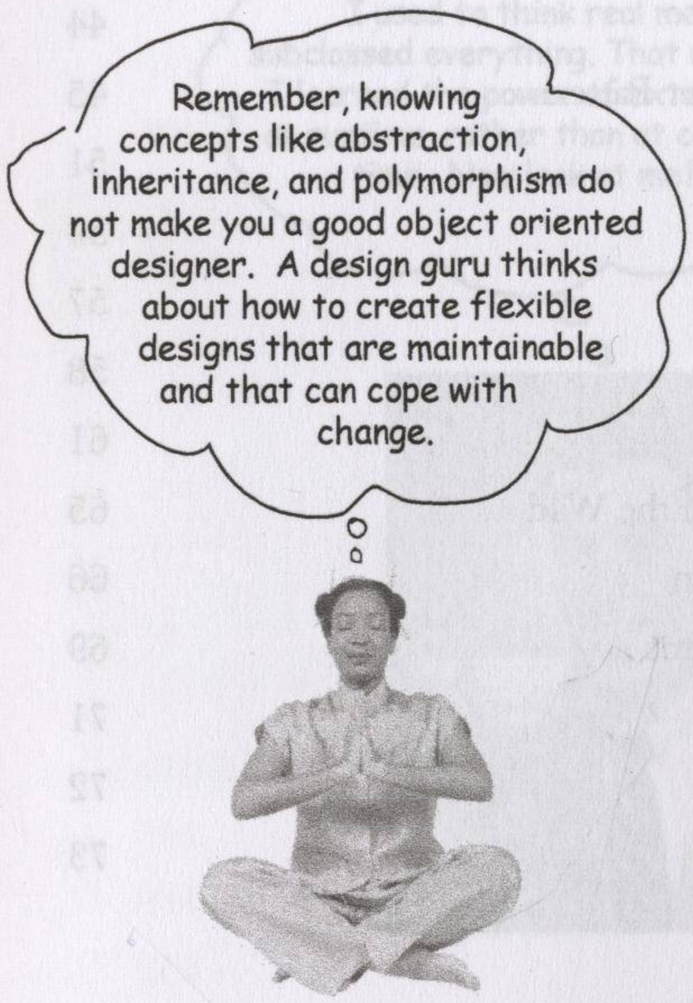
# Table of Contents (summary)



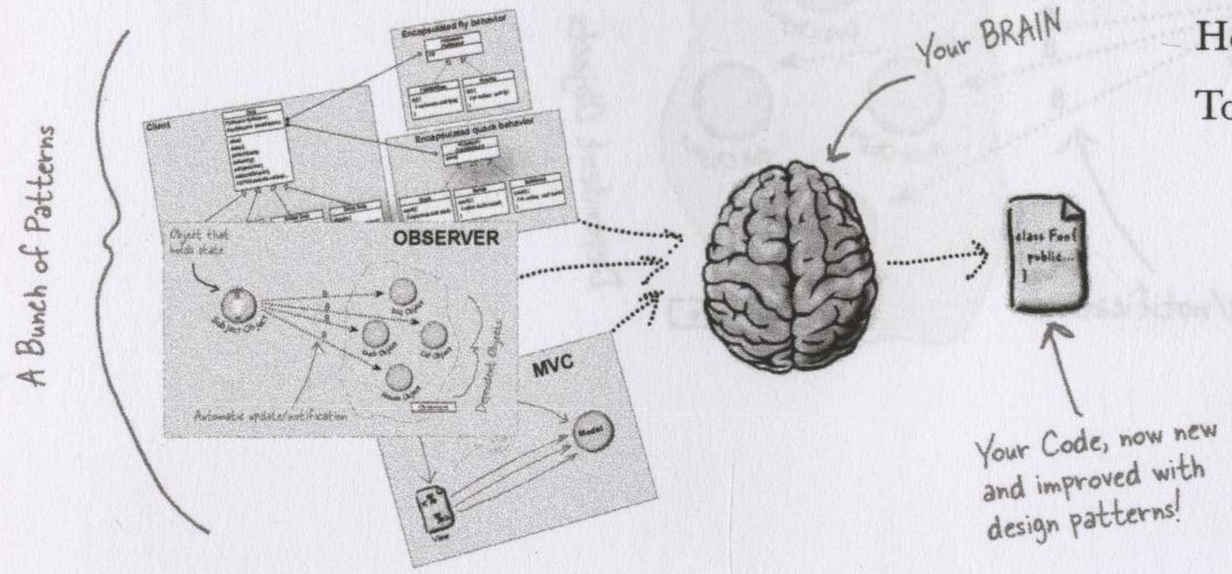
# Table of Contents (the real thing)

# 1 Welcome to Design Patterns

**Someone has already solved your problems.** In this chapter, you'll learn why (and how) you can exploit the wisdom and lessons learned by other developers who've been down the same design problem road and survived the trip. Before we're done, we'll look at the use and benefits of design patterns, look at some key object-oriented (OO) design principles, and walk through an example of how one pattern works. The best way to use patterns is to *load your brain* with them and then *recognize places* in your designs and existing applications where you can *apply them*. Instead of *code* reuse, with patterns you get *experience* reuse.



