



Ready-bake Code

```
menu.add(exit);
menuBar.add(menu);
controlFrame.setJMenuBar(menuBar);

bpmTextField = new JTextField(2);
bpmLabel = new JLabel("Enter BPM:", SwingConstants.RIGHT);
setBPMButton = new JButton("Set");
setBPMButton.setSize(new Dimension(10, 40));
increaseBPMButton = new JButton(">>");
decreaseBPMButton = new JButton("<<");
setBPMButton.addActionListener(this);
increaseBPMButton.addActionListener(this);
decreaseBPMButton.addActionListener(this);

 JPanel buttonPanel = new JPanel(new GridLayout(1, 2));

buttonPanel.add(decreaseBPMButton);
buttonPanel.add(increaseBPMButton);

 JPanel enterPanel = new JPanel(new GridLayout(1, 2));
enterPanel.add(bpmLabel);
enterPanel.add(bpmTextField);
 JPanel insideControlPanel = new JPanel(new GridLayout(3, 1));
insideControlPanel.add(enterPanel);
insideControlPanel.add(setBPMButton);
insideControlPanel.add(buttonPanel);
controlPanel.add(insideControlPanel);

bpmLabel.setBorder(BorderFactory.createEmptyBorder(5,5,5,5));
bpmOutputLabel.setBorder(BorderFactory.createEmptyBorder(5,5,5,5));

controlFrame.getRootPane().setDefaultButton(setBPMButton);
controlFrame.getContentPane().add(controlPanel, BorderLayout.CENTER);

controlFrame.pack();
controlFrame.setVisible(true);
}

public void enableStopMenuItem() {
    stopMenuItem.setEnabled(true);
}

public void disableStopMenuItem() {
    stopMenuItem.setEnabled(false);
}
```

```

}

public void enableStartMenuItem() {
    startMenuItem.setEnabled(true);
}

public void disableStartMenuItem() {
    startMenuItem.setEnabled(false);
}

public void actionPerformed(ActionEvent event) {
    if (event.getSource() == setBPMButton) {
        int bpm = Integer.parseInt(bpmTextField.getText());
        controller.setBPM(bpm);
    } else if (event.getSource() == increaseBPMButton) {
        controller.increaseBPM();
    } else if (event.getSource() == decreaseBPMButton) {
        controller.decreaseBPM();
    }
}

public void updateBPM() {
    int bpm = model.getBPM();
    if (bpm == 0) {
        bpmOutputLabel.setText("offline");
    } else {
        bpmOutputLabel.setText("Current BPM: " + model.getBPM());
    }
}

public void updateBeat() {
    beatBar.setValue(100);
}
}

```

The Controller

```

package headfirst.combined.djview;

public interface ControllerInterface {
    void start();
    void stop();
    void increaseBPM();
    void decreaseBPM();
    void setBPM(int bpm);
}

```

ready-bake code: controller

Ready-bake Code



```
package headfirst.combined.djview;

public class BeatController implements ControllerInterface {
    BeatModelInterface model;
    DJView view;

    public BeatController(BeatModelInterface model) {
        this.model = model;
        view = new DJView(this, model);
        view.createView();
        view.createControls();
        view.disableStopMenuItem();
        view.enableStartMenuItem();
        model.initialize();
    }

    public void start() {
        model.on();
        view.disableStartMenuItem();
        view.enableStopMenuItem();
    }

    public void stop() {
        model.off();
        view.disableStopMenuItem();
        view.enableStartMenuItem();
    }

    public void increaseBPM() {
        int bpm = model.getBPM();
        model.setBPM(bpm + 1);
    }

    public void decreaseBPM() {
        int bpm = model.getBPM();
        model.setBPM(bpm - 1);
    }

    public void setBPM(int bpm) {
        model.setBPM(bpm);
    }
}
```

The Heart Model

```

package headfirst.combined.djview;

public class HeartTestDrive {
    public static void main (String[] args) {
        HeartModel heartModel = new HeartModel();
        ControllerInterface model = new HeartController(heartModel);
    }
}

package headfirst.combined.djview;
public interface HeartModelInterface {
    int getHeartRate();
    void registerObserver(BeatObserver o);
    void removeObserver(BeatObserver o);
    void registerObserver(BPMObserver o);
    void removeObserver(BPMObserver o);
}

package headfirst.combined.djview;
import java.util.*;

public class HeartModel implements HeartModelInterface, Runnable {
    ArrayList beatObservers = new ArrayList();
    ArrayList bpmObservers = new ArrayList();
    int time = 1000;
    int bpm = 90;
    Random random = new Random(System.currentTimeMillis());
    Thread thread;

    public HeartModel() {
        thread = new Thread(this);
        thread.start();
    }

    public void run() {
        int lastrate = -1;

        for (;;) {
            int change = random.nextInt(10);
            if (random.nextInt(2) == 0) {
                change = 0 - change;
            }
            int rate = 60000/(time + change);
            if (rate < 120 && rate > 50) {
                time += change;
                for (BeatObserver o : beatObservers)
                    o.update(lastrate, rate);
                for (BPMObserver o : bpmObservers)
                    o.update(lastrate, rate);
            }
            lastrate = rate;
        }
    }
}

```

ready-bake code: heart beat model

```
        notifyBeatObservers();
        if (rate != lastrate) {
            lastrate = rate;
            notifyBPMObservers();
        }
    }
    try {
        Thread.sleep(time);
    } catch (Exception e) {}
}
public int getHeartRate() {
    return 60000/time;
}

public void registerObserver(BeatObserver o) {
    beatObservers.add(o);
}

public void removeObserver(BeatObserver o) {
    int i = beatObservers.indexOf(o);
    if (i >= 0) {
        beatObservers.remove(i);
    }
}

public void notifyBeatObservers() {
    for(int i = 0; i < beatObservers.size(); i++) {
        BeatObserver observer = (BeatObserver)beatObservers.get(i);
        observer.updateBeat();
    }
}

public void registerObserver(BPMObserver o) {
    bpmObservers.add(o);
}

public void removeObserver(BPMObserver o) {
    int i = bpmObservers.indexOf(o);
    if (i >= 0) {
        bpmObservers.remove(i);
    }
}

public void notifyBPMObservers() {
    for(int i = 0; i < bpmObservers.size(); i++) {
        BPMObserver observer = (BPMObserver)bpmObservers.get(i);
        observer.updateBPM();
    }
}
```

Ready-bake Code



The Heart Adapter

```

package headfirst.combined.djview;
public class HeartAdapter implements BeatModelInterface {
    HeartModelInterface heart;

    public HeartAdapter(HeartModelInterface heart) {
        this.heart = heart;
    }

    public void initialize() {}

    public void on() {}

    public void off() {}

    public int getBPM() {
        return heart.getHeartRate();
    }

    public void setBPM(int bpm) {}

    public void registerObserver(BeatObserver o) {
        heart.registerObserver(o);
    }

    public void removeObserver(BeatObserver o) {
        heart.removeObserver(o);
    }

    public void registerObserver(BPMObserver o) {
        heart.registerObserver(o);
    }

    public void removeObserver(BPMObserver o) {
        heart.removeObserver(o);
    }
}

```

Alright, now you're ready for a bright new world filled with Design Patterns. But, before you go, one of those new design patterns you've come across is one that you'll encounter all too frequently—Design Patterns themselves! There are plenty of them, and they can be quite useful if you know how to use them. So, here's a brief introduction to some of the most common ones. You'll learn more about them in Chapter 10. For now, just take a look at the first few through the introduction on the next page.

ready-bake code: heart beat controller

The Controller

Ready-bake Code



```
package headfirst.combined.djview;

public class HeartController implements ControllerInterface {
    HeartModelInterface model;
    DJView view;

    public HeartController(HeartModelInterface model) {
        this.model = model;
        view = new DJView(this, new HeartAdapter(model));
        view.createView();
        view.createControls();
        view.disableStopMenuItem();
        view.disableStartMenuItem();
    }

    public void start() {}

    public void stop() {}

    public void increaseBPM() {}

    public void decreaseBPM() {}

    public void setBPM(int bpm) {}
}
```

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.



I've been thinking
about the three-part
definition, and I don't think
it defines a pattern at all.

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*.

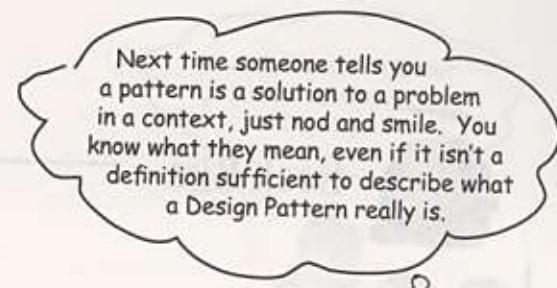
Q: Am I going to see pattern descriptions that are stated as a problem, a context and a solution?

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.



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?

better living with patterns

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.

SINGLETTON Object Creational

Intent
El algoritmo, en el que se lleva la vida de un solo objeto, que responde a los servicios que recibe de su entorno. El objeto es el centro de la actividad y se encarga de manejar las interacciones con el resto del sistema.

Motivation
El algoritmo, en el que se lleva la vida de un solo objeto, que responde a los servicios que recibe de su entorno. El objeto es el centro de la actividad y se encarga de manejar las interacciones con el resto del sistema.

Applicability
Este patrón es útil cuando se necesita tener un solo objeto que responda a los servicios que recibe de su entorno. El objeto es el centro de la actividad y se encarga de manejar las interacciones con el resto del sistema.

Structure

Participants
• Clase Singleton
• Clases que interactúan con Singleton
• Clases que dependen de Singleton

Collaborations
• El objeto Singleton interactúa con las clases que dependen de él para proveerles servicios.

Consequences
• El patrón Singleton provoca problemas de concurrencia al permitir que solo exista una instancia de la clase.

Implementation/Sample Code

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

Known Uses
Este patrón es común en sistemas web para manejar sesiones de usuario.

Related Patterns
Este patrón es similar al patrón Factory Method, ya que ambos manejan la creación de objetos.

