Design Patterns

Akash Anjanappa

Contents

[Singleton Pattern: 2](#_Toc507627873)

[Façade design pattern: 5](#_Toc507627874)

[Adapter Pattern 7](#_Toc507627875)

[Strategy Pattern: 10](#_Toc507627876)

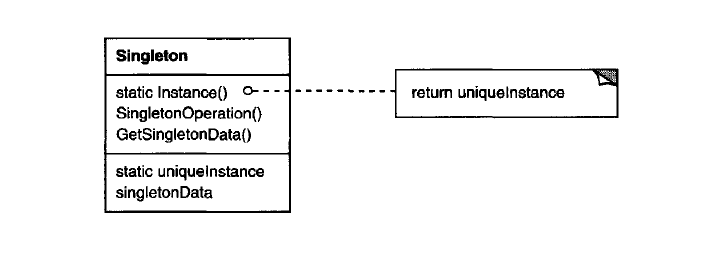
[Observer Pattern 13](#_Toc507627877)

[Decorator Pattern: 19](#_Toc507627878)

Design patterns

# Singleton Pattern:

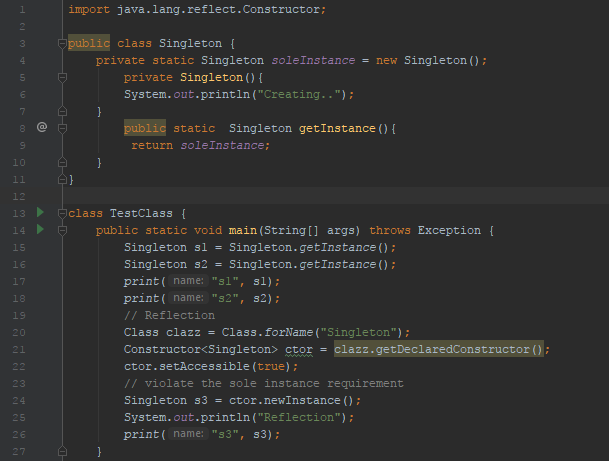
* Ensures that only one instance of a class is created.
* Provide a global point of access to it
* A solution is to make the class itself responsible for keeping track of its sole instance.
  + Hide the constructor of the class(declared private).
  + Define a public static operation that returns the sole instance of the class.

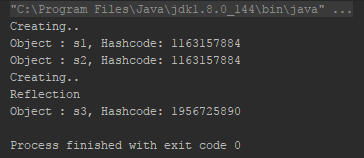


A screenshot of a cell phone

Description generated with very high confidence

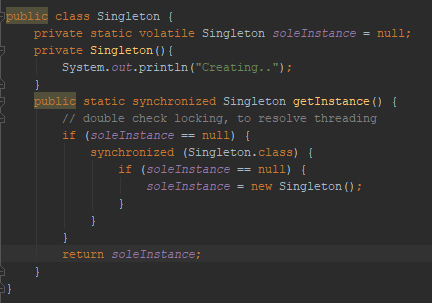
With reflection it can be changed:





* Reflection – how to fix?

Lazily initialize and check of the instance is already created with in the constructor.



Note: synchronized is not required for the method… Also note the volatile keyword.

**Question**: Which classes in JDK uses singleton pattern?  
Answer: java.lang.Runtime : In every Java application there is only one Runtime instance that allows the application to interface with the environment it is running. The getRuntime is equivalent to the getInstance() method of the singleton class.

**Question**: Can the singleton class be subclassed?  
**Answer**: Frankly speaking singleton is just a design pattern and it can be subclassed. However it is worth to understand the logic or requirement behind subclassing a singleton class as the child class might not inherit the singleton pattern objective by extending the Singleton class. However the subclassing can be prevented by using the final keyword in the class declaration.

**Question**: Can there be multiple instance of singleton using cloning?  
**Answer**: That was a good catch! What do we do now? To prevent the another instance to be created of the singleton instance we can throw exception from inside the clone() method.

# Façade design pattern:

* Provide a unified interface to a set of interfaces in a subsystem. Facade defines a higher-level interface that makes the subsystem easier to use.
* Wrap a complicated subsystem with a simpler interface.

