

table of c.

I changed the order of pattern to be studied by their file name.
Among them, first 10 patterns might use even in j2ee application development, the remaining 5 patterns just get to know. (13 - Jul - 2009 done 1st round)

Table of Contents (summary)

05 - Jan - 2008

13 - Jul - 2009

finished 1st round this whole
book on 14 - July - 09 3:49 AM

	Intro	xxv
1	Welcome to Design Patterns: <i>an introduction</i>	1
2	Keeping your Objects in the know: <i>the Observer Pattern</i>	37
3	Decorating Objects: <i>the Decorator Pattern</i>	79
4	Baking with OO goodness: <i>the Factory Pattern</i>	109
5	One of a Kind Objects: <i>the Singleton Pattern</i>	169
6	Encapsulating Invocation: <i>the Command Pattern</i>	191
7	Being Adaptive: <i>the Adapter and Facade Patterns</i>	235
8	Encapsulating Algorithms: <i>the Template Method Pattern</i>	275
9	Well-managed Collections: <i>the Iterator and Composite Patterns</i>	315
10	The State of Things: <i>the State Pattern</i>	385
11	Controlling Object Access: <i>the Proxy Pattern</i>	429
12	Patterns of Patterns: <i>Compound Patterns</i>	499
13	Patterns in the Real World: <i>Better Living with Patterns</i>	577
14	Appendix: <i>Leftover Patterns</i>	611

Table of Contents (the real thing)

Intro

Your brain on Design Patterns. Here *you* are trying to *learn* something, while here your *brain* is doing you a favor by making sure the learning doesn't *stick*. Your brain's thinking, "Better leave room for more important things, like which wild animals to avoid and whether naked snowboarding is a bad idea." So how *do* you trick your brain into thinking that your life depends on knowing Design Patterns?

Who is this book for?	xxvi
We know what your brain is thinking	xxvii
Metacognition	xxix
Bend your brain into submission	xxxi
	xxxiv
	xxxv

design patterns are used only in source code level.

but not at runtime how the objects are linked or used. since without objects interaction one with other nothing big work can be done.

so, the design pattern, helps at source level for , easy maintainence, extension, understanding and etc.

Design 13 Better Living with Patterns

Patterns in the Real World



Ahhhh, now you're ready for a bright new world filled with **Design Patterns**. But, before you go opening all those new doors of opportunity, we need to cover a few details that you'll encounter out in the real world – that's right, things get a little more complex than they are here in Objectville. Come along, we've got a nice guide to help you through the transition on the next page...

The Objectville Guide to Better Living with Design Patterns



Please accept our handy guide with tips & tricks for living with patterns in the real world. In this guide you will:

- ☞ Learn the all too common misconceptions about the definition of a "Design Pattern."
- ☞ Discover those nifty Design Pattern Catalogs and why you just have to get one.
- ☞ Avoid the embarrassment of using a Design Pattern at the wrong time.
- ☞ Learn how to keep patterns in classifications where they belong.
- ☞ See that discovering patterns isn't just for the gurus; read our quick HowTo and become a patterns writer too.
- ☞ Be there when the true identity of the mysterious Gang of Four is revealed.
- ☞ Keep up with the neighbors - the coffee table books any patterns user must own.
- ☞ Learn to train your mind like a Zen master.
- ☞ Win friends and influence developers by improving your patterns vocabulary.

Design Pattern defined

We bet you've got a pretty good idea of what a pattern is after reading this book. But we've never really given a definition for a Design Pattern. Well, you might be a bit surprised by the definition that is in common use:

A Pattern is a solution to a problem in a context.

That's not the most revealing definition is it? But don't worry, we're going to step through each of these parts, context, problem and solution:

The **context** is the situation in which the pattern applies. This should be a recurring situation.

The **problem** refers to the goal you are trying to achieve in this context, but it also refers to any constraints that occur in the context.

The **solution** is what you're after: a general design that anyone can apply which resolves the goal and set of constraints.

Example: You have a collection of objects.

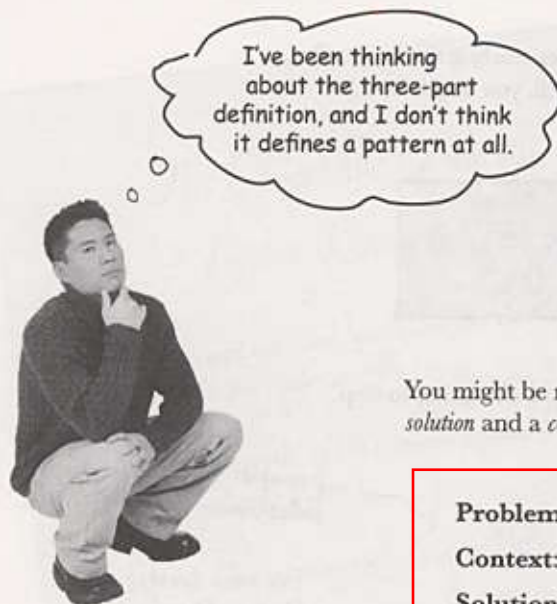
You need to step through the objects without exposing the collection's implementation.

Encapsulate the iteration into a separate class.

This is one of those definitions that takes a while to sink in, but take it one step at a time. Here's a little mnemonic you can repeat to yourself to remember it:

"If you find yourself in a context with a problem that has a goal that is affected by a set of constraints, then you can apply a design that resolves the goal and constraints and leads to a solution."

Now, this seems like a lot of work just to figure out what a Design Pattern is. After all, you already know that a Design Pattern gives you a solution to a common recurring design problem. What is all this formality getting you? Well, you're going to see that by having a formal way of describing patterns we can create a *catalog* of patterns, which has all kinds of benefits.



You might be right; let's think about this a bit... We need a *problem*, a *solution* and a *context*:

Problem: How do I get to work on time?

Context: I've locked my keys in the car.

Solution: Break the window, get in the car, start the engine and drive to work.

We have all the components of the definition: we have a problem, which includes the goal of getting to work, and the constraints of time, distance and probably some other factors. We also have a context in which the keys to the car are inaccessible. And we have a solution that gets us to the keys and resolves both the time and distance constraints. We must have a pattern now! Right?

BRAIN POWER

We followed the Design Pattern definition and defined a problem, a context, and a solution (which works!). Is this a pattern? If not, how did it fail? Could we fail the same way when defining an OO Design Pattern?

Looking more closely at the Design Pattern definition

Our example does seem to match the Design Pattern definition, but it isn't a true pattern. Why? For starters, we know that a pattern needs to apply to a recurring problem. While an absent-minded person might lock his keys in the car often, breaking the car window doesn't qualify as a solution that can be applied over and over (or at least isn't likely to if we balance the goal with another constraint: cost).

