut keys are a valid alternative for menu/command selection. However, it is difficult to assign consistently intuitive shortcut-characters to a wide range of commands or menu options. Thus some shortcut-keys seem arbitrary and require a certain learning effort.
- Typing in natural words is more intuitive and easier to learn than shortcut or function keys. However, typing is not always appropriate: Not all users are skilled typers. Typing with mobile devices is very awkward and slow and thus inappropriate for accelerating interaction.

Solution Selecting objects or actions via speaking them can significantly speed up interaction. This is especially true for frequent users to whom the command and item names are well known.

The designer should simply include the identical names of the menu options or the items of the drop down list (or combobox) into the speech recognition vocabulary in order to enable seamless learning of interaction shortcuts.

There is frequently more than one appropriate alternative wording for the desired action. The designer should check which synonyms should be included into the speech recognition grammar. User tests, including tests with simulated speech functionality[1] might be useful to elicit the most intuitive wordings for some system functionality.

Even when the designer has elicited the most intuitive command wordings, it is not guaranteed that especially first time and occasional users are able to anticipate them, too. The drop down list or menus should not be removed from the user interface.

Consequences

- Screen clutter and the need of menu navigation can be minimised.

[1]cf. Wizard of Oz tests: A human agent, the *wizard*, simulates speech functionality.

- There is no more need for the user to remember arbitrary action-key-mappings or to obey to strict menu hierarchies.
- Typing can be reduced to a minimum.
- If the selection set is large then speech recognition performance may deteriorate, especially when there are similarly sounding words. Even worse: some wordings might be ambiguous within the application context. If this cannot be avoided, the application has to provide clarification dialogs. In the worst case all speed advantages might be lost.

Rationale Users prefer speech input to input descriptive data, or to select objects among large or invisible sets (Grasso et al. 1998; Oviatt et al. 2000).

Known Uses NoteBook is a multi-modal notebook implemented on NeXT. The user can edit textual notes, and browse the created notes. Whereas the content-editing is only supported via typing, browsing, deleting and creating notes can be done via button clicks or voice commands alternatively (Nigay & Coutaz 1993).

Speech recognition packages such as ViaVoice can be integrated into the operating system in order to control standard applications.

MacOS provides built-in speech control.[2]

VoiceLauncher from Treoware enables speech input for Treos, Centro, and Tungsten—T3 devices.[3]

Microsoft Voice Command can be used to enable speech input for Windows Mobile based smartphones (such as HP's iPAQ 514[4]). Using this software extension, the user can show up the calendar or contact details in one interaction step.[5]

Related Patterns This pattern is a refinement of *Content-appropriate Input* and *Context-appropriate Modality Usage*.

This pattern can be used along with Tidwell's *Autocompletion* (Tidwell 2005) to enhance Tidwell's *Dropdown Chooser* (Tidwell 2005) with speech functionality.

As *Dropdown Choosers* are used in *Form*s (Tidwell 1999; Sinnig et al. 2004, cf. www.welie.com) *Voice-based Interaction Shortcuts* are used in *Speech-enabled Forms.*

[2]http://www.apple.com/macosx/features/speech/

[3]http://treoware.com/voicelauncher.html

[4]http://www.call-magazin.de/handy-mobilfunk/handy-mobilfunk-nachrichten/hps-erstes-smartphone-der-ipaq-514-hoert-aufs-wort_20628.html

[5]http://www.microsoft.com/windowsmobile/voicecommand/features.mspx

3.2.2 Speech-enabled Form

Context The user has to input structured data which can be mapped to some kind of *form* consisting of a set of atomic *fields*.

Devices such as PDAs do not provide a keyboard for comfortable string input.

In other situations the device may support keyboard input but the user has only one hand available for interacting with the system.

Problem How to simplify string input in form filling applications?

Forces

- Selecting areas in 2D-space is accomplished very comfortably with a pointing device but string input via pointing (with on-screen keyboards) is very awkward.
- Values for some form items (academic degree, nationality etc.) are restricted and can be input using drop down choosers (comboboxes). But this may lead to screen clutter and additional navigation and scrolling.
- Speech recognition is very comfortable for selecting invisible items but the input of unconstrained text suffers from recognition errors.

Solution Whereever possible determine acceptable values for each form field. Support value selection via *Dropdown Choosers* and, alternatively, via voice commands.

Let the user select from the desired form field via pointing and value input via speech. The speech input complexity can be reduced, as only the vocabulary of the selected form item needs to be activated at the time.

Consequences

- The user can comfortably combine pen input for selecting input fields with speech for value specification.
- Navigation and scrolling in drop down lists can be avoided.
- Constraining the voice recognition vocabulary according to the selected text field helps to avoid speech recognition errors.
- Speech recognition errors might occur anyway. In case of poor recognition performance all speed advantages might be lost due to the need of error corroborration.

Rationale Users prefer speech input to input descriptive data, or to select objects among large or invisible sets (Grasso & Finin 1997; Grasso et al. 1998; Oviatt et al. 2000).

Cohen et al. (2000) compared standard direct-manipulation with the QuickSet pen/voice multi-modal interface. Multi-modal interaction was significantly faster.

Known Uses Mobile Systems such as BBN's Portable Voice Assistant (Bers et al. 1998), Microsoft's MiPad (Miyazaki 2002; MiP) and IBM's Personal Speech Assistant (IBM; Comerford et al. 2001) are good examples.

In MiPad the user can create e-mail messages via *Tap And Talk*.[6] The user can select the addressee field and the speech recognition vocabulary is constrained to addressbook entries. If the user selects the subject or message field an unconstrained vocabulary is selected so that the user can input unconstrained text.

As a further example one could cite the QuickSet System (Cohen et al. 2000).

The multi-modal facilities offered by X+V (XHTML and VoiceXML) and supported by the Opera Browser are heavily focussed on this *Speech-enabled Form* paradigma (IBM 2002, 2003b, a, 2004).

Related Patterns This pattern is a refinement of *Content-appropriate Input*, *Context-appropriate Modality Usage* and *Maximum Communication Bandwidth*.

This pattern is a multi-modal extension of *Form* as found in Tidwell (1999) and Sinnig et al. (2004) and the speech-based *Form Filling* (Schnelle et al. 2005; Schnelle & Lyardet 2006).

It is implemented using the pattern *Voice-based Interaction Shortcut* in the same way as *Forms* are implemented using patterns such as *Dropdown Chooser* and *Autocompletion*.

For error handling consider to use *Multi-modal N-Best-Selection* and *Spelling-based Hypothesis Reduction* (to be described in forthcoming papers).

[6]http://research.microsoft.com/srg/mipad.aspx

3.2.3 Speech-enabled Palette

Alternative Name: Speech as Third Hand / Speech-enhanced Action Space

Context Direct manipulation allows the user to edit visually presented objects directly. In order to manipulate these objects the user has to select sometimes special tools. This means that the user has to leave the manipulation area with the mouse in order to select the desired menu item and then reenter the manipulation area in order to proceed the manipulation action. This might be very annoying, especially in drawing applications.

Problem How to enable the user to select tools from the palette without having to deplace the mouse between canvas and palette or the hand between mouse and keyboard?

Forces

- Both graphic manipulation tasks and selecting tools from a palette are accomplished very comfortably via pointing. But performing both subtasks alternately several times as is needed in design applications is very annoying and time-consuming.
- Using context menus which are opened on right clicks may reduce but not avoid totally pointing distance. Additionally the context menu (unless transparent) obscures the main manipulation area.
- Using the keyboard instead of the mouse for selecting commands may solve this problem in some cases. However, there might arise a new one: The user has to change his right hand between mouse and keyboard which is time-consuming, as well.
- Another solution would be to assign graphic manipulation tasks to the right hand which controls the mouse and action/menu selection tasks to the left hand which remains on the keyboard and inputs shortcut keys.[7] But the user would have to remember awkward key mappings and possibly to look down to the keyboard to find the desired key.

Solution Allow the user to select tools using speech input. Each tool on the palette should have a meaningful name which is being made obvious to the user (because it is displayed constantly or via tooltips) to allow seamless learning.

Consequences

- The user can speak the desired tool without the need to replace the mouse cursor between tool palette and manipulation area.
- The screen and especially the main manipulation area is not obscured by popup windows or menus.
- The right hand can stay on the mouse and need not be replaced between keyboard and mouse.

[7]This kind of solution is proposed in Welie's pattern *Helping Hands*.

- There is no need to remember awkard key mappings or to look down to the keyboard.
- As users are able to use the motor and vocal channels of their brain simultaneously, combining spoken commands and pointing speeds up interaction significantly.
- Speech recognition errors might occur. In case of poor recognition performance some speed advantages might be lost due to the need of error corroborration.

Rationale Studies conducted by Ren et al. (2000) have revealed that the combination of pointing devices such as pen or mouse with speech input is fruitful in both CAD systems and map-based interfaces. This way, interaction performance can be increased.

Cohen et al. (2000) compared standard direct-manipulation with the QuickSet pen/voice multi-modal interface. Multi-modal interaction was significantly faster.

Known Uses Graphic applications (Gorniak & Roy 2003; Hiyoshi & Shimazu 1994; Milota 2004), CAD-systems (Ren et al. 2000) and sketching applications (Sedivy & Johnson 2000) are examples which allow the user to select a tool of the palette via speech without removing the mouse cursor from the graphics manipulation area.

As further examples one could cite VoicePaint (Nigay & Coutaz 1993), MICASSEM (McCaffery et al. 1998) and QuickSet (Cohen et al. 2000).

Related Patterns This pattern is a refinement of *Content-appropriate Input*, *Context-appropriate Modality Usage* and *Maximum Communication Bandwidth.*

This pattern is an alternative of van Welie's *Helping Hands* (van Welie 2003).

3.2.4 Gesture-enhanced Natural Speech

Context Some applications require the input of composed commands consisting of several parameters.

Consider copying one object to another location which consists of inputting *the command*, *the object to be selected* and *the destination*.

Consider setting up an email message: Input *the command*, input *receivers of the message*.

Consider searching a location in a map-based application: Input *the command*, input *area constraints* (square C 5), input *keywords* (italian restaurants).

Problem How to enable the user to quickly input composed commands consisting of several parameters?

Forces

- Complex descriptive commands can be input efficiently via typed or spoken command languages. But consindering some parameters such as file names, directory locations, positions on a city map, users rather remember where these are than how these are named internally.
- An alternative would be offering the user to input complex commands via speech. But some parameters such as file names, directory locations, positions on a city map are frequently too cryptic and won't be pronounced in a predictable way.
- Inputting spatial parameters or selecting objects displayed on the screen is most easily done via pointing. But inputting actions or textual parameters would lead to one or more additional interaction steps (button clicks, navigation in menus, scrolling through drop-down lists).
- Early systems[8] combined pointing gestures with typed natural language input. This way the user could select objects directly via pointing and input commands and queries with the keyboard. However this way, the user has to change his hands regularly between keyboard and pointing devices which slows down interaction.

Solution Let the user interact via natural speech and provide pointing gestures simultaneously to specify locations or interactive objects. Consider folllowing cases:

- The user selects a file, says "copy this file there" and selects the target location.
- The user draws a rectangle onto a map and says "are there any supermarkets?"
- The user clicks the button *create mail* and says "to Margret Smith".

At first blush it seems to be simply an application of the pattern *Voice-based Interaction Shortcut*. But in the case of *Gesture-enhanced Natural Speech* the single interaction steps need not be done in a strict sequence but may overlap in time. This requires the

[8]cf. Shoptalk (Oviatt 2003; Cohen et al. 1989) and XTRA (Kobsa et al. 1986; Wahlster 1991)

application of multi-modal recognition technology and grammar formats for specifying multi-modal input such as MM-DCG (Shimazu et al. 1994). Consider approaches such as those described in Shimazu & Takashima (1996), Vo (1998), Bui & Rajman (2004) and Rajman et al. (2004) which include interaction corpus collection and training of recognition classifiers.

First time users need a way to explore the interface. That's why *Gesture-enhanced Natural Speech* should not be a replacement for alternative interaction styles such as direct manipulation but rather intergrated into them.

Consequences

- Commands and textual data can be input as text.
- Parameters such as file or directory names, locations on a map etc. can be input directly and naturally via pointing. There is no need to invent and remember cryptic names.
- Typing and recognition errors can be reduced as pointing replaces the input of cryptic strings.
- The user is able to utter simultaneously spoken language queries and pointing gestures: This way inputting spatial parameters and selecting objects can be done using pointing devices. At the same time, the input of textual data can be done via spoken language. This way the user can choose to use the most appropriate input modalities without having to change hands between different input devices.
- Screen clutter, the need of drop-down menus and popup dialogs which would obscure the potentially scarse screen space can be minimised when combining pointing with spoken natural language input. There is no need of additional buttons, dropdown menus, popup dialogs.
- Even if user input can be accelerated this way, recognition errors might compromise this advantage. Clarification and error corroboration dialogs have to be designed with care. This is especially true when natural speech input is supported. The system should find a stable way of error corroboration without discouraging the user to make use of efficient interaction styles in future.

Rationale Cohen et al. (1989), Cohen (1992) and Huls & Bos (1995) have shown the plausibility of combining direct manipulation and written natural language for some tasks.

Users prefer speech input to input descriptive data, or to select objects among large or invisible sets (Grasso & Finin 1997; Grasso et al. 1998; Oviatt et al. 2000). Pointing devices are preferred for inputting spacial or sketch-based data.

Known Uses This is one of the patterns found in the first multi-modal systems. Bolt (1980) describes a voice- and gesture-based interface which integrates pointing with natural language. The title of Bolt's article outlines this pictorially: "Put that there".

Siroux et al. (1998) describe the GEORAL map-based system which allows the user to input spoken multi-token natural language utterances while pointing to the relevant position on the map. The user can touch on a locality (by pointing on a single point or describing a zone) while asking questions such as *Are there any beaches in this locality?*, *Where are the campsites?* or *Show me the castles in this zone.*

Further examples are CUBRICON (Oviatt 2003), MVIEWS (Cheyer 1998), MATCH (Hastie et al. 2002; Johnston et al. 2002), SmartKom (Portele et al. 2003; Reithinger et al. 2003), ARCHIVUS (Lisowska et al. 2005), COMPASS2008 (Aslan et al. 2005), RASA (McGee & Cohen 2001) and DAVE_G (Rauschert et al. 2002).

Related Patterns This pattern is a refinement of *Content-appropriate Input*, *Context-appropriate Modality Usage* and *Maximum Communication Bandwidth.*

This pattern is related to *Composed Command* as found in Tidwell (1999). Furthermore it is related to *Location-sensitive Gesture.*

There are similarities to *Voice-based Interaction Shortcut* but Gesture-enhanced Natural Speech explicitly supports parallel processing of gesture and speech.