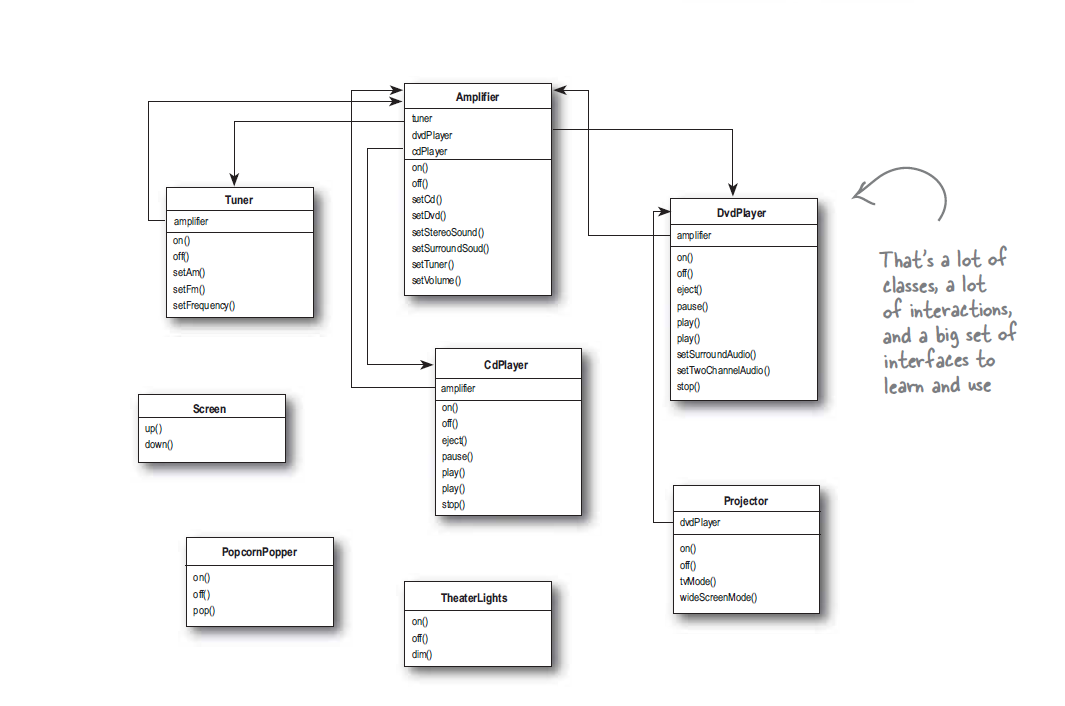
**Advantage of Facade Pattern**

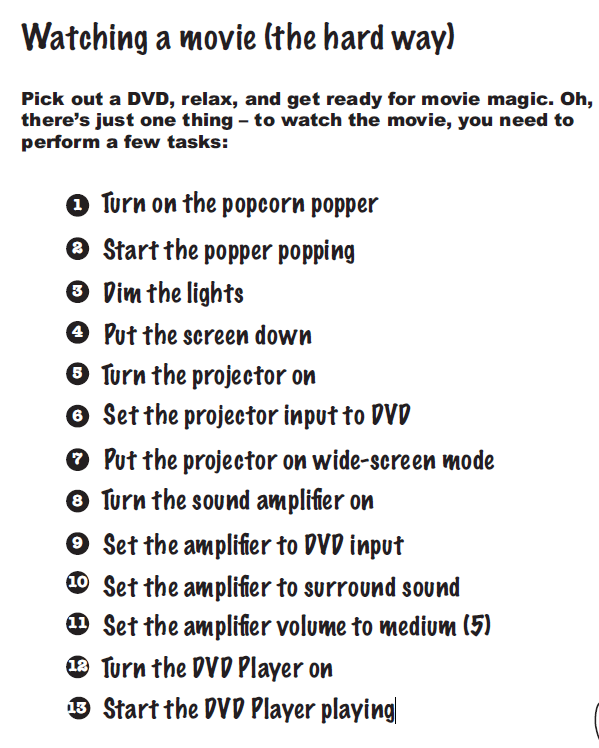
* It shields the clients from the complexities of the sub-system components.
* It promotes loose coupling between subsystems and its clients.

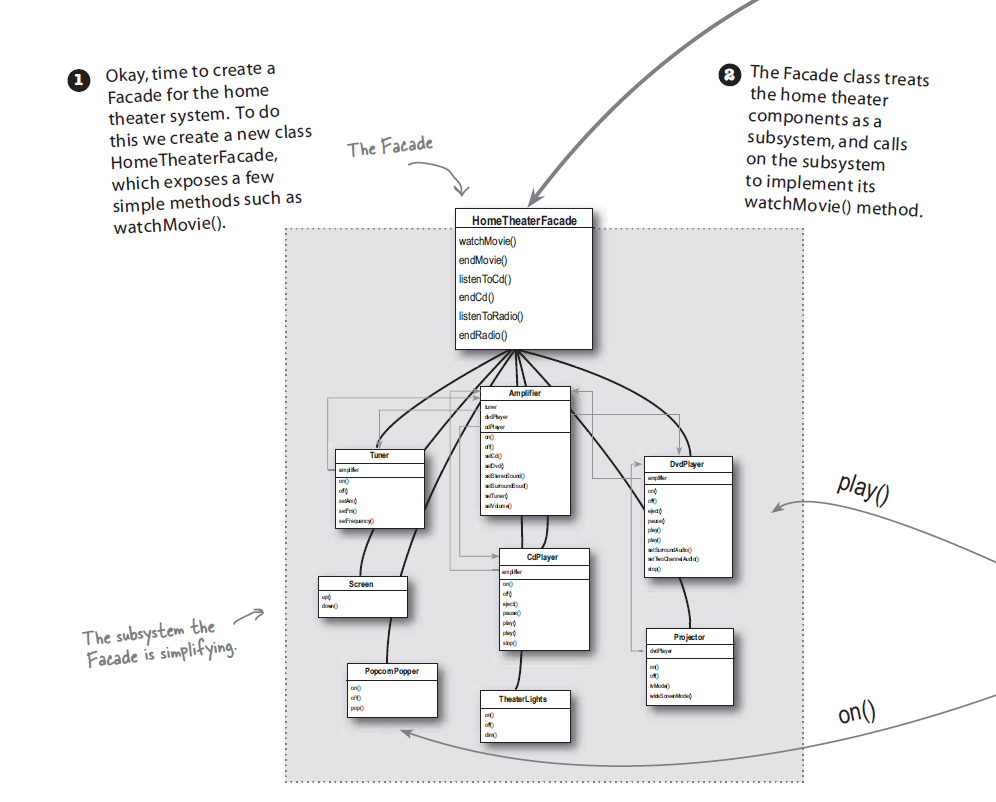
**Usage of Facade Pattern:**

It is used:

* When you want to provide simple interface to a complex sub-system.
* When several dependencies exist between clients and the implementation classes of an abstraction.



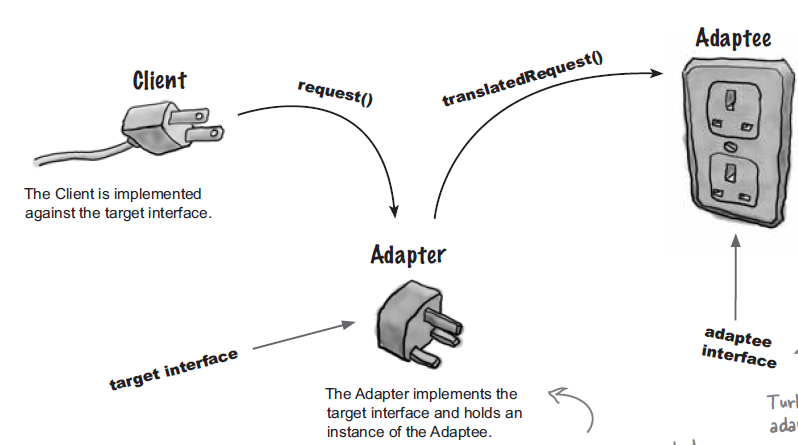


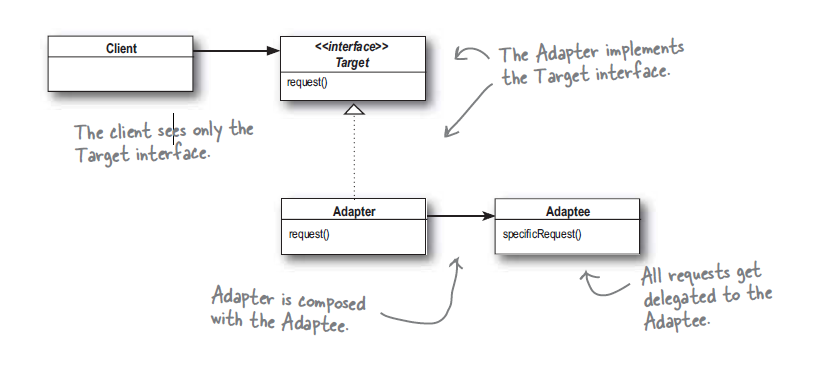


# Adapter Pattern

***Definition:***

The adapter pattern converts the interface of a class into another interface clients expect. Adapter lets classes work together that couldn’t otherwise because of incompatible interfaces.



