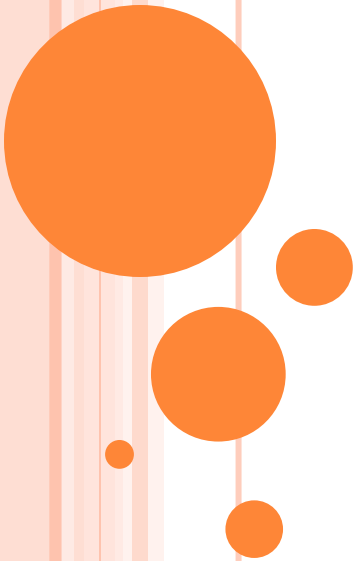


# Design Patterns

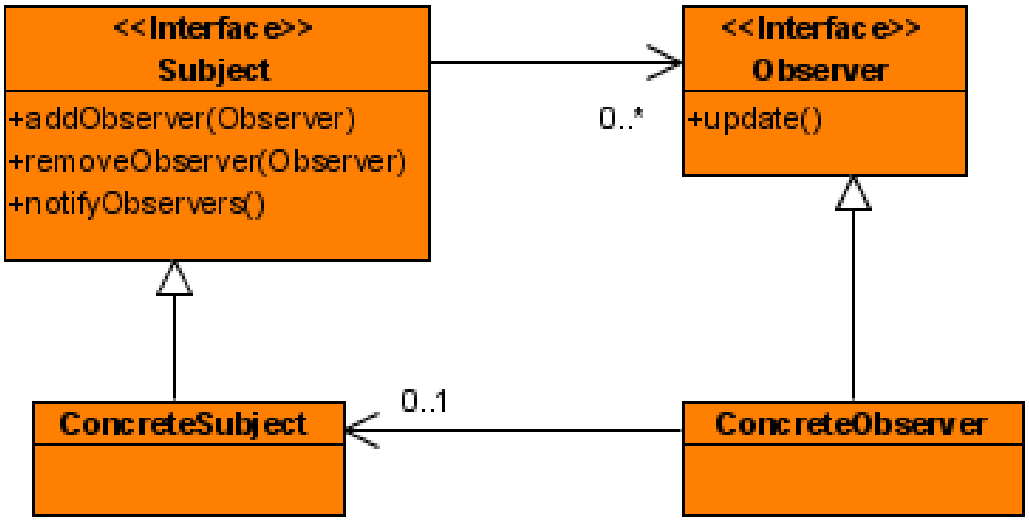
Presented By: Nick Butt

# Open-Closed Systems

- An open-closed system is when you working code is open to extension and closed from modification.
- Most all design patters are used to abide to this.
- So, even though in some instances it may be easier to modify exiting code then use a design pattern you run the risk of unexpected behavior from you so called “working” code!!

# The Observer Pattern

- Defines a one-to-many relationship between a set of objects.
- When the state of one object changes all its dependents are notified.



# The Observer Pattern (Cont…)

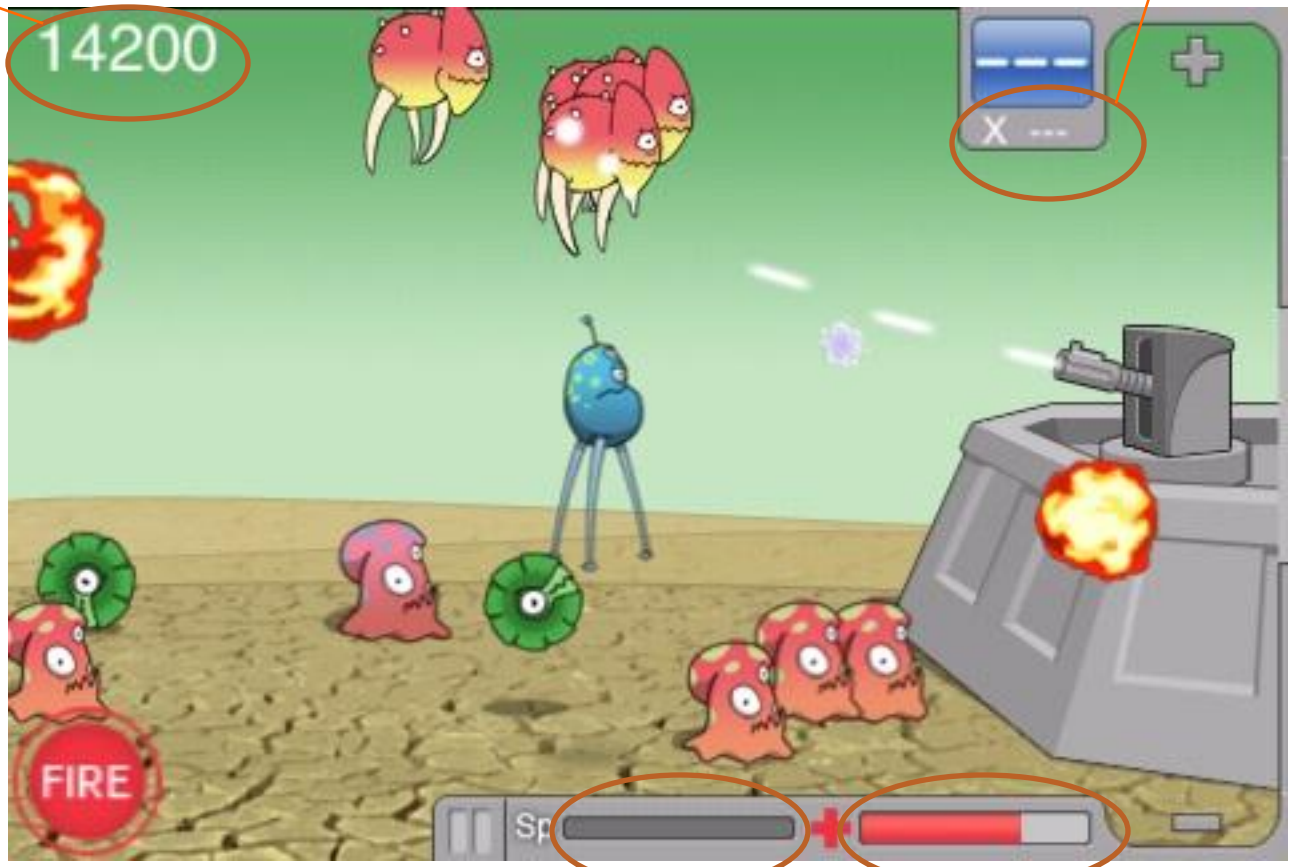
- Lets say you have an program that gives multiple different representation of data.
  - Instead of all representation knowing the data and constantly updating themselves, The data will update the representations whenever the data changes.