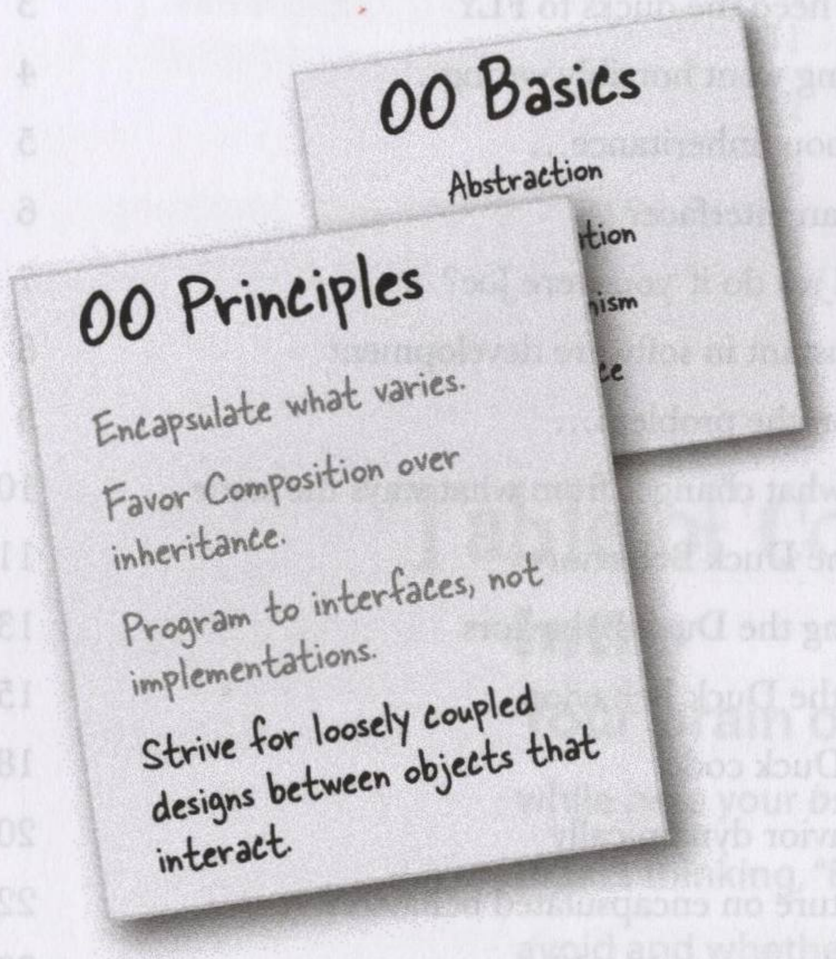


# 2 the Observer Pattern

## Keeping your Objects in the Know

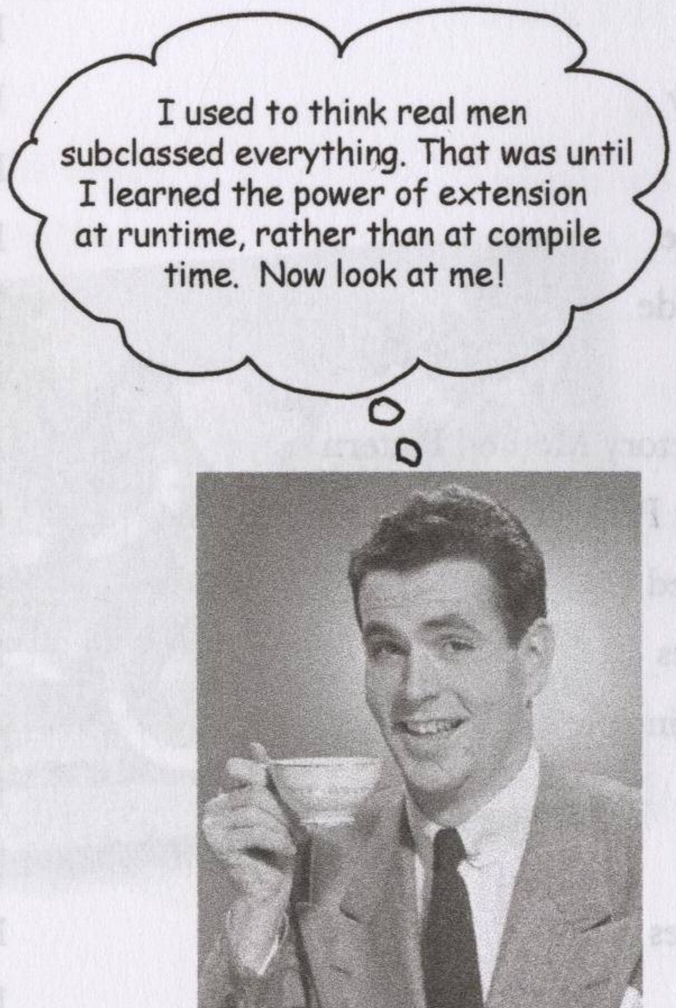
**You don't want to miss out when something interesting happens, do you?** We've got a pattern that keeps your objects *in the know* when something they *care about* happens. It's the Observer Pattern. It is one of the most commonly used design patterns, and it's incredibly useful. We're going to look at all kinds of interesting aspects of Observer, like its *one-to-many relationships* and *loose coupling*. And, with those concepts in mind, how can you help but be the life of the Patterns Party?



the Decorator Pattern

# 3 Decorating Objects

**Just call this chapter "Design Eye for the Inheritance Guy."** We'll re-examine the typical overuse of inheritance and you'll learn how to decorate your classes at runtime using a form of object composition. Why? Once you know the techniques of decorating, you'll be able to give your (or someone else's) objects new responsibilities *without making any code changes to the underlying classes.*

