Industrial Experience with Design Patterns

Kent Beck, First Class Software * James O. Coplien, AT&T †
Ron Crocker, Motorola Inc. ‡ Lutz Dominick, Siemens AG §
Gerard Meszaros, Bell Northern Research ¶ Frances Paulisch, Siemens AG ‖
John Vlissides, IBM Research **

Abstract

A design pattern is a particular prose form of recording design information such that designs which have worked well in the past can be applied again in similar situations in the future. The availability of a collection of design patterns can help both the experienced and the novice designer recognize situations in which design reuse could or should occur.

We have found that design patterns: 1) provide an effective "shorthand" for communicating complex concepts effectively between designers, 2) can be used to record and encourage the reuse of "best practices", 3) capture the essential parts of a design in compact form, e.g. for documentation of existing software architectures.

Since the patterns community is one that shares information in an open forum and builds on the experiences of others, we chose to submit a joint paper on our industrial experiences with patterns. We focus on the lessons learned in our respective industrial settings as a first step towards answering the questions "Patterns sound very promising, but how are they actually used in the industry and what benefits, if any, do they bring in practice?"

We proceed by briefly describing each of our respective experiences with patterns. This is followed by a joint "lessons learned" section and conclusion.

*First Class Software, P.O. Box 226, Boulder Creek, CA 95006, USA; E-mail: kentb@ix.netcom.com

†AT&T Bell Laboratories, 1000 E. Warrenville Rd., Naperville, IL 60566, USA; E-mail: cope@research.att.com

‡Motorola Inc., 1501 W. Shure Dr., Arlington Heights, IL 60004, USA; E-mail: crocker@cig.mot.com

§Siemens AG, ZFE T SE 2, D-81730 München, Germany; E-mail: Lutz.Dominick@zfe.siemens.de

¶Current address: Object Systems Group, 250 Sixth Ave. SW Suite 1200, Calgary, Alberta, Canada T2P 3H7; E-mail: gerard@osgcorp.com

‖Siemens AG, ZFE T SE 2, D-81730 München, Germany; E-mail: Frances.Paulisch@zfe.siemens.de

**IBM T.J. Watson Research Center, P.O. Box 704, Yorktown Heights, NY 10598, USA; E-mail: vlis@watson.ibm.com

1 Introduction

Software developers have a strong tendency to reuse designs that have worked well for them in the past and, as they gain more experience, their repertoire of design experience grows and they become more proficient. Unfortunately, this design reuse is usually restricted to personal experience and there is usually little sharing of design knowledge among developers. A *design pattern* is a particular form of recording design information such that designs which have worked well in particular situations can be applied again in similar situations in the future by others. The availability of a catalog of design patterns can help both the experienced and the novice designer recognize situations in which design reuse could or should occur. Such a collection is time-consuming to create, but it is our experience that the invested effort pays off.

A pattern is said to be a "solution to a problem in a context". The basic structure consists of a name for the pattern, a problem statement, a context in which the problem occurs, and a description of the solution together with additional information such as the associated tradeoffs, a list of where this pattern has been applied etc. The form consists of structured prose and sketches (such as OMT diagrams and interaction diagrams). There is general agreement that the pattern identifies a set of "forces" or constraints which are subsequently resolved in the solution.

Design patterns have received a lot of attention lately, especially in the object-oriented community. The reason for the recent interest in design patterns is not the novelty of the designs themselves, but rather the vision that a diverse community of experienced software practitioners, communicating mostly via the internet, can share and collectively grow a set of design repertoires in the form of patterns. The patterns community is sufficiently enthused about the prospective advantages to be gained by making this design knowledge explicit in the form of patterns, that hundreds of patterns have been written, discussed and distributed.

1.1 A brief history of design patterns

Software design patterns had their origin in the late 1980's when Ward Cunningham and Kent Beck developed a set of patterns for developing elegant user interfaces in Smalltalk [5]. At around the same time, Jim

0270-5257/96 $5.00 © 1996 IEEE
Proceedings of ICSE-18

Coplien was developing a catalog of language-specific C++ patterns called idioms [9]. Meanwhile, Erich Gamma recognized the value of explicitly recording recurring design structures while working on his doctoral dissertation on object-oriented software development [16]. These people and others met and intensified their discussions on patterns at a series of OOPSLA workshops starting in 1991 organized by Bruce Anderson [4, 3] and by 1993 the first version of a catalog of patterns was in draft form (summarized in [17]) which eventually formed the basis for the first book on design patterns [18]. All of these activities were influenced by the works of Christopher Alexander, a building architect and urban planner [2, 1] who coined the term "pattern" to refer to recurring designs in (building) architecture. In the summer of 1993, a small group of pattern enthusiasts formed the "Hillside Generative Patterns Group" and subsequently organized the first conference on patterns called the "Pattern Languages of Programming" (PLoP) in 1994 [11].

1.2 The patterns community

The success of the PLoP conference in August '94 and the unveiling of the so-called "Gang-of-Four" [18] book at October '94 OOPSLA created a surge of interest in the topic of design patterns. Due to their basically simple nature, patterns are subject to "overhype", particularly by those who do not fully understand what the real capabilities are or how hard it is to write good patterns. Several mailing lists have been set up by Ralph Johnson at the University of Illinois and this has led to the development of an internet-based community of software developers interested in patterns. A World-Wide-Web site[1] is also maintained at the University of Illinois which serves as a central location for information on patterns. Most of the active members of this online patterns community are practically-oriented experienced software developers and, as such, they are quick to correct any overly-high expectations placed on patterns by newcomers.

