

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.

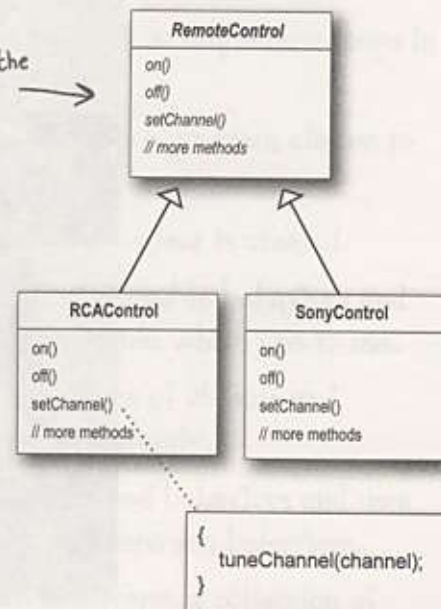
A scenario

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.

This is an abstraction. It could be an interface or an abstract class.

Every remote has the same abstraction.

Lots of implementations, one for each 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.

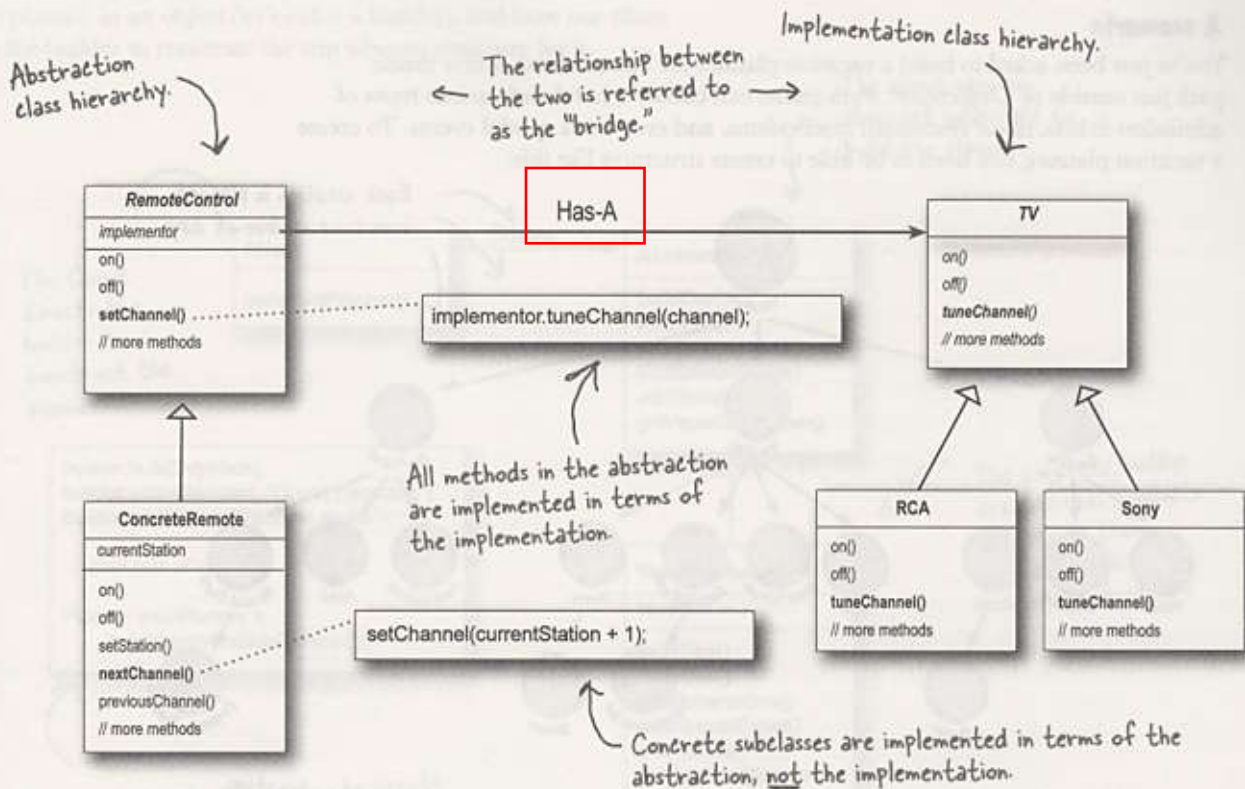
So your dilemma is that the remotes are going to change and the TVs are going to change. You've already *abstracted* the user interface so that you can vary the *implementation* over the many TVs your customers will own. But you are also going to need to vary the abstraction because it is going to change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

Using this design we can vary only the TV implementation, not the user interface.

