

Articles and events of interest to the software development community

In This Issue:

Can Patterns Be Harmful?

- p.1

Net Objectives Courses

- p.3

Upcoming Seminars

- p.6

Employee Spotlight: Doug Shimp

- p.9



Address:
25952 SE 37th Way
Issaquah WA 98029
Telephone:
(404)593-8375
Email:
info@netobjectives.com

Can Patterns Be Harmful?

By Alan Shalloway

***Originally Published in Cutter IT Journal
September 2003***

Let me answer this unequivocally. *It depends.* Figured out I'm a consultant yet? Actually, let me explain. It *does* depend. It depends on what you think patterns are. Many people have a misunderstanding of patterns, and for those people, patterns *can* be harmful. However, when people get past their early understanding of patterns, and come to comprehend them in a richer way, *patterns cannot be harmful.* This article will explain both views, so I think that by the end of it, I will have answered the question unequivocally.

What is a Pattern?

I like to start by agreeing on terms. A common definition for a "pattern" is "a solution to a problem for a given context." Given that context, (a version of) that solution should work again, although you may have to tweak it a bit to make it work.

We find patterns by looking for problems that occur again and again in similar situations (the context). For a given problem, if we can identify a set of high quality solutions that take a similar approach to solving it, then we call that approach "the pattern" – the solution to the problem for that situation.

That is fairly abstract. Let's turn to a concrete example for software engineering: The Strategy Pattern.

The Strategy Pattern

Suppose your software has several different algorithms that it can use. How does it choose which one to use? And how can you introduce more algorithms to choose from in a way that is

easy to maintain? That is the *recurring problem.*

The challenge is that there are several actions or "behaviors" going on here: choosing the algorithm to use and then using the chosen algorithm. If one object or one software module must do both, things become complicated in a hurry. Each time you add another algorithm, the management of the set of algorithms and the task of choosing becomes more complex.

The *context* is that the object that is choosing the algorithm to use should not be "coupled" – tightly tied to – that particular algorithm. We want a solution that is very flexible from run to run that makes it easy to add new algorithms whenever we want.

The *common solution* for the problem and context is to treat all the algorithms in a similar way, to make them interchangeable as far as the choosing object is concerned. This requires every algorithm to implement a common interface and then invoking an algorithm through that common interface.

The description of the solution may be described as follows:

1. Make all of the algorithms have the same interface.
2. Separate the task of "using an algorithm" from "choosing which algorithm to use." (In the pattern nomenclature, the using object is the **Context** and the object calling the **Context** is the **Client**. The **Client** calls the **Context**. You transfer the role of choosing which algorithm to use from the **Context** to the **Client**.)
3. Make the **Client** pass a reference to an algorithm to the **Context**. This frees, thereby freeing the **Context** from making any decisions about which algorithm to use.

This solution “solves” the problem because you can now add more and more algorithms without increasing the complexity of the **Context** object.

This is a proper use of the Strategy Pattern. Later I’ll show a case of an improper use of the Strategy Pattern.

Examples of a Strategy pattern design and an implementation for a class that needs to be able to perform different kinds of encryption are shown in Figures 1 and 2, respectively.

Note that in this example, we have a Configuration object that knows which algorithm to use. It could just as well been the **Client** object that knew which algorithm to use. The Strategy Pattern does not specify which object does the deciding of which algorithm to use. The pattern just states that the **Context** object does not do the deciding. It tells you to put the “choosing logic” outside the scope of the “using object.”

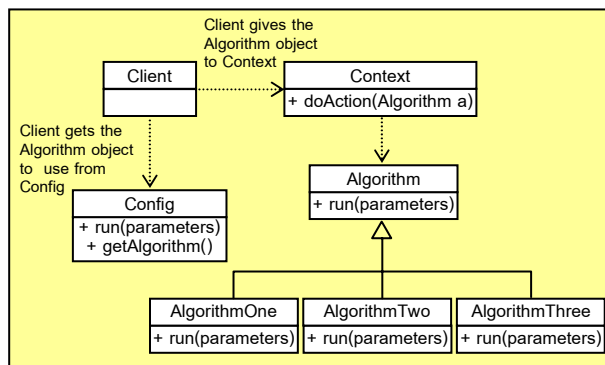


Figure 1. Class diagram for a Strategy Pattern to perform different types of encryption

```

class Client {
    Algorithm myAlgorithm;
    myAlgorithm= Config.getAlgorithm();
    . . .
    myContext.doAction( myAlgorithm);
}

class Context {
    public void doAction( Algorithm
anAlgorithm) {
    . . .
    anAlgorithm.run(s);
    . . .
}
}
  
```

Figure 2. Code fragment for a Strategy Pattern to perform different types of encryption

What Are Patterns Really About?