Subject: Data Observers: Representations

# Example

Level Score

Ammo Count

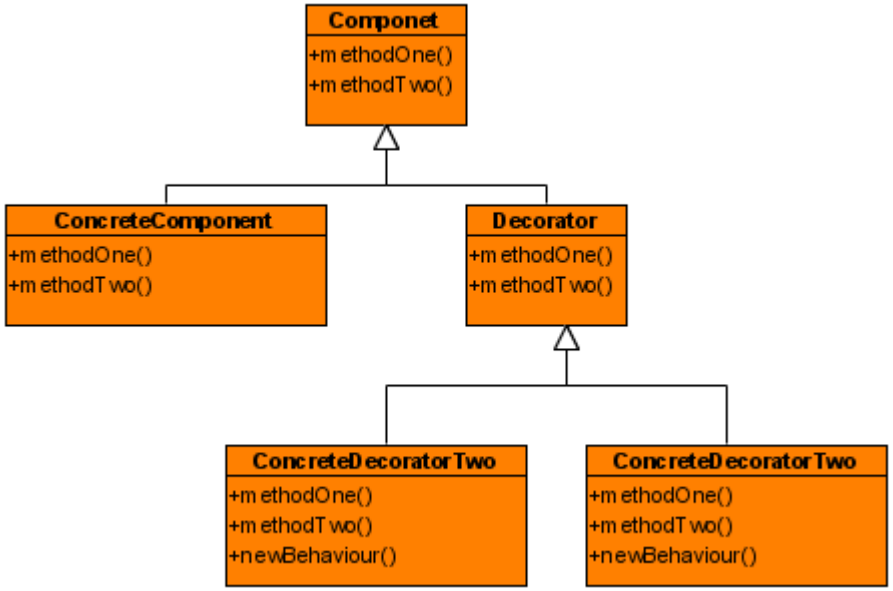



Special Bar

Health Bar

# Decorator Pattern

- Attaches additional responsibilities to an object dynamically.
- Provides a flexible alternative to subclassing for extending functionality.



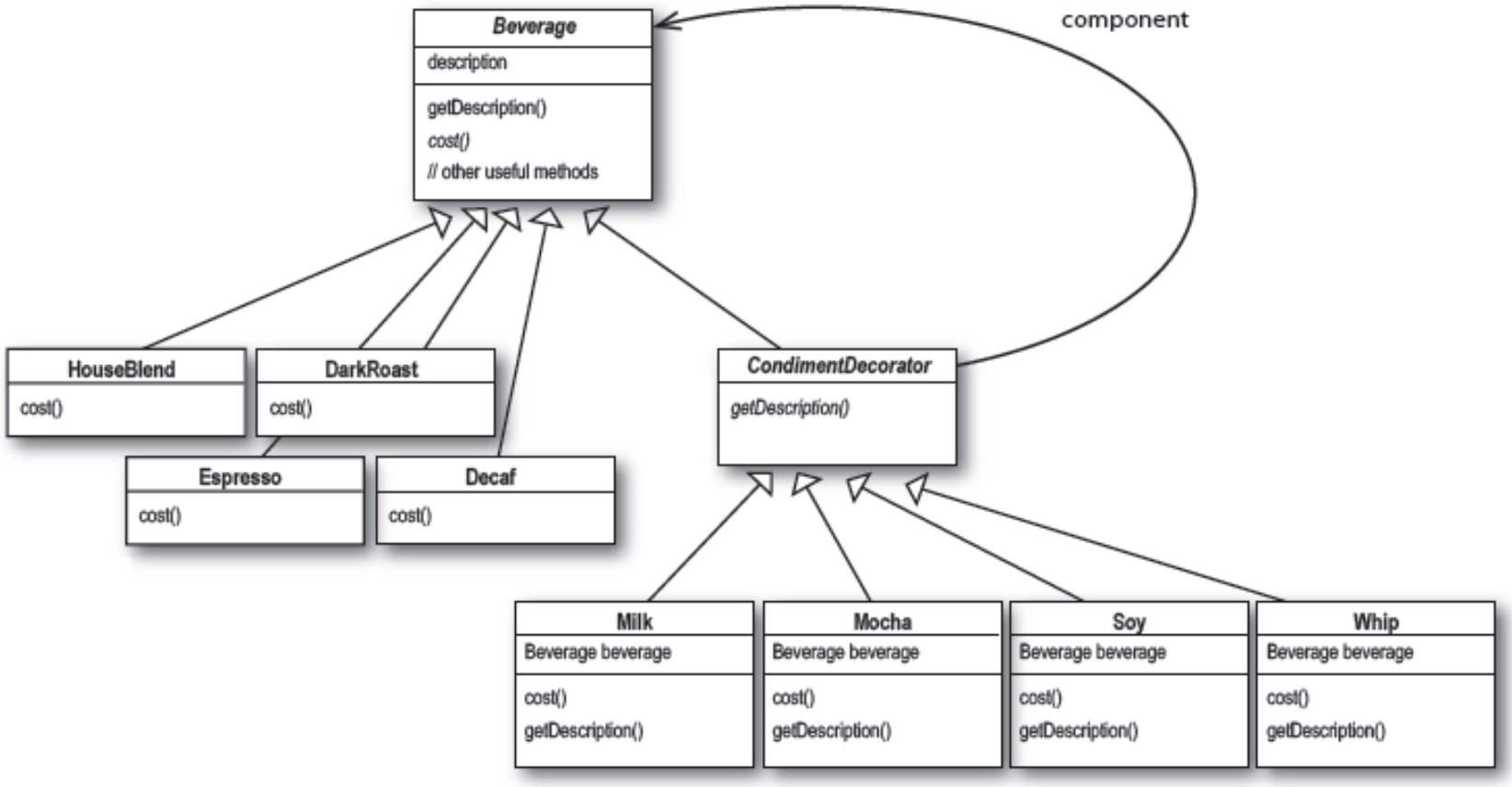
# Decorator Pattern (Cont…)