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

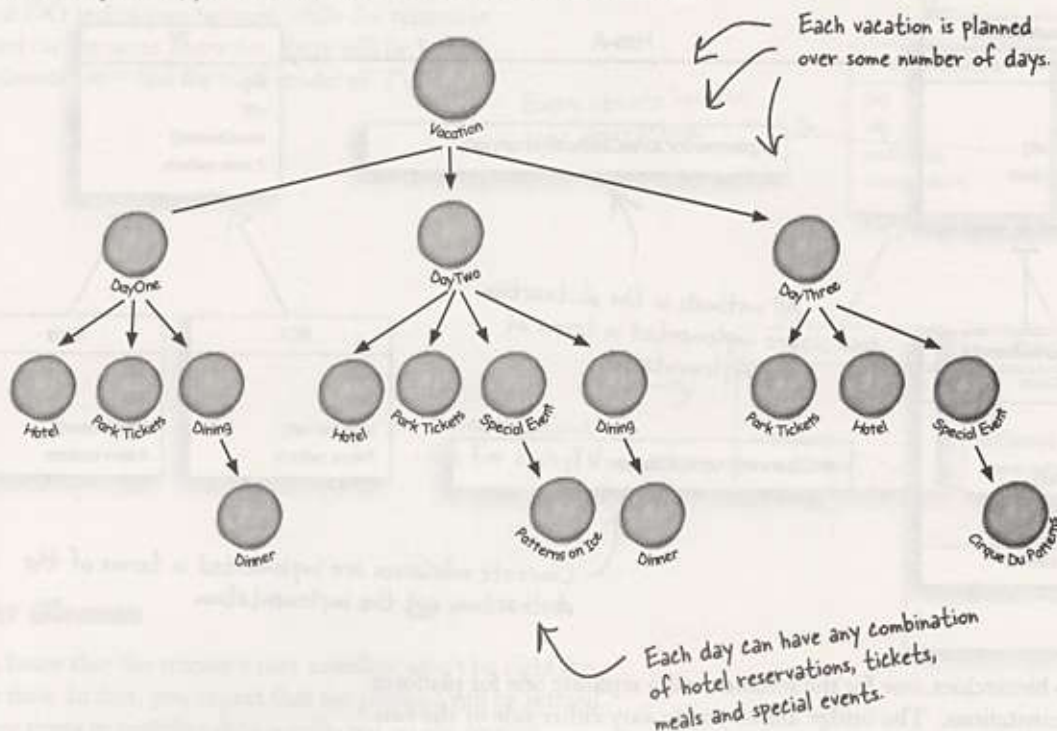
- Useful in graphic and windowing systems that need to run over multiple platforms.
- Useful any time you need to vary an interface and an implementation in different ways.
- Increases complexity.

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:



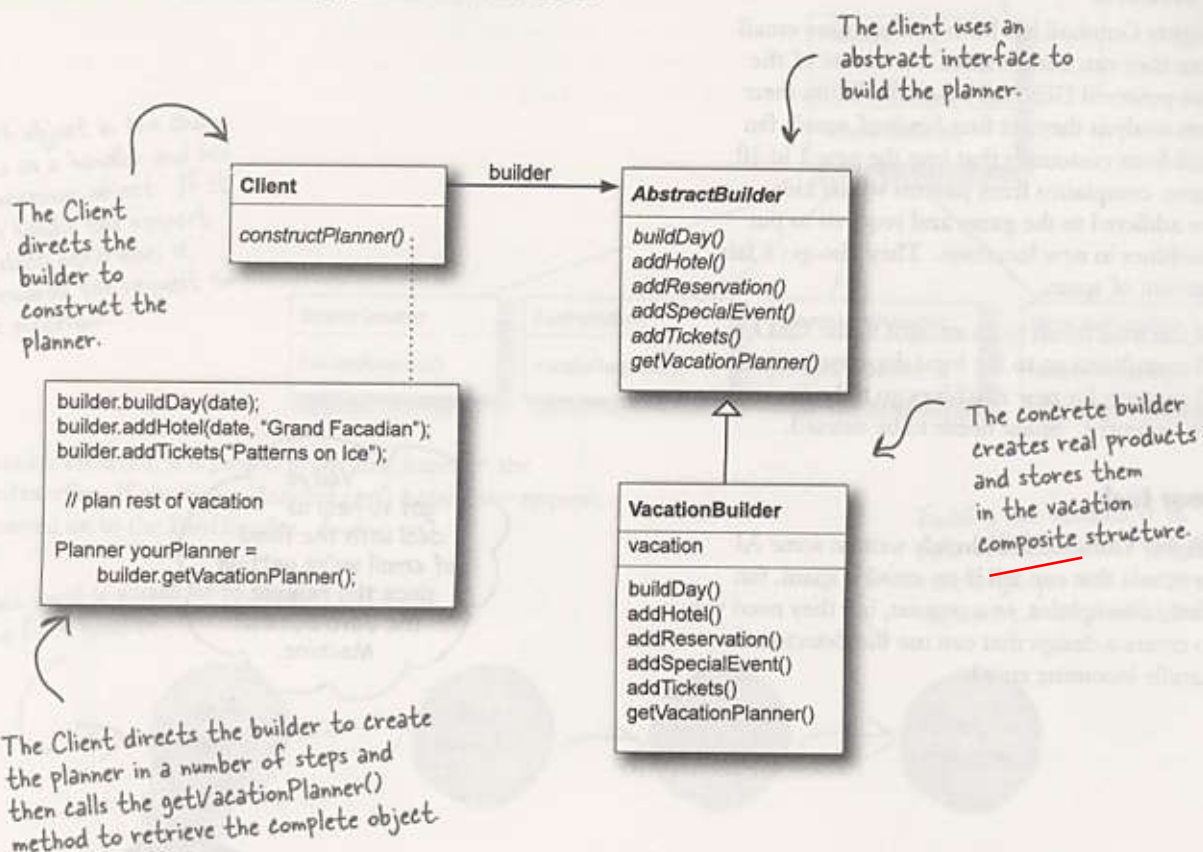
You need a flexible design

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.

both, factory method, and abstract factory are single step construction of object

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.

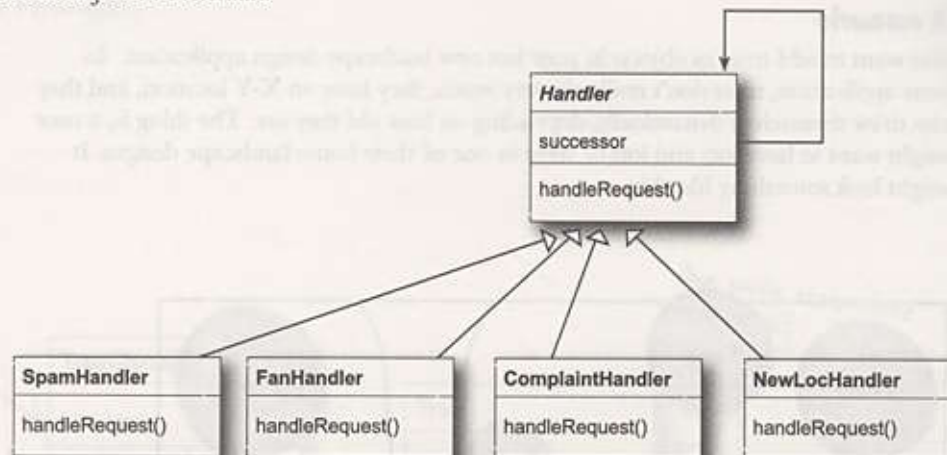


You've got to help us deal with the flood of email we're getting since the release of the Java Gumball Machine.

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...

Each email is passed to the first handler.

Email is not handled if it falls off the end of the chain - although, you can always implement a catch-all handler.



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

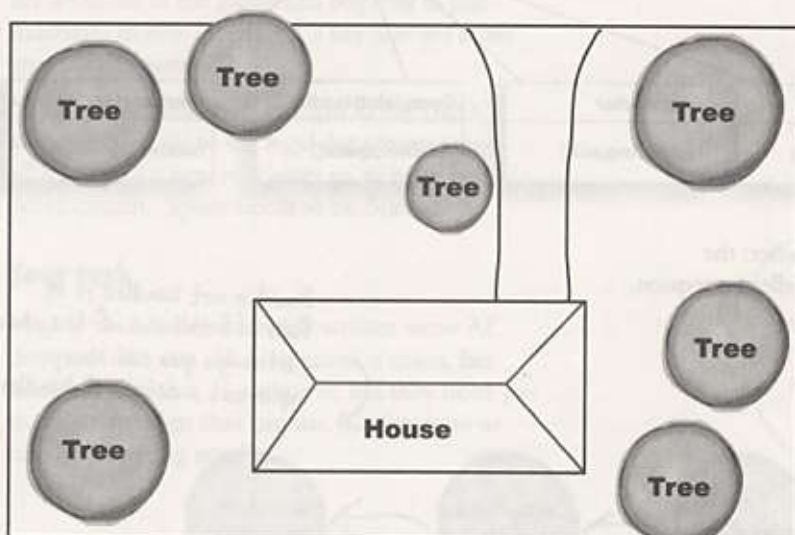
- Commonly used in windows systems to handle events like mouse clicks and keyboard events.
- Execution of the request isn't guaranteed; it may fall off the end of the chain if no object handles it (this can be an advantage or a disadvantage).
- Can be hard to observe the runtime characteristics and debug.