3.2.5 Location-sensitive Gesture

Context There are some frequently used action commands such as *delete* which require some additional parameters.

Some devices support pointing actions being performed with so-called *direct pointing devices* such as graphic tablets and pens.

There is no keyboard available or the user has only one hand free for interaction and this hand controls the pointing device.

Speech input is not supported or not appropriate due to context factors.

Problem How to enable the user to input easily and quickly commands consisting of selecting items and performing actions on them?

Forces

- Commands can be input efficiently textually or via speech. Command parameters such as the selected files can be mapped onto text, too. But when neither keyboard nor speech input is available annoying on-screen keyboards have to be used.
- Selecting areas or objects displayed on screen can be done easily via pointing. Inputting commands via pointing is possible, too. But this would lead to at least one additional interaction step (clicking on buttons, navigating through menus, scrolling etc.) and to cluttered screens.

Solution Let the user interact with the system as he would do with paper: draw meaningful symbols / pen gestures onto the object of interest – encircle items or cross them out, draw arrows and the like. Onset and offset pen positions can be interpreted as positional parameters.

Pen gestures have to be thoroughly planned. One aspect is that different gestures should be designed in a way that they are not too similar and too likely to be confused by the system. Design support might be provided by toolkits such as the one described in Long et al. (2001).

Consequences

- There is no more need to find an awkward textual representation for items displayed on screen. The user does not have to remember strange names to type them in or to try to pronounce them. He has simply to recognise the desired object displayed on screen.
- There is no need to require keyboard or speech input.
- Screen clutter, scrolling and menu navigation can be avoided.
- The user can input a complex action in one simple step: Drawing a meaningful gesture onto an object displayed on screen leads to opening, copying, deleting and the like. In usual graphic user interfaces the user would have to select the object first and then select an action which would require more steps.

- Gesture recognition is not always fully reliable because of the high variability of human gestures. Pen gestures can be misinterpreted: The system can miss user input or recognise some, where there is none.
- Slips of pen can be misinterpreted as gestures and lead to unwanted actions.
- Although gestures should be designed to be meaningful first time users will require some time of training before mastering the interfaces with all its advantages.

Rationale Gestures are easy to learn because they provide a means of natural interaction. Furthermore they provide a means of *terse and powerful interaction*, because both position and movement patterns can be exploited to convey information (Baudel & Beaudouin-Lafon 1993).

Known Uses The QuickSet System (Cohen et al. 1997, 2000) allows the user to place military units onto a map via drawing a military icon directly onto a map. The form of the icon designates the type of military base and the drawing position corresponds to the desired target location. Furthermore QuickSet supports specific editing gestures such as for *crossing out* (deleting) objects on the map.

Further examples can be found in TAPAGE (Cheyer & Julia 1998) and the systems described in di Fiore et al. (2004) and Ou et al. (2003).

Related Patterns This pattern is a refinement of *Content-appropriate Input*, *Context-appropriate Modality Usage* and *Maximum Communication Bandwidth.*

This pattern is related to *Composed Command* as found in Tidwell (1999) and *Gesture-enhanced Natural Speech.*

Variant An imaginable variant of this pattern uses handwriting instead of gestures, as with handwriting you can simultaneously input spatial and linguistic information, too. However, combining separated deictic gestures with subsequent (and not parallel) handwriting allows more precise input of spatial data. Such a combined input is supported by the multi-modal map-based application described in Cheyer & Julia (1998) and by the MATCH system (Hastie et al. 2002; Johnston et al. 2002).

4 Conclusion

This paper outlines an emerging pattern language for multimodal interaction which is far from being complete. Despite the research history of over twenty years, multimodality is still a research-centric field. It begins to reach some dissemination in the fields of automotive, industrial and mobile applications. That is why interaction design support is needed. Interaction design patterns constitute a challenging and exciting approach to this domain.

5 Acknoledgements

I want to thank my shepherd Allan Kelly and the writers workshop group in Bergen comprising Robert Hanmer (chair), Bettina Biel, Birthe Böhm, Met-Mari Nielsen, Matthias Rieger and Bart Van Rompaey for the helpful comments and suggestions.

References

IBM
IBM Wireless: Personal Speech Assistant. `http://www-1.ibm.com/industries/wireless/doc/content/resource/technical/296625104.html`. – IBM

MiP
MiPad: Speech Powered Prototype to Simplify Communication Between Users and Handheld Devices. `http://www.microsoft.com/presspass/features/2000/05-22mipad.asp`. – Microsoft

Aslan et al. 2005
Aslan, Ilhan; Xu, Feiyu; Uszkoreit, Hans; Krüger, Antonio; Steffen, Jörg: Crosslingual Interaction for Mobile Tourist Guide Applications. In: al., M. M. (Hrsg.): *INTETAIN 2005*. Berlin and Heidelberg: Springer, 2005 (LNAI 3814), S. 3–12

Baddeley 1986
Baddeley, Alan D.: *Working memory*. Oxford: Clarendon Pr., 1986 (Oxford psychology series ; 11.)

Baddeley 2003
Baddeley, Alan D.: *Human memory. Theory and practice*. Hove [u.a.]: Psychology Press, 2003

Baudel & Beaudouin-Lafon 1993
Baudel, Th.; Beaudouin-Lafon, M.: Charade: remote control of objects using free-hand gestures. In: *Commun. ACM* 36 (1993), Nr. 7, S. 28–35. `http://dx.doi.org/http://doi.acm.org/10.1145/159544.159562`. – DOI http://doi.acm.org/10.1145/159544.159562. – ISSN 0001–0782

Bernsen 1994a
Bernsen, Niels O.: Why are analogue graphics and natural language both needed in HCI. In: Paternò, F. (Hrsg.): *Design, Specification and Verification of Interactive Systems. Proceedings of the Eurographics Workshop*, 1994, S. 165–179

Bernsen 1994b
Bernsen, Niels O.: Why are analogue graphics and natural language both needed in HCI. In: Paternò, F. (Hrsg.): *Design, Specification and Verification of Interactive Systems. Proceedings of the Eurographics Workshop*, 1994, S. 165–179

Bernsen 1999
Bernsen, Niels O.: *Multimodality in Language and Speech Systems - from theory to design support tool.* Lectures at the 7th European Summer School on Language and Speech Communication (ESSLSC). `http://www.nis.sdu.dk/ nob/modalitytheory.html`. Version: July 1999

Bernsen 2001
Bernsen, Niels O.: Multimodality in language and speech systems – from theory to design support tool. In: Granström, B. (Hrsg.): *Multimodality in Language and Speech Systems.* Dordrecht: Kluwer, 2001

Bernsen & Verjans 1995
Bernsen, Niels O.; Verjans, Steven: From task domain to human-computer interface. An information mapping methodology. In: *Esprit Basic Research Project AMODEUS-2 Working Paper RP5-TM-WP17*, 1995

Bers et al. 1998
Bers, J.; Miller, S.; Makhoul, J.: Designing conversational interfaces with multimodal interaction. In: *DARPA Workshop on Broadcast News Understanding Systems* DARPA, 1998, S. 319–321

Bolt 1980
Bolt, Richard A.: „Put-that-there“: Voice and gesture at the graphics interface. In: *SIGGRAPH '80: Proceedings of the 7th annual conference on Computer graphics and interactive techniques.* New York, NY, USA: ACM Press, 1980. – ISBN 0–89791–021–4, S. 262–270

Bui & Rajman 2004
Bui, T.; Rajman, M.: *Rapid Dialogue Prototyping Methodology.* `citeseer.ist.psu.edu/bui04rapid.html`. Version: 2004

Bürgy 2002
Bürgy, Christian: *An Interaction Constraints Model for Mobile and Wearable Computer-Aided Engineering Systems in Industrial Applications*, Department of Civil and Environmental Engineering, Carnegie Mellon University, Pittsburgh, Pennsylvania, USA, Diss., 2002

Calvary et al. 2005
Calvary, G.; Coutaz, J.; Dáassi, O.; Balme, L.; Demeure, A.: Towards a New Generation of Widgets for Supporting Software Plasticity: The „Comet“. In: Bastide, R. (Hrsg.); Palanque, P. (Hrsg.); Roth, J. (Hrsg.): *EHCI-DSVIS 2004, LNCS 3425*, Springer, 2005, S. 306–324

Cheyer 1998
Cheyer, A.: MVIEWS: Multimodal tools for the video analyst. In: *International*

Conference on Intelligent User Interfaces (IUI'98). New York: ACM Press, 1998, S. 55–62

Cheyer & Julia 1998
CHEYER, Adam; JULIA, Luc: Multimodal Maps: An Agent-Based Approach. In: *Multimodal Human-Computer Communication, Systems, Techniques, and Experiments.* London, UK: Springer-Verlag, 1998. – ISBN 3–540–64380–X, S. 111–121

Cohen et al. 1997
COHEN, P. R.; JOHNSTON, M.; MCGEE, D.; OVIATT, Sh.; PITTMAN, J.; SMITH, I.; CHEN, L.; CLOW, J.: QuickSet: multimodal interaction for distributed applications. In: *MULTIMEDIA '97: Proceedings of the fifth ACM international conference on Multimedia.* New York, NY, USA: ACM Press, 1997. – ISBN 0–89791–991–2, S. 31–40

Cohen et al. 2000
COHEN, Ph.; MCGEE, D.; CLOW, J.: The efficiency of multimodal interaction for a map-based task. In: *Proceedings of the sixth conference on Applied natural language processing.* San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2000, S. 331–338

Cohen 1992
COHEN, Philip R.: The role of natural language in a multimodal interface. In: *UIST '92: Proceedings of the 5th annual ACM symposium on User interface software and technology.* New York, NY, USA: ACM Press, 1992. – ISBN 0–89791–549–6, S. 143–149

Cohen et al. 1989
COHEN, Philip R.; DALRYMPLE, Mary; MORAN, Douglas B.; FERNANDO; PEREIA, C. N.; SULLIVAN, Joseph W.; JR, Robert A. G.; SCHLOSSBERG, Jon L.; TYLER, Sherman W.: Synergistic use of direct manipulation and natural language. In: *Human Factors in Computing Systems, CHI '89*, ACM Press, 1989, S. 227–233

Comerford et al. 2001
COMERFORD, L.; FRANK, D.; GOPALAKRISHNAN, P.; GOPINATH, R.; SEDIVY, J.: The IBM Personal Speech Assistant. In: *Proc. of IEEE ICASSP'01* DARPA, 2001, S. 319–321

Elting et al. 2002
ELTING, Christian; ZWICKEL, Jan; MALAKA, Rainer: Device-dependant modality selection for user-interfaces: an empirical study. In: *IUI '02: Proceedings of the 7th international conference on Intelligent user interfaces.* New York, NY, USA: ACM Press, 2002. – ISBN 1–58113–459–2, S. 55–62

di Fiore et al. 2004
FIORE, Fabian di; VANDOREN, Peter; REETH, Frank van: Multimodal Interaction in a Collaborative Virtual Brainstorming Environment. In: LUO, Y. (Hrsg.): *1st International Conference on Cooperative Design, Visualization & Engineering (CDVE 2004).* Berlin, Heidelberg: Springer, September 2004, S. 47–60

Gorniak & Roy 2003
GORNIAK, Peter; ROY, Deb: Augmenting user interfaces with adaptive speech commands. In: *ICMI '03: Proceedings of the 5th international conference on Multimodal interfaces.* New York, NY, USA: ACM Press, 2003. – ISBN 1–58113–621–8, S. 176–179

Grasso & Finin 1997
Grasso, M.A.; Finin, T.: Task integration in multimodal speech recognition environments. In: *Crossroads* 3 (1997), Nr. 3, S. 19–22

Grasso et al. 1998
Grasso, Michael A.; Ebert, David S.; Finin, Timothy W.: The integrality of speech in multimodal interfaces. In: *ACM Trans. Comput.-Hum. Interact.* 5 (1998), Nr. 4, S. 303–325. `http://dx.doi.org/http://doi.acm.org/10.1145/300520.300521`. – DOI http://doi.acm.org/10.1145/300520.300521. – ISSN 1073–0516

Hastie et al. 2002
Hastie, H. W.; Johnston, M.; Ehlen, P.: CONTEXT-SENSITIVE HELP FOR MULTIMODAL DIALOGUE. In: *ICMI '02: Proceedings of the 4th IEEE International Conference on Multimodal Interfaces.* Washington, DC, USA: IEEE Computer Society, 2002. – ISBN 0–7695–1834–6

Hiyoshi & Shimazu 1994
Hiyoshi, Mayumi; Shimazu, Hideo: Drawing pictures with natural language and direct manipulation. In: *Proceedings of the 15th conference on Computational linguistics.* Morristown, NJ, USA: Association for Computational Linguistics, 1994, S. 722–726

Huls & Bos 1995
Huls, C.; Bos, E.: Sutdies into full integration of language and action. In: *Proceedings of the International Conference on Cooperative Multimiodal Communication (CMC/95)*, 1995, S. 161–174

IBM 2002
IBM (Hrsg.): *Developing X+V Applications Using the Multimodal Toolkit and Browser.* IBM, October 2002. – IBM

IBM 2003a
IBM (Hrsg.): *Multimodal Application Design Issues.* IBM, December 2003. – IBM

IBM 2004
IBM (Hrsg.): *XHTML+Voice Programmer's Guide.* Version 1.0. IBM, February 2004. – IBM

IBM 2003b
IBM Pervasive Computing (Hrsg.): *Developing Multimodal Applications using XHTML+Voice.* IBM Pervasive Computing, January 2003. – IBM

Johnston et al. 2002
Johnston, M.; Bangalore, S.; Vasireddy, G.; Stent, A.; Ehlen, P.; Walker, M.; Whittaker, S.; Maloor, P.: MATCH: An Architecture for Multimodal Dialogue Systems. In: *Proceedings of the 40th Annual Meeting of the Association for Computation Linguistics (ACL)* Association for Computational Linguistics (ACL, 2002, S. 376–383

Kobsa et al. 1986
Kobsa, A.; Allgayer, J.; Reddig, C.; Reithinger, N.; Schmauks, D.; Harbusch, K.; Wahlster, W.: Combining Deictic Gestures and Natural Language for Referent

Identification. In: *Proc. 11th International Conf. On Computational Linguistics.* Bonn, Germany, 1986, S. 356–361

Lisowska et al. 2005
Lisowska, Agnes; Rajman, Martin; Bui, Trung H.: ARCHIVUS: A System for Accessing the Content of Recorded Multimodal Meetings. In: Bengio, S. (Hrsg.); Bourland, H. (Hrsg.): *MLMI2004*, 2005 (LNCS 3361), S. 291–304

