CS342: Software Design

November 21, 2017

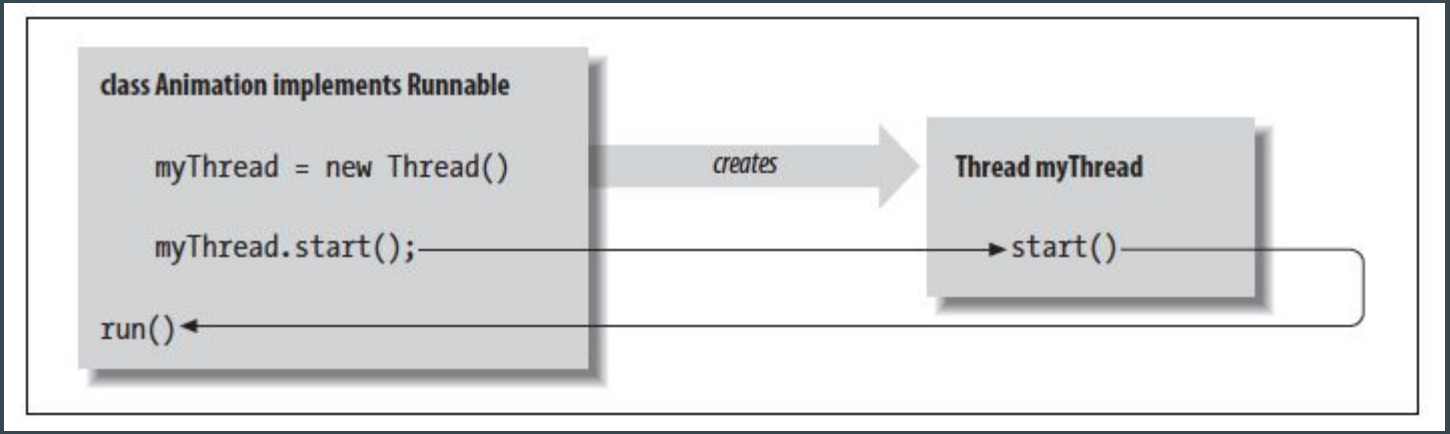
Runnable interface: create “threading” object

- Thread is a flow of control within a program
- Thread vs. process
- All execution in Java is associated with a Thread object. Main thread
- Thread begins by run() method of implementation of Runnable

```
public interface Runnable {
    abstract public void run( );
}
```

```
1  class HelloComponent4 extends JComponent
2      implements MouseMotionListener, ActionListener, Runnable
3  {
4      public HelloComponent4( String message ) {
5          Thread t = new Thread( this );
6          t.start( );
7      }
8      public void run( ) {
9          try {
10         while(true) {
11             blinkState = !blinkState; // Toggle blinkState.
12             repaint( ); // Show the change.
13             Thread.sleep(300);
14         }
15         } catch (InterruptedException ie) { }
16     }
```

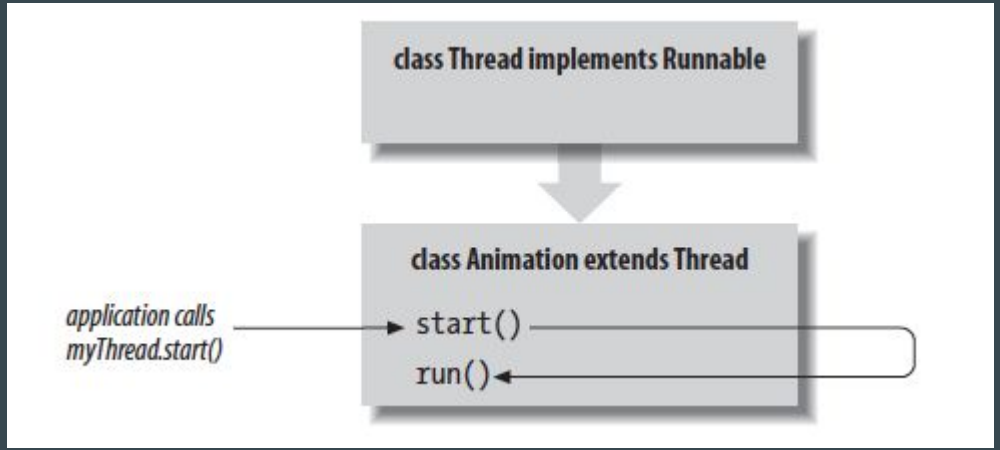
How does this work?



Create a thread by inherit Thread class

This is so convenient, right?

```
class DrawPicture extends Thread {
    boolean keepGoing = true;
    public void run( ) {
        while ( keepGoing ) {
        // do something
        }
    }
}
```



Why do we favor implementing Runnable interface?

By subclassing Thread, the new object is an instance of Thread

- It getting heavy and dirty
- Exposing all public properties and function

I want my class to have ability to thread, not special case of a thread

Implement vs. Inherit

Example: I want to be able to program with Java..

Thread methods and coordination

Deprecated methods

- resume, stop, suspend
- Unexpected behavior, deadlock

sleep() method: make thread to sit idle for specified time

interrupt(): wakes up a thread

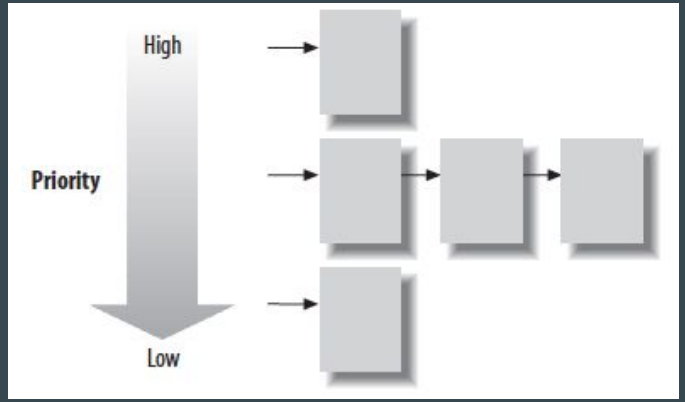
join(): block the caller until thread completes.

yield(), wait(), notify(), notifyAll()