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:

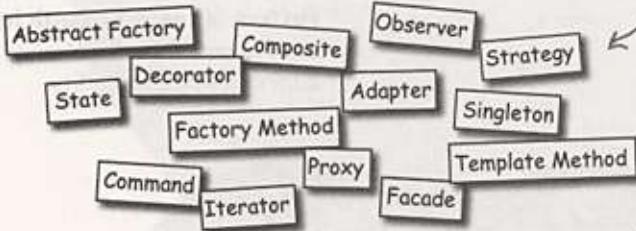
Pattern	Description
Decorator	Wraps an object and provides a different interface to it.
State	Subclasses decide how to implement steps in an algorithm.
Iterator	Subclasses decide which concrete classes to create.
Facade	Ensures one and only object is created.
Strategy	Encapsulates interchangeable behaviors 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.

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 tough! But give it your best shot and then check out the answers on the next page.

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

Each of these patterns belongs in one of those categories

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

Creational

Behavioral

Structural

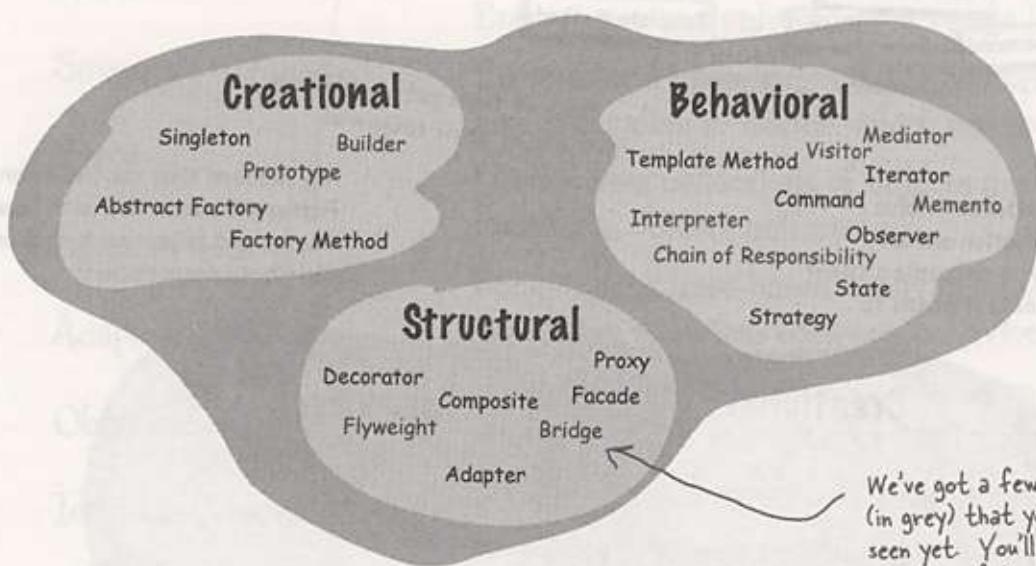
Structural patterns let you compose classes or objects into larger structures.

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.

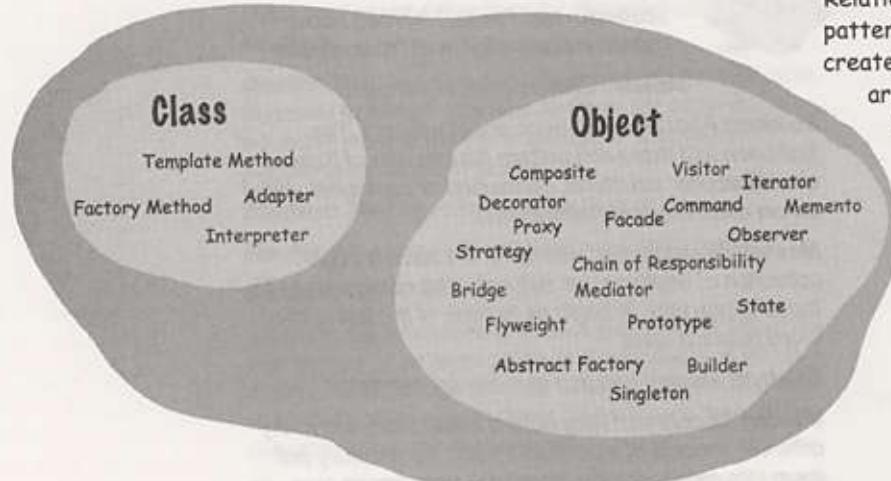


Structural patterns let you compose classes or objects into larger structures.

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

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.

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.



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.



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

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.

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.

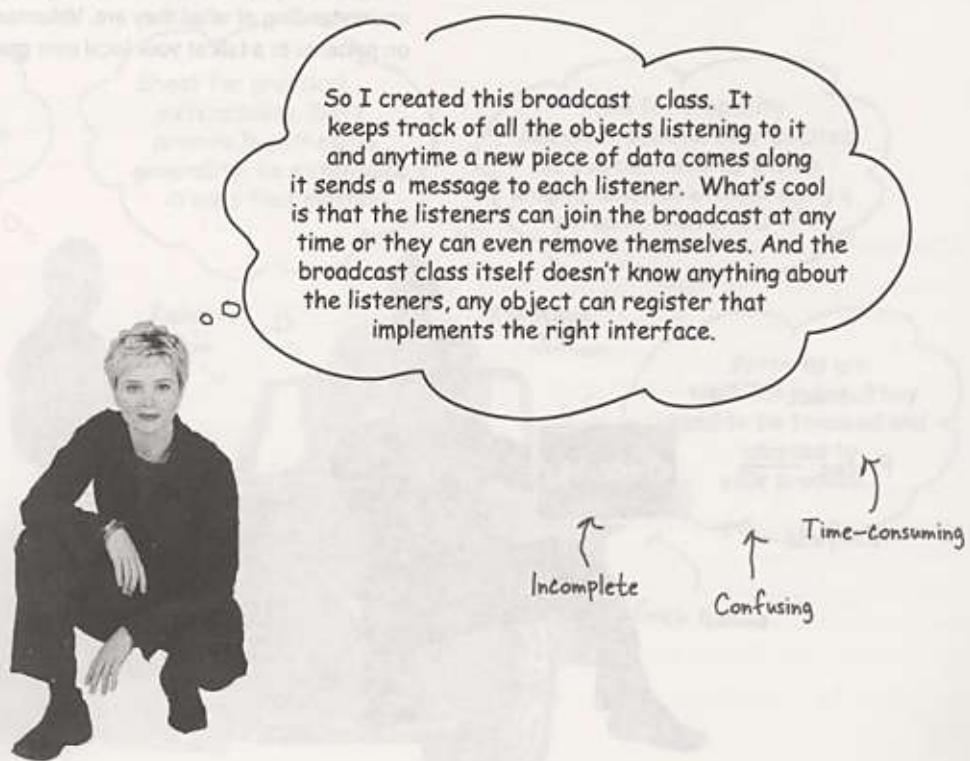
Don't let us discourage you, though. When a Design Pattern is the right tool for the job, the advantages are many.

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.



five ways to share your vocabulary

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!

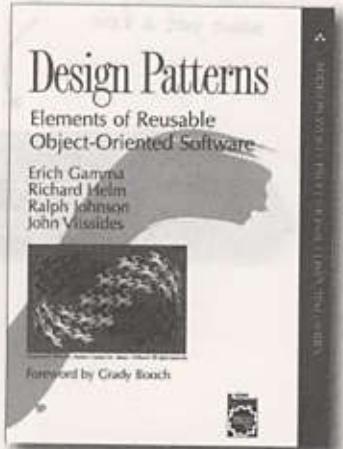
How did they get that name? No one knows for sure; it's just a name that stuck. But think about it: if you're going to have a "gang element" running around Objectville, could you think of a nicer bunch of guys? In fact, they've even agreed to pay us a visit...

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).

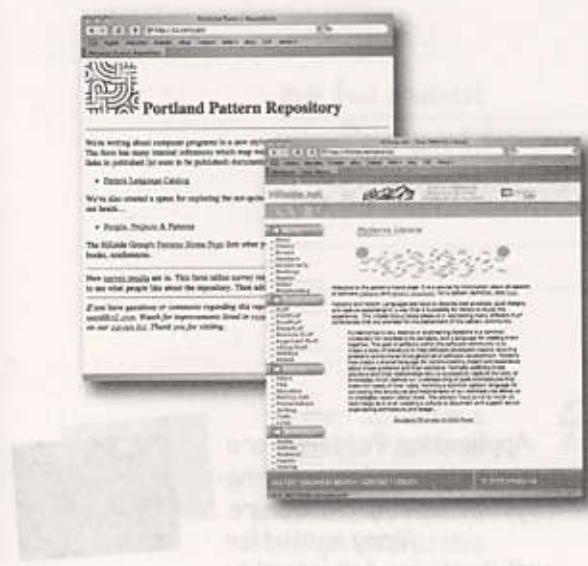
The next time you're in the mood for some deep, engaging reading, pick up *The Timeless Way of Building* and *A Pattern Language*. You'll see the true beginnings of Design Patterns and recognize the direct analogies between creating "living architecture" and flexible, extensible software.

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.

Architecture Patterns

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:

Anihilating 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.

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://c2.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.

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://c2.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

OO Patterns

- Proxy - Provides a placeholder for control access
- Comp - A Composite or more! solves a r

Your Patterns Here!

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.



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? Hmm, 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.