It also fails in a couple of other ways: first, it isn't easy to take this description, hand it to someone and have him apply it to his own unique problem. Second, we've violated an important but simple aspect of a pattern: we haven't even given it a name! Without a name, the pattern doesn't become part of a vocabulary that can be shared with other developers.

Luckily, patterns are not described and documented as a simple problem, context and solution; we have much better ways of describing patterns and collecting them together into *patterns catalogs*.

Q: Am I going to see pattern descriptions that are stated as a problem, a context and a solution?

A: Pattern descriptions, which you'll typically find in pattern catalogs, are usually a bit more revealing than that. We're going to look at pattern catalogs in detail in just a minute; they describe a lot more about a pattern's intent and motivation and where it might apply, along with the solution design and the consequences (good and bad) of using it.

Q: Is it okay to slightly alter a pattern's structure to fit my design? Or am I going to have to go by the strict definition?

A: Of course you can alter it. Like design principles, patterns are not meant to be laws or rules; they are *guidelines* that you can alter to fit your needs. As you've seen, a lot of real-world examples don't fit the classic pattern designs.

However, when you adapt patterns, it never hurts to document how your pattern differs from the classic design – that way, other developers can quickly recognize the patterns you're using and any differences between your pattern and the classic pattern.

Next time someone tells you a pattern is a solution to a problem in a context, just nod and smile. You know what they mean, even if it isn't a definition sufficient to describe what a Design Pattern really is.



Q: Where can I get a patterns catalog?

A: The first and most definitive patterns catalog is *Design Patterns: Elements of Reusable Object-Oriented Software*, by Gamma, Helm, Johnson & Vlissides (Addison Wesley). This catalog lays out 23 fundamental patterns. We'll talk a little more about this book in a few pages.

Many other patterns catalogs are starting to be published in various domain areas such as enterprise software, concurrent systems and business systems.



Geek Bits

May the force be with you

The Design Pattern definition tells us that the *problem* consists of a *goal* and a *set of constraints*.

Patterns gurus have a term for these: they call them forces. Why? Well, we're sure they have their own reasons, but if you remember the movie, the force "shapes and controls the Universe."

Likewise, the forces in the pattern definition shape and control the solution.

Only when a solution balances both sides of the force (the light side: your goal, and the dark side: the constraints) do we have a useful pattern.

This "force" terminology can be quite confusing when you first see it in pattern discussions, but just remember that there are two sides of the force (goals and constraints) and that they need to be balanced or resolved to create a pattern solution. Don't let the lingo get in your way and may the force be with you!



Frank: Fill us in, Jim. I've just been learning patterns by reading a few articles here and there.

Jim: Sure, each pattern catalog takes a set of patterns and describes each in detail along with its relationship to the other patterns.

Joe: Are you saying there is more than one patterns catalog?

Jim: Of course; there are catalogs for fundamental Design Patterns and there are also catalogs on domain specific patterns, like EJB patterns.

Frank: Which catalog are you looking at?

Jim: This is the classic GoF catalog; it contains 23 fundamental Design Patterns.

Frank: GoF?

Jim: Right, that stands for the Gang of Four. The Gang of Four are the guys that put together the first patterns catalog.

Joe: What's in the catalog?

Jim: There is a set of related patterns. For each pattern there is a description that follows a template and spells out a lot of details of the pattern. For instance, each pattern has a *name*.

Frank: Wow, that's earth-shattering – a name! Imagine that.

Jim: Hold on Frank; actually, the name is really important. When we have a name for a pattern, it gives us a way to talk about the pattern; you know, that whole shared vocabulary thing.

Frank: Okay, okay. I was just kidding. Go on, what else is there?

Jim: Well, like I was saying, every pattern follows a template. For each pattern we have a name and a few sections that tell us more about the pattern. For instance, there is an Intent section that describes what the pattern is, kind of like a definition. Then there are Motivation and Applicability sections that describe when and where the pattern might be used.

Joe: What about the design itself?

Jim: There are several sections that describe the class design along with all the classes that make it up and what their roles are. There is also a section that describes how to implement the pattern and often sample code to show you how.

Frank: It sounds like they've thought of everything.

Jim: There's more. There are also examples of where the pattern has been used in real systems as well as what I think is one of the most useful sections: how the pattern relates to other patterns.

Frank: Oh, you mean they tell you things like how *state* and *strategy* differ?

Jim: Exactly!

Joe: So Jim, how are you actually using the catalog? When you have a problem, do you go fishing in the catalog for a solution?

Jim: I try to get familiar with all the patterns and their relationships first. Then, when I need a pattern, I have some idea of what it is. I go back and look at the Motivation and Applicability sections to make sure I've got it right. There is also another really important section: Consequences. I review that to make sure there won't be some unintended effect on my design.

Frank: That makes sense. So once you know the pattern is right, how do you approach working it into your design and implementing it?

Jim: That's where the class diagram comes in. I first read over the Structure section to review the diagram and then over the Participants section to make sure I understand each classes' role. From there I work it into my design, making any alterations I need to make it fit. Then I review the Implementation and Sample code sections to make sure I know about any good implementation techniques or gotchas I might encounter.

Joe: I can see how a catalog is really going to accelerate my use of patterns!

Frank: Totally. Jim, can you walk us through a pattern description?

All patterns in a catalog start with a name. The name is a vital part of a pattern – without a good name, a pattern can't become part of the vocabulary that you share with other developers.

The motivation gives you a concrete scenario that describes the problem and how the solution solves the problem.

The applicability describes situations in which the pattern can be applied.

The participants are the classes and objects in the design. This section describes their responsibilities and roles in the pattern.

The consequences describe the effects that using this pattern may have: good and bad.

Implementation provides techniques you need to use when implementing this pattern, and issues you should watch out for.

Known uses describes examples of this pattern found in real systems.

SINGLETON Object Creation

Intent
 To allow a class to have one instance only, and to provide a global point of access to that instance.

Motivation
 In design, when you have a class that represents a single object, and you want to ensure that there is only one instance of that class, you can use the Singleton pattern. This pattern ensures that there is only one instance of the class, and it provides a global point of access to that instance.

Applicability
 Use the Singleton pattern when you need to ensure that there is only one instance of a class, and you want to provide a global point of access to that instance.

Structure

Participant
Singleton
Singleton
Singleton

Participants
 The Singleton pattern has one participant: the Singleton class. This class is responsible for creating and managing the single instance of the class.

Collaborations
 The Singleton pattern has no collaborations.

Consequences
 The Singleton pattern has several consequences:

1. It ensures that there is only one instance of the class.
2. It provides a global point of access to that instance.
3. It ensures that the instance is created only once.
4. It ensures that the instance is shared by all clients.
5. It ensures that the instance is destroyed only once.
6. It ensures that the instance is not created if it already exists.

Implementation/Sample Code
 The Singleton pattern can be implemented in several ways. Here is a sample implementation in Java:

```

public class Singleton {
    private static Singleton instance;

    private Singleton() {}

    public static Singleton getInstance() {
        if (instance == null) {
            instance = new Singleton();
        }
        return instance;
    }
}
    
```

Known Uses
 The Singleton pattern is used in many real-world systems. For example, it is used to implement the global object in a database system.