- Lets say you had to design an program for the cash register at specialty coffee shop. You have multiple types of drinks each need to be uniquely represented in the software.
  - Instead of all representing all the drinks individually you can represent each ingredient and use the decorator pattern to resolve the cost.

Component: Espresso, Decaf, Dark Roast, etc . . .

Decorators: Steamed Milk, Cream, Mocha, Cinnamon, Caramel, Vanilla, Soy ,etc . . .

# Example


Beverage
description
getDescription()
cost()
// other useful methods
component
HouseBlend
cost()
DarkRoast
cost()
Espresso
cost()
Decaf
cost()
CondimentDecorator
getDescription()
Milk
Beverage beverage
cost()
getDescription()
Mocha
Beverage beverage
cost()
getDescription()
Soy
Beverage beverage
cost()
getDescription()
Whip
Beverage beverage
cost()
getDescription()

# Lets Make A Drink

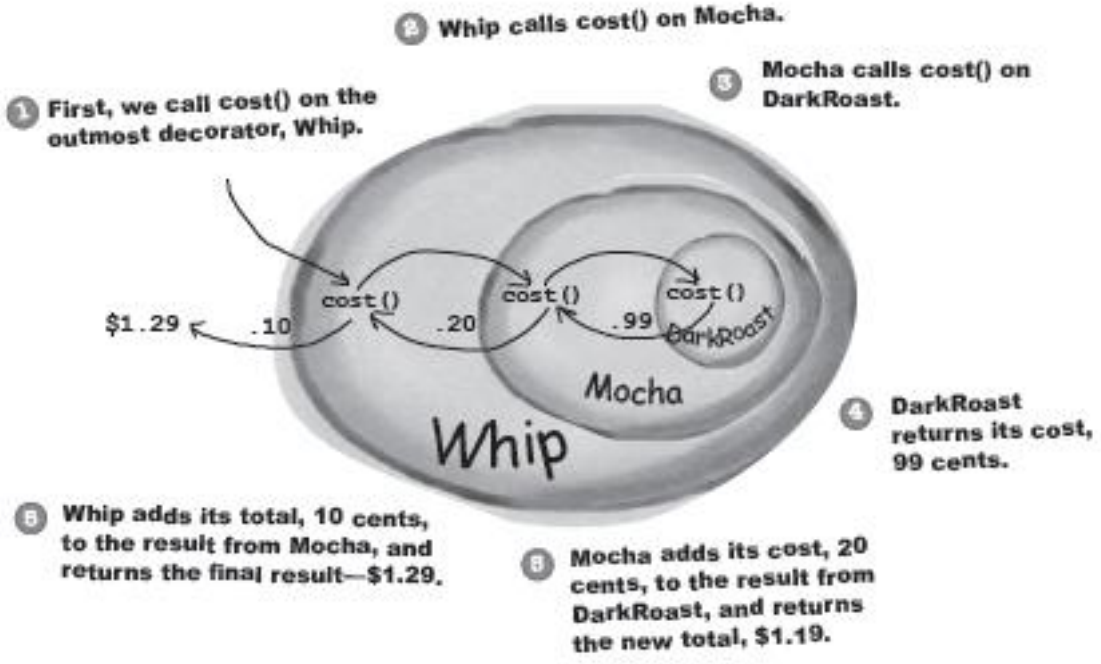
```
Beverage beverage2 = new DarkRoast();
beverage2 = new Mocha(beverage2);
beverage2 = new Mocha(beverage2);
beverage2 = new Whip(beverage2);
System.out.println(beverage2.getDescription()
        + " $" + beverage2.cost());
```

Make a DarkRoast object.

Wrap it with a Mocha.

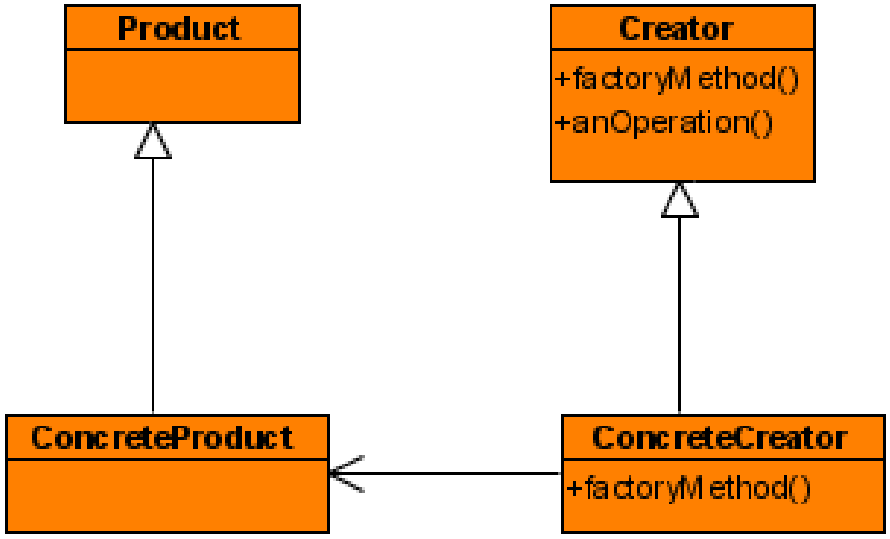
Wrap it in a second Mocha.