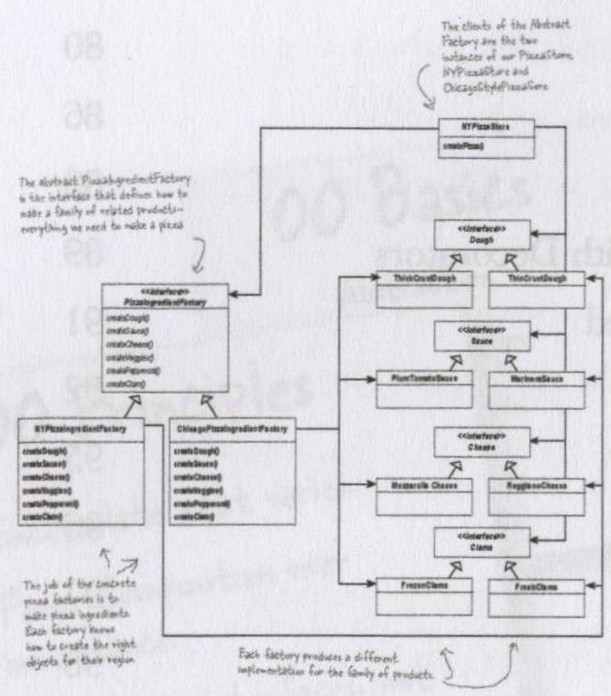


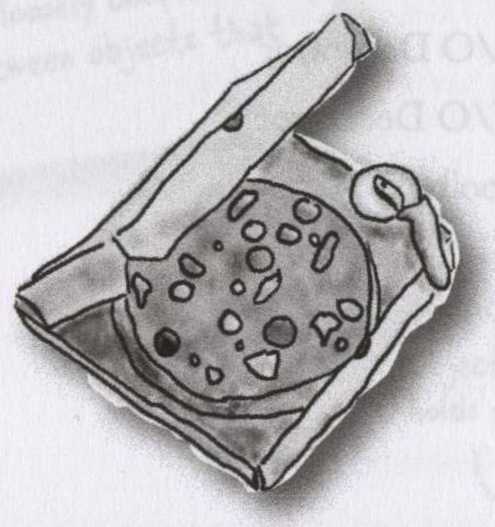
# 4 the Factory Pattern

## Baking with OO Goodness

**Get ready to bake some loosely coupled OO designs.**

There is more to making objects than just using the **new** operator. You'll learn that instantiation is an activity that shouldn't always be done in public and can often lead to *coupling problems*. And we don't want *that*, do we? Find out how Factory Patterns can help save you from embarrassing dependencies.