Long et al. 2001
Long, A. C.; Landay, James A.; Rowe, Lawrence A.: "Those look similar!" issues in automating gesture design advice. In: *PUI '01: Proceedings of the 2001 workshop on Perceptive user interfaces.* New York, NY, USA: ACM Press, 2001, S. 1–5

Luz & Bernsen 2001
Luz, S.; Bernsen, N. O.: A tool for interactive advice on the use of speech in multimodal systems. In: *Journal of VLSI Signal Processing* 29 (2001), S. 129–137

Mayer & Moreno 1998
Mayer, R. E.; Moreno, R.: A Split-Attention Effect in Multimedia Learning: Evidence for Dual Processing Systems in Working Memory. In: *Journal of Educational Psychology* 90 (1998), Nr. 2, S. 312–320

McCaffery et al. 1998
McCaffery, Fergal; McTear, Michael F.; Murphy, Maureen: A Multimedia Interface for Circuit Board Assembly. In: *Multimodal Human-Computer Communication, Systems, Techniques, and Experiments.* London, UK: Springer-Verlag, 1998. – ISBN 3–540–64380–X, S. 213–230

McGee & Cohen 2001
McGee, David R.; Cohen, Philip R.: Creating tangible interfaces by augmenting physical objects with multimodal language. In: *IUI '01: Proceedings of the 6th international conference on Intelligent user interfaces.* New York, NY, USA: ACM Press, 2001. – ISBN 1–58113–325–1, S. 113–119

Milota 2004
Milota, André D.: Modality fusion for graphic design applications. In: *ICMI '04: Proceedings of the 6th international conference on Multimodal interfaces.* New York, NY, USA: ACM Press, 2004. – ISBN 1–58113–995–0, S. 167–174

Miyazaki 2002
Miyazaki, J.: *Discussion Board System with modality variation: From Multimodality to User Freedom*, Tampere University, Diplomarbeit, 2002

Nigay & Coutaz 1993
Nigay, Laurence; Coutaz, Joëlle: A design space for multimodal systems: concurrent processing and data fusion. In: *Human Factors in Computing Systems, INTERCHI '93 Conference Proceedings*, ACM Press, 1993, S. 172–178

Ou et al. 2003
Ou, Jiazhi; Fussell, Susan R.; Chen, Xilin; Setlock, Leslie D.; Yang, Jie: Gestural

communication over video stream: supporting multimodal interaction for remote collaborative physical tasks. In: *ICMI '03: Proceedings of the 5th international conference on Multimodal interfaces.* New York, NY, USA: ACM Press, 2003. – ISBN 1–58113–621–8, S. 242–249

Oviatt 2003
Oviatt, Sharon: Multimodal interfaces. In: Jacko, J. (Hrsg.); Sears, A. (Hrsg.): *The Human-Computer Interaction Handbook: Fundamentals, Evolving Technologies and Emerging Applications.* Mahwah, NJ: Lawrence Erlbaum Assoc., 2003, S. 286–304

Oviatt et al. 2000
Oviatt, Sharon; Cohen, Phil; Wu, Lizhong; Vergo, John; Duncan, Lisbeth; Suhm, Bernahrd; Bers, Josh; Holzman, Thomas; Winograd, Terry; Landay, James; Larson, Jim; Ferro, David: Designing the User Interface for Multimodal Speech and Pen-based Gesture Applications: State-of-the-Art Systems and Future Research Directions. In: *Human Computer Interaction* 15 (2000), Nr. 4, S. 263–322

Oviatt & Kuhn 1998
Oviatt, Sharon; Kuhn, Karen: Referential features and linguistic indirection in multimodal language. In: *Proceedings of the International Conference on Spoken Language Processing* Bd. 6, ASSTA, 1998, S. 2339–2342

Portele et al. 2003
Portele, Thomas; Goronzy, Silke; Emele, Martin; Kellner, Andreas; Torge, Sunna; Vrugt, Jürgen te: SmartKom-Home – An Advanced Multi-Modal Interface to Home Entertainment. In: *EUROSPEECH-2003*, 2003, S. 1897–1900

Rajman et al. 2004
Rajman, M.; Bui, T.H.; Rajman, A.; Seydoux, F.; Quarteroni, S.: Assessing the usability of a dialogue management system designed in the framework of a rapid dialogue prototyping methodology. In: *Acta Acustica united with Acustica 2004*, 2004

Rauschert et al. 2002
Rauschert, I.; Agrawal, P.; Sharma, R.; Fuhrmann, S.; Brewer, I.; MacEachren, A.: Designing a human-centered, multimodal GIS interface to support emergency management. In: *GIS '02: Proceedings of the 10th ACM international symposium on Advances in geographic information systems.* New York, NY, USA: ACM Press, 2002. – ISBN 1–58113–591–2, S. 119–124

Reithinger et al. 2003
Reithinger, N.; Alexandersson, J.; Becker, T.; Blocher, A.; Engel, R.; Löckelt, M.; Müller, J.; Pfleger, N.; Poller, P.; Streit, M.; Tschernomas, V.: SmartKom: adaptive and flexible multimodal access to multiple applications. In: *ICMI '03: Proceedings of the 5th international conference on Multimodal interfaces.* New York, NY, USA: ACM Press, 2003. – ISBN 1–58113–621–8, S. 101–108

Ren et al. 2000
Ren, Xiangshi; Zhang, Gao; Dai, Guozhong: An Experimental Study of Input Modes for Multimodal Human-Computer Interaction. In: *ICMI '00: Proceedings of the Third International Conference on Advances in Multimodal Interfaces.* London, UK: Springer-Verlag, 2000. – ISBN 3–540–41180–1, S. 49–56

Schnelle & Lyardet 2006
Schnelle, Dirk; Lyardet, Fernando: Voice User Interface Design Patterns. In: *Proceedings of 11th European Conference on Pattern Languages of Programs (EuroPlop 2006)*, 2006. – to appear

Schnelle et al. 2005
Schnelle, Dirk; Lyardet, Fernando; Wei, Tao: Audio Navigation Patterns. In: *Proceedings of EuroPLoP 2005*, 2005, S. 237–260

Sedivy & Johnson 2000
Sedivy, J.; Johnson, H.: Multimodal tool support for creative tasks in the visual arts. In: *Knowledge-Based Systems* 13 (2000), December, Nr. 7–8, S. 441–450

Shimazu et al. 1994
Shimazu, Hideo; Arita, Seigo; Takashima, Yosuke: Multi-Modal Definite Clause Grammar. In: *COLING 1994*, 1994, S. 832–836

Shimazu & Takashima 1996
Shimazu, Hideo; Takashima, Yosuke: Multi-Modal-Method: A Design Method for Building Multi-Modal Systems. In: *COLING 1996*, 1996, S. 925–930

Sinnig et al. 2004
Sinnig, Daniel; Gaffar, Ashraf; Reichart, Daniel; Seffah, Ahmed; Forbrig, Peter: Patterns in Model-Based Engineering. In: *CADUI*, 2004, S. 195–208

Siroux et al. 1998
Siroux, J.; Guyomard, M.; Multon, F.; Remondeau, Ch.: Modeling and Processing of Oral and Tactile Activities in the GEORAL System. In: *Multimodal Human-Computer Communication, Systems, Techniques, and Experiments*. London, UK: Springer-Verlag, 1998. – ISBN 3–540–64380–X, S. 101–110

Srinivasan & Jovanis 1997
Srinivasan, R.; Jovanis, P. P.: Effect of In-Vehicle Route Guidance Systems on Driver Workload and Choice of Vehicle Speed: Findings from a Driving Simulator Experement. In: Noy, I. Y. (Hrsg.): *Ergonomics and Safety of Intelligent Driver Interfaces*. Mahwah, NJ: Lawrence Erlbaum, 1997, S. 97–114

Tidwell 1999
Tidwell, J.: COMMON GROUND: A Pattern Language for Human-Computer Interface Design, 1999

Tidwell 2005
Tidwell, Jenifer: *Designing Interfaces: Patterns for Effective Interaction Design*. O'Reilly, 2005

Vo 1998
Vo, Minh T.: *A Framework and Toolkit for the Construction of Multimodal Learning Interfaces*, School of Computer Science, Computer Science Department, Carnegie Mellon University, Diss., 1998

Wahlster 1991
Wahlster, Wolfgang: User and discourse models for multimodal communication. In: Sullivan, J. W. (Hrsg.); Tyler, S. W. (Hrsg.): *Intelligent User Interfaces*, ACM Press, 1991, S. 45–67

van Welie & Trætteberg 2000
Welie, M. van; Trætteberg, H.: Interaction Patterns in User Interfaces. In: *Proceedings of the Seventh Pattern Languages of Programs Conference*, 2000

van Welie 2001
Welie, Martijn van: *Task-based User Interface Design*, Dutch Graduate School for Information and Knowledge Systems, Vrije Universiteit Amsterdam, Diss., 2001

van Welie 2003
Welie, Martijn van: GUI Design Patterns. In: *www.welie.com – ...patterns in Interaction Design*, 2003

Wickens 1992
Wickens, C. C.: *Engineering Psychology and Human Performance.* New York: Harper Collins, 1992

Wickens et al. 1984
Wickens, C. D.; Vidulich, M.; Sandry-Garza, D.: Principles of S-R-C compatibility with spatial and verbal tasks: The role of display-control interfacing. In: *Human Factors* 26 (1984), S. 533–534

Wickens 1980
Chapter: The Structure of Attentional Resources. In: Wickens, C.C.: *Attention and Performance VIII.* Hillsdale, NJ: Lawrence Erlbaum, 1980, S. 239–257

More Patterns for Software Companies (VikingPLoP 2007)

Allan Kelly - allan@allankelly.net

Abstract

These patterns extend the author's work on how software companies operate. Together with earlier patterns the three patterns presented here look at how combinations of products and services are marketed, how customers are managed and how services are designed and delivered.

The patterns are presented here are: ACCOUNT MANAGEMENT, SALES/TECHNICAL DOUBLE ACT and PACKAGED SERVICES.

1 Introduction

Many patterns have been written concerned with the design and architecture of software systems, e.g. (Gamma et al. 1995; Manolescu et al. 2006; Schmidt et al. 2000) to name a few. Other patterns have been written describing the organizational development of software organizations, e.g. (Bricout et al. 2004; Coplien and Harrison 2004; Marquardt 2004) among many. The patterns presented here are concerned with business strategy and operations of software companies.

Organizational structure will constrain the strategies available to an organization and conversely the strategies a company pursues often dictate organizational structure. For example staffing levels will be effected by the use of DOMAIN EXPERTISE IN ROLES (Coplien and Harrison 2004). Similarly, a strategy of utilising offshore development may bring JOIN FOR COMPLETION (Bricout et al. 2004) into use.

Through such mechanisms the patterns used at one level in the organization constrain the options available at another level. As Conway (1968) suggested, the organizational structure will influence the system structure. However, it is also true that the system structure can effect the organizational structure (Hvatum and Kelly 2005). As Figure 1 shows, we can think of each level partially constraining the others.

In contrast with strategy the tactics and implementation detail are often regarded as less important. So it is that some patterns may appear to be relatively unimportant. However, when viewed from a different perspective these details can take on significant, and even strategic important. There are no firm boundaries between what is tactical and what is strategic, details considered tactical today may be strategic in future (Mintzberg 1994). Therefore one should not apply the labels strategic, tactical or implementation too quickly.

Copyright (c) 2007 by Allan Kelly

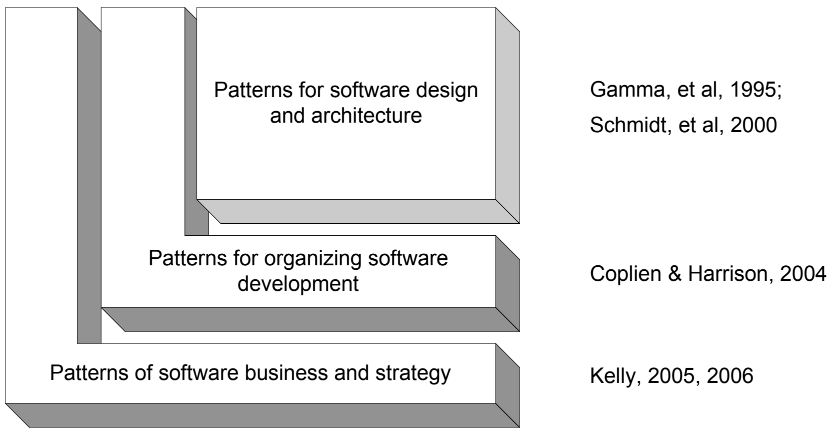

Patterns for software design and architecture
Gamma, et al, 1995;
Schmidt, et al, 2000
Patterns for organizing software development
Coplien & Harrison, 2004
Patterns of software business and strategy
Kelly, 2005, 2006

Figure 1 - Patterns at one level are partially constrained by patterns at other levels

The author's earlier work (2005a; Kelly 2005b; 2006b) set out a framework for applying pattern thinking in the business domain. The patterns in this paper add to a series of patterns concerned with the interplay of product and service offerings from an organization. Many software companies struggle to effectively deliver services alongside products. Collectively these patterns explore why companies do this and how they can do it effectively. Thumbnails of earlier patterns in this series are given below.

The term *software company* is used broadly to refer to any commercial organization that is reliant on sales of software based products to generate revenue. This includes sellers of packaged software (e.g. Adobe), sellers of custom software solutions (e.g. Accenture) and sellers of online software as a service product (e.g. SalesForce.com). The definition does not include companies that develop their own software for internal use (e.g. CitiGroup). Although such organizations may learn from these patterns, they are not the primary focus of this work.

2 Audience

These patterns are intended to codify several common business practices in a pattern language so that they may be better understood, communicated and studied. The patterns given here are intended for those interested in how corporate strategies may be applied. This group includes existing managers, future managers and entrepreneurs.

In particular it is hoped that those who are on the receiving end of such strategies and tactics will find these patterns informative and useful. Too often companies fail to explain strategies and tactics to those whose work is affected. For example, in the case of software companies it may be far from obvious that an ACCOUNT MANAGEMENT pattern is being applied. Understanding what a company is attempting, why and the implications can be beneficial to all.

The patterns in this paper, and others in the series may be read and applied outside the domain of software companies. They may be applied to technology companies in general and to non-technology companies in some instances. The author has chosen to confine the domain and context of these patterns to software companies for two reasons. Firstly this is the domain the author knows and has experience in. Secondly, limiting the domain helps maintain the brevity of the patterns. Despite these deliberate limitations the author believes many of these patterns may be applied in contexts outside the software domain.