14 Appendix

Appendix: Leftover Patterns



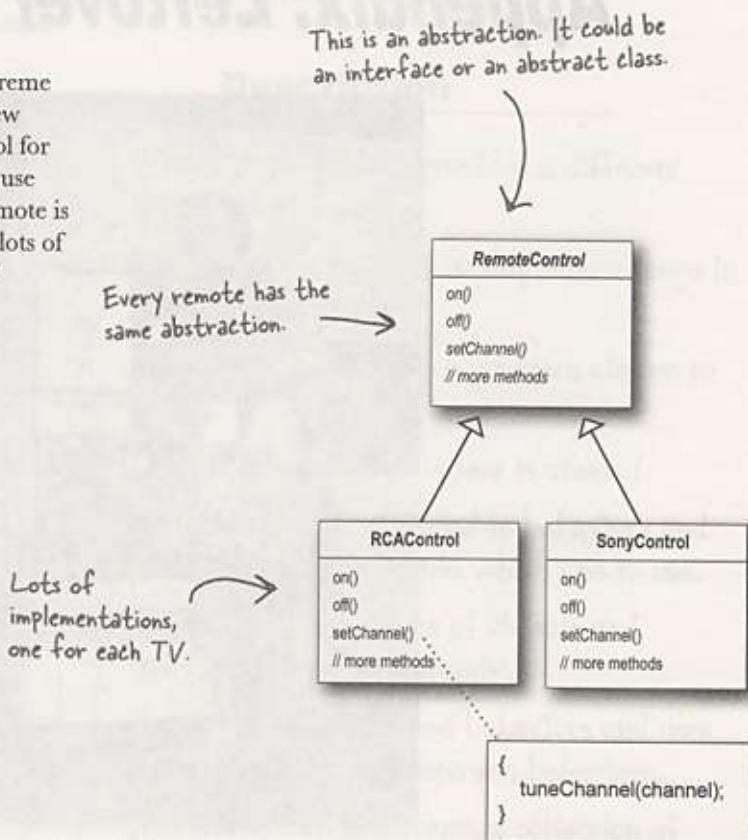
Not everyone can be the most popular. A lot has changed in the last 10 years. Since *Design Patterns: Elements of Reusable Object-Oriented Software* first came out, developers have applied these patterns thousands of times. The patterns we summarize in this appendix are full-fledged, card-carrying, official GoF patterns, but aren't always used as often as the patterns we've explored so far. But these patterns are awesome in their own right, and if your situation calls for them, you should apply them with your head held high. Our goal in this appendix is to give you a high level idea of what these patterns are all about.

Bridge

Use the Bridge Pattern to vary not only your implementations, but also your abstractions.

A scenario

Imagine you're going to revolutionize "extreme lounging." You're writing the code for a new ergonomic and user-friendly remote control for TVs. You already know that you've got to use good OO techniques because while the remote is based on the same *abstraction*, there will be lots of *implementations* – one for each model of TV.



Your dilemma

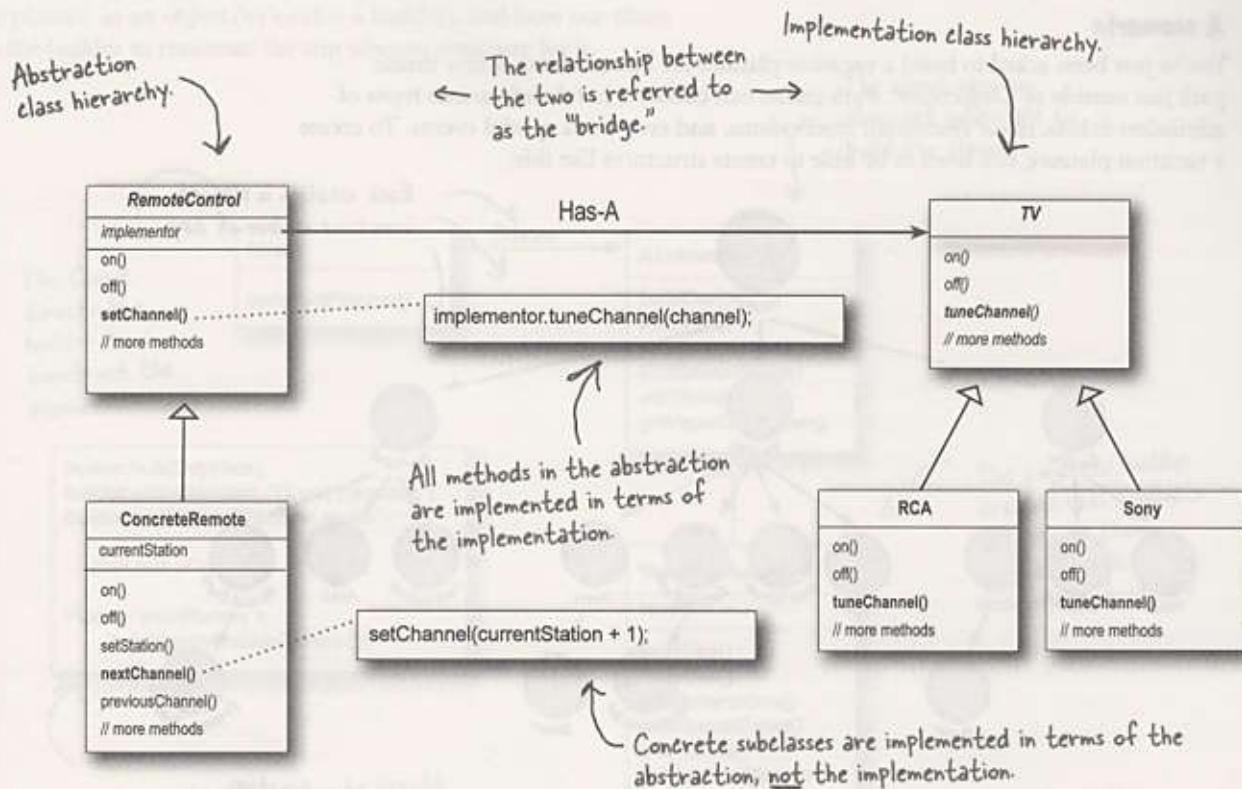
You know that the remote's user interface won't be right the first time. In fact, you expect that the product will be refined many times as usability data is collected on the remote control.

So your dilemma is that the remotes are going to change and the TVs are going to change. You've already *abstracted* the user interface so that you can vary the *implementation* over the many TVs your customers will own. But you are also going to need to *vary the abstraction* because it is going to change over time as the remote is improved based on the user feedback.

So how are you going to create an OO design that allows you to vary the implementation *and* the abstraction?

Why use the Bridge Pattern?

The Bridge Pattern allows you to vary the implementation *and* the abstraction by placing the two in separate class hierarchies.



Now you have two hierarchies, one for the remotes and a separate one for platform specific TV implementations. The bridge allows you to vary either side of the two hierarchies independently.

Bridge Benefits

- Decouples an implementation so that it is not bound permanently to an interface.
- Abstraction and implementation can be extended independently.
- Changes to the concrete abstraction classes don't affect the client.

Bridge Uses and Drawbacks

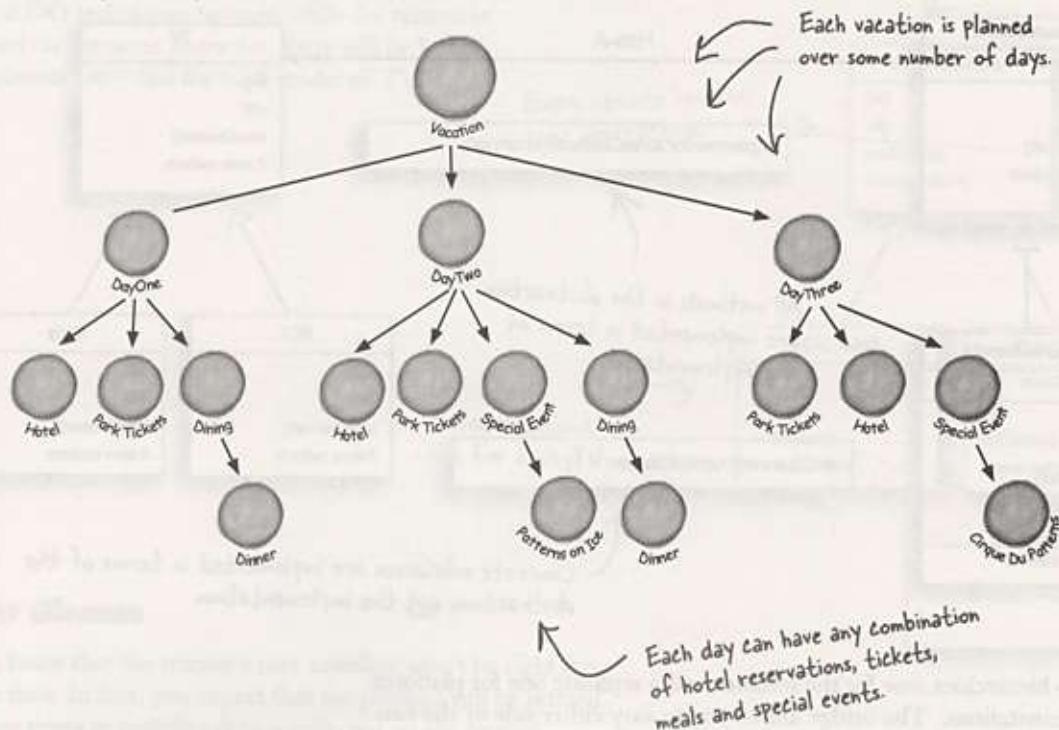
- Useful in graphic and windowing systems that need to run over multiple platforms.
- Useful any time you need to vary an interface and an implementation in different ways.
- Increases complexity.

Builder

Use the Builder Pattern to encapsulate the construction of a product and allow it to be constructed in steps.

