Understanding and Using Patterns in Software Development

Dirk Riehle and Heinz Züllighoven

UBILAB, Union Bank of Switzerland.
Bahnhofstrasse 45, CH-8021 Zürich, Switzerland
E-mail: riehle@ubilab.ubs.ch

University of Hamburg, Germany.
Vogt-Kölln-Straße 30D, D-22527 Hamburg, Germany.
Heinz.Zuellighoven@informatik.uni-hamburg.de

Abstract

Patterns have shown to be an effective means of capturing and communicating software design experience. However, there is more to patterns than software design patterns: We believe that patterns work for software development on several levels. In this paper we explore what we have come to understand as crucial aspects of the pattern concept, relate patterns to the different models built during software design, discuss pattern forms and how we think that patterns can form larger wholes like pattern handbooks.

1 Introduction

Design patterns have become a hotly discussed topic in software development. We and many other researchers have been using and experimenting with patterns over the last years. We have applied patterns and observed their usage within software development. We have used and seen several definitions of patterns, and we have experimented with pattern forms. The emerging literature shows a flourishing and fruitful diversity of pattern definitions, forms and applications. But still the question remains, if and to what extend patterns will become an established concept.

In this paper, we summarize our experiences. We present what we perceive to be crucial characteristics of a pattern and we argue that it is important to distinguish between different pattern types so that they can be related properly to the main models built during software development. We then analyze different pattern forms used to describe and present patterns based on their intended use. In closing, we propose the integration of patterns into what in our opinion should be a pattern handbook.

We present work on patterns in a survey style. Our own experience is based on the development of interactive software systems both in a research and in an industrial setting (Bäumer et al., 1996). It has been consolidated as a coherent approach called the Tools and Materials Metaphor. We use patterns from this approach (Riehle & Züllighoven, 1995) and from the seminal work of Gamma et al. (1995) to illustrate the arising issues.

Section 2 presents definitions of the term pattern and discusses different aspects which we perceive to be important. Section 3 distinguishes between different pattern types and relates them to the models built during software development. Section 4 discusses pattern presentation forms. Section 5 discusses our experiences when dealing with pattern sets and how we order them. Section 6 reflects about presenting pattern sets as a larger whole, that is a pattern handbook. Section 7 shortly summarizes related work and section 8 draws some conclusions to round up the paper.

2 Pattern Definitions and Characteristics

In this section, we present our definition of the term pattern, shortly review other definitions and then discuss properties of patterns which we consider to be relevant for software development.

2.1 Pattern definitions

What do we mean when we use the word "pattern?"

A pattern is the abstraction from a concrete form which keeps recurring in specific non-arbitrary contexts.

This definition is deliberately more general than other definitions found in the literature. Thus, it does not confine our notion of pattern to software design, and it is not restricted to a specific use of patterns. The by now

To be published in *Theory and Practice of Object Systems* 2, 1 (1996).

famous definition of pattern as "a solution to a recurring problem in a context" is geared towards solving problems in design. We avoid this specialization and take up this issue in section 4 where we discuss different presentation forms for patterns based on their intended use.

What can be found in the literature?

Alexander[1] (1979, page 247) writes: "Each pattern is a three part rule, which expresses a relation between a certain context, a problem, and a solution." He goes on to explain that a pattern is a relationship between forces that keep recurring in a specific context and a configuration which resolves these forces. In addition, a pattern is also a rule that explains how to create the particular configuration which resolves the forces within that context.

This "pattern as a rule" definition has been highly influential. Following Alexander, Gamma et al. (1995, page 2-3) define a pattern to be the solution to a recurring problem in a particular context, applicable not only to architecture but to software design as well. Concrete patterns are expressed using classes and objects, but they nevertheless represent solutions to problems in particular contexts. This is the most widely spread notion of pattern today, and it has been adopted by many other researchers, for example by Beck & Johnson (1994), Schmidt (1995) and Buschmann et al. (1996). Coplien has been very successful applying the same notion of pattern to organizational issues arising in software development projects (Coplien, 1995; Coplien, 1996).

An alternative definition is given by Coad. He originally draws on a dictionary, defining a pattern to be "a fully realized form, original, or model [...] for imitation" (Coad, 1992). Specializing this for object-orientation, he defines the notion of pattern for object models as "a template of objects with stereotypical responsibilities and interactions" (Coad et al., 1995).

2.2 Form and context

To gain a better understanding of our definition of pattern, we next elaborate what we mean by "form" and "context." We defined the notion of pattern to be the abstraction from a recurring concrete form. What does such a form look like?

The form of a pattern consists of a finite number of visible and distinguishable components and their relationships.

Essentially, we define the notion of form through its representation as a set of interacting components and their relationships. Thus, a pattern has both structural and dynamic properties. A component need not to be a software component, but can be any kind of technical or non-technical entity. Definitions of specialized pattern types will characterize more precisely which components and relationships are used.

Take an example: When analyzing an application domain we try to identify those objects of the domain which can be interpreted as either tools or materials. These are two types of components. The relevant relationship is that of tools working on appropriate materials. This is the pattern of *the Distinction of Tools and Materials.*[2]

A pattern is used to create, identify and compare instances of this pattern.

A pattern is first of all a mental concept derived from our experience. This experience and our reflection on it lead us to recognize recurring patterns, which, being established as a concept of their own, guide our perception. We then use patterns to perceive and recognize instances of patterns. Essentially, patterns only materialize through their instances. As they restrict our perception, patterns guide our way of interacting with the world. Patterns are relevant both in the "real world" and in software design.

Back to our example, the pattern of the Distinction of Tools and Materials helps us to *recognize* concrete instances of it. So a pencil is seen as a tool used to fill in a form as its material. Using the form of the pattern as a template helps us to identify and compare concrete instances of it. In this way, the pattern serves as an *analysis pattern* for interpreting and understanding an application domain.

A pattern can also be used to *create* instances of it. In software design, the general distinction between tools and materials serves as a (high-level) software design pattern which will lead to a software architecture where tool objects work on material objects. It is important to realize that we thus can closely link application domain concepts to software architecture. The architectures of the systems we build (Bürkle et al., 1995; Riehle et al., 1996) heavily make use of this pattern.

---

1 Christopher Alexander is the architect who has explored the concept of pattern in the domain of architecture. His work has had a strong influence on work on patterns in computer science. This can be seen from the name of the premier conference of the field, called "Pattern Languages of Programming." Alexander coined the term pattern language (Alexander, 1979).

2 The notion of tool or of material itself is not of a pattern but of a metaphorical nature. These terms were originally borrowed from handcrafts and philosophical reflection on it (Budde & Züllighoven, 1992).

Pattern instances appear only in specific contexts which raise and constrain the forces that give birth to the concrete form.

A pattern is a form that appears in a context. The relevance of this distinction can be illustrated by drawings of MC Escher or others used in psychology, where the perception of what is the context is needed to establish the form in the "foreground." Pattern and context are complementary: A context constrains the pattern and gives birth to its form while the form fits only into certain contexts. Changing the context will change the form and changing the form has to go hand in hand with changing the context.

Both form and context of a pattern are abstractions derived from our concrete experiences. Applying a pattern means bringing it into a new concrete context of use. The forces of this use context have to fit the form of the pattern. Otherwise we have a mismatch leading to potentially costly adaptation work.

The context for the pattern of tool and material distinction is expert human work. Tools are designed to offer great flexibility and ease of use for coping with the tasks at hand while all the necessary materials have to be there to produce the desired outcome of work.

The form of a pattern is finite, but the form of its instances need not to be finite. The context is potentially infinite.

The form describing a pattern consists of a finite number of components and relationships. A concrete instance, however, may have an open number of components which are defined recursively and created on demand. An example is the *Chain of Responsibility* pattern (Gamma et al., 1995) which consists of the roles Client, Handler and Successor. The pattern form is finite, while concrete instances may lead to a chain of successors of arbitrary length.

In addition, the context is potentially infinite, which means that in the general case the relevant constraints and forces driving the form cannot be described in a finite list. This result is relevant to formal approaches of applying patterns. A method or a tool which claims to determine the right pattern given some finite input, that is a set of forces or requirements, has to prove in advance why the context of the pattern can be described sufficiently well by this input so that it is possible to make such a decision.

We think that it is possible in many situations to pragmatically enumerate the relevant forces based on our experience and omit what we understand to be less relevant forces. The choice, however, is never deterministic then.

As a consequence, a pattern can only be understood and properly used with respect to the background of experience and reflection in the domain of its context.

Patterns emerge from experience and can only be applied based on proper experience. This is a consequence of the context being infinite and informal in general, since every description of the context lists only what the originator of the pattern perceives as being relevant. This can be of great help for the beginner with little experience. But he or she will be in trouble if the pattern is applied in a new context which was not foreseen by its originator. Transferring patterns to a new context requires experience and insight into both its original and the new context.

As another consequence, the form describing a pattern can be formalized, but not its context.

It is obvious, that patterns as a finite form can be described in a crisp way (we will come to that in section 4). It can be expected that proper formalizations of patterns will be found. Early work on this topic has been carried out, for example, under the label of behavioral specification (Helm et al., 1990). Recent formalizations which have been successfully applied to patterns include the work of Lieberherr et al. (1994).

The context cannot be completely formalized since it is infinite and therefore largely informal.

A pattern should be presented in a way that it can be understood and properly used by others in their work and professional language.

Even if one does not restrict the meaning of the term pattern to software design, the use aspect of pattern is relevant. We argue that a pattern should be useful for improving its originator's work. But a pattern shows its potential only when it has been accepted and (re-) used by others in the profession. Then, it has been proven as an abstractable form for dealing with similar problems in recurring contexts. And only then, patterns can enter the professional language as terms.

2.3 Consolidated example

As mentioned, the central pattern our approach is the *Distinction of Tools and Materials*. This pattern says: "When looking at an application domain for developing a software system, separate tools from materials and analyze the ways tools are used to work on materials." Figure 1 illustrates some typical tools and materials in the office domain.

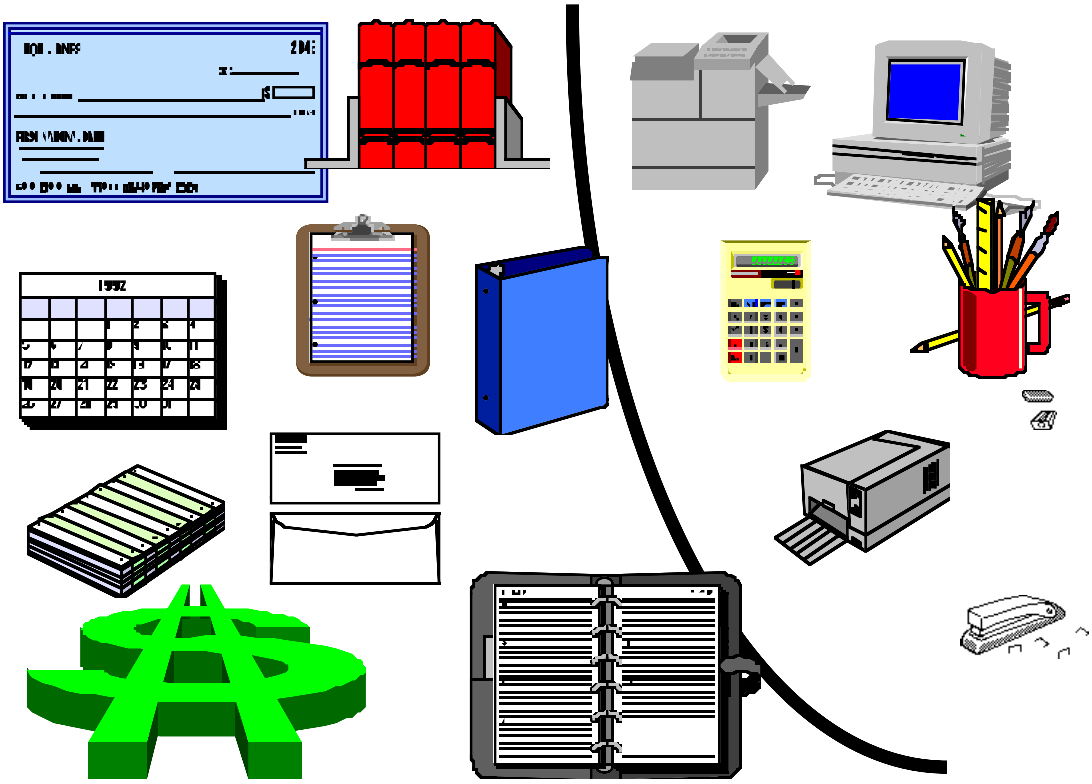

Figure 1: An exemplary collection of tools and materials in the office domain. Typical materials are shown on the left and tools are shown on the right.

The static structure of this pattern consists of two main components, tools and materials, and one relationship between them, the work task. Dynamically, we use a tool to work on a material. The tool presents the material we are working on, with the focus on the current work task, thereby highlighting certain properties of a material and ignoring others.

We find this pattern recurring frequently in office work and workshop-like settings, where people apply tools as means of work on materials to produce some result in order to fulfill their work tasks. Thus, we identify instances of this pattern when a bank representative uses a form sheet (a material) and a specialized calculator (a tool) while consulting a customer in order to sell potential loans (the result). Another example might be an electronic phone (the tool) in a telephony system which is applied to a phone list (a material) in order to ring up customers for introducing a new product (the result). So, this pattern helps to solve one of the main problems of software design, namely to identify those objects which are relevant to the system (cf. Jacobson et al. (1992)).

The context of this pattern are work situations that require skills and knowledge to perform the tasks at hand. It works best, when a high level of flexibility and ease of handling is required. Since tools are geared towards individual work styles and habits in changing work situations, this pattern does not fit well for routine and highly standardized work. This type of context has forces which call for different patterns, like that of an automaton working on materials.

The Distinction of Tools and Materials pattern is not only found in many application domains but is familiar to software developers as well. Developers usually use a large number of tools, for example browsers, text retrievers or cross referencers, to work on their materials, for example source code or its higher level representations. Other examples are project editors (tools) with project configurations (materials) or debuggers (tools) with running or halted software systems (materials).

Patterns like the Distinction of Tools and Materials are more than a guide in analyzing an application domain or a construction aid when designing software. They have entered the language of our software teams and are the terms we use when describing and discussing software designs (cf. Schmidt (1995)). So, patterns have become part of the project culture, and a new developer joining the team has to assimilate these terms and the concepts behind them to arrive at a level of understanding where he or she can actively use and revise these patterns.

3 Patterns and Models

Having presented our basic understanding of the concept of pattern, we will now explore its variants and show how they relate to the different models built during software development.

Software engineering distinguishes between three main models: an application domain model, a software design model and an implementation model, the last two being abbreviated as software design and implementation.

We think that it is a great advantage to relate different types of patterns to each of these models. These pattern types are defined in terms of the more abstract definition given above, so they share its properties.

3.1 Conceptual patterns

In order to design something useful, we need a conceptual model of the application domain that evolves with our system (Jackson, 1983; Greenspan et al., 1994). Such an application domain model doesn't have to be formal, rather it has to be understandable by all parties concerned. Usually, it consists of a set of related descriptions based on the concepts and terms of the application domain, comprising the different viewpoints of the various groups involved in the software development process. Thus, a conceptual model should use the language of the application domain. As viewpoints are always related to personal beliefs and values, these models cannot be proved correct in a formal way. They have to be comprehensive and are subject to discussions and negotiations about what is and what should be developed. So, the right choice of patterns should relate the language used in an application domain to the terms and elements used in the conceptual model of this application domain. We call this type of pattern a conceptual pattern:

A conceptual pattern[3] *is a pattern whose form is described by means of the terms and concepts from an application domain.*

Conceptual patterns guide our perception of an application domain. They help us to understand the domain and the tasks at hand. Additionally, they provide the concepts and language to discuss our understanding of the application domain with experts and potential users. We envision future systems and situations of work by mentally constructing them using the conceptual patterns.

Thus, conceptual patterns comprise both a kind of world view and a guideline for perceiving, interpreting and changing the world.

This link between the "real world" and the conceptual model can be strengthened. So, it is important not just to use terms of the application domain's language but to carefully select metaphors. These metaphors, as understandable "mental pictures", are supportive when taking the step from the situation at hand to the design of the future system (Carol et al., 1988). Linking metaphors like *tool* and *material* to patterns like the Distinction of Tools and Materials helps to bridge the gap between application domain and software design. Therefore:

Conceptual patterns should be based on metaphors rooted in the application domain.

Conceptual patterns don't serve a general purpose – they don't fit any conceivable context. We always have to find the balance between too abstract and too specialized patterns and contexts. If a pattern is too abstract, it is too general to really guide analysis and design. So, "active collaborating object" may be a pattern and even a metaphor, but it is too general to be useful. On the other hand, a very specific pattern might not be used beyond a single project and therefore will not become part of everyday practice and a development culture. The Distinction of Tools and Materials pattern is on the right level of abstraction for workplace computing, but according to our experience it is not useful for real time software. Therefore:

Conceptual patterns should be geared towards a restricted application domain .

Doing so, they have to be on the right level abstraction striking the balance between being too generic and too concrete. In this respect, the Distinction of Tools and Materials pattern can be compared with other conceptual patterns and metaphors like agents or media (CACM, 1994).

3.2 Design patterns

Looking at the activities related to the technical design of a system, we need a model which relates to the conceptual models of the application domain but takes into account the need for reformulating this conceptual model in terms of the formal restrictions of a software system. This is the traditional software design model. It is geared towards software construction:

[3] Conceptual patterns have originally been called "Interpretations- und Gestaltungsmuster" (interpretation and high-level design patterns) in Riehle (1995). For reasons of brevity this has been shortened to conceptual pattern.

A design pattern is a pattern whose form is described by means of software design constructs, for example objects, classes, inheritance, aggregation and use-relationship.

We use software design patterns to build and understand a software design model. A design pattern describes the structure and dynamics of its components and clarifies their interplay and responsibilities. This definition addresses the whole scale of software design ranging from software architecture issues (IEEE, 1995) to so-called micro architectures (Gamma et al., 1993). Here, we see a close connection between design patterns and frameworks. A framework should incorporate and instantiate design patterns, in order to "enforce" the reuse of designs in a constructive way (Beck & Johnson, 1994).

We think that it is important to have as little semantic difference between the conceptual model and the software design model as possible, at least for the application-related "core" of the design model. This leads to the following rationale of software design models:

Design patterns should fit or complement the conceptual space opened by the conceptual patterns.

Software design is facilitated considerably, if design patterns can be related to the conceptual patterns used to describe the application domain model, that is if they help to realize the conceptual patterns and metaphors on the concrete design level.

3.3 Programming patterns

In constructing the software system, we bring together the application domain with the software design model. This results in a system implementation, the third relevant model. Implementations are technical artifacts which run as a formalism on a computer and can be utilized as an application supporting its users. Programming languages provide the notation for this model. On this level, we find a third type of pattern:

A programming pattern is a pattern whose form is described by means of programming language constructs.

We use these patterns to implement a software design. It is based on programming experience. Programming patterns vary among programming cultures (cf. Coplien's idioms (1992)). An example of an implementation pattern is the well-known C style loop

```
for (i=0; i<max; i++) {...}
```

found in almost every C program.

It is important to realize that even on this rather low technical level there is something like programming cultures and styles. It goes beyond the scope of this paper to detail this discussion. As an example take the pattern of Procedure/Function/Predicate interface design for classes as proposed by Meyer (1988). This conforms to a specific view of designing software called Design by Contract (Meyer, 1991).

3.4 Model and pattern interrelationships

All three types of patterns should be brought together in a coherent approach. This means first of all linking the right type of pattern to the respective model: We carry out application domain analysis and high-level system design using conceptual patterns. We develop software designs using design patterns. And we implement software systems using programming patterns.

The models are related to each other via their patterns: Conceptual patterns are a high-level view that has to be substantiated in any actual design through the use of design patterns. Design patterns have to fit the context set up by the conceptual patterns. In Riehle & Züllighoven (1995) we presented the conceptual patterns of the Tools and Materials Metaphor approach and supplemented them with several software design patterns that make the conceptual patterns concrete in every software design.

The relationship between conceptual patterns and software design patterns can again be illustrated using the conceptual pattern of the Distinction of Tools and Materials. On a software design level, it directly leads to the design pattern of Tool and Material Coupling as illustrated in figure 2: A tool object is coupled with a material object via an aspect class. An aspect class represents a specific way to work on a material and expresses this through behavior and abstract state declared in a class interface. It should be derived from the intended work tasks. A material class inherits from all those aspect classes that capture the possible ways of working on this material.

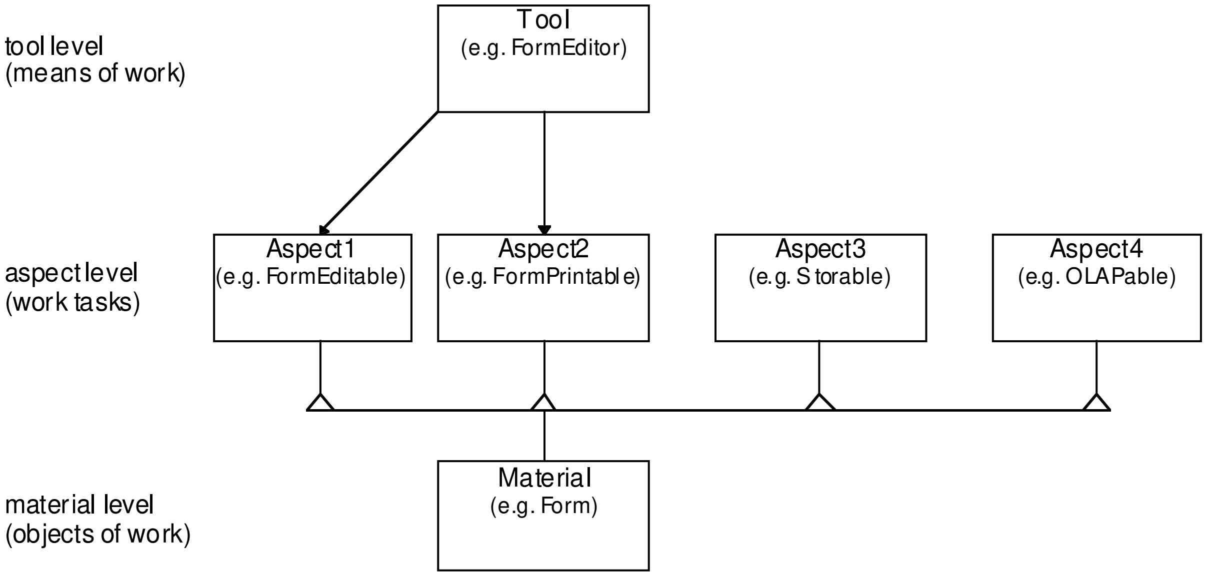

Figure 2: An example for the application of the software design pattern Tool and Material Coupling. A FormEditor tool works on a Form material using one or more aspect classes offered by the material.

4 Pattern Description Forms

Hoare once remarked that every abstraction needs a notation to give it a form (Hoare, 1972). This holds for true for patterns as well. Most researchers seem to agree that patterns should be presented in a structured form. This is what Alexander et al. (1977) did and what is apparent in Gamma et al. (1995) and Coplien & Schmidt (1995). Thus, we have tested and analyzed several pattern forms.

The most important result is that the best way to describe a pattern depends on the intended usage of the pattern. This may not be surprising at first, but it is a conclusion that does not seem to have drawn much attention to it yet. From the literature and our work, three pattern forms stand out.

4.1 The Alexandrian form

In Coplien & Schmidt (1995) we and many other researchers chose a form of presentation which consists of at least the three sections *Problem*, *Context* and *Solution*. The Problem section describes concisely the problem at hand, the Context section describes the situations in which the problem occurs as well as the arising forces and constraints for a possible solution, and the Solution section describes how to resolve the forces within that context.

The intended use of this pattern form is to guide users to generate solutions for the described problems.

We follow Berczuk (1994) in calling this the Alexandrian form, since it is the basic form proposed by Alexander et al. (1979). Characteristic for this description form is its generativity. The solution section is usually written in such a way that it guides the actual instantiation of the pattern for concrete problem solutions. For example, Buschmann et al. (1996) present explicit step by step instructions for implementing their patterns.

Beck & Johnson (1994) write "that patterns can be used to derive architectures much as a mathematical theorem can be proved from a set of axioms." They demonstrate this generative power to motivate and explain the architecture of HotDraw, a graphical editor framework, and show how the patterns help them doing so.

4.2 The Design Pattern Catalog form

Gamma et al. (1995) use a structured form for presenting design patterns. They use a template which has been tailored to the description of object-oriented software design patterns.

The template consists of a number of sections, each one describing a certain aspect of the overall pattern. Most of the attention is paid to describing the structure of the pattern, both static and dynamic aspects, and the various ways of using it. The structure and its dynamics are what we have called the form of a pattern. The motivation, applicability and consequences sections try to capture the context of the pattern.

The intended use of this pattern form is also to help users create solutions to problems. This time, however, they focus less on when to apply the pattern but more on the actual structure and dynamics of the pattern itself.

Thus, this pattern form is more descriptive rather than generative. However, as Beck & Johnson (1994) point out, the patterns from Gamma et al. (1995) could be easily rewritten to be more generative. This ultimately makes clear that the pattern form is to be distinguished from the pattern itself. Rather it is subject to the intended use of the pattern.

We have found the Design Pattern Catalog template to be an excellent choice for presenting object-oriented design patterns which are well understood and can stand on their own. It covers all relevant issues we have come to think about. It strikes a good balance between the rather general pattern/context form discussed below, and the more specialized Alexandrian form.

Though Gamma et al. think of their patterns as solutions to problems in design, it has not lead them to use the Alexandrian form. Their structure and dynamics sections are presented in such a way that their pattern description can be used both to create concrete instance of the pattern as well as to recognize the patterns in existing systems.

4.3 A general form

Based on the considerations of section 2, we have experimented with a rather general pattern form which consists of the two sections *Context* and *Pattern*. The Context section describes the context, constraints and forces that give birth to the pattern. The pattern itself is described in the Pattern section, which presents the form the pattern takes on in the discussed context.[4] Riehle (1996) is a good example of the application of this form.

The intended use of this description form is to discuss the structure and dynamics of the recurring form and its context without promoting a specific way of using the pattern.

This rather general description form is based on our understanding of patterns as finite forms that emerge in specific contexts, so we separate context and form and view it as a duality rather than a single entity. This form has no generative power per se, but can be used for different purposes. It can be used to create instances of the pattern, or it can be used as a template to match and recognize instances of the pattern in existing systems, or it can be used to compare similar patterns, etc.

This general applicability comes with a price: Since the pattern description tries to abstain from a particular use of the pattern, it is less useful for a specific application than a pattern described in a way to support a particular use. We therefore supplement the general description form with additional sections that discuss how to use the pattern for a particular purpose. Riehle (1996), for example, supplements the general form with sections on design and implementation to let developers apply the patterns more easily as solutions to recurring problems.

4.4 Comparison and discussion

We believe that the pattern/context pair form is suitable for pattern descriptions in general. If the descriptions are to support a specific pattern use, they have to be supplemented with additional sections discussing this application.

If a particular way of using or applying the pattern is to be emphasized, more specific forms might be preferable. The Design Pattern Catalog form, for example, is well suited to describe object-oriented software design patterns, without overly emphasizing the generative potential of patterns.

We do not claim that one form is superior over the other, rather, that they have different intents. The Design Pattern Catalog form may be the first choice for object-oriented software design patterns of general applicability, the Alexandrian form may best be suited for matching problem situations and describing how to solve them, and the pattern/context pair form may be the form of choice for general presentations in which a pattern description core is to be adapted to different use situations.

5 Pattern Sets

We now have to go beyond single patterns and focus on possibly large collections of patterns. The aim here is twofold: On the one hand we need to structure these large collections in order to ease their understanding and usability. On the other hand we want to restrict the design space for the various types of software systems (cf. Jackson (1994)).

---

4 We could have called the Pattern section Form section as well.

This section addresses ordering and presentation of patterns into sets. It introduces the notion of background from which patterns emerge. The background is captured by what we call a leitmotif, that is a shared vision incorporating the views, believes and values of software developers that shape a software system as a whole.

5.1 Ordering patterns

Mature patterns are not isolated but relate to other patterns. This relationship may either be an embedding or an interaction. So we need to find ordering schemes and relations to arrange and describe sets of related patterns.

Each pattern is always to be viewed within its context. While the description of the context should be kept together with the description of the pattern, it is not part of the pattern itself. Pattern and context are complementary. In order to understand such a pattern/context pair, we need to understand the range of its applicability, which is another context. Again, this context may be captured by another more coarse-grained or more abstract pattern. A hierarchy of pattern/context pairs emerges that can be presented as a directed graph.

Thus, we order patterns as nodes of a directed graph. For textual presentation, we linearize the graph breadth first. Each pattern/context pair is preceded by all pattern/context pairs that are needed to understand it.

We don't demand to avoid cyclic pattern relationships. In fact, very often cyclic descriptions of terms are a helpful means for understanding complex situations. We solve resulting ambiguities pragmatically by forward references.

The ordering relationship between two patterns is based on the relationship of the contexts of each pattern. If one of the contexts comprises the other and can be understood as embedding it, or if it is needed to understand the other context, than it has to precede it in the graph.

Here the three levels of patterns come into play again. Conceptual patterns provide the background and motivation for the respective design patterns which in turn provide the context for programming patterns. Therefore:

Conceptual patterns logically precede design patterns which logically precede programming patterns.

Figure 3 shows an excerpt of such an ordering for some of the patterns for the Tools and Materials Metaphor.

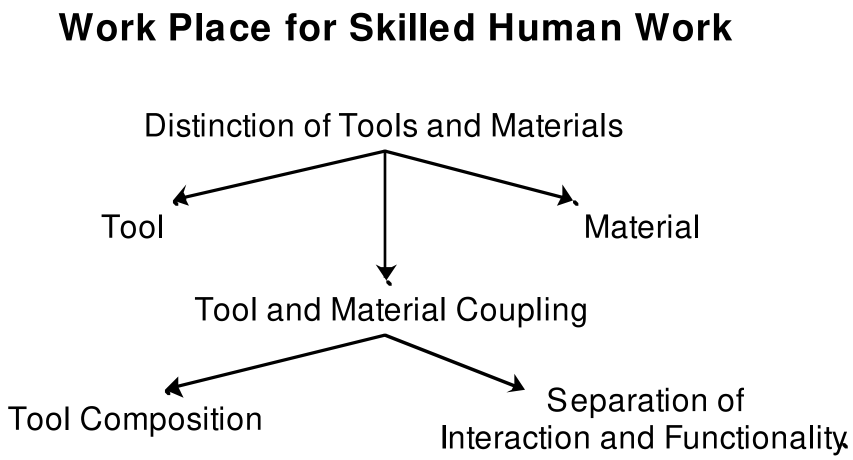

Figure 3: The conceptual pattern Distinction of Tools and Materials relies on the metaphors tool and material. The conceptual patterns provide the context in which to interpret the design patterns Tool and Material Coupling, Tool Composition and Separation of Interaction and Functionality (Riehle & Züllighoven, 1995). The patterns are to be seen in front of the overall background of a work place enabling skilled human work.

5.2 Pattern background

When ordering patterns, we have to identify the first pattern/context pair that embeds all the following patterns. But what about the context of this first pattern/context pair? How can we describe the overall context that lets us understand the entire pattern set? Here we have reached the notorious point of systems theory where we are obviously heading towards an infinite regression of embedding contexts. Thus, we avoid a "first pattern" approach but lay open the general background of the patterns. This background as the initial step into a set of possibly formal pattern descriptions cannot be formal itself.

Patterns can only be communicated and understood sufficiently well if authors and readers share a common background in the relevant domains. Every set of patterns addresses a particular domain and thus its impact and usability will only be understood by those with both sufficient knowledge and experience in this domain.

This does not mean, however, that patterns can only be understood by those who have had the same experiences and level of training or skills as their originators. A successful vehicle for addressing this part of the background are case studies as in (Gamma et al., 1995) or a first separate pattern introducing the application domain as in (Johnson, 1992).[5]

The background is presented as an introductory text to the patterns. It describes the problem domain, the main objectives of the patterns, and it tries to make explicit what has shaped the patterns and their contexts on an overall scale.

The background is formed by the cultural, social and individual history, experiences and knowledge of people who establish and work with patterns in their contexts. It cannot be formalized or fully described, but it has to be shared to some degree in order to establish a mutual understanding and a shared practice.

5.3 Leitmotif for software development

Software is developed by people. Though often ignored, this means that personal beliefs and values are driving forces for application design. This holds true in general but particularly for the design of interactive software systems. What are the "real requirements?" Do we want to design software to automate work or to empower human qualifications and skilled work? Do we see computers as "systems" or "machines" or do we view them as "tools" or "media" (Maaß & Oberquelle, 1992)?

These questions have a significant impact on the form and content of a set of patterns. Thus, we feel that it is important to make underlying assumptions and values explicit. Once these aspects have been laid open, they can be made subject to discussions by the parties concerned.

This "world view" underlying software development can be made explicit by what we call a leitmotif:

A leitmotif is a general principle that guides software development. It is the "broad picture" which can evoke a vision of the future system. It makes the underlying beliefs and values explicit that drive software design as a whole.

Our leitmotif, as has been mentioned, is the well equipped workplace for qualified human work at which experts carry out their work and take over the responsibility for it. This leitmotif is the crucial part of our background and the key to understanding the patterns we use.

The leitmotif already addresses or relates to the conceptual patterns. They detail and explain what type of software we have in mind when we talk about work places for expert human work.

6 Towards a Pattern Handbook

By now we have discussed the notion of pattern and how to relate it to other patterns and its context. We have identified three general pattern types, each of them linked to the main models needed in software development. Then, we have argued that patterns are best understood if their background is made explicit. On the overall system design scale, this lead us to the identification and use of a leitmotif.

If we want to use patterns as central building blocks for the different models, we need a means to coherently integrate and present the background, leitmotif, patterns and metaphors. To achieve this integration, three major terms have been proposed: pattern languages (Alexander et al., 1977; Coplien & Schmidt, 1995), pattern catalogs (Gamma et al., 1995), and pattern systems (Buschmann et al., 1996; Schmidt, 1996).

We avoid the notion of pattern language, since we think that our pattern sets are neither used in a linguistic nor in a computer science sense of the term language.[6] Neither does the notion of pattern system seem to be appropriate, since it is not clear to us how a system of patterns relates to the notion of system as defined in systems' theory. Pattern catalogs, finally, are more a loosely coupled collection of patterns than the coherent set of patterns guided by a leitmotif which we are thinking of.

We therefore propose to (re-)use the notion of handbook (Anderson, 1993), being defined as a handy work of print that concisely summarizes the relevant concepts from a domain.[7]

---

5 We think that Johnson's "first pattern" isn't a pattern but what we're calling background here.

6 From the discussions at the last Pattern Languages of Programming conference we got the impression that despite the conference's title our reluctance to use this term is shared by many other researchers in this domain.

7 Derived from (Brockhaus & Wahrig, 1981).

The term that sticks out of this definition is "domain." Obviously, there is a need to focus on a domain when writing a software engineer's pattern handbook. This is in line with (Jackson, 1994): A handbook has to provide the conceptual framework for guiding the developer in making crucial design decisions about the kind of computer support and automation in a specific application domain. Taking into consideration what we have said about patterns and their contexts, developers first of all have to acquire sufficient knowledge in the domains that may effect the development of the system. Then, a pattern handbook can link patterns to domain knowledge, standard software solutions and architectures useful in that domain.

Such a pattern handbook for software development might consist of the following sections:

- An introduction and overview that presents the leitmotif as well as the background needed to understand the patterns.
- An outline of the characteristics of the respective application domain, like typical workplaces with their objects and means of work, types of cooperation and communication or standard problems and their solutions.
- A section on an appropriate development strategy which, in our case, considers issues like evolutionary software development with prototypes and participatory design (Floyd & Gryczan, 1992; Budde et al., 1992; CACM, 1993).
- A structured pattern set starting with conceptual patterns that are followed by software design patterns which are finally complemented by programming patterns (depending on the level of detail appropriate).

First steps towards this handbook have been taken in (Riehle & Züllighoven, 1995), where we still called it a pattern language, and in (Riehle, 1995) where we first thought about the notion of handbook in the pattern context.

7 Related Work

The most wide-spread notion of pattern today is the one given by Gamma et al. (1995) which is based on Alexander (1979) as discussed in section 2. In addition, Coad presents a more general definition (1995). Our own definition presented in section 2 is also closer to the dictionary than Alexander's original notion.

Pattern forms abound, as is apparent from the Pattern Languages of Programming conference series. The main variants are the Alexandrian form (Alexander et al., 1977) and the template used in the design pattern catalog (Gamma et al., 1995).

There isn't much work on formalizing design patterns yet. A precursor to patterns is the work of Helm et al. (1990) on Contracts which are specifications of behavioral compositions. Lieberherr et al. have worked on adaptive programming (1994) which is based on propagation patterns that can be used to specify the design patterns from Gamma et al. (1995). Cowan et al. (1995) have also applied their ADV approach to the formal specification of design patterns.

There are many pattern types possible within software development. Some deal with analysis and design activities (Kerth, 1995; Whitenack, 1995), some deal with organizational issues (Cain et al., 1996; Berczuk, 1996). There are conceptual patterns (Riehle & Züllighoven, 1995), software design patterns (Gamma et al., 1995) and programming patterns (Coplien, 1992).

One can think of several dimensions along which to organize patterns. Gamma et al. (1995) organize their patterns in two dimensions, one based on a pattern's purpose (creational, structural, behavioral) and the other one based on its scope (object or class based). Buschmann & Meunier (1995) organize patterns along the dimensions of granularity (architecture, software design, implementation), structural principles (abstraction, encapsulation, separation of concerns, coupling and cohesion) and functionality (creation, communication, access, organization). Pree (1995) finally introduces meta patterns as abstractions from the patterns in Gamma et al. (1995).

Pattern languages can be found in the Pattern Languages of Programming conference proceedings (Coplien & Schmidt, 1995; Vlissides et al., 1996). We know of at least two pattern systems (Buschmann et al., 1996; Schmidt, 1996). Currently, there is one pattern catalog (Gamma et al., 1995).

Today, some experience reports on using patterns in software development exist. Schmidt (1995) reports on the use of design patterns in distributed computing projects. Beck et al. (1996) is a joint paper of several renowned scientists and practitioners who reflect on their experience with design patterns and their impact on software development projects. Zimmer (1995) reports about the use of design patterns to reorganize a hypermedia application. We have recently submitted a paper discussing our experience with patterns and their embedding within our overall approach (Bäumer et al., 1996).

8 Conclusions

In this paper we took stock of our and other researchers experiences. We discussed what we perceive as crucial aspects of the pattern concept. We presented pattern levels and related them to the different models built during software design, followed by a presentation of different pattern description forms currently in use. We discussed how patterns relate with each other and can be ordered to form larger wholes. Then we introduced the notion of background and leitmotif to supplement patterns and integrate them. Finally we argued for the need of pattern handbooks to round up efforts of presenting large pattern sets.

We believe that the last issue, the presentation of large pattern sets, be it in the form of pattern languages, catalogs, systems or handbooks, will be one of the most important issues the patterns community has to deal with in the future. We are directing our efforts to the writing of such a pattern handbook that will summarize our experiences as patterns. We are convinced that it will serve us well as an aid in our projects and we will very carefully observe its application and impact on our software development projects.

Acknowledgments

We wish to thank Steve Berczuk, Frank Buschmann, Douglas Lea and Douglas Schmidt for reviewing and commenting on the paper in its several stages. Dirk wishes to thank Walter Bischofberger for his support and comments on (Riehle, 1995) on which this paper is based. We also thank Robert Ingram and David James for helping us to improve the writing of this paper.

Bibliography

Alexander, C. (1979). *The Timeless Way of Building.* New York: Oxford University Press.

Alexander, C., Ishikawa, S., & Silverstein, M. (1977). *A Pattern Language.* New York: Oxford University Press.

Anderson, B. (1993). Workshop Report: Towards an Architecture Handbook. OOPSLA '92 Addendum, *OOPS Messenger* (April 1993).

Bäumer, D., Gryczan, G., Lilienthal, C., Riehle, D., Strunk, W., Wulf, M., & Züllighoven, H. (1996). The Tools and Materials Approach—Analysis, Design and Construction of Interactive Object-Oriented Systems. Submitted for publication.

Beck, K., & Johnson, R. (1994). Patterns Generate Architectures. ECOOP '94, LNCS 821, *Conference Proceedings* (pp. 139-149). Berlin, Heidelberg: Springer-Verlag.

Beck, K., Coplien, J.O., Crocker, J., Dominick, L., Meszaros, G., Paulisch, F., & Vlissides, J. (1996). Industrial Experience with Design Patterns. ICSE-18, *Conference Proceedings*. Los Alamitos: IEEE Press, 1996.

Berczuk, S. (1994). Finding Solutions Through Pattern Languages. *IEEE Computer* 27, 12 (December 1994).

Berczuk, S. (1996). A Pattern Language for Ground Processing of Science Satellite Telemetry. In (Vlissides et al.,1996).

Budde, R., Kautz, K., Kuhlenkamp, K., & Züllighoven, H. (1992). *Prototyping.* Berlin, Heidelberg: Springer-Verlag.

Budde, R., & Züllighoven, H. (1992). Software Tools in a Programming Workshop. In (Floyd et al., 1992), (pp. 252-268).

Bürkle, U., Gryczan, G., & Züllighoven, H. (1995). Object-Oriented System Development in a Banking Project: Methodology, Experiences, and Conclusions. *Human Computer Interaction 10, 2&3 (1995),* 293-336.

Brockhaus & Wahrig. (1981). *German Dictionary (in German).* G. Wahrig, H. Krämer & H. Zimmerman (Eds.). Wiesbaden, Germany: F. A. Brockhaus.

Buschmann, F., & Meunier, R. (1995). A System of Patterns. In Coplien & Schmidt (1995), (pp. 325-343).

Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P., & Stal, M. (1996). *Pattern-Oriented Software Architecture—A System of Patterns.* Wiley & Sons Ltd.

CACM. (1993). Special Issue on Participatory Design. *Communications of the ACM* 36, 4 (June 1993).

CACM. (1994). Special Issue on Agents. *Communications of the ACM* 37, 7 (July 1994).

Cain, B.G., Coplien, J.O., & Harrison, N.B. (1996). Social Patterns in Productive Software Development Organizsations. *Annals of Software Engineering* (1996). To appear.

Caroll, J.M., Mack, R.L., & Kellogg, W.A. (1988). Interface Metaphors and User Interface Design. In M. Helander (Ed.) *Handbook of Human-Computer Interaction* (pp. 67-85). Amsterdam: North-Holland.

Coad, P. (1992). Object-Oriented Patterns. *Communications of the ACM* 35, 9 (September 1992), 152-159.

Coad, P., North, D., & Mayfield, M. (1995) *Object Models: Strategies, Patterns & Applications.* Prentice-Hall.

Coplien, J. (1992). *Advanced C++: Programming Styles and Idioms.* Reading, Massachusetts: Addison-Wesley.

Coplien, J. (1995). A Generative Development-Process Pattern Language. In (Coplien & Schmidt, 1995), (pp. 183-238).

Coplien, J., & Schmidt, D. C. (Eds). (1995). *Pattern Languages of Program Design.* Reading, Massachusetts: Addison-Wesley.

Coplien, J. (1996). The Human Side of Patterns. *C++ Report* 8, 1 (January 1996), 73-80.

Cowan, D.D., & Lucena, C.J.P. (1995). Abstract Data Views: An Interface Specification Concept to Enhance Design for Reuse. *IEEE Transactions on Software Engineering* 21, 3 (March 1995), 229-241.

Floyd, C., & Gryczan, G. (1992). STEPS - A Methodological Framework for Cooperative Software Development with Users. EWHCI '92, *Conference Proceedings.*

Floyd, C., Züllighoven, H., Budde, R., & Keil-Slawik, R. (1992). *Software Development and Reality Construction.* Berlin, Heidelberg: Springer-Verlag.

Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (1993). Design Patterns: Abstraction and Reuse of Object-Oriented Design. ECOOP '93, LNCS-707, *Conference Proceedings* (pp. 406-431). Berlin, Heidelberg: Springer-Verlag.

Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (1995). *Design Patterns: Elements of Reusable Design.* Reading, Massachusetts: Addison-Wesley.

Greenspan, S., Mylopoulos, J., & Borgida, A. (1994). On Formal Requirements Modeling Languages: RML Revisited. ICSE-16, *Conference Proceedings* (pp. 135-147). Los Alamitos, California: IEEE Computer Society Press.

Helm, R., Holland, I.M., & Gangopadhyay, D. (1990). Contracts: Specifying Behavioral Compositions in Object-Oriented Systems. OOPSLA '90, *SIGPLAN Notices* 25, 10 (October 1990), 169-180.

Hoare, C.A.R. (1972). Notes on Data Structuring. In E. Dijkstra, C.A.R. Hoare, & O.-J. Dahl (Eds.), *Structured Programming* (pp. 83-220). Academic Press.

IEEE. (1995). Special Issue on Software Architecture. *IEEE Transactions on Software Engineering* 21, 4 (April 1995).

Jackson, M. (1983). *System Development.* Englewood Cliffs, New Jersey. Prentice Hall.

Jackson, (1994). Problems, Methods and Specialisation. *IEE Software Engineering Journal* 9, 6 (November 1994), 249-255.

Jacobson, I. (1992). *Object-Oriented Software Engineering.* Reading, Massachusetts: Addison-Wesley.

Johnson, R. (1992). Documenting Frameworks using Patterns. OOPSLA '92, *ACM SIGPLAN Notices* 27, 10 (October 1992).

Kerth, N. (1995). Caterpillar's Fate: A Pattern Language for the Transformation from Analysis to Design. In (Coplien & Schmidt, 1995), 293-320.

Lieberherr, K.J., Silva-Lepe, I., & Xiao, C. Adaptive Object-Oriented Programming. *Communications of the ACM* 37, 5 (May 1994), 94-101.

Maaß, S., & Oberquelle, H. (1992). Perspectives and Metaphors for Human-Computer Interaction. In (Floyd et al., 1992), (pp. 233-251).

Meyer, B. (1988). *Object-Oriented Software Construction.* Englewood-Cliffs, New Jersey: Prentice-Hall.

Meyer, B. (1991). Design by Contract. In D. Mandrioli & B. Meyer (Eds.), *Advances in Object-Oriented Software Engineering*, (pp. 1-50). New York, London: Prentice-Hall.

Parnas, D.L., & Clemens, P.C. (1986). A Rational Design Process: How and Why to Fake It. *IEEE Transactions on Software Engineering*, 12, 2 (February 1986), 251-257.

Pree, W. (1995). *Design Patterns for Object-Oriented Software Development.* Reading, Massachusetts: Addison-Wesley.

Riehle, D. (1995). *Patterns—Exemplified through the Tools and Materials Metaphor.* Masters Thesis (in German). UBILAB Technical Report 95.6.1. Zürich, Switzerland: Union Bank of Switzerland.

Riehle, D., & Züllighoven, H. (1995). A Pattern Language for Tool Construction and Integration Based on the Tools and Materials Metaphor. In (Coplien & Schmidt, 1995), 9-42.

Riehle, D. (1996). Patterns for Encapsulating Class Trees. In (Vlissides et al., 1996).

Riehle, D., Schäffer, B., & Schnyder, M. (1996). Design of a Smalltalk Framework for the Tools and Materials Metaphor. *Informatik/Informatique* 3 (February 1996), 20-22.

Schmidt, D. (1995). Using Design Patterns to Develop Reusable Object-Oriented Communications Software. *Communications of the ACM* 38, 10 (October 1995), 65-74.

Schmidt, D. (1996). A System of Reusable Design Patterns for Application-Level Gateways. *Theory and Practice of Object Systems.* This issue.

Vlissides, J., Kerth, N., & Coplien, J. (Eds.). (1996). *Pattern Languages of Program Design, Volume 2.* Reading, Massachusetts: Addison-Wesley.

Whitenack, B. (1995). RAPPeL: A Requirements-Analysis-Process Pattern Language for Object Oriented Development. In (Coplien & Schmidt, 1995), 259-292.

Zimmer, W. (1995). Using Design Patterns to Reorganize an Object-Oriented Application. In E. Casais (Ed.), *Architectures and Processes for Systematic Software Construction* (pp. 171-183). FZI-Publication 1/95, Forschungszentrum Informatik Karlsruhe, 1995.