Many people believe that what the Strategy Pattern says is, “when you have a set of potential algorithms to use, create an interface for all of the algorithms and have the class that uses the algorithms receive a reference to a specific algorithm from a client object”. This shows the problem in a context and shows how the problem is solved. It boils the pattern down to a class diagram solution, detailing which class or object contains what. Unfortunately, while this is useful information, it is only part of the pattern – a possible way in which the pattern may manifest itself. The truly more valuable aspect of the pattern is not in the class diagram.

Patterns Resolve Forces

Christopher Alexander states as much at the end of The Timeless Way of Building by saying that:

“At this final stage, the patterns are no longer important: the patterns have taught you to be receptive to what is real.”¹

In my classes, I joke that instead of telling me this on page 545 of a 549-page book, Alexander could have told me on page 2 and saved me a lot of reading! However, I go on to say that by the time someone reads this, he or she already knows this to be true.

“The patterns aren’t important – the forces in the problem domain and how to resolve them are.”

I prefer to talk about patterns in terms of the “forces” (the essential constraints and goals and requirements) in the problem and particular context: what those forces are, how they relate to each other, and how they can be resolved. It is the relationship of these three issues – problem, context, solution, and the possible resolutions – that is really what comprises the pattern.

¹ Christopher Alexander is the architect often credited with inspiring many experts in the software community to investigate patterns in software. His books The Timeless Way of Building [1] and A Pattern Language [2] offer tremendous insights to the software design patterns community.

My hope is to help you to see patterns in this way now: rather than simply seeing a particular implementation of a pattern, to be able to step back and see forces in a problem and how they are being resolved. This gives you the power and flexibility to address all sorts of problems in the future rather than being locked into one specific instance.

Mandates to help you see

As a first step in shifting our view from the implementation of the Strategy Pattern to the

forces of the Strategy Pattern, let's look at the primary mandates of the "Gang of Four."² Not surprisingly, the Strategy pattern reflects all of these mandates:

1. Program to an interface, not an implementation.
2. Favor object-composition over class inheritance.
3. Consider what should be variable in your design. Focus on encapsulating the concept that varies.

Net Objectives - What We Do

When you've taken a course from Net Objectives, you will see the world of software development with new clarity and new insight. Our graduates often tell us they are amazed at how we can simplify confusing and complicated subjects, and make them seem so understandable, and applicable for everyday use. Many of our students remark that it is the best training they have ever received.

The following courses are among our most often requested. This is not a complete list, though it is representative of the types of courses we offer

Agile Development Best Practices - This 2-day course analyzes what it means to be an agile project, and provides a number of best practices that provide and/or enhance agility. Different agile practices from different processes (including RUP, XP and Scrum) are discussed.

Use Case Based Requirements Analysis - This 3-day course provides theory and practice in deriving software requirements from use cases. This course is our recommendation for almost all organizations, and is intended for audiences that mix both business and software analysts.

Effective Object-Oriented Analysis, Design and Programming In C++, C#, Java, or VB.net - This 3,4, or 5-day course goes beyond the basics of object-oriented design and presents 14 practices which will enable your developers to do object-oriented programming effectively. These practices are the culmination of the best coding practices of eXtreme Programming and of Design Patterns.

Design Patterns Explained: A New Perspective on Object-Oriented Design - This 3-day course teaches several useful design patterns as a way to explain the principles and strategies that make design patterns effective. It also takes the lessons from design patterns and expands them into both a new way of doing analysis and a general way of building application architectures.

SW Dev Using an Agile (RUP, XP, SCRUM) Approach and Design Patterns - This 5-day course teaches several design patterns and the principles underneath them, the course goes further by showing how patterns can work together with agile development strategies to create robust, flexible, maintainable designs

Refactoring, Unit Testing, and Test-Driven Development - This 2 day course teaches how to use either Junit, NUnit or CppUnit to create unit tests. Lab work is done in both Refactoring and Unit Testing together to develop very solid code by writing tests first. The course explains how the combination of Unit Testing Refactoring can result in emerging designs of high quality.