Thread scheduling - round-robin

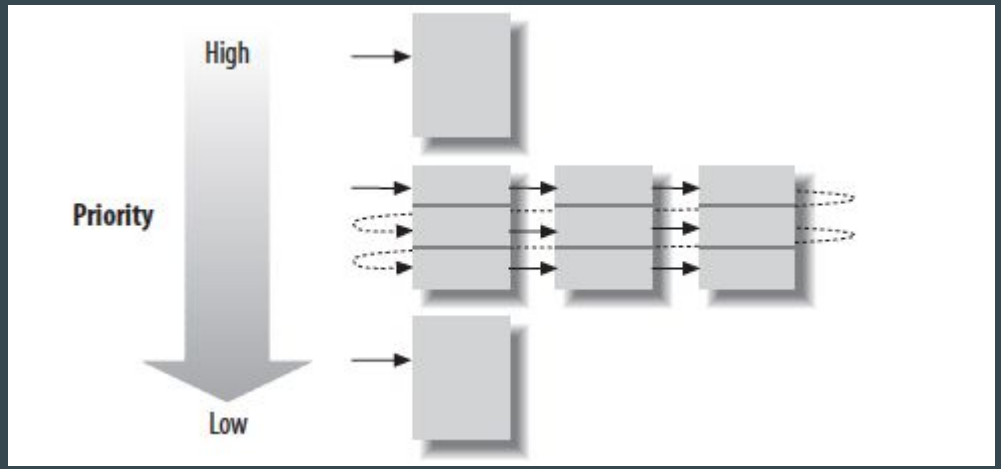
Scheduler picks up next thread when

- Sleeps, by calling Thread.sleep( ) or wait( )
- Waits for a lock, in order to run a synchronized method
- Blocks on I/O, for example, in a read( ) or accept( ) call
- Explicitly yields control, by calling yield( )
- Terminates, by completing its target method or with a stop( ) call (deprecated)



Thread scheduling - time slicing

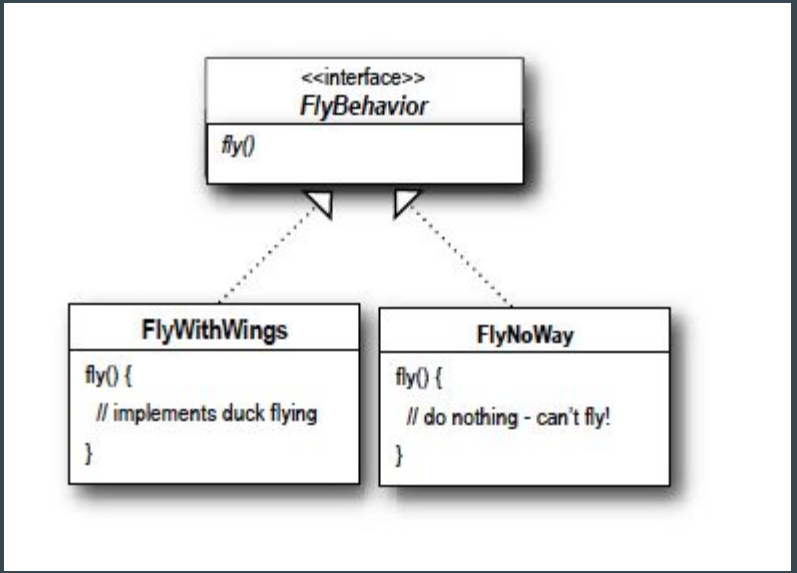
Thread processing is chopped up so that each thread runs for a short period



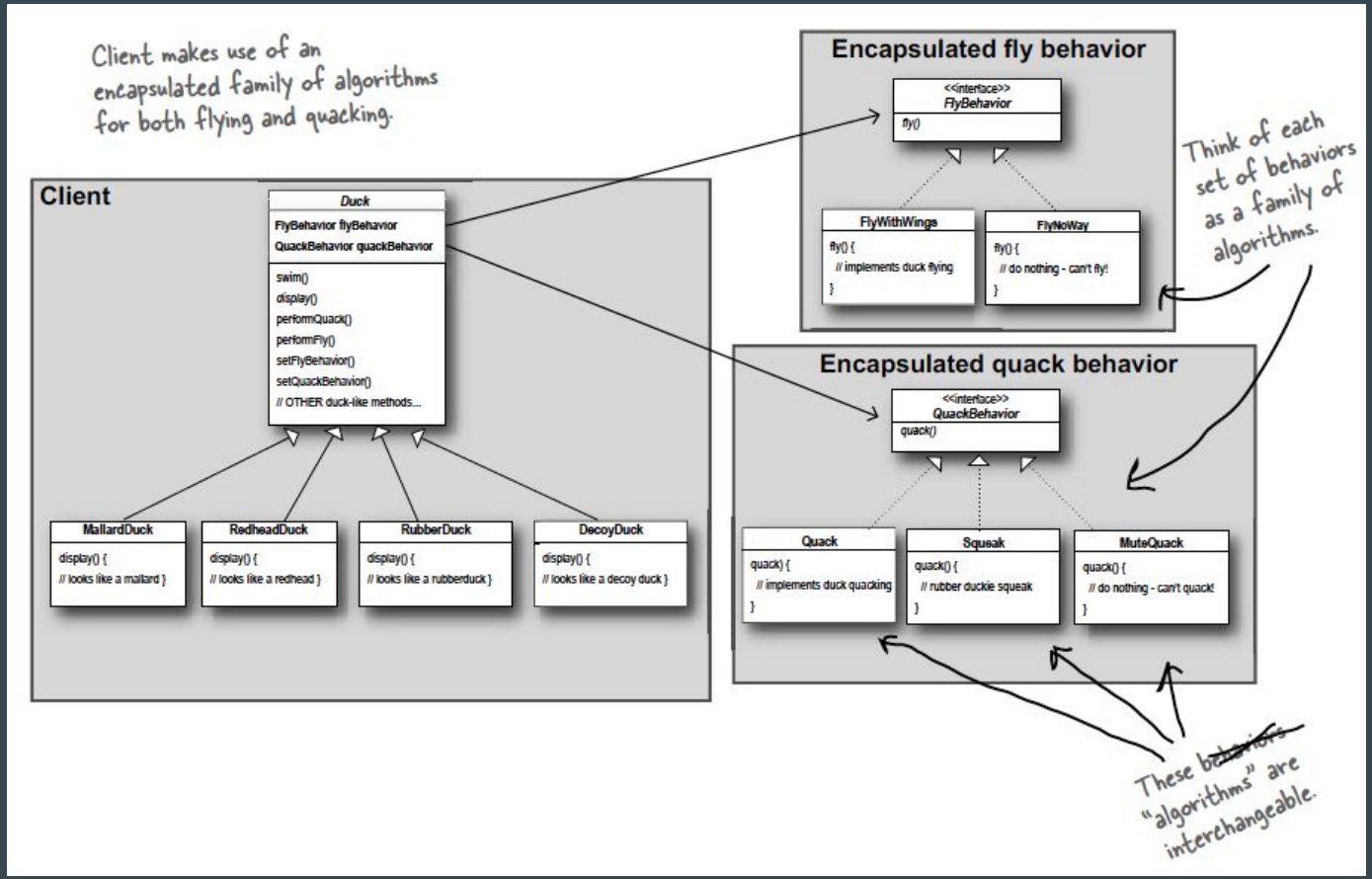
Encapsulation

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

- Separating what changes from what stays the same
- Program to an interface, not to an implementation



