Context Is Key: The Power of Pattern Languages

by Steve Berczuk

"Do you know patterns?" This question often comes up in the context of employment interviews, technical exchanges, and other situations in which people want to find out how much a person knows about the state of the art. On the surface this is a reasonable question; patterns capture important knowledge about how to build systems. The question that one rarely hears is, "Do you know how to *use* patterns?" Patterns taken one at a time can require some skill to apply effectively.

Blindly applying patterns can be counterproductive. A common complaint is that new readers of *Design Patterns* [5] start applying patterns everywhere without much thought about why it might make sense. Rather than leading to better code, this approach leads to a mess. To fully provide benefit, the patterns need to be part of a pattern language that helps you to understand the *context* of each pattern.

In this article, I'll explain what context means for patterns, why it matters, and why you as a reader/learner don't really need to understand patterns as a form to benefit from a well-written pattern language. I will use examples from a pattern language that I developed with Brad Appleton, which is published in the book *Software Configuration Management Patterns* [3]. This pattern language illustrates how key software configuration management (SCM) practices interact with other aspects of an agile development environment, such as testing, to help you understand why the practices make sense.

WHAT ARE PATTERNS?

A "pattern" about a software system describes a known *good* solution that developers have used with success in the past. Patterns are not about new ideas; they are about making it easier for you to leverage the experience of experts in a domain. Patterns are an excellent mechanism for capturing tacit knowledge [7]. Patterns can be about any aspect of software development, ranging from architecture to organization structure to testing practices.

Patterns are not just about the format in which the knowledge is documented, although the particular format has advantages. Writing something in pattern form does not make it a pattern.

The dictionary definition of pattern is only superficially related to a pattern in the way that we are speaking; Merriam-Webster Online defines a pattern first as "a form or model proposed for imitation." Another definition is closer to what we're talking about here: "a discernible coherent system based on the intended interrelationship of component parts." To completely explain what a pattern is and what a pattern language is, we need to refer to the writings of the architect and builder Christopher Alexander, who inspired the writers of the earliest software patterns.

According to Alexander, a pattern "describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice" [2]. Alexander describes the "essential" quality of patterns:

> The pattern solves a problem. It is not merely "a" pattern, which one might or might not use on a hillside. It is a desirable pattern; for a person who wants to farm a hillside, and prevent it from erosion, he must create this pattern in order to maintain a stable and healthy world. In this sense, the pattern not only tells him how to create the pattern of terracing, if he wants to; it also tells him that it is essential for him to do so, in certain particular contexts, and that he must create this pattern there [1].

One aspect of patterns that is often overlooked is that a pattern does not exist in a vacuum. In *The Timeless Way of Building,* Alexander writes:

> We see, in summary, that every pattern we define must be formulated in the form of a rule which establishes a relationship between a context, a system of forces which arise in that context, and a configuration which allows these forces to resolve themselves in that context [1].

Any decision is based on decisions that we have made in the past, and each decision leads to other choices.

The context is what makes patterns more useful than other knowledge collections, such as "best practices." Context makes it explicit that the decisions we make do not occur in isolation. Any decision is based on decisions that we have made in the past, and each decision leads to other choices.

Most published patterns do not take advantage of context, leaving the readers to assemble the patterns on their own. An expert is more often a person who can put ideas together rather than someone who simply knows a number of unconnected ideas. Assembling patterns into a pattern language and making the context explicit is a great way to transfer knowledge of how to assemble small bits of experience into a useful whole. One could argue that "real" patterns only exist as part of a pattern language. The question is whether that connection to a pattern language is implicit (the pattern language in your head) or explicit (a documented pattern language).

When learning new things, we often need help in understanding the big picture. We tend to focus on details and short-term goals. By viewing the elements of a solution in context, we can grow to understand a problem and build a truly good solution.

Pattern languages are not just about a certain writing style or a certain form. When reading a pattern language, you do not even need to be aware of the form. An author can document a pattern language and even ignore a pattern form, and the reader will still benefit. The key things an author of a pattern language must consider are the relationship between the patterns in the language and writing those patterns in a way that expresses the context relationship. If you have an understanding of the "theory" of patterns, you may get more out of a pattern language, but if you do not, a pattern language will step you through the problem domain in a way that helps you understand how these things fit together.

WHAT IS CONTEXT?

A common, simple definition for pattern is "solution to a problem in a context." The idea of context is what makes patterns very useful and distinguishes patterns from best practices and other ways of capturing knowledge. Context is often not used to its full potential in the pattern world.

Merriam-Webster Online defines context as "the interrelated conditions in which something exists or occurs." This captures the basic idea of what context is, but it leaves open the question of how to define what these conditions are. In a pattern language, you capture these conditions by referring to the other patterns in the pattern language. You say that a pattern occurs *in the context of* another pattern.

Often, patterns address context by including an "intent" section that describes what the pattern is trying to do. This helps you narrow down whether the pattern is relevant, but it removes some precision. By defining the context in terms of patterns rather than colloquially, you leave less room for misinterpretation. Since pattern names can become part of the vocabulary of a team, referring to a specific pattern as "the context" gives you a concise and accurate way of describing what the preconditions for applying the pattern are.

Consider the following definitions of the conditions in which the Visitor pattern could occur:

- **Intent:** "Represent an operation to be performed on the elements of an object structure. Visitor lets you define a new operation without changing the classes of the elements on which it operates" [5].
- **Context:** When using a Composite or an Interpreter, you want to represent an

operation to be performed on the elements, without changing the classes of those elements.

The first bulleted item is the intent section of the Visitor pattern in *Design Patterns*. The second is a rewrite of it using the idea of context, adding information from the "Related Patterns" section of the same chapter. The intent version is a good general description that will make sense when you already have a general understanding of the problem. The context description guides you through building a system by combining patterns. The second is less general than the first, but it is also more useful for a learner.

It is helpful to think about patterns as structures that the pattern language shows you how to combine to build a bigger, useful structure. Patterns work to support each other. Saying that "Composite is part of the context for Visitor" means two things:

1. Consider using Visitor if you are going to use Composite.
2. Visitor supports Composite (or helps you to realize a Composite).

Say you are building a table and you want to talk about the relationship between the tabletop and the legs. The legs make sense in the context of the tabletop, and the tabletop works best when you have legs.

Many mathematically inclined people have tried to define a context relationship more precisely than "an interrelated condition." Some have used a UML contains definition; others have use "completes." If you are writing pattern languages, having a concrete understanding of what context means is important. If you are using a pattern, the formal meaning should fade into the background.

It is important to remember that context relationships are not about the temporal order in which you implement the patterns. They are about how the pattern structures relate to each other. One advantage of using context well is that since the patterns are closely related to each other by context, the patterns can, in turn, represent small, understandable units of information.

SOME PATTERNS FOR AGILE SOFTWARE DEVELOPMENT

To illustrate how a pattern language approach can be useful, I will describe how some of the patterns in a pattern language for using SCM in an agile environment fit together. This section will not describe the patterns in detail but will illustrate how a pattern language that places each pattern in context can be more useful than a simple catalog of patterns.[1]

The SCM pattern language is a bit different from many pattern collections because it addresses a crucial but often ignored aspect of software development — the way version control and build practices interact with testing practices to help establish the rhythm of the development process [6]. Its patterns cover diverse but related areas, such as:

- Code lines and branching
- Build and build management
- Developers' workspaces
- Testing (unit, integration, and regression)

The reason for covering all of this ground is that all of these elements are related. Each pattern supports the others. You could certainly decide to implement one alone, but your development process, and your product, would be the worse for it. Since the pattern language makes the dependency explicit, you are less likely to pick a pattern at random and find yourself surprised when it didn't turn out as planned.

A Sample Pattern

Here is an excerpt from one of the patterns in the SCM pattern language:

> *You have an evolving codeline that has code intended to work with a future product release. You are doing most of your work on a Mainline.*
>
> *When you are working in a dynamic development environment, many people are changing the code. Team members are working toward making the system better, but any change can break the system, and changes can conflict. This pattern helps you balance stability*

[1]There are other published patterns that use a pattern language structure; for example, Martin Fowler's *Patterns of Enterprise Application Architecture* [4].

and progress in an active development effort.

How do you keep a rapidly evolving codeline stable enough to be useful?

[Problem details omitted here]

Institute policies that are effective in making your main development line stable enough for the work it needs to do. Do not aim for a perfect active development line but for a mainline that is usable and active enough for your needs.

[Solution details omitted here]

The section in italics is the context section. The section in bold is the problem summary. In the complete pattern, a detailed description of the problem follows the problem summary. Then comes the solution summary (in bold italics above), followed by details of the solution.

Overview of the Language

Figure 1 shows the patterns in the language that this article discusses. The patterns toward the top of the diagram build on the patterns on the bottom, and each pattern makes sense in the context of the ones above it.

Mainline and Active Development Line

The pattern language starts with the idea of developing code on a single development line, a Mainline. There are other codeline models, but agile development favors simplicity, and the simplest development structure is a Mainline. One can, however, do Mainline development and have your project proceed at a glacial pace. The Active Development Line pattern says, in the context of a Mainline environment, let your codeline evolve rapidly, but keep it useful. This pattern depends on a number of other patterns that allow for a build and test environment, as well as for alternate codelines to support tasks where "active" isn't best. Here are some of the issues that you need to address:

- You need to allow developers to work on their tasks while the codeline is changing. Private Workspace addresses this.
- You have to make sure that the developers are not likely to check in a change that will break the build. Private System Build provides a way for developers to do a build locally that matches the centralized integration build. Smoke Test gives developers a set of tests to run (quickly) to ensure that the application isn't broken.
- Once you have released an application and need to provide patches, you want to emphasize caution over speed. The pattern language includes a Release Line pattern that supports Active Development Line.
- Given that there will be different types of codelines, you need a way to express how each should be used. Codeline Policy does this.

Smoke Test illustrates the interdependence of the patterns very clearly. There is a tendency in organizations to err on the side of caution and require exhaustive testing. Tests that run too long can cause the rate of change to the codeline to be very low. On the other hand,

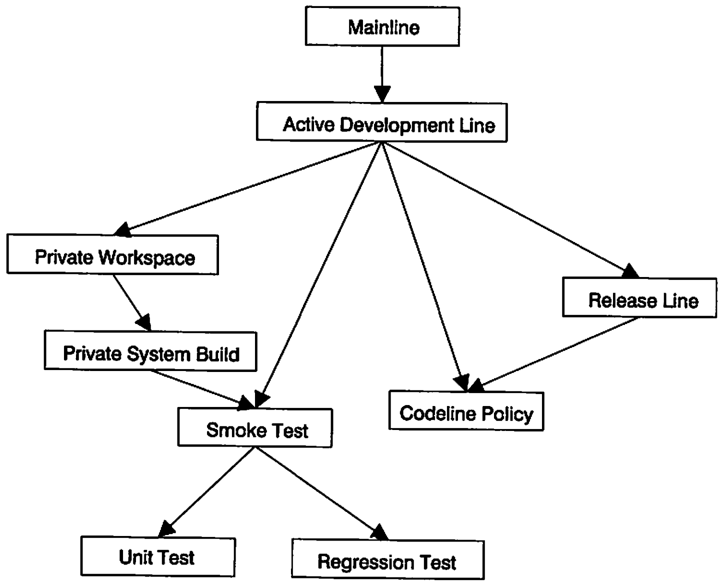

Figure 1 — Some of the patterns in the SCM pattern language.

short tests will catch fewer errors. You balance these forces by providing for Unit Tests that the developers can run to test the local change more comprehensively, as well as thorough regression tests that may run on the integration machine. Each set of tests reduces the possibility of an error making it all the way through the system while allowing for speed where it matters most.

THE MISSING LINKS

Why don't we see more use of context in published patterns? The answer varies from person to person. One reason is that the idea of context is hard to understand, and the writer of a set of patterns has often internalized the context relationships and isn't aware of what is missing. Another reason is that some patterns authors get overly concerned about form (the how) and don't step back to realize that the language is the harder thing to explain. Also, there are often intellectual property issues involved in relating to other patterns.

None of this is to say that there isn't value in many of the published patterns that are out there now. They are very useful tools for building good systems once you understand the basic vocabulary. The next step for patterns authors is to write patterns and pattern languages that help the novice get up to speed quickly. Software development is complicated enough that hands-on learning and mentoring will be the best way to learn for some time, but with some work on pattern languages, we can make it easier to share the important part of knowledge: how to put things together.

To build good systems, we need to be more aware of the connections between different parts of a development process. Patterns and pattern languages are excellent tools for capturing knowledge that spans areas of expertise.

REFERENCES

1. Alexander, Christopher. *The Timeless Way of Building*. Oxford University Press, 1979.

2. Alexander, Christopher, Sara Ishikawa, and Murray Silverstein, with Max Jacobson, Ingrid Fiksdahl-King, and Shlomo Angel. *A Pattern Language*. Oxford University Press, 1977.

3. Berczuk, Stephen P., and Brad Appleton. *Software Configuration Management Patterns: Effective Teamwork, Practical Integration*. Addison-Wesley, 2002.

4. Fowler, Martin. *Patterns of Enterprise Application Architecture*. Addison-Wesley, 2003.

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

6. Kane, David, David Dikel, and James R. Wilson. "Patterns for Managing the Rhythm of E-Commerce." *Cutter IT Journal*, Vol. 14, No. 1, January 2001, pp. 20-29.

7. May, Daniel, and Paul Taylor. "Knowledge Management with Patterns." *Communications of the ACM*, Vol. 46, No. 7, July 2003, pp. 94-99.

Steve Berczuk is an independent consultant based in Arlington, Massachusetts. He has been developing object-oriented software applications since 1989. Mr. Berczuk is the coauthor (with Brad Appleton) of Software Configuration Management Patterns: Effective Teamwork, Practical Integration. *He has an M.S. in operations research from Stanford University and an S.B. in electrical engineering from MIT.*

Mr. Berczuk can be reached at E-mail: steve@berczuk.com; Web site: www.berczuk.com.