Lessons Learned from Using Design Patterns in Industry Projects

Dirk Riehle

SAP Research, SAP Labs, LLC
3412 Hillview Ave, Palo Alto, CA 94304, U.S.A.
dirk@riehle.org, www.riehle.org

Abstract. Design patterns help in the creative act of designing, implementing, and documenting software systems. They have become an important part of the vocabulary of experienced software developers. This article reports about the author's experiences and lessons learned with using and applying design patterns in industry projects. The article not only discusses how using patterns benefits the design of software systems, but also how firms can benefit further from developing a firm-specific design language and how firms can motivate and educate developers to learn and develop this shared language.

Keywords. Design pattern, pattern language, design language, design communication, design collaboration, design implementation, design documentation.

1 Introduction

The notion of design pattern has been defined as the abstraction from a common solution to a recurring design problem in a given context [1] [2]. A well-known example is the Observer pattern from the Design Patterns book (a.k.a. the "gang of four" book [1]) which solves the problem of managing state dependencies between objects through the introduction of registration and callback interfaces.

A design pattern like Observer introduces terms like "observer" and "subject" that become part of the language that developers speak when they go about designing, implementing and documenting software systems. This article discusses different uses of design patterns and how to be effective at them. The article is based on the author's experiences with using and applying design patterns in industry projects since 1995 (Section 2). The discussion focuses not only on design patterns, but also on the overall vocabulary and language that a software development team or a firm typically speaks. The article shows how design patterns can fit into such a design language, the firm's software architectures, and its programming practices.

Prior work has shown how a firm can benefit from using and applying design patterns [7] [11] [12]. This article discusses how these benefits can be enhanced further

through the development and use of a firm-specific design language (Section 3 and 4). Such a firm's design language is something that needs to be learned and that keeps evolving. New employees need to be brought up to speed with the language and new insights need to work their way into the language. Hence, this article presents the author's experiences with running study groups and other educational measures that are used to make new employees become more productive faster. Finally, the article presents experiences with advanced study groups and writers' workshops to refine a firm's understanding of its own software patterns and architecture (Section 5).

2 Base of Experiences

The experiences presented in this article are based on the author's involvement with the industry projects shown in Table 1. The author was employed full-time with the respective firm pursuing the project.

Table 1. List of projects the author was involved in from 1995 to 2006

Firm	Project Name	Time Frame	# Developers	Author's Role
UBS	KMU Desktop	1995-1996	13	Developer
UBS	KMU Desktop	1996-1998	15	In-house Consultant
KMU Desktop stands for "Kleine und Mittlere Unternehmen" i.e. small and medium-size businesses. This is a banking application for commercial lending [32] [31].				
UBS	GBO Project	1997-1999	7	In-house Consultant
GBO Project stands "Global Business Objects" and was an attempt to consolidate the globally distributed applications of UBS, a Swiss Bank, under one object model [32] [34].				
Skyva	Skyva	1999-2002	15	Architect
Skyva	Skyva	1999-2002	45	In-house Consultant
Skyva was the name of both the company and its software product for managing supply chains. The author's team focused on the core language, runtime, and tools for a UML virtual machine [33]. The author consulted to all of development, 45 developers in total.				
Bayave	Bayave	2004-2006	5	Chief Architect
Bayave Software GmbH develops on-demand business software for small businesses.				

The core of the experiences reported about is based on the time frame from 1995-2002 when the author started systematically introducing design patterns into the software development processes he was involved in.

After leaving UBS and joining Skyva, the author made specific choices and consulted in-house to foster the creation of a firm-specific design language based on design patterns. His later work at Bayave repeated some of those experiences.

Typically, the author was both a developer and/or architect and an in-house consultant, to build trust and to ensure his feet were on the ground.

Since 2006, the author has only been involved with smaller research projects that did not influence the experiences presented in this article in any significant way.

3 Uses of Design Patterns

This section reviews the key uses of design patterns the author has found in industrial software development projects. It also discusses some common misconceptions.

The dominant uses of patterns in software development are to facilitate

- communication,
- implementation (by hand),
- and documentation of software systems

as documented before [7] [10] [11] [12] [13].

In addition, some research and development efforts have gone into pattern-based code generation, which is now supported by commercially available tools like IBM Rational's Software Modeler or Borland's Together. However, the pattern-oriented features of these tools have yet to reach maturity, in particular in their expressiveness for capturing design patterns as well as in their support for round-trip engineering. In the author's opinion, design pattern based code generation has not had any significant impact on the industry's design practices; the article discusses some of the reasons for this in Section 3.4 on misconceptions.

3.1 Communication

By far the dominant use of design patterns the author has seen in his projects is in informal and creative communication between software developers [13] [7]. Working on a white-board or discussing a design at the water cooler, developers use pattern-based vocabulary to refer to their prior and shared design experiences as well as the generally known descriptions of one or more patterns. By referring to such prior experiences, developers bundle knowledge about design constraints, the contexts in which they apply, and their solution in one apt name. They use this knowledge in evaluating and deciding on design options to solve the design problems at hand.

Consider Figure 1. Here, on a white-board, a developer might be explaining to a colleague how he intends to handle incoming user requests to a web server in such a way that these requests can be undone, bundled into multi-step requests, made persistent, and replayed at some later time on some other server. The developer explains how he encapsulates a request as a command object, how he creates composite com-

mand objects to bundle atomic steps into more comprehensive actions, and how a command processor keeps track of the sequence of overall user actions such that multi-step undo and replay becomes possible.

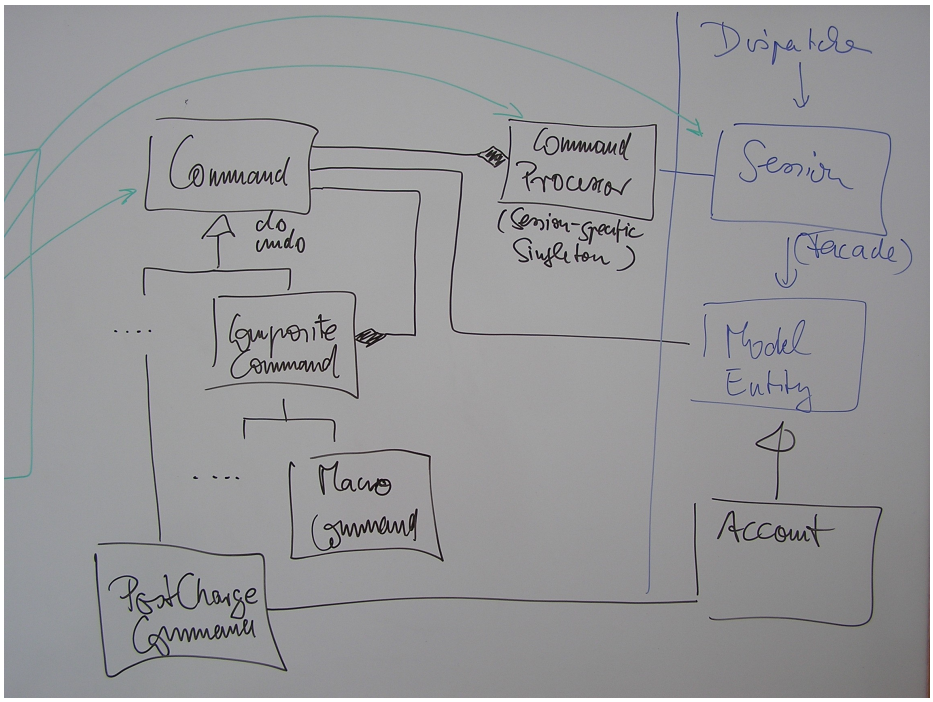

Figure 1. An illustration of a design described using the Command and related patterns.

The second developer, who knows the Command, Composite, and Command Processor patterns [1] [3] and who has previously used these patterns, is likely to grasp the idea of the design quickly. She proceeds to provide feedback, points to problems and suggests alternative solutions, using equally refined professional language. In the well-known back and forth of a creative discussion, design patterns make software developers more efficient and more effective. This is simply because they need fewer words to refer to comprehensive problem solutions and yet remain clear about what they are saying while using design ideas that have been proved to work well.

The goal of using design patterns in informal communication is to make developers more effective. In the author's projects, developers typically settled quickly into a work mode where ambiguities are removed as they surface by refining what has been said. Completeness and correctness were not primary goals. The point was to have a discussion that lead to an incomplete but shared understanding of some design aspects of the overall system. This shared understanding allowed developers to continue on their paths, working forward, and reconvening as they run into problems with the design some time down the road.

3.2 Implementation

The second most important use of design patterns lies in implementing software systems. One misconception with design patterns is that they are on the "design level" only and are disconnected from the code. This is clearly not true [4] [7] [22] [25]. Design patterns, as the descriptions in the Design Patterns book vividly show, are closely connected to code [1]. There is not a single design pattern description in the seminal work that doesn't come with comprehensive code examples. This is not to say that there is only one way of turning a pattern into source code (see Section 3.4 on common misconceptions). An experienced developer has multiple templates in the back of his or her mind that he uses and adapts as he translates the results of a design discussion into code.

Consider the white-board discussion from Section 3.1. After the developers decide that the sketched-out design is a good way forward, the first developer returns to his workstation to define the interfaces and class structures, followed by an initial implementation and some test cases. While writing the code, the developer balances in his head the specifics of the discussed design, the canonical implementations described in the respective pattern descriptions, some other implementations that only this developer may have seen, and the general needs of the problem at hand. During this implementation, the developer faces design and implementation problems that were not discussed with the other developer, but that come up as he is working on the code. Most of these problems are solved on the spot using informed judgment, but a few may be so tricky that they are left unsolved and reserved for future discussion with other developers.

The role of design patterns in this activity is to inform the details of writing source code: What methods to group together and to put on what interface or class, what the exact signature of a method should look like, in what order to call other methods, and so on. Design patterns inform such decisions, because they provide a more comprehensive "bigger" picture of the design and yet are specific enough to lead to code based on prior experiences. In the example from Section 3.1, the developer has to decide whether the undo/redo methods of the Command interface are visible to everyone or only the CommandProcessor, whether the addChildCommand method is on the Command or CompositeCommand interface, and so on.

Figure 2 shows the ValueReader interface from the Serializer design pattern implementation in the JValue framework for Value Objects [14] [15] [16]. A developer familiar with the Serializer pattern can immediately recognize the purpose of the interface and derive the structure and behavior of its implementations.

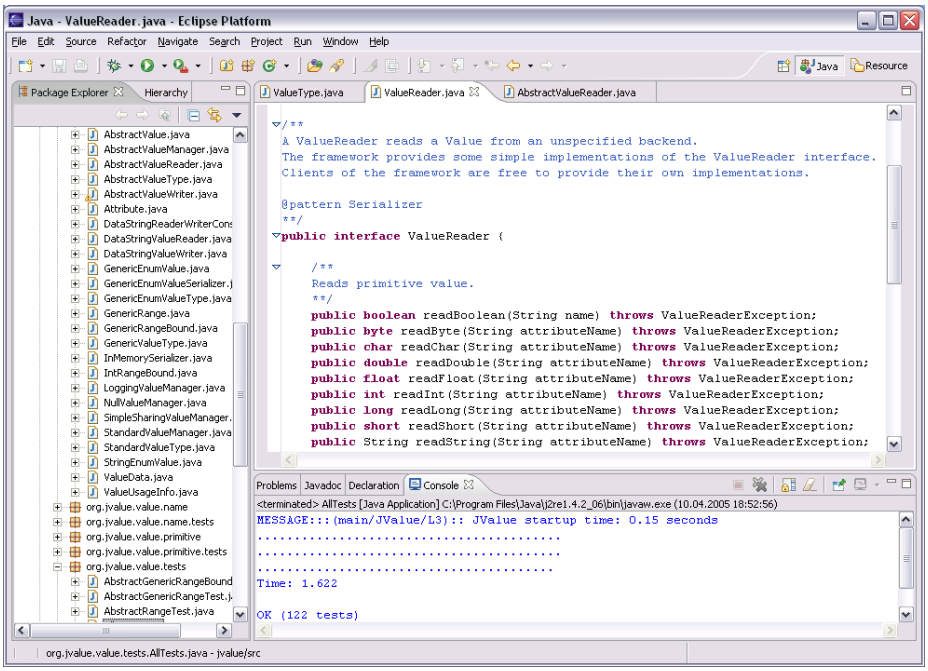

Figure 2. The ValueReader interface of the Serializer pattern implementation in the JValue framework.

In the author's projects, experienced software developers used their knowledge of design patterns and their implementation variants on a continued on-going basis to transfer a design into code. This benefits everyone: Because other developers know these stylized design pattern implementations as well, they can more quickly understand from the code which pattern is being implemented and what the consequences for the system's structure and behavior are. Thus, using and applying design patterns made implementing designs and comprehending source code easier.

3.3 Documentation

The third main use of design patterns the author has found is to aid documentation. For example, in a typical word processor document, a developer might describe a design by showing its class structure. He or she explains that the class structure implements a set of design patterns and then shows how that structure achieves its purpose by referring to the elements of the design patterns, their structural relationships, and their protocols of interaction. Basically, design patterns give a developer a vocabulary that he can use to document a design in a more succinct way than possible with only regular prose.

Figure 3 illustrates such informal documentation using a design pattern annotation form suggested by Erich Gamma [10]. The figure displays several core classes from

the JHotDraw framework for graphical editors [17]. A blue box next to a class annotates the class as playing a particular participant role in the pattern application, as described in the original design pattern documentation. Where it says “Observer: Subject-2” the Figure class plays the Subject role from the Observer pattern. It says ‘2’ because there is more than one application of the Observer pattern in this design (not shown in Figure 3).

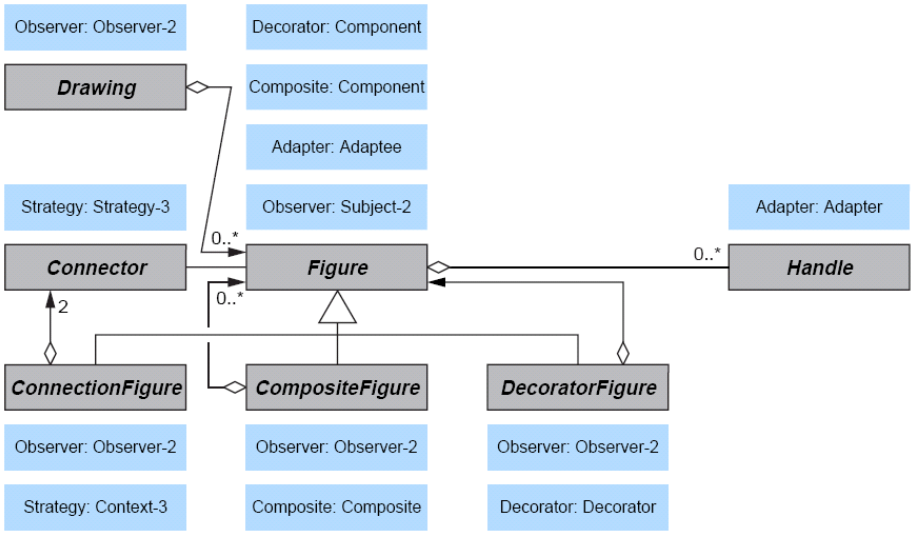

Figure 3: Documentation of the core design of the JHotDraw framework using design pattern annotations.

In some projects, developers documented object-oriented designs in a more formal fashion using UML tools. UML’s design pattern feature was used to annotate collaborations as design pattern instances. Frequently, the author found that developers took these class diagrams from the UML tool and copied them over into a word processor document or a wiki where they were explained in prose again. Wherever or whatever the master document, the lesson is clear: Experienced developers use design patterns to express the structure and dynamics of their designs. It speaks to them and those who are reading the design.

Like in design pattern guided implementation, it is the author’s experience that using design patterns in documenting designs makes developers more effective and helps readers comprehend designs faster.

3.4 Misconceptions

By far the most common mistake that surfaced in the projects was to confuse a design pattern with a design template or even a code template. This mistake is easy to make; after all, the structure diagram in the Design Patterns book suggests that there is only

one particular structure that is to be considered as the pattern. This is an incorrect interpretation. The structure diagram in the Design Patterns book is an illustration of the most common form the pattern may take when it gets applied, but it really is only one of many forms [1] [4] [10], as confirmed by John Vlissides, one of the authors of the Design Patterns book [22].

Figure 4 shows three different structure illustrations of the Composite pattern. The first one is the structure diagram of the original description. The second one is discussed as a variant in the implementation section of the original description. The third one is a variant of the pattern used in one of the firms the author worked for. (See Section 4 on firm-specific design languages.) All three illustrations are truthful to the idea and description of the Composite design pattern.

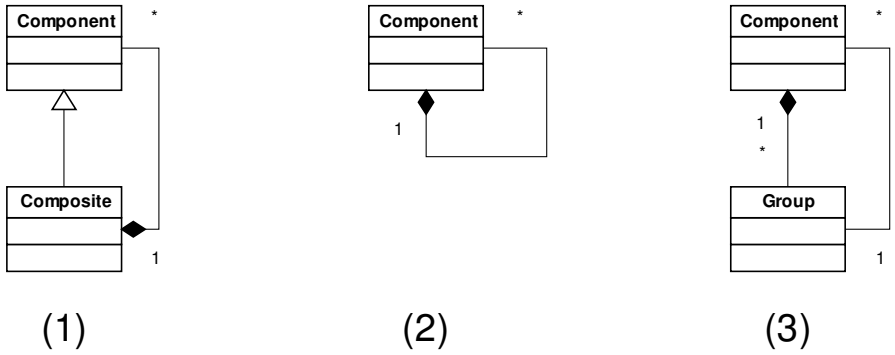

Figure 4: Three different illustrations of common variations of the Composite design pattern.

It is important to note that these illustrations are just that: illustrations. There is no well-defined pattern specification language underlying these class diagrams. These diagrams are specific designs with classes that have general names for the purpose of invoking the idea of how an application of the pattern could look like: They are an illustration, not a (formal and precise) specification.

It should not come as a surprise that design patterns can be implemented in different ways. Not even the authors of the Design Patterns book could foresee all the possible circumstances in which to find a solution to a recurring problem. For example, the Design Patterns book was written when multi-threading was less common than today; thus, most of the patterns ignore this contextual force. In a multi-threaded context, however, the Singleton pattern is likely to fail when implemented as illustrated in the book. For the successful use of the Singleton pattern in multi-threaded applications, according to this author's experience, the notion of single exemplar needs to be changed to a context-specific singleton, which typically leads to a rather different implementation.

The UML 2.0 specification suggests using the modeling concept of Collaboration to illustrate design patterns. However, UML collaborations are limited in many ways.

First, the notion of a pattern in UML is that of a specific (template-like) design element and is therefore too specific to allow for the many variations in which a pattern can come. Second, the notion of pattern in UML is constrained to collaborations, that is, descriptions of how objects collaborate to achieve some purpose. The author's work on using collaborations (a.k.a. role models) to describe patterns [5] supports the idea that many design patterns are best viewed as abstractions from recurring collaborations between objects. However, not all patterns are behavioral in nature. The Null Object pattern, for example, is wholly structural [18] and cannot be illustrated well by the UML notion of pattern.

To resolve the conflict between the original notion of a pattern as an idea that may take many forms and the desire to have exactly one precise specification, the author's projects have found it useful to distinguish between a design pattern, its variations, templates for those variations, and implementations or applications of those templates [6] [13]. A pattern is the idea that solves a problem, and that idea can show itself in many different variations, depending on the context. Formal specifications of these variations are viewed as templates that can be instantiated to give users a specific design and a specific implementation. The relationship between pattern and template is 1-to-n, and the relationship between template and application is 1-to-n as well. This distinction solves the conflict between the desire to formalize and the need for flexibility.

4 A Firm's Design Language

The author's projects benefited from using and applying design patterns as discussed in the previous section. However, the projects went further: They adapted design patterns and design vocabulary to their specific processes and products. The author saw this happening in the projects at UBS and he actively supported and steered this process while working at Skyva and Bayave, see Table 1 and Section 5. This section discusses the additional benefits projects harvested from investing into project and firm-specific design languages.

Design patterns are an important part of the language that experienced software developers use when talking about system design and implementation. Other components of this language that the author found are

- project and firm-specific variations of design patterns,
- project and firm-specific patterns discovered in the firm's products and systems,
- the architecture of the firm's products and systems, and
- the firm's programming practices.

In a well-working software organization, these parts come together to form what the author of this article, inspired by [8] [23], is calling a "design language". A design language consists of the words and concepts that developers use to effectively communicate about the design and implementation of their work products. It is rooted in

everyday natural language and enhanced by the technical concepts developers are dealing with in software design.

4.1 Firm-specific pattern variations

Recall the definition of design pattern given in the introduction: “An abstraction from a common solution to a recurring design problem in a given context.” Often neglected, understanding a pattern’s context is important. This is because the forces that represent a context shape what a good solution for the problem is.

In a firm that provides a particular product or system that system and its requirements represent the main set of forces, or context, of any pattern application. The system and its architecture, whether it is object-oriented, event-driven, or data-flow-oriented, details many of the forces and leads to firm-specific variations of design patterns. These are general design patterns that have found a specific recurring variation in the system or product of a software organization.

Figure 4 shows variations of the Composite pattern. The Composite pattern lets developers represent and manage an object tree. Each object is a component (node in the tree), and some components are composite objects consisting of further component objects, some of which may be composites, etc. An object tree emerges from this common problem solution.

On the right, Figure 4 shows a firm-specific variation of the Composite pattern that was used at Skyva. Rather than having components (nodes) maintain links to their sub-components (child nodes), an intermediate Group object maintains these links. Thus, a component has a group object which in turn holds the links to the next lower level of objects in the tree.

This indirection may seem like unnecessary complexity in the general situation, however, it was a necessity for the meta-data driven software system under development. This system had a description layer that determined the system structure at runtime and could also change at runtime. For that reason, it couldn’t be known exactly how many different types of trees an object might be a node in. Rather, the developers had to assume it could be any number. An example of such a situation is the modeling of a firm’s organizational structure. Here, one may need to represent a person’s position in many different hierarchies: First, there is the regular line reporting hierarchy, then there maybe any number of projects each of which introduces its own reporting hierarchy, there may be a technical reporting relationship, etc. Organizational modeling is one situation where this design pattern variant is used, business process modeling is another.

The firm-specific variation shown in Figure 4 solves this problem by letting a node have multiple group objects, each of which provides access to the sub-nodes in a different tree. You can see this by the 1-to-n relationship between Component and Group in the UML diagram of this pattern variation.

For the project at Skyva, recognizing this variation of the Composite pattern as a firm-specific variation of the pattern represented a major step forward. We quickly

standardized our vocabulary on terms like Component and Group and became more effective at discussing our designs wherever this pattern variation showed up.

4.2 Firm-specific design patterns

It is not just variations of known patterns: At Skyva and Bayave the developers also found new and not yet documented patterns. Developers documented them when they felt they were relevant for their work. At Skyva, the author's team not only discovered the Composite pattern variation of Figure 4, but also discovered what was called the Navigator pattern. The Navigator pattern describes a generalized traversal algorithm over an object graph using meta-data to find the next target of the navigation. The team used "a navigator" to find a particular component in the graph. Using "navigator" and ancillary terms, the team could quickly and effectively discuss its designs.

Later the author's team learned that this pattern could also be found in implementations of the XPath/XQuery specification, which are a part of most XML processing engines these days. Hence, what was thought of as a firm-specific pattern apparently was not, and there were other people who had discovered the same pattern or at least were using it.

It is likely that a firm-specific pattern or a firm-specific pattern variation isn't that firm-specific after all and can be generalized. However, as long as there is no generally known description of these patterns available, a firm does well to come up with its own description (see Section 5.2 on study groups). Once more, the lesson that the author learned was that recognizing patterns and standardizing vocabulary makes developers more effective.

4.3 Firm-specific architecture and coding practices

A firm's design vocabulary and language comprises not only design patterns but also architecture and programming. It is not important whether an architectural style is called an architecture pattern or whether a coding trick is called a programming pattern [7]. What is important is that a firm's developers recognize a recurring theme as such and develop a common vocabulary around it, most likely by writing up the pattern and discussing it among themselves.

Prior work has shown how firms can benefit from using design patterns [11] [12]. During his work at UBS, the author observed that using design pattern terminology is part of a more comprehensive domain- and firm-specific language that developers speak when going about their work. At Skyva then, as well as at Bayave later, he explicitly developed this language using an on-going elicitation process. Uncommon terms, or common terms with specific meanings, were called out and discussed and documented on the firm's wiki, just like the design patterns.

In the author's experience, developing and maintaining a shared firm-specific design language can enhance the benefits derived from using design patterns greatly. When asked how design patterns had helped them, developers almost always provid-

ed positive responses. When asked how important the firm-specific design language was, responses were always enthusiastic, because the design language was custom-made to support the projects' daily work routines and usually did so well.

5 Learning the Language

During the work at UBS it had became clear that projects could become more effective by developing and using a shared design language. However, it was unclear what exactly the language consisted of and how to learn and refine it. Consequently, at Skyva, the author experimented with different forms of learning and reflection [8] to help new developers learn the design language, to refine the language together with experienced developers, to promote design patterns, and to discover yet hidden patterns.

The main vehicles of helping learning and reflection employed are

- individual tutoring,
- study groups (a.k.a. reading groups) [19] and
- writers' workshops [9].

Both the study groups and the writers' workshops groups were both internal and external to the firm, serving different purposes.

5.1 Individual tutoring

In the author's experience, an open and collaborative atmosphere that encourages sharing of design ideas, discussion of these ideas, and conclusions that lead to standardized vocabulary requires an atmosphere of trust, where ideas are welcome and everyone benefits.

The best way that to create this atmosphere is to lead by example and to give ideas and experience away freely. Whenever a new developer got on board, the author would sit down with the developer, helping them understand design patterns in general as well as our firm-specific variations.

This demonstrated the value of design patterns and that the firm cares about it. Once a new developer recognized these benefits, the developer was ready to join the foundational study group. The study group in turn would relieve the author and other senior technical members of the team from the time burden that such tutoring creates.

5.2 Study groups

A study group is a voluntary typically weekly get-together of 1-2 hours that would study a pattern or paper. At Skyva, the author instituted two types of study groups.

One studied well-established foundational material like the Design Patterns book, and one covered new and advanced patterns, concepts, and techniques.

The first type of study group, the foundational study group, tried to get new people up to speed, both on general as well as firm-specific material, including but not restricted to design patterns. Through this study group and the ensuing discussions, developers automatically soaked up the firm's language, as the natural thing was to discuss and discover applications of the design patterns in our day-to-day work.

The author has found that the following aspects were critical to making a study group a success and to maintain its momentum:

- At least one senior technical member of the team participated and facilitated the study group;
- The material chosen for review was relevant to the participants' work;
- Senior management clearly supported the study group and its activities.

This type of study group started over after about a year. There was no fixed curriculum, but the material the group reviewed was generally foundational and included patterns from the Design Patterns book. Members could join and leave as they pleased, and the group had many developers attending only sessions of particular interest to them.

The second type of study group, the advanced study group, reviewed current material from outside the company like new patterns from the annual Pattern Languages of Programming conference. In addition, the advanced group reviewed and discussed new material from the firm itself in a writer's workshop, see Section 5.3.

Firm-internal study groups can learn from outside efforts. The longest running cross-firm patterns study group that the author is aware of is the Silicon Valley Patterns group [20]. When asked by the author of this article, the group provided the following best practices for keeping the study group alive and interesting [21]:

1. Bring in authors (of the material under review)
2. Provide a safe setting (in particular, stop trouble makers early)
3. Say your names (to create a community atmosphere)
4. Insist on preparation (so meetings can be effective)
5. Encourage everyone (so nobody feels left behind)
6. Reflect and experiment (with the way the group works)
7. Meet in a comfortable place (in a cafe, not at work!)
8. One person at a time (give everyone room to speak)
9. Bring in laptops (to gather material or experiment on the side)
10. Select by consensus (so the group supports the curriculum)

The author of this article can confirm the Silicon Valley Patterns group experiences from his own study groups. Additional lessons learned in the author's firm-internal study groups were:

- Have a motivated group leadership that sticks with the group
- Be current on what's going on and bring it into the group
- Have a good connection to authors and don't be afraid to invite them

- Have at least one good moderator and facilitator
- Welcome everyone and discourage arrogance
- Organize through a mailing list and/or a wiki
- Meet consistently and regularly

The firm-internal study groups that the author started had no end date but just kept meeting regularly. There was always something new to discuss. Of particular interest were patterns that came out of the firm itself, enhancing and sharpening the firm's design language.

5.3 Writers' Workshops

Study groups typically study other people's material. Writers' workshops are a means for studying someone else's work with the goal of providing the writer with candid feedback [9]. In a writers' workshop, writer joins a group of readers who are reviewing the writer's work. The writer is not allowed to speak but can observe how workshop participants struggle with the material. From this, the writer learns how to improve the material. Richard P. Gabriel brought writers' workshops into the patterns community, where they have become an important part of patterns conferences [24].

In all cases, this article's author introduced writers' workshops to his firm-internal advanced study group meetings with the goal of developing and refining firm-specific material. The aforementioned Composite pattern variation and the Navigator pattern, as well as other published patterns like Value Object, Serializer, and Role Object went through such writers' workshops and benefited greatly, both in terms of contents and presentation.

A firm-internal workshop can be pragmatic, and pattern descriptions and other materials can be run through it multiple times, before they are considered polished documentation. Feedback always included both form and contents. The author has found participants to be quite enthusiastic about it, mostly because they were familiar with the material and the examples and can provide excellent feedback, refinement, and generalization.

The firm itself benefits from better documentation, a standardized vocabulary, and in the end more effective software developers.

6 Related Work

The main body of this article already mentions most of the related work, and this Section summarizes it and relates it to the work presented in this article.

In general, the usefulness of patterns in software systems design has been anecdotally reported about many times [1] [3] [6] [7] [10] [11] [12].

Also, many have reported on the breadth of applicability of patterns [2] [3] [7] [13] [24] that goes beyond design and extends to architecture and programming as

well as specific domains like user interfaces and business processes [24] [27] [28] [29] [30].

Other disciplines have made a case for design languages and a broader focus on the vocabulary used in innovation and design [8] [10] [23] without necessarily applying this thinking to software design or design patterns.

This article goes beyond this related work by summarizing the benefits of using and applying software patterns, as learned in industry projects (Section 3). It extends these experiences into the broader realm of design languages, which have been considered and worked upon, but to the best of this author's knowledge not in this form for software design (Section 4). Finally, this article presents specific lessons learned and guidance for using and applying design patterns through building up firm-internal know-how around a firm's design language using study groups and writers' workshops (Section 5). While study groups and writers' workshop have been researched in general, to the best of this author's knowledge they have not been as coherently integrated with the software development process and the use and application of design patterns as presented in this article.

7 Conclusions

Core activities of software development like design discussions on a white-board, implementation of such designs in code, as well as their documentation can greatly benefit from design patterns and the shared vocabulary they provide. Effective use of design patterns depends on avoiding common mistakes like those that equate a design pattern with its structure diagram and generated code. Combined with other software design concepts and embedded in a firm's design language, patterns are an important contribution to making software development more effective. This article confirms prior work on the general benefits of using design patterns and adds to it experiences with developing and maintaining a firm-specific design language. Such a design language needs to be nurtured through study groups and writers' workshops. The experiences of this article's author show that such a design language can make software development even more effective and more enjoyable than possible with standard design patterns alone.

Acknowledgments

I would like to thank the anonymous reviewers for helping me improve this article.

References

1. Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides. Design Patterns. Addison Wesley, 1995.
2. Christopher Alexander, S. Ishikawa, M. Silverstein. A Pattern Language. Oxford University Press, 1977.
3. Frank Buschmann, Regine Meunier, Hans Rohnert, Pete Sommerlad, Michael Stahl. Pattern-Oriented Software Architecture. Wiley, 1995.
4. Uwe Zdun, Paris Avgeriou. “Modeling Architectural Patterns Using Architectural Primitives.” In Proceedings of OOPSLA 2005. ACM Press.
5. Dirk Riehle. “Composite Design Patterns.” In Proceedings of OOPSLA 1997. ACM Press, 1997.
6. Dirk Riehle. “The Perfection of Informality: Tools, Templates, and Patterns.” Cutter IT Journal 16, 9. Page 22-26.
7. Dirk Riehle and Heinz Züllighoven. “Using Patterns in Software Development.” TAPOS 2, 1. Wiley, 1996.
8. Donald Schön. The Reflective Practitioner. 1983.
9. Richard P. Gabriel. Writers’ Workshops and the Work of Making Things. Addison Wesley, 2004.
10. John Vlissides. Pattern Hatching. Addison-Wesley, 1998.
11. Kent Beck, James O. Coplien, Ron Crocker, Lutz Dominick, Gerard Meszaros, Frances Paulisch. “Industrial Experience with Design Patterns.” In Proceedings of the 18th International Conference on Software Engineering, IEEE Press,1996.
12. Douglas C. Schmidt. “Experience Using Design Patterns to Develop Reusable Object-Oriented Communication Software.” Communications of the ACM (October 1995).
13. F. Buschmann, K. Henney, D.C. Schmidt. Pattern-Oriented Software Architecture Volume 5: On Patterns and Pattern Languages. John Wiley and Sons, 2007.
14. Dirk Riehle. “Value Object.” In Proceedings of the 2006 Conference on Pattern Languages of Programming (PLoP ’06). ACM Press, 2006.
15. Dirk Riehle. The JValue Framework for Java Value Objects. See http://www.jvalue.org.
16. Dirk Riehle, Wolf Siberski, Dirk Bäumer, Daniel Megert, Heinz Züllighoven. “Serializer.” In Pattern Languages of Program Design 3. Addison-Wesley, 1998. Chapter 17.
17. Erich Gamma. JHotDraw. See http://www.jhotdraw.org.
18. Bobby Woolf. “Null Object.” In Pattern Languages of Program Design 3. Addison-Wesley, 1998. Chapter 1.
19. Joshua Kerievsky. “A Learning Guide to Design Patterns.” See http://www.industriallogic.com/papers/learning.html.
20. The Sillicon Valley Patterns Group. See http://www.siliconvalleypatterns.org.
21. Tracy Bialik and Russ Ruffer. Personal Communication, 2005.
22. John Vlissides. Personal Communication, 2001.
23. Stewart Brand. How Buildings Learn: What Happens After they are Built. Penguin, 1994.
24. James Coplien, Douglas Schmidt (editors). Pattern Languages of Program Design. Addison-Wesley, 1995.
25. Paris Avgeriou, Uwe Zdun. “Architectural Patterns Revisited – A Pattern Language.” In Proceedings of the 10th European Pattern Languages of Programming Conference. Universitatsverlag Konstanz, 2005.
26. Neil Harrison. “Organizational Patterns for Teams.” In Pattern Languages of Program Design 2. Addison Wesley, 1996.

27. John Vlissides, James Coplien, Norm Kerth (editors). Pattern Languages of Program Design 2. Addison-Wesley, 1996.
28. Robert Martin, Dirk Riehle, Frank Buschmann (editors). Pattern Languages of Program Design 3. Addison-Wesley, 1998.
29. Neil Harrison, Brian Foote, Hans Rohnert (editors). Pattern Languages of Program Design 4. Addison-Wesley, 2000.
30. Dragos Manolescu, Markus Voelter, James Noble (editors). Pattern Languages of Program Design 5. Addison Wesley, 2005.
31. Dirk Riehle, Bruno Schäffer, Martin Schnyder. “Design of a Smalltalk Framework for the Tools and Materials Metaphor.” *Informatik/Informatique* (February 1996). Page 20-22.
32. Dirk Riehle. Framework Design: A Role Modeling Approach. Dissertation, No. 13509. Zürich, Switzerland, ETH Zürich, 2000.
33. Dirk Riehle, Steven Fraleigh, Dirk Bucka-Lassen, Nosa Omorogbe. “The Architecture of a UML Virtual Machine.” In *Proceedings of the 2001 Conference on Object-Oriented Programming Systems, Languages, and Applications* (OOPSLA ‘01). ACM Press, 2001. Page 327-341.
34. Walter Bischofberger, Michael Guttman, and Dirk Riehle. "Global Business Objects: Requirements and Solutions." In *Proceedings of the 1996 Ubilab Conference, Zürich.* Edited by Kai-Uwe Mätzel and Hans-Peter Frei. Konstanz, Germany: Universitätsverlag Konstanz, 1996. Page 79-98.