**ClassDiagram:**  


The client sees only the target interface and not the adapter. The adapter implements the target interface. Adapter delegates all requests to Adaptee.

1. The client makes a request to the adapter by calling a method on it using the target interface.
2. The adapter translates that request on the adaptee using the adaptee interface.
3. Client receive the results of the call and is unaware of adapter’s presence.

**Example:**

Suppose you have a Bird class with fly() , and makeSound()methods. And also a ToyDuck class with squeak() method. Let’s assume that you are short on ToyDuck objects and you would like to use Bird objects in their place. Birds have some similar functionality but implement a different interface, so we can’t use them directly. So we will use adapter pattern. Here our client would be ToyDuck and adaptee would be Bird.

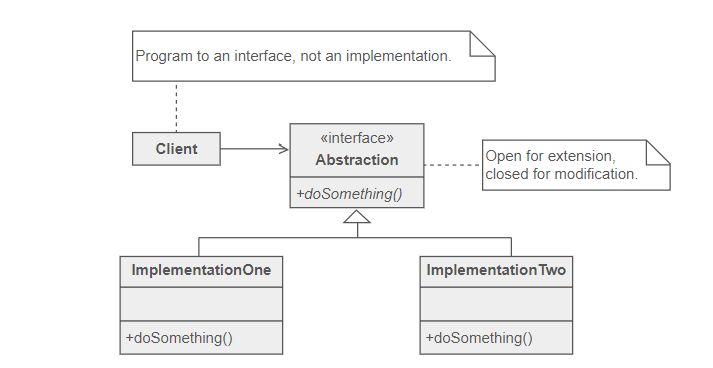
Below is Java implementation of it.

|  |
| --- |
| // Java implementation of Adapter pattern    interface Bird  {      // birds implement Bird interface that allows      // them to fly and make sounds adaptee interface      public void fly();      public void makeSound();  }    class Sparrow implements Bird  {      // a concrete implementation of bird      public void fly()      {          System.out.println("Flying");      }      public void makeSound()      {          System.out.println("Chirp Chirp");      }  }    interface ToyDuck  {      // target interface      // toyducks dont fly they just make      // squeaking sound      public void squeak();  }    class PlasticToyDuck implements ToyDuck  {      public void squeak()      {          System.out.println("Squeak");      }  }    class BirdAdapter implements ToyDuck  {      // You need to implement the interface your      // client expects to use.      Bird bird;      public BirdAdapter(Bird bird)      {          // we need reference to the object we          // are adapting          this.bird = bird;      }        public void squeak()      {          // translate the methods appropriately          bird.makeSound();      }  }    class Main  {      public static void main(String args[])      {          Sparrow sparrow = new Sparrow();          PlasticToyDuck toyDuck = new PlasticToyDuck();            // Wrap a bird in a birdAdapter so that it          // behaves like toy duck          ToyDuck birdAdapter = new BirdAdapter(sparrow);            System.out.println("Sparrow...");          sparrow.fly();          sparrow.makeSound();            System.out.println("ToyDuck...");          toyDuck.squeak();            // bird behaving like a toy duck          System.out.println("BirdAdapter...");          birdAdapter.squeak();      }  } |

# Strategy Pattern:

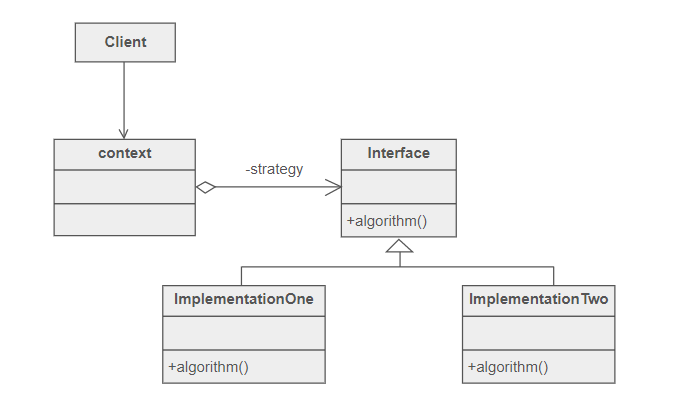
The **strategy pattern** (also known as the **policy pattern**) is a [behavioral](https://en.wikipedia.org/wiki/Behavioral_design_pattern" \o "Behavioral design pattern) [software design pattern](https://en.wikipedia.org/wiki/Design_pattern_(computer_science)) that enables selecting an [algorithm](https://en.wikipedia.org/wiki/Algorithm) at runtime.

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



"Program to an interface, not an implementation".

A Strategy defines a set of algorithms that can be used interchangeably.