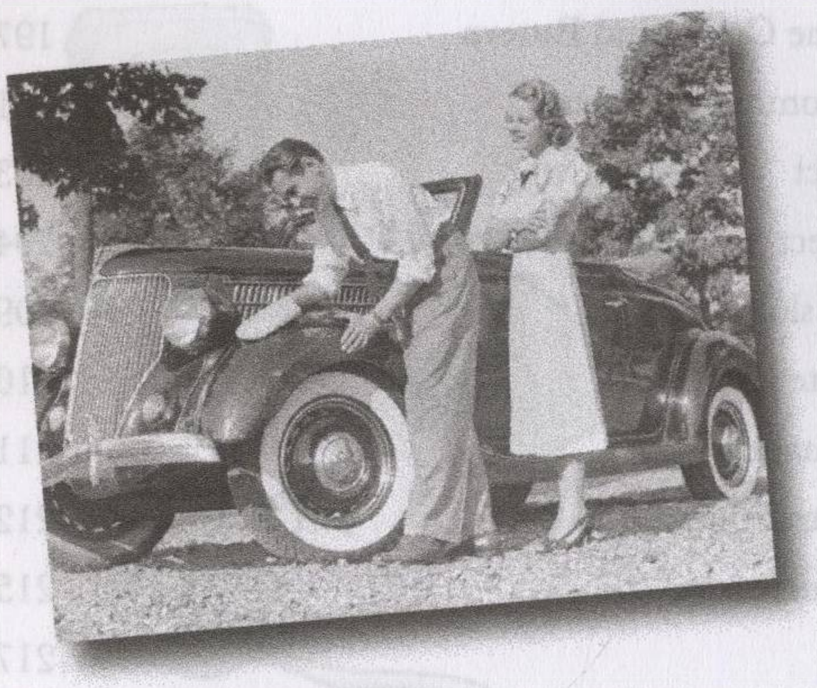


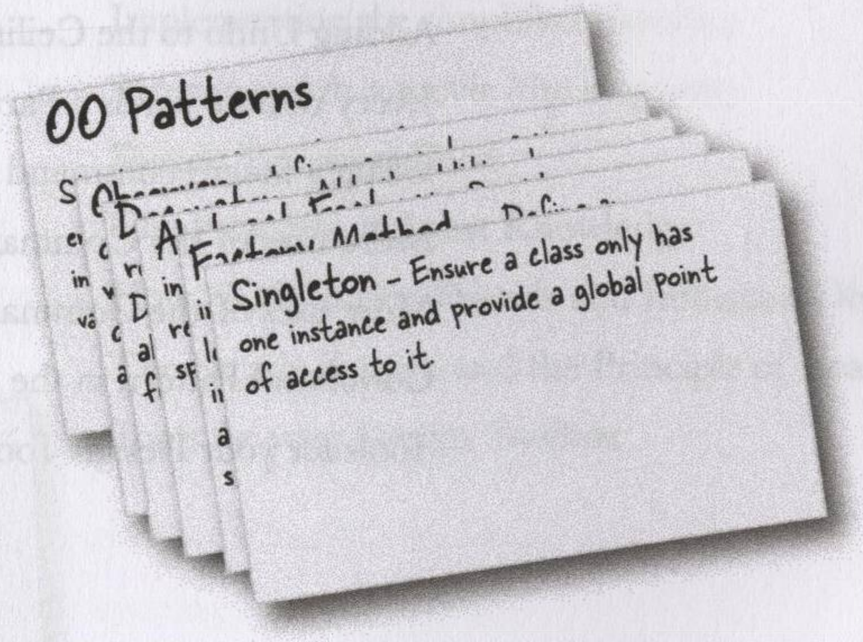
the Singleton Pattern

# 5 One-of-a-Kind Objects

**Our next stop is the Singleton Pattern, our ticket to creating one-of-a-kind objects for which there is only one instance, ever.** You might be happy to know that of all patterns, the Singleton is the simplest in terms of its class diagram; in fact, the diagram holds just a single class! But don't get too comfortable; despite its simplicity from a class design perspective, it's going to require some deep object-oriented thinking in its implementation. So put on that thinking cap, and let's get going.





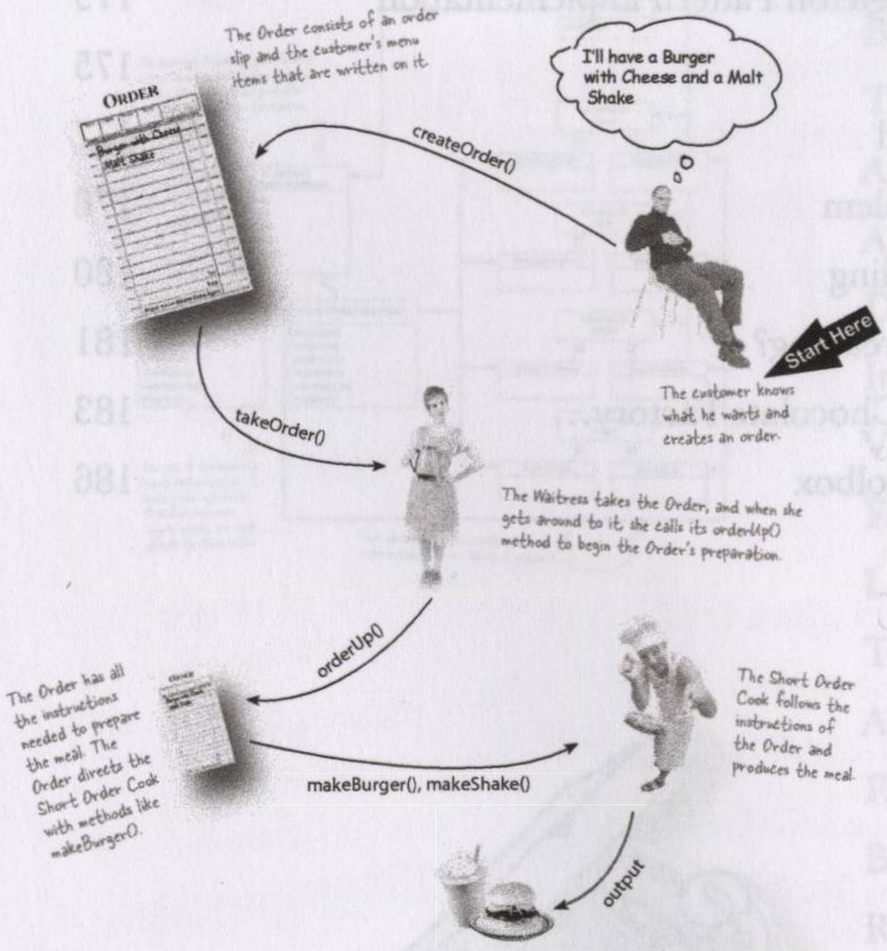


# the Command Pattern

## 6 Encapsulating Invocation

**In this chapter, we take encapsulation to a whole new level: we're going to encapsulate method invocation.**

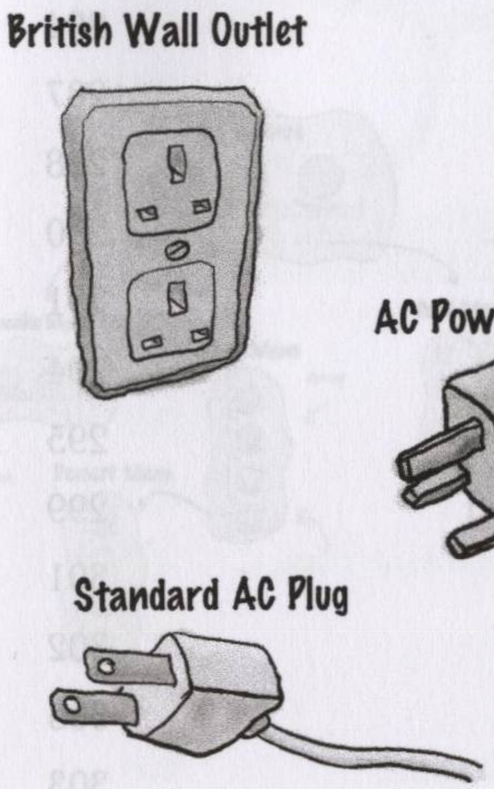
That's right—by encapsulating method invocation, we can crystallize pieces of computation so that the object invoking the computation doesn't need to worry about how to do things, it just uses our crystallized method to get it done. We can also do some wickedly smart things with these encapsulated method invocations, like save them away for logging or reuse them to implement undo functionality in our code.



