Head First Design Patterns

## Strategy pattern

* Design principle: Identify the aspects of your application that vary and separate them from what stays the same.

Take the parts that vary and encapsulate them, so that later you can alter or extend the parts that vary without affecting those that don’t.

* Design principle: Program to an interface, not an implementation

interface FlyBehavior {

fly();

}

class FlyWithWings {

fly(); // Implement duck flying

}

class FlyNoWay {

fly(); // Do nothing. Can’t fly

}

class Duck {

FlyBehavior flyBehavior; // Can use setter to change the behavior

void performFly() {

flyBehavior.fly();

}

}

* Design principle: Favor composition over inheritance.

Not only does it let you encapsulate a family of algorithms into their own set of classes, but it also lets you change behavior at runtime.

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

## Observer pattern

* **Observer pattern**: Defines a one-to-many dependency between objects so that when one object changes state, all of its dependents are notified and update automatically
* Design principle: Strive for loosely coupled designs between objects that interact.

Loosely coupled designs allow us to build flexible OO system that can handle changes because they minimize the interdependency between objects.

* Java build in interface

java.util.Observable;

java.util.Observer;

Problems: is a class, need to inherit; setChanged() is protected, violates favor composition over inheritance.

## Decorator pattern

* Design principle: classes should be open for extension, but closed for modification

(Open-Closed principle)

* + The goal is to allow classes to be easily extended to incorporate new behavior without modifying existing code.
  + Be careful when choosing the area of code that need to be extended; applying the Open-Closed principle everywhere is wasteful, unnecessary, and can lead to complex, hard to understand code.
* **Decorator pattern:** attaches additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality.

Decorators have the same type as the objects they are going to decorate. So here using inheritance to achieve the type matching, but not behavior.

abstract class Beverage {

String description;

String getDescription() { return description; }

abstract int cost();

}

abstract class CondimentDecorator extends Beverage {

abstract String getDescription();

}

class Espresso extends Beverage {

Espresso() { description = “Espresso”;

int cost() { return 199; }

}

class Mocha extends CondimentDecorator {

Beverage beverage;

Mocha(Beverage beverage) { this.beverage = beverage; }

String getDescription() {

return beverage.getDescription() + “, Mocha”;

int cost() { return 20 + beverage.cost(); }

}

Beverage beverage = new Espresso();

beverage = new Mocha(beverage);

* Real world decorators: Java I/O

FileInputStream, BufferedInputStream, FilterInputStream

## Factory method pattern

* **Factory method pattern:** defines an interface for creating an object, but lets subclasses decide which class to instantiate. Factory method lets a class defer instantiation to subclasses.
* Design principle: Depend upon abstractions. Do not depend upon concrete classes.

(Dependency inversion principle)

Guidelines:

* + No variable should hold a reference to a concrete class
  + No class should derive from a concrete class
  + No method should override an implemented method of any of its base classes
  + Strive for this guideline, but should not follow all the time.
* **Abstract factory pattern**: Provides an interface for creating families of related or dependent objects without specifying their concrete classes
* Comparison between factory method and abstract factory:
  + Factory method creates object through inheritance; extend a class and override a factory method
  + Abstract factory creates object through object composition; provide an abstract type for creating a family of products. Subclasses of this type define how the products are produced. Need to change interface to create more products. Usually a big interface.

// Factory method

abstract alass PizzaStore { abstract Pizza createPizza(); }

class NYPizzaStore extends PizzaStore { Pizza createPizza() { } }

abstract class Pizza; class CheesePizza extends Pizza;

// Abstract factory

interface PizzaIngredientFactory {

Cheese createCheese();

Sauce createSauce(); }

class NYPizzaIngredientFactory implements PizzaIngredientFactory {

…

}

interface Cheese; MozzarellaCheese implements Cheese; ReggianoCheese implements Cheese;

## Singleton pattern

* **Singleton pattern:** Ensures a class has only one instance, and provides a global point of access to it
* To solve multithreading problem:
  + public static synchronized Singleton getInstance();
  + private static Singleton instance = new Singleton();
  + private static volatile Singleton instance;

public static Singleton getInstance() {

if (instance == null) {

synchronized(Singleton.class) {

if (instance == null) { // Need to check again

instance = new Singleton();