² The "Gang of Four" is a colloquialism for Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides, authors of *Design Patterns: Elements of Reusable Object-Oriented Software*[4].

Volume 1, Issue 1
January 2004
©2004, Net Objectives,
All Rights Reserved



Address:
25952 SE 37th Way
Issaquah WA 98029
Telephone:
(404)593-8375
Email:
info@netobjectives.com

Program to an interface, not an implementation.

Essentially this means considering the object you are dealing with as a black box. Do not consider how it *does* its tasks, just consider the *interface* (set of public methods) that it allows you to use. This is very consistent with Bertrand Meyer's notions of "Design by Contract."³ The Strategy pattern follows this approach because each **Algorithm** can only be accessed by their interface. The **Context** object is not aware of any algorithm's implementation.

Favor object-composition over class inheritance.

When learning object-oriented design, many developers are taught that when an alternative way is needed to accomplish something, inheritance should be used. In other words, if you have a class **Chip** that gets a status of a hardware component, stores it in a string, encrypts that status and then transmits it, you can add a new kind of encryption by subclassing **Chip** and overriding the encrypt method (see Figure 3).

This works well for handling one variation. However, if something else starts to vary (say, how you transmit the status or whether you need to compress it before encrypting), you get an exponentially increasing number of possible combinations. Using inheritance alone requires a correspondingly large number of classes (one for each combination). In older code, we often find an increasingly complex inheritance hierarchy handling many, many special cases and becoming more brittle and harder to change as time goes by.

On the other hand, if you use the Gang of Four's suggestion, you would have the **Chip** object contain a reference to an object that handles the encryption for you (see Figure 4). This would

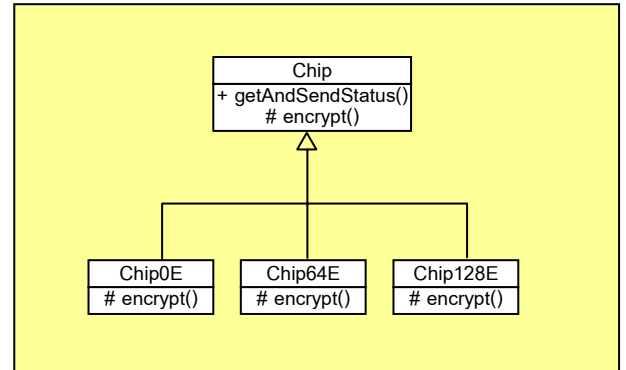


Figure 3. Handling variation with inheritance

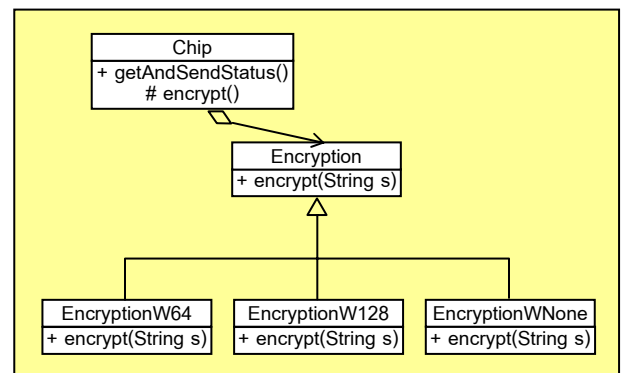


Figure 4. Handling variation using composition (and inheritance)

allow you to use polymorphism (or any other method of handling variation) to handle the different encryption schemes. If you have other functional variations as well, you could create a new object for each variation. If you are also following the "design-to-interface" mandate, you can create as many of these new objects as needed without greatly increasing complexity since your existing reference works for all of them.

The difference between the use of inheritance in

³ See *Object-Oriented Software Construction*, Meyer, B. [6]

About the author -- Alan Shalloway

Alan Shalloway is the CEO and senior consultant of Net Objectives. Since 1981, he has been both an OO consultant and developer of software in several industries. Alan is a frequent speaker at prestigious conferences around the world, including: SD Expo, Java One, OOP, OOPSLA. He is the primary author of *Design Patterns Explained: A New Perspective on Object-Oriented Design* and is currently co-authoring three other books in the software development area. He is a certified Scrum Master and has a Masters in Computer Science from M.I.T.

these two examples is that in Figure 3, I use inheritance on **Chip** directly while in Figure 4, **Chip** contains a reference to **Encryption**, which uses inheritance. From **Chip's** perspective, I achieve the variation in encryption methods through inheritance while in Figure 4, I use *containment* on the **Encryption** object.