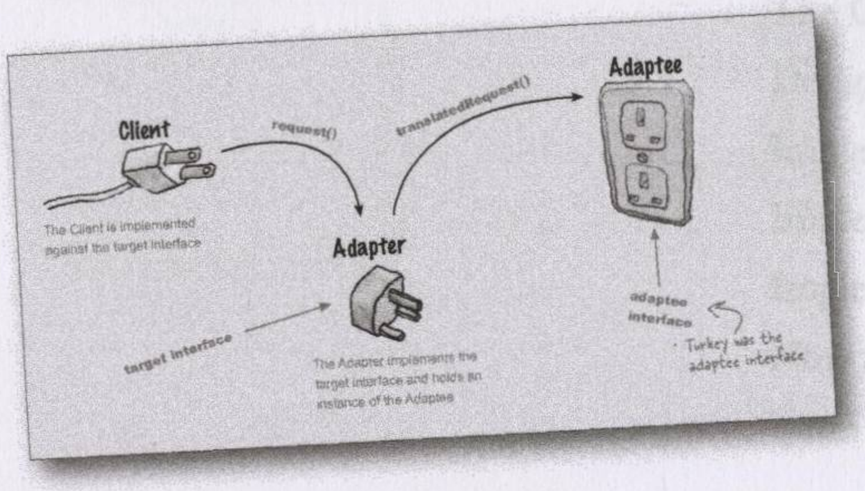
# 7 the Adapter and Facade Patterns

## Being Adaptive

**In this chapter we're going to attempt such impossible feats as putting a square peg in a round hole.** Sound impossible? Not when we have Design Patterns. Remember the Decorator Pattern? We **wrapped objects** to give them new responsibilities. Now we're going to wrap some objects with a different purpose: to make their interfaces look like something they're not. Why would we do that? So we can adapt a design expecting one interface to a class that implements a different interface. That's not all; while we're at it, we're going to look at another pattern that wraps objects to simplify their interface.





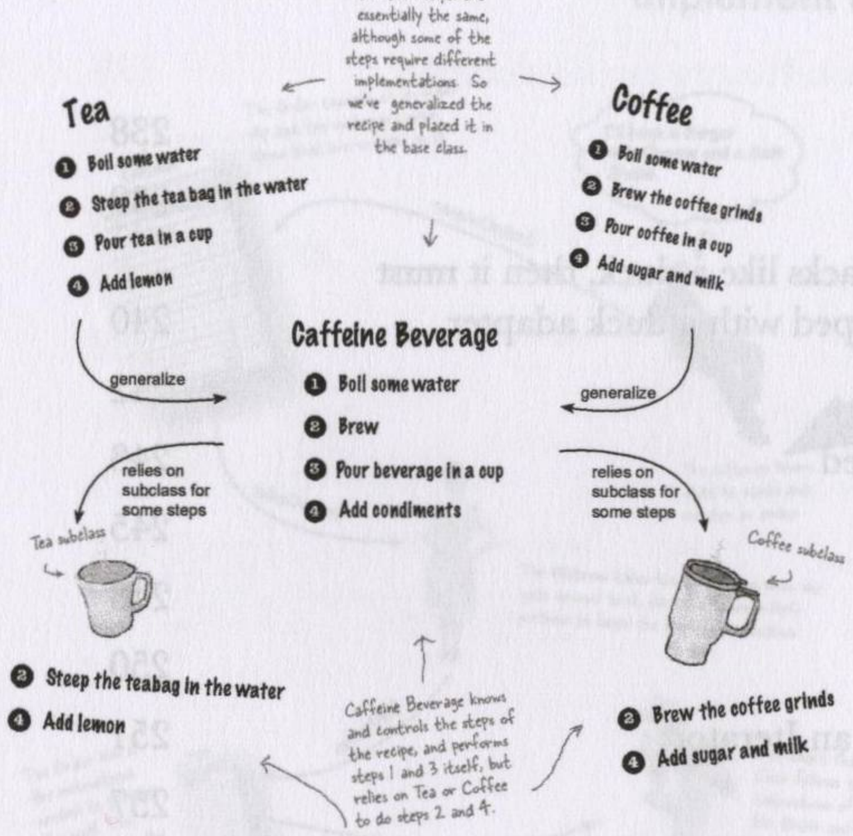



# the Template Method Pattern

# 8 Encapsulating Algorithms

**We've encapsulated object creation, method invocation, complex interfaces, ducks, pizzas...what could be next?**

We're going to get down to encapsulating pieces of algorithms so that subclasses can hook themselves right into a computation anytime they want. We're even going to learn about a design principle inspired by Hollywood. Let's get started...