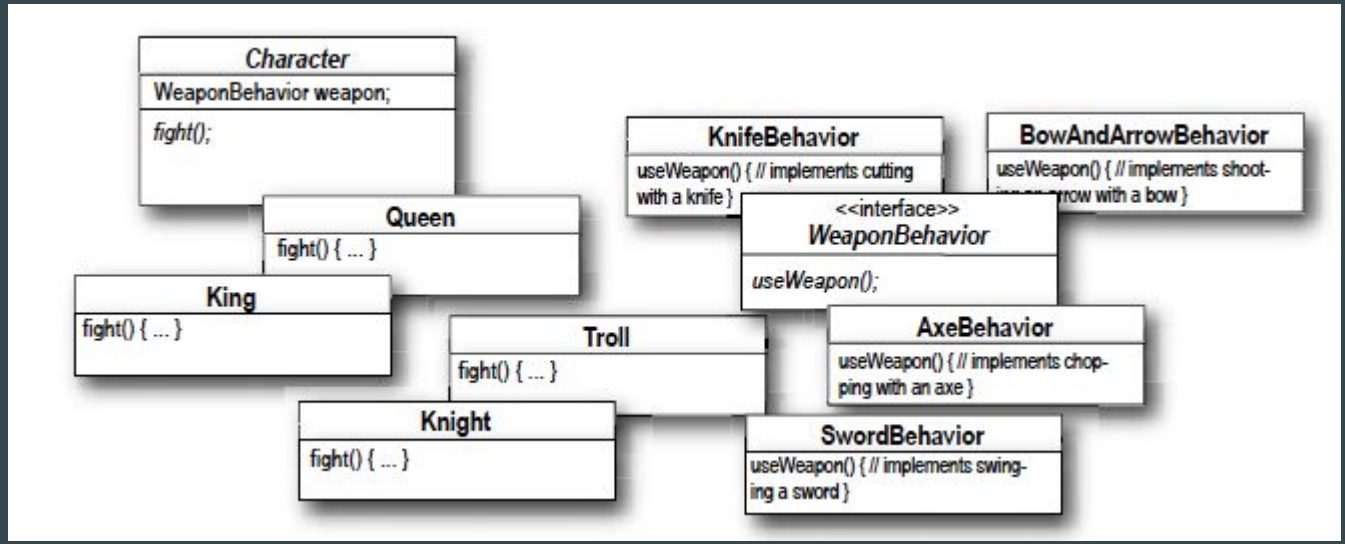
Has-a vs. Is-a


Client makes use of an encapsulated family of algorithms for both flying and quacking.
Client
Duck
FlyBehavior flyBehavior
QuackBehavior quackBehavior
swim()
display()
performQuack()
performFly()
setFlyBehavior()
setQuackBehavior()
// OTHER duck-like methods...
MallardDuck
display() {
// looks like a mallard }
RedheadDuck
display() {
// looks like a redhead }
RubberDuck
display() {
// looks like a rubberduck }
DecoyDuck
display() {
// looks like a decoy duck }
Encapsulated fly behavior
<<interface>>
FlyBehavior
fly()
FlyWithWings
fly() {
// implements duck flying
}
FlyNoWay
fly() {
// do nothing - can't fly!
}
Think of each set of behaviors as a family of algorithms.
Encapsulated quack behavior
<<interface>>
QuackBehavior
quack()
Quack
quack() {
// implements duck quacking
}
Squeak
quack() {
// rubber duckie squeak
}
MuteQuack
quack() {
// do nothing - can't quack!
}
These behaviors "algorithms" are interchangeable.

Strategy pattern

Defines a family of algorithms, encapsulates each one, and makes them interchangeable.

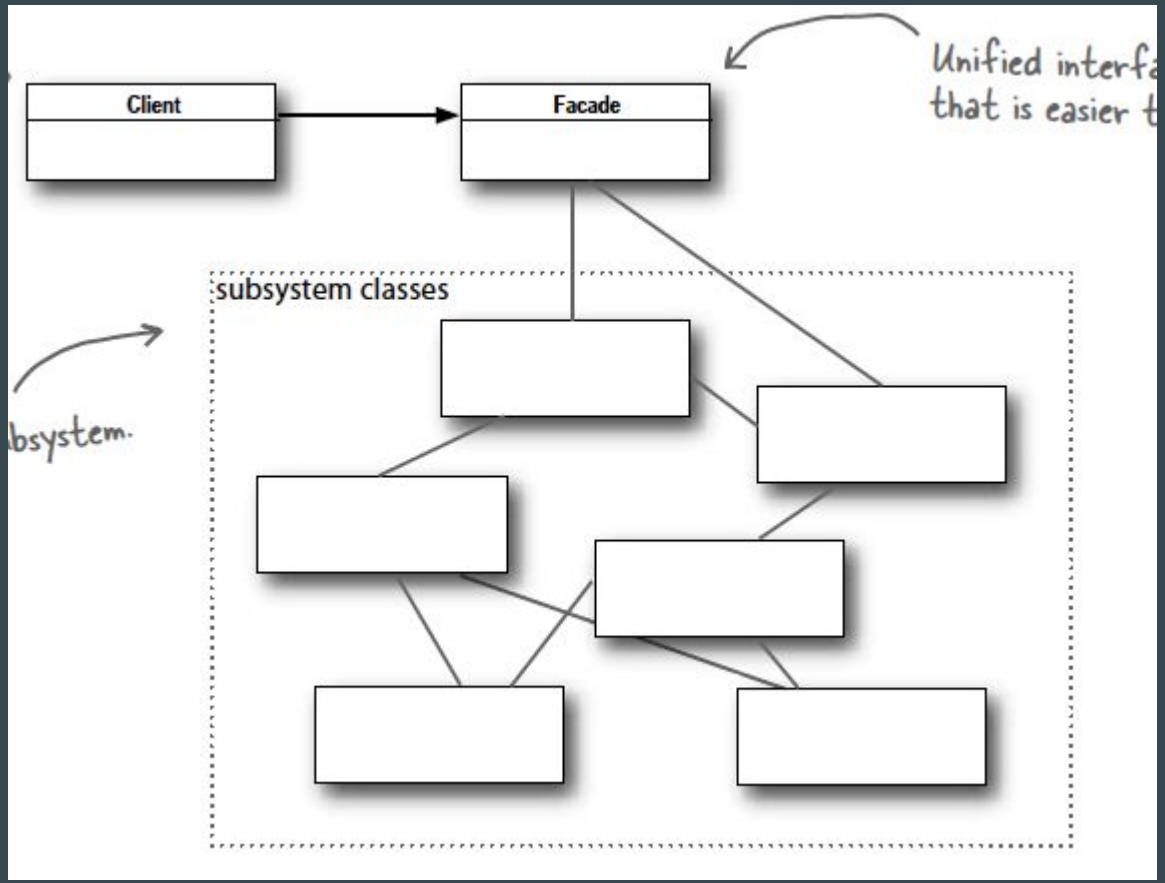
Strategy lets the algorithm vary independently from clients that use it.



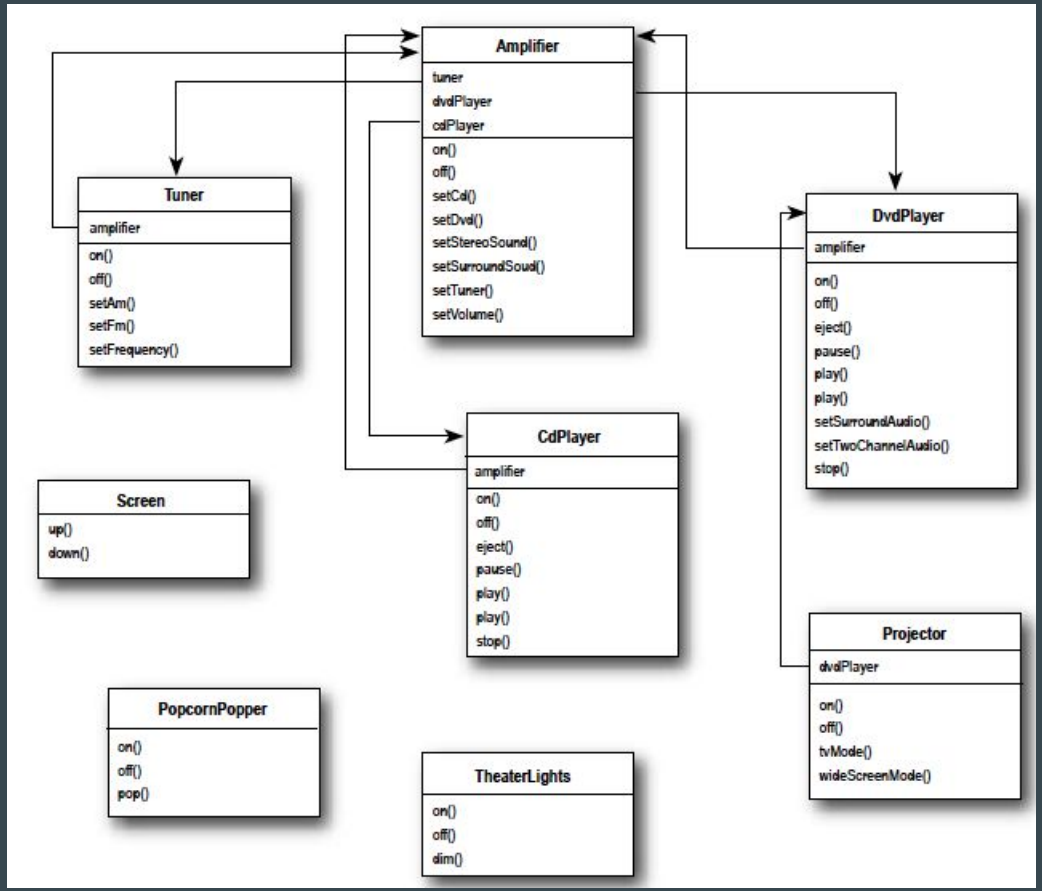
Facade pattern defined

Provides a unified interface to a set of interfaces in a subsystem.

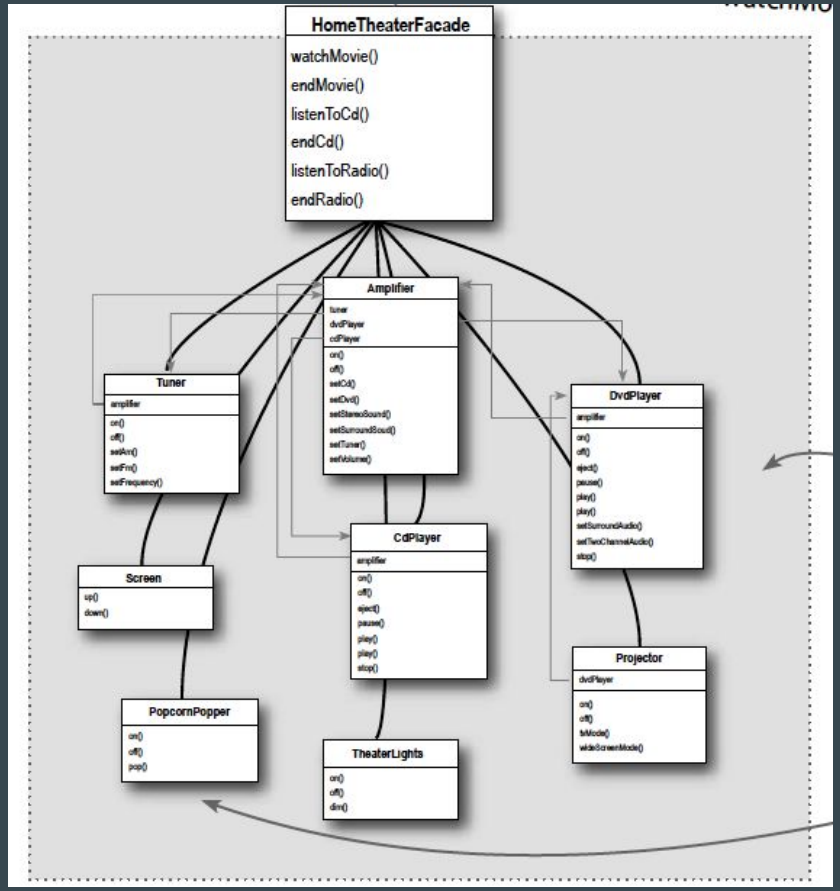
- Avoid tight coupling between client and subsystem
- Better structured and easier to read code



Home theater components



HomeTheaterFacade



Treat components as a subsystem

- Orchestrates functions of components to perform a higher level task
- Provide a simplified interface to client.
- Integrate, not encapsulate: subsystem still available to client
- Decouples client from components

The Principle of Least Knowledge

Talk only to your immediate friends..

- Reduce interactions between objects
- If lots of objects have dependencies with each other, it's hard to make changes, and systems becomes fragile

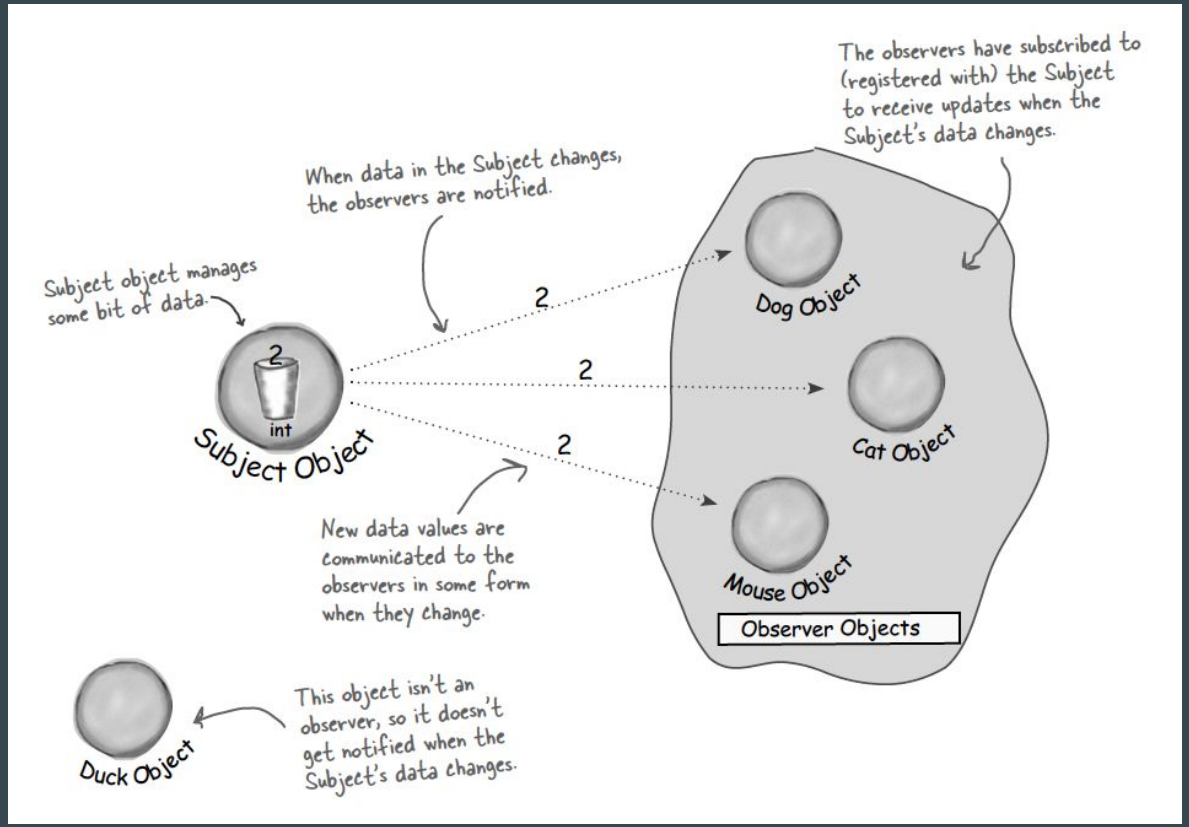
Guideline: an object's method only calls methods that belong to

- The object itself
- Objects passed in as a parameter
- Any objects the method creates
- Any components of the object

Implying don't call methods of an object returned by calling other methods

- Otherwise, we increase the number of objects that the class direct knows

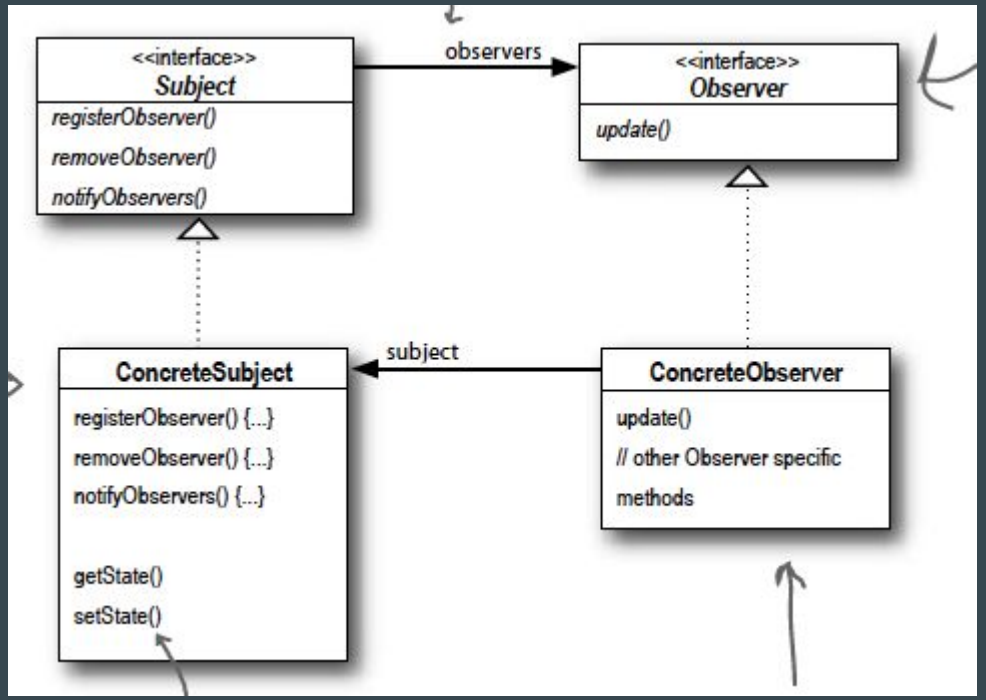
Observer pattern: publish & subscribe



One to many dependency between objects

- When source/subject /publisher state changes, all dependents/observer /subscribers are notified and updated automatically
- There are many different ways to implement

Class diagram

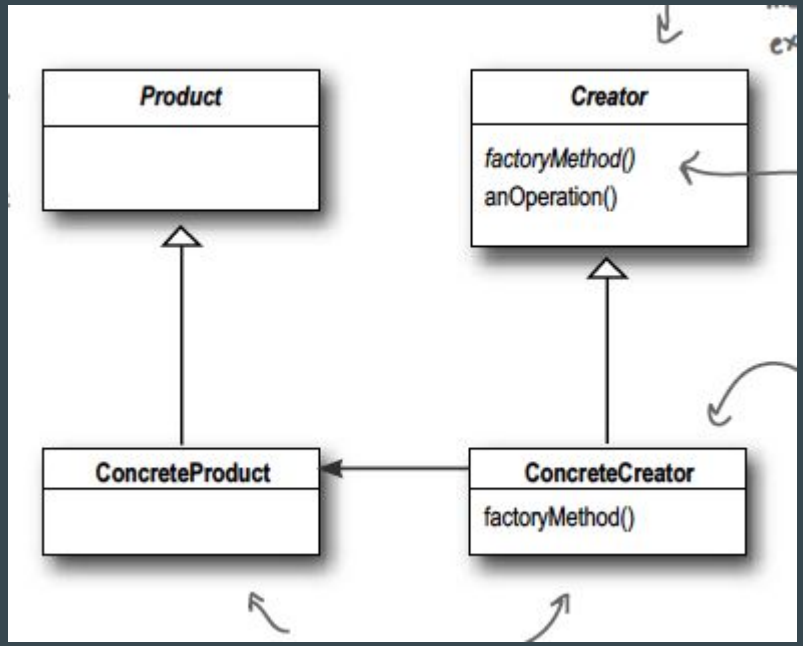


Loosely coupled objects

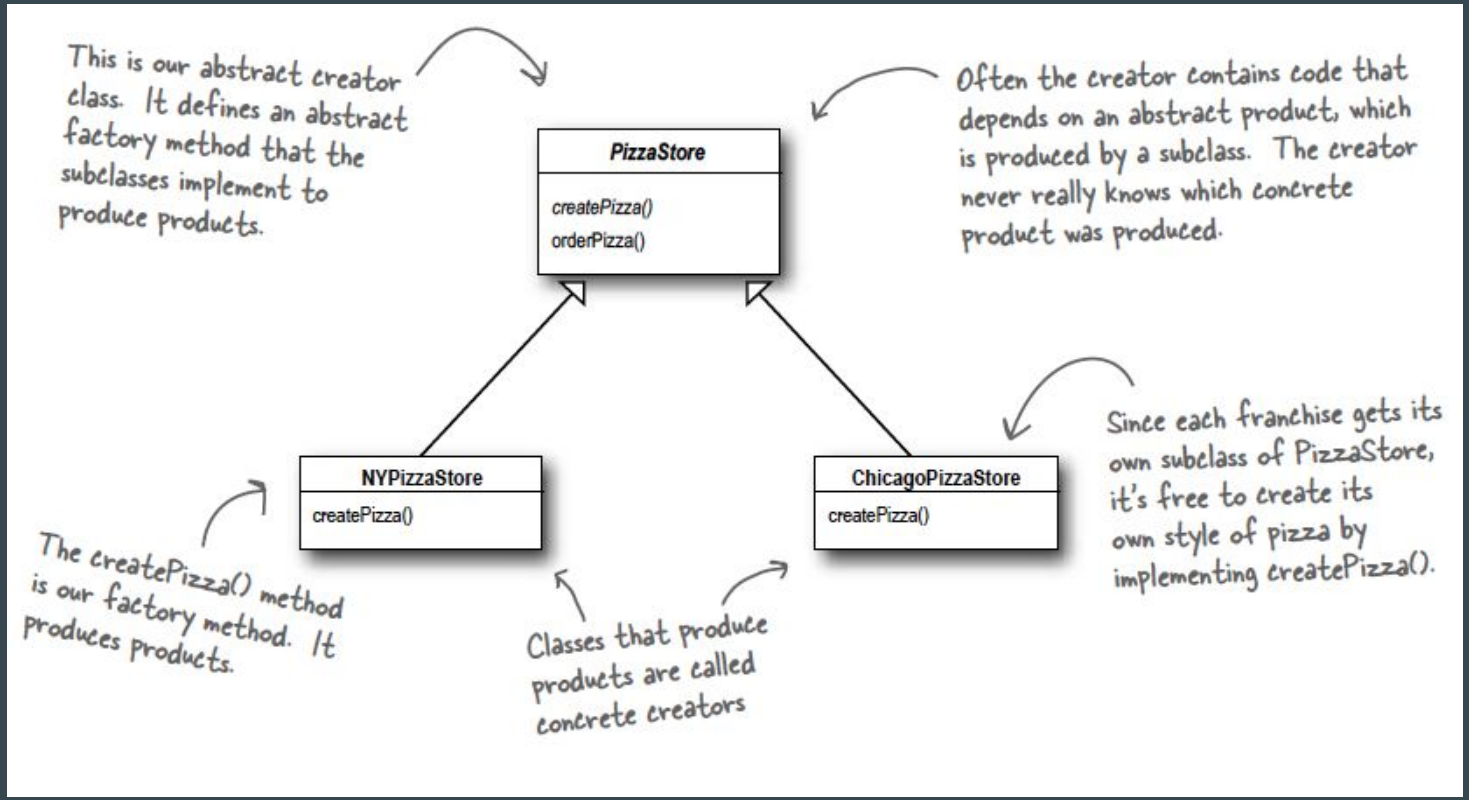
- Subjects and observers barely know each other
- A subject class only implements the interface, it doesn't know or care who subscribe to it
- Observer can be added any time
- Reuse subjects and observers
- Changing either doesn't affect the other

Factory pattern definition

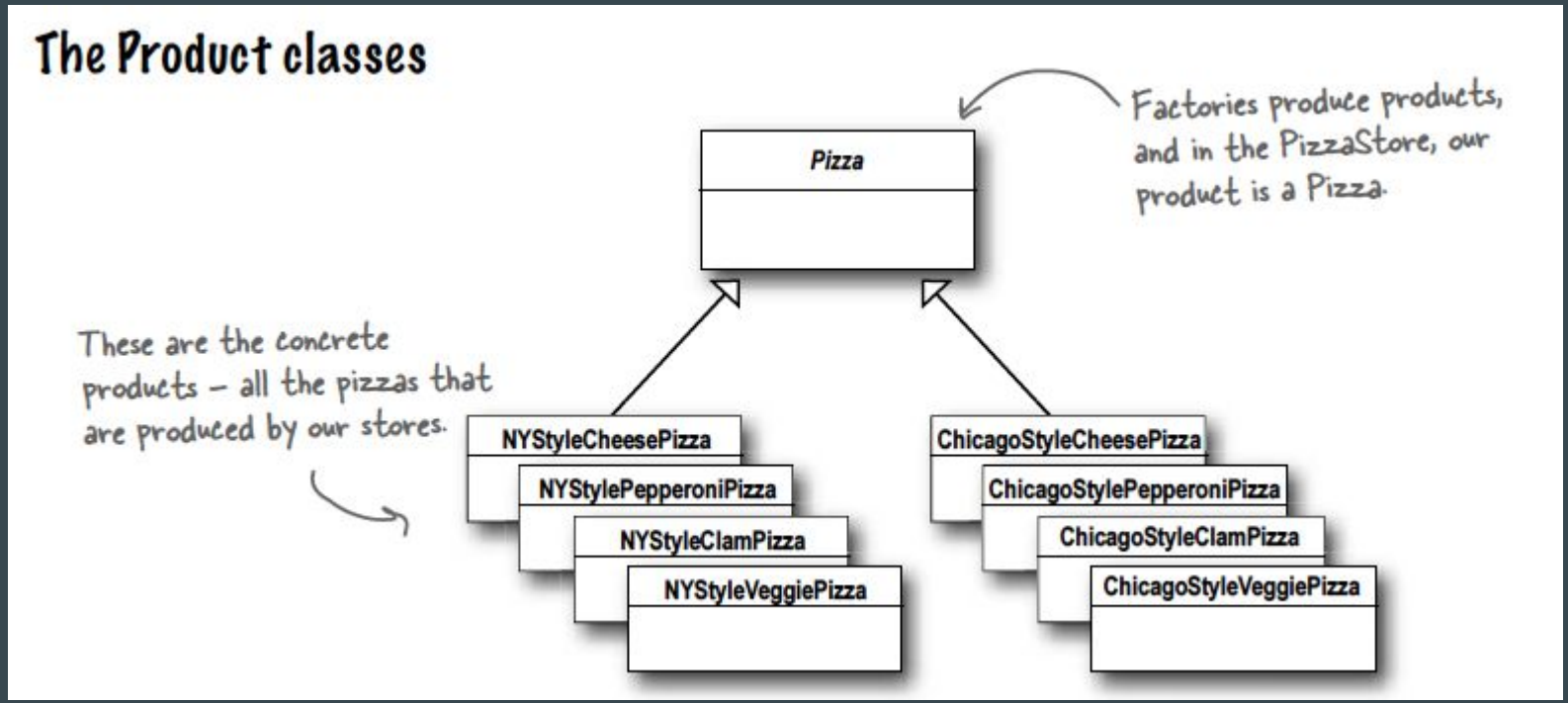
The Factory Method Pattern defines an interface for creating an object, but lets subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses



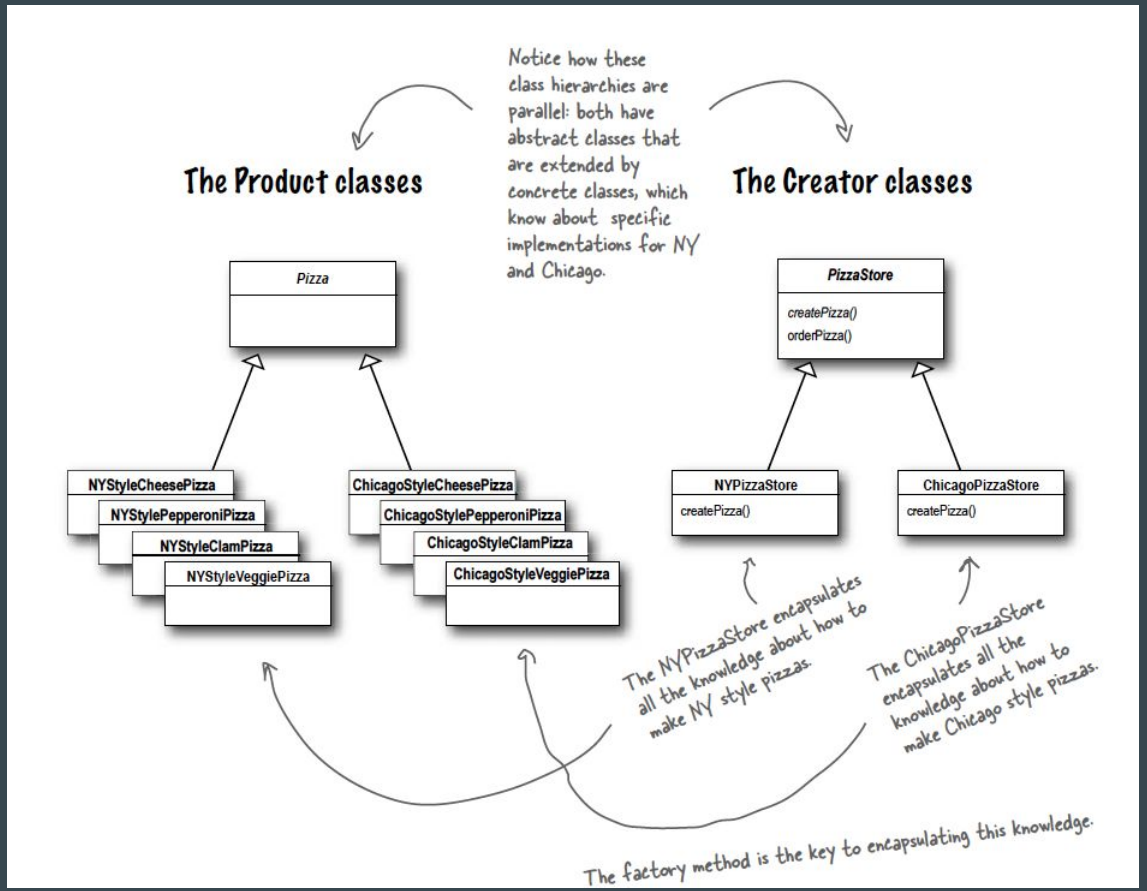
Factory (creator) classes



Product class



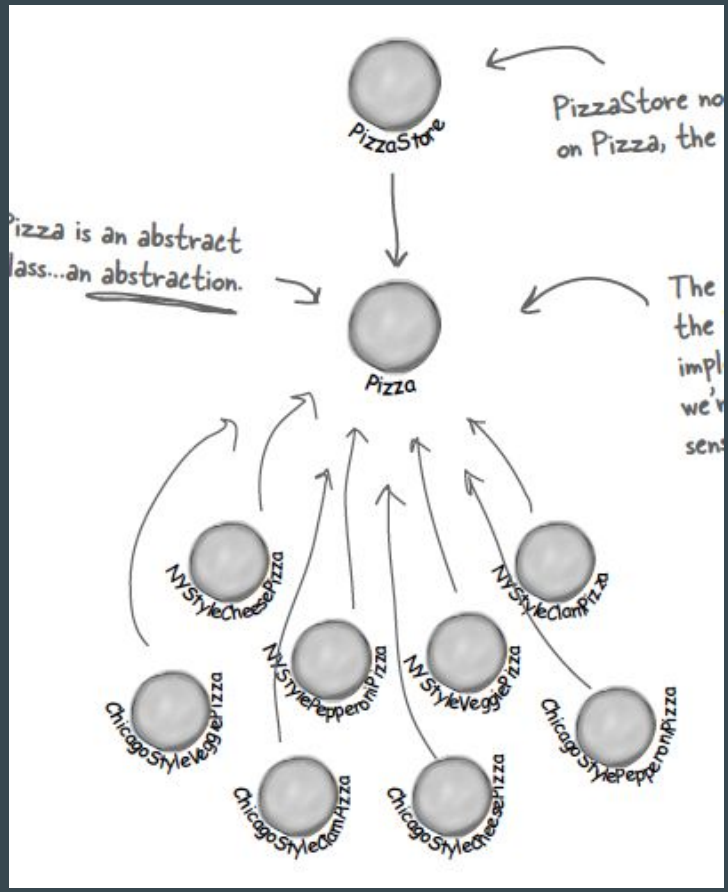
Parallel class hierarchy



The Dependency Inversion Principle

Depend upon abstractions. Do not depend upon concrete classes.