The practical nature of patterns themselves and the people writing and using patterns should not be underestimated. As Ralph Johnson once wrote [20]: "One of the distinguishing characteristics of computer people is the tendency to go "meta" at the slightest provocation. Instead of writing programs, we want to invent programming languages. Instead of inventing programming languages, we want to create systems for specifying programming languages. There are many good reasons for this tendency, since a good theory makes it a lot easier to solve particular instances of the problem. But if you try to build a theory without having enough experience in the problem, you are unlikely to find a good solution. Moreover, much of the information in a design is not derived from first principles, but obtained by experience."

Presumably due to the nature of patterns being used to record and reuse **existing** design knowledge, the patterns community has been said to have an "aggressive disregard for originality" [15]. As a concrete example of this, all design patterns in [18] are based on designs which occur in two or more existing, real applications. Perhaps because no one feels like they "own" a particular design, there is a distinct feeling that the members of the patterns community are working towards a common goal in developing a broad collection of patterns as opposed to the competitive nature common to other disciplines (e.g. which person publishes a certain theorem first).

Since the patterns community is one that shares information in an open forum and builds on the experiences of others, it seemed natural to us to submit a joint paper on our experiences with patterns. We focus, in particular, on the lessons learned in our respective industrial settings as a first step towards answering the questions "Patterns sound very promising, but how are they actually used in the industry and what benefits, if any, do they bring in practice?"

2 Industrial experience with patterns

2.1 Smalltalk Best Practice Patterns – Kent Beck (First Class Software)

I have been writing what I intend to grow into a comprehensive system of patterns for Smalltalk programming, called the Smalltalk Best Practice Patterns (SBPP). I'll report here on the status of these patterns and my experience teaching them to and watching them used by two clients developing commercial software in Smalltalk.

The SBPP are intended to accelerate the pace at which teams of Smalltalk developers begin realizing the benefits of objects and Smalltalk by communicating the techniques used by expert Smalltalkers. Although many patterns are still under development, a core set of patterns are finished that cover most of the important design and coding problems.

The best developed section contains 90 patterns for coding. It presents successful tactics for Smalltalk – naming conventions, reuse of the collection classes, common control flow patterns, and code formatting. The emphasis throughout is on communicating through code. The patterns are intended to generate code that meets the simple style rule "say everything once and only once". The section on design has 15 patterns, most of which exist only in outline form. When finished, they are intended to cover similar material to *Design Patterns* [18]. I teach these patterns using presentations similar to "Patterns Generate Architectures" [6]. The section on user interface design has 25 patterns for designing user interfaces and 15 patterns for implementing them in Smalltalk. These patterns are not yet ready to be taught. The final section covers project management. These 30 patterns focus on the non-programming tasks of programmers – testing, documentation, and scheduling.

2.1.1 Hewitt Associates

Hewitt Associates has a group of five Smalltalk programmers working on the next generation of a system implemented originally on large, mainframe computers. They have extensive experience with objects, although the team members have varying levels of familiarity with objects and all are new to Smalltalk.

[1] http://st-www.cs.uiuc.edu/users/patterns/patterns.html

Initially, the team met once a week for several months. As the coding patterns became available, they discussed a few patterns a week. Now that production coding has begun, discussing and learning about programming style is done primarily as part of group code reviews. I spent two days with the team when they started coding seriously. We alternated working on projects with presentation and discussion of the most important patterns.

The resulting code is remarkably good. The most experienced members are making excellent design decisions that I only appreciate after having them explained carefully to me. Even the junior members of the team, new to Smalltalk and objects, are writing idiomatic Smalltalk code. I have noticed the pattern titles becoming part of the spoken vocabulary of the team – "Oh, that's a *Parameters Object*", "We need a *Guard Clause* here".

2.1.2 Orient Overseas Container Limited (OOCL)

OOCL has a much more ambitious effort, with 25-30 developers working to replace centralized applications with a worldwide distributed architecture. The project grew very quickly, which has resulted in some chaos as the team tries to find a common identity and culture.

David Ornstein and I introduced patterns two ways. First, we held two "Smalltalk Bootcamps", where teams of 10-12 develop a simple application from requirements to tested, documented, shipping code in three days. We interspersed discussion of important patterns and software engineering issues with frantic development. The lessons learned here seem to stick very well. In contrast, patterns presented in lecture style were not learned as readily.

An activity we held with some success was a Pattern Bowl. We chose a piece of code to review. We divided the audience (25 developers) into two teams. Each team got points for recognizing the presence or absence of patterns in the review code in a limited amount of time. The winners received guardianship of a token trophy until the next Pattern Bowl. We were happy with the results for two reasons. First, the code in question got a very thorough review. Everyone in the room had a pretty good grasp of what it did and how it did it. You could use a Pattern Bowl to communicate critical shared code. Second, the teams were forced to discuss the meaning of patterns, because there were penalties for mis-identifying a pattern.

Overall, patterns have had a big impact mostly on the early members of the team, five or six bright new Smalltalkers we spent a lot of individual time with. Their designs are sophisticated, their code idiomatic. Later additions, including some experienced Smalltalk programmers, showed reluctance to simply follow the dictates of the patterns, preferring their own style. The unfinished state of the patterns has definitely made teaching them to experienced programmers more difficult.

I have always tried to write my patterns with a substantial section in the middle that presented the motivation for the pattern, why possible alternatives don't work, and led up to the conclusion. OOCL asked me early on to strip all that out, leaving patterns with a name, a problem statement, and a solution. I put together such an abridged version. It has been widely used as a reference and development guide, often being posted on cubicle walls within sight of the workstation.

2.1.3 Conclusion

I have seen the SBPP, even in their half-finished state, have dramatic effects on the quality and quantity of code produced by teams. I am pleased at how the patterns often encourage good code not by admonishing against mistakes, but by presenting a positive set of habits. The effects on communication of adding the names of patterns to the team's shared vocabulary is emerging as a powerful positive force.