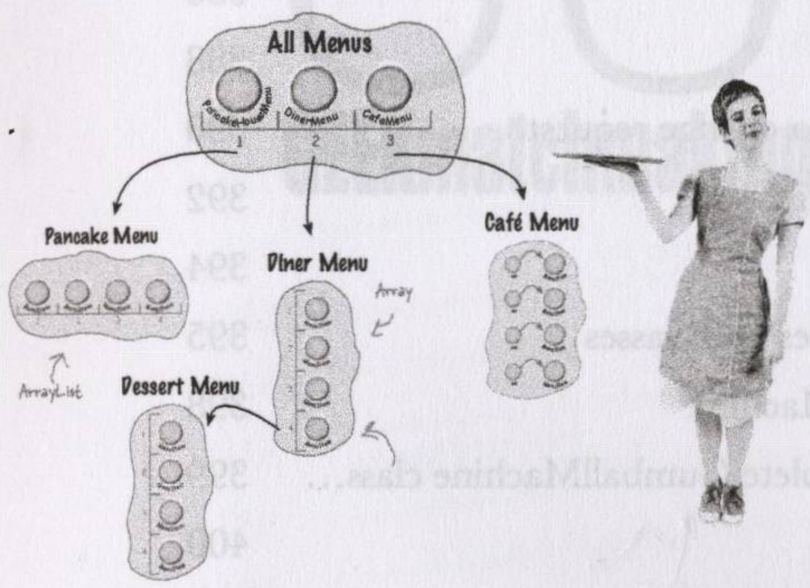


# the Iterator and Composite Patterns

9

## Well-Managed Collections

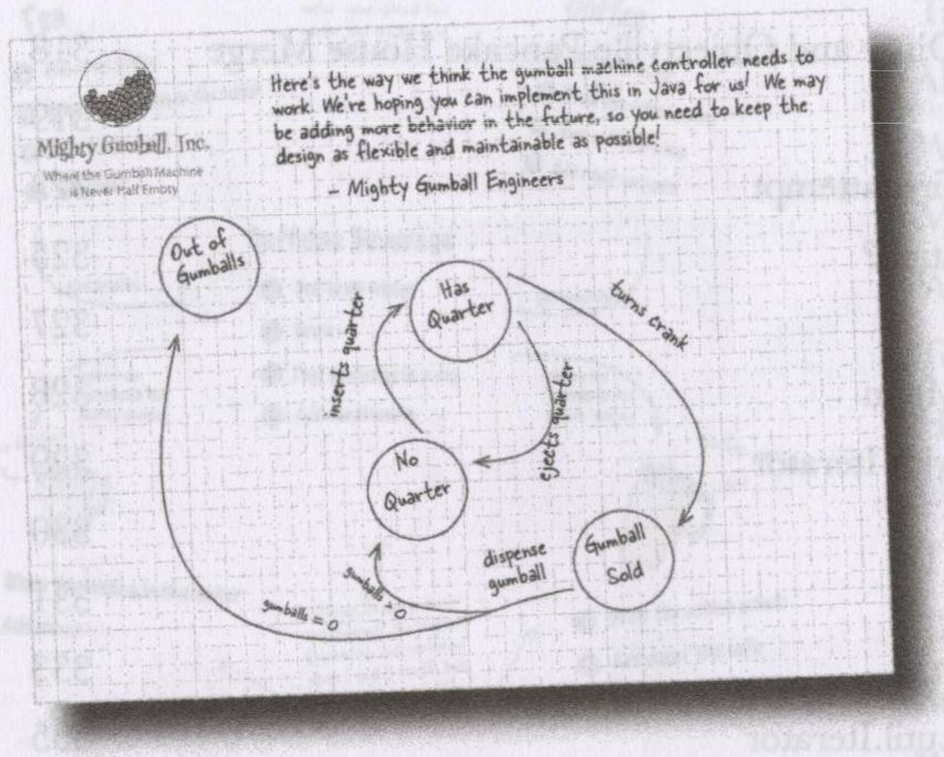
**There are lots of ways to stuff objects into a collection.** Put them into an Array, a Stack, a List, a hash map—take your pick. Each has its own advantages and tradeoffs. But at some point your clients are going to want to iterate over those objects, and when they do, are you going to show them your implementation? We certainly hope not! That just wouldn't be professional. Well, you don't have to risk your career; in this chapter you're going to see how you can allow your clients to iterate through your objects without ever getting a peek at how you store your objects. You're also going to learn how to create some super collections of objects that can leap over some impressive data structures in a single bound. And if that's not enough, you're also going to learn a thing or two about object responsibility.



# 10 the State Pattern

## The State of Things

**A little-known fact: the Strategy and State Patterns were twins separated at birth.** You'd think they'd live similar lives, but the Strategy Pattern went on to create a wildly successful business around interchangeable algorithms, while State took the perhaps more noble path of helping objects to control their behavior by changing their internal state. As different as their paths became, however, underneath you'll find almost precisely the same design. How can that be? As you'll see, Strategy and State have very different intents. First, let's dig in and see what the State Pattern is all about, and then we'll return to explore their relationship at the end of the chapter.