- Further step beyond “Program to an interface, not to an implementation”
- High-level components should not depend on low-level components
- Both high level and low level components should depend on abstractions
- Both PizzaStore and concrete pizza classes depend on Pizza abstraction



The Open-Closed Principle

Classes should be open for extension, but closed for modification

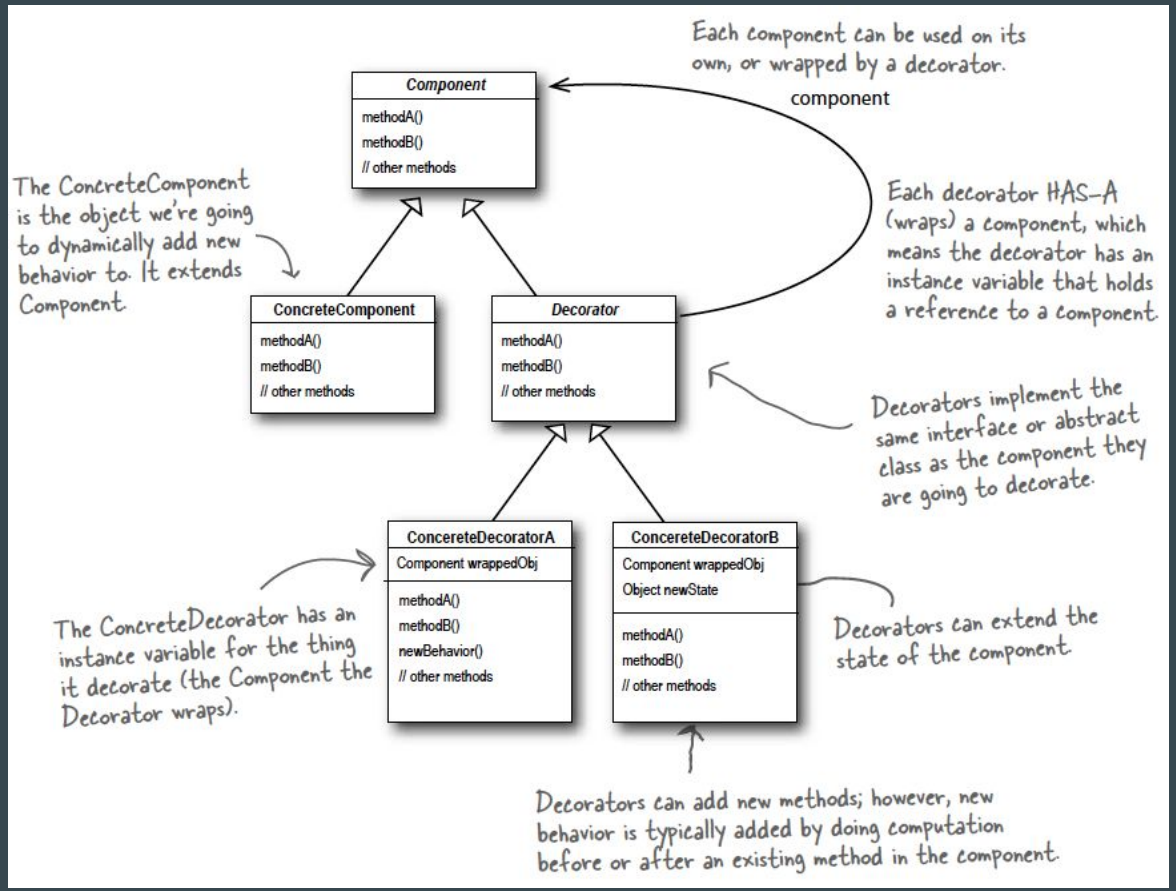
- Allow classes to be easily extended to incorporate new behavior without modifying existing code
- Make it resilient to change and flexible to take new functionality to meeting changing requirements

There are multiple OOP techniques to implement the principle.

- Observer pattern: add/remove observers

Decorator pattern

- The Decorator Pattern attaches additional responsibilities to an object dynamically.
- Decorators provide a flexible alternative to subclassing for extending functionality



Decorators

- Decorators have the same supertype as the objects they decorate.
- You can use one or more decorators to wrap an object.
- Given that the decorator has the same supertype as the object it decorates, we can pass around a decorated object in place of the original (wrapped) object.
- The decorator adds its own behavior either before and/or after delegating to the object it decorates to do the rest of the job.
- Objects can be decorated at any time, so we can decorate objects dynamically at runtime with as many decorators as we like.

Singleton pattern: Create one-of-a-kind objects

There is only one instance of the class

- Ensures a class has only one instance, and provides a global point of access to it.
- Create a class and letting it manage a single instance of itself, preventing any other class from creating a new instance on its own
- Whenever you need an instance, just query the class and it will hand you back the single instance
- Usage: thread pools, caches, dialog boxes, objects that handle preferences and registry settings, objects used for logging, and objects that act as device drivers to devices like printers and graphics cards

Singleton benefits and risk

There is only one instance of the class

- Better resource usage: you don’t have to create a new object when you need it and then destroy it after you are done
- Performance gain
- Commonly used in SOA: instantiate a service
- One of a kind means clients may fight to use it!
- Thread-safe?

How to make singleton thread-safe?

1. Use synchronized modifier -- expensive, decrease performance 100x. Use it only when ‘getInstance()’ isn’t critical
2. Eagerly create instance.

```
public class Singleton {
    private static Singleton uniqueInstance = new Singleton();
    private Singleton() {}
    public static Singleton getInstance() {
        return uniqueInstance;
    }
}
```

3. Double check locking: First check to see if an instance is created, and if not, THEN use synchronizes (see next page).

Double check locking

```
public class Singleton {
   private volatile static Singleton uniqueInstance;
   private Singleton() {}
   public static Singleton getInstance() {
   if (uniqueInstance == null) {
      synchronized (Singleton.class) {
         if (uniqueInstance == null) {
            uniqueInstance = new Singleton();
         }
      }
   }
   return uniqueInstance;
   }
}
```

“volatile” ensures multiple threads handle the uniqueInstance variable correctly when it is being initialized to the Singleton instance.

Iterator pattern

Provides a way to access the elements of an aggregate object sequentially without exposing its underlying representation.

- Allows traversal of the elements of an aggregate without exposing the underlying implementation
- Places the task of traversal on the iterator object, not on the aggregate, which simplifies the aggregate interface and implementation, and places the responsibility where it should be

Single responsibility principle

A class should have only one reason to change

- When design class, assign it with one responsibility and one only
- A class should only do one thing and do it well

Cohesion: a measure of how closely a class or a module supports a single purpose or responsibility

- How closely methods/functions of a class are related
- High cohesion class == adhering to single responsibility principle

Single responsibility principle: intra-class structure; Least knowledge : inter-class

- High cohesion, low coupling == good design