}

} } return instance; }

## Command pattern

* **Command pattern**: Encapsulates a request as an object, thereby letting you parameterize other objects with different requests, queue or log requests, or support undoable operations.

interface Command { void execute(); }

LightOnCommand implements Command {

Light light;

LightOnCommand(Light light) { this.light = light; }

void execute() { light.on();} }

Some usages: Queuing request; Logging requests

## Adapter and façade patterns

* **Adapter pattern**: converts the interface of a class into another interface the clients expect. Adapter lets classes work together that couldn’t otherwise because of incompatible interfaces.
  + Client makes a request to the adapter by calling a method on the target interface
  + Adapter translates the request into on or more calls on the adaptee using adaptee interfaces
  + Client receive the results of the call and never knows there is an adapter doing the translation
* **Façade pattern**: Provides a unified interface to a set of interfaces in a subsystem. Façade defines a higher-level interface that makes the subsystem easier to use.

A façade not only simplifies an interface, it decouples a client from a subsystem of components. But can still expose subsystem components.

* Façade and adapter may wrap multiple classes, but a façade’s intent is to simplify, while an adapter’s is to convert the interface to something different.
* Design principle: Talk only to your immediate friends.

(Principle of least knowledge)

Guidelines: take any object; now from any method in that object, only invoke methods that belong to:

* + The object itself
  + Objects passed in as a parameter to the method
  + Any object the method creates or instantiates
  + Any components of the object

public float getTemp() { // Not following the rule

Thermometer thermometer = station.getThermometer();

return thermometer.getTemperature();

}

public float getTemp() { // Follow the rule

return station.getThermometer().getTemperature();

}

* + Keep method calls in bounds

## Template method pattern

* **Template method pattern**: defines the skeleton of an algorithm in a method, deferring some steps to subclasses. Template method lets subclasses redefine certain steps of an algorithm without changing the algorithm’s structure

Defines the steps of an algorithm and allows subclasses to provide the implementation for one or more steps

Sample: Sort with comparison method

* Hook method: A method that is declared in the abstract class, but only given an empty or default implementation. This gives subclasses the ability to “hook into” the algorithm at various points, if they wish; a subclass is also free to ignore it.
* Design principle: Don’t call us, we’ll call you

Provides a way to prevent complex dependency so that the system design is hard to understand. Allows the low-level components to hook into a system, but the high-level components determine when they are needed and how.

## Iterator and composite pattern

## State pattern

* **State pattern**: Allows an object to alter its behavior when its internal state changes. The object will appear to change its class

class Context {

State state;

private void setState(State state);

void request() { state.handle(); }

}

interface State { void handle(); }

class StateA implements State { void handle(); }

class StateB implements State { void handle(); }

## Proxy pattern

* **Proxy pattern**: provides a surrogate or placeholder for another object to control access to it

Use the proxy pattern to create a representative object that controls access to another object, which may be remote, expensive to create or in need of securing.

Pattern

## Compound Pattern

Patterns of patterns

MVC pattern (Add screenshort of page 530)

## Better living with patterns

* Design pattern defined: A pattern is a solution to a problem in a context.
* **OO Principles**
  + Encapsulate what varies
  + Favor composition over inheritance
  + Program to interfaces, not implementation
  + 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 will call you
  + A class should have only one reason to change

## Other patterns

* **Bridge**

Use the bridge pattern to vary only your implements, but also your abstractions

Page 613

* **Builder:**

Use the builder pattern to encapsulate the construction of a product and allow it to be constructed in steps

* **Chain of responsibility**

Use the chain of responsibility pattern when you want to give more than one object a chance to handle a request

* **Flyweight**

Use the flyweight pattern when one instance of a class can be used to provide many virtual instances

Used when a class has many instances, and they can all be controlled identically

* **Interpreter**

Use the interpreter pattern to build an interpreter for a language

* **Mediator**

Use the mediator pattern to centralize complex communications and control between related objects

* **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 use requests an “undo”

* **Prototype**

Use the prototype pattern when creating an instance of a given class is either expensive or complicated

* **Visitor**

Use the visitor pattern when you want to add capabilities to a composite of objects and encapsulations is not important