Many of the examples are drawn from outside the software domain. These examples have been chosen primarily because they clearly illustrate the pattern in question. Such examples also demonstrate the wider applicability of these patterns.

3 Patterns and Sequences

The patterns presented in this paper form part of a growing *pattern language*. As additional patterns are added more are identified. Patterns within this language are assembled together in *sequences*. It is natural to find the application of one pattern creates the need, or opportunity, to apply another pattern. There is no mandated, or even *right*, sequence through the language; each organization needs to find the sequence(s) that works for it.

Patterns, by their nature, capture existing knowledge rather than create new knowledge. In some cases this knowledge may not have been captured before, although *known* to some individuals the knowledge may only have existed tacitly inside the heads of individuals. Alternatively the knowledge may be embedded in working practices, processes or market mechanisms.

These Patterns draw on experience and existing literature. Much of this knowledge only exists as heuristics, or tacit knowledge, known only to individuals and management groups**.** Presenting this knowledge in pattern form allows the knowledge to be communicated and combined with other knowledge. Once captured these heuristics can be examined, enhanced, refined or even deprecated.

By documenting this knowledge in literature it can be made more accessible to a wider audience. These patterns should make this knowledge accessible to the managers, engineers and others who need it and are tasked with implementing the strategies.

Most of the patterns presented here have been identified by the author from his own experience and investigation. During the pattern review process, (shepherding and conference workshop review) additional patterns have been identified by reviewers. Patterns are by their nature generative, as more are identified and documented more become apparent; and as patterns are applied the need for others is revealed.

When patterns are applied together they are said to form a pattern sequence. There may be many ways of combining the patterns in a pattern language, and each pattern may appear in multiple sequences. Pattern sequences show the order patterns are combined in order to make a whole.

It is not always obvious from a pattern description which patterns should be applied together. Even if the pattern writer could specify this information they may choose not too, either for the benefit of brevity, or to leave the reader with options. Pattern sequences are used to describe which patterns are applied in tandem and to describe the effect on the whole when several patterns are combined.

Naturally there are many ways in which patterns may be applied. Some patterns are larger than others, they describe a large thing to build. The building of this thing requires the use of smaller patterns. These in turn may require multiple patterns to build. For example, in *A Pattern Language* Alexander starts with patterns for distributing towns and cities in a region. He moves on to describe the organization of the town and from there to the individual buildings.

It is not essential to apply every pattern in a language or a sequence. We choose which patterns to apply and which not too. Few, if any, patterns are without negative consequences along with the positive ones. In some cases we may decide that despite the positive attributes we will not apply a pattern. There is nothing automatic in the application of patterns; the decision to use, or not to use, a pattern is purely a human one. Consequently the application of a single pattern language may result in different systems being created.

(As an aside, it is worth noting that this implies that mechanical automatons cannot apply a pattern language to create a whole system without human intervention. A reoccurring themes in software engineering pattern literature are the automatic discover and application of patterns. An understanding of pattern languages and sequences so why this is not possible.)

When applying a pattern language we will be faced with choices. Not only must we choose whether to apply a pattern or not but on occasions we will have to choose between different patterns. For example, faced with limited space for a house we may be forced to choose between WORKSPACE ENCLOSURE and DRESSING ROOMS (Alexander 1977). Pattern writers cannot foresee every context, problem or force that may lead to modifications when applying a pattern. Human judgement is needed to select and adjust individual patterns and sequences.

On other occasions we may find that the application of one pattern forces us to use another. The negative consequences of applying one pattern will create forces, resolving these forces may require the use of another pattern.

So it is that patterns from a common language are applied in sequence. Such a sequence forms a path through a pattern language, the result is a single whole (Coplien and Harrison 2004).

A sequence may be a well known one or it may be one we have devised ourselves. Individual patterns may play a role in multiple sequences; indeed the outcome of applying one pattern in two different sequences may be different. Even when patterns come from the same language not all possible sequences will be useful or even make sense. Patterns taken from different languages might work together, or they might not.

The sequences contained in this paper, indeed in any pattern paper containing sequences, are merely suggestions and record what the author has seen work. All but the smallest problems are likely to differ in some element from previous problems. We should not expect to be able to apply a previous pattern sequence exactly. Readers are encouraged to make up their own sequences.

When writing patterns it is natural to find one pattern leading to another. In writing this collection of business patterns the discovery and documentation of one pattern has more often then not led to the discovery of another pattern. Thus as patterns are describe sequences are mapped out.

The application and creation of patterns is an exercise in stepwise-refinement. The details of a large pattern are often implemented with a set of smaller patterns. These in turn may require the use of several smaller patterns, and so on.

In part the patterns one discovers depends on the granularity of patterns, one lengthy pattern may cover many scenarios. Alternatively, one short pattern may require several more patterns to cover the same scenarios. It is the writer's decision to decide which course best explains the problems and solutions to the reader.

Figure 2 shows how the patterns in this paper connect with the other patterns in this series. In this sequence we envisage a start-up company that uses SERVICES BEFORE PRODUCTS in order to bootstrap itself into business. Once in established the company uses START-UP SERVICES FOR PRODUCTS to help new customers use their products. Over time the company continues to support customers by using CONTINUING SERVICES FOR PRODUCTS.

At this point the established company faces a number of opportunities, some of which are complementary and others mutually exclusive. The company may decide to change the nature of its products business, it might decide that the supply of services are a more lucrative endeavour and adopt SERVICES TRUMP PRODUCTS and COMPLEMENTOR, NOT COMPETITOR. Whether moving to a service only model or continuing to supply products the company may also adopt a PACKAGED SERVICES model to simplify the sale and delivery of service offerings.

When services are the main offering from the company the role is constant. But when services are offered to supplement a product their role their role changes over time. Recognising this change will help organize and structure the services provided.

Whether pursing the services route or not, the company may decide to leverage its existing customer base by using SAME CUSTOMER, DIFFERENT PRODUCT. In order to implement this pattern ACCOUNT MANAGEMENT can be used. However since active customer management is a time consuming business the company may also adopt SALES/TECHNICAL DOUBLE ACT to spread the work.

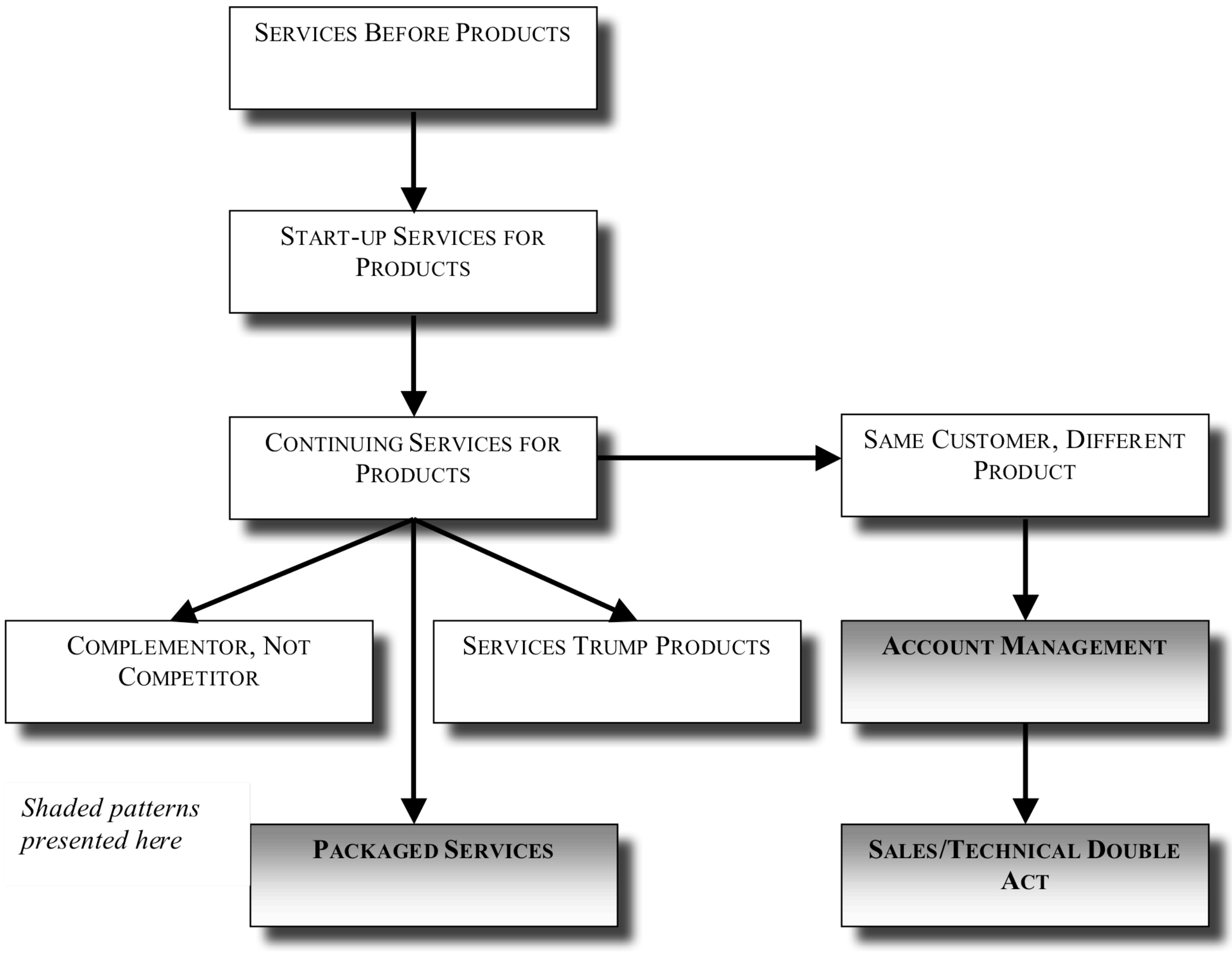

SERVICES BEFORE PRODUCTS
START-UP SERVICES FOR PRODUCTS
CONTINUING SERVICES FOR PRODUCTS
SAME CUSTOMER, DIFFERENT PRODUCT
COMPLEMENTOR, NOT COMPETITOR
SERVICES TRUMP PRODUCTS
ACCOUNT MANAGEMENT
Shaded patterns presented here
PACKAGED SERVICES
SALES/TECHNICAL DOUBLE ACT

Figure 2 - Map of the Products & Services Pattern Language and possible sequences

4 The Patterns

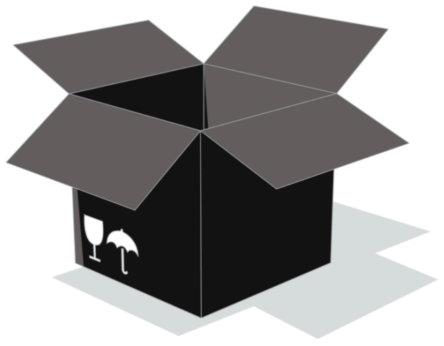
4.1 Pattern thumbnails

PACKAGED SERVICES Page 9	Services can complement a product offering and provide a good revenue stream. However, they can also be expensive to operate. Treating services more like products can make them easier to sell and help keep costs down. Therefore package them as products with defined cost and outcome.
ACCOUNT MANAGEMENT Page 13	Existing customers are an asset to your business; it is more cost effective to sell more to existing customers than find and sell to new customers. There is value in the relationship itself. Therefore, actively manage the relationship. You can improve customer retention and provide opportunities for new sales and co-operation.
SALES/TECHNICAL DOUBLE ACT Page 17	Managing the commercial and technical aspects of a customer relationship can be a big job and requires different skills. Therefore use two people, one with a technical focus and one with a commercial focus to manage the different aspects of the customer relationship.

PRODUCTS AND SERVICES (Kelly 2006a)	Technically complicated products are not commodities; they can be hard to use. Therefore, offer services to help the customers in addition to the product, e.g. a support desk and training courses.
SERVICES BEFORE PRODUCTS (Kelly 2005b)	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 increase understanding of the market before you start work on your product.
START-UP SERVICES FOR PRODUCTS (Kelly 2005b)	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.
CONTINUING SERVICES FOR PRODUCTS (Kelly 2005b)	Complex products often require ongoing maintenance and support. The company that makes the product already knows a lot about the product is well positioned to do this activity too. By sharing knowledge between services and products operations both can be improved.
COMPLEMENTOR, NOT COMPETITOR (Kelly 2005b)	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.
SERVICES TRUMP PRODUCTS (Kelly 2005b)	Your company has been successful selling products but you are running out of growth, you may already be loosing money. Therefore, use your knowledge of the products to move up the value chain and sell services instead of or in addition to products.
SAME CUSTOMER, DIFFERENT PRODUCT (Kelly 2007)	Existing customers are easier to sell to than new ones. But if you only have one product you have nothing more to sell. Therefore have additional products you can sell to existing customers.

4.2 Packaged Services



Compare two descriptions taken from the internet on 27 November 2007:

"Blue Skyline offers a mixture of consultancy and mentoring to assist the team at the same time as enabling the delivery of the system." http://www.blueskyline.com

"We help companies in the chemicals industry drive their performance to new heights by capitalizing on important business and technology opportunities." http://www.accenture.com

Which gives the best description of what the consultants actually do?

Context Your business delivers technology services to corporate customers.

Problem **How do you explain to customers what your services are?**

Forces Services can infinitely flexible, but that makes it difficult to explain to customers what those services are. The more variable the service is the harder it is to explain. Using value statements and generalisations in the descriptions makes it difficult to explain what you do in a few words.

Customers expect consistency in service delivery. They may come to know and trust an individual consultant. But if they only buy this consultant's time you loose the flexibility sell her expertise anywhere else. If each consultancy assignment depends on a named individual consultant it is difficult to grow a business. To be effective consultancy businesses need to be able to swap individuals on assignments.

Many customer problems look alike on the surface; managing a data centre for corporation X can be a lot like managing a data centre for corporation Y. But there are also unique problems; applications developed for corporation X might be very different to the ones for corporation Y.

Customers often engage consultants to reduce costs, but sometimes they are looking for strategic services to create a competitive advantage. And sometimes cost reduction is strategic.

Solution

Think of your services like products; explain what you do as a well-defined product. Demonstrate that you understand the customer problems your services are addressing. Add product-like attributes to your services. Market your services as products with defined problems, defined actions, and defined outcomes.

Segment your customers, potential customers and their problems; identify the common problems that occur again and again. Devise common service solutions that can address these problems. (Separate the unique problems and deal with them as unique projects.)

Initially you need to work on marketing. Market your services as products. Next you need to work on your delivery to create common solutions to common problems.

Marketing:

- Identify the common problems, common causes and common 'pain points.'
- Produce case studies and datasheets for your services. Show how your services solved the problems.
- Identify organizations that you expect to have the same issues and engage with them.