A scenario

You've just been asked to build a vacation planner for Patternsland, a new theme park just outside of Objectville. Park guests can choose a hotel and various types of admission tickets, make restaurant reservations, and even book special events. To create a vacation planner, you need to be able to create structures like this:



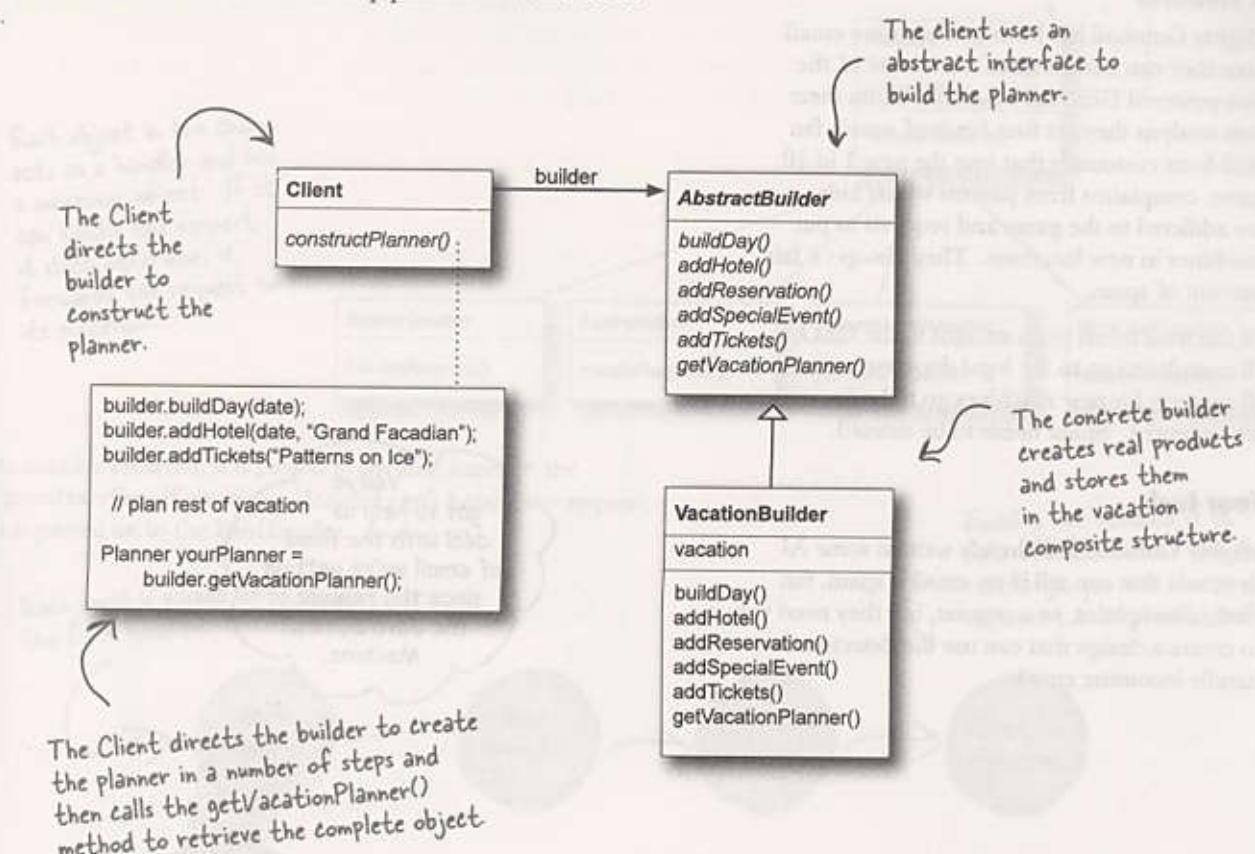
You need a flexible design

Each guest's planner can vary in the number of days and types of activities it includes. For instance, a local resident might not need a hotel, but wants to make dinner and special event reservations. Another guest might be flying into Objectville and needs a hotel, dinner reservations, and admission tickets.

So, you need a flexible data structure that can represent guest planners and all their variations; you also need to follow a sequence of potentially complex steps to create the planner. How can you provide a way to create the complex structure without mixing it with the steps for creating it?

Why use the Builder Pattern?

Remember Iterator? We encapsulated the iteration into a separate object and hid the internal representation of the collection from the client. It's the same idea here; we encapsulate the creation of the trip planner in an object (let's call it a builder), and have our client ask the builder to construct the trip planner structure for it.



Builder Benefits

- Encapsulates the way a complex object is constructed.
- Allows objects to be constructed in a multistep and varying process (as opposed to one step factories).
- Hides the internal representation of the product from the client.
- Product implementations can be swapped in and out because the client only sees an abstract interface.

Builder Uses and Drawbacks

- Often used for building composite structures.
- Constructing objects requires more domain knowledge of the client than when using a Factory.

Chain of Responsibility

Use the Chain of Responsibility Pattern when you want to give more than one object a chance to handle a request.

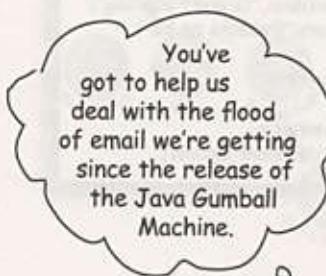
A scenario

Mighty Gumball has been getting more email than they can handle since the release of the Java-powered Gumball Machine. From their own analysis they get four kinds of email: fan mail from customers that love the new 1 in 10 game, complaints from parents whose kids are addicted to the game and requests to put machines in new locations. They also get a fair amount of spam.

All fan mail needs to go straight to the CEO, all complaints go to the legal department and all requests for new machines go to business development. Spam needs to be deleted.

Your task

Mighty Gumball has already written some AI detectors that can tell if an email is spam, fan mail, a complaint, or a request, but they need you to create a design that can use the detectors to handle incoming email.

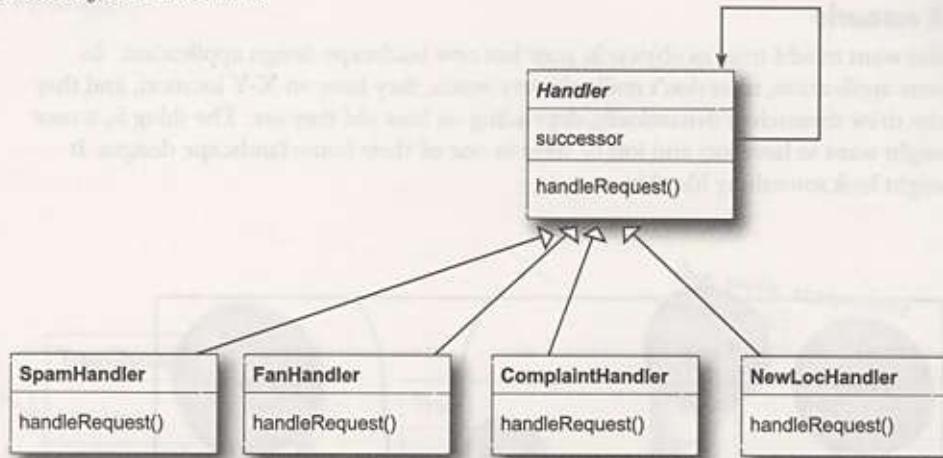


You've got to help us deal with the flood of email we're getting since the release of the Java Gumball Machine.

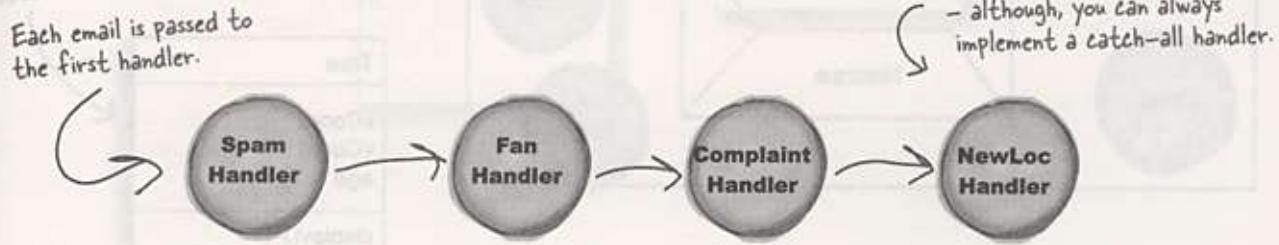
How to use the Chain of Responsibility Pattern

With the Chain of Responsibility Pattern, you create a chain of objects that examine a request. Each object in turn examines the request and handles it, or passes it on to the next object in the chain.

Each object in the chain acts as a handler and has a successor object. If it can handle the request, it does; otherwise, it forwards the request to its successor.



As email is received, it is passed to the first handler: the SpamHandler. If the SpamHandler can't handle the request, it is passed on to the FanHandler. And so on...



Chain of Responsibility Benefits

- Decouples the sender of the request and its receivers.
- Simplifies your object because it doesn't have to know the chain's structure and keep direct references to its members.
- Allows you to add or remove responsibilities dynamically by changing the members or order of the chain.

Chain of Responsibility Uses and Drawbacks

- Commonly used in windows systems to handle events like mouse clicks and keyboard events.
- Execution of the request isn't guaranteed; it may fall off the end of the chain if no object handles it (this can be an advantage or a disadvantage).
- Can be hard to observe the runtime characteristics and debug.

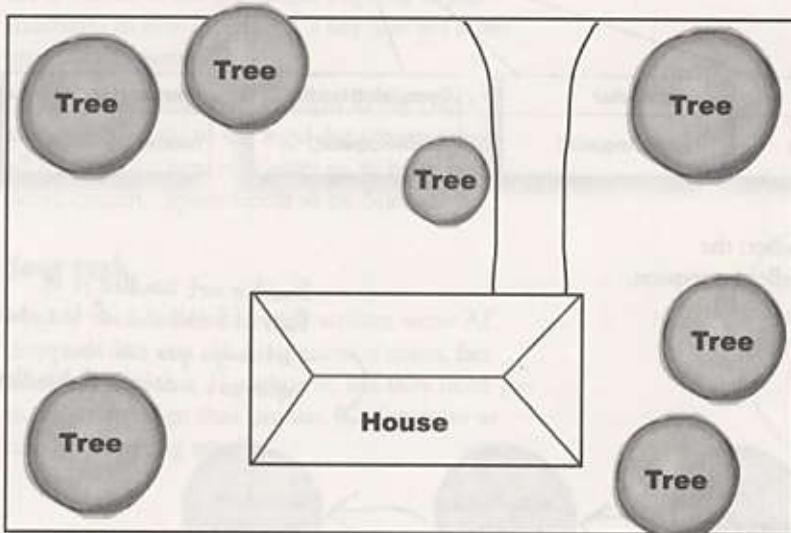
flyweight pattern

Flyweight

Use the Flyweight Pattern when one instance of a class can be used to provide many “virtual instances.”

A scenario

You want to add trees as objects in your hot new landscape design application. In your application, trees don't really do very much; they have an X-Y location, and they can draw themselves dynamically, depending on how old they are. The thing is, a user might want to have lots and lots of trees in one of their home landscape designs. It might look something like this:



Each Tree instance maintains its own state.

Tree
xCoord
yCoord
age
display() {
// use X-Y coords
// & complex age
// related calcs
}

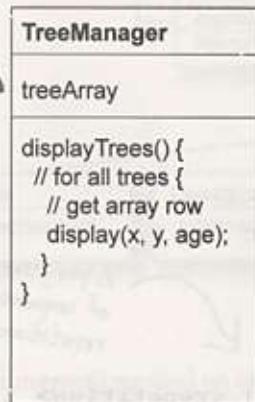
Your big client's dilemma

You've just landed your "reference account." That key client you've been pitching for months. They're going to buy 1,000 seats of your application, and they're using your software to do the landscape design for huge planned communities. After using your software for a week, your client is complaining that when they create large groves of trees, the app starts getting sluggish...

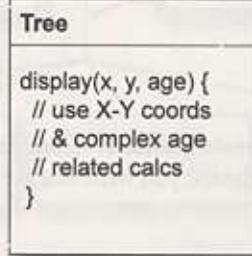
Why use the Flyweight Pattern?

What if, instead of having thousands of Tree objects, you could redesign your system so that you've got only one instance of Tree, and a client object that maintains the state of ALL your trees? That's the Flyweight!

All the state, for ALL
of your virtual Tree
objects, is stored in
this 2D-array.



One, single, state-free
Tree object



Flyweight Benefits

- Reduces the number of object instances at runtime, saving memory.
- Centralizes state for many "virtual" objects into a single location.

Flyweight Uses and Drawbacks

- The Flyweight is used when a class has many instances, and they can all be controlled identically.
- A drawback of the Flyweight pattern is that once you've implemented it, single, logical instances of the class will not be able to behave independently from the other instances.

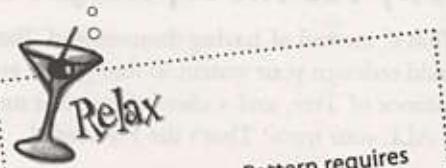
Interpreter

Use the Interpreter Pattern to build an interpreter for a language.

A scenario

Remember the Duck Pond Simulator? You have a hunch it would also make a great educational tool for children to learn programming. Using the simulator, each child gets to control one duck with a simple language. Here's an example of the language:

```
right;           Turn the duck right
while (daylight) fly;   Fly all day...
quack;          ...and then quack.
```



The Interpreter Pattern requires some knowledge of formal grammars. If you've never studied formal grammars, go ahead and read through the pattern; you'll still get the gist of it.

Now, remembering how to create grammars from one of your old introductory programming classes, you write out the grammar:

```
expression ::= <command> | <sequence> | <repetition>
sequence ::= <expression> ';' <expression>
command ::= right | quack | fly
repetition ::= while '(' <variable> ')' <expression>
variable ::= [A-Z,a-z]+
```

A program is an expression consisting of sequences of commands and repetitions ("while" statements).

A sequence is a set of expressions separated by semicolons.

We have three commands: right, quack, and fly.

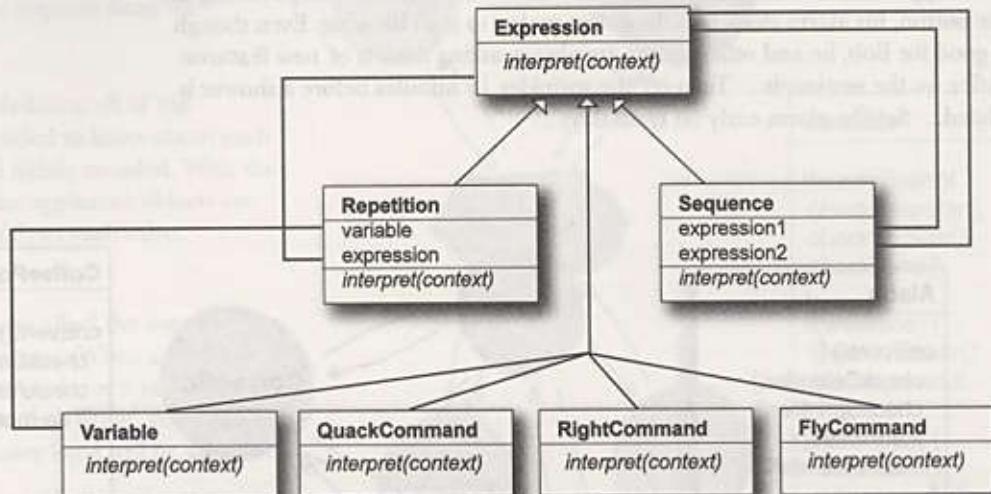
A while statement is just a conditional variable and an expression.

Now what?

You've got a grammar; now all you need is a way to represent and interpret sentences in the grammar so that the students can see the effects of their programming on the simulated ducks.

How to implement an interpreter

When you need to implement a simple language, the Interpreter Pattern defines a class-based representation for its grammar along with an interpreter to interpret its sentences. To represent the language, you use a class to represent each rule in the language. Here's the duck language translated into classes. Notice the direct mapping to the grammar.



To interpret the language, call the `interpret()` method on each expression type. This method is passed a context – which contains the input stream of the program we're parsing – and matches the input and evaluates it.

Interpreter Benefits

- Representing each grammar rule in a class makes the language easy to implement.
- Because the grammar is represented by classes, you can easily change or extend the language.
- By adding additional methods to the class structure, you can add new behaviors beyond interpretation, like pretty printing and more sophisticated program validation.

Interpreter Uses and Drawbacks

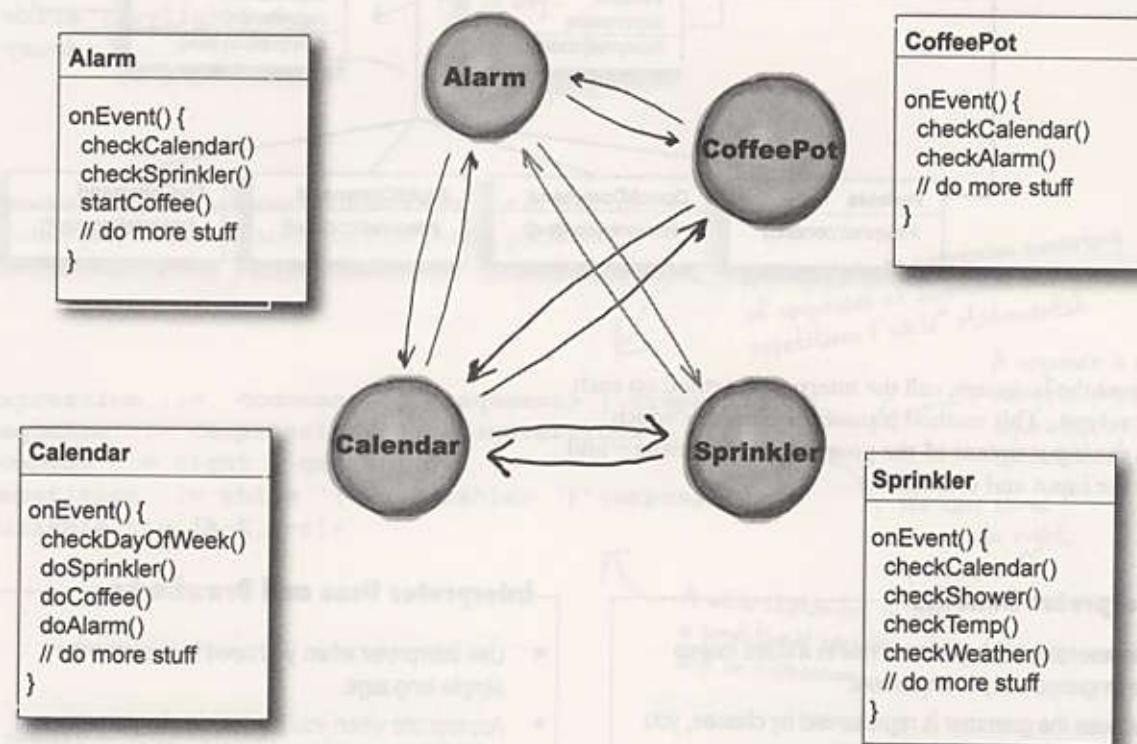
- Use interpreter when you need to implement a simple language.
- Appropriate when you have a simple grammar and simplicity is more important than efficiency.
- Used for scripting and programming languages.
- This pattern can become cumbersome when the number of grammar rules is large. In these cases a parser/compiler generator may be more appropriate.

Mediator

Use the Mediator Pattern to centralize complex communications and control between related objects.

A scenario

Bob has a Java-enabled auto-house, thanks to the good folks at HouseOfTheFuture. All of his appliances are designed to make his life easier. When Bob stops hitting the snooze button, his alarm clock tells the coffee maker to start brewing. Even though life is good for Bob, he and other clients are always asking for lots of new features: No coffee on the weekends... Turn off the sprinkler 15 minutes before a shower is scheduled... Set the alarm early on trash days...



HouseOfTheFuture's dilemma

It's getting really hard to keep track of which rules reside in which objects, and how the various objects should relate to each other.

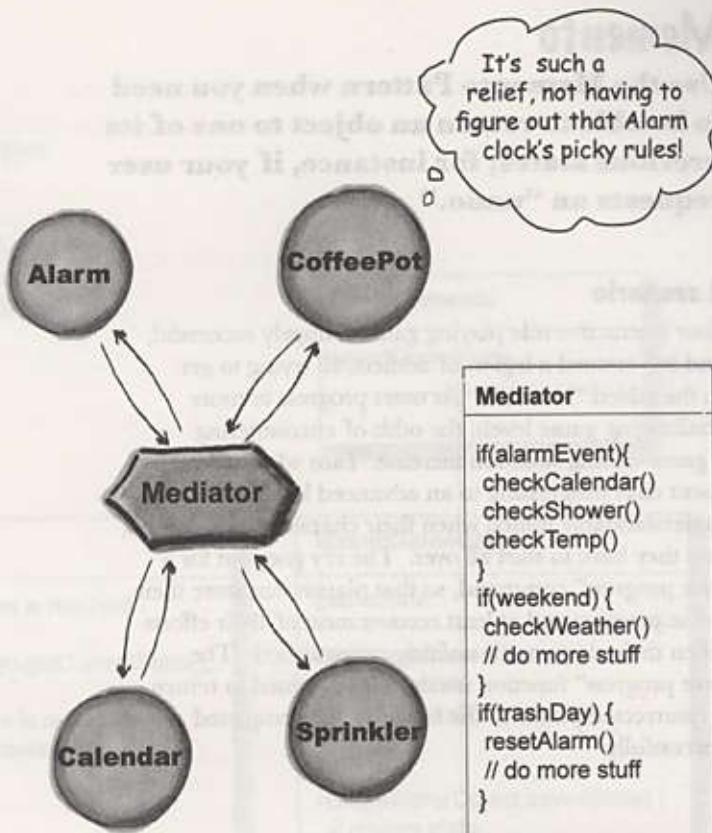
Mediator in action...