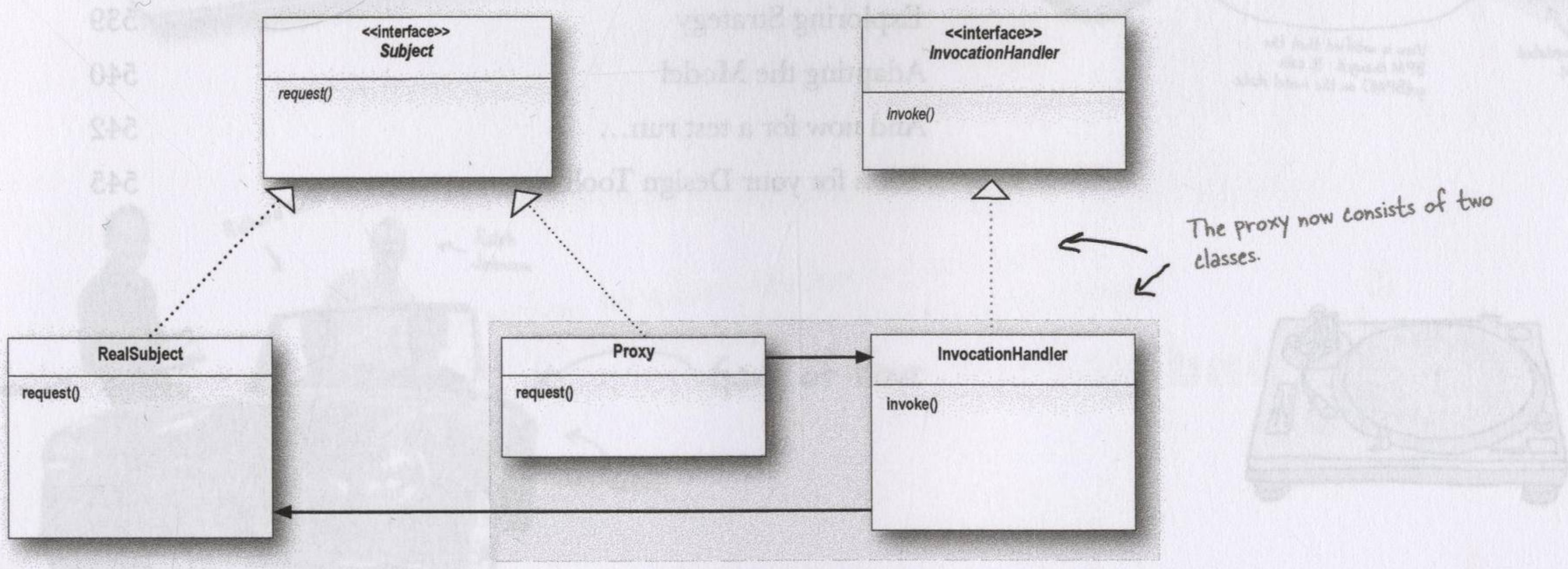



the Proxy Pattern

# 11 Controlling Object Access

**Ever play good cop, bad cop?** You're the good cop and you provide all your services in a nice and friendly manner, but you don't want everyone asking you for services, so you have the bad cop control access to you. That's what proxies do: control and manage access. As you're going to see, there are lots of ways in which proxies stand in for the objects they proxy. Proxies have been known to haul entire method calls over the internet for their proxied objects; they've also been known to patiently stand in for some pretty lazy objects.



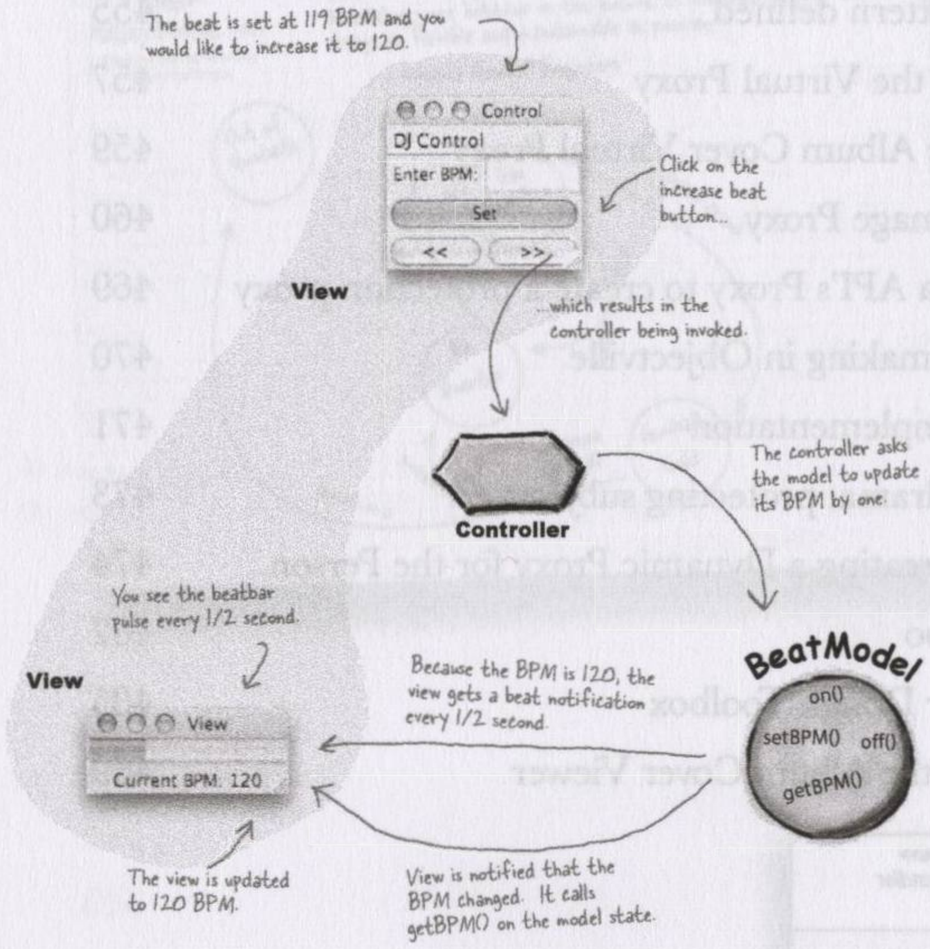



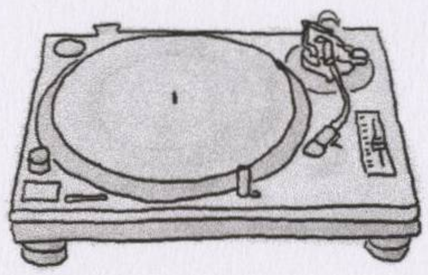
compound patterns

# 12 Patterns of Patterns

**Who would have ever guessed that Patterns could work together?** You've already witnessed the acrimonious Fireside Chats (and you haven't even seen the Pattern Death Match pages that the editor forced us to remove from the book), so who would have thought patterns can actually get along well together? Well, believe it or not, some of the most powerful OO designs use several patterns together. Get ready to take your pattern skills to the next level; it's time for compound patterns.