Related Patterns
 The Singleton pattern is related to the Factory Method pattern and the Abstract Factory pattern.

This is the pattern's classification or category. We'll talk about these in a few pages.

The intent describes what the pattern does in a short statement. You can also think of this as the pattern's definition (just like we've been using in this book).

The structure provides a diagram illustrating the relationships among the classes that participate in the pattern.

Collaborations tells us how the participants work together in the pattern.

Sample code provides code fragments that might help with your implementation.

Related patterns describes the relationship between this pattern and others.

there are no Dumb Questions

Q: Is it possible to create your own Design Patterns? Or is that something you have to be a "patterns guru" to do?

A: First, remember that patterns are *discovered*, not created. So, anyone can discover a Design Pattern and then author its description; however, it's not easy and doesn't happen quickly, nor often. Being a "patterns writer" takes commitment.

You should first think about why you'd want to – the majority of people don't *author* patterns; they just *use* them. However, you might work in a specialized domain for which you think new patterns would be helpful, or you might have come across a solution to what you think is a recurring problem, or you may just want to get involved in the patterns community and contribute to the growing body of work.

Q: I'm game; how do I get started?

A: Like any discipline, the more you know the better. Studying existing patterns, what they do and how they relate to other patterns is crucial. Not only does it make you familiar with how patterns are crafted, it prevents you from reinventing the wheel. From there you'll want to start writing your patterns on paper, so you can communicate them to other developers; we're going to talk more about how to communicate your patterns in a bit. If you're really interested, you'll want to read the section that follows these Q&As.

Q: How do I know when I really have a pattern?

A: That's a very good question: you don't have a pattern until others have used it and found it to work. In general, you don't have a pattern until it passes the "Rule of Three." This rule states that a pattern can be called a pattern only if it has been applied in a real-world solution at least three times.

So you wanna be a design
patterns star?

Well listen now to what I tell.

Get yourself a patterns
catalog,

Then take some time and
learn it well.

And when you've got your
description right,

And three developers agree
without a fight,

Then you'll know it's a
pattern alright.

↑
To the tune of "So you wanna
be a Rock'n Roll Star."

So you wanna be a Design Patterns writer

Do your homework. You need to be well versed in the existing patterns before you can create a new one. Most patterns that appear to be new, are, in fact, just variants of existing patterns. By studying patterns, you become better at recognizing them, and you learn to relate them to other patterns.

Take time to reflect, evaluate. Your experience – the problems you've encountered, and the solutions you've used – are where ideas for patterns are born. So take some time to reflect on your experiences and comb them for novel designs that recur. Remember that most designs are variations on existing patterns and not new patterns. And when you do find what looks like a new pattern, its applicability may be too narrow to qualify as a real pattern.

Get your ideas down on paper in a way others can understand. Locating new patterns isn't of much use if others can't make use of your find; you need to document your pattern candidates so that others can read, understand, and apply them to their own solution and then supply you with feedback. Luckily, you don't need to invent your own method of documenting your patterns. As you've already seen with the GoF template, a lot of thought has already gone into how to describe patterns and their characteristics.

Have others try your patterns; then refine and refine some more. Don't expect to get your pattern right the first time. Think of your pattern as a work in progress that will improve over time. Have other developers review your candidate pattern, try it out, and give you feedback. Incorporate that feedback into your description and try again. Your description will never be perfect, but at some point it should be solid enough that other developers can read and understand it.

Don't forget the rule of three. Remember, unless your pattern has been successfully applied in three real-world solutions, it can't qualify as a pattern. That's another good reason to get your pattern into the hands of others so they can try it, give feedback, and allow you to converge on a working pattern.

Use one of the existing pattern templates to define your pattern. A lot of thought has gone into these templates and other pattern users will recognize the format.



★ WHO DOES WHAT? ★

Match each pattern with its description:

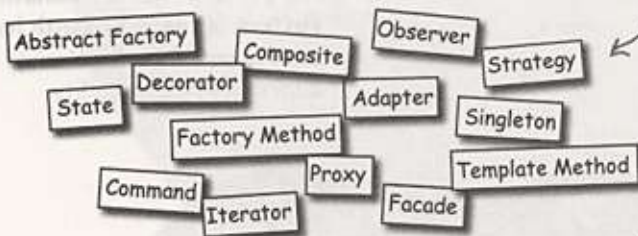
Pattern	Description
Decorator	Wraps an object and provides a different interface to it.
State	Subclasses decide how to implement steps in an algorithm.
Iterator	Subclasses decide <u>which concrete classes to create</u> .
Facade	Ensures one and only object is created.
Strategy	Encapsulates interchangeable behaviors and uses delegation to decide which one to use.
Proxy	Clients treat collections of objects and individual objects uniformly.
Factory Method	Encapsulates state-based behaviors and uses delegation to switch between behaviors.
Adapter	Provides a way to traverse a collection of objects without exposing its implementation.
Observer	Simplifies the interface of a set of classes.
Template Method	Wraps an object to provide new behavior.
Composite	Allows a client to create families of objects without specifying their concrete classes.
Singleton	Allows objects to be notified when state changes.
Abstract Factory	Wraps an object to control access to it.
Command	Encapsulates a request as an object.

Organizing Design Patterns

As the number of discovered Design Patterns grows, it makes sense to partition them into classifications so that we can organize them, narrow our searches to a subset of all Design Patterns, and make comparisons within a group of patterns.

In most catalogs you'll find patterns grouped into one of a few classification schemes. The most well-known scheme was used by the first pattern catalog and partitions patterns into three distinct categories based on their purposes: Creational, Behavioral and Structural.

Sharpen your pencil

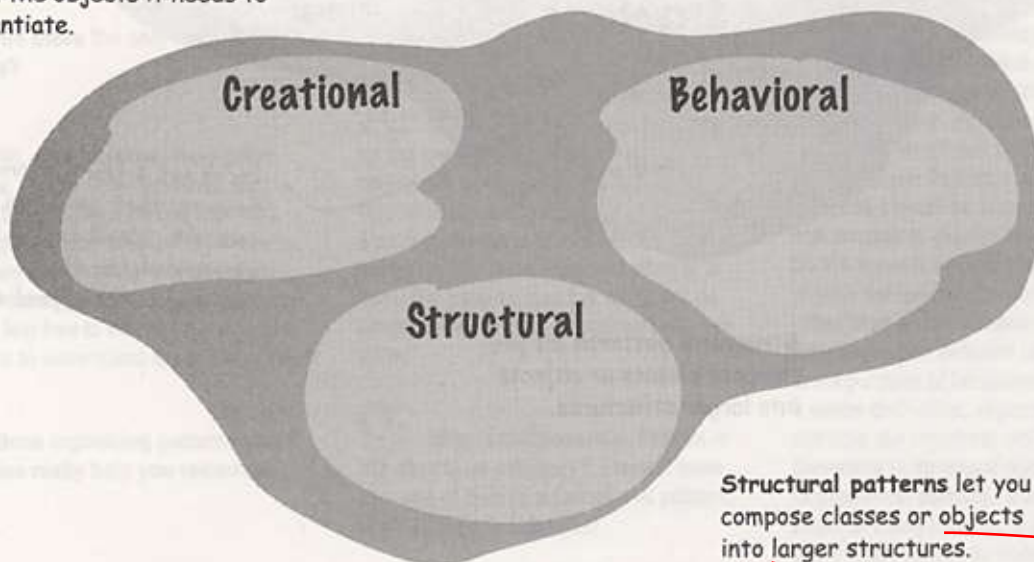


Read each category description and see if you can corral these patterns into their correct categories. This is a toughy! But give it your best shot and then check out the answers on the next page.