Wrap it in a Whip.

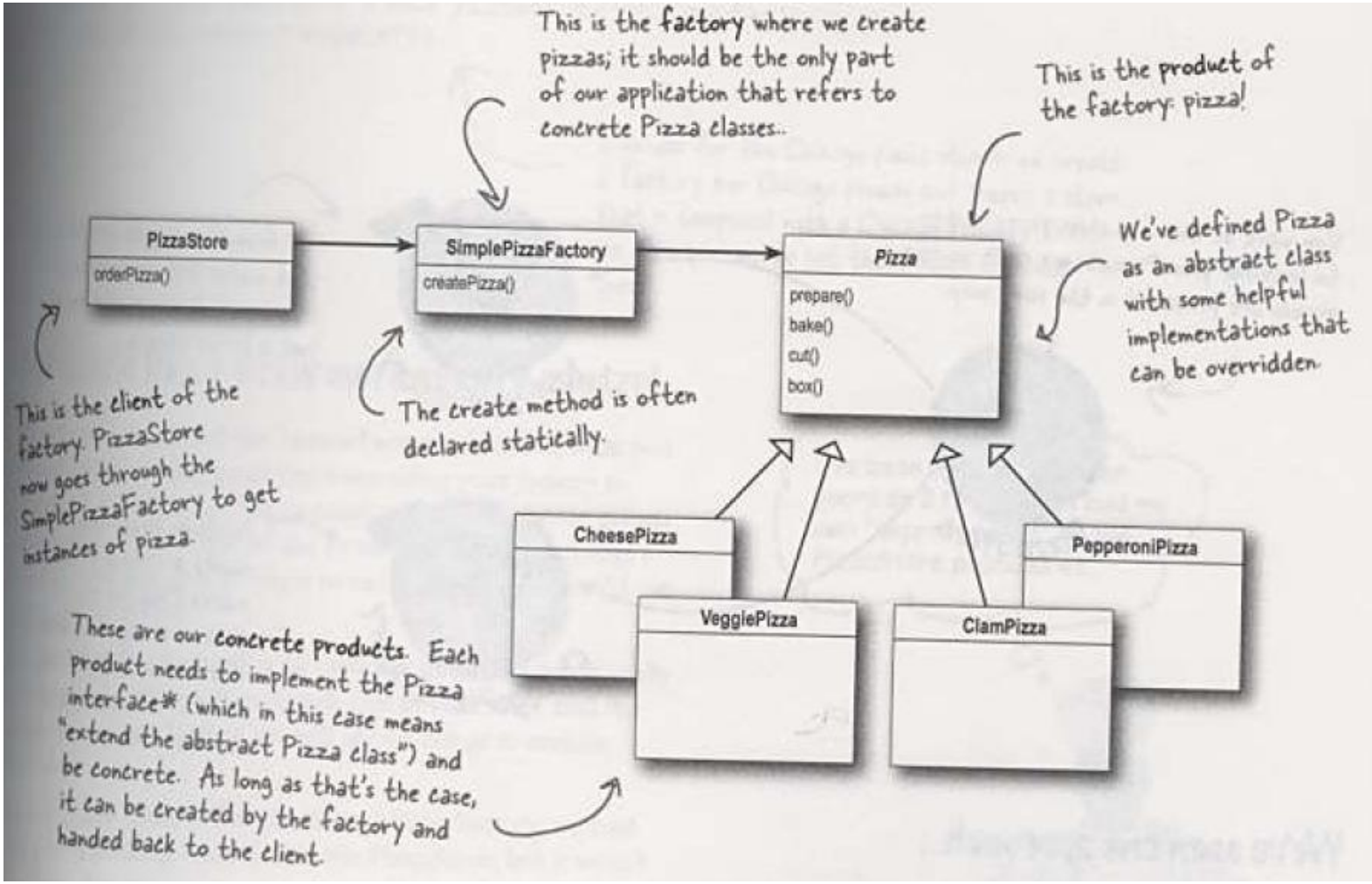


# Factory Pattern

- Provides an interface for creating families of related or dependent objects without specifying their concrete class.

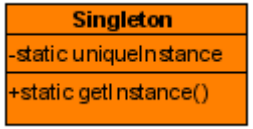


# Factory Pattern (Cont…)



# Singleton Pattern

- Ensures a class only has one instance, and provide a global point of access to it.



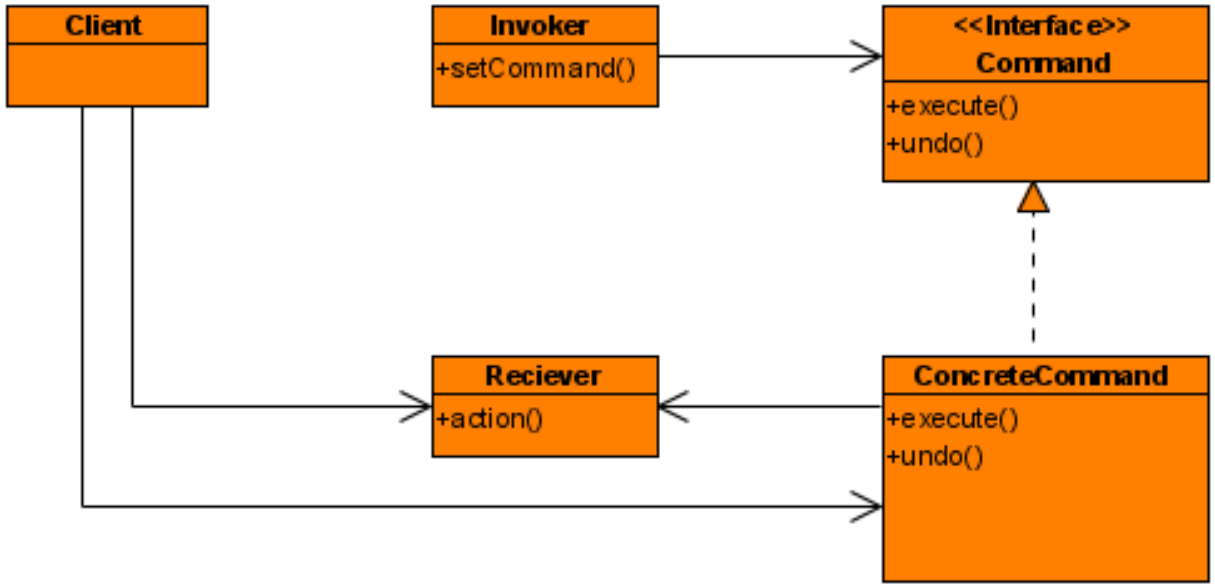
# Singleton Pattern (Cont…)

- When would this be useful?

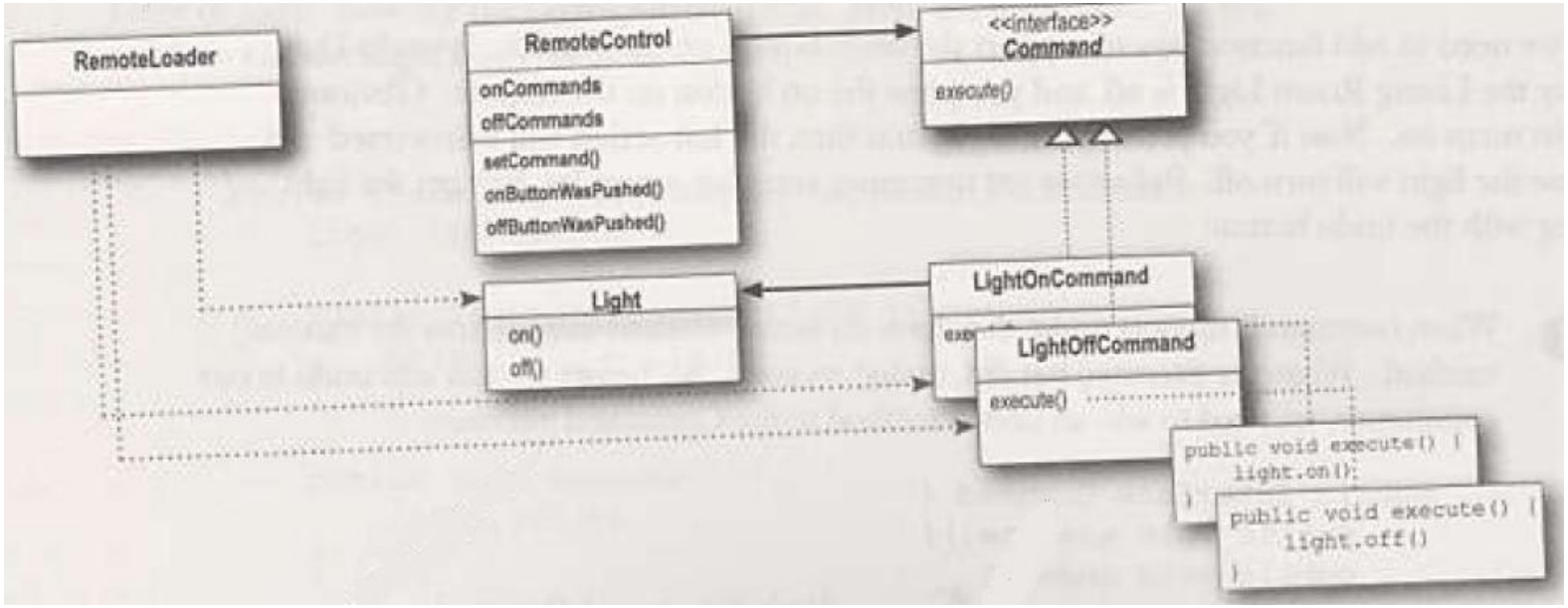

- Sound and Game Manager.
  - Using singleton pattern on the soundManager and gameManager only one instance will exist and can be access from anywhere in the project.

# Command Pattern

- Encapsulates a request as an object, thereby letting you parameterize other objects with different responses, queues or log requests, and support undoable operations.



# Command Pattern (Cont…)




Receiver : Light
Commands: LightOn, LightOff
Invoker: Remote

# How to Undo?

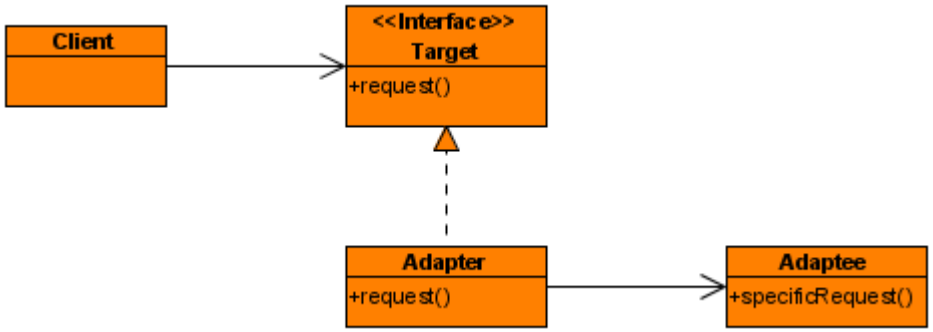
- Controller has a setCommand() method which knows what command to execute. So,

```
public class LightOnCommand implements Command {
    Light light;

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

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

    public void undo() {
        light.off();
    }
}
```

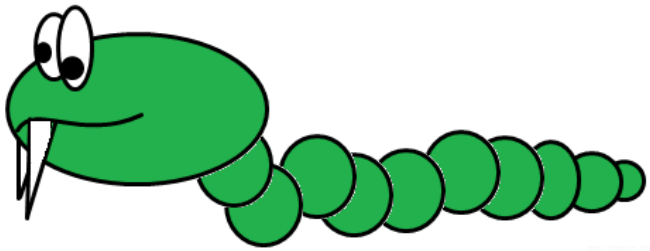
execute() turns the light on, so undo() simply turns the light back off.

# Adapter Pattern

- Converts the interface of a class into another interface the client expects. Adapter lets classes work together that couldn't otherwise because of incompatible interfaces.



# Adapter Pattern (Cont…)



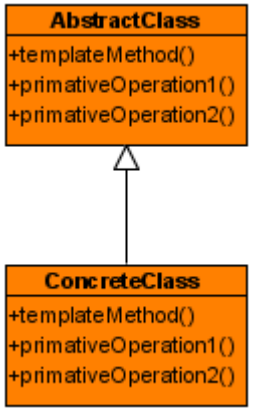
Methods:
updateX(dt);
updateY(dt);
updateAngle(dt);
AttackPlayer();
getHurt(int hp);

Methods:
update(dt){
  updateX(dt);
  updateY(dt);
   updateAngle(dt);
}
attack(){
  attackPlayer()
}
updateHeath(int hp){
  getHurt(int hp)
}

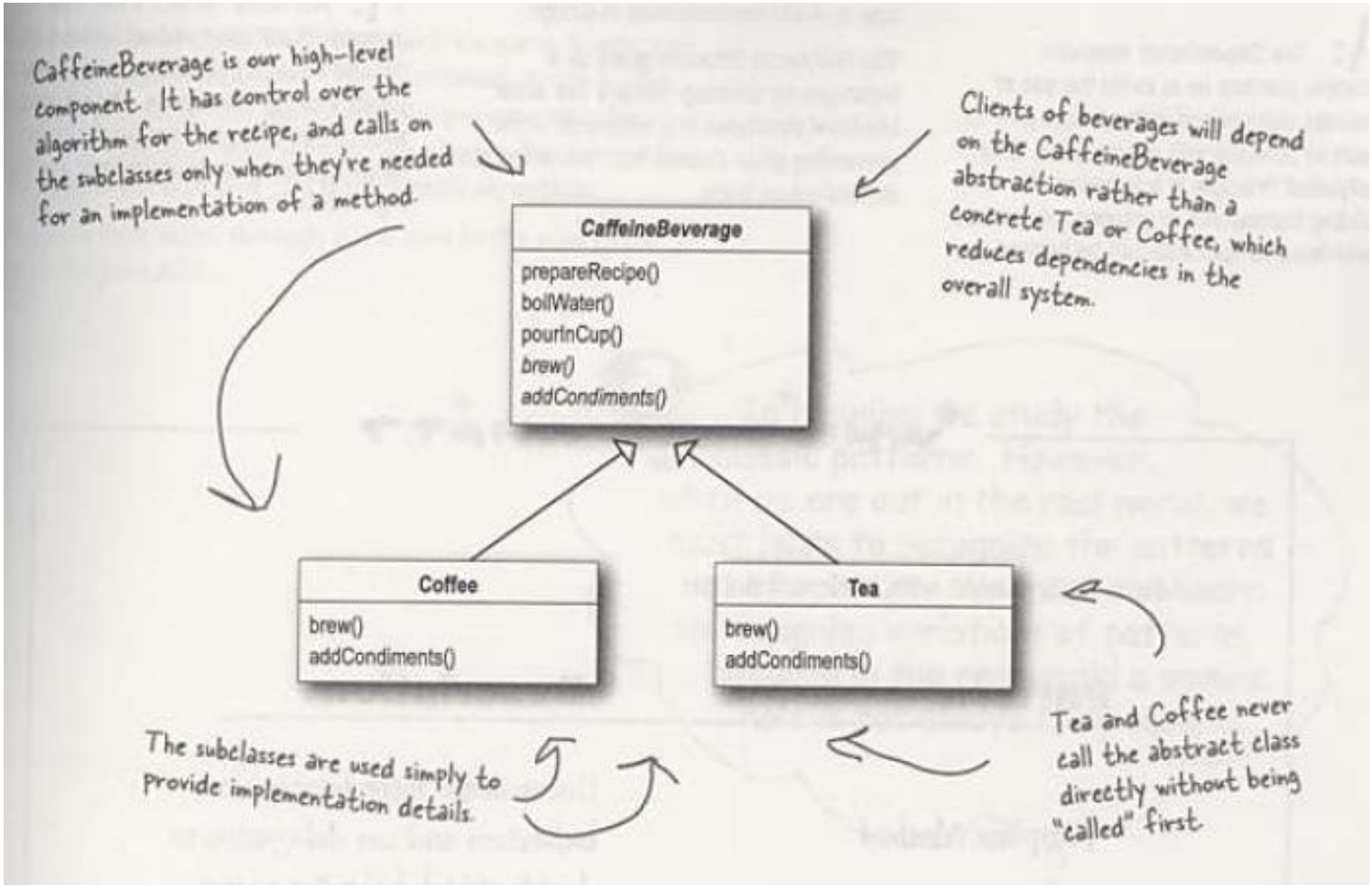

Methods:
update(dt);
attack();
updateHeath(int hp);

# Template Method Pattern

- Defines the skeleton of an algorithm in a method, deferring some steps to subclasses. Template method allows subclasses redefine certain steps of an algorithm without changing the algorithm's structure.

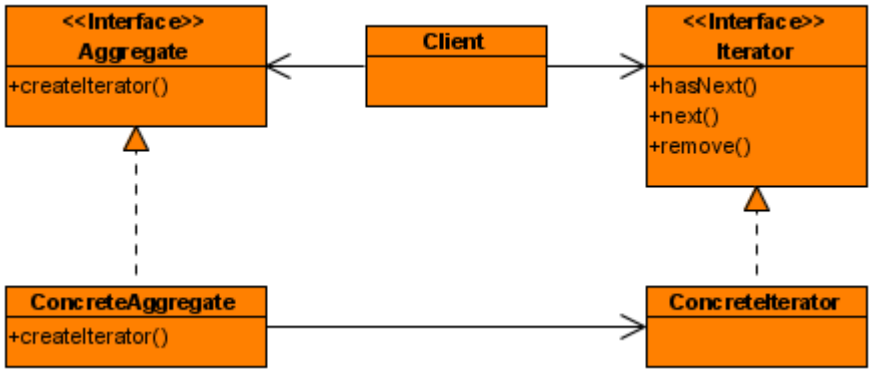


# Template Method Pattern (Cont…)

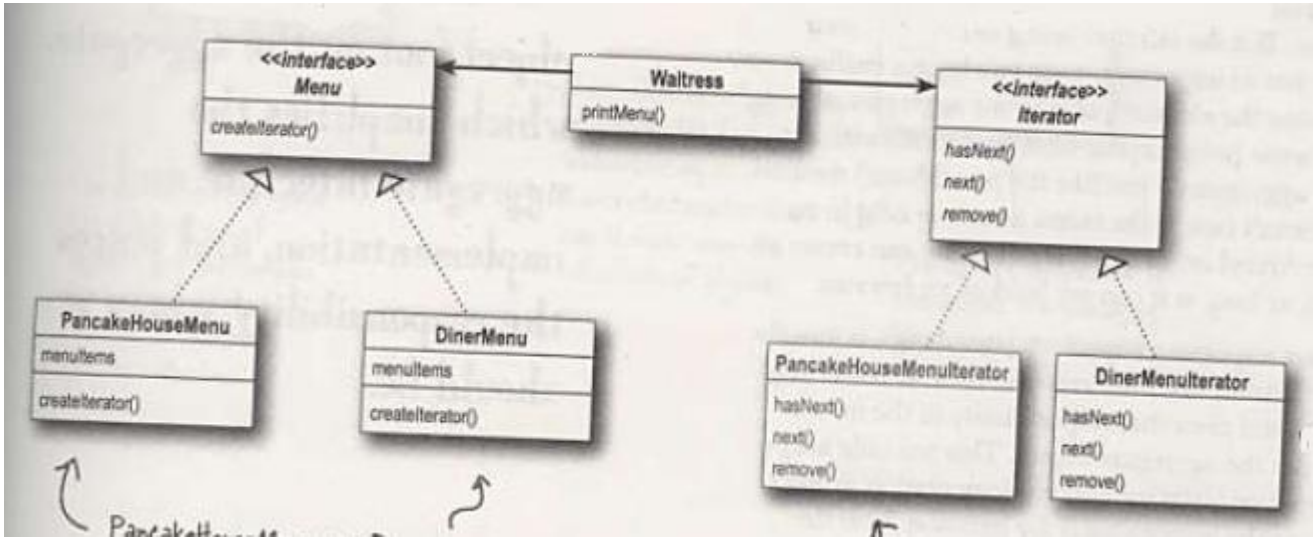


# Iterator & Composite Pattern

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



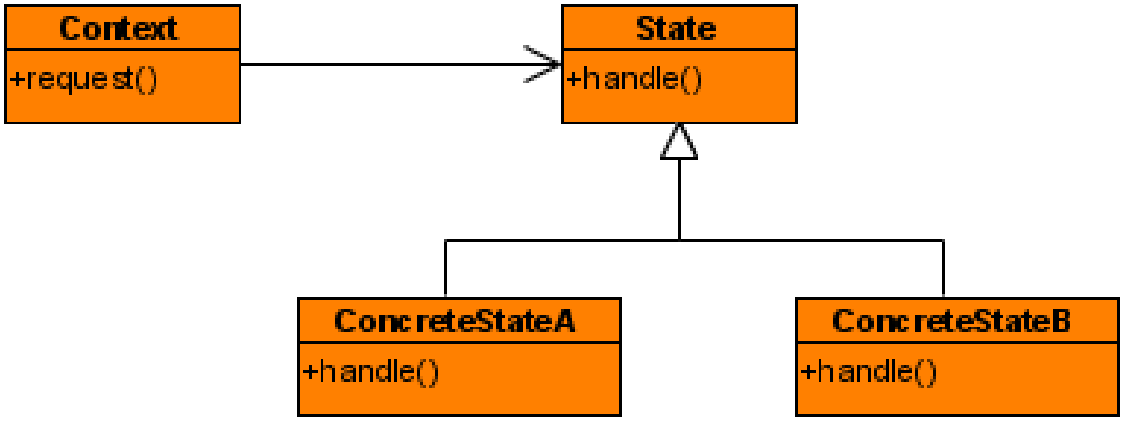
# Iterator & Composite Pattern (Cont…)



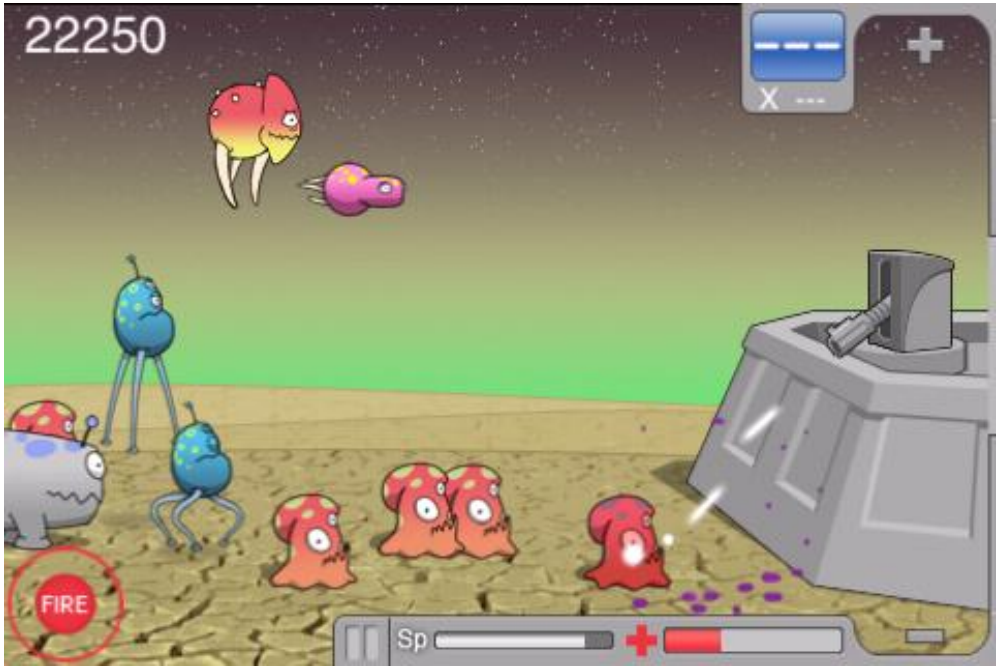

-Instead of the waitress knowing everything about the menu and all of the items, The waitress knows only a menu and an Iterator.

# State Pattern

- Allows an object to alter its behavior when its internal state changes. The object will appear to change its class.



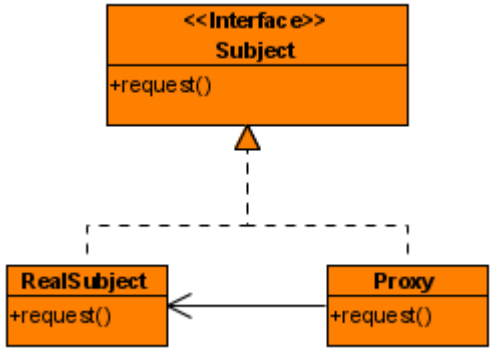
# State Pattern (Cont…)



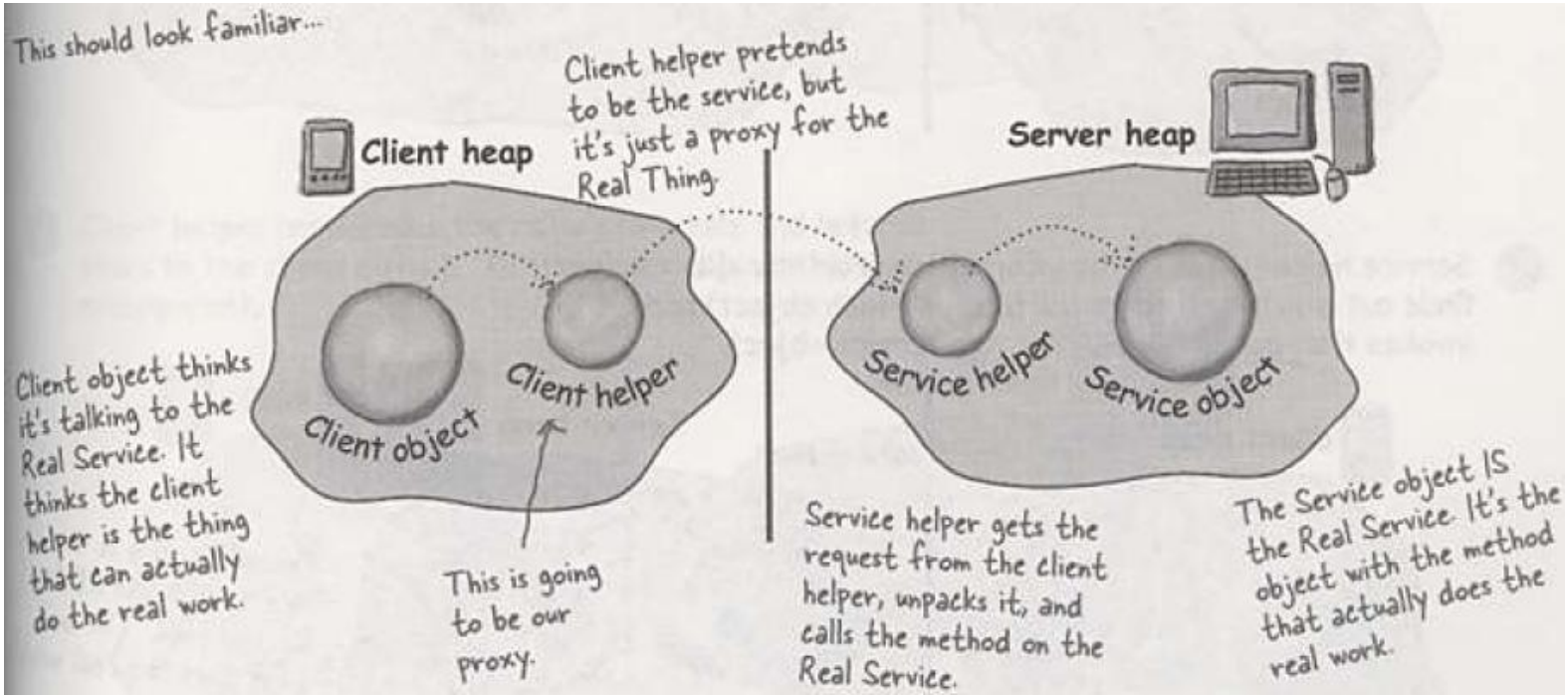

- Change user multiTouch input response based on the state of the Player. Such as if the player has the Laser or Rocket equipped, or if a interactive special is enabled.

# Proxy Pattern

- Provides a surrogate or placeholder for another object to control access to it.



# Proxy Pattern (Cont…)




Should be used if the class you are sending over the network has anything that is non-Serializable (ex Images)

# References


- Freeman, E., Freeman, E., Bates, B., & Sierra, K. (2004). *Head first: design patters*. O'Reilly Media.

- Butt, N. R. B. (2009, October 21). *Alien outpost*. Retrieved from www.newislandinteractive.com/alienoutpost


# Thank You!! Any Questions??

Alien Outpost Available now in the App Store.
Check under “Hot New Games”!!