Flyweight

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 }

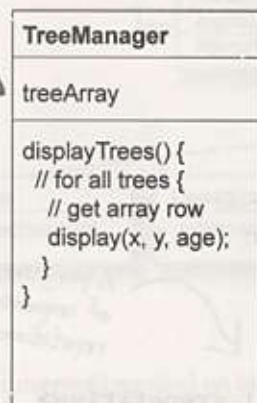
Your big client's dilemma

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...

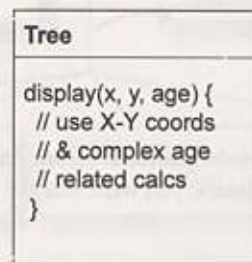
Why use the Flyweight Pattern?

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.

might now used at all.

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;`
`while (daylight) fly;`
`quack;`

← Turn the duck right
 ← Fly all day...
 ← ...and then quack.

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?

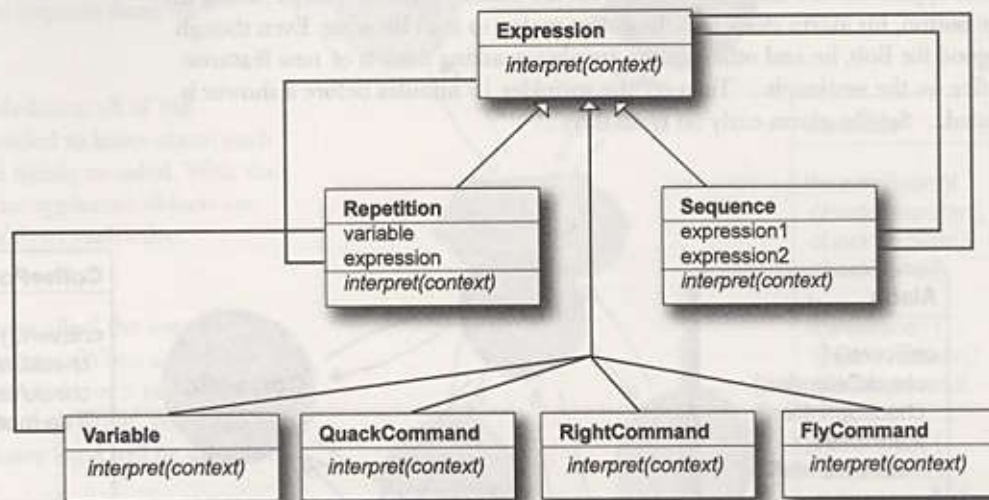
You've got a grammar; now all you need is a way to represent and interpret sentences in the grammar so that the students can see the effects of their programming on the simulated ducks.



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.

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

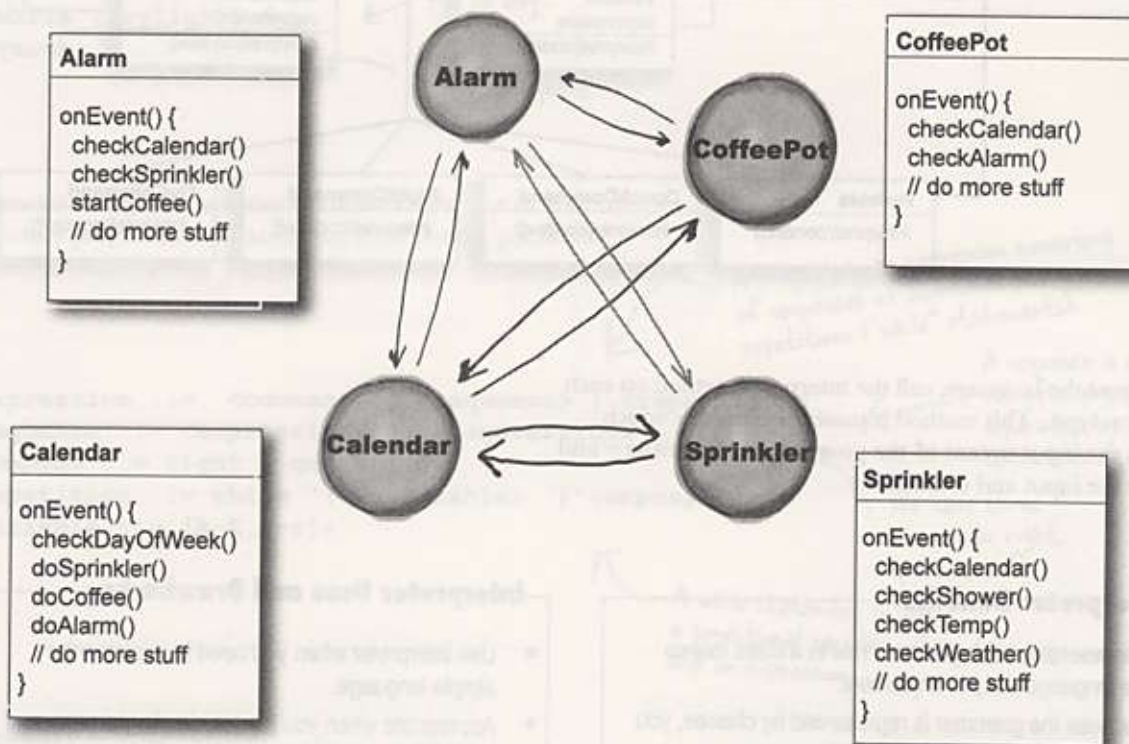
- Use interpreter when you need to implement a simple language.
- Appropriate when you have a simple grammar and simplicity is more important than efficiency.
- Used for scripting and programming languages.
- This pattern can become cumbersome when the number of grammar rules is large. In these cases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

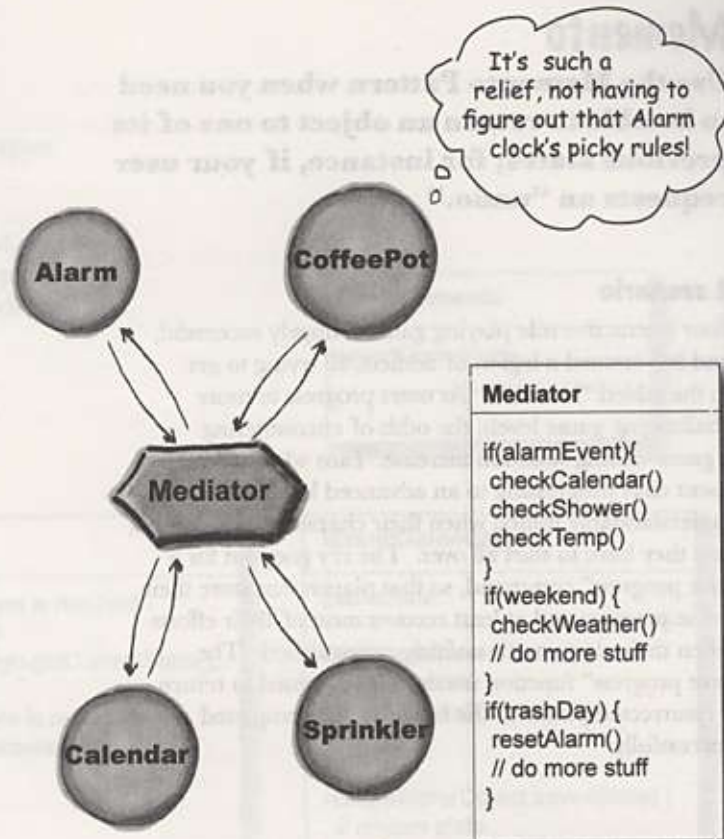
Mediator in action...

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

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.



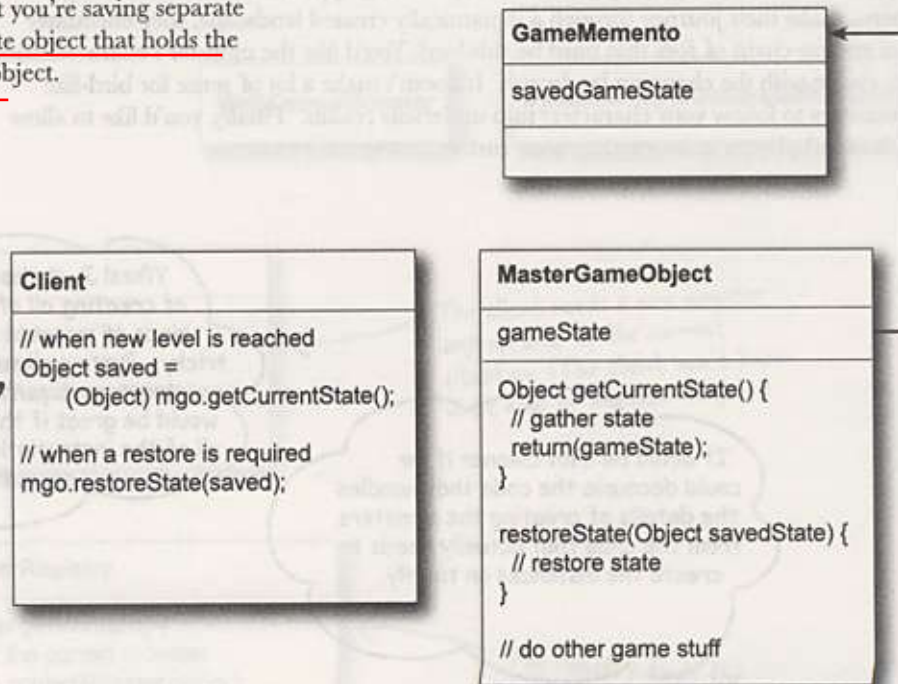
The Memento at work

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.

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

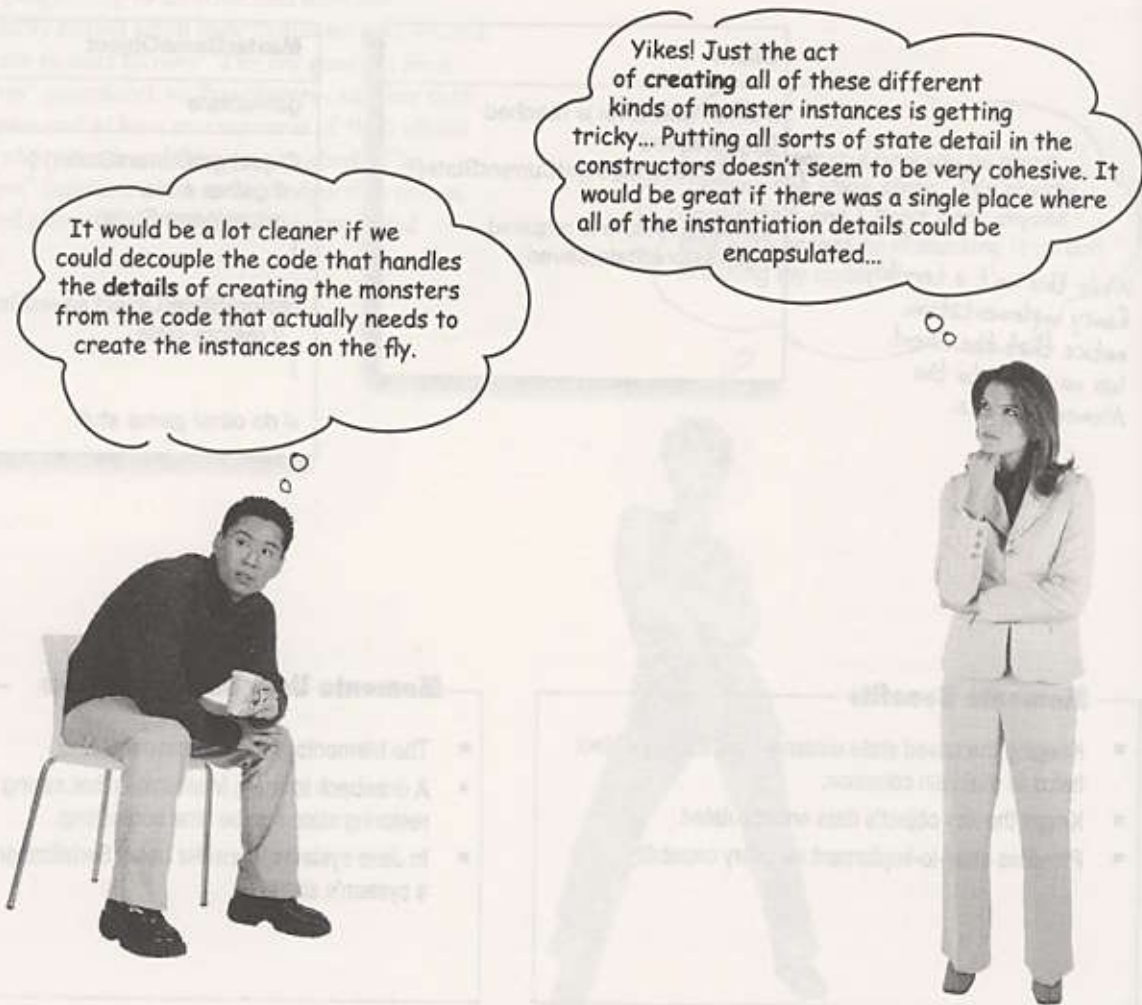
- The Memento is used to save state.
- A drawback to using Memento is that saving and restoring state can be time consuming.
- In Java systems, consider using Serialization to save a system's state.

Prototype

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

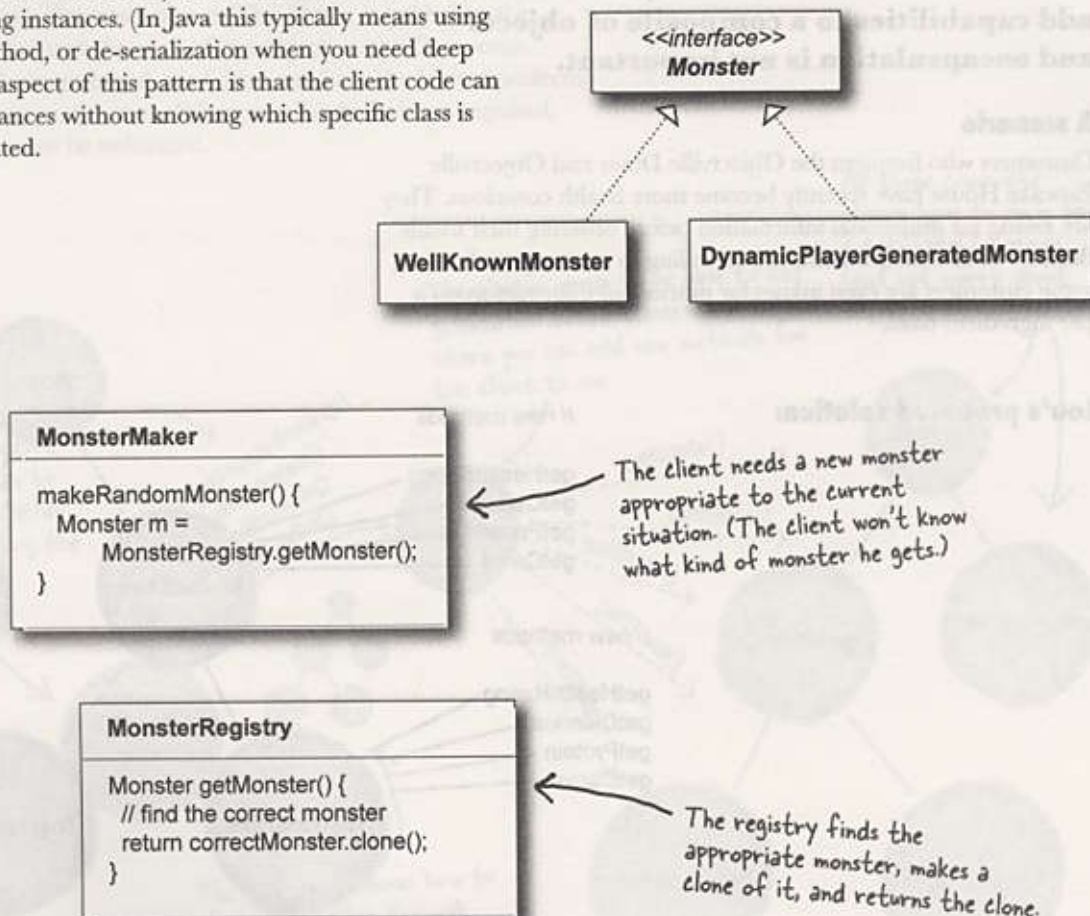
A scenario

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.



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.

since clone()
method available
from Object

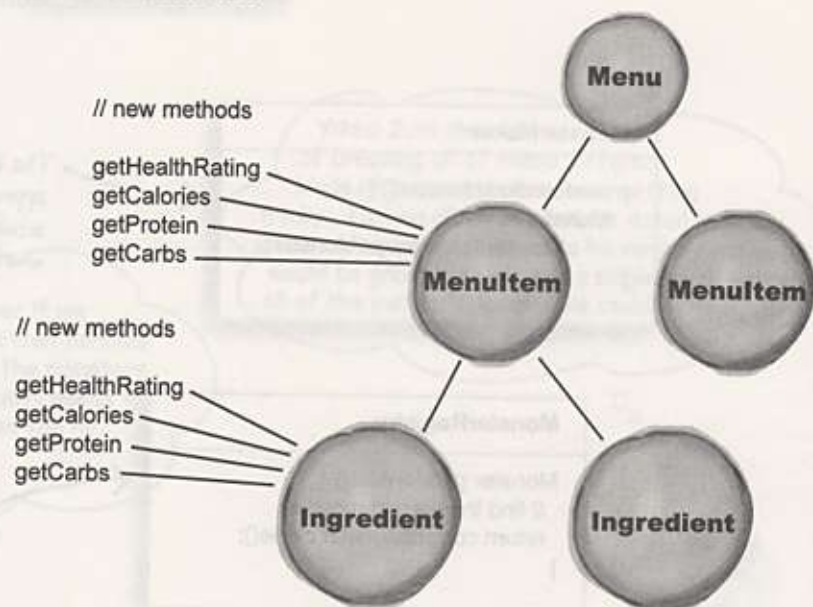
Visitor

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:



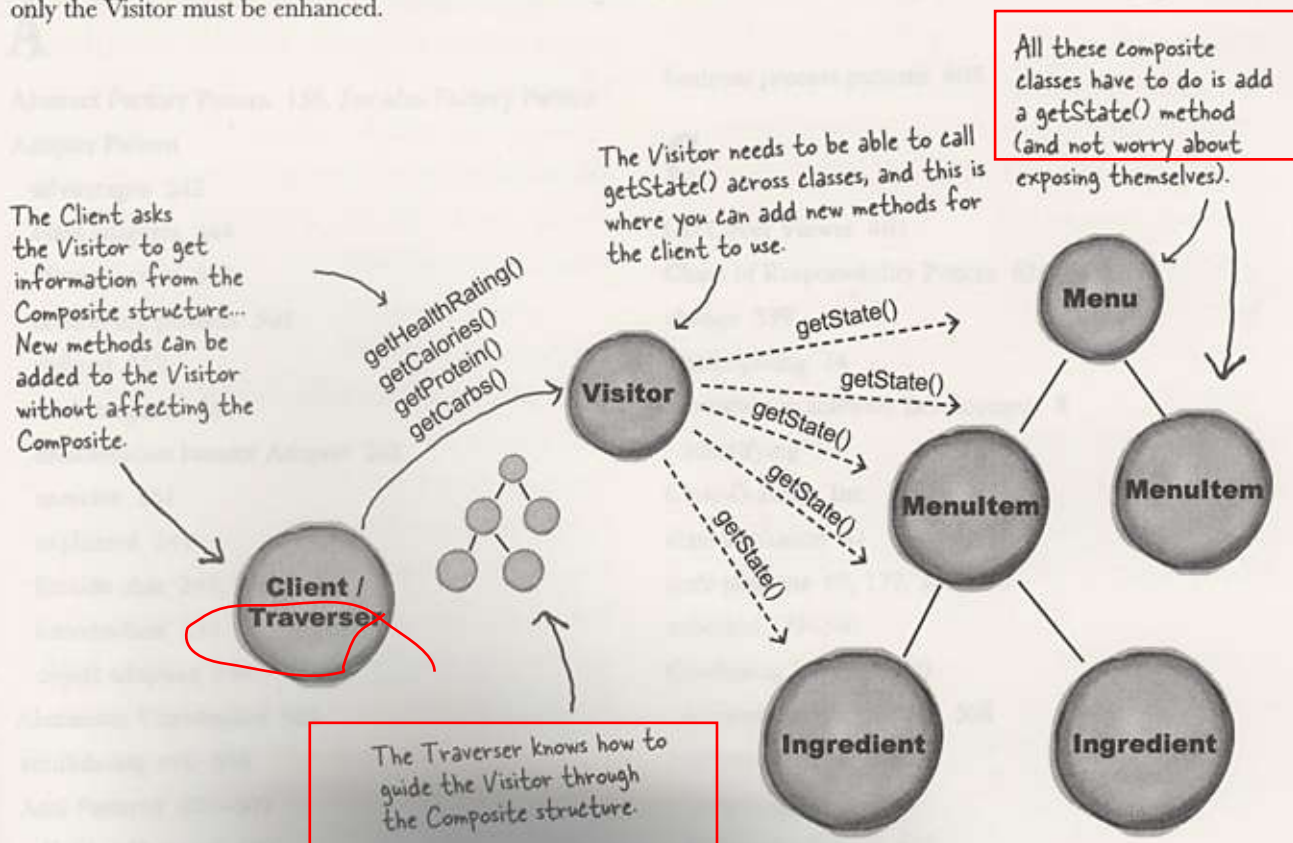
Mel's concerns...

"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..."

i hope, traversal API of DOM of JAXP might follow the visitor pattern

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.



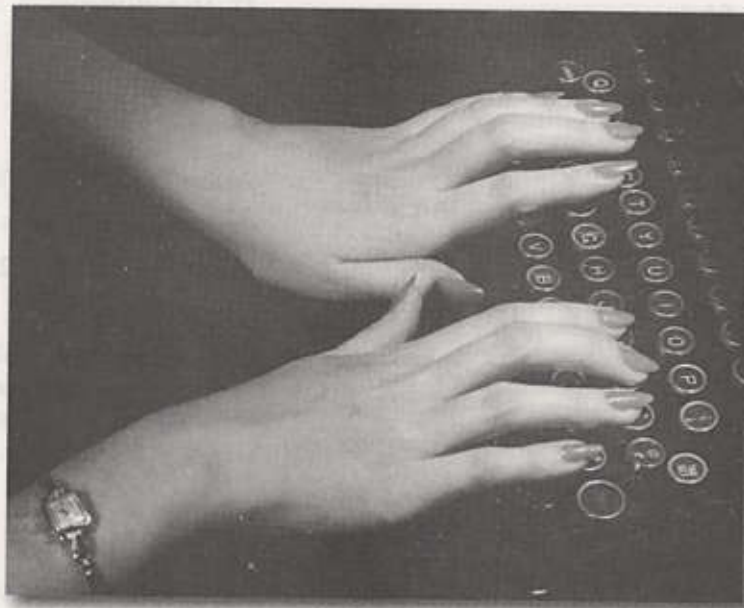
Visitor Benefits

- 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.

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.

Better than

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

Software Development/Java

"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

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

ISBN 0-596-00712-4

US \$44.95

In Canada



9 780596 007126



6 36920 007121 8

\$ 69 95

O'REILLY®