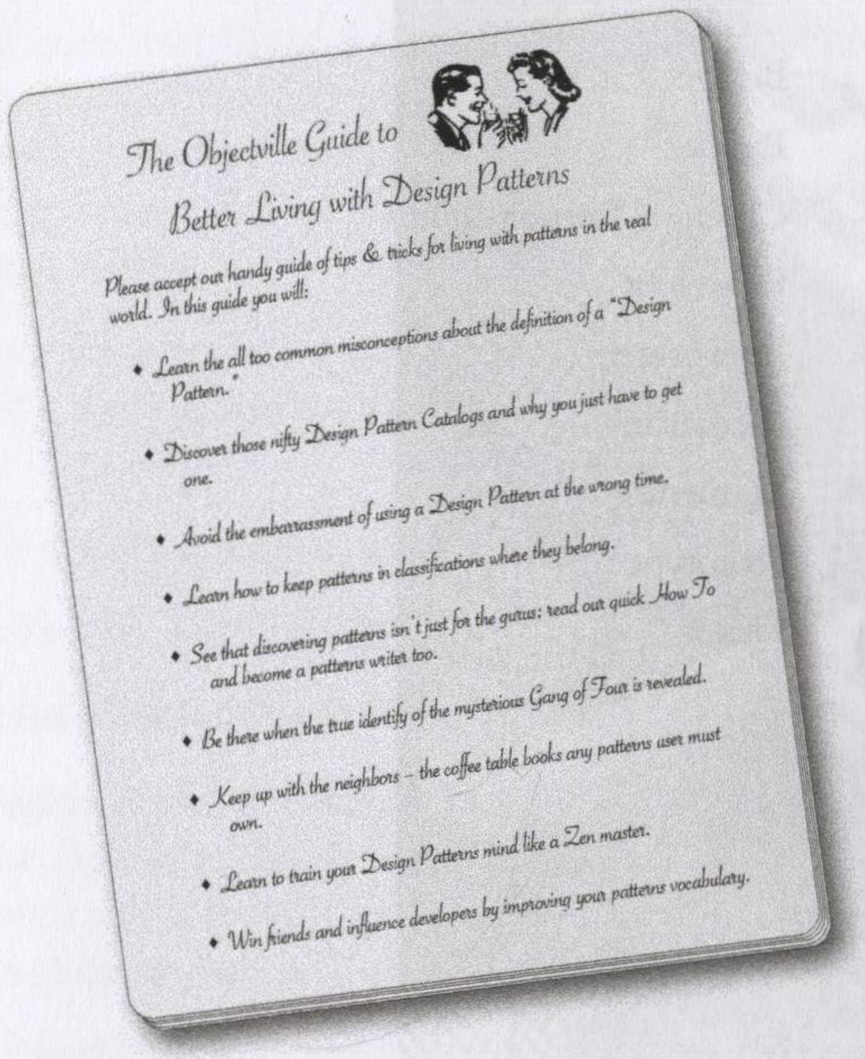


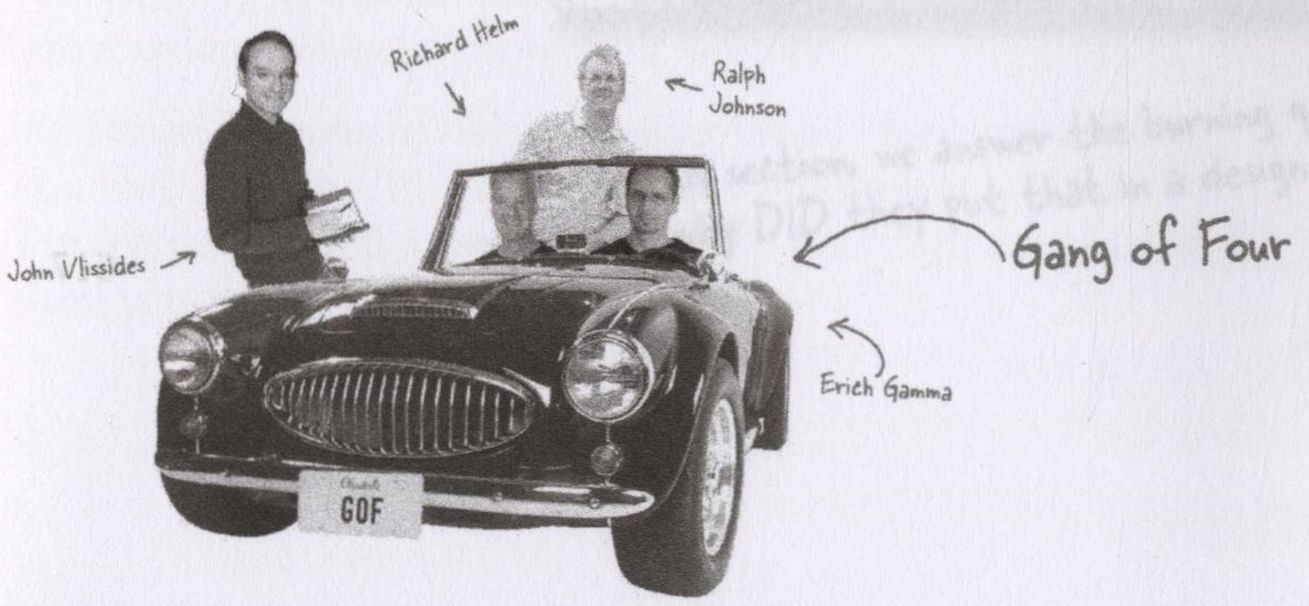
better living with patterns

# 13 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...





# 14 Appendix: Leftover Patterns

**Not everyone can be the most popular.** A lot has changed in the last 25+ 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 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.