Said another way, in the first case, I am using inheritance to *specialize* **Chip**. In the second case, I am using inheritance to *categorize* the various encryption algorithms under a common type.

Consider what should be variable in your design. Focus on encapsulating the concept that varies.

This mandate was a bit confusing to me at first.⁴ To me, encapsulation meant "data-hiding". But to the Gang of Four, it means something else: Encapsulating the concept that varies is **encapsulation of type**.

Notice in Figure 1 how the **Client** object has no coupling to the fact that the different concrete implementations (**AlgorithmOne**, **AlgorithmTwo**, **AlgorithmThree**) even exist. Their existence is hidden (hence encapsulated) by **Algorithm**.⁵ Furthermore, the rules for encryption (i.e., which one should be used under what circumstances) are also encapsulated.

The Forces in the Strategy Pattern

Now, I want to look in more depth at the forces in the Strategy Pattern. They are:

1. Many potential algorithms (business rules) exist.
2. There can be many ways the algorithms vary:
 - Minimal information required to do their tasks
 - Different ways to instantiate them
 - Different situations in which they apply
 - Different persistence implications

⁴ To be honest, I didn't fully understand the one about composition either, but I wasn't confused (just wrong).

⁵ This is one reason we sub-titled our book "A New Perspective of Object-Oriented Design".[7] We didn't mean that we had come up with a new perspective, but that design patterns themselves create a new perspective. We are just explaining it

⁶ Although XP (eXtreme Programming) is in the forefront of this approach, most agile processes endorse some variant of it. If you are not familiar with XP, I highly suggest reading either Kent Beck's *Extreme Programming Explained* [3] or Bob Martin's *Agile Software Development: Principles, Patterns and Practices* [5].

3. Our **Client** application should not be encumbered with having to know all of the business rules possible, as this would both add complexity and limit future variation.

To resolve these forces, the Strategy Pattern says to:

1. Consider the cost of the **Client's** knowing which algorithms it uses. This represents the potential savings of the pattern.
2. Consider the cost of making the algorithms interchangeable. This represents the cost of implementing the pattern.
3. Compare the cost to implement the pattern to the savings it achieves.
4. Decide whether to use the pattern based on this comparison.

This is a bit of over-simplification. Depending upon the software development practices you are following, both deciding on what the costs are and how you determine what to do based on them may vary considerably. This is because what you consider to *be* the savings of the pattern is different for different design approaches.

Different Approaches to Design

There are two popular philosophies today about design. One says developers should do a complete design before starting to code. The other says you should start developing a small part of the system immediately and guide your design with up-front testing and feedback from the customer. As new requirements arise, refactor the design.⁶ The software is developed in small iterations (typically 1-4 weeks) allowing you to adjust to feedback from your customer to ensure you are on the right track.

Evaluating the savings likely will be different for these different approaches. Those taking a "design up front" point of view, consider the savings to be all of the savings that will be

Volume 1, Issue 1
January 2004
©2004, Net Objectives,
All Rights Reserved



Address:
25952 SE 37th Way
Issaquah WA 98029
Telephone:
(404)593-8375
Email:
info@netobjectives.com

achieved for the entire system. In other words, the cost of the **Client** being coupled to the algorithms includes all algorithms that the **Client** eventually needs to know. However, agile developers (particularly Extreme Programmers following the mantra of "doing the simplest thing") consider only those algorithms that are in this iteration (and there may be only *one* algorithm in this iteration).

This "pay now" or "pay later" philosophical difference is often at the heart of why some people say patterns can be harmful. Someone with a design-up-front point of view may say, "hey, I've got only one algorithm now, but I'm bound to get others later, so I'll set up an interface *now*." Such individuals believe that putting in effort now to assist where variations can emerge later will save time down the road.

To join a discussion about this article, please go to <http://www.netobjectivesgroups.com/6/ubb.x> and look under the E-zines category.

Upcoming Free Seminars in the Puget Sound Area

Presenter	Seminar	Date
Dan Rawsthorne	Comparing RUP, XP, and Scrum: Mixing a Process Cocktail for Your Team	Feb 26, 2004
Scott Bain	Emergent Design: Design Patterns and Refactoring for Agile Development	April 1, 2004
Dan Rawsthorne	Introduction to Use Cases	May 6, 2004
Scott Bain	Mock Objects in Endo Testing	June 10, 2004

Go to the URL below to register or for more information

www.netobjectives.com/events/pr_main.htm#FreePresentations

Seminars - Descriptions

Comparing RUP, XP, and Scrum: Mixing a Process Cocktail for Your Team - This seminar discusses how combining the best of some popular processes can provide a successful software development environment for your project.

Emergent Design: Design Patterns and Refactoring for Agile Development - The two approaches of creating quality, high-level, up-front designs with design patterns or relying on emergent design using refactoring as espoused by XP seem opposed to each other. This seminar illustrates why design patterns and refactoring are actually two sides of the same coin.

Introduction to Use Cases- In this seminar we present different facets of the lowly Use Case; what they are, why they're important, how to write one, and how to support agile development with them.

Mock Objects in Endo Testing – This seminar explains the basics of unit testing, how to use unit tests to drive coding forward (test-first), and how to resolve some of the dependencies that make unit testing difficult.

If you are interested in any of these offerings, if your user group or company is interested in Net Objectives making a free technical presentation to them, or if you would like to be notified of upcoming Net Objectives events, please visit our website, or contact us by the email address or phone number below:

www.netobjectives.com • mike.shalloway@netobjectives.com • 404-593-8375

Volume 1, Issue 1
January 2004
©2004, Net Objectives,
All Rights Reserved



Address:
25952 SE 37th Way
Issaquah WA 98029
Telephone:
(404)593-8375
Email:
info@netobjectives.com

The problem with this, however, is that change can happen *anywhere* later. Following this approach everywhere can both overcomplicate your design and waste time because you often implement things that are never needed.⁷

If you force-fit a pattern into a situation where the pattern doesn't apply, you are, by definition, not following the pattern.

The Strategy pattern, however, doesn't suggest this. The pattern describes forces. It is up to you to decide if implementing the pattern is called for.⁸ In other words, if you force-fit a pattern into a situation where the pattern doesn't apply, you are, by definition, not following the pattern. It is therefore not appropriate to say the pattern itself is harmful.

Using the Forces Without the Implementation

On the other hand, the Strategy pattern can be very useful to an XPer even when he or she writes no code that actually implements the pattern. Consider the previous situation if you were following XP coding practices.⁹ If, in your current iteration, you had only one algorithm to implement, you'd likely have your **Client** just call it directly. However, you might also recognize this as a likely place to use a Strategy pattern later if there are other algorithms present in future stories.

XP would say the same thing — its coding practices are geared toward easy code changes. However, the potential need for an implementation of the Strategy pattern underscores this idea and makes you more likely to follow the XP practices you should. Paying attention to this doesn't take any extra work or code; it just reinforces standard XP practices.

Let's consider two XP coding practices: "coding by intention" and "once and only once."

"Coding by intention" says that if, while writing

one method, you find you need to implement a function, pretend a method that accomplishes that function already exists. Give it an intention-revealing name, write down an appropriate parameter list, and actually implement it later.

"Once and only once" means just that — code things once and only once. In other words, avoid duplication. That way, if a change is required in the future, we need only touch our code in one place.

Coding by intention makes it easier to implement the Strategy pattern because the object using the algorithm (i.e., the object that will become the **Context** object if the Strategy pattern is ever implemented) will likely end up having a method called something like "runAlgorithm()." Later, if you need to implement the Strategy pattern, you need only modify this method, causing little or no ripple affect to the rest of the code. The once and only once rule ensures that if there are several steps in calling this algorithm, you'd have only one place containing these steps (otherwise you'd have duplication). Thus, knowing that new algorithms may become available merely reinforces things you should be doing in XP anyway. In other words, knowledge of the Strategy pattern reinforces the good coding rules that XP mandates. It doesn't require that you implement the pattern before it is called for.

This makes patterns a two-way street. My awareness of the forces in the patterns assists me in applying XP coding practices. On the other hand, following XP coding practices makes it easier to implement the patterns later if they are needed (this is often called refactoring to patterns).

Additionally, the whole idea of incremental development, which is at the heart of XP and all agile processes, implies that we should reject the concept that code must decay in favor of the notion that code can evolve. To do this, however, we must hold ourselves to a rigorous standard — each time we touch the code we will make it at least a *little bit better*— and *never any worse*. This will lead us to use the Strategy Pattern when

⁷ One additional aspect of Agile's "pay later" approach is to minimize the cost of paying later with automated testing and risk mitigation.

⁸ On the variant of a well-known theme, I will suggest that hammers aren't bad. Used properly, hammers can be very useful tools. Used improperly, they can be very destructive.

⁹ I consider myself an Agile developer, not an XPer, because my approach includes a broader use of the customer than XP dictates. However, the issues involved in this article relate almost equally well to all Agile / XP developers.

Volume 1, Issue 1
January 2004
©2004, Net Objectives,
All Rights Reserved



Address:
25952 SE 37th Way
Issaquah WA 98029
Telephone:
(404)593-8375
Email:
info@netobjectives.com

Another Example of Patterns as Forces: The Decorator Pattern

The Decorator pattern comes into play when there are a variety of optional functions that can precede or follow another function that is always executed. For example, prior to sending a transmission (which is always done) it may be that you want to encrypt, compress, translate data-check, or any number of other things in any order. How would you best do this? The Decorator pattern's implementation tells you to make a linked list of the optional functions ending with the Transmission object. The "decorating" objects should all have the same interface as the Transmission object.

This implementation can actually be a very bad design. For example, let's say that these "decorators"(i.e., the optional functionality) are created by different development groups. Let's further state that certain exceptions may be thrown by the system and need to be handled by each of the decorators. In a perfect world, you could have confidence that each of these groups will do what they are supposed to do – catch these exceptions properly. However, what if they do not? What if an exception is thrown and the group writing the code doesn't catch it? The entire system may crash at this point.

Another solution is to have the Client object catch the exception. However, this loses some of the value of the Decorator pattern in that your Client object now needs to do more than it did before.

A more robust solution is to implement a collection object that has the same interface as the Transmission object. This calls the "decorators" and catches any required exceptions in case they don't. In fact, this might have the added advantage of making it so the decorating objects no longer needed to have the same interface.

The point is that you can view the Decorator pattern as having the following forces:

1. Several optional functions exist.
2. These decorators may or may not be following all of the rules they should.
3. You need some way to invoke these decorating objects, in different orders as needed, without encumbering the Client object.
4. You don't want to encumber your application with knowing which of these to use (or even that they exist).

Thinking of the Decorator in this light makes a distinction between the *purpose* of the Decorator pattern and the *implementation* of the Decorator pattern.

it improves the code in some way; for instance, by making the **Context** object simpler.

Misapplying the Strategy Pattern

Ignoring the forces of the pattern, however, can lead to its improper use regardless of design philosophy. For example, if the algorithms require input data that is widely different for each algorithm, it may take as much (or more) work to resolve them all to a common interface. Handling varying return data could be just as problematic. This would be akin to forcing a square peg into a

round hole and is counterproductive.

Unfortunately, some people will say "The pattern tells me to do this."¹⁰ No it doesn't. The pattern tells you about forces and a potential implementation. Knowing patterns does not mean you no longer have to think. It just helps you decide what to think *about*.

To put it another way, the Strategy pattern is about externalizing a variation in order to encapsulate it. Doing so protects the **Context** object from the variation, but at a cost. If it's harder to externalize the variation than it is to maintain it in the **Context** object, then the pattern does not apply. Of course, the pattern

¹⁰ Personally, I don't think this is any more responsible than saying "the voices in my head told me to do this."

Volume 1, Issue 1

January 2004

©2004, Net Objectives,
All Rights Reserved



Address:

25952 SE 37th Way
Issaquah WA 98029

Telephone:

(404)593-8375

Email:

info@netobjectives.com

tells you this in the first place.

It is easy to do this if you follow logic such as – “patterns are class diagrams, patterns are good, I should therefore design like the patterns’ class diagrams.” You will not make this mistake if you consider the forces that comprise a pattern.

The naming of patterns, unfortunately, seems to have emphasized that patterns are real and tangible and therefore *are* their class diagrams. The names actually are meant to be a short-hand for the forces and for a *potential* (example) solution.

Summary

We have seen that while a pattern has an implementation, it is better not to think of the pattern as *being* its implementation. If we *do* think of a pattern as mandating solutions in particular situations, this is not unlike having a hammer and looking for nails. Patterns are about helping us resolve forces in our problem domain, about seeing these forces. Using patterns still requires us to think — in fact, maybe more so. However, knowing patterns helps us know what we should be thinking about. Patterns used improperly can cause damage, but then, in essence, we aren’t using the pattern, but rather an incomplete knowledge of patterns. Patterns used properly, can only help us as they help us pay attention to the important issues of the problems we are trying to solve.

Acknowledgements

Special thanks to James R. Trott, Alex Chaffee, and the Net Objectives team for assisting me in writing this article.

References

1. Alexander, Christopher. The Timeless Way of Building. Oxford University Press, 1979. (Both a brilliant technical book as well as a book that is fun to read. I highly recommend reading this if you have any interest in architecture, philosophy, or software.)
2. Alexander, Christopher, Sara Ishikawa, and Murray Silverstein, with Max Jacobson, Ingrid Fiksdahl- King, and Shlomo Angel. A Pattern Language. Oxford University Press, 1977. (A companion text to The Timeless Way of Building.)
3. Beck, Kent. Extreme Programming Explained: Embrace Change. Addison-Wesley, 1999. (Useful whether or not you think you will ever follow its principles and practices. A groundbreaking work in our industry.)
4. Gamma, Erich, Richard Helm, Ralph Johnson, and John Vlissides. Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, 1995. (Still the best reference on design patterns, even if it’s starting to become somewhat dated.)

Employee Spotlight – Doug Shimp

Net Objectives is pleased to announce the addition of Doug Shimp as a senior consultant. Doug will present courses and seminars in the Chicago area.

Doug has 15 years experience in the technology field where he has played all roles in software development. Doug’s unique distinction is his focus on the interaction of technology and corporate cultural issues.



Doug is certified by Cockburn and Associates to teach “Writing Effective Use Cases” and by Advanced Development Methods as a Scrum Master.

Volume 1, Issue 1
January 2004
©2004, Net Objectives,
All Rights Reserved



Address:
25952 SE 37th Way
Issaquah WA 98029
Telephone:
(404)593-8375
Email:
info@netobjectives.com

5. Martin, Robert C. Agile Software Development: Principles, Patterns, and Practices. Prentice Hall, 2002. (Destined to be a classic if not one already. Brilliant.)

6. Meyer, Bertrand. Object-Oriented Software Construction, 2nd edition. Prentice Hall, 2000. (Although the size of this book is a bit intimidating, it is a brilliant book. Read it a little at a time and reflect on what he has to say.)

7. Shalloway, Alan, and James R. Trott. Design Patterns Explained: A New Perspective on Object-Oriented Design. Addison-Wesley, 2001. (At the risk of sounding self-serving, this is the best text on introducing a developer to what patterns are. Read before reading the Design patterns book by the Gang of Four.)

Other Seminars We Can Give

Pattern Oriented Development: Design Patterns From Analysis To Implementation - This seminar discusses how design patterns can be used to improve the entire software development process - not just the design aspect of it.

Agility and Ceremony: How Can They Co-Exist? - This seminar provides a short overview of Agility and the challenges one faces in a High Ceremony environment.

Transitioning to Agile - More and more companies are beginning to see the need for Agile Development. In this seminar, we discuss what problems agility will present and how to deal with these.

The Need for Agility - This seminar is designed to demonstrate how to balance the necessity for up-front analysis and design with the need to get feedback about how the project is going.

Use Cases and the Ever-Unfolding Story - This seminar introduces use case based requirements analysis, including the use of the Ever Unfolding Story for refining use cases into more detailed requirements.

Effective Coding Practices - This seminar introduces three straightforward coding practices that support design patterns, refactoring and test-driven-development: Encapsulating Construction, Coding by Intention, and Considering Testability before Coding.

Unit Testing For Emergent Design - This seminar illustrates why design patterns and refactoring are actually two sides of the same coin.

C# for Java and C++ Developers - This seminar will give a complete overview of the major language features of C#, and will also examine some best-practice issues.

Effective C# - This seminar is intended for programmers who are not object-oriented experts and who want to learn how to use C# effectively.

and 3 on key XML topics:

XML Data Binding With Castor

XSLT - Step By Step

The JDOM Alternative For XML Parsing

If your user group or company is interested in Net Objectives making a free technical presentation for them, or if you would like to be notified of upcoming Net Objectives events, please visit our website, or contact us by the email address or phone number below.

Volume 1, Issue 1

January 2004

©2004, Net Objectives,
All Rights Reserved



Address:
25952 SE 37th Way
Issaquah WA 98029
Telephone:
(404)593-8375
Email:
info@netobjectives.com

For Additional Information:

www.netobjectives.com • mike.shalloway@netobjectives.com • 404-593-8375