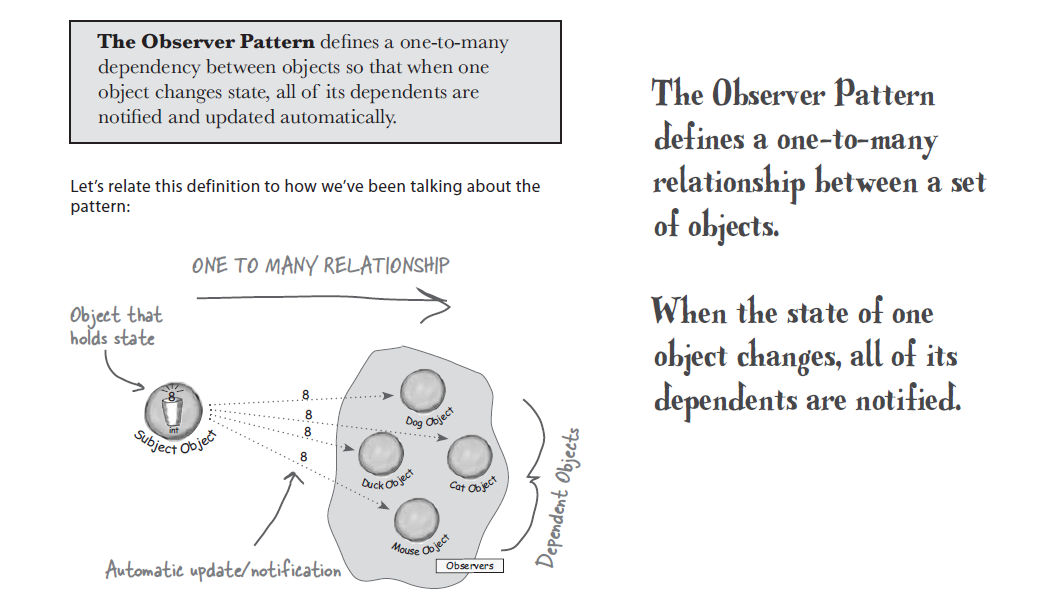


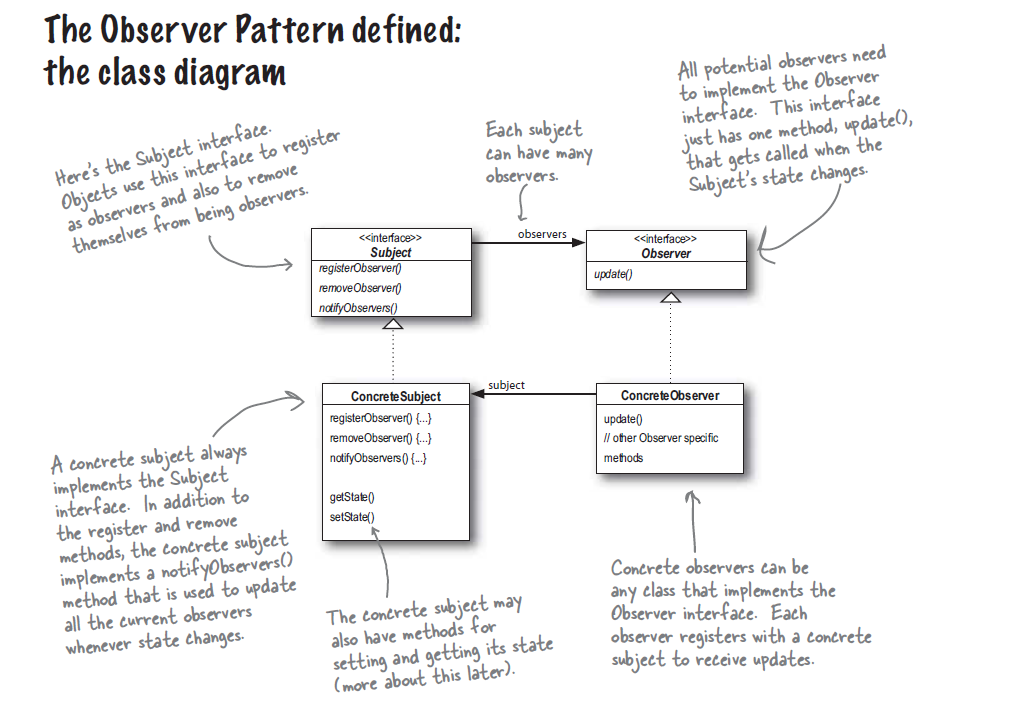
Check list

1. Identify an algorithm (i.e. a behavior) that the client would prefer to access through a "flex point".
2. Specify the signature for that algorithm in an interface.
3. Bury the alternative implementation details in derived classes.
4. Clients of the algorithm couple themselves to the interface.
5. **import** **java.util.ArrayList**;
6. **import** **java.util.List**;
7. **public** **class** **StrategyPatternWiki** {
8. **public** **static** void main(**final** String[] arguments) {
9. Customer firstCustomer = **new** Customer(**new** NormalStrategy());
10. *// Normal billing*
11. firstCustomer.add(1.0, 1);
12. *// Start Happy Hour*
13. firstCustomer.setStrategy(**new** HappyHourStrategy());
14. firstCustomer.add(1.0, 2);
15. *// New Customer*
16. Customer secondCustomer = **new** Customer(**new** HappyHourStrategy());
17. secondCustomer.add(0.8, 1);
18. *// The Customer pays*
19. firstCustomer.printBill();
20. *// End Happy Hour*
21. secondCustomer.setStrategy(**new** NormalStrategy());
22. secondCustomer.add(1.3, 2);
23. secondCustomer.add(2.5, 1);
24. secondCustomer.printBill();
25. }
26. }
27. **class** **Customer** {
28. **private** List<Double> drinks;
29. **private** BillingStrategy strategy;
30. **public** Customer(**final** BillingStrategy strategy) {
31. **this**.drinks = **new** ArrayList<Double>();
32. **this**.strategy = strategy;
33. }
34. **public** void add(**final** double price, **final** int quantity) {
35. drinks.add(strategy.getActPrice(price\*quantity));
36. }
37. *// Payment of bill*
38. **public** void printBill() {
39. double sum = 0;
40. **for** (Double i : drinks) {
41. sum += i;
42. }
43. System.out.println("Total due: " + sum);
44. drinks.clear();
45. }
46. *// Set Strategy*
47. **public** void setStrategy(**final** BillingStrategy strategy) {
48. **this**.strategy = strategy;
49. }
50. }
51. **interface** **BillingStrategy** {
52. double getActPrice(**final** double rawPrice);
53. }
54. *// Normal billing strategy (unchanged price)*
55. **class** **NormalStrategy** **implements** BillingStrategy {
56. @Override
57. **public** double getActPrice(**final** double rawPrice) {
58. **return** rawPrice;
59. }
60. }
61. *// Strategy for Happy hour (50% discount)*
62. **class** **HappyHourStrategy** **implements** BillingStrategy {
63. @Override
64. **public** double getActPrice(**final** double rawPrice) {
65. **return** rawPrice\*0.5;
66. }
67. }