Each of these patterns belongs in one of those categories

Creational patterns involve object instantiation and all provide a way to decouple a client from the objects it needs to instantiate.

Any pattern that is a **Behavioral Pattern** is concerned with how classes and objects interact and distribute responsibility.



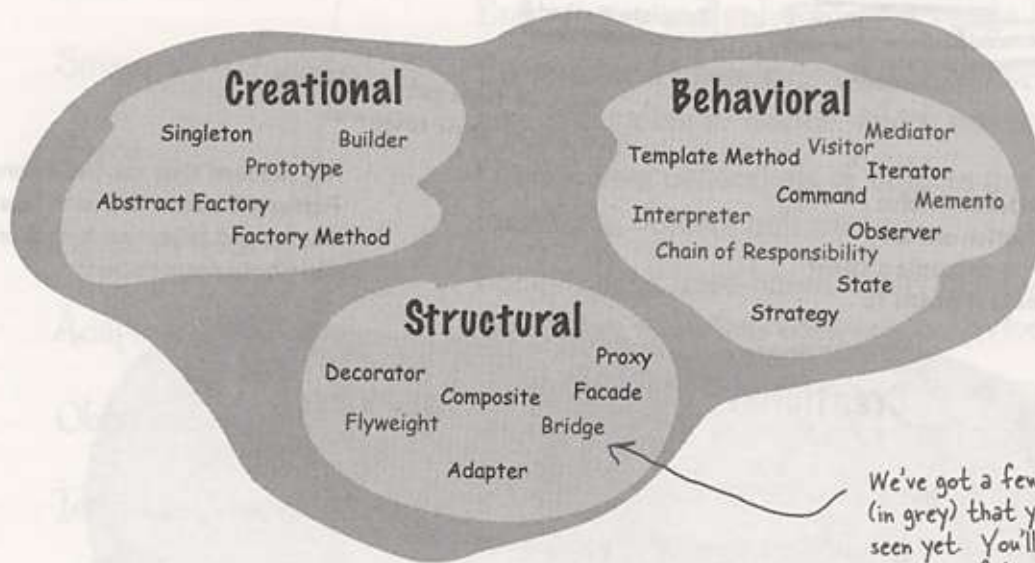
Structural patterns let you compose classes or objects into larger structures.

Solution: Pattern Categories

Here's the grouping of patterns into categories. You probably found the exercise difficult, because many of the patterns seem like they could fit into more than one category. Don't worry, everyone has trouble figuring out the right categories for the patterns.

Creational patterns involve object instantiation and all provide a way to decouple a client from the objects it needs to instantiate.

Any pattern that is a **Behavioral Pattern** is concerned with how classes and objects interact and distribute responsibility.



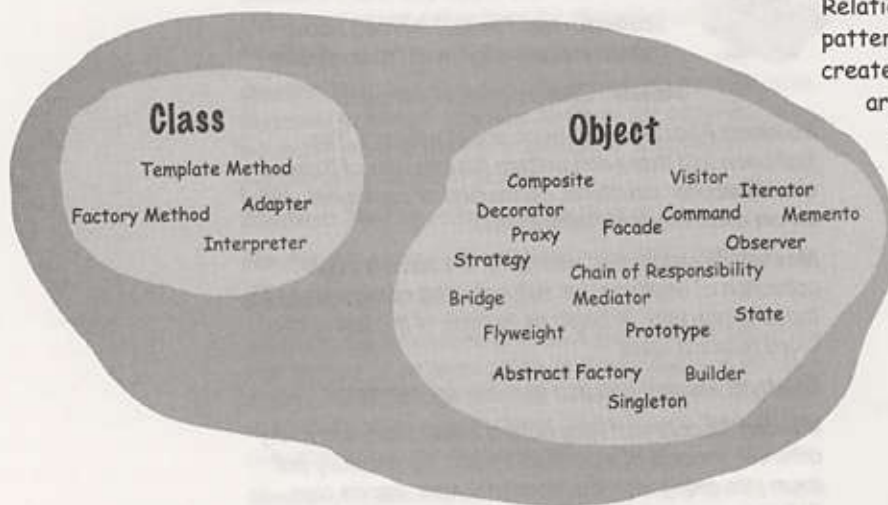
We've got a few patterns (in grey) that you haven't seen yet. You'll find an overview of these patterns in the appendix.

Structural patterns let you compose classes or objects into larger structures.

Patterns are often classified by a second attribute: whether or not the pattern deals with classes or objects:

Class patterns describe how relationships between classes are defined via inheritance. Relationships in class patterns are established at compile time.

Object patterns describe relationships between objects and are primarily defined by composition. Relationships in object patterns are typically created at runtime and are more dynamic and flexible.



Notice there's a lot more object patterns than class patterns!

Q: Are these the only classification schemes?

A: No, other schemes have been proposed. Some other schemes start with the three categories and then add subcategories, like "Decoupling Patterns." You'll want to be familiar with the most common schemes for organizing patterns, but also feel free to create your own, if it helps you to understand the patterns better.

Q: Does organizing patterns into categories really help you remember them?

there are no Dumb Questions

A: It certainly gives you a framework for the sake of comparison. But many people are confused by the creational, structural and behavioral categories; often a pattern seems to fit into more than one category. The most important thing is to know the patterns and the relationships among them. When categories help, use them!

Q: Why is the Decorator Pattern in the structural category? I would have thought of that as a behavioral pattern; after all it adds behavior!

A: Yes, lots of developers say that! Here's the thinking behind the Gang of Four classification: structural patterns describe how classes and objects are composed to create new structures or new functionality. The Decorator Pattern allows you to compose objects by wrapping one object with another to provide new functionality. So the focus is on how you compose the objects dynamically to gain functionality, rather than on the communication and interconnection between objects, which is the purpose of behavioral patterns. It's a subtle distinction, especially when you consider the structural similarities between Decorator (a structural pattern) and Proxy, (a behavioral pattern). But remember, the intent of these patterns is different, and that's often the key to understanding which category a pattern belongs to.