With a Mediator added to the system, all of the appliance objects can be greatly simplified:

- They tell the Mediator when their state changes.
- They respond to requests from the Mediator.

Before adding the Mediator, all of the appliance objects needed to know about each other... they were all tightly coupled. With the Mediator in place, the appliance objects are all *completely decoupled* from each other.

The Mediator contains all of the control logic for the entire system. When an existing appliance needs a new rule, or a new appliance is added to the system, you'll know that all of the necessary logic will be added to the Mediator.



Mediator Benefits

- Increases the reusability of the objects supported by the Mediator by decoupling them from the system.
- Simplifies maintenance of the system by centralizing control logic.
- Simplifies and reduces the variety of messages sent between objects in the system.

Mediator Uses and Drawbacks

- The Mediator is commonly used to coordinate related GUI components.
- A drawback of the Mediator pattern is that without proper design, the Mediator object itself can become overly complex.

memento pattern

Memento

Use the Memento Pattern when you need to be able to return an object to one of its previous states; for instance, if your user requests an “undo.”

A scenario

Your interactive role playing game is hugely successful, and has created a legion of addicts, all trying to get to the fabled “level 13.” As users progress to more challenging game levels, the odds of encountering a game-ending situation increase. Fans who have spent days progressing to an advanced level are understandably miffed when their character gets snuffed, and they have to start all over. The cry goes out for a “save progress” command, so that players can store their game progress and at least recover most of their efforts when their character is unfairly extinguished. The “save progress” function needs to be designed to return a resurrected player to the last level she completed successfully.

Just be careful how you go about saving the game state. It's pretty complicated, and I don't want anyone else with access to it mucking it up and breaking my code.

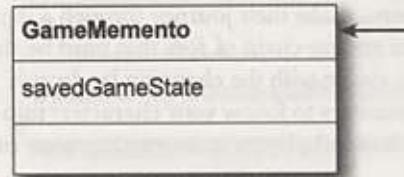


The Memento at work

The Memento has two goals:

- Saving the important state of a system's key object.
- Maintaining the key object's encapsulation.

Keeping the single responsibility principle in mind, it's also a good idea to keep the state that you're saving separate from the key object. This separate object that holds the state is known as the Memento object.



Client

```

// when new level is reached
Object saved =
    (Object) mgo.getCurrentState();

// when a restore is required
mgo.restoreState(saved);
  
```

MasterGameObject

```

gameState

Object getCurrentState() {
    // gather state
    return(gameState);
}

restoreState(Object savedState) {
    // restore state
}

// do other game stuff
  
```

While this isn't a terribly fancy implementation, notice that the Client has no access to the Memento's data.

Memento Benefits

- Keeping the saved state external from the key object helps to maintain cohesion.
- Keeps the key object's data encapsulated.
- Provides easy-to-implement recovery capability.

Memento Uses and Drawbacks

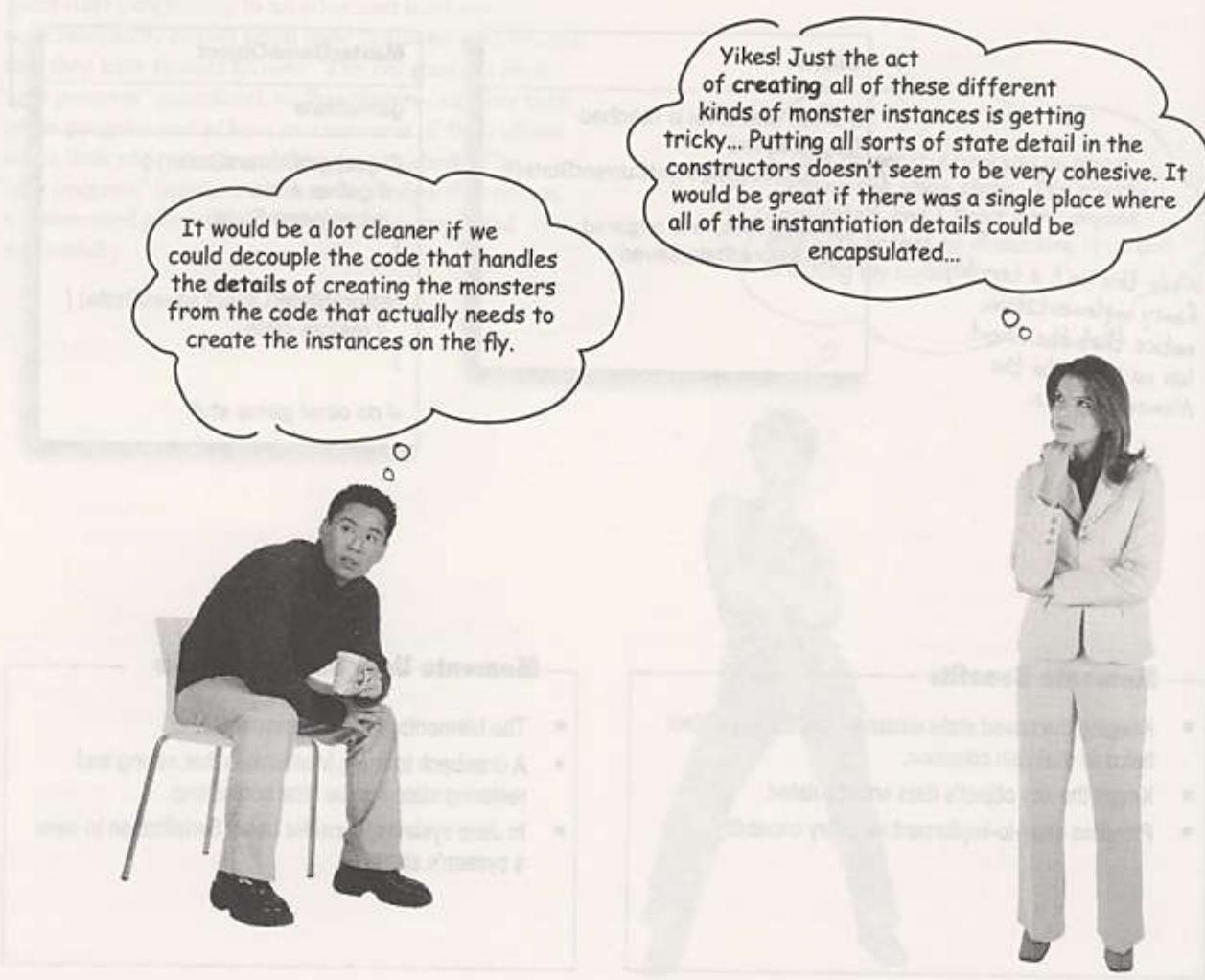
- The Memento is used to save state.
- A drawback to using Memento is that saving and restoring state can be time consuming.
- In Java systems, consider using Serialization to save a system's state.

Prototype

Use the Prototype Pattern when creating an instance of a given class is either expensive or complicated.

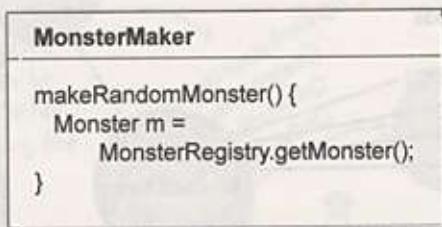
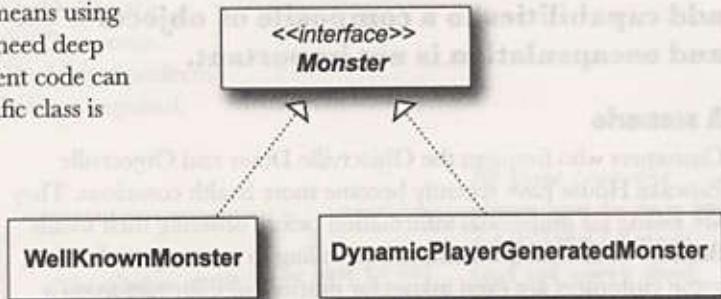
A scenario

Your interactive role playing game has an insatiable appetite for monsters. As your heroes make their journey through a dynamically created landscape, they encounter an endless chain of foes that must be subdued. You'd like the monster's characteristics to evolve with the changing landscape. It doesn't make a lot of sense for bird-like monsters to follow your characters into underseas realms. Finally, you'd like to allow advanced players to create their own custom monsters.

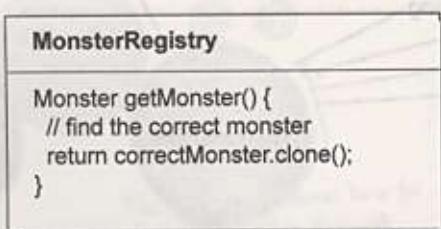


Prototype to the rescue

The Prototype Pattern allows you to make new instances by copying existing instances. (In Java this typically means using the `clone()` method, or de-serialization when you need deep copies.) A key aspect of this pattern is that the client code can make new instances without knowing which specific class is being instantiated.



The client needs a new monster appropriate to the current situation. (The client won't know what kind of monster he gets.)



The registry finds the appropriate monster, makes a clone of it, and returns the clone.

Prototype Benefits

- Hides the complexities of making new instances from the client.
- Provides the option for the client to generate objects whose type is not known.
- In some circumstances, copying an object can be more efficient than creating a new object.

Prototype Uses and Drawbacks

- Prototype should be considered when a system must create new objects of many types in a complex class hierarchy.
- A drawback to using the Prototype is that making a copy of an object can sometimes be complicated.

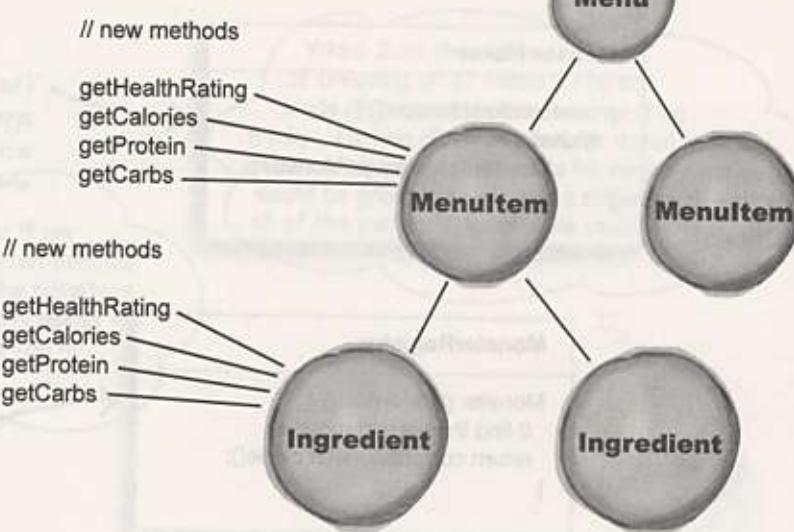
Visitor

Use the Visitor Pattern when you want to add capabilities to a composite of objects and encapsulation is not important.

A scenario

Customers who frequent the Objectville Diner and Objectville Pancake House have recently become more health conscious. They are asking for nutritional information before ordering their meals. Because both establishments are so willing to create special orders, some customers are even asking for nutritional information on a per ingredient basis.

Lou's proposed solution:

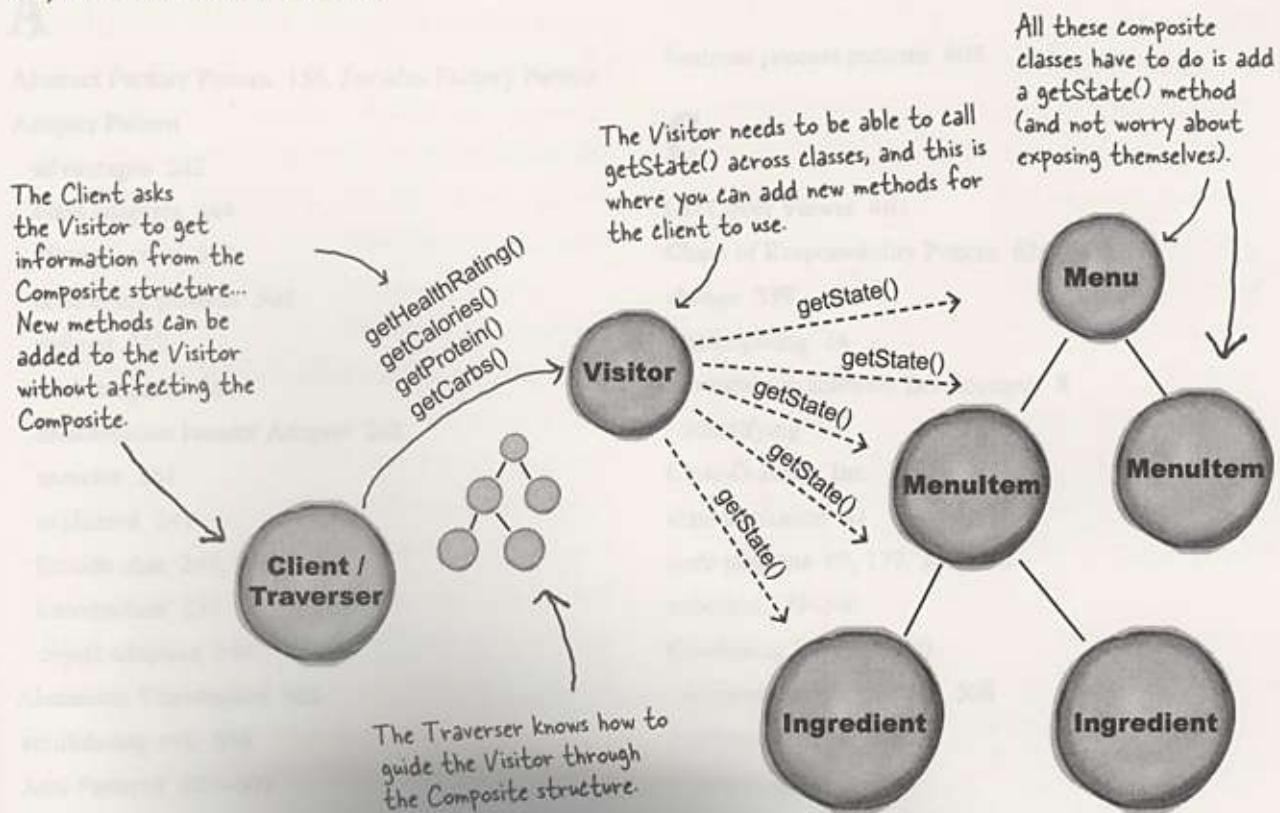


Mel's concerns...

"Boy, it seems like we're opening Pandora's box. Who knows what new method we're going to have to add next, and every time we add a new method we have to do it in two places. Plus, what if we want to enhance the base application with, say, a recipes class? Then we'll have to make these changes in three different places..."

The Visitor drops by

The Visitor must visit each element of the Composite; that functionality is in a Traverser object. The Visitor is guided by the Traverser to gather state from all of the objects in the Composite. Once state has been gathered, the Client can have the Visitor perform various operations on the state. When new functionality is required, only the Visitor must be enhanced.



Visitor Benefits

- Allows you to add operations to a Composite structure without changing the structure itself.
- Adding new operations is relatively easy.
- The code for operations performed by the Visitor is centralized.

Visitor Drawbacks

- The Composite classes' encapsulation is broken when the Visitor is used.
- Because the traversal function is involved, changes to the Composite structure are more difficult.

Index

A

Abstract Factory Pattern 156. *See also* Factory Pattern
Adapter Pattern
advantages 242
class adapters 244
class diagram 243
combining patterns 504
defined 243
duck magnets 245
Enumeration Iterator Adapter 248
exercise 251
explained 241
fireside chat 247, 252–253
introduction 237
object adapters 244
Alexander, Christopher 602
annihilating evil 606
Anti-Patterns 606–607
 Golden Hammer 607
application patterns 604
architectural patterns 604

B

Bridge Pattern 612–613
Builder Pattern 614–615
bullet points 32, 74, 105, 162, 186, 230, 270, 311, 380, 423, 491, 560, 608

business process patterns 605

C

CD Cover Viewer 463
Chain of Responsibility Pattern 616–617
change 339
 anticipating 14
constant in software development 8
identifying 53
Choc-O-Holic, Inc. 175
class explosion 81
code magnets 69, 179, 245, 350
cohesion 339–340
Combining Patterns 500
 Abstract Factory Pattern 508
 Adapter Pattern 504
 class diagram 524
 Composite Pattern 513
 Decorator Pattern 506
 Observer Pattern 516
Command Pattern
 class diagram 207
 command object 203
 defined 206–207
 introduction 196
 loading the Invoker 201

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

Simple Factory 114
Singleton Pattern 177
State Pattern 410
Strategy Pattern 24
Template Method Pattern 289
use 29
versus frameworks 29
versus libraries 29
Visitor Pattern 628–629
Design Principles. *See* Object Oriented Design Principles
Design Puzzle 25, 133, 279, 395, 468, 542
Design Toolbox 32, 74, 105, 162, 186, 230, 270, 311, 380, 423, 491, 560, 608
DJ View 534
domain specific patterns 604

E

Elvis 526
encapsulate what varies 8–9, 75, 136, 397, 612
encapsulating algorithms 286, 289
encapsulating behavior 11
encapsulating iteration 323
encapsulating method invocation 206
encapsulating object construction 614–615
encapsulating object creation 114, 136
encapsulating requests 206
encapsulating state 399

F

Facade Pattern
advantages 260
and Principle of Least Knowledge 269
class diagram 264

defined 264
introduction 258
Factory Method Pattern 134. *See also* Factory Pattern
Factory Pattern
Abstract Factory
and Factory Method 158–159, 160–161
class diagram 156–157
combining patterns 508
defined 156
interview 158–159
introduction 153
Factory Method
advantages 135
and Abstract Factory 160–161
class diagram 134
defined 134
interview 158–159
introduction 120, 131–132
up close 125
Simple Factory
defined 117
introduction 114
family of algorithms. *See* Strategy Pattern
family of products 145
favor composition over inheritance 23, 75
fireside chat 62, 247, 252, 308, 418, 472–473
Five minute drama 48, 478
Flyweight Pattern 618–619
forces 582
Friedman, Dan 171

G

Gamma, Erich 601

H-P

Gang of Four 583, 601

Gamma, Erich 601

Helm, Richard 601

Johnson, Ralph 601

Vlissides, John 601

global access point 177

gobble gobble 239

Golden Hammer 607

guide to better living with Design Patterns 578

Gumball Machine Monitor 431

H

HAS-A 23

Head First learning principles xxx

Helm, Richard 601

Hillside Group 603

Hollywood Principle, The 296

and the Dependency Inversion Principle 298

Home Automation or Bust, Inc. 192

Home Sweet Home Theater 255

Hot or Not 475

I

inheritance

disadvantages 5

for reuse 5–6

versus composition 93

interface 12

Interpreter Pattern 620–621

inversion 141–142

IS-A 23

Iterator Pattern

advantages 330

ABC bookish

ABC multibookish

and collections 347–349

and Composite Pattern 368

and Enumeration 338

and Hashtable 343, 348

class diagram 337

code magnets 350

defined 336

exercise 327

external iterator 338

for/in 349

internal iterator 338

introduction 325

java.util.Iterator 332

Null Iterator 372

polymorphic iteration 338

removing objects 332

J

Johnson, Ralph 601

K

KISS 594

L

Law of Demeter. *See* Principle of Least Knowledge

lazy instantiation 177

loose coupling 53

M

magic bullet 594

master and student 23, 30, 85, 136, 592, 596

Matchmaking in Objectville 475

Mediator Pattern 622–623

Memento Pattern 624–625

middleman 237

Mighty Gumball, Inc. 386

Model-View-Controller

Adapter Pattern 546

and design patterns 532

and the Web 549

Composite Pattern 532, 559

introduction 529

Mediator Pattern 559

Observer Pattern 532

ready-bake code 564–576

song 526

Strategy Pattern 532, 545

up close 530

Model 2 549. *See also* Model-View-Controller

and design patterns 557–558

MVC. *See* Model-View-Controller

N

Null Object 214, 372

O

Objectville Diner 26, 197, 316, 628

Objectville Pancake House 316, 628

Object Oriented Design Principles 9, 30–31

Dependency Inversion Principle 139–143

encapsulate what varies 9, 111

favor composition over inheritance 23, 243, 397

Hollywood Principle 296

one class, one responsibility 185, 336, 339, 367

Open-Closed Principle 86–87, 407

Principle of Least Knowledge 265

program to an interface, not an implementation 11, 243, 335

strive for loosely coupled designs between objects that interact 53

Observable 64, 71

Observer Pattern

class diagram 52

code magnets 69

combining patterns 516

cubicle conversation 55

defined 51–52

fireside chat 62

Five minute drama 48

introduction 44

in Swing 72–73

Java support 64

pull 63

push 63

one-to-many relationship 51–52

OOPSLA 603

Open-Closed Principle 86–87

oreo cookie 526

organizational patterns 605

P

part-whole hierarchy 356. *See also* Composite Pattern

patterns catalog 581, 583, 585

Patterns Exposed 104, 158, 174, 377–378

patterns in the wild 299, 488–489

patterns zoo 604

Pattern Honorable Mention 117, 214

Pizza shop 112

Portland Patterns Repository 603

Q-Y

Principle of Least Knowledge 265–268
disadvantages 267
program to an implementation 12, 17, 71
program to an interface 12
program to an interface, not an implementation 11, 75
Prototype Pattern 626–627
Proxy Pattern
and Adapter Pattern 471
and Decorator Pattern 471, 472–473
Caching Proxy 471
class diagram 461
defined 460
Dynamic Proxy 474, 479, 486
and RMI 486
exercise 482
fireside chat 472–473
`java.lang.reflect.Proxy` 474
Protection Proxy 474, 477
Proxy Zoo 488–489
ready-bake code 494
Remote Proxy 434
variants 471
Virtual Proxy 462
image proxy 464
publisher/subscriber 45

Q

Quality, The. *See* Quality without a name
Quality without a name. *See* Quality, The

R

refactoring 354, 595
remote control 193, 209

Remote Method Invocation. *See also* RMI
remote proxy 434. *See also* Proxy Pattern
reuse 13, 23, 85

RMI 436

S

shared vocabulary 26–28, 599–600
sharpen your pencil 5, 42, 54, 61, 94, 97, 99, 124, 137, 148, 176, 183, 205, 225, 242, 268, 284, 322, 342, 396, 400, 406, 409, 421, 483, 511, 518, 520, 589
Simple Factory 117
SimUDuck 2, 500
Singleton Pattern
advantages 170, 184
and garbage collection 184
and global variables 185
and multithreading 180–182
class diagram 177
defined 177
disadvantages 184
double-checked locking 182
interview 174
up close 173

Single Responsibility Principle 339. *See also* Object Oriented Design Principles: one class, one responsibility

skeleton 440

Starbuzz Coffee 80, 276

state machines 388–389

State Pattern

and Strategy Pattern 411, 418–419
class diagram 410
defined 410

disadvantages 412, 417

introduction 398

sharing state 412

static factory 115

Strategy Pattern 24

and State Pattern 411, 418–419

and Template Method Pattern 308–309

encapsulating behavior 22

family of algorithms 22

fireside chat 308

stub 440

T

Template Method Pattern

advantages 288

and Applet 307

and java.util.Arrays 300

and Strategy Pattern 305, 308–309

and Swing 306

and the Hollywood Principle 297

class diagram 289

defined 289

fireside chat 308–309

hook 292, 295

introduction 286

up close 290–291

The Little Lisper 171

thinking in patterns 594–595

tightly coupled 53

U

undo 216, 227

user interface design patterns 605

V

varies. *See* encapsulate what varies

Visitor Pattern 628–629

Vlissides, John 601

W

Weather-O-Rama 38

when not to use patterns 596–598

Who Does What? 202, 254, 298, 379, 422, 487, 588

Why a duck? 500

wrapping objects 88, 242, 252, 260, 473, 508. *See also* Adapter Pattern, Decorator Pattern, Facade Pattern, Proxy Pattern

Y

your mind on patterns 597

Colophon



All interior layouts were designed by Eric Freeman, Elisabeth Freeman, Kathy Sierra and Bert Bates. Kathy and Bert created the look & feel of the Head First series. The book was produced using Adobe InDesign CS (an unbelievably cool design tool that we can't get enough of) and Adobe Photoshop CS. The book was typeset using Uncle Stinky, Mister Frisky (you think we're kidding), Ann Satellite, Baskerville, Comic Sans, Myriad Pro, Skippy Sharp, Savoye LET, Jokerman LET, Courier New and Woodrow typefaces.

Interior design and production all happened exclusively on Apple Macintoshes—at Head First we're all about "Think Different" (even if it isn't grammatical). All Java code was created using James Gosling's favorite IDE, *vi*, although we really should try Erich Gamma's Eclipse.

Long days of writing were powered by the caffeine fuel of Honest Tea and Tejava, the clean Santa Fe air, and the grooving sounds of Banco de Gaia, Cocteau Twins, Buddha Bar I-VI, Delerium, Enigma, Mike Oldfield, Olive, Orb, Orbital, LTJ Bukem, Massive Attack, Steve Roach, Sasha and Digweed, Thievery Corporation, Zero 7 and Neil Finn (in all his incarnations) along with a heck of a lot of acid trance and more 80s music that you'd care to know about.

Head First
Institute

And now, a final word from the Head First Institute...

Our world class researchers are working day and night in a mad race to uncover the mysteries of Life, the Universe and Everything—before it's too late.

Never before has a research team with such noble and daunting goals been assembled. Currently, we are focusing our collective energy and brain power on creating the ultimate learning machine. Once perfected, you and others will join us in our quest!

You're fortunate to be holding one of our first prototypes in your hands. But only through constant refinement can our goal be achieved. We ask you, a pioneer user of the technology, to send us periodic field reports of your progress, at fieldreports@wickedlysmart.com

And next time you're in Objectville,
drop by and take one of our behind
the scenes laboratory tours.



Head First Design Patterns

"I received the book yesterday and started to read it... and I couldn't stop. This is très "cool." It is fun, but they cover a lot of ground and they are right to the point. I'm really impressed."

—Erich Gamma,
IBM Distinguished Engineer, and
coauthor of *Design Patterns*

"I feel like a thousand pounds of books have just been lifted off of my head."

—Ward Cunningham,
inventor of the Wiki and
founder of the Hillside Group

"This book is close to perfect, because of the way it combines expertise and readability. It speaks with authority and it reads beautifully."

—David Gelernter, Professor of Computer Science, Yale University

"One of the funniest and smartest books on software design I've ever read."

—Aaron LaBerge,
VP Technology, ESPN.com

Software Development/Java

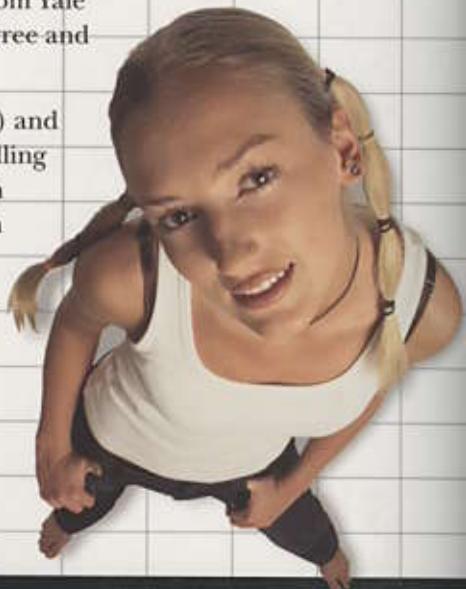
You know you don't want to reinvent the wheel (or worse, a flat tire), so you look to design patterns—the lessons learned by those who've faced the same software design problems. With design patterns, you get to take advantage of the best practices and experience of others, so that you can spend your time on...something else. Something more challenging. Something more complex. Something more *fun*. You want to learn:

- The patterns that *matter*
- When to use them, and *why*
- How to *apply* them to your own designs, *right now*
- When *not* to use them (how to avoid pattern fever)
- OO design principles on which patterns are based

Most importantly, you want to learn design patterns in a way that won't put you to sleep. If you've read a Head First book, you know what to expect—a visually rich format designed for the way your brain works. Using the latest research in neurobiology, cognitive science, and learning theory, *Head First Design Patterns* will load patterns into your brain in a way that sticks. In a way that makes you better at solving software design problems, and better at speaking the language of patterns with others on your team.

Eric Freeman and Elisabeth Freeman are authors, educators, and technology innovators. After four years leading digital media and Internet efforts at the Walt Disney Company, they're applying some of that pixie dust to their own media, including this book. Eric and Elisabeth both hold computer science degrees from Yale University: Elisabeth holds an M.S. degree and Eric a Ph.D.

Kathy Sierra (founder of javaranch.com) and Bert Bates are the creators of the best-selling Head First series and developers of Sun Microsystems Java developer certification exams.



Visit O'Reilly on the Web at www.oreilly.com
In Canada

ISBN 0-596-00712-4

US \$44.95

9 780596 007126

\$ 69 95
6 36920"00712" 8

O'REILLY®