Delivery:

- Break the services down into repeatable steps and where possible offer a defined price for a defined benefit or outcome. Commonality will allow economies of scale to be extracted.
- Consultants need to start assignment thinking about what they have done before and what they can reuse.
- Consultants need to be trained to find, and rewarded for finding, commonalities across services and service engagements.
- Consultants need to be motivated to share personal findings with each other.

You will need to decide the financial model behind your service products. A defined problem resulting in a defined outcome suggests a fixed price service rather than charging for services on a time and material basis.

Consequences Treating your services like products makes it easier to describe what you actually do, and what the end result is. The more you make your services look like products the more consistency customers can expect, and will come to expect.

Customers are buying a specific product not a specific individual so it is easier to swap consultants during the assignment. This does not mean they will welcome the replacement of an experienced consultant with a new hire. Sometimes it may pay to send new hires out as “shadow consultants” (no charge to customers) until they learn the basics.

Commonality benefits customers because services are delivered more quickly, at a lower cost with fewer complications. However, offering different customers the same packaged service treats all customers the same – all solutions come from the same *cookie-cutter*. The specific needs of an individual customer may be lost. Where needs are different they must be treated differently.

When a customer is seeking to minimise cost they may be happy with a cookie-cutter approach because it delivers maximum cost reduction. But then they will not recognise any competitive advantage if you deliver them the same services as their competitors.

Over time commoditisation of these services may occur. When this happens you may either lead the transition to commoditisation or change your strategy.

Managing services like products entails cost. You will need to appoint product managers or senior consultants who are responsible for identifying and managing the service products.

Offering a customer a fixed price on a service contract can be difficult and leave little room for unexpected problems. Indeed many organizations find charging for unexpected problems to be profitable. (Consult the discussion in CONTINUING PRODUCTS FOR SERVICES (Kelly 2005b).)

Variations -

Examples “These days, IBMers talk about "productising" services, turning them into clearly defined offerings that can be marketed and delivered in much the same way that new mainframe computers are. [IBM’s] small and medium-sized business unit, for example, now distributes a catalogue outlining its main services.” *Financial Times* (Waters 2006)

Also known as -

Related work -

Sources Financial Times 11 July 2006 - “IBM repackages brain power” (Waters 2006). Image from iStockPhoto.com (4179993)

4.3 Account Management

Big customers who spend a lot of money with you can represent a big chunk of your income so they are not "just another sale." When your product is important to their company you are more than just a supplier and your relationship is about more than just products. There is value in the relationship itself not just the sales.

Understanding your relationship will help you better serve your customers, secure future revenue and create opportunities to increase your profits.

Context You are selling technical products and services to corporate customers. SAME CUSTOMER, DIFFERENT PRODUCT (Kelly 2007) suggests you benefit the most when you sell more products to your existing customers. You might be using

Problem **How do you avoid losing existing customers? How do you understand what customers really want?**

Forces Finding and selling to new customers is expensive but, by definition, existing customers already have at least one of your product(s) so there is no obvious sale to be made.

Corporate customers face multiple opportunities and problems in their own business and market. Some of these issues may create opportunities for your products and services but you need to know what these issues are.

Making a sale should create opportunities for further sales and support contracts. But your customers are your competitor's prospects; you still need to ensure your customers remain your customers. You want them to buy more from you but once you have made the sale you need some reason to stay in contact.

Sales staff are selected and rewarded on the basis of their ability to win sales, but managing an account over the long term requires more than just selling. Customer may be deterred from talking to people in your company if every time they do a salesman tries to make another sale.

Solution

Treat customers as valued collaborators; continue to actively work with customers after a sale has closed. Appoint named account managers who can build a relationship with both the enterprise and the individuals who work there.

Seek to understand how the customer is using your products, the challenges facing the customer and opportunities that exist for helping customers meet these challenges. Ask lots of questions: *How they are doing with the product? Was it what they expected? Do they need any help? What else could the product do?*

Managing a customer account goes beyond selling and there is more to keeping customers happy than selling at a low price. It includes the post-sales experience: support services, training and customer follow-up. This is provided by a team not an individual.

Rather than focus on the next sale, focus on keeping your customer happy. In the process find out what else they need and who else in the organization may benefit from use of your product. Continue to learn about your customer's needs and their problems. When the time is right offer them your solutions.

Create a culture that encourages ongoing contact and dialogue with customers. Build continuity in the relationship; be responsive to the customer needs and de-emphasis contact based purely on sales. Aim to stay involved over the long time and build a trusted relationship with customers.

Sales people may not be the right people to manage an ongoing relationship. While they may be good at opening doors, making first contact and closing a deal they may lack the skills and motivation to maintain an ongoing relationship.

One option is to split the sales and account management roles. Once a sale is made, or even before, introduce an account manager who will continue the relationship and look after the customer. However some sales people may resist "handing over" an account they have won. Alternatively supplement your sales people with account managers who look after the account when there are no sales in prospect. Use SALES/TECHNICAL DOUBLE ACT in both cases to split commercial and technical issues. Product managers (and business analysis) can supplement account managers to increase the depth of customer understanding.

When recruiting account managers look for people who will be interested in building a relationship rather than just making the next sale. Account managers who are simply sales staff working on commission may not be motivated to keep a relationship going when there is not sale in prospect. Balance remuneration so staff can afford to build the relationship rather than just sell, sell, sell.

According to McKenzie (2001) a customer relationship is a conversation with exchanges. There is value in the relationship itself, not just the product/money exchanges. Customers who see value in the relationship will continue the conversation by buying more products. Active account management represents an investment to maintain and increase that value.

No one pattern or single set of actions can guarantee your customers return to buy more from you. By building a trusting relationship and continuing to learn about your customers you should at least see problems before they occur, and position yourself to find opportunities.

Consequences Selling additional products and services to existing customer can be cost effective and information rich. Technology change creates opportunities for everyone: customers, competitors and yourself.

Engaging with a customer on a regular basis will allow you to learn their future growth plans and requirements. Knowing customers' future needs can inform your own business decisions leading to better products and benefiting both customer and supplier.

Competitors – especially new entrants – lack the customer assets you have. Investing in your customers will create a deeper relationship thus making it more difficult for competitors to poach business.

Account managers will need to make visits to the customer and spend time to understand the customer. Too much customer contact may annoy the customer and make them feel they are being constantly sold too. Having other points of contact, like customer care and product managers, will help build trust and collect information without a sales motivation.

An active account management programme will cost. You will need to employ additional staff; pay salary, travel and entertainment expenses even when sales are not being made. Such expenses may be seen as easy savings when times are tough but they represent investment in your relationship with customers and keep open the prospect of future sales.

Variations -

Examples A London software company supplied applications to most of the major players in the mobile telecoms market. The remaining sales prospects were small fry. New sales had to come from selling more products to existing big customers so it was important to create a positive sales, and post-sales, experience. A hard sales approach might secure the immediate sale but damage the relationship and future prospects. Account management was handled by a SALES/TECHNICAL DOUBLE ACT, one for commercial issues (sales) and one for technical issues (everything else).

Another London software company, this time in the media sector, had salesmen make an initial sale. They then handed accounts over to dedicated account managers. However the hand-over was poorly defined, sales staff didn't like giving up customers and the account managers lacked technical skills. In some cases it worked, in others it didn't.

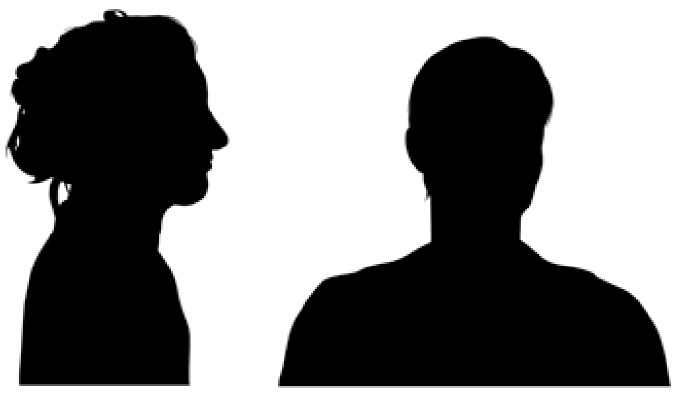
Also known as -

Related work This pattern can be used to help implement ITS A RELATIONSHIP NOT A SALE from Customer Interaction Patterns (Rising 2000). BUILD TRUST and other patterns from the same language are also useful.

CONTINUING SERVICES FOR PRODUCT (Kelly 2005b) describes how to continue services as additions to your product sale. Services like technical support and training can generate continuing revenue over the lifetime of a product.

Sources Image from iStockPhoto.com (4583601)

4.4 Sales/Technical Double Act



Allan was responsible for evaluating and selecting an enterprise search engine. Downloading and installing the trial software was easy but then technical problems and questions arose.

When the search engine salesman called he brought a technical consultant with him. The consultant was knowledgeable about the things the salesman wasn't and could discuss technical issues in detail. Even after the sale the technical consultant kept in touch.

Context You are using ACCOUNT MANAGEMENT to sell high margin technical products to business customers.

Problem **How do your avoid overwhelming your account managers with commercial and technical issues? - Both before the sale and the after.**

Forces Selling a technical product involves more than talking about technology; there are commercial (e.g. price) issues to discuss. But, technical people aren't usually good at commercial aspects and sales people aren't usually proficient in technical aspects.

Even when you can find someone who can cover the commercial and technical aspects of a product there is often too much for one person to take in. Technical products often require in-depth technical knowledge and commercial knowledge.

Within customer organizations the people who make the technical decisions are often different from the people who make the decisions on expenditures. These groups may expect to deal with different levels of seniority and expertise in your organizations.

Discussing commercial and technical questions for a complex product takes a lot of time and energy. But you don't want to spend all your energy on these questions. At the same time as negotiating the deal you want to gain insights into your customer's business and how they want to use the product.

Solution **Have your customer account managers work in pairs, one handles the commercial aspects of the product and the other handles the technical aspects.** This will allow you to hire the best possible sales people and technical people for your product. Individuals can focus on what they do best rather than trying to master diverse skills.

Technical managers should come from a technical domain and should be trained in-depth on the product. Some technical managers may come from internal groups like development or support. Sales people may not need to understand the product in-depth but they should know the benefits and advantages of its application. Each group needs to respect the other and refer questions when appropriate.

While technical managers may be involved in pre-sales calls their contact with customers should also extend beyond the initial sale. As technical manager win the trust and confidence of customer's staff your overall corporate relationship will deepen. You will better understand your customers and serve them better.

Technical managers may help clients with technical support issues, configuration, installation and training. Be careful to not overload the technical manager with too much work, most likely they will be working with several customers. With growth you may want to create dedicated groups to deal with specific issues and relieve pressure from technical managers, e.g. a technical support desk and a training team.

The sales oriented commercial managers can concentrate on the financial and business aspects of the deal, e.g. pricing, terms and conditions, license renewals, support agreements, etc. They can take a strategic view and look for opportunities to sell more products.

Both managers should talk regularly about the customer, their current needs and their future needs. They should meet with the customer regularly and conduct periodic account reviews that bring together everyone involved with managing the customer account. Such reviews can help identify sales prospects and future client needs.

Consequences Using more than one person to manage the customer relationship allows people to specialise in what they are good at. It is easier to find dedicated individuals than expect individuals to be proficient in very different fields.

Having more than one person involved in a customer relationship acts as a safeguard against people leaving your company. Losing a sales person can be unfortunate; if they take your customers with them it can be a disaster. With two people managing the account you can provide continuity.

Where more than one person is involved in a sale the customers may become confused about who deals with which aspects. Even if customers wish to clarify the relationship you might prefer to leave the boundaries vague. Blurring the lines may create opportunities for extra contact and information exchange. Still be careful not to confuse your customer too much.

Customers will receive better commercial and technical service. You will gather more information because different staff talk about different things to different people. Over time you will gain a more complete view of your customer. Customer employees will come to trust and share information with your representatives. Having two views of the customer will improve your understanding of the customer and issues, particularly political ones, involved in a sale.

Having multiple account managers further increases the costs of managing a customer account. This is feasible for high margin products, for low margin products you might need to use service teams rather than individuals.

Variations When customers are very large one account manager may not be enough to cover all contacts. Multiple account managers will allow responsibilities to be divided. Different managers may deal with different customer divisions or geographical areas.

Examples Many organizations employ *pre-sales consultants* or *sales engineers* as technical contacts before the sale is made. This is a form of double act but sometimes ends once the sale is made.

Also known as -

Related work Sales/Technical Double Act can be used to help implement ACCOUNT MANAGEMENT.

Software developers sometimes use DEVELOPING IN PAIRS (Coplien and Harrison 2004) to increase productivity. One developer reviews work as it is performed and helps with the decision process. This is a form of double act but the developers have similar skills and periodically switch the roles of reviewer and coder.

Sources Image from iStockPhoto.com

5 Acknowledgements

I am indebted to my shepherd for VikingPLoP 2007, Uwe Zdun, for his time, comments and observations. I would also like to thank the participants of the VikingPLoP workshop who reviewed this paper, and in particular Cecilia Haskins, Jim Coplien, Jan Reher, Jürgen Salecker and Kristian Elof Sørensen.

6 History

Date	Event
December 2007	Post conference revision published to web prior to proceedings.
September 2007	VikingPLoP 2007 conference
Summer 2007	Shepherding for VikingPLoP
January 2007	Extracted from EuroPLoP submission
December 2006	First draft

7 Bibliography

Alexander, C., et al. 1977. A Pattern Language: Oxford University Press.

Bricout, V., D. Heliot, A. Cretoiu, Y. Yang, T. Simien and L. Hvatum. 2004. "Patterns for Managing Distributed Product Development Teams." In 9th European Conference on Pattern Languages of Programs (EuroPLoP), eds. K. Marquardt and D. Schutz.

Conway, M.E. 1968. "How do committees invent?" Datamation(April 1968).

Coplien, J.O. and N.B. Harrison. 2004. Organizational Patterns of Agile Software Development. Upper Saddle River, NJ: Pearson Prentice Hall.

Gamma, E., R. Helm, R. Johnson and J. Vlissides. 1995. Design Patterns - Elements of Reusable Object-Oriented Software. Reading, MA: Addison-Wesley.

Hvatum, L. and A. Kelly. 2005. "What do we think of Conway's Law not?" In 10th European Conference on Pattern Languages of Programs (EuroPLoP), eds. A. Longshaw and W. Zdun. Irsee, Germany: UVK Universitatssverlag Knstanz GmbH.