Solution: Pattern Categories



Master and Student...

Master: Grasshopper, you look troubled.

Student: Yes, I've just learned about pattern classification and I'm confused.

Master: Grasshopper, continue...

Student: After learning much about patterns, I've just been told that each pattern fits into one of three classifications: structural, behavioral or creational. Why do we need these classifications?

Master: Grasshopper, whenever we have a large collection of anything, we naturally find categories to fit those things into. It helps us to think of the items at a more abstract level.

Student: Master, can you give me an example?

Master: Of course. Take automobiles; there are many different models of automobiles and we naturally put them into categories like economy cars, sports cars, SUVs, trucks and luxury car categories.

Master: Grasshopper, you look shocked, does this not make sense?

Student: Master, it makes a lot of sense, but I am shocked you know so much about cars!

Master: Grasshopper, I can't relate **everything** to lotus flowers or rice bowls. Now, may I continue?

Student: Yes, yes, I'm sorry, please continue.

Master: Once you have classifications or categories you can easily talk about the different groupings: "If you're doing the mountain drive from Silicon Valley to Santa Cruz, a sports car with good handling is the best option." Or, "With the worsening oil situation you really want to buy a economy car, they're more fuel-efficient."

Student: So by having categories we can talk about a set of patterns as a group. We might know we need a creational pattern, without knowing exactly which one, but we can still talk about creational patterns.

Master: Yes, and it also gives us a way to compare a member to the rest of the category, for example, "the Mini really is the most stylish compact car around", or to narrow our search, "I need a fuel efficient car."

Student: I see, so I might say that the Adapter pattern is the best structural pattern for changing an object's interface.

Master: Yes. We also can use categories for one more purpose: to launch into new territory; for instance, "we really want to deliver a sports car with ferrari performance at miata prices."

Student: That sounds like a death trap.

Master: I'm sorry, I did not hear you Grasshopper.

Student: Uh, I said "I see that."

Student: So categories give us a way to think about the way groups of patterns relate and how patterns within a group relate to one another. They also give us a way to extrapolate to new patterns. But why are there three categories and not four, or five?

Master: Ah, like stars in the night sky, there are as many categories as you want to see. Three is a convenient number and a number that many people have decided makes for a nice grouping of patterns. But others have suggested four, five or more.



Thinking in Patterns

Contexts, constraints, forces, catalogs, classifications... boy, this is starting to sound mighty academic. Okay, all that stuff is important and knowledge is power. But, let's face it, if you understand the academic stuff and don't have the *experience* and practice using patterns, then it's not going to make much difference in your life.

Here's a quick guide to help you start to *think in patterns*. What do we mean by that? We mean being able to look at a design and see where patterns naturally fit and where they don't.



Your Brain on Patterns

Keep it simple (KISS)

First of all, when you design, solve things in the simplest way possible. Your goal should be simplicity, not "how can I apply a pattern to this problem." Don't feel like you aren't a sophisticated developer if you don't use a pattern to solve a problem. Other developers will appreciate and admire the simplicity of your design. That said, sometimes the best way to keep your design simple and flexible is to use a pattern.

Design Patterns aren't a magic bullet; in fact they're not even a bullet!

Patterns, as you know, are general solutions to recurring problems. Patterns also have the benefit of being well tested by lots of developers. So, when you see a need for one, you can sleep well knowing many developers have been there before and solved the problem using similar techniques.

However, patterns aren't a magic bullet. You can't plug one in, compile and then take an early lunch. To use patterns, you also need to think through the consequences on the rest of your design.

You know you need a pattern when...

Ah... the most important question: when do you use a pattern? As you approach your design, introduce a pattern when you're sure it addresses a problem in your design. If a simpler solution might work, give that consideration before you commit to using a pattern.

Knowing when a pattern applies is where your experience and knowledge come in. Once you're sure a simple solution will not meet your needs, you should consider the problem along with the set of constraints under which the solution will need to operate — these will help you match your problem to a pattern. If you've got a good knowledge of patterns, you may know of a pattern that is a good match. Otherwise, survey patterns that look like they might solve the problem. The intent and applicability sections of the patterns catalogs are particularly useful for this. Once you've found a pattern that appears to be a good match, make sure it has a set of consequences you can live with and study its effect on the rest of your design. If everything looks good, go for it!

There is one situation in which you'll want to use a pattern even if a simpler solution would work: when you expect aspects of your system to vary. As we've seen, identifying areas of change in your design is usually a good sign that a pattern is needed. Just make sure you are adding patterns to deal with *practical change* that is likely to happen, not *hypothetical change* that may happen.

Design time isn't the only time you want to consider introducing patterns, you'll also want to do so at refactoring time.

Refactoring time is Patterns time!

Refactoring is the process of making changes to your code to improve the way it is organized. The goal is to improve its structure, not change its behavior. This is a great time to reexamine your design to see if it might be better structured with patterns. For instance, code that is full of conditional statements might signal the need for the State pattern. Or, it may be time to clean up concrete dependencies with a Factory. Entire books have been written on the topic of refactoring with patterns, and as your skills grow, you'll want to study this area more.

so, refactoring is not only just rename of method of classes.

Take out what you don't really need. Don't be afraid to remove a Design Pattern from your design.

No one ever talks about when to remove a pattern. You'd think it was blasphemy! Nah, we're all adults here; we can take it.

So when do you remove a pattern? When your system has become complex and the flexibility you planned for isn't needed. In other words, when a simpler solution without the pattern would be better.

If you don't need it now, don't do it now.

Design Patterns are powerful, and it's easy to see all kinds of ways they can be used in your current designs. Developers naturally love to create beautiful architectures that are ready to take on change from any direction.

Resist the temptation. If you have a practical need to support change in a design today, go ahead and employ a pattern to handle that change. However, if the reason is only hypothetical, don't add the pattern, it is only going to add complexity to your system, and you might never need it!

Center your thinking on design, not on patterns. Use patterns when there is a natural need for them. If something simpler will work, then use it.





Master and Student...

Master: Grasshopper, your initial training is almost complete. What are your plans?

Student: I'm going to Disneyland! And, then I'm going to start creating lots of code with patterns!

Master: Whoa, hold on. Never use your big guns unless you have to.

Student: What do you mean, Master? Now that I've learned design patterns shouldn't I be using them in all my designs to achieve maximum power, flexibility and manageability?

Master: No; patterns are a tool, and a tool that should only be used when needed. You've also spent a lot of time learning design principles. Always start from your principles and create the simplest code you can that does the job. However, if you see the need for a pattern emerge, then use it.

Student: So I shouldn't build my designs from patterns?

Master: That should not be your goal when beginning a design. Let patterns emerge naturally as your design progresses.

Student: If patterns are so great, why should I be so careful about using them?

Master: Patterns can introduce complexity, and we never want complexity where it is not needed. But patterns are powerful when used where they are needed. As you already know, patterns are proven design experience that can be used to avoid common mistakes. They're also a shared vocabulary for communicating our design to others.

Student: Well, when do we know it's okay to introduce design patterns?

Master: Introduce a pattern when you are sure it's necessary to solve a problem in your design, or when you are quite sure that it is needed to deal with a future change in the requirements of your application.

Student: I guess my learning is going to continue even though I already understand a lot of patterns.

Master: Yes, grasshopper; learning to manage the complexity and change in software is a life long pursuit. But now that you know a good set of patterns, the time has come to apply them where needed in your design and to continue learning more patterns.

Student: Wait a minute, you mean I don't know them ALL?

Master: Grasshopper, you've learned the fundamental patterns; you're going to find there are many more, including patterns that just apply to particular domains such as concurrent systems and enterprise systems. But now that you know the basics, you're in good shape to learn them!

Your Mind on Patterns



BEGINNER MIND

"I need a pattern for Hello World."

The Beginner uses patterns everywhere. This is good: the beginner gets lots of experience with and practice using patterns. The beginner also thinks, "The more patterns I use, the better the design." The beginner will learn this is not so, that all designs should be as simple as possible. Complexity and patterns should only be used where they are needed for practical extensibility.

As learning progresses, the Intermediate mind starts to see where patterns are needed and where they aren't. The intermediate mind still tries to fit too many square patterns into round holes, but also begins to see that patterns can be adapted to fit situations where the canonical pattern doesn't fit.



INTERMEDIATE MIND

"Maybe I need a Singleton here."



ZEN MIND

"This is a natural place for Decorator."

The Zen mind is able to see patterns where they fit naturally. The Zen mind is not obsessed with using patterns; rather it looks for simple solutions that best solve the problem. The Zen mind thinks in terms of the object principles and their trade-offs. When a need for a pattern naturally arises, the Zen mind applies it knowing well that it may require adaptation. The Zen mind also sees relationships to similar patterns and understands the subtleties of differences in the intent of related patterns. The Zen mind is also a Beginner mind — it doesn't let all that pattern knowledge overly influence design decisions.

WARNING: Overuse of design patterns can lead to code that is downright over-engineered. Always go with the simplest solution that does the job and introduce patterns where the need emerges.

Wait a minute; I've read this entire book and now you're telling me NOT to use patterns?



Of course we want you to use Design Patterns!

But we want you to be a good OO designer even more.

When a design solution calls for a pattern, you get the benefits of using a solution that has been time tested by lots of developers. You're also using a solution that is well documented and that other developers are going to recognize (you know, that whole shared vocabulary thing).

However, when you use Design Patterns, there can also be a downside. Design Patterns often introduce additional classes and objects, and so they can increase the complexity of your designs. Design Patterns can also add more layers to your design, which adds not only complexity, but also inefficiency.

Also, using a Design Pattern can sometimes be outright overkill. Many times you can fall back on your design principles and find a much simpler solution to solve the same problem. If that happens, don't fight it. Use the simpler solution.


Don't let us discourage you, though. When a Design Pattern is the right tool for the job, the advantages are many.

Don't forget the power of the shared vocabulary

We've spent so much time in this book discussing OO nuts and bolts that it's easy to forget the human side of Design Patterns – they don't just help load your brain with solutions, they also give you a shared vocabulary with other developers. Don't underestimate the power of a shared vocabulary, it's one of the *biggest benefits* of Design Patterns.

Just think, something has changed since the last time we talked about shared vocabularies; you've now started to build up quite a vocabulary of your own! Not to mention, you have also learned a full set of OO design principles from which you can easily understand the motivation and workings of any new patterns you encounter.

Now that you've got the Design Pattern basics down, it's time for you to go out and spread the word to others. Why? Because when your fellow developers know patterns and use a shared vocabulary as well, it leads to better designs, better communication and, best of all, it'll save you a lot of time that you can spend on cooler things.



So I created this broadcast class. It keeps track of all the objects listening to it and anytime a new piece of data comes along it sends a message to each listener. What's cool is that the listeners can join the broadcast at any time or they can even remove themselves. And the broadcast class itself doesn't know anything about the listeners, any object can register that implements the right interface.

Incomplete

Confusing

Time-consuming

Top five ways to share your vocabulary

- 1 In design meetings:** When you meet with your team to discuss a software design, use design patterns to help stay "in the design" longer. Discussing designs from the perspective of Design Patterns and OO principles keeps your team from getting bogged down in implementation details and prevent many misunderstandings.
- 2 With other developers:** Use patterns in your discussions with other developers. This helps other developers learn about new patterns and builds a community. The best part about sharing what you've learned is that great feeling when someone else "gets it!"
- 3 In architecture documentation:** When you write architectural documentation, using patterns will reduce the amount of documentation you need to write and gives the reader a clearer picture of the design.
- 4 In code comments and naming conventions:** When you're writing code, clearly identify the patterns you're using in comments. Also, choose class and methods names that reveal any patterns underneath. Other developers who have to read your code will thank you for allowing them to quickly understand your implementation.
- 5 To groups of interested developers:** Share your knowledge. Many developers have heard about patterns but don't have a good understanding of what they are. Volunteer to give a brown-bag lunch on patterns or a talk at your local user group.



Cruisin' Objectville with the Gang of Four

You won't find the Jets or Sharks hanging around Objectville, but you will find the Gang of Four. As you've probably noticed, you can't get far in the World of Patterns without running into them. So, who is this mysterious gang?

Put simply, "the GoF," which includes Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides, is the group of guys who put together the first patterns catalog and in the process, started an entire movement in the software field!

How did they get that name? No one knows for sure; it's just a name that stuck. But think about it: if you're going to have a "gang element" running around Objectville, could you think of a nicer bunch of guys? In fact, they've even agreed to pay us a visit...

The GoF launched the software patterns movement, but many others have made significant contributions, including Ward Cunningham, Kent Beck, Jim Coplien, Grady Booch, Bruce Anderson, Richard Gabriel, Doug Lea, Peter Coad, and Doug Schmidt, to name just a few.



Your journey has just begun...

Now that you're on top of Design Patterns and ready to dig deeper, we've got three definitive texts that you need to add to your bookshelf...



The definitive Design Patterns text

This is the book that kicked off the entire field of Design Patterns when it was released in 1995. You'll find all the fundamental patterns here. In fact, this book is the basis for the set of patterns we used in *Head First Design Patterns*.

You won't find this book to be the last word on Design Patterns – the field has grown substantially since its publication – but it is the first and most definitive.

Picking up a copy of *Design Patterns* is a great way to start exploring patterns after *Head First*.

The authors of *Design Patterns* are affectionately known as the "Gang of Four" or GoF for short.

Christopher Alexander invented patterns, which inspired applying similar solutions to software.