Experienced programmers often resist adopting patterns. I suspect the best way to engage developers with strong notions of how things ought to be done is to encourage them to modify and extend the patterns with their own favorite tricks.

Patterns make good projects better. They do not resurrect bad projects. Most of the many things that can go wrong with a project can still go wrong, whether or not patterns are used. Patterns solve a limited (but critically important) set of communication problems with team development, and make individuals more productive. They cannot substitute for effective project management.

2.2 Pattern in AT&T – James Coplien

2.2.1 AT&T patterns programs

There are many independent patterns efforts afoot across AT&T; we touch on just a few of them here.

Fault-tolerant architectures: Patterns capture proven, mature practices in a domain such as building architecture or software design. AT&T has several core competencies that are fundamental to our history of quality customer service. High-availability system design and fault-tolerant software are among these core competencies. Many of these core competencies can be captured as patterns, since they solve a wide variety of reliability and availability problems that arise during architecture and design.

We approached two development communities and asked management to point us to their experts on operations, administration, maintenance and provisioning. This program of "pattern mining" collected dozens of patterns from a handful of experts. We refined these patterns and captured them on-line in HTML[2] where they were made available to the general AT&T research and development community.

Process patterns: We have used patterns in the domain of process and organization, as well as in the domain of software architecture. Patterns are a literary form that conveys a solution to a problem in a

[2]HyperText Markup Language, the publishing language of the World Wide Web.

context: though most practitioners are exploring architectural patterns, there is no reason to limit them to software design. We have found the recurring patterns of outstanding software development organizations through an extensive research program [10]. We can use those patterns to solve organizational and process problems.

Object patterns: Little of our patterns work relates to the object paradigm. Objects are just one way of partitioning systems, and they are not always the best way to organize high-availability or fault-tolerant architectures. Besides, there are many more proven, mature patterns in the architectures of legacy systems than there are in the young, rapidly changing object-oriented systems.

Early work in AT&T to gather proven C++ *programming idioms* has culminated in a collection of widely used programming techniques [9]. One can think of these as proto-patterns; they were in fact one of the foundations from which contemporary patterns practice grew. The seminal *Design Patterns* book [18] built on these and other patterns to provide a general, language-independent collection of patterns by which object-oriented programming competency might be judged. These patterns are seeing wide use in mature AT&T projects. We have steered some young object-oriented projects away from patterns, however. Most new object-oriented projects must learn a design method and a new programming language, in addition to building a new architecture. We have noticed that incorporating more than three significantly new practices in a project increases risk, so patterns are put off until the project masters the initial changes.

2.2.2 How patterns have helped us

Training: We have just started to use the fault-tolerance and high-availability patterns in architectural training. There are two aspects to this training: pattern training per se, and pattern supplements to architectural training. Pattern training is largely for organizations that are "pattern consumers". These organizations are building new projects, using patterns as audits and drivers for design. We have found this training to be effective on many levels. Not only do attendees deepen their understanding of patterns in general and of specific core competency patterns, but they deepen their appreciation for architecture and telecommunications foundations. Most of these courses are conducted as workshops that are highly participatory, with design exercises and pattern-writing exercises. We believe that it is difficult for designers to appreciate patterns fully, unless they have written one.

Some architecture courses are slowly adopting patterns as an adjunct to materials presented in a traditional format. So far, we haven't found this use of patterns to be a significant aid to the learning process. Patterns are probably perceived as a distraction to the traditional educational structures, and we conjecture that pattern-based architecture education might work better if the whole course were pattern-based. We plan further work in this area.

Architecture documentation: In our pattern mining exercise, a new development project was the client for patterns extracted from contemporary projects. When architects from the contemporary project saw the patterns, they saw a solution to a problem that had been plaguing them for some time. Earlier attempts to capture the project architecture had failed to resolve the tradeoffs between a good description of the vertical architecture and architectural layering; patterns provided a way to unify those two perspectives. The original "source" organization is now one of the most active pattern organizations in AT&T, mining its own patterns as architecture documentation.

Shaping New Architectures: By "mining" the fault-tolerant patterns of contemporary AT&T software systems, we can lay the groundwork for emerging and future project architectures. Much support for the emerging patterns work in AT&T came from a new project for which high availability is of paramount importance. The new project is evaluating the fault-tolerance and high-availability patterns gleaned from contemporary systems to see which ones are well-suited to the new system's market and technology.

Requirements Acclimation: Requirements documents draw on market foresight and experience. Most analysts focus on the market foresight of the sales and marketing force, but draw on their personal anecdotes or on review input for the experience component. Patterns provide a written experience base that can feed the requirements process in the following way. As formative projects acquire patterns from their peers and predecessors, they go through them to select those that address problems in the project requirements. Once in a while, a pattern will solve a problem that seems like it *should* be in requirements, but the requirement is found to be missing. Such requirements are added to subsequent editions of the requirements document. We did not foresee this benefit of patterns at the outset, but it has proven to be a valuable use of patterns in new projects.

Process Assessment: We use the process patterns to assess the health of development organizations. Our process research effort receives many requests for process improvement assistance; we use the process patterns as one set of tools to identify and remedy problems. These patterns, which have been published [10], are being similarly used in many companies outside AT&T.

2.2.3 Yet to be done

Designers find individual patterns illuminating and inspirational. We have patterns at all levels, from architectural frameworks down to design patterns and idioms [7]. The number of total patterns numbers in the hundreds. Scale is a major obstacle to systematic and effective patterns usage.

We are currently evaluating pattern organizing schemes, indexing schemes, and other attacks on the

scale of the pattern knowledge base. Bob Hanmer has instituted an indexing scheme where the *Intent* appears as part of the index entry, but not as part of the pattern itself. We are also planning to work with knowledge engineers to help organize patterns according to expected search criteria.

2.3 Design patterns at Motorola – Ron Crocker

Much like AT&T, Motorola has several independent efforts investigating the use of design patterns for system development. Unfortunately, I can only discuss with any substance the effort that I'm involved with[3].

2.3.1 Design patterns vs. software architectures

At Motorola Cellular Infrastructure Group (CIG), recent efforts in applying design patterns to the development process have centered on the relationship between software architecture and design patterns. For some time now, the focus of the systematic improvement efforts at CIG have centered around finding an approach for system development that allows for "large-grained" reuse [13]. Initially, this program focused on the use of object-oriented approaches early in the life cycle, primarily to provide a foundation for this reuse. These attempts were not totally successful. Analyzing these projects indicated some common characteristics that effectively limited any large-grain reuse, including:

- Strong coupling of OO artifacts within a single product
- Short-term needs superseded longer-term needs, even when the benefits were clear.

These findings are not particularly surprising given the strong product-oriented culture of Motorola. However, reaching corporate goals of a factor of 10 improvement in time-to-market requires substantially less work in development – you simply can't do the same amount of work in 1/10th the time.

Enter the centralized software architecture organization, lead by the Strategic Software Technologies organization within CIG [14, 12]. As an organization, CIG has accumulated considerable domain expertise and has some very seasoned software architects. In evaluating several purported software architectures, again we found some common symptoms:

- A lack of preciseness in the specification made them ambiguous.
- The architects developed their own terminology to talk about concepts that we would have immediately recognized had they used "our" vocabulary.
- We did not have direct/immediate access to the architects.

[3] Another effort is documented elsewhere [24].

Each of these problems led directly to communication problems, which lessens the effectiveness of the architecture. Because the architectures are ambiguous, they can be interpreted in ways other than intended. Because the language was "foreign", the ambiguities tend to be amplified and the architectures become product-centric. Finally, questions about the architecture have nowhere to be directed and are hence left unanswered.

Our search for technology solutions turned to design patterns. From previous readings, we knew that design patterns offered an approach for describing architectural entities independent from their implementation. We were concerned about the roots of design patterns coming from the object-oriented community, since our organization has little OO experience. Our approach was to simply not use design patterns in an OO form. We would use design patterns to capture problem-domain-specific entities in an implementation-independent way for sharing across projects (and products).

2.3.2 Current status

So far, we have a small catalog of design patterns focused on (in telephony terms) fault management. There is already an implicit design pattern being used in many of our products for handling faults in the equipment. It's robust and understood by the senior technical staff. The problem with this pattern is that it's only implicit. It exists in the heads of the senior people and in the code. In the cases where we reuse this pattern, the pattern is "rediscovered" from the code and re-implemented, often with minor improvements. None of these improvements, however, affect the basic "higher-order" pattern. These are the sort of patterns that we will be cataloging. Based on some near-term results using the fault management pattern, other problem areas are being identified for "patternification". Our expectation is that these patterns will interact to form a fabric of patterns for telephony.

2.3.3 Pattern applicability spaces

We have a model of the world depicted roughly in Figure 1. We separate the development process into three large "buckets": Products, Problem-Space (entities), and Solution-Space (entities). The Products are implementations of solutions for specific customer use. CIG examples of products would include base stations, cellular telephone switches, and customer database products. Each of these products is rooted in its problem-space entities. Base stations require mobility management capabilities and radio management capabilities. These capabilities tend to be largely independent of both the product itself and implementations of the product. The issues identified above (product-specific nature of OO artifacts and specialized architectural language) have the effect of masking the inherent problem-space nature of these capabilities. The solution-space is where we implement both problem-independent capabilities and product-specific instances of the problem-space capabilities. For example, for the majority of the patterns described in [18] we would consider solely solution-space architectures ("Implementation Architectures") that

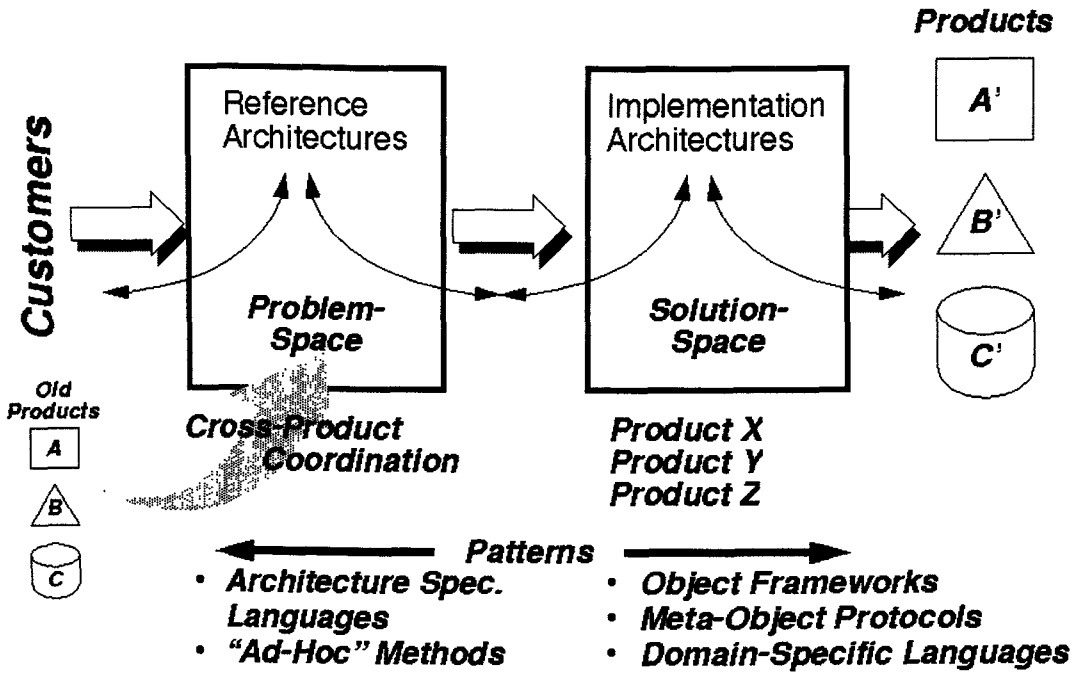

Figure 1: Architectural Spaces

are problem-space independent; other problem-spaces may see those as both problem-space and solution-space patterns.

Each of the spaces has an architectural basis. The Problem-Space architecture we call "Reference Architecture" to indicate that it is not a concrete implementation but rather a guide to developing products incorporating these problem-space entities. We view the critical aspects of these architectures being the definition of the (behavioral aspects of the) entities and their interactions, and therefore focus less on the particular implementation issues. The Solution-Space architectures we call "Implementation Architectures" since their primary focus is on particular instances of products.

This brings us to consider technologies that can aid in describing the architectures in the given spaces. We consider design patterns a technology that spans the spaces, and believe that design patterns represent a technology that can be used to smooth the transition between spaces and final products. Other technologies we have investigated (object frameworks, meta-object protocols, and application-specific languages) tend to reside in the solution-space, as they apply more directly to the issues relating to implementing designs.

2.3.4 Summary

There are two thrusts in our use of design patterns. The first is in using the technology to encapsulate problem-space entities for larger-grained reuse across product families as described above. The other is in using object frameworks and application-specific languages to implement these patterns for easier implementation. Those investigations are on-going and not at a point to report progress. Nevertheless, we have seen some effects of using design patterns in our efforts so far:

- Design patterns have little to do with object-oriented technology. This technology is independent of object-oriented technology. The software systems from which we are extracting design patterns are not object-oriented, and the resulting design patterns are not object-oriented. These design patterns can be implemented using object-oriented designs, but it is not required to be this way.

- Design patterns represent a mechanism for easily sharing design information among groups of architects. We have found that with the design patterns we have written, they have been quickly understood by both the senior architects and the product developers. Other approaches have been less successful in bridging this gap.

- Writing good design patterns is difficult and time-consuming. In our efforts so far, we have spent much time on understanding how to write good design patterns so that they provide enough information to the reader to be useful. Our initial design patterns have gone through many iterations to ensure quality. This implies that only high-value problems should be captured using design patterns, and therefore choosing the appropriate problems becomes an issue.

- It is hard to quantify the impact of design patterns on our development effort. Currently, there are no metrics capable of distinguishing the impact of design patterns from other changes in our development process. Without further efforts on such metrics, we will never know the true benefit of this technology.

2.4 Experiences using patterns at BNR – Gerard Meszaros

At BNR, the research and development subsidiary of NorTel (formerly known as Northern Telecom), we first became aware of the term "patterns" at OOPSLA 1993. We instantly recognized that we had been doing something very similar for quite some time as part of a major re-engineering effort of our DMS-100 family of telephone switches [23]. We have used the "pattern" and similar forms to capture project knowledge in a number of areas. While many of these patterns are specific to our problem domain and form the basis of our competitive advantage, we freely publish the more generic ones in the recognition that we get far more in return for a relatively small investment. The patterns we write and use can be roughly categorized as process/method patterns and technical patterns.

2.4.1 Process/Method patterns

Capturing a design methodology as patterns: As part of developing a new architecture to allow rapid development and delivery of telecommunications services (a.k.a. "Features"), we realized that service developers would require guidance in using the architecture. We began to develop a "service design" methodology. As the "pattern form" was as yet undeveloped, we captured the methodology as a series of "semantic models" starting with requirements and domain model, leading to the architecture model, the design model and finally the implementation model. Specific aspects of each model were identified and the heuristics for transforming them to the related aspect of the next model were captured.

Many of these patterns were "prescriptive" in that they described how to get from one model to another. As an example, a number of the patterns describe how to find and identify similar concepts in different requirements documents and capture the common concepts in the domain model of a service. These patterns effectively are a "recipe" for doing abstraction for people to whom this does not come naturally.

Architecting Method: In the process of re-architecting our call processing system, we have come to recognize a number of key patterns of behavior of architects that lead to good architecture. Many of these patterns are technical in nature. We have captured a number of these in [19] for review and publication at PLoP-95.

The non-technical patterns include ones such as "Just say NO to Politics" (let the project managers solve the question of how the work is divided; architects should concentrate on ensuring that the design decisions are made for technical reasons.)

2.4.2 Technical patterns

We had discovered a number of recurring patterns in the design of telephone services. We had coined terms for many of these, such as *modifier service* (a service which observes another service and adds additional behavior at appropriate points.)

The patterns mailing list on the internet gave us early access to the patterns that were to be published in [18]. We also invited Richard Helm to come teach an introductory course on these patterns. We recognized many of the patterns in our system, often to the point of being able to list our own specializations of the general patterns being described.

We quickly found ourselves expressing our designs in terms of these patterns. They gave us a precise yet concise way of synchronizing our thoughts which saved a lot of effort. No longer did we have to describe a key portion of the design since we had a common understanding of what was meant by "this object is using the *Observer Pattern* to monitor this other object."

Patterns in Software Architecture: We have found patterns to be particularly useful for defining and describing software architectures. Many patterns (*Observer*, *Strategy*, *Composite*, *Half-Object Plus Protocol* to name a few) are particularly useful when defining the the architecture of a system because they encapsulate potential changes to the system. The actual mechanisms used to implement these patterns can vary widely based on cost-space tradeoffs but can be hidden from the core objects (business objects) involved.

2.4.3 Reflections on the BNR experience

Personality Types: Using patterns written by others only takes an open mind; writing patterns takes a special mind! Most people whom we have exposed to the concept of patterns can quickly become proficient at using the common ones. But we have found that only a small percentage of people can write patterns. With respect to patterns, there are three kinds of people: those who see patterns everywhere and can describe them, those who can recognize patterns but can not describe them easily, and those who are oblivious to the pattern surrounding them. This difference seems to stem from a basic orientation of people to focus on similarities as opposed to differences between things.

Impact of Patterns: We have not attempted to measure the impact of patterns on productivity but we have noticed that communication between people with a "shared space" of patterns is quicker, more complete, and less likely to be misunderstood. At the programming level, we have had people design what might be rather complex designs much more quickly than expected by using one or more design patterns.

2.5 Patterns in industrial automation at Siemens – Frances Paulisch/Lutz Dominick

Various operating divisions at Siemens are investigating the effectiveness of using patterns to improve their software production and these activities are coordinated through our department. Many of the software design patterns that are not subject to non-disclosure are being published by our colleagues in [8]. In this section, we focus on our particular project where we are investigating the effectiveness of applying patterns to technologically-oriented applications like the process control of steel mills.

2.5.1 Identifying an initial set of patterns

Our project team, the "pattern mentors", consists of two (software) pattern specialists and two (industrial automation) domain specialists. The first step was to identify potential patterns in interviews with domain experts and then to iteratively refine them (again in consultation with domain experts). In each round we focused on a specific knowledge area. We invited the experts to give a short introductory talk about the solutions they used in their projects and we introduced the notion of patterns. Then we had a discussion to discover the patterns that the projects teams had been using intuitively. Roughly three interviews were required to finish a set of patterns. In their final form, the domain experts agreed that the pattern met our two major criteria of:

- correctly representing the problem-/solution-pair and
- being a useful representation of knowledge demanded by their projects.

In one case two experts initially claimed that their solutions to a similar problem were incompatible with each other, but after seeing the problem-/solution-pair posed as a pattern, agreed that their solutions were indeed very similar.

As an additional "sanity check" we also presented several patterns to experts of a different but related area who had not taken part in the discussion. The level of detail used in the pattern-form was found to be appropriate for providing an understanding of the related areas.

The current state of our project is such that the final proof, the evaluation of the effectiveness of these patterns in concrete steel mill projects, has not yet been achieved, but we are working towards this goal. Our work demonstrates that a small team of people with knowledge of both patterns and of the domain can build up a set of domain-specific patterns which serve as a basis for demonstrating the effectiveness of patterns to the domain experts. Once such a set of essential patterns has been identified and the domain experts have agreed to the effectiveness of their representation in pattern-form, how should one go about extending the set of patterns?

2.5.2 Identifying additional patterns

Ideally, a domain expert should be the pattern author because they have the best knowledge of the domain, but there are several hindrances which must be overcome to accomplish this. The domain experts

- need time to learn what patterns are and how to identify and use them,
- need practice at abstracting away detail and writing patterns, and
- are so tied up in their daily projects that they find it hard to take the first hurdle and actually write patterns.

Although necessary during the introduction of patterns into an organization, it is exceedingly difficult to write patterns, as we did, based on second-hand experience. Doug Lea of SUNY Oswego, who was in a similar situation consulting with avionics engineers developing a set of online design patterns for avionics control systems as part of the Adage project [21], reports that he wrote many of the patterns himself after consultation with domain experts for reasons similar to what we experienced [22].

2.5.3 Making patterns available online

To make the patterns more accessible and attractive to the domain experts, we recorded all of our patterns in HTML in a platform-independent online catalog of patterns. This catalog was organized as a set of three axes which relate to the application domain (in our case the level of automation, the physical structure of the milling machine, and the product-quality features of the milled steel).

The online catalog allows the use of multiple entry points, navigation among the patterns, and a hierarchical structure. The navigation aspect is especially important when the pattern collection grows larger than about 50 patterns which can no longer be linearly organized in book-form. We used links to hide information which is not immediately relevant to the user so that they can see that the information is there if they want it, but are not distracted by it. Furthermore, many terms are connected to an online glossary which resulted from a partial domain analysis of the application area. It is too early to tell how useful this online collection of patterns is to the domain experts, but initial indications are positive.

2.5.4 Initial experience in using patterns

Our initial experience in using patterns indicates that patterns are more likely to be accepted and applied if a significant portion of the design is covered by either a group of low-level patterns or a single higher-level "architectural" pattern. Our users expect some kind of tool support, especially when they are faced with ca. 30 or more patterns. In cases where no appropriate technological design pattern is judged to be ideally suited, the users tend to choose structure-oriented patterns such as "pipe-and-filter" or "layered architecture" over process-oriented patterns.

2.5.5 Future directions

Many of the realizations made within the software reuse community, such as

- the importance of high-level mangement commitment, and
- the effectiveness of making a strict distinction between the teams responsible for developing components and those responsible for identifying and maintaining them

apply equally well to the industrial use of patterns.

We have noticed a strong relationship between the technological design patterns and software design patterns. The technological design patterns we have discovered thus far are planned to serve as the basis for a software application framework for the process-automation of steel mills. Here, we are particularly interested in investigating the interplay between the technological and the software design patterns (e.g. representing the process-control of a conveyor belt as a pipe-and-filter architecture).

2.6 Design patterns in design reviews – John Vlissides

Having served as a consultant to a half-dozen companies, I'm struck by the similarities in what they all try to do. Each project has its unique aspects, certainly, but they are mere variables in a recurring formula. Every project has included a user interface component communicating with some sort of computation component, usually backed by a database. Every project sought to decouple these components to one degree or another. Everyone wanted to use object technology, though not everyone understood why. And while the average experience level varied, every development team struggled with the design process: false starts, iteration, and delays were the norm. These recurrences are mostly beneficial; they let one know what to expect and how to impart the most benefit. But two recurring problems proved troublesome. The following sections describe these problems and how design patterns have helped me deal with them.

2.6.1 Unearthing the design and its rationale

The first of these irritants was the quasi-courtroom tactics I had to adopt to get to the truth of a design. Developers usually had trouble explaining the gist of what they had done, either because they had no means to express it or because they honestly didn't know. I was confronted with one spaghetti class diagram after another. The unstated hope was that I would come to understand the design by sheer osmosis. In reality, there was never time for that.

My only recourse was relentless interrogation. I would ask question after question until I had built up a consistent mental model of the system. Inevitably that would involve backtracking—someone would contradict what was said earlier, causing a partial collapse of my mental model. Sometimes the collapse would come only after we had gone down a series of blind alleys. The more successful attempts along these lines tended to raise more questions than they answered: Why did you design it that way? Is what seems to be gratuitous complexity really worthwhile? What are your assumptions, and why are they realistic? What happens six months from now when I need new capability X?

Which leads me to the second irritant: shallow design rationale. Often the developers simply didn't know why a design was the way it was. No one bothered writing down the reasons for each major change to the design, let alone the incremental ones. As a result, we had to reverse-engineer the design choices time and again—an uncomfortable process for all concerned.

2.6.2 Enter design patterns

After four years of this, things finally began to change when in early 1993 I started incorporating early drafts of material that eventually became *Design Patterns* [18] into my consulting engagements. Rough as that material was, it gave me something concrete to offer in the way of exemplary designs. It also focused my thinking so that I could more readily identify designs based on what the developers were trying to do. No longer did I have to assume that they had developed something entirely new for me to fathom. Instead, I considered the flexibility they were pursuing as a way to isolate a design pattern. Then I could concentrate on mapping the classes they had defined to those in the pattern. If there was some semblance of correspondence, I could feel good about their design and offer constructive criticism immediately. If I could see no correspondence, then I would introduce the pattern to them. Sometimes the flexibility they sought was ill-defined or spurious; the pattern would elude me in those cases. Thus the catalog of design patterns became a kind of sounding board, a test suite for valid design. Of course, this experience helped us refine the patterns themselves.

2.6.3 Sharing design patterns

For all these benefits, though, the burden of pattern application fell largely on my shoulders. The patterns weren't complete or polished enough to give to the development teams ahead of time. I trotted them out as needed, but because they were hard to share with others, they tended to stay confined to my head. Their consummate benefits didn't emerge until the team members could internalize them as well.

That couldn't happen until *Design Patterns* appeared on bookshelves in late 1994. For my first major engagement thereafter, I insisted that each developer read and understand the book prior to our meeting. I had no delusions about this request; I thought few would read it all, let alone understand it. But that was their responsibility, and I expected most people to have at least looked at it.

As it turned out, not only had everyone read it, but a core group (5 out of 12) had a remarkably good grasp of the patterns we discussed. There was also enthusiasm, not just for design patterns but for the developers' own design as well, because they found that they had used some design patterns unwittingly.

Patterns ...	FCS	AT&T	Motorola	BNR	Siemens	IBM
are a good communications medium	√	√	√	√	√	√
are extracted from working designs	√	√	√	√	√	√
capture design essentials	√	√	√	√	√	√
enable sharing of "best practices"	√		√	√	√	√
are not necessarily object-oriented		√	√	√	√	
should be introduced through mentoring	√				√	√
are difficult/time-consuming to write			√	√	√	
require practice to write					√	√

Table 1: Sources for Summary Observations

Seeing the design patterns was a vindication of sorts—it legitimized approaches they had been unsure of.

2.6.4 The biggest payoff: communication

But the best part of the encounter was the high level of communication we achieved. We discussed designs not in terms of classes and objects and methods but to a great extent in terms of design pattern concepts: participants, applicability, consequences, trade-offs. Discussion remained at the design pattern level unless and until there was a controversy, at which point we might drop down to the nuts and bolts. But that was infrequent. I'm happy to report that pattern concepts dominated our discussions.

In fact, I came away from this engagement feeling a satisfaction I hadn't felt after any other, and I attribute it unreservedly to the use of design patterns by *all* concerned, not just myself. Another engagement along these lines has been scheduled for this fall. The project under review will be a different one, with another, somewhat larger development team at its helm. As a further twist, several of the team members from the earlier engagement will participate. They will act as I did, but to small subgroups of the overall team. That will help spread the burden and hopefully permit even more incisive discussions.

3 Lessons learned

Despite our diverse backgrounds and experiences, several common lessons can be drawn from our own experiences with patterns as well as from our colleagues in the patterns community. Table 1 shows the sources of the summary observations listed below. Although in some cases it is difficult to give a binary answer, checkmarks indicate that this company has made this experience. Unfortunately, we do not have any measurable data on the impact of patterns available yet, at least not in a form we could currently publish. But the consistency among our experiences with patterns, leads us to believe in the value of lessons listed here.

Patterns serve as a good team communications medium. Typically, when several pattern-aware software developers are discussing various potential solutions to a problem, they use the pattern names as a precise and concise way to communicate complex concepts effectively.

Patterns are extracted from working designs. Each design pattern discussed above was extracted from existing, working designs (and in the case of the organizational patterns of AT&T, from existing organization) and not created without experience. The design patterns capture the essence of working designs in a form that makes them usable in future work, including specifics about the context that makes the patterns applicable or not.

Patterns capture the essential parts of a design in a compact form. This compact representation helps developers and maintainers understand and therefore not contort the architecture of a system. Making this often only implicitly understood knowledge explicit allows for more effective software development.

Patterns can be used to record and encourage the reuse of "best practices". This is especially important for helping less-experienced developers produce good designs faster. A collection of design patterns in handbook-form is useful for teaching software engineering. However, note that, in partial contrast to handbooks from other engineering disciplines, a design pattern is not a rule to be followed blindly, but rather it should serve as a guide to the designer and/or provide alternatives when being applied to a particular situation.

Patterns are not necessarily object-oriented. Although the design patterns as we describe them come from the object-oriented community, there is nothing inherent in design patterns that makes them

object-oriented. Not coincidentally, there is nothing inherent in object-oriented programs that make them candidate sources for design patterns. Our experiences have shown that design patterns can be found in a variety of software systems, independent of the methods used in developing those systems.

The use of pattern mentors in an organization can speed the acceptance of patterns. Pattern mentors can help provide a balance between encouraging good design practices based on patterns and discouraging overly high expectations of designs based on patterns. Initially, pattern mentors can help developers recognize the patterns that they already use in their application domain and show how they could be reused in subsequent projects. Pattern mentors should also watch that the wrong patterns are not applied to a problem (i.e. people tend to reuse things that they know and the same temptation will apply to patterns, regardless of whether the pattern actually fits the problem)[4].

Good patterns are difficult and time-consuming to write. Writing good patterns is a skill that does not come easy. Furthermore, the writing of a pattern typically involves an iterative process in which the pattern is presented to others and/or applied in projects, relevant comments are incorporated, and the process repeated until the result is adequate. However, we have found that, as one gains experience at writing patterns, the effort for recognizing and writing them is reduced.

Pattern practice is of utmost importance. After the initial phase of learning about patterns by seeing many good examples, one comes to appreciate the true value of patterns best from recognizing and writing them oneself.

4 Conclusions

In our joint experience, we have seen that the use of patterns can have a dramatic impact on the way a team develops software. The improved communication through patterns alone is a valuable asset. Giving novices the opportunity to learn from positive examples which already form the basis of a shared team vocabulary can help speed their contribution to the team. On the other hand, good patterns are hard to write, especially for those developers to whom abstraction does not come naturally. It is difficult to find a balance between the advantages and disadvantages, especially when measurable results are not yet available. It is clear that many people in the software engineering community recognize the emergence of patterns, but only few have had any opportunity, until now, to learn about their benefits and drawbacks in practice.

[4] "To someone with a hammer, everything looks like a nail."

References

[1] Christopher Alexander. *The Timeless Way of Building.* Oxford University Press, New York, 1979.

[2] Christopher Alexander et al. *A Pattern Language: Towns, Buildings, Construction.* Oxford University Press, New York, 1977.

[3] B. Anderson and P. Coad. Patterns workshop. In *OOPSLA'93 Addendum to the Proceedings*, Washington, D.C., January 1994. ACM Press.

[4] Bruce Anderson. Towards an architecture handbook. In *OOPSLA Addendum to the Proceedings.* ACM Press.

[5] Kent Beck. Using a pattern language for programming. In *Addendum to the Proceedings of OOPSLA'87*, volume 23,5 of *ACM SIGPLAN Notices*, page 16, May 1988.

[6] Kent Beck and Ralph Johnson. Patterns Generate Architecture. In *European Conference on Object-Oriented Programming (ECOOP)*, 1994.

[7] Frank Buschmann and Regine Meunier. A system of patterns. In James O. Coplien and Douglas C. Schmidt, editors, *Pattern Languages of Program Design.* Addison-Wesley, 1995.

[8] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal. *Pattern-Oriented Software Architecture: A System of Patterns.* John Wiley and Sons, 1996. (in preparation).

[9] James O. Coplien. *Advanced C++: Programming Styles and Idioms.* Addison-Wesley, 1992.

[10] James O. Coplien. A generative development-process pattern language. In James O. Coplien and Douglas C. Schmidt, editors, *Pattern Languages of Program Design.* Addison-Wesley, 1995.

[11] James O. Coplien and Douglas C. Schmidt, editors. *Pattern Languages of Program Design.* Addison-Wesley, 1995.

[12] R. Crocker and J. Engelsma. Continuing investigations into an organizational-wide software architecture. In *ICSE-17 Workshop on Software Architecture*, April 1995.

[13] R. T. Crocker. Reaching for '10X' improvements - why OO isn't the answer! In *Proc. of 10th International Conference on Advanced Science and Technology*, pages 91–96, March 1994.

[14] J. Engelsma and G. P. Saxena. Building competence in software architecture at Motorola's Cellular Infrastructure Group. In *OOPSLA '94 Workshop on Software Architectures*, Oct. 1994.

[15] Brian Foote. quoted during the PLoP '94 conference (see [CS95]), 1994.

[16] Erich Gamma. *Object-Oriented Software Development based on ET++.* PhD thesis, University of Zurich, Institut für Informatik, 1991. (in German). Also available through Springer-Verlag, Berlin, 1992.

[17] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. Design Patterns: Abstraction and Reuse of Object-Oriented Design. In O. Nierstrasz, editor, *European Conf. on Object-Oriented Programming (ECOOP)*, Kaiserslautern, Germany, July 1993. Springer Verlag, LNCS 707.

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

[19] Allen Hopley. Levels of abstraction. In *Pattern Languages of Programming Conference*, 1995.

[20] Ralph E. Johnson. Why a conference on pattern languages? *Software Engineering Notes*, 19(1):50–52, January 1994.

[21] Doug Lea. Design patterns for avionics control systems. Available through WWW site http://st-www.cs.uiuc.edu/users/patterns/patterns.html, 1994.

[22] Doug Lea. personal communication, 1995.

[23] Gerard Meszaros. Software architecture in BNR. In David Garlan, editor, *Proc. of 1st Intl. Workshop on Architectures for Software Systems*, 1995. held in cooperation with ICSE-17.

[24] D. Schmidt. Experience using design patterns to develop reuseable object-oriented communication software. *Communications of the ACM*, October 1995.