Kelly, A. 2005a. "A few more business patterns." In EuroPLoP 2005, eds. A. Longshaw and W. Zdun. Irsee, Germany: UVK Universitassverlag Konstanz GmbH.

Kelly, A. 2005b. "Business Strategy Patterns for Technology Companies." In VikingPLoP 2005. Espoo, Finland.

Kelly, A. 2006a. "Patterns for Technology Companies." In EuroPLoP, eds. L. Hvatum and W. Zdun. Irsee, Germany: UVK Universitassverlag Konstanz GmbH.

Kelly, A. 2006b. "Positioning Business Patterns." In EuroPLoP, eds. W. Zdun and L. Hvatum. Irsee, Germany: UVK Universitassverlag Konstanz GmbH.

Kelly, A. 2007. "More patterns for Technology Companies." In EuroPLoP, eds. L. Hvatum and T. Schümmer. Irsee, Germany: UVK Universitassverlag Konstanz GmbH.

Manolescu, Dragos-Anton, Markus Voelter and James Noble. 2006. Pattern Languages of Program Design 5. Upper Saddle River, N.J. ; London: Addison-Wesley.

Marquardt, K. 2004. "Ignored Architecture, Ignored Architect." In 9th European Conference on Pattern Languages of Programs (EuroPLoP), eds. K. Marquardt and D. Schutz. Irsee, Germany: UVK Universitatssverlag Knstanz GmbH.

McKenzie, R. 2001. The Relationship-Based Enterprise: McGraw-Hill Ryerson.

Mintzberg, H. 1994. The Rise and Fall of Strategic Planning: FT Prentice Hall.

Rising, L. 2000. "Customer Interaction Patterns." In Pattern Languages of Program Design 4, eds. N.B. Harrison, B. Foote and H. Rohnert: Addison-Wesley.

Schmidt, D., M. Stal, H. Rohnert and F. Buschmann. 2000. Pattern-Oriented Software Architecture. Chichester: Wiley.

Waters, R. 2006. "IBM repackages its brain power." In Financial Times.

Easy GUI maintenance

Met-Mari Nielsen
Systematic Software Engineering A/S
Søren Frichs Vej 39
DK-8000 Århus C
+45 89432000
mmn@systematic.dk

Abstract: Throughout an applications lifetime style guides are employed to create and maintain consistency. Features covered in such documents range from code syntax to interaction flows and visual elements. The patterns presented in this paper will help control and maintain aspects of the GUI in the simplest possible way, by addressing these considerations in the system architecture and ensuring that the GUI design is easily re-configured.

Intended audience

If you are in the process of designing architecture and/or defining an implementation processes these patterns will help you to reduce development and test effort as the application evolves.

As the aim of these patterns is to reduce testing and implementation effort introduced by the use of visual guidelines, people with interest in visual guidelines, such as user experience engineers, graphics designers and testers may be relieved to know that such patterns exist and will have an interest to promote such patterns in the development process.

Note that although this paper is written using a pattern format I have only observed two projects actually implement the solutions described. If you have any extensions, comments and actual uses of these "proto-patterns" please feel free to contact me at mmn@systematic.dk.

Table of contents

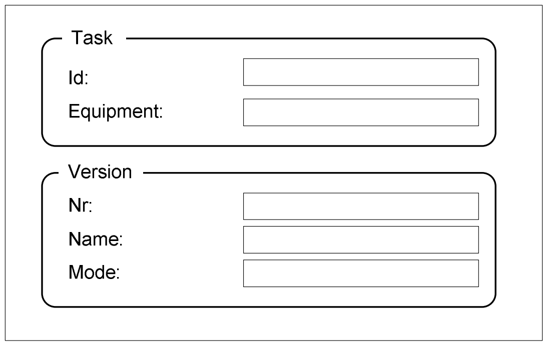
Visual Consistency Framework

Thumbnail: It has been decided that your project needs some visual consistency guidelines. When, how and where can you verify that the application adheres to this?

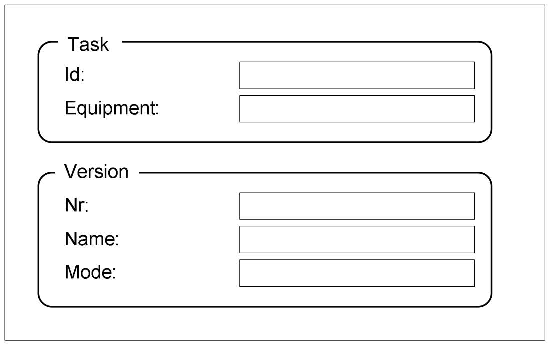
When a guideline, such as a color guide, is implemented in the code and not subsequently changed or overruled, the code itself will assure that the guide is adhered to. If the guideline is encoded in a central well-known place, implementation of GUI-features will be easier. Utilize an object oriented component framework and use inheritance to limit the "hotspots" of code impacted by visual consistency guidelines.

Problem

Visual consistency is one of the elements that give an UI a "professional" look and to achieve this projects often use some kind of guidelines and/or process such as review or test. Below is an example of common mistake when implementing a GUI component:



Example 1: Distance between the fields in Task and Version varies



Example 2: Distance between fields is constant

In Example 1 the spacing between individual fields will probably look okay if the two panels, Task and Version, are implemented and reviewed separately, but when put together on a single screen the layout looks slightly unbalanced, and users may think that the items in Version are more related to each other than the items in Task. If this is not your aim then Example 2 is what you want in the final version.

Getting the picture

The following stories are brief descriptions of two actual projects. In the first case the implemented layout characteristics worked and in second case it did not.

The team implemented a school-maintenance system (covering scheduling of classes and teachers, computation of salaries and administration of pupils) using Smalltalk and VisualWorks for the code. The style guide had been coded into the framework from the very beginning and was actually never used in its paper incarnation. It was an integrated part of the component hierarchy and everybody knew to look in the code for the GUI definitions of BaseWindow, BaseButton and BaseLabel and not to overrule these settings. The team used these settings to implement complex schema-components and to reconfigure the GUI when screen real estate became scarce. There was never an actual need to verify distances between input labels or sizes of components against the written style guide.

Another project implemented a large diagnostic tool showing system status data, using JAVA and Swing components. From the very beginning all GUI constants were grouped and implemented in property files, but according to feature and not component type. All GUI settings were separated from the actual implementation via these property files (to the degree of each label having uniquely named width and height properties). Each time a new window was added to the application the property had to be hunted down in the property file, or more commonly visually checked that it approximated the guide. It was even harder to ensure that labels from different screenshots, showing similar data, would have the same layout. Not to mention the rework each time a parameter needed to be shown with new layouts such as number of decimals or another text alignment. Actually the project had to have separate work tasks just for *estimating* the impact of such rework.

Discussion

Both the projects had their textual style guides in place and visual consistency guidelines played a large part in these documents. None the less none of the developers on either project ever really read the style guide.

The team implementing the diagnostic tool relied on formal reviews of the GUI as part of the development process. They also used focus groups, expert users making draft designs and several reviews during the implementation phase to ensure usability and visual consistency. The school maintenance project relied on one small user group to define requirements and two tester/expert users to verify that requirements were met. After initial development no special considerations towards usability or visual consistency were made, it was not needed during the last year or so of development.

Style guides in textual form are common tools in software projects and two excellent guides on how to develop and use style guides are found in [Wilson] and [Gayle]. But personal experience says that developers will avoid reading documentation if they can and just copy what they need from the existing code base. It can be tedious work to verify that a printed style guide actually is followed as opposed to an implemented one; its time-consuming tasks both to verify and implement graphical content. If one can ease the "burden" of copy paste coding and still ensure that visual consistency is maintained, time and effort can be eased when it comes to verifying that the GUI actually follows the guide lines.

Existing GUI frameworks are often large and full of settings and details to ensure that the framework can accommodate all kinds of programmer needs. Most projects do not have resources to ensure the framework is used as intended. Somebody will find something that works and that will be it. Sometimes it will be a team member, who knows how to use the framework, but this will not always be the case and there will always be copy-paste programming to propagate the bad solutions as well as the good ones.

[Reed & Davies] consider the following elements important: font styles, font sizes, colors and contrast. When combining these elements in a conceptual hierarchy and applying this hierarchy consistently throughout the application· the user can easily decode the information displayed by the system and react accordingly. The problem then remains to settle on limited set of graphical effects and how maintain this set throughout the applications life time.

Solution

Use your architecture to impose limits on your GUI framework. Enforce visual consistency rather than implementing ad-hoc solutions and relying on visual tests of guide line compliance. Develop the framework in a controlled way to keep it simple but applicable to *your* project.

The solution consists of equal parts programming and process as underlined in the following sections.

Do not rely solely on some off-the-shelf framework without analyzing your project specific needs first. Most applications usually settle on some small subset of components during development and if this subset is defined and controlled early in the projects lifetime this can add greatly to visual consistency.

Control is important. Even though most of these frameworks have default settings that allow them to comply with industry standards, it is possible even with a fixed set of guidelines, for two developers or two teams to develop widely different interfaces.

Ensuring that developers use a baseline, such as a project-specific set of components helps emphasize and control when implementation deviates from the common framework.

```
public abstract class MyBaseGroupBox extends JPanel{
            private Insets baseInsets;

            public MyBaseGroupBox(){
                        setVisualElements();
            }

            private void  setVisualElements(){
                        GridBagLayout layoutManager = new GridBagLayout();
                        setLayout(layoutManager);
                        baseInsets = new Insets(3, 20, 10, 15);
            }

            public void addElement(Component element, GridBagConstraints constraints){
                        if (constraints == null){
                                    constraints = new GridBagConstraints();
                        }
                        constraints.insets = baseInsets;
                        add(element, constraints);
            }
}
```

Code example 1: Simple (JAVA SWING) example of an abstract root class controlling margins

When it comes to simple and “global” stuff which are the guts of visual consistency-parameters, data that define visual consistency should be controlled using a hierarchy.

Using inheritance to define this hierarchy will make it very clear what impact changes will have on the system, which components are affected and what subclasses are needed to support all necessary visual elements.

Code design

Ensure that root classes are defined for each type of visual element, and let these classes “hard-code” the needed settings defined in the style guide.

A good way to signify a root class is give it a ‘base’-name.

Inheriting from the class MyBaseGroupBox shown in Code example 1 would ensure that components adhere to a guideline stating that components inside panels should have the following margins: top = 3, left = 20, bottom = 10 and right = 15.

Development Process

Let your development process ensure that:

All GUI-elements use these root classes, without overruling the base settings.

Set up checkpoints, such as code reviews, where deviations from the general component interface can be caught and discussed before being added to the projects code base.

Keep deviations few and well documented. It should be clarified both in code and the textual style guide why, when and where deviations were made.

Consequences

Using a component hierarchy as suggested above would eliminate inconsistencies such as the one in Example 1. The initial effort on deciding how, where and what interfaces to implement will pay off when it comes to the later stages of implementing and testing.

All GUI elements by default follow the visual guide, and thus testing is reduced to ensuring that the root-classes are implementing the needed settings.

If a GUI element does not adhere to the guide, the deviation is documented and facilitates tracking of visual inconsistencies.

Related patterns

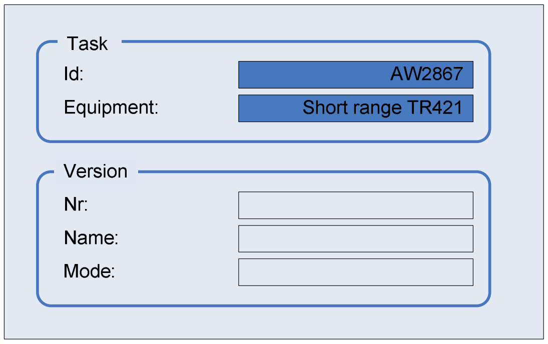
If the system accumulates deviations from the root classes or visual guide lines are changing then look to “Evolution of Visual Guidelines”.

Evolution of Visual Guidelines

Thumbnail: Laying down a style guide in concrete doesn’t work well within an iterative development process. Visual guides are often redefined during the implementation phase, fonts are not readable, labels are too small, margins are badly balanced, and screen real-estate is limited. But as systems mature and content is released GUI changes become expensive in terms of development hours and systems testing. To lower the cost of GUI-implementation and recapturing let your architecture contain a minimal implementation of “Visual Consistency Framework” and expand this in small and controlled increments.

Problem

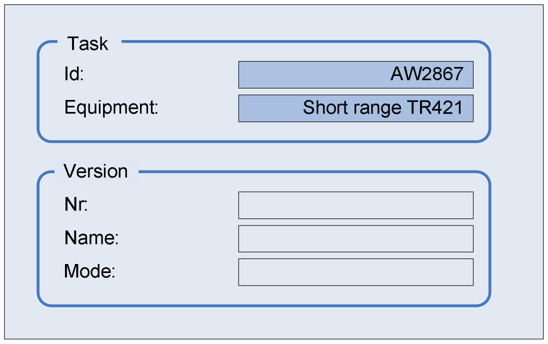
Visual consistency is one of the elements that give an UI a “professional” look. As customer needs are analyzed, implemented and tested, visual guidelines change. If the architecture is not in place incremental updates to visual consistency becomes close to impossible.



Task
Id: AW2867
Equipment: Short range TR421

Version
Nr:
Name:
Mode:

Example 3: Same color on panel-borders and background on the disabled fields



Task
Id: AW2867
Equipment: Short range TR421

Version
Nr:
Name:
Mode:

Example 4: Changing background color on disabled fields makes data more readable

When potential users are introduced to a working GUI, development is often at a stage where process, architecture and 3rd-party tools have been decided. If you haven't made allowance for user feedback, how will you get from Example 3 to Example 4 late in the development process. User objections at this stage could range from the fact that it's just plain unreadable or that screenshots print badly to the fact that the color signifies something special in a system already part of the user domain. You can never uncover all user needs, or even what they believe they need, up front.

Getting the picture

The following examples are brief descriptions of two actual projects. In first case the architecture and design decisions worked against the visual consistency patterns, in the second visuals was what sold the product.

A company decided to develop a multiplayer online game. The GUI-interaction components would be based on an open-source library, Qube, which had no generic controls. Qube only had limited capabilities for importing fonts, displaying text and had no window/component hierarchies. Thus the GUI-framework was developed in-house in C++. All graphics design lay in the hands of one chief designer. To ensure consistency within the game-elements, graphics design included visual consistency of user controls and in-game windows. From the beginning visual consistency elements was only implemented when needed, adhering to "the rule of three" [Fowler]. This gave the GUI freedom to move later in the process when feedback was received. New framework elements were added as game concepts were implemented and their graphics elements were designed. There was no need at any point of the development process to decide upon a fixed future set of components.

Another project had decided to use C#, windows forms and Infragistics components to define the user interface for a "very large and complex database system". Using windows forms provided the project with default implementation of visual consistency guidelines. Initially fast GUI-development was ensured by the GUI-builder and its grid-guide for placing components. But about 5 months after project kick-off it became clear that some new guidelines were needed and these guidelines would cover already implemented content. Using windows "default" styles had left considerable space for developer-specific visual styles even when adhering to the textual style guide. Unfortunately the GUI-builder did not politely tolerate reverse-engineering of its code files and tended to crash the development environment. The project opted to live with these crashes and with each style-change update the relevant components, one by one.

Discussion

If the GUI-architecture is in place, incremental updates to component look-and-feel need not have great impact on the code. But often the team runs into problems. Even if the product is developed with a focus on usability, the style guide is not quite done when the first features are getting implemented. And style guides change over time as needs change. A font once thought nice doesn't work in the needed point size, adding new elements to an already developed GUI may change the margins impacting layout for the whole application. Adding user interface "polish" is an activity postponed to late in the development process, when the layout is finalized.

The team developing computer game designed their architecture such that changes to look-and-feel would not have great impact on development and test. Their focus was such that importing and maintaining the library of graphical assets should be easy and that GUI was easily reconfigured during development. The team developing the database system decided to minimize the use of application-specific components. They also wanted to avoid GUI-code being written by developers. But as GUI guidelines developed a growing amount of time was spent on refactoring and reviewing the GUI.

If the project team grows beyond a few developers and the technologies chosen hinder GUI refactoring, then the GUI will likely suffer from growing visual inconsistency. This result in a project where knowledge gained (regarding customer needs and professional look and feel for that particular domain) is not utilized. Besides the obvious consequence: that the application will loose that overall "professional" look, developers and User Interaction Specialists will loose job satisfaction knowing that the job done should have been better.

A GUI style guide will never stop evolving unless you stop listening to users or terminate development on the project. Even if you have implemented a "Visual consistency framework" it can easily evolve into the generic monolith of GUI frameworks that we all have encountered or even implemented at some point in our careers. The story about the "Generic component" that grew in complexity and at last had to be refactored at great cost or even disposed (at great cost) is one often told about GUI components. When designing and evolving a GUI such tendencies must be kept in check.

Solution

Guideline and implement only the components needed for the GUI to keep consistency, and use a rule like "the rule of three" to determine when something is generic.

Consider a "Visual Consistency Framework" as a means to accomplish an architecture that supports easy system wide GUI changes, but keep it limited to avoid the "Generic component". Use your development process to determine what component parameters are relevant for refactoring/extending the framework.

Code design

The initial phase of the project visual guidelines need not cover the whole problem domain. Thus the component framework need only cover the initial set of guidelines, just long as there *is* a central "hotspot" in the code from where the GUI-settings can be controlled. Limit the component framework to the bare basics and extend as needed. Labels, buttons and window definitions will cover most initial demands.

```
public abstract class MyBaseTextField extends JTextField {
        public MyBaseTextField (){
                setVisualElements();
        }

        private void setVisualElements(){
                setSize(new Dimension(50, 25));
                setHorizontalAlignment(JTextField.RIGHT);
                setForeground(Color.white);
                setBackground(Color.Black);
        }
}
```

Code example 2: Abstract root class controlling colour, size and alignment.

```
public abstract class MyBaseTextField extends JTextField {
        public MyBaseTextField (){
                setVisualElements();
        }

        private void setVisualElements(){
                setSize(new Dimension(50, 25));
                setHorizontalAlignment(JTextField.RIGHT);
                ~~setForeground(Color.white);~~
                setBackground(Color.Blue);
        }
}
```

Code example 3: Changing background colour on all textfields inheriting from MyBaseTextField

In Code example 2 and 3 MyBaseTextField easily makes the requested change in background color (described in Example 3 and 4). Even if MyBaseTextField had not been controlling color settings, the change from JAVA defined settings to application specific settings would not be difficult. If all relevant components inherit from MyBaseTextField it is a matter of overriding JtextField's setBackground method and documenting the deed.

Development Process

Couple the code design with a development process which ensures that when new GUI elements are needed or deviations grow beyond one or two GUI elements, the code is refactored and a new base for these elements is included in the framework. If there is some sort of non-code based documentation of the visual guidelines, remember to mirror the new base settings there as well.

Consequences

If a GUI element does not adhere to the guide, the deviation is documented and it's easy to fingerprint such changes by examining the component hierarchy and thus determine when the style guide should change.

Experiments with new styles can be easily prototyped and tested as the code changes are limited to the base-classes.

Verification of new global guides can be carried out by glancing through the GUI to verify that the new style looks okay in already implemented features. You already know that all GUI elements adhere to the visual consistency guide, and thus there is no need to measure screen elements manually. If something looks wrong then it's easy to redefine the layout.

Related patterns

Implementing a "Visual Consistency Framework" will allow "low-cost" refactoring of the GUI

Known uses

The pictures used to illustrate the patterns all come from projects where I in some way or another had opportunity to work with the user interface. The oldest project was initiated in 1994 (the one with Smalltalk), while some are to run for a few years yet.

Acknowledgments

Thanks to Peter Sommerlad for his shepherding and to Katrine Ravn for encouragement.

References

[Gale]	Stephen Gale " A Collaborative Approach to Developing Style Guides", CHI 96 APRIL 13-18, 1996
[Wilson]	Chauncey E. Wilson, "Guidance on Style Guides: Lessons Learned", in STC Usability SIG Newsletter: Usability Interface, Vol 7, No. 4, April 2001
[Leadly et. Al.]	Brenda Leadley, Haunani Pao, Sara Douglas " Creating a User Experience Culture at a Non-Software Company" ACM International Conference Proceeding Series; Vol. 135
[Reed & Davies]	David Reed, Joel Davies "The convergence of computer programming and graphic design", Journal of Computing Sciences in Colleges, Volume 21 , Issue 3 (February 2006) Pages: 179 – 187
[Fowler]	Martin Fowler "Refactoring Improving the Desing of Existing Code", Addison-Wesley 1999

Content Adaptation for Mobile Web Applications

Bettina Biel, Volker Gruhn

University of Leipzig
Department of Computer Science
Klostergasse 3, 04109 Leipzig, Germany
[biel,gruhn]@ebus.informatik.uni-leipzig.de

www.lpz-ebusiness.de

Abstract: Since multiple devices with different needs access mobile Web applications, content must be adapted to each device's capabilities. However, to maintain different versions of one application's content is tedious. Therefore automatic authoring and adaption of content becomes necessary. Depending on whether content can be generated from scratch or whether it already exists or is possibly provided by a third party, implementations for content adaptation vary. This paper presents two patterns: one that generates device specific content at the server, and another one that adapts existing content at an intermediate component.

1. Introduction

Users accessing the Internet with mobile devices such as cell phones, smart phones, or Personal Digital Assistants (PDAs) are faced with constrained resources and processing capabilities making the mobile devices non-suitable for the vast majority of web pages that were designed for larger screens, i.e. laptops and desktop computers.

Reading and interacting with a mobile web application is more difficult than interacting in a desktop environment due to small screens, cumbersome input methods and environmental distractions. If content providers have to deal with a lot of pages, pictures and video data, they do not want a costly manual re-design for each type of mobile device. Of course, this problem can be handled by producing for the lowest common denominator, but the resulting pages do not meet user requirements: The majority of devices cannot provide their usual and optimal interactions; and the resulting usability is poor. Hence, content needs to be adapted to the capabilities of the different devices. Content adaptation renders one original content version for different devices by selecting, generating, or modifying content.

Mobile Web applications run on a Web server and allow user interaction only via a Web browser (client). Such client/server architectures fall into a spectrum of full and thin client architectures, or flexible client/server architectures [1]. There are three basic approaches to adapt content: server-side, intermediate, and client-side content adaptation.

Copyright © 2007 by Bettina Biel, Volker Gruhn

It depends on the distribution to what extend a developer can control the response to a browser request. Server-side dynamic content adaptation can generate marked-up content from raw material in well-known formats which allows full control, intermediate content adaptation and client-side adaptation have to process existing content with an existing markup.

This paper presents two patterns. SERVER-SIDE DYNAMIC CONTENT ADAPTATION is a server-side-only content generation pattern, INTERMEDIATE CONTENT ADAPTATION can be implemented on the server-side or on an intermediate tier (e.g. a proxy). A pattern for CLIENT-SIDE CONTENT ADAPTATION is presently being investigated.

The next section shows how the patterns of this paper are related. It is followed by a section that contains the patterns and a summary. In the Appendix, a table lists summaries of related patterns.

2. Relations between the Patterns

SERVER-SIDE DYNAMIC CONTENT ADAPTATION for the server-side-only content generation and INTERMEDIATE CONTENT ADAPTATION for the server-side or intermediate tier (e.g. a proxy) address one main goal, i.e. CONTENT ADAPTATION.

Fig. 1 illustrates the relations between the three patterns. Note that the arrows follow UML class diagram semantics, i.e. the specialized patterns inherit all information provided by their parent pattern.

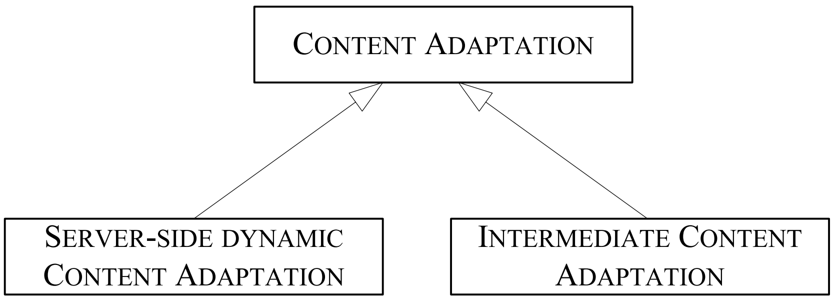

CONTENT ADAPTATION
SERVER-SIDE DYNAMIC CONTENT ADAPTATION
INTERMEDIATE CONTENT ADAPTATION

Figure 1. Relations between the patterns of this paper

Sections of the patterns include context, scenario, problem, forces, solution, consequences, related patterns and evidence/known uses.

To help the reader keep the relation between the patterns in mind, the parent pattern's sections are labeled "common". This general pattern will not have a section about specific solutions, consequences and evidence/known uses. Analogous, the specialized patterns do not include the sections scenario and problem.

3. Patterns

This section describes the patterns: first of all the parent pattern containing the common ground (esp. regarding the scenario, problem and forces), followed by the inheriting patterns SERVER-SIDE DYNAMIC CONTENT ADAPTATION and INTERMEDIATE CONTENT ADAPTATION.

Pattern: CONTENT ADAPTATION

Common Context: Multiple devices with different needs access mobile Web applications, that have to deliver content suitable for each device.

Common Scenario: A user traveling by train wants to get some information about her train connection and tries to access the mobile Web site of a railroad company. But her mobile phone cannot display the web site properly: the browser cannot interpret the markup code and shows an error message.

Common Problem: Different mobile devices have different input/output capabilities and thus require data in different formats. To prepare content only for a few particular devices would exclude many users, and development and maintenance of the different versions would be very time-consuming. How do you adapt dynamic or static content automatically to device properties?

Common Forces:

- There are a lot of different mobile devices, and for a smooth user experience, it is necessary to present content that is adapted to the needs of a certain mobile device.
- Manually re-authored content maintenance is very time-consuming; and simple automatic re-authoring does not produce user friendly results. Often the same content is just minimized, resulting in too many and too small pictures, too small font sizes, too many interaction steps, too large forms, too much costly network traffic. For these reasons, a more sophisticated approach to deal with content is necessary, although this will consume effort at the beginning of the development process.
- Content must be prepared with some editorial skill: it must be written for the Internet (see the pattern INVERSE PYRAMID WRITING STYLE in Duyne et al. [2], summarized in the Appendix).
- It may be necessary to establish a Secure Socket Layer (SSL) to ensure secure direct connections between two parties. This is important for user trust and acceptance in m-business, e.g. for the use of services such as m-payment, m-banking, e-mail. Customers will not use such services if there are no reliable security mechanisms like for example SSL-technology.
- If there are adaption-based changes to web pages and content, copyright questions regarding abridged text or cut pictures might be raised.
- Due to the extra time delay introduced by the adaption of content, scalability and resulting performance problems have to be considered.

Common Solution: *Use semantic information, device profiles and according style sheets to adapt content to device properties.*

Related Patterns: Different client-server architectures make different approaches possible. The pattern refers to three specialized patterns: SERVER-SIDE DYNAMIC CONTENT ADAPTATION presents a server-side-only content generation; INTERMEDIATE CONTENT ADAPTATION can be implemented on the server-side or on an intermediate tier (e.g. a proxy); a pattern for CLIENT-SIDE CONTENT ADAPTATION, that adapts the content at the client, is presently being investigated (and therefore not in the scope of this paper).

Pattern: SERVER-SIDE DYNAMIC CONTENT ADAPTATION

Context: Consider a thin client architecture with a thin mobile client without caching and data storage and a fat server that adapts the content and stores data. The server is responsible for all logic: It will generate content for each kind of device, considering the format of the code, the screen size, and the input/output capabilities of the devices.

Forces:

- A set of new markup tags and attributes could be implemented to write device-specific source code, but such an extension of markup standards like HTML would not be understood by many browsers.
- *For general forces, see parent pattern* CONTENT ADAPTATION.