The definitive Patterns texts

Patterns didn't start with the GoF; they started with Christopher Alexander, a Professor of Architecture at Berkeley – that's right, Alexander is an *architect*, not a computer scientist. Alexander invented patterns for building living architectures (like houses, towns and cities).

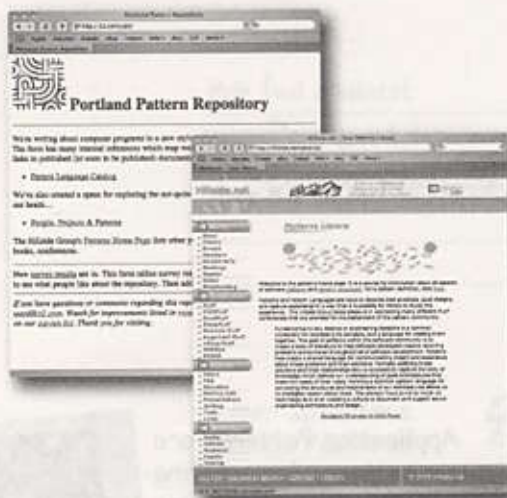
The next time you're in the mood for some deep, engaging reading, pick up *The Timeless Way of Building* and *A Pattern Language*. You'll see the true beginnings of Design Patterns and recognize the direct analogies between creating "living architecture" and flexible, extensible software.

So grab a cup of Starbuzz Coffee, sit back, and enjoy...



Other Design Pattern resources

You're going to find there is a vibrant, friendly community of patterns users and writers out there and they're glad to have you join them. Here's a few resources to get you started...



Websites

The Portland Patterns Repository, run by Ward Cunningham, is a WIKI devoted to all things related to patterns. Anyone can participate. You'll find threads of discussion on every topic you can think of related to patterns and OO systems.

<http://c2.com/cgi/wiki?WelcomeVisitors>

The Hillside Group fosters common programming and design practices and provides a central resource for patterns work. The site includes information on many patterns-related resources such as articles, books, mailing lists and tools.

<http://hillside.net/>



Conferences and Workshops

And if you'd like to get some face-to-face time with the patterns community, be sure to check out the many patterns related conferences and workshops. The Hillside site maintains a complete list. At the least you'll want to check out OOPSLA, the ACM Conference on Object-Oriented Systems, Languages and Applications.

The Patterns Zoo

As you've just seen, patterns didn't start with software; they started with the architecture of buildings and towns. In fact, the patterns concept can be applied in many different domains. Take a walk around the Patterns Zoo to see a few...



Architectural Patterns are used to create the living, vibrant architecture of buildings, towns, and cities. This is where patterns got their start.

Habitat: found in buildings you like to live in, look at and visit.

Habitat: seen hanging around 3-tier architectures, client-server systems and the web.

Application Patterns are patterns for creating system level architecture. Many multi-tier architectures fall into this category.



Field note: MVC has been known to pass for an application pattern.



Domain-Specific Patterns are patterns that concern problems in specific domains, like concurrent systems or real-time systems.

Help find a habitat

J2EE

Business Process Patterns describe the interaction between businesses, customers and data, and can be applied to problems such as how to effectively make and communicate decisions.



Seen hanging around corporate boardrooms and project management meetings.

Help find a habitat

Development team

Customer support team

Organizational Patterns describe the structures and practices of human organizations. Most efforts to date have focused on organizations that produce and/or support software.



User Interface Design Patterns address the problems of how to design interactive software programs.

Habitat: seen in the vicinity of video game designers, GUI builders, and producers.

Field notes: please add your observations of pattern domains here:

Annihilating evil with Anti-Patterns

The Universe just wouldn't be complete if we had patterns and no anti-patterns, now would it?

If a Design Pattern gives you a general solution to a recurring problem in a particular context, then what does an anti-pattern give you?

An **Anti-Pattern** tells you how to go from a problem to a BAD solution.

You're probably asking yourself, "Why on earth would anyone waste their time documenting bad solutions?"

Think about it like this; if there is a recurring bad solution to a common problem, then by documenting it we can prevent other developers from making the same mistake. After all, avoiding bad solutions can be just as valuable as finding good ones!

Let's look at the elements of an anti-pattern:

An anti-pattern tells you why a bad solution is attractive. Let's face it, no one would choose a bad solution if there wasn't something about it that seemed attractive up front. One of the biggest jobs of the anti-pattern is to alert you to the seductive aspect of the solution.

An anti-pattern tells you why that solution in the long term is bad. In order to understand why it's an anti-pattern, you've got to understand how it's going to have a negative effect down the road. The anti-pattern describes where you'll get into trouble using the solution.

An anti-pattern suggests other patterns that are applicable which may provide good solutions. To be truly helpful an anti-pattern needs to point you in the right direction; it should suggest other possibilities that may lead to good solutions.

Let's have a look at an anti-pattern.



An anti-pattern always looks like a good solution, but then turns out to be a bad solution when it is applied.

By documenting anti-patterns we help others to recognize bad solutions before they implement them.

Like patterns, there are many types of anti-patterns including development, OO, organizational, and domain specific anti-patterns.

Here's an example of a software development anti-pattern.

Just like a Design Pattern, an anti-pattern has a name so we can create a shared vocabulary.

The problem and context, just like a Design Pattern description.

Tells you why the solution is attractive.

The bad, yet attractive solution.

How to get to a good solution.

Example of where this anti-pattern has been observed.

Adapted from the Portland Pattern Repository's WIKI at <http://e2.com/> where you'll find many anti patterns and discussions.



Anti-Pattern

Name: Golden Hammer

Problem: You need to choose technologies for your development and you believe that exactly one technology must dominate the architecture.

Context: You need to develop some new system or piece of software that doesn't fit well with the technology that the development team is familiar with.

Forces:

- The development team is committed to the technology they know.
- The development team is not familiar with other technologies.
- Unfamiliar technologies are seen as risky.
- It is easy to plan and estimate for development using the familiar technology.

Supposed Solution: Use the familiar technology anyway. The technology is applied obsessively to many problems, including places where it is clearly inappropriate.

Refactored Solution: Expanding the knowledge of developers through education, training, and book study groups that expose developers to new solutions.

Examples:

Web companies keep using and maintaining their internal homegrown caching systems when open source alternatives are in use.