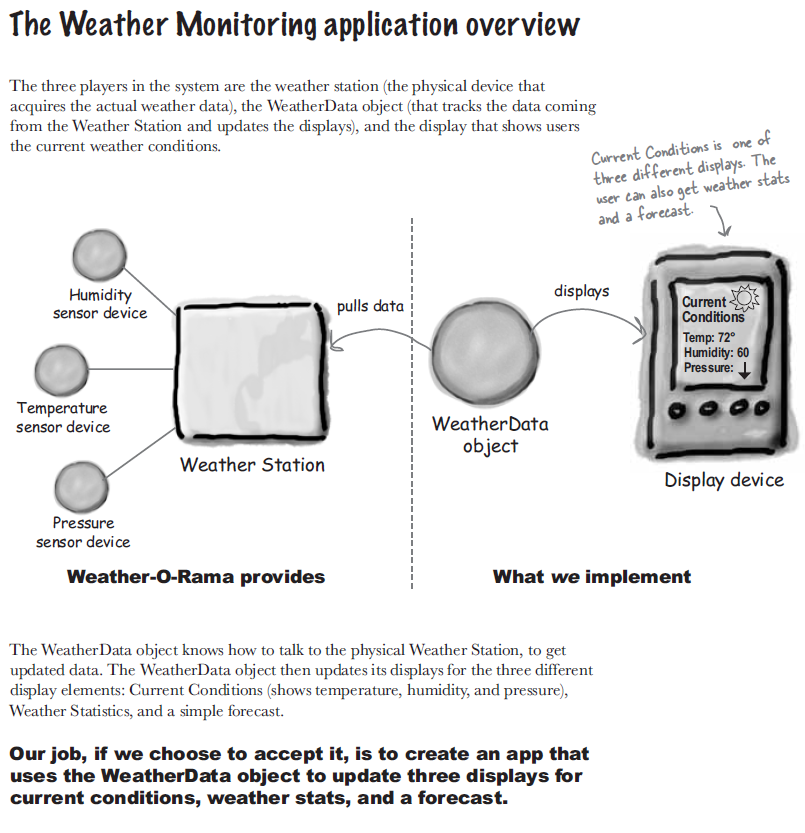
# Observer Pattern

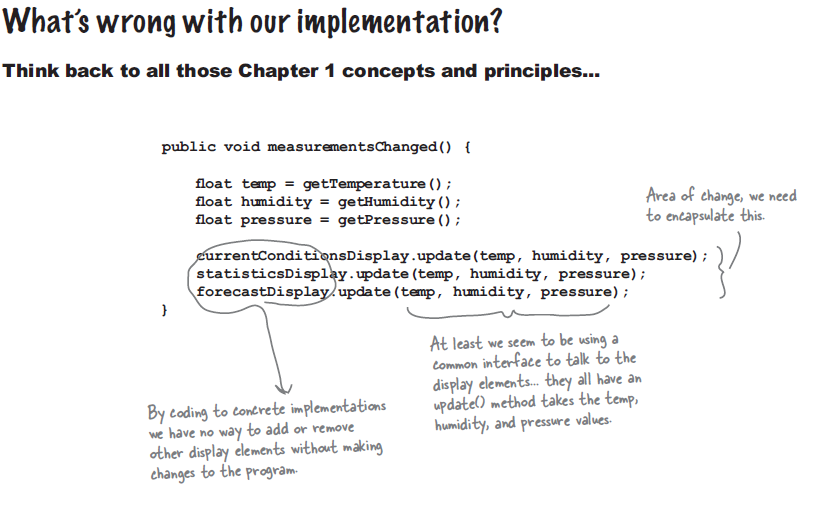
The observer pattern defines a one-to-many dependency between objects so that when one object changes state, all of its dependents are notified and updated automatically.



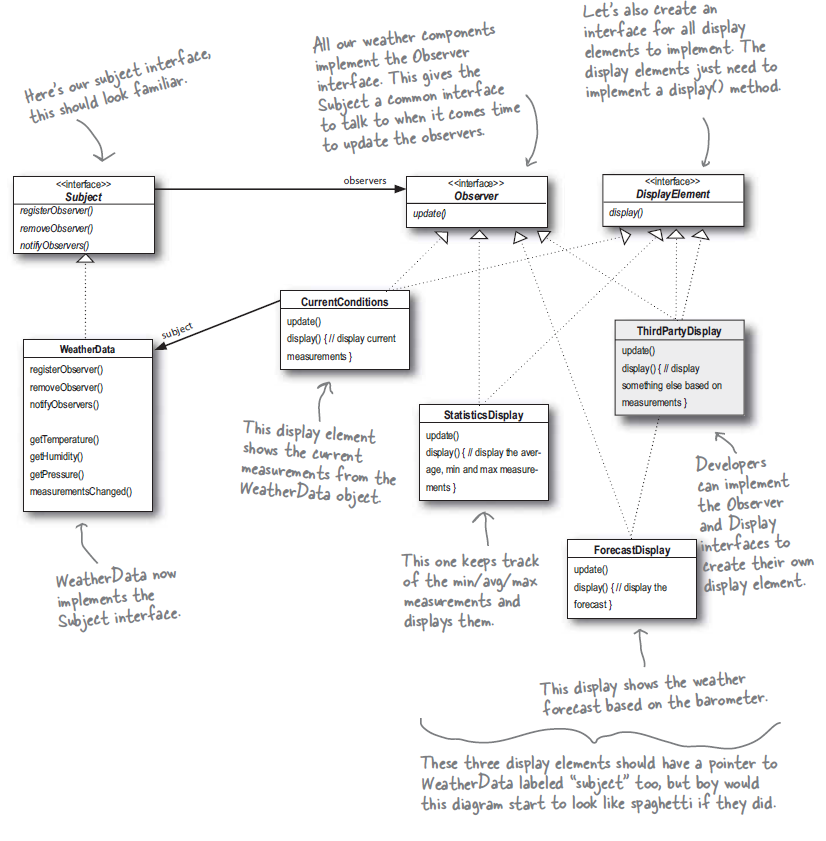


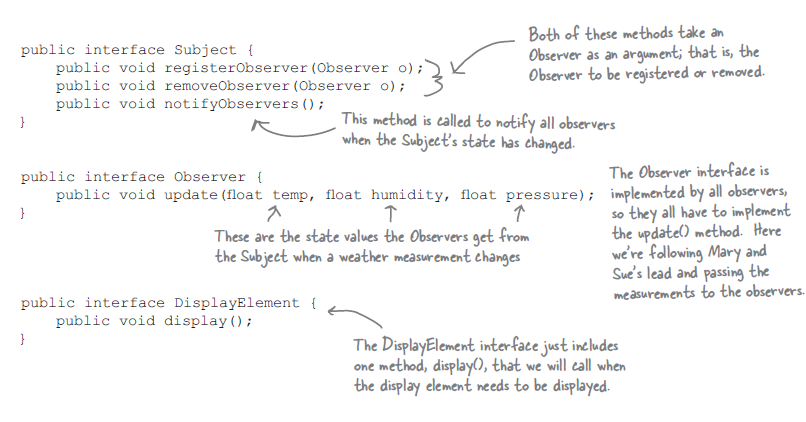
Example:

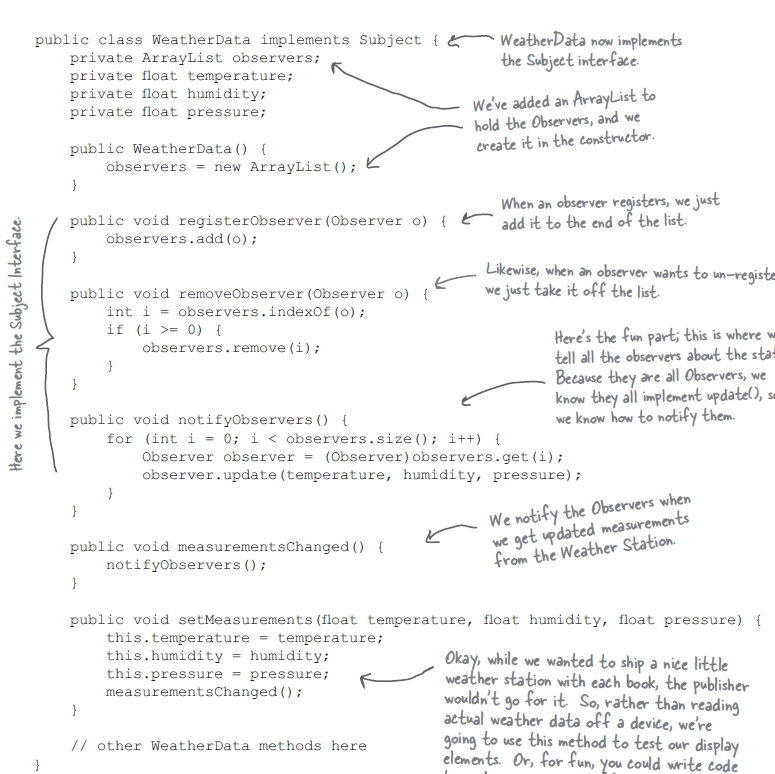


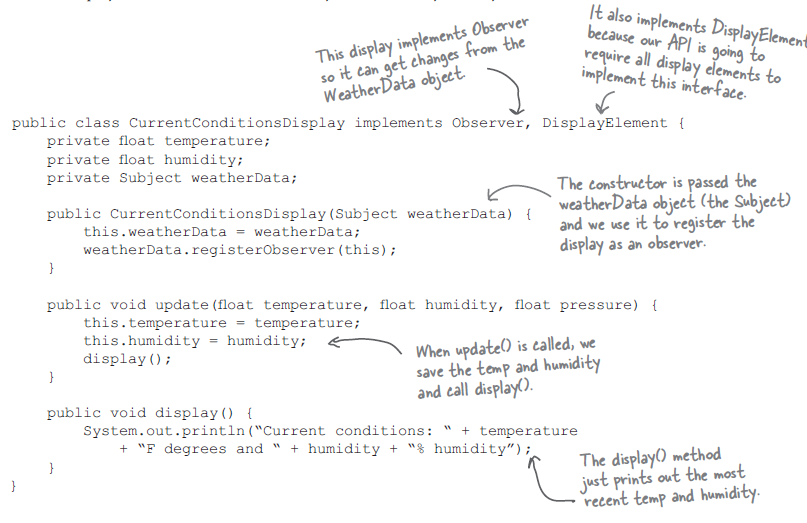


With Observer Pattern:

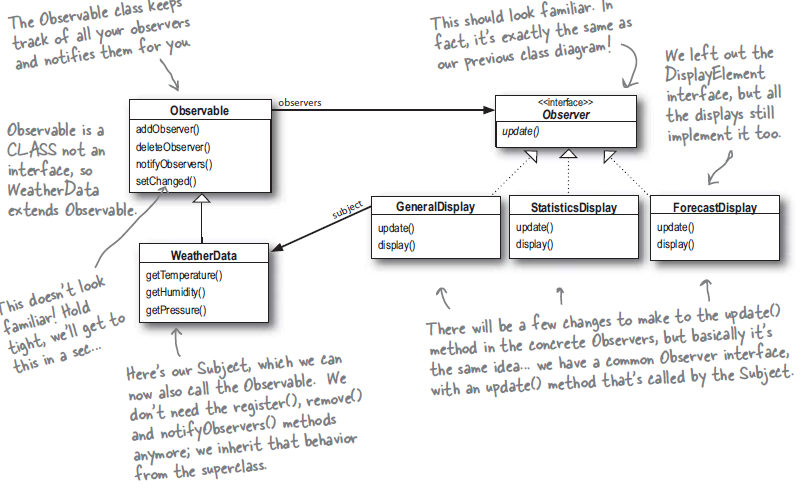


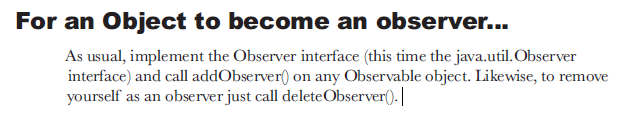


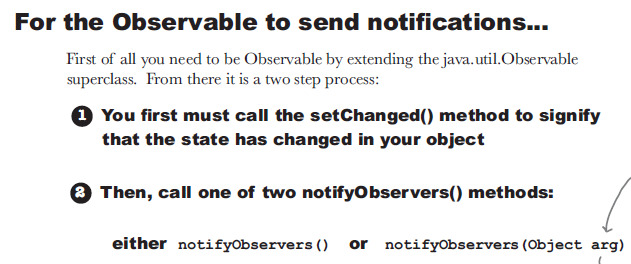








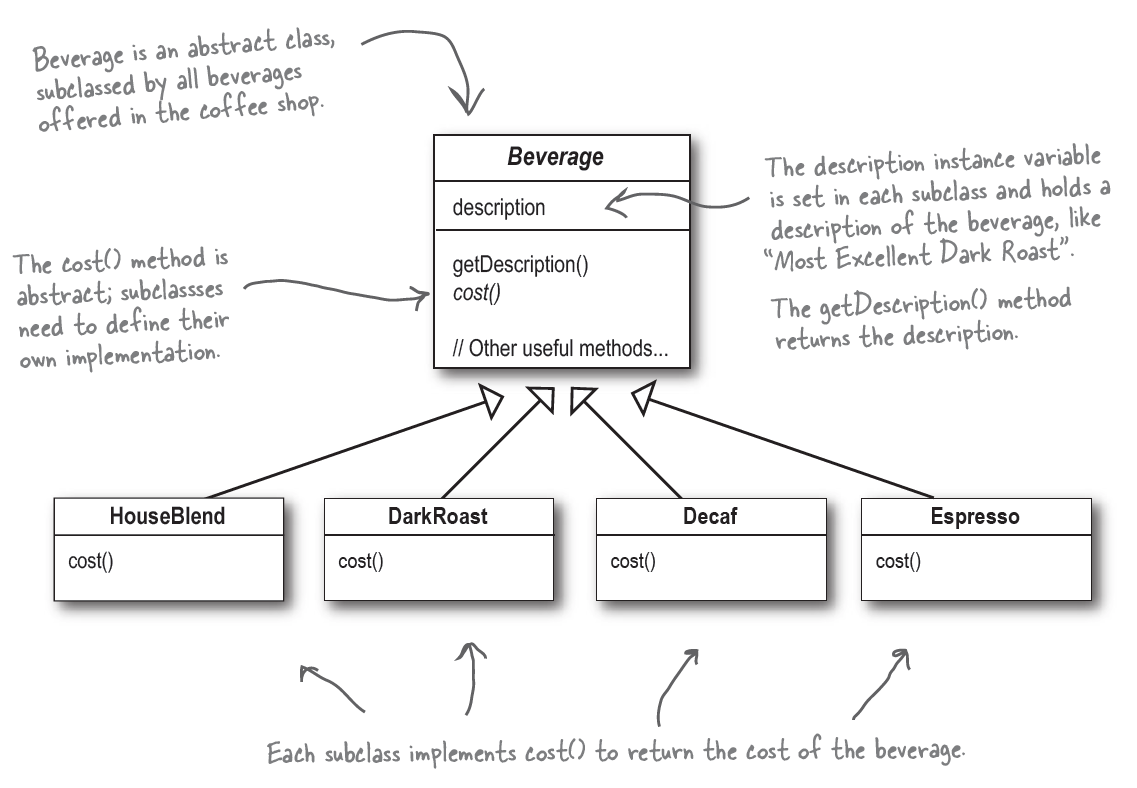




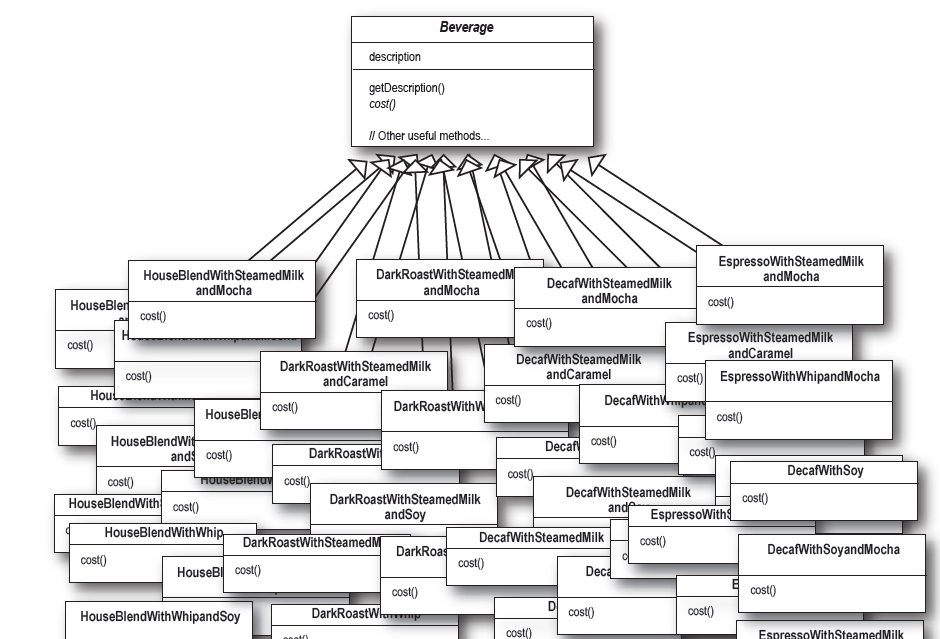
# Decorator Pattern:

Consider a coffee shop…design the classes you would need for them…

Problem: Inheritance…



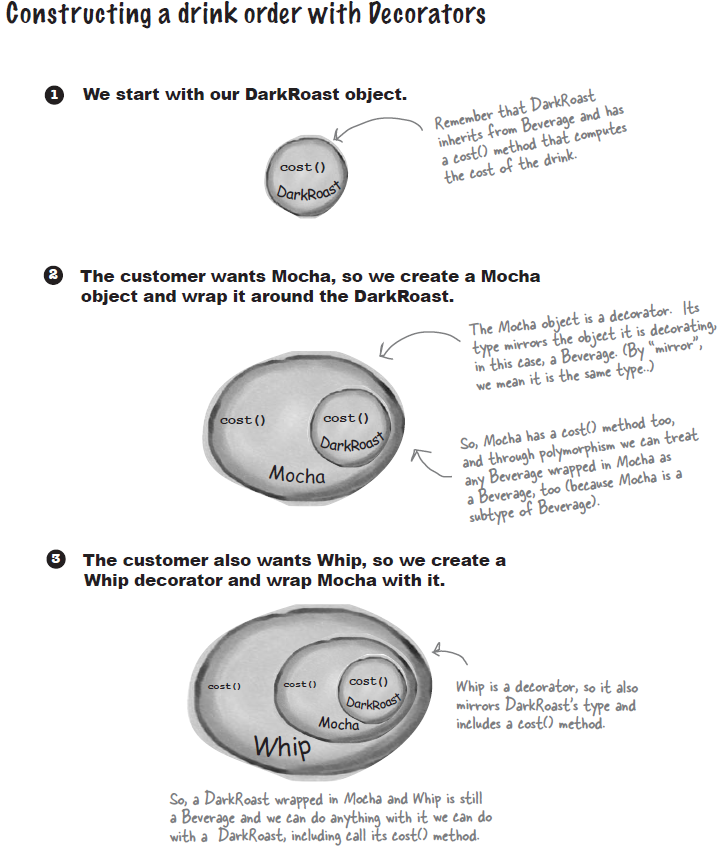
Adding additional condiments, there will be a class explosion… there will be huge no of subclasses when using inheritance.

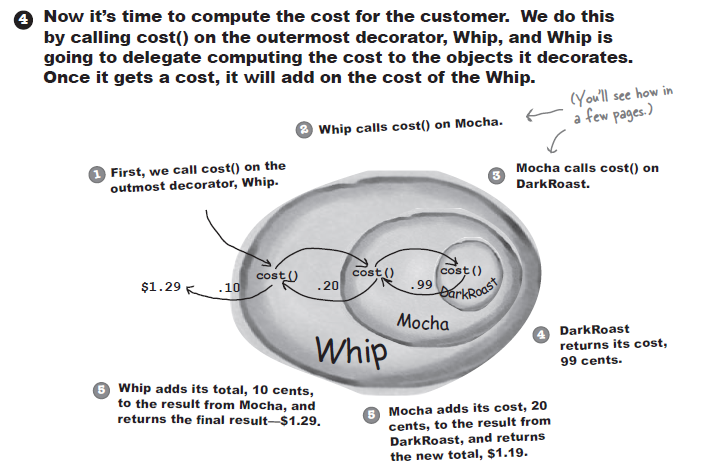


Consider using decorator pattern for the above problem… this can be solved as given below:

if the customer wants a Dark Roast with Mocha and Whip, then we’ll:

1. Take a DarkRoast object
2. Decorate it with a Mocha object
3. Decorate it with a Whip object
4. Call the cost() method and rely on delegation to add on the condiment costs

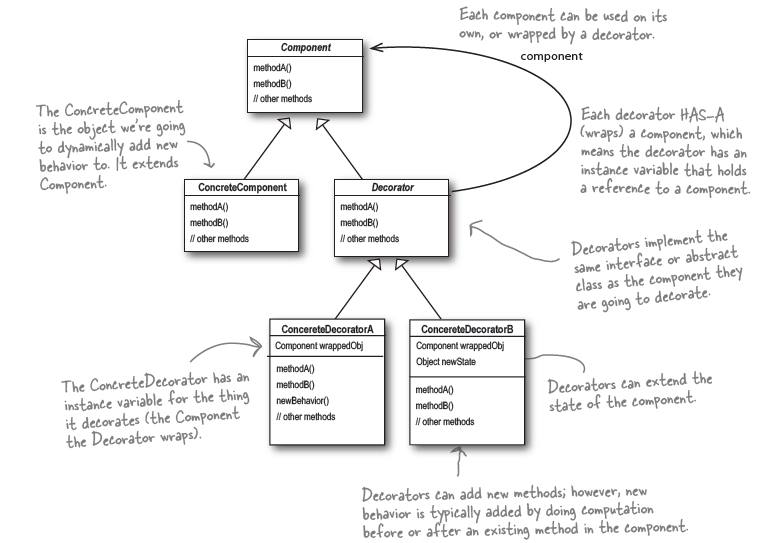




Definition:

The Decorator Pattern attaches additional responsibilities to an object dynamically.

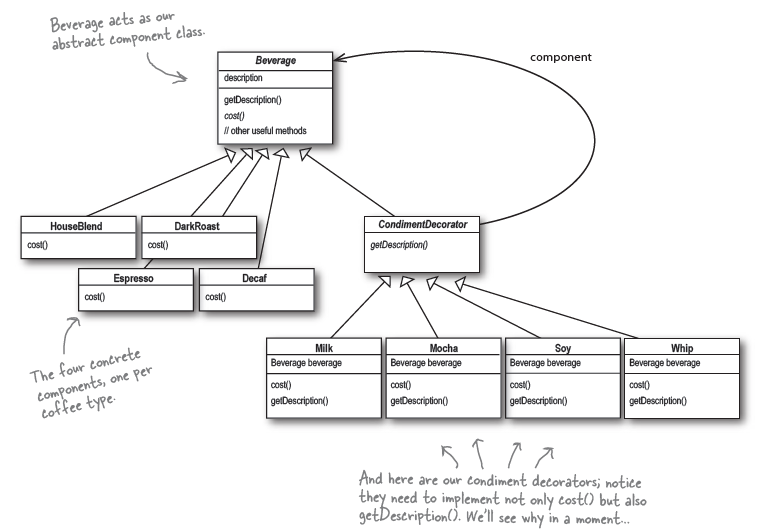
Decorators provide a flexible alternative to subclassing for extending functionality

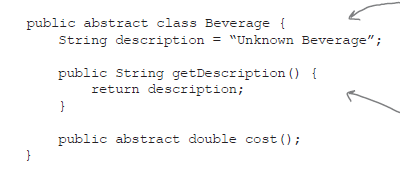


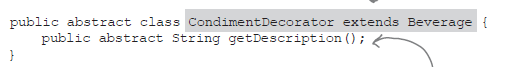
Key points:

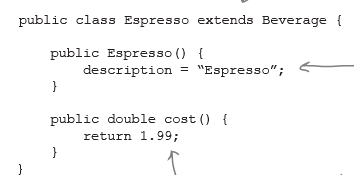
1. The decorator has the same type as the Component – We will subclass ( extend / implement ) the Component class
2. Decorators change the behavior of their components by adding new functionality before and/or after (or even in place of) method calls to the component.
3. It basically uses composition and delegation to add new behaviors at runtime
4. The Decorator Pattern involves a set of decorator classes that are used to wrap concrete components.
5. The Decorator Pattern provides an alternative to subclassing for extending behavior.
6. You can wrap a component with any number of decorators.

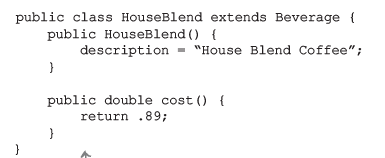
Coffee Shop with decorator pattern:

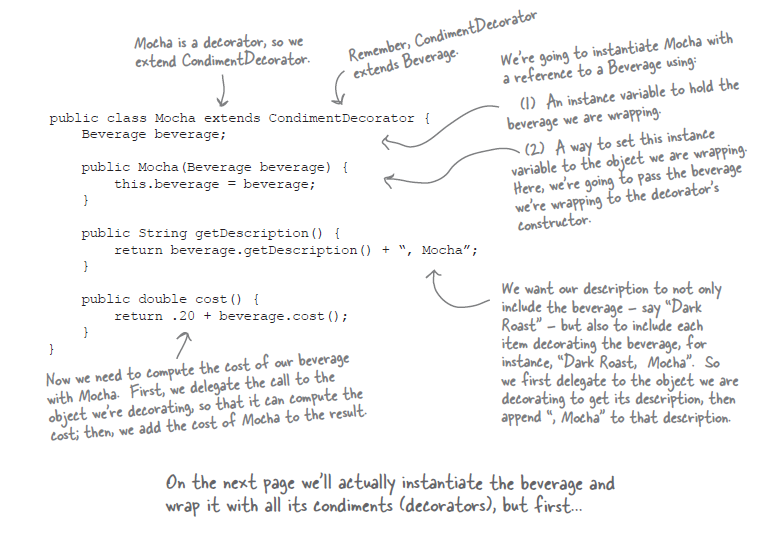


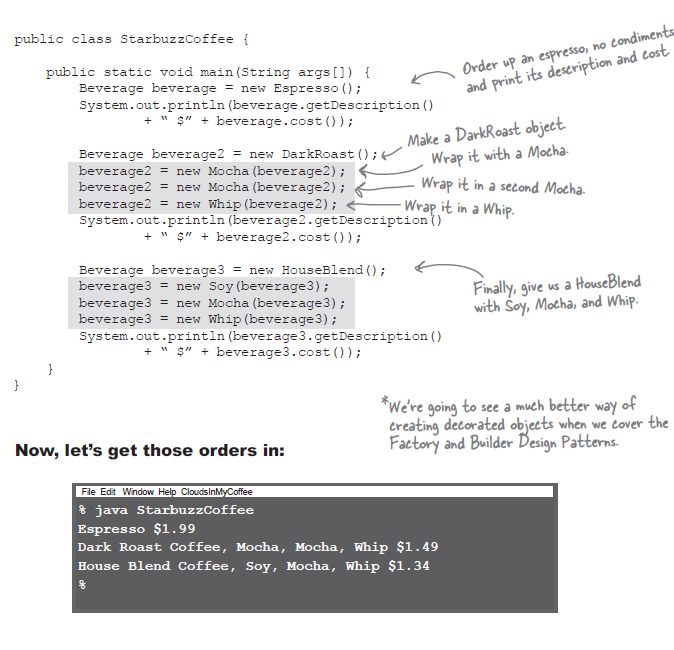












Java Decorators:

