Tools for your Design Toolbox

You've nearly made it through the first chapter! You've already put a few tools in your OO toolbox; let's make a list of them before we move on to Chapter 2.

OO Basics

Abstraction

Encapsulation

Polymorphism

Inheritance

We assume you know the OO basics like abstraction, encapsulation, polymorphism, and inheritance. If you are a little rusty on these, pull out your favorite object-oriented book and review, then skim this chapter again.

OO Principles

Encapsulate what varies.

Favor composition over inheritance.

Program to interfaces, not implementations.

We'll be taking a closer look at these down the road and also adding a few more to the list.

OO Patterns

Strategy – defines a family of algorithms, encapsulates each one, and makes them interchangeable. Strategy lets the algorithm vary independently from clients that use it.

Throughout the book, think about how patterns rely on OO basics and principles.

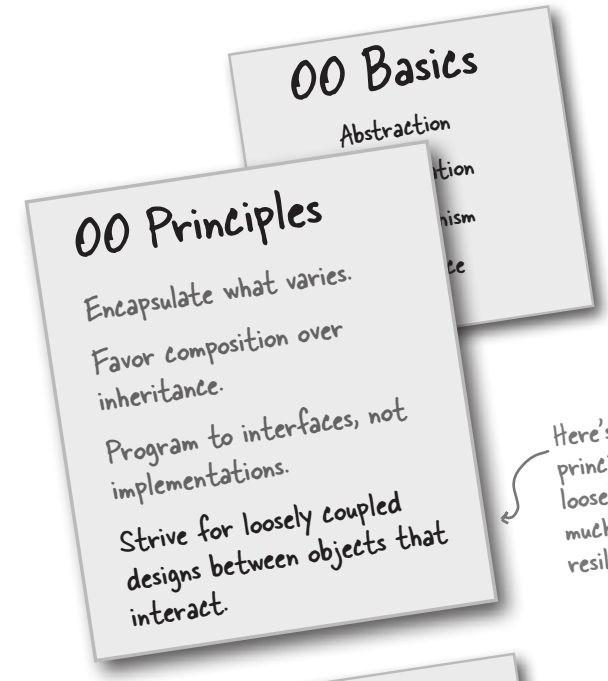
One down, many to go!

BULLET POINTS

- Knowing the OO basics does not make you a good OO designer.
- Good OO designs are reusable, extensible, and maintainable.
- Patterns show you how to build systems with good OO design qualities.
- Patterns are proven object-oriented experience.
- Patterns don't give you code, they give you general solutions to design problems. You apply them to your specific application.
- Patterns aren't *invented*, they are *discovered*.
- Most patterns and principles address issues of *change* in software.
- Most patterns allow some part of a system to vary independently of all other parts.
- We often try to take what varies in a system and encapsulate it.
- Patterns provide a shared language that can maximize the value of your communication with other developers.

Tools for your Design Toolbox

Welcome to the end of Chapter 2. You've added a few new things to your OO toolbox...

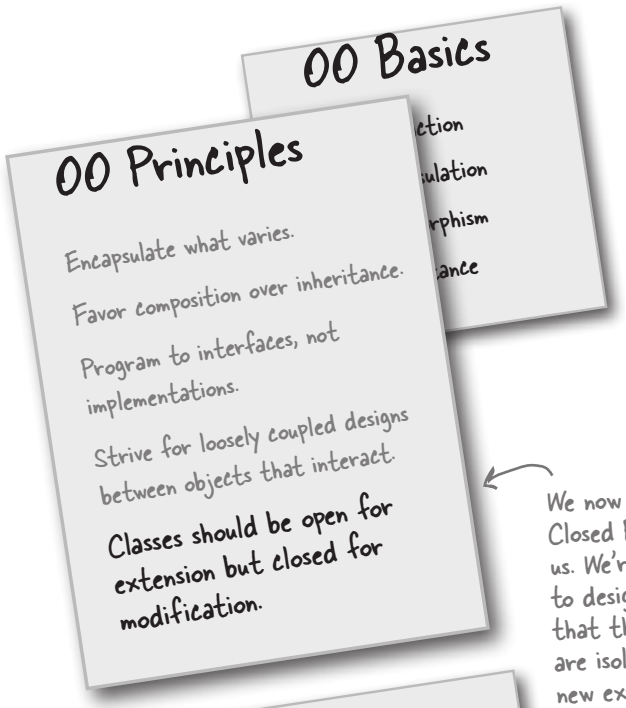


BULLET POINTS

- The Observer Pattern defines a one-to-many relationship between objects.
- Subjects update Observers using a common interface.
- Observers of any concrete type can participate in the pattern as long as they implement the Observer interface.
- Observers are loosely coupled in that the Subject knows nothing about them, other than that they implement the Observer interface.
- You can push or pull data from the Subject when using the pattern (pull is considered more "correct").
- Swing makes heavy use of the Observer Pattern, as do many GUI frameworks.
- You'll also find the pattern in many other places, including RxJava, JavaBeans, and RMI, as well as in other language frameworks, like Cocoa, Swift, and JavaScript events.
- The Observer Pattern is related to the Publish/Subscribe Pattern, which is for more complex situations with multiple Subjects and/or multiple message types.
- The Observer Pattern is a commonly used pattern, and we'll see it again when we learn about Model-View-Controller.

Tools for your Design Toolbox

You've got another chapter under your belt and a new principle and pattern in the toolbox.

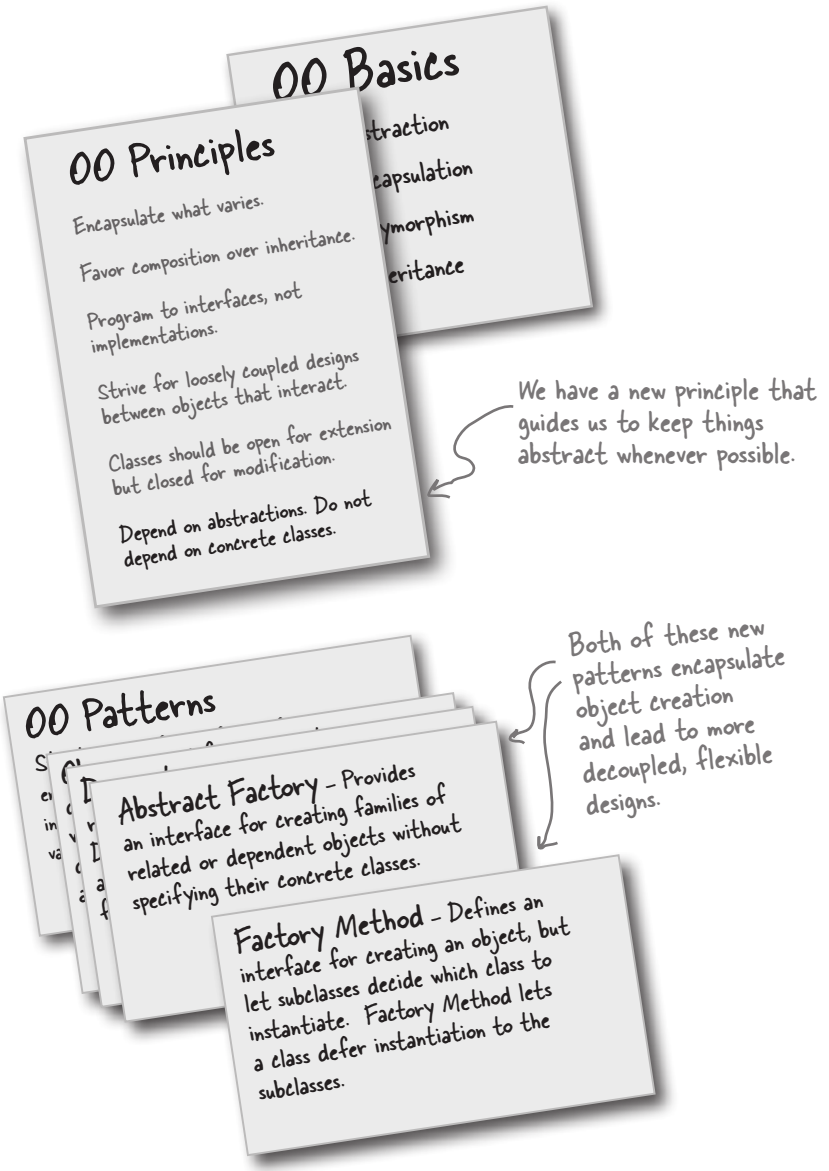


BULLET POINTS

- Inheritance is one form of extension, but not necessarily the best way to achieve flexibility in our designs.
- In our designs we should allow behavior to be extended without the need to modify existing code.
- Composition and delegation can often be used to add new behaviors at runtime.
- The Decorator Pattern provides an alternative to subclassing for extending behavior.
- The Decorator Pattern involves a set of decorator classes that are used to wrap concrete components.
- Decorator classes mirror the type of the components they decorate. (In fact, they are the same type as the components they decorate, either through inheritance or interface implementation.)
- Decorators change the behavior of their components by adding new functionality before and/or after (or even in place of) method calls to the component.
- You can wrap a component with any number of decorators.
- Decorators are typically transparent to the client of the component—that is, unless the client is relying on the component's concrete type.
- Decorators can result in many small objects in our design, and overuse can be complex.

Tools for your Design Toolbox

In this chapter, we added two more tools to your toolbox: Factory Method and Abstract Factory. Both patterns encapsulate object creation and allow you to decouple your code from concrete types.

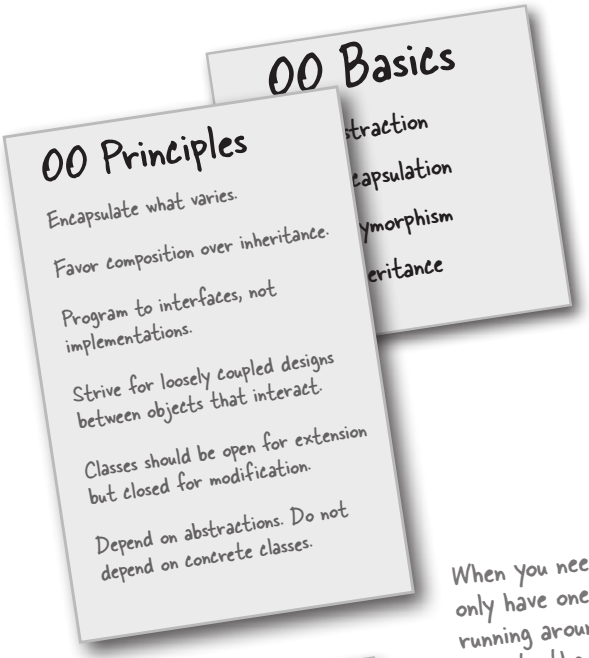


BULLET POINTS

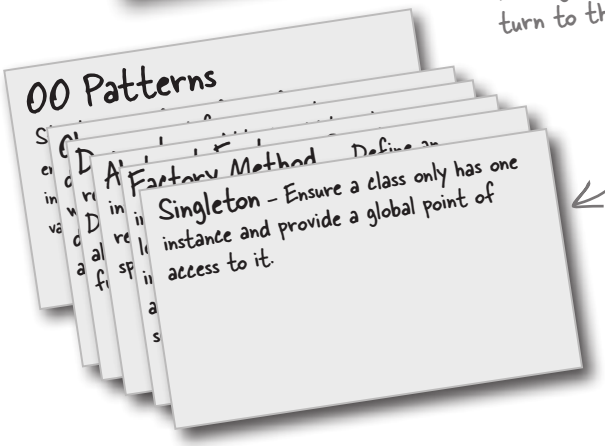
- All factories encapsulate object creation.
- Simple Factory, while not a bona fide design pattern, is a simple way to decouple your clients from concrete classes.
- Factory Method relies on inheritance: object creation is delegated to subclasses, which implement the factory method to create objects.
- Abstract Factory relies on object composition: object creation is implemented in methods exposed in the factory interface.
- All factory patterns promote loose coupling by reducing the dependency of your application on concrete classes.
- The intent of Factory Method is to allow a class to defer instantiation to its subclasses.
- The intent of Abstract Factory is to create families of related objects without having to depend on their concrete classes.
- The Dependency Inversion Principle guides us to avoid dependencies on concrete types and to strive for abstractions.
- Factories are a powerful technique for coding to abstractions, not concrete classes.

Tools for your Design Toolbox

You've now added another pattern to your toolbox. Singleton gives you another method of creating objects—in this case, unique objects.



When you need to ensure you only have one instance of a class running around your application, turn to the Singleton.



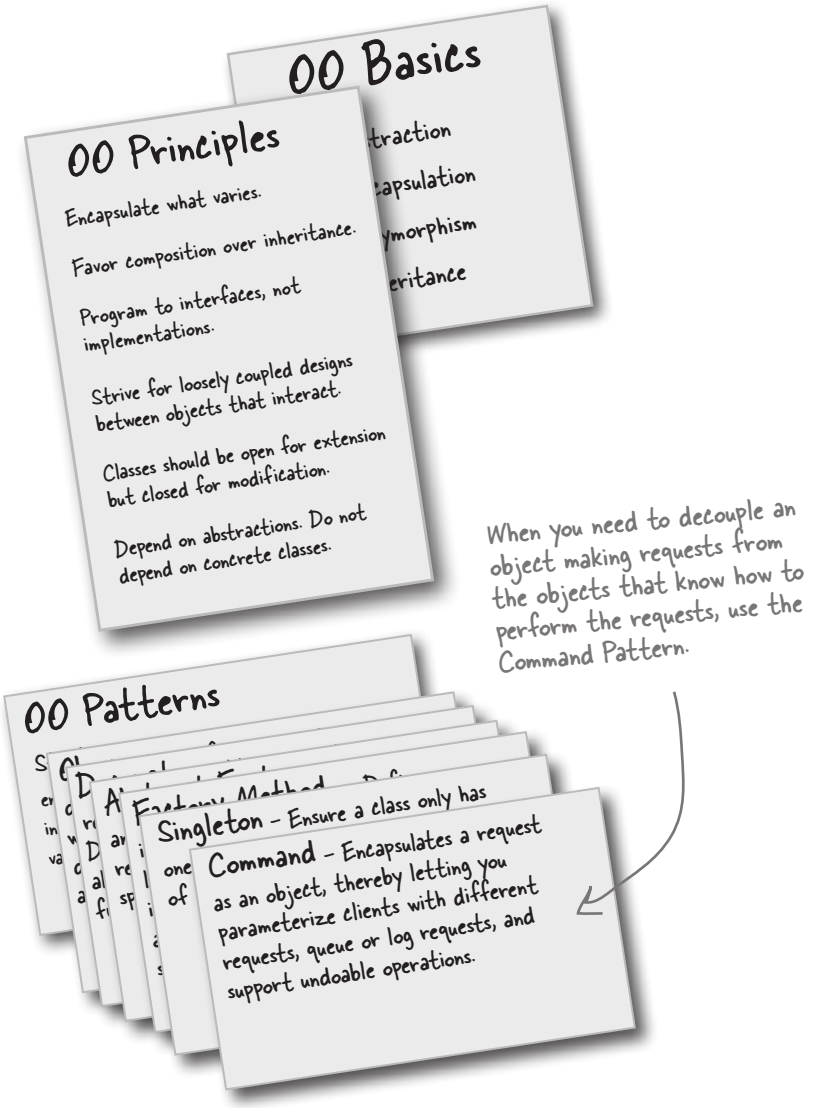
As you've seen, despite its apparent simplicity, there are a lot of details involved in Singleton's implementation. After reading this chapter, though, you're ready to go out and use Singleton in the wild.

BULLET POINTS

- The Singleton Pattern ensures you have at most one instance of a class in your application.
- The Singleton Pattern also provides a global access point to that instance.
- Java's implementation of the Singleton Pattern makes use of a private constructor, a static method combined with a static variable.
- Examine your performance and resource constraints and carefully choose an appropriate Singleton implementation for multithreaded applications (and we should consider all applications multithreaded!).
- Beware of the double-checked locking implementation; it isn't thread safe in versions before Java 5.
- Be careful if you are using multiple class loaders; this could defeat the Singleton implementation and result in multiple instances.
- You can use Java's enums to simplify your Singleton implementation.

Tools for your Design Toolbox

Your toolbox is starting to get heavy! In this chapter we've added a pattern that allows us to encapsulate methods into Command objects: store them, pass them around, and invoke them when you need them.

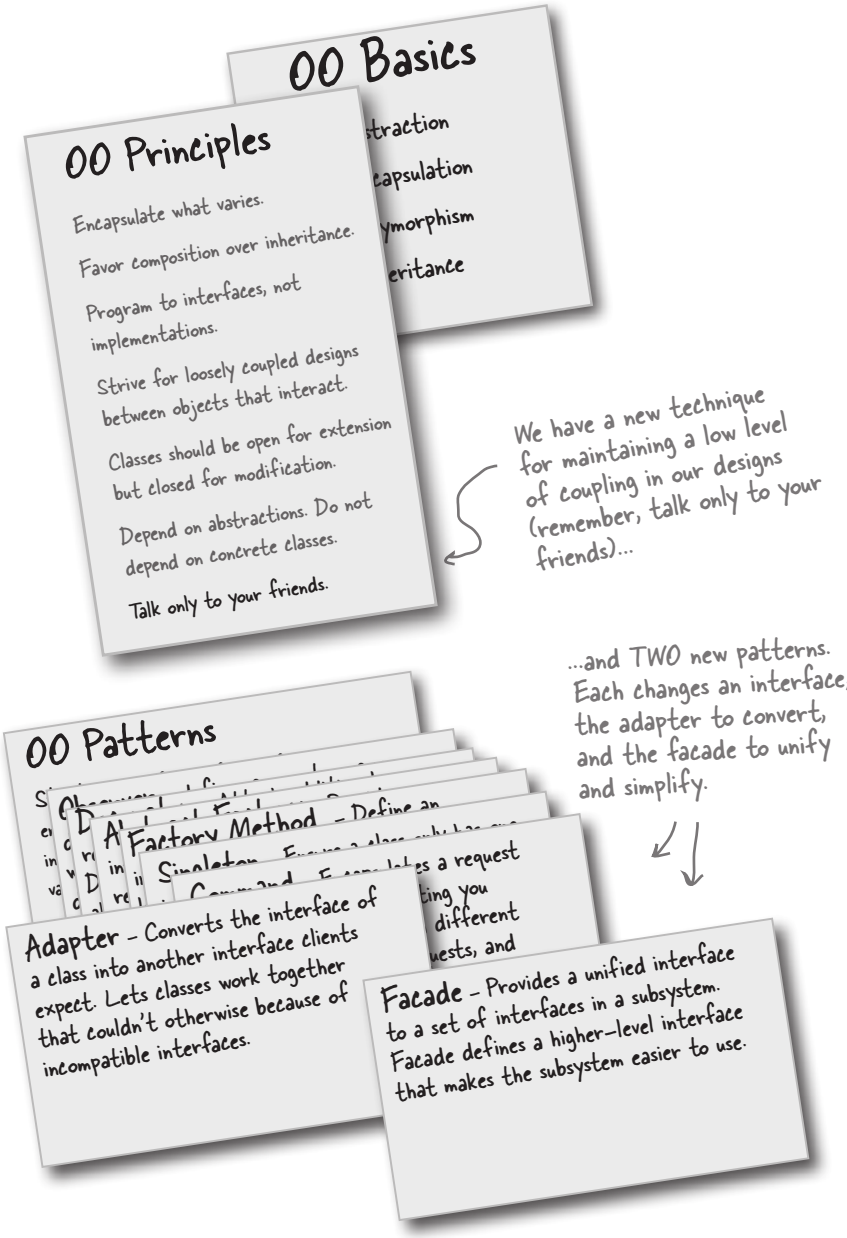


BULLET POINTS

- The Command Pattern decouples an object making a request from the one that knows how to perform it.
- A Command object is at the center of this decoupling and encapsulates a receiver with an action (or set of actions).
- An invoker makes a request of a Command object by calling its execute() method, which invokes those actions on the receiver.
- Invokers can be parameterized with Commands, even dynamically at runtime.
- Commands may support undo by implementing an undo() method that restores the object to its previous state before the execute() method was last called.
- MacroCommands are a simple extension of the Command Pattern that allow multiple commands to be invoked. Likewise, MacroCommands can easily support undo().
- In practice, it's not uncommon for "smart" Command objects to implement the request themselves rather than delegating to a receiver.
- Commands may also be used to implement logging and transactional systems.

Tools for your Design Toolbox

Your toolbox is starting to get heavy! In this chapter we've added a couple of patterns that allow us to alter interfaces and reduce coupling between clients and the systems they use.



BULLET POINTS

- When you need to use an existing class and its interface is not the one you need, use an adapter.
- When you need to simplify and unify a large interface or complex set of interfaces, use a facade.
- An adapter changes an interface into one a client expects.
- A facade decouples a client from a complex subsystem.
- Implementing an adapter may require little work or a great deal of work depending on the size and complexity of the target interface.
- Implementing a facade requires that we compose the facade with its subsystem and use delegation to perform the work of the facade.
- There are two forms of the Adapter Pattern: object and class adapters. Class adapters require multiple inheritance.
- You can implement more than one facade for a subsystem.
- An adapter wraps an object to change its interface, a decorator wraps an object to add new behaviors and responsibilities, and a facade "wraps" a set of objects to simplify.

Tools for your Design Toolbox

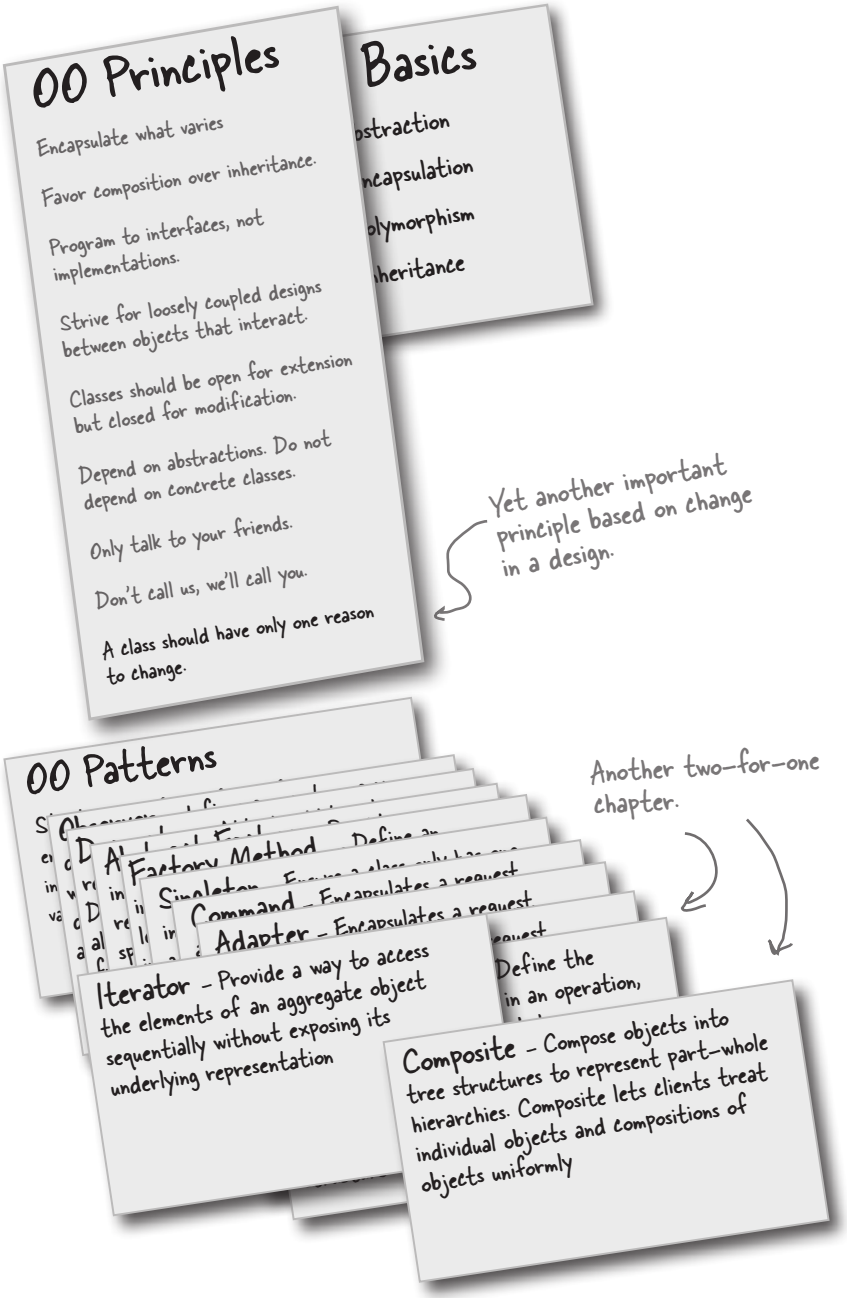
We've added Template Method to your toolbox. With Template Method, you can reuse code like a pro while keeping control of your algorithms.

BULLET POINTS

- A template method defines the steps of an algorithm, deferring to subclasses for the implementation of those steps.
- The Template Method Pattern gives us an important technique for code reuse.
- The template method's abstract class may define concrete methods, abstract methods, and hooks.
- Abstract methods are implemented by subclasses.
- Hooks are methods that do nothing or default behavior in the abstract class, but may be overridden in the subclass.
- To prevent subclasses from changing the algorithm in the template method, declare the template method as final.
- The Hollywood Principle guides us to put decision making in high-level modules that can decide how and when to call low-level modules.
- You'll see lots of uses of the Template Method Pattern in real-world code, but (as with any pattern) don't expect it all to be designed "by the book."
- The Strategy and Template Method Patterns both encapsulate algorithms, the first by composition and the other by inheritance.
- Factory Method is a specialization of Template Method.

Tools for your Design Toolbox

Two new patterns for your toolbox—two great ways to deal with collections of objects.

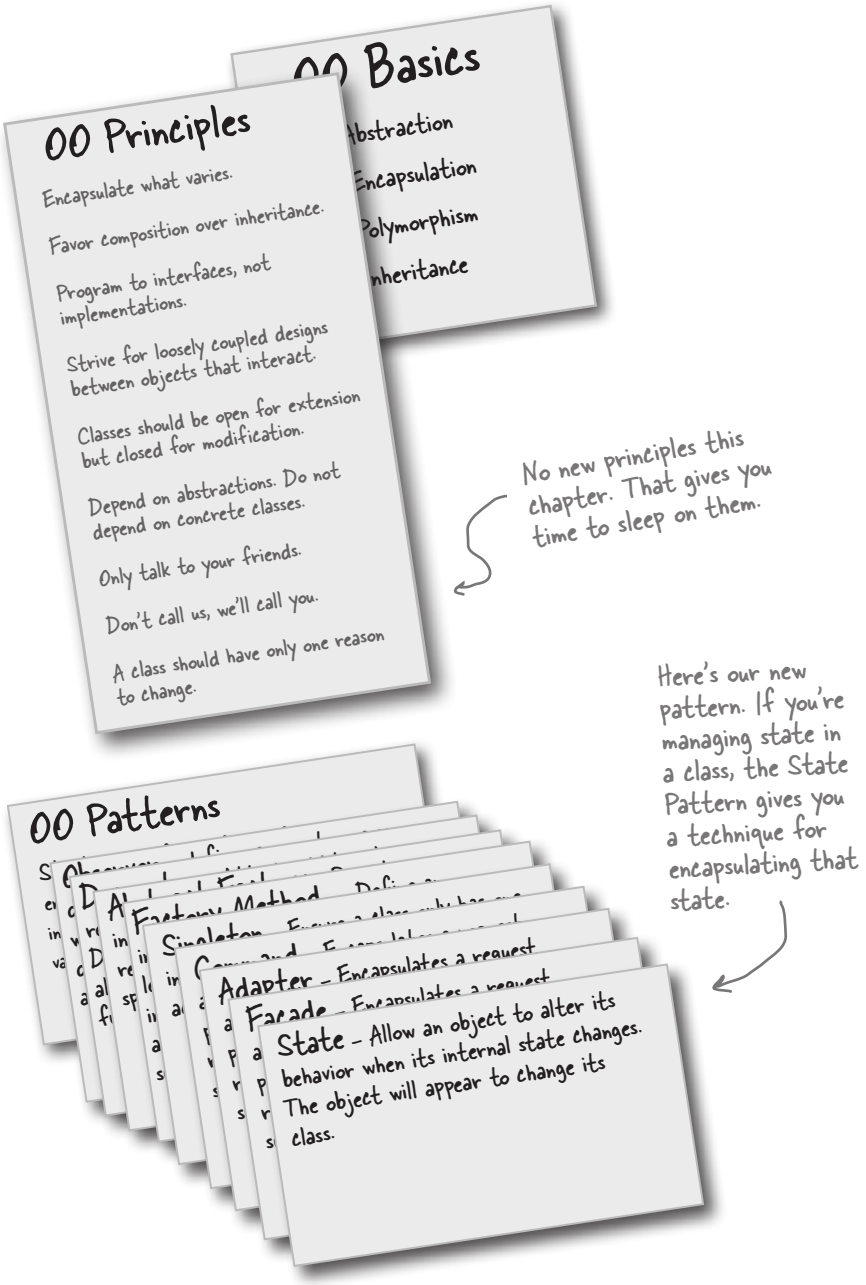


BULLET POINTS

- An Iterator allows access to an aggregate's elements without exposing its internal structure.
- An Iterator takes the job of iterating over an aggregate and encapsulates it in another object.
- When using an Iterator, we relieve the aggregate of the responsibility of supporting operations for traversing its data.
- An Iterator provides a common interface for traversing the items of an aggregate, allowing you to use polymorphism when writing code that makes use of the items of the aggregate.
- The Iterable interface provides a means of getting an iterator and enables Java's enchanced for loop.
- We should strive to assign only one responsibility to each class.
- The Composite Pattern allows clients to treat composites and individual objects uniformly.
- A Component is any object in a Composite structure. Components may be other composites or leaves.
- There are many design tradeoffs in implementing Composite. You need to balance transparency and safety with your needs.

Tools for your Design Toolbox

It's the end of another chapter; you've got enough patterns here to breeze through any job interview!

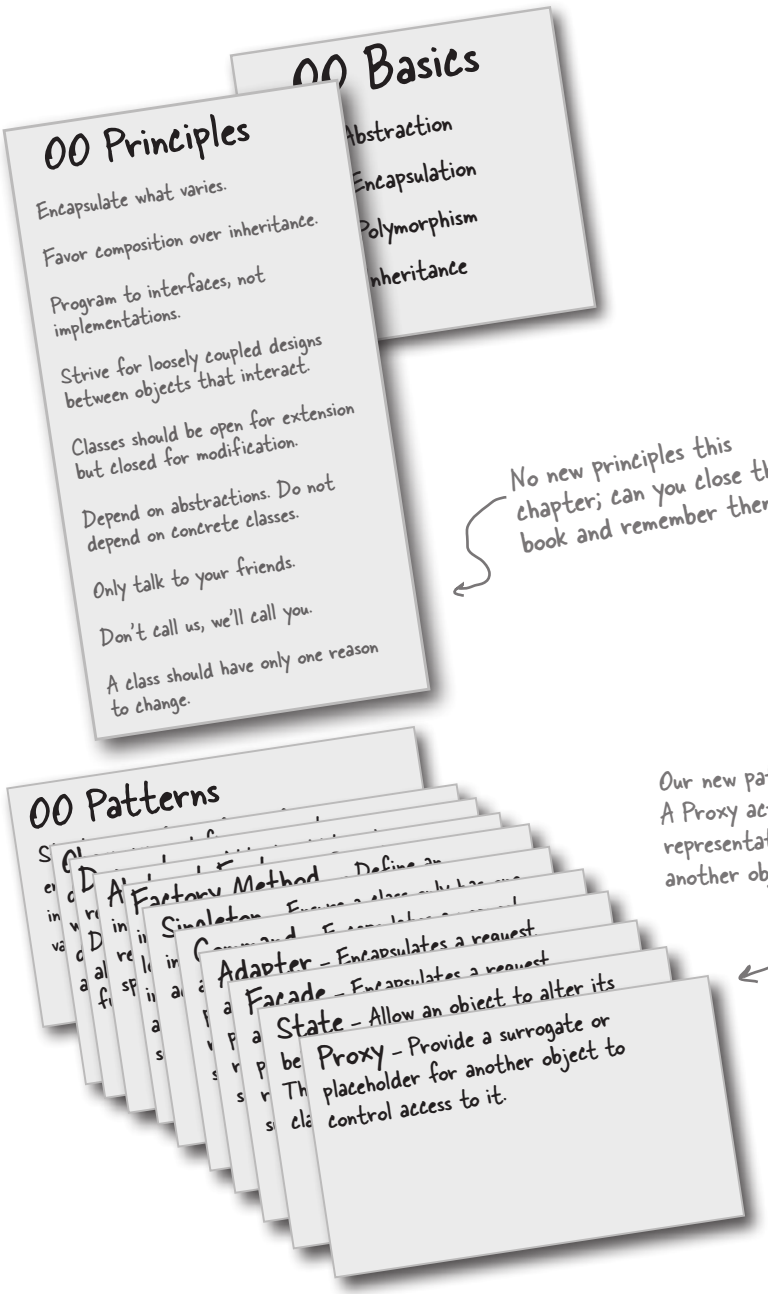


BULLET POINTS

- The State Pattern allows an object to have many different behaviors that are based on its internal state.
- Unlike a procedural state machine, the State Pattern represents each state as a full-blown class.
- The Context gets its behavior by delegating to the current state object it is composed with.
- By encapsulating each state into a class, we localize any changes that will need to be made.
- The State and Strategy Patterns have the same class diagram, but they differ in intent.
- The Strategy Pattern typically configures Context classes with a behavior or algorithm.
- The State Pattern allows a Context to change its behavior as the state of the Context changes.
- State transitions can be controlled by the State classes or by the Context classes.
- Using the State Pattern will typically result in a greater number of classes in your design.
- State classes may be shared among Context instances.

Tools for your Design Toolbox

Your design toolbox is almost full; you're prepared for almost any design problem that comes your way.

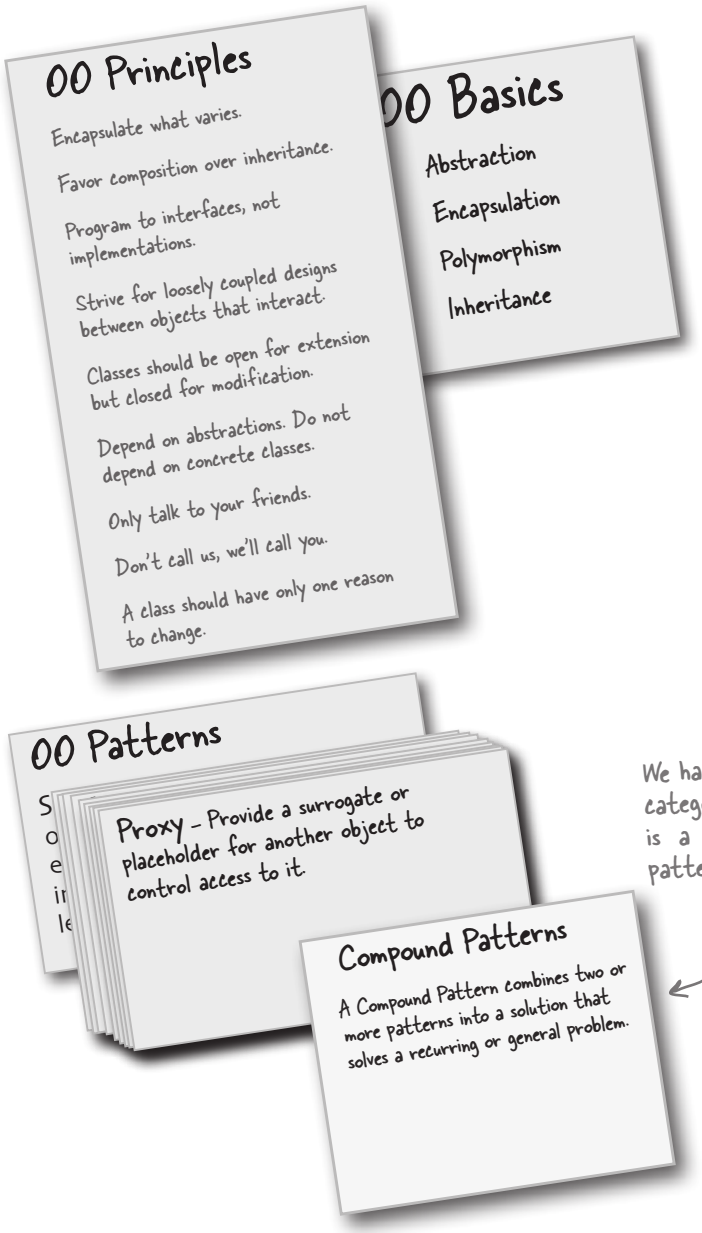


BULLET POINTS

- The Proxy Pattern provides a representative for another object in order to control the client's access to it. There are a number of ways it can manage that access.
- A Remote Proxy manages interaction between a client and a remote object.
- A Virtual Proxy controls access to an object that is expensive to instantiate.
- A Protection Proxy controls access to the methods of an object based on the caller.
- Many other variants of the Proxy Pattern exist including caching proxies, synchronization proxies, firewall proxies, copy-on-write proxies, and so on.
- Proxy is structurally similar to Decorator, but the two patterns differ in their purpose.
- The Decorator Pattern adds behavior to an object, while Proxy controls access.
- Java's built-in support for Proxy can build a dynamic proxy class on demand and dispatch all calls on it to a handler of your choosing.
- Like any wrapper, proxies will increase the number of classes and objects in your designs.

Tools for your Design Toolbox

You could impress anyone with your design toolbox. Wow, look at all those principles, patterns, and now, compound patterns!

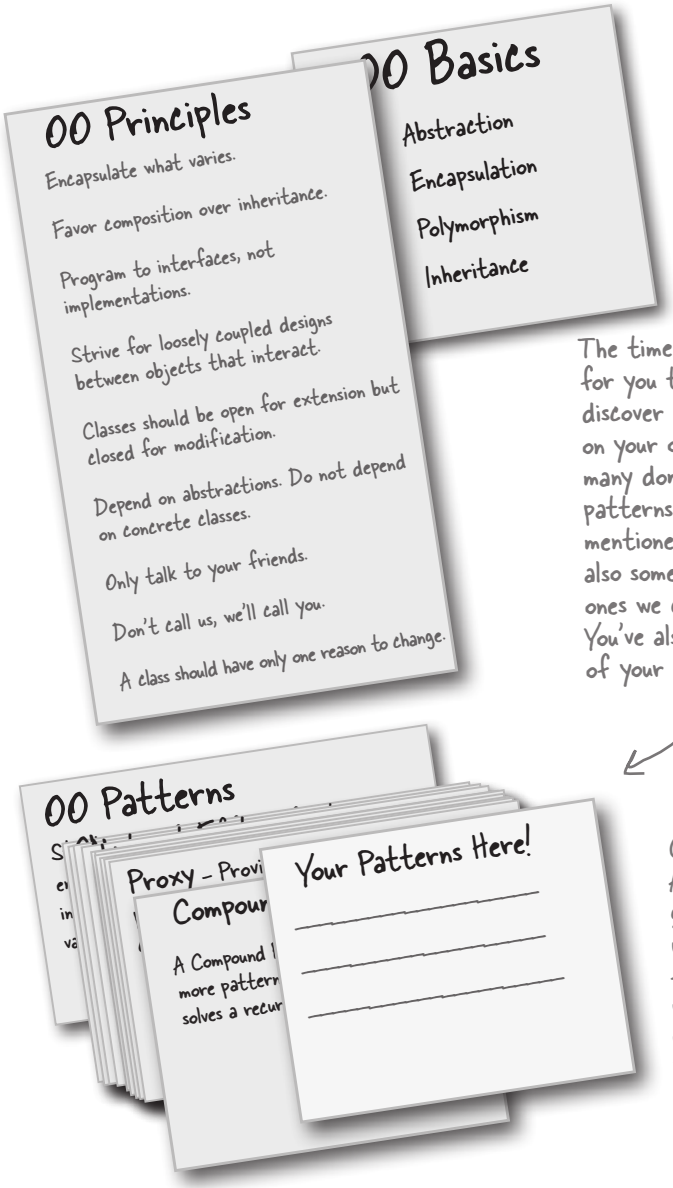


BULLET POINTS

- The Model View Controller (MVC) Pattern is a compound pattern consisting of the Observer, Strategy, and Composite Patterns.
- The model makes use of the Observer Pattern so that it can keep observers updated yet stay decoupled from them.
- The controller is the Strategy for the view. The view can use different implementations of the controller to get different behavior.
- The view uses the Composite Pattern to implement the user interface, which usually consists of nested components like panels, frames, and buttons.
- These patterns work together to decouple the three players in the MVC model, which keeps designs clear and flexible.
- The Adapter Pattern can be used to adapt a new model to an existing view and controller.
- MVC has been adapted to the web.
- There are many web MVC frameworks with various adaptations of the MVC pattern to fit the client/server application structure.

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...



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 Patterns 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.