D-G

Command Pattern, continued

logging requests 229

macro command 224

Null Object 214

queuing requests 228

undo 216, 220, 227

Composite Pattern

and Iterator Pattern 368

class diagram 358

combining patterns 513

composite behavior 363

default behavior 360

defined 356

interview 376–377

safety 367

safety versus transparency 515

transparency 367, 375

composition 23, 85, 93, 247, 309

compound pattern 500, 522

controlling access 460. *See also* Proxy Pattern

creating objects 134

crossword puzzle 33, 76, 163, 187, 231, 271, 310, 378, 490

cubicle conversation 55, 93, 195, 208, 387, 397, 433, 583–584

D

Decorator Pattern

and Proxy Pattern 472–473

class diagram 91

combining patterns 506

cubicle conversation 93

defined 91

disadvantages 101, 104

fireside chat 252–253

interview 104

introduction 88

in Java I/O 100–101

structural pattern 591

Dependency Inversion Principle 139–143

and the Hollywood Principle 298

Design Patterns

Abstract Factory Pattern 156

Adapter Pattern 243

benefits 599

Bridge Pattern 612–613

Builder Pattern 614–615

categories 589, 592–593

Chain of Responsibility Pattern 616–617

class patterns 591

Command Pattern 206

Composite Pattern 356

Decorator Pattern 91

defined 579, 581

discover your own 586–587

Facade Pattern 264

Factory Method Pattern 134

Flyweight Pattern 618–619

Interpreter Pattern 620–621

Iterator Pattern 336

Mediator Pattern 622–623

Memento Pattern 624–625

Null Object 214

object patterns 591

Observer Pattern 51

organizing 589

Prototype Pattern 626–627

Proxy Pattern 460

next 3 pages
just
duplicated

Annihilating evil with Anti-Patterns

The Universe just wouldn't be complete if we had patterns and no anti-patterns, now would it?

If a Design Pattern gives you a general solution to a recurring problem in a particular context, then what does an anti-pattern give you?

An **Anti-Pattern** tells you how to go from a problem to a BAD solution.

You're probably asking yourself, "Why on earth would anyone waste their time documenting bad solutions?"

Think about it like this: if there is a recurring bad solution to a common problem, then by documenting it we can prevent other developers from making the same mistake. After all, avoiding bad solutions can be just as valuable as finding good ones!

Let's look at the elements of an anti-pattern:

An anti-pattern tells you why a bad solution is attractive. Let's face it, no one would choose a bad solution if there wasn't something about it that seemed attractive up front. One of the biggest jobs of the anti-pattern is to alert you to the seductive aspect of the solution.

An anti-pattern tells you why that solution in the long term is bad. In order to understand why it's an anti-pattern, you've got to understand how it's going to have a negative effect down the road. The anti-pattern describes where you'll get into trouble using the solution.

An anti-pattern suggests other patterns that are applicable which may provide good solutions. To be truly helpful an anti-pattern needs to point you in the right direction; it should suggest other possibilities that may lead to good solutions.

Let's have a look at an anti-pattern.



An anti-pattern always looks like a good solution, but then turns out to be a bad solution when it is applied.

By documenting anti-patterns we help others to recognize bad solutions before they implement them.

Like patterns, there are many types of anti-patterns including development, OO, organizational, and domain specific anti-patterns.

Here's an example of a software development anti-pattern.

Just like a Design Pattern, an anti-pattern has a name so we can create a shared vocabulary.

The problem and context, just like a Design Pattern description.

Tells you why the solution is attractive.

The bad, yet attractive solution.

How to get to a good solution.

Example of where this anti-pattern has been observed.

Adapted from the Portland Pattern Repository's WIKI at <http://e2.com/> where you'll find many anti patterns and discussions.



Anti-Pattern

Name: Golden Hammer

Problem: You need to choose technologies for your development and you believe that exactly one technology must dominate the architecture.

Context: You need to develop some new system or piece of software that doesn't fit well with the technology that the development team is familiar with.

Forces:

- The development team is committed to the technology they know.
- The development team is not familiar with other technologies.
- Unfamiliar technologies are seen as risky.
- It is easy to plan and estimate for development using the familiar technology.

Supposed Solution: Use the familiar technology anyway. The technology is applied obsessively to many problems, including places where it is clearly inappropriate.

Refactored Solution: Expanding the knowledge of developers through education, training, and book study groups that expose developers to new solutions.

Examples:

Web companies keep using and maintaining their internal homegrown caching systems when open source alternatives are in use.



Tools for your Design Toolbox

You've reached that point where you've outgrown us. Now's the time to go out in the world and explore patterns on your own...

OO Principles

Encapsulate what varies.
 Favor composition over inheritance.
 Program to interfaces, not implementations.
 Strive for loosely coupled designs between objects that interact.
 Classes should be open for extension but closed for modification.
 Depend on abstractions. Do not depend on concrete classes.
 Only talk to your friends.
 Don't call us, we'll call you.
 A class should have only one reason to change.

OO Basics

Abstraction
 Encapsulation
 Polymorphism
 Inheritance

The time has come for you to go out and discover more patterns on your own. There are many domain-specific patterns we haven't even mentioned and there are also some foundational ones we didn't cover. You've also got patterns of your own to create.

OO Patterns

Proxy - Proxy placeholder for control access

Comp

A Composite or more solves a r

Your Patterns Here!

Check out the Appendix, we'll give you a heads up on some more foundational patterns you'll probably want to have a look at.

BULLET POINTS

- Let Design Patterns emerge in your designs, don't force them in just for the sake of using a pattern.
- Design Patterns aren't set in stone; adapt and tweak them to meet your needs.
- Always use the simplest solution that meets your needs, even if it doesn't include a pattern.
- Study Design Pattern catalogs to familiarize yourself with patterns and the relationships among them.
- Pattern classifications (or categories) provide groupings for patterns. When they help, use them.
- You need to be committed to be a patterns writer: it takes time and patience, and you have to be willing to do lots of refinement.
- Remember, most patterns you encounter will be adaptations of existing patterns, not new patterns.
- Build your team's shared vocabulary. This is one of the most powerful benefits of using patterns.
- Like any community, the patterns community has its own lingo. Don't let that hold you back. Having read this book, you now know most of it.

Leaving Objectville...



Boy, it's been great having you in Objectville.

We're going to miss you, for sure. But don't worry – before you know it, the next Head First book will be out and you can visit again. What's the next book, you ask? Hmmm, good question! Why don't you help us decide? Send email to booksuggestions@wickedlysmart.com.

Exercise solutions

WHO DOES WHAT?

Match each pattern with its description:

Pattern	Description
Decorator	Wraps an object and provides a different interface to it.
State	Subclasses decide how to implement steps in an algorithm.
Iterator	Subclasses decide which concrete classes to create.
Facade	Ensures one and only object is created.
Strategy	Encapsulates interchangeable behaviors and uses delegation to decide which one to use.
Proxy	Clients treat collections of objects and individual objects uniformly.
Factory Method	Encapsulates state-based behaviors and uses delegation to switch between behaviors.
Adapter	Provides a way to traverse a collection of objects without exposing its implementation.
Observer	Simplifies the interface of a set of classes.
Template Method	Wraps an object to provide new behavior.
Composite	Allows a client to create families of objects without specifying their concrete classes.
Singleton	Allows objects to be notified when state changes.
Abstract Factory	Wraps an object to control access to it.
Command	Encapsulates a request as an object.