Patterns & XP

Joshua Kerievsky
Industrial Logic, Inc
Joshua@industriallogic.com
January, 2000

Abstract

Patterns and Extreme Programming (XP) both provide invaluable aid to those who design and develop software. But XP has thus far focused heavily on Refactoring while remaining all but silent about Patterns. In this paper, I ask why, and ultimately show how Patterns are better when they are implemented the XP way, and how XP is better when it includes the use of Patterns.

Acknowledgements

Many thanks to Kent Beck, Martin Fowler and Ward Cunningham for kindly reviewing this paper.

We start our programming careers not knowing very much and producing software that reflects our inexperience: we create code that is bloated, buggy and brittle, hard to maintain and hard to enhance. Over time, we become better software designers: we learn from authors, experts and our own mistakes. Now we write highly flexible software that is generic and robust. When asked to write a new system, we know to inquire about current and future requirements, so that we can design software to handle current and future needs.

At this stage in our careers, Extreme Programming tells us that we often over-engineer software. We've learned from our mistakes and don't want to repeat them, so we make great efforts to produce flexible and robust designs early in the life of a system. Unfortunately, we don't realize that all our work will be meaningless and wasteful if the system never needs such degrees of flexibility and robustness. We've over-engineered.

I've over-engineered. Honestly, it's kind of fun to sit in a room with other designers and think how to design software to accommodate many current and future requirements. We get to take all the lessons we've learned, especially the best practices, and apply them to the design. We often know that the list of requirements will change, but users or customers are always changing requirements. Nevertheless, we think we can be brilliant enough to design software that will be so flexible that it won't really be a problem when the requirements change.

Classic over-engineering.

Today, Extreme Programming would expose this folly. It says that we must learn to let designs emerge and not anticipate what will be. XP says "*do the simplest thing that could possibly work*" because "*you aren't gonna need it.*" And Kent Beck says:

> You need to choose the best way to work within a value system of communication, simplicity, feedback, and courage to keep you from over-engineering out of fear [Beck1 00].

Agreed. However, I must now quote my friend Norm Kerth. Norm has been around a while and has seen a lot. A year ago I asked him what he thought of XP. He said:

> I like everything that is in XP. What concerns me is what is not in XP [Kerth 99].

At the time, I just thought Norm was being conservative. But now I'm not sure.

Noticeably absent from XP is the practice of using Patterns. While some of XP's founders helped pioneer and build the patterns community, no one has yet strongly articulated how Patterns fit into XP.

For a while, that simply didn't bother me. But now, it does.

It bothers me because my experiences with Patterns and XP lead me to believe that using Patterns is better in the context of XP practices, and XP practices are better when they include Patterns.

This will take some explaining. I'll start by describing some of my experiences with Patterns and XP.

Starting in 1995, I began to immerse myself in Patterns. I was learning the Patterns literature, leading a weekly study group on Patterns, designing and developing software with Patterns, and organizing and running UP (an international conference about Using Patterns). It would be an understatement to say that I was enthusiastic about Patterns.

At the time, like many who first learn Patterns, I was a bit over-anxious to use them. That is not a good thing because it can make designs more complex than they need to be. But I wouldn't learn that until I started to learn about Refactoring.

Around 1996, I was first exposed to Refactoring. I started experimenting with it and quickly observed that Refactoring was leading me *away* from certain principles I'd learned in my studies of Patterns.

For instance, one of the mantras of the landmark book, *Design Patterns: Elements of Reusable Object-Oriented Software*, is:

> *Program to an interface, not an implementation* [GHJV1 95]

The authors of *Design Patterns* do an excellent job of explaining why you'd want to follow this advice. In nearly every pattern, there is a discussion of how your software becomes less flexible and less changeable when you program to a specific implementation. Interfaces nearly always come to the rescue.

But what if you don't need that flexibility or changability? Why begin a design anticipating a need that may never arise? This was an awakening for me. So around that time I recorded the following Java Idiom:

> **Don't Distinguish Between Classes And Interfaces**
>
> I used to place an "I" at the end of the names of my interfaces. But as I continue to learn more about the rhythm of refactoring, I'm starting to see the wisdom in making class and interface names look the same. Here's why: during development you know that you could use an interface to make something really flexible (vary the implementation) but there may be no real need to vary the implementation today. So instead of "over designing" by anticipating too much, you stay simple and make the thing a class. And somewhere you write a method signature that expects an object of that class type. Then, a few days, weeks, months later, there is a definite

> "need" for the interface. So you convert the original class into an interface, create an implementation class (that implements the new interface) and let your original signature (or signatures) remain unchanged [Kerievsky 96].

I continued to learn similar lessons as I studied Refactoring, and gradually, the way I used Patterns began to change. I would no longer develop full-blown implementations of a pattern up front. Now, I would be more judicious: if a Pattern could solve a design problem, if it could provide a way to implement a requirement, I would use it, but I would begin with the simplest implementation of the Pattern that I could code. Later, when enhancements or modifications were required, I would make the implementations more flexible or robust.

This new way of working with Patterns was much better: It saved me time and made my designs simpler.

As I continued to learn more about XP, I soon began to consider the fact that those who were articulating what XP is, and how it works, weren't saying anything about Patterns. The focus on the development side seemed to have shifted exclusively to Refactoring. Build a little, test a little, refactor a little, and repeat.

Well, what happened to Patterns?

The common answer I received was that Patterns encourage over-engineering, while Refactoring keeps things simple and light.

Now, I like Refactoring about as well as anyone – I reviewed two manuscripts of Martin Fowler's book on the subject and knew that it was destined to become a classic. But I also like Patterns and have found them to be invaluable in helping people learn how to design better software. So how could XP not include Patterns?!

I wrote about my unease with respect to this issue on the *Portland Pattern Repository*. I asked if the perfect XP team would consist of programmers and a coach who know nothing about Patterns, but who rely solely on Refactoring to "*let the code go where it needs to go.*" Ron Jeffries, who is arguably the world's most experienced XP practitioner, debated this subject with me and wrote:

> A beginner can't listen to the code and hear what it says. He needs to learn patterns (in the generic sense) of code quality. He needs to see good code (and, I suppose, bad) in order to learn to make good code.
>
> A question, and I mean it to be a question, is whether patterns as presently constituted help with this. I think Beck's Smalltalk Best Practice Patterns do help, because they are very micro. I think Design Patterns are more iffy, as the patterns and discussion get pretty big sometimes, and they may make big solutions seem desirable. Martin Fowler's excellent Analysis

> Patterns offer the same peril, the selection of a big solution when a small one would do. [Jeffries 99]

A very interesting perspective on Patterns. While I've come to see that Patterns can be implemented and used judiciously, Ron seems to think that they are dangerous because they "*make big solutions seem desirable*." Elsewhere, Ron observes the common occurrence of how people who first learn Patterns are over-anxious to use them.

I can't argue with the latter observation. Like anything new, even XP practices, people can be over-anxious to use them. But do patterns really encourage big solutions when small ones would do?

I think a lot depends on how you define and use Patterns. For instance, I have observed that many beginning Patterns users think that a Pattern is the same as its structure diagram (or Class diagram). Yet when I point out to them that the Pattern can really be implemented in different ways, depending on needs, they begin to see that the diagram shows just one of many ways to implement the Pattern.

There are simple implementations of a pattern, and sophisticated implementations. The trick is to discover the problem that a pattern addresses, match that problem to you current problem, and then match the simplest pattern implementation (solution) to your problem. When you do that, you aren't using big solutions when small ones would do. You're leveraging best practices to solve your problems.

Difficulties can arise when people aren't well educated about Patterns. Ron mentioned the way that Patterns are "presently constituted"—that is to say, how they are communicated by authors today. I would agree that the Patterns literature has some flaws. The books on Patterns are dense, and it can take time to understand the problems that Patterns solve so that you can intelligently match a Pattern to your particular needs.

This matching is extremely important. If you get it wrong, you could be over-engineering or just screwing up your design altogether. Experienced Patterns users do make mistakes and then often see the problems that result. But these experts are armed with a host of other Patterns that can help them in the face of their mismatch. So they often end up swapping out a less-than-ideal Pattern for one that better suits their needs.

So how do you become an experienced Patterns user? I have found that unless people devote significant study to Patterns, they will be in danger of misunderstanding them, over-using them and over-engineering with them.

But is that a reason to avoid them?

I think not. I've found them to be so useful, on so many projects, that I couldn't imagine designing and developing software without them. I believe that a thorough study of Patterns is well worth the effort.

So is XP keeping silent about Patterns because the feeling is that they'll be misused?

If that is the case, perhaps the question becomes how can we leverage the wisdom in Patterns, while avoiding the misuse of Patterns within the context of XP development?

Here, I think we must return to the Design Patterns Book. In the concluding remarks under the section "What to Expect From Design Patterns" and the sub-section, "A Target for Refactoring", the authors write:

> Our design patterns capture many of the structures that result from refactoring. Using these patterns early in the life of a design prevents later refactorings. But even if you don't see how to apply a pattern until after you've built your system, the pattern can still show you how to change it. Design patterns thus provide targets for your refactorings. [GHJV2 95]

This is the idea we need: *Targets for your Refactorings*. This is the bridge between Refactoring and Patterns. It perfectly describe my own evolution in how I've come to use Patterns: start simple, think about Patterns but keep them on the back-burner, make small refactorings, and move these refactorings towards a Pattern (or Patterns) only when there is a genuine need for them.

This process, which requires discipline and careful judgement, would fit nicely into the XP fold of best practices.

And this approach is certainly quite different from intentionally not knowing about or using Patterns but simply relying on Refactoring to incrementally improve a design.

The danger of relying exclusively on Refactoring is this: without targets, people may make small design improvements, but their overall design will ultimately suffer because it lacks the order, simplicity, and effectiveness that come from intelligently using Patterns.

To quote Kent Beck himself: Patterns Generate Architectures [Beck2 94].

But Patterns don't ensure disciplined usage. If we use them too much or too soon in a design, we're back to the over-engineering problem. We must therefore answer the question, "*When is it safe to introduce Patterns into the life cycle of a design?*" Recall the quote from the Design Patterns book above:

> Using these patterns early in the life of a design prevents later refactorings.

This is a tricky proposition. If we don't know the ground rules for when to deploy a Pattern, then we can easily over-engineer early in the design lifecycle.

Again, it all comes down to matching a project's problems to the correct Patterns.

I must here recount certain experiences I've had developing software for various industries.

For one client, my team and I were asked to create software in Java that would be the cool, interactive version of their Web site. The client did not have any Java programmers, but nevertheless wanted this software to be written in such a way that they could modify its behavior wherever and whenever they wanted, without having to make programming changes. A tall order!

After some analysis of their needs, it became clear that the Command pattern would play an essential role in the design. We would write Command objects, and let these Commands control the entire behavior of the software. The users would be empowered to parameterize the Commands, order them, and choose where and when they would run.

This solution worked perfectly, and the Command pattern was the key to our success. So here we didn't wait to Refactor our way to using the Command Pattern. Instead, we saw a need up front, and programmed the software using Command from the start.

On another project, a system was required to run as a stand-alone application *and* on the Web. The Builder pattern played a huge role in this system. Without it, I shudder to think what sort of bloated design would been cobbled together. The Builder Pattern simply lives for solving problems like how to run on different platforms or in different environments. It was therefore a good early choice of a Pattern.

At this point, I must make it clear that even though Patterns were introduced early in the design lifecycle, they were still implemented in their most primitive forms to start. Only later, when additional functionality was required, were these implementations either altered or upgraded.

An example will make this clear.

The above-mentioned software that was controlled exclusively via Commands was implemented with multi-threaded code. There were times when two threads might be using the same MacroCommand to run a sequence of Commands. But we had not originally bothered to make our MacroCommand thread-safe. So when we started to encounter weird bugs because of this, we had to reconsider our implementation. The question was, would it make sense to invest time to make our MacroCommand thread safe or was there an easier way to solve the problem?

It turned out that the easier way to solve the problem, and avoid over-engineering, was to simply have two separate instances of MacroCommand used by each thread. We were able to implement that solution in 30 seconds. Compare that to the time it would have taken to implement a thread-safe MacroCommand.

This example shows how the XP philosophy of keeping things simple has such an affect on how Patterns are programmed and used. Without the drive towards simplicity, over-engineered solutions, like the thread-safe MacroCommand, can easily proliferate.

So the relationship between simplicity and Patterns is important.

When programmers need to make design decisions, it's important that they try to keep their designs simple, since simple designs are usually far easier to maintain and extend than large, complex designs. We already know that Refactoring is meant to keep us on the simple path: It encourages us to take small, simple steps, to improve our designs incrementally, and to avoid over-engineering.

But what about Patterns? Do they help us stay simple?

Some would argue that they don't. They think that Patterns, while useful, tend to complicate designs. They see Patterns as causing a proliferation of objects and an over reliance on object composition.

This perspective is really the result of a naïve understanding of how to successfully use Patterns. Once again, experience equips a Patterns user to avoid complicated designs, proliferations of objects, and too much object composition.

Experienced users of Patterns actually make their designs simpler when they use Patterns. Again, I'll go to a real example to make my point clear.

JUnit is the simple, useful, Java testing framework, written by Kent Beck and Erich Gamma. It is an excellent piece of software, dense with well-selected and simply implemented Patterns.

As an experiment, I recently asked some folks to DeGoF JUnit; that is, to remove the Design Patterns from JUnit to see what it would look like without them. This was a very interesting exercise, since it made the participants think really hard about when it is appropriate to introduce a Pattern into a system.

To illustrate a lesson they learned, we will DeGoF a few extensions that were added to JUnit in version 2.1.

JUnit has an abstract class called TestCase, from which all concrete test classes descend. TestCase provides no way to run a test multiple times, nor does it provide a way to run a test within its own thread. Erich and Kent implemented repeatable tests and thread-based tests quite elegantly using the Decorator Pattern. But what if a team or a pair of programmers didn't know Decorator? Let's see what they might develop and assess how simple it would be.

Here's what Test Case looked like in version 1.0 of the JUnit framework (*comments and numerous methods omitted for brevity*):


```
public abstract class TestCase implements Test {
	private String fName;

	public TestCase(String name) {
		fName= name;
	}

	public void run(TestResult result) {
		result.startTest(this);
		setUp();

		try {
			runTest();
		}
		catch (AssertionFailedError e) {
			result.addFailure(this, e);
		}
		catch (Throwable e) {
			result.addError(this, e);
		}

		tearDown();
		result.endTest(this);
	}

	public TestResult run() {
		TestResult result= defaultResult();
		run(result);
		return result;
	}

	protected void runTest() throws Throwable {
		 Method runMethod= null;
		 try {
		     runMethod= getClass().getMethod(fName, new Class[0]);
		 } catch (NoSuchMethodException e) {
		             e.fillInStackTrace();
		     throw e;
		 }

		 try {
		     runMethod.invoke(this, new Class[0]);
		 }
		 catch (InvocationTargetException e) {
		             e.fillInStackTrace();
		     throw e.getTargetException();
		 }
		 catch (IllegalAccessException e) {
		             e.fillInStackTrace();
		     throw e;
		 }
	}

	public int countTestCases() {
		return 1;
	}
}
```

The new requirements call for allowing tests to run repeatedly, in their own threads, or both.

Inexperienced programmers usually sub-class when they get new requirements like this. But here, since they know that some TestCases will need to be able to run repeatedly in a thread or repeatedly run TestCases in separate threads, the programmers know that they need to give this some more thought.

One way to implement this would be to just add all the functionality to the TestCase class itself. Many developers, especially those who don't know Patterns, would do this without worrying about the negative effects of bloating their classes. They have to add functionality, so they'll add it where they can. The following code might be their implementation:

```
public abstract class TestCase implements Test {
	private String fName;
	private int fRepeatTimes;

	public TestCase(String name) {
		this(name, 0);
	}

	public TestCase(String name, int repeatTimes) {
		fName = name;
		fRepeatTimes = repeatTimes;
	}

	public void run(TestResult result) {
		for (int i=0; i < fRepeatTimes; i++) {
			result.startTest(this);
			setUp();

			try {
				runTest();
			}
			catch (AssertionFailedError e) {
				result.addFailure(this, e);
			}
			catch (Throwable e) {
				result.addError(this, e);
			}

			tearDown();
			result.endTest(this);
		}
	}

	public int countTestCases() {
		return fRepeatTimes;
	}
}
```

Notice how the `run(TestResult result)` method is a little bigger. They've also added another constructor on TestCase. No big deal so far. And here we could say that if this was all they had to do, using Decorator would be overkill.

Now, how about running a TestCase in its own thread? Again, here is another possible implementation:


```
public abstract class TestCase implements Test {
	private String fName;
	private int fRepeatTimes;
	private boolean fThreaded;

	public TestCase(String name) {
		this(name, 0, false);
	}

	public TestCase(String name, int repeatTimes) {
		this(name, repeatTimes, false);
	}

	public TestCase(String name, int repeatTimes, boolean threaded) {
		fName = name;
		fRepeatTimes = repeatTimes;
		fThreaded = threaded;
	}

	public void run(TestResult result) {
		if (fThreaded) {
			final TestResult finalResult= result;
			final Test thisTest = this;
			Thread t= new Thread() {
				public void run() {
					for (int i=0; i < fRepeatTimes; i++) {
						finalResult.startTest(thisTest);
						setUp();

						try {
							runTest();
						}
						catch (AssertionFailedError e) {
						   finalResult.addFailure(thisTest, e);
						}
						catch (Throwable e) {
						   finalResult.addError(thisTest, e);
						}

						tearDown();
						finalResult.endTest(thisTest);
					}
				}
			};
			t.start();
			result = finalResult;
		} else {
			for (int i=0; i < fRepeatTimes; i++) {
				result.startTest(this);
				setUp();

				try {
					runTest();
				}
				catch (AssertionFailedError e) {
					result.addFailure(this, e);
				}
				catch (Throwable e) {
					result.addError(this, e);
				}

				tearDown();
				result.endTest(this);
```

```
                }
            }
        }

        public int countTestCases() {
            return fRepeatTimes;
        }
}
```

Hmm, this is starting to look pretty bad. We now have three constructors to support these two new features, and the `run(TestResult result)` method has mushroomed in size.

Despite all the new code, our programmers have still not met the requirements: we still can't run repeated tests that each execute in their own thread. We'd have to add more code for that. I'll spare you.

Refactoring could help this code a little. But consider for a moment what we'd have if just one more requirement comes in. JUnit 3.1 now supports four different TestCase Decorators, which can be easily combined to get the functionality you need. And yet the JUnit implementation is simple–it doesn't create cluttered code. It keeps the TestCase class simple and lightweight by decorating TestCases only when needed, and in whatever order or combinations a user likes.

This is clearly an example of how Patterns help to keep designs simple. It also shows how inexperienced developers can improve their designs by knowing which Patterns to target during Refactorings.

Using Patterns to develop software is intelligent, but if you lack experience with Patterns, it can also be dangerous. For this reason, I am a great advocate of Patterns Study Groups. Such groups allow people to become proficient with Patterns at a steady pace with the help of their peers.

Patterns are most useful when people know them and use them in a disciplined way: The XP way. Using Patterns the XP way encourages developers to keep designs simple and Refactor to Patterns solely based on need. It encourages the use of Patterns early in a design when they are critical. It encourages the correct matching of problems with Patterns that help solve them. And finally, it encourages developers to write simple implementations of Patterns, which they may evolve as needed.

Patterns are indeed more useful in the context of XP, and XP development is more likely to succeed when it includes the use of Patterns.

References

[Beck1 00] Beck, Kent. Email on extremeprogramming@egroups.com, January 2000.

[Beck2 94] *Patterns Generate Architectures*, Kent Beck and Ralph Johnson, ECOOP 94

[GHJV1 95] *Design Patterns: Elements of Reusable Object-Oriented Software*, by Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides.

[GHJV2 95] *Design Patterns: Elements of Reusable Object-Oriented Software*, by Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides. Pages 353-354

[Jeffries 99] Jeffries, Ron. Patterns And Extreme Programming. *Portland Pattern Repository*. December, 1999

[Kerth 99] Kerth, Norm. Conversation, circa March, 1999.

[Kerievsky 96] Kerievsky, Joshua. Don't Distinguish Between Classes And Interfaces. *Portland Pattern Repository*. Circa 1996