Solution: *Implement dynamic generation of tailored content for mobile devices by using semantic information in raw content, device profiles and according style sheets.*

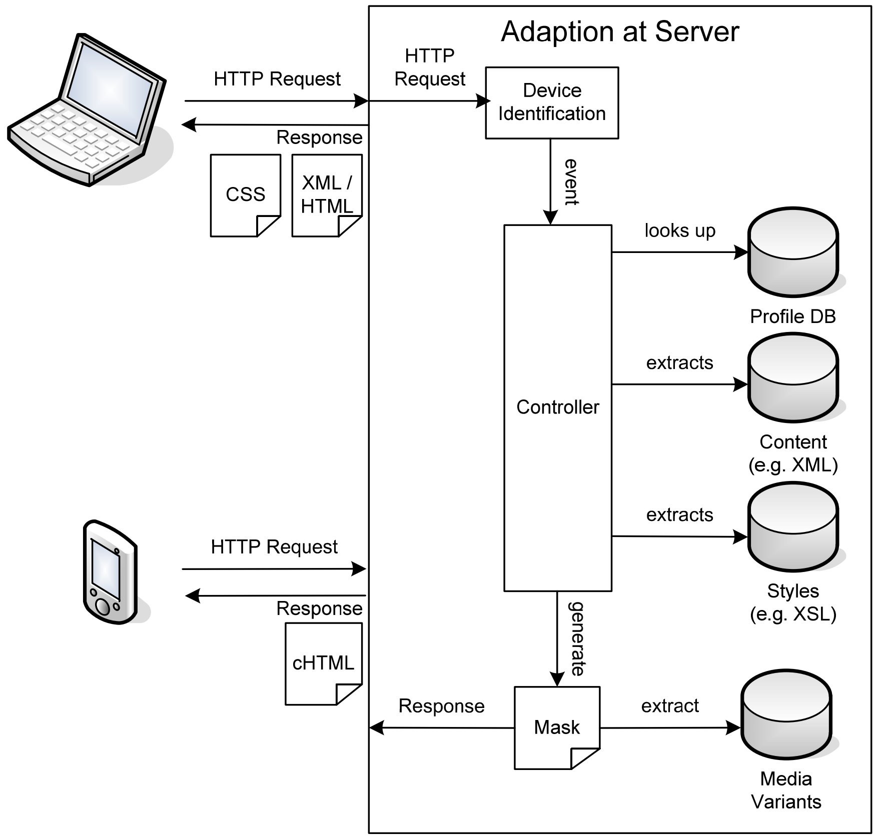


Figure 2. Coarse-grained structure of SERVER-SIDE DYNAMIC CONTENT ADAPTATION

A client sends a HTTP request to a server. Using this information the (mobile) device type can be detected by a DEVICE SELECTOR. A controller uses a profile data base to determine the device's capabilities and constraints. It holds all necessary information for the adaptation: screen size, input/output capabilities, and the requested source code format of the device. The controller will use this information to decide on a device class. This device class is needed for the selection of a specific style sheet of a class which will be used by the controller to process the content data and generate a mask that will be sent to the device later.

The raw content needs to include Meta information, i.e. semantic information that indicates possible fragmentation of content and content's priorities. It should be prioritized what content *must* and what content *can* be presented (depending on importance) and what atomic elements have to be presented on one page together, i.e. which dialog fragments must not be separated (for example form fields) to ensure a user-friendly SMALL SCREEN PAGE FLOW. The page flow consists of all paths possible that a user can take "walking through" an application. This information is important for pagination.

Depending on the device class a controller can decide what page flow is used. This implies that different page flow versions have to exist for different screen size classes. Style sheets for each device class are used to provide information about how the controller should render the elements and what media variants the source code should refer to (see Fig. 1). This way, rule changes at that level do not require recompilation of the transcoding component, and many different device-specific style sheets can be provided.

This solution is also usable for language translation, Web accessibility and speech-enabling efforts.

Consequences:

+ Full control of the content generation is allowed and should provide the best results as the server generates adapted content.
o The pattern resents a useful realization of the concept "once authored, used everywhere". It is necessary to spend time to design content and its structure. Although extra time spent at the beginning saves time in the maintenance of the basic content compared to the labour-intensive maintaining of multiple content versions.
+ This solution prioritizes content to decide what is sent to a smaller device. Depending on the language that is used, a sophisticated control is possible.
+ Regarding security of transactions for example in m-commerce applications, this pattern allows to establish direct Secure Socket Layer (SSL) connections.
+ There are also no copyright issues if one's own intellectual property is changed.
+ There is only the typical delay for generated web pages at run-time.
+ The pages/masks that are generated are device specific.
o The quality of the results depends on correct and stable device detection. A device profile data base has to be kept up to date and complete (for example using the User Agent Profile that describes all characteristics of a device available at public or commercial Web sites).

Related Patterns: If a third party has to deal with delivered content and needs to prepare it for processing, the server-side solution is not very helpful: alternatively, an INTERMEDIATE CONTENT ADAPTATION can be accomplished. The parent pattern of both patterns is CONTENT

ADAPTATION. The pattern DIALOG FLOW MANAGER ([3], summarized in the Appendix) can be used to control such page flows dynamically.

Evidence/Known Uses: DiaGen generates content using an annotated XML-language, but does not provide multimedia elements [5]. Goebel et al. [6] present a Java software architecture for the adaptation process including device identification, classification, session management, data input validation, dialog fragmentation, and transcoding.

Pattern: INTERMEDIATE CONTENT ADAPTATION

Context: The content cannot be generated but is received from a third party, e.g. there are many Web sites to be adapted, or existing content should be used as is.

Consider a Web proxy (or an extra tier on the server side) which enables browsing of mobile web applications on PDAs or mobile phones without changes on browsers or servers, by working as an intermediary between clients and server.

Forces

- *For general forces, see parent pattern* CONTENT ADAPTATION.

Solution: *Implement a dynamic adaption of content, by using a pipeline that holds all information how content is produced in response to a request.*

The central idea of a pipeline {Buschmann, 2002 #190} is to combine components in a sequence configuration to process incoming data that is passed along. The output of one component is piped as input into another component.

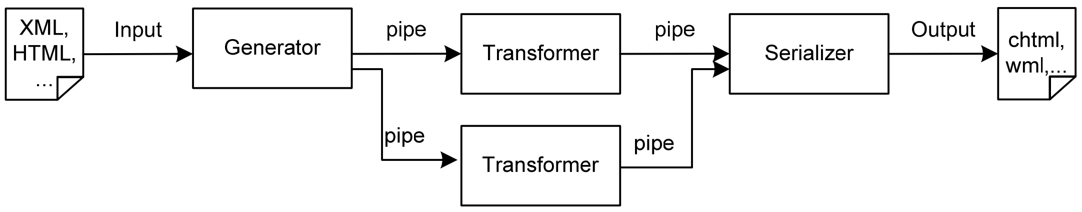


Figure 3. Coarse-grained structure of INTERMEDIATE CONTENT ADAPTATION

There are three types of pipeline components for this pattern. *Generators* take the static or dynamic content from an incoming request and provide a method to feed data (i.e. generate XML as SAX events). This stream of events is processed by one or more *transformers*. Transformers use style sheets to transform the raw content into other content by using style sheets. At the end of the pipeline, the stream of events is serialized by a component called *serializer* that produces a HTTP response.

The best results can be achieved by using style sheets for different device classes and if received content is organized in a very structured way (XML, XHTML). If the content is not very structured, a generator has to transform non-processable content into standardized processable content by using annotations (in the source code text or by an external annotation file). The annotation needs to hold Meta information, i.e. semantic information that indicates possible fragmentation of content and content's priorities. It should be prioritized what content *must* and what content *can* be presented (depending on importance) and what atomic elements have to be presented on one page together, i.e. which dialog fragments must not be separated (for example form fields) to ensure a user-friendly SMALL SCREEN PAGE FLOW. The page flow consists of all paths possible that a user can take "walking through" an application. This information is important for pagination and can be used by the style sheets.

Which style sheet can be used for one content type or device class is defined in one pipeline document (see Listing 1), defining a work flow, i.e. the sequence and which requests are diverted to go through a particular set of components.

```
<?xml version="1.0" encoding="iso-8859-1"?>
<map:sitemap xmlns:map="http://apache.org/cocoon/sitemap/1.0">
    <!-- use the standard components -->
    <map:components>
        <map:generators default="file"/>
        <map:transformers default="xslt"/>
        <map:readers default="resource"/>
        <map:serializers default="html"/>
        <map:selectors default="browser"/>
        <map:matchers default="wildcard"/>
    </map:components>
    <!-- let cocoon know how to process requests -->
    <map:pipelines>
        <map:pipeline>
            <!-- respond to *.html requests with
                 our docs processed by doc2html.xsl -->
            <map:match pattern="*.html">
                <map:generate src="{1}.xml"/>
                <map:transform src="doc2html.xsl"/>
                <map:serialize type="html"/>
            </map:match>
        </map:pipeline>
    </map:pipelines>
</map:sitemap>
```

Listing 1. The minimal “sitemap” configuration for the Cocoon Framework [4] defines components and then defines in what order and how they are used.

To reduce response time, not all content should be generated anew. XSLT-based transcoding lacks of performance due to the time necessary for interpretation. It is recommended to pre-process the code, i.e. general content, graphical and multimedia elements for typical screen sizes should be prepared for variant selection according to the quality of service and the detected device type.

Consequences:

+ This pattern might be accomplished by a third party, who receives third party content and needs to prepare it for processing. By that, the content adaptation can be outsourced.

+ It is possible to use the pattern as an add-on to an existing Web site that can remain.

+ Although content is tailored to the devices needs, the quality of the results depends on to what extent the pipeline can gain full control of the content and the annotation.

o The less structured and multifaceted the original content is the more difficult and time-consuming it becomes to implement generators. It might become too costly to implement this pattern.

- If this solution is implemented as a proxy, it can access but not decode and modify the content that flows through a Secure Socket Layer (SSL) connection; at least not without violating security and provoking messages to users about this.

- If there are adaption-based changes to Web pages and content, copyright questions might be raised. This has to be negotiated with the content providers.

- XSLT-based transcoding lacks of performance due to the time necessary for interpretation. Response time can be reduced, I not all content should be generated anew but if the code is pre-processed and stored in a data base and selected at run time (variant selection).

Related Patterns: The pattern SERVER-SIDE DYNAMIC CONTENT ADAPTATION does not need to use annotation in an external file. Instead, this Meta information is included in the raw content from which device specific content is generated. The parent pattern of both patterns is CONTENT ADAPTATION. The pattern implements a Pipeline (PIPES AND FILTERS, {Buschmann, 2002 #190}).

Evidence/Known Uses: IBM WebSphere Transcoding Publisher is a plug-in for the product "Caching Proxy" and combines transcoding technology with caching proxy functions on one location. It uses a chain of filters for the dynamical adaption of web content. It uses the WebSphere Transcoding Publisher's XML-based annotation language that is used to identify and extract specific portions of a document, without having to touch the HTML source. [7] Cocoon is an open-source publishing framework that uses a pipeline that organizes in what order different kinds of contents are adapted, i.e. generated into a SAX stream (prepared formats are for example XML, XHTML, XML-based Web services, XSP, JSP, Flash, Python, WebDAV, Plain structured text), transformed and serialized to new documents (for example XML, HTML, PDF, OpenOffice, RTF, Flash, SVG, Text) [4].

4. Summary

We presented patterns that show how content can be dynamically adapted for mobile use. The main difference between the patterns is what content can be used and how the content is adapted. SERVER-SIDE DYNAMIC CONTENT ADAPTATION allows full control of the content generation and enables best results as the server generates adapted content, and presents a realization of the concept "once authored, used everywhere". The INTERMEDIATE-SIDE CONTENT ADAPTATION is expected to work with very good structured content to provide optimal results. A pattern for CLIENT-SIDE CONTENT ADAPTATION, that adapts the content at the client, is presently being investigated.

5. Acknowledgments

We would like to thank our shepherds Wolfgang Keller and Alejandra Garrido for their valuable comments that helped to improve the paper. We would also like to thank the participants of the VikingPLoP 2007 workshop for their insightful feedback and suggestions. The Applied Telematics/e-Business Group is endowed by Deutsche Telekom AG.

6. Appendix: Thumbnails

This table lists *thumbnails* of related patterns. These patterns address the problem of allowing different mobile devices to access a desktop or mobile web application.

Table 1: Pattern Thumbnails for improving screen control multichannel access

Pattern Name	Problem	Solution
DIALOG FLOW MANAGER [3]	How do you ensure that nested dialogs can be handled in an intuitive way? For predictability, systems should be able to encapsulate complex tasks and remember how far users have come in performing a task.	Introduce a central dialog manager that is responsible for managing the dialog flow dynamically.
DEVICE SELECTOR (work-in-progress)	There are multiple mobile device types, each representing a different channel, i.e. with different output formats, interaction schemes and network connectivity. How can a system decide which kind of content it has to provide?	Classify the devices using a profile database.
SMALL SCREEN PAGE FLOW (work-in-progress)	Different screen sizes and limits to transmitted file sizes (“deck size”) make it necessary to decompose the original content into many pages. How do you ensure that certain dialog fragments are not separated?	Decide which content is relevant, and which is additional. Define small page flows[1] at design time, i.e. parts of a complete page flow that must or can be presented together.
MULTICHANNEL ACCESS (work-in-progress)	Mobile devices' capabilities range too widely to manually provide and maintain content for all of them. A web application should be accessible via each channel and present content that is tailor-made. How do you provide interpretable masks and the smaller dialog flow for different devices?	Use a profile database to look up device properties, break up the dialog flow into a suitable number of steps and generate the masks afterwards.

[1] The page flow consists of all paths possible that a user can take “walking through” an application.

Pattern Name	Problem	Solution
MODEL VIEW CONTROLLER (first described in [8])	How can data be presented using many different presentation styles?	The three components *model* (data storage), *view* (different ways of content presentation) and *controller* (management of application and presentation logic) have different and strictly separated tasks – to enable many views of one data model (for example a desktop web channel and different mobile web channels).
INVERSE-PYRAMID WRITING STYLE [2]	"People move about quickly on the Web, skimming for information or key words. If a site's writing is not quick and easy to grasp, it is usually not read" [2]	"Start with a concise but descriptive headline, and continue with the most important points. Use less text than you would for print, in a simple writing style that uses bullets and numbered lists to call out information. Place embedded links in your text to help visitors find more information about a related topic. Experiment with different writing styles for entertainment purposes." [2]

7. Bibliography

1. Jing, J., A.S. Helal, and A. Elmagarmid, *Client-Server Computing in Mobile Environments.* ACM Computing Surveys, 1999. 31(2).
2. Duyne, D.K.V., J. Landay, and J.I. Hong, *The Design of Sites: Patterns, Principles, and Processes for Crafting a Customer-Centered Web Experience*. 2002, Boston, MA Addison-Wesley.
3. Biel, B. and V. Gruhn, *Dialog Flow Manager*, in *11th European Conference on Pattern Languages of Programs (EuroPLoP 2006)*. 2006: Irsee, Germany.
4. Cocoon. *The Apache Cocoon Project*. 1999 [cited 2007 30 Apr 07]; Available from: http://cocoon.apache.org/.
5. Book, M., V. Gruhn, and M. Lehmann, *Automatic Dialog Mask Generation for Device-Independent Web Applications*, in *6th International Conference on Web Engineering (ICWE2006)*. 2006: Palo Alto, CA, USA.
6. Goebel, S., et al., *Software Architecture for the Adaptation of Dialogs and Contents to Different Devices*, in *ICOIN 2002*. 2002, I. Chong, LNCS 2344.
7. Spinks, R., et al., *Document clipping with annotation*. 2001.
8. Reenskaug, T., *MODELS - VIEWS - CONTROLLERS.* Xerox PARC technical note December 1979, 1979.