# Software Design Patterns

Grehg Hilston

# Agenda

- Python
- Object Oriented Programming Review
- Class Diagrams
- Patterns
    - Description
    - UML Diagrams
    - Real World Example
    - (If Requested) Source Code Demonstration
- Anti-Patterns

## Goals

- Understand a little bit more of Python
- Leverage already solved problems
- Build a vocabulary to simplify discussions
- Minimize code changes
- Write less code

## Juypter Notebook Time

## UML - Unified Modeling Language

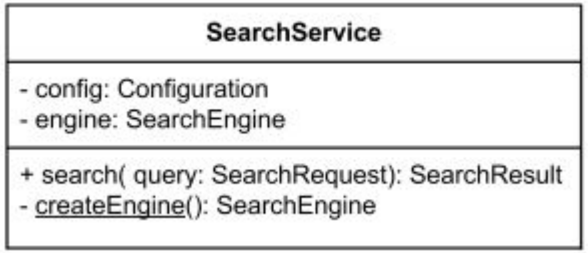
- Most popular - Class Diagrams
  - Use Case Diagrams
  - Sequence Diagrams
  - State Machine Diagrams…

## Access Modifiers

- Public - Visible by all
- Protected - Visible by this class and subclasses
- Private - Visible by this class
- Always be as private as possible; limit exposure

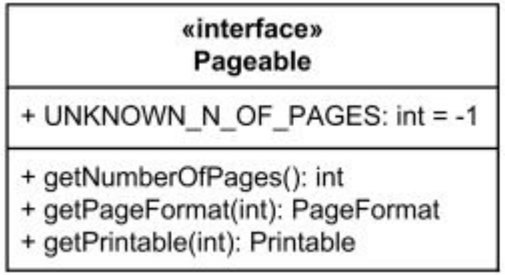
# UML - Class Diagram

Class

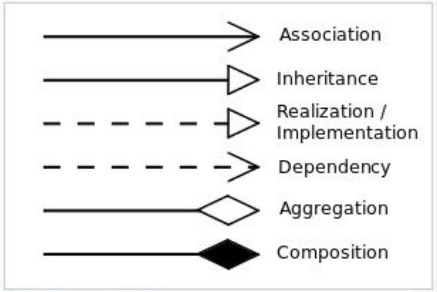


```
SearchService
- config: Configuration
- engine: SearchEngine
+ search( query: SearchRequest): SearchResult
- createEngine(): SearchEngine
```

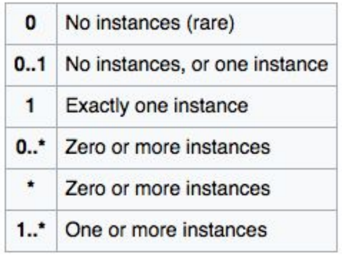
Interface



```
«interface»
Pageable
+ UNKNOWN_N_OF_PAGES: int = -1
+ getNumberOfPages(): int
+ getPageFormat(int): PageFormat
+ getPrintable(int): Printable
```

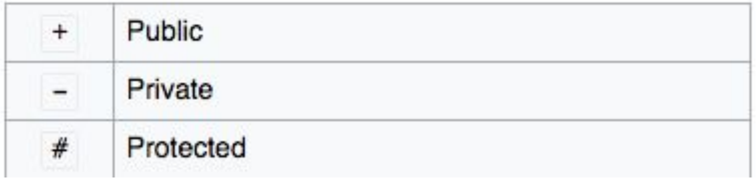


UML relations notation



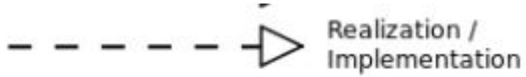
| | |
|---|---|
| 0 | No instances (rare) |
| 0..1 | No instances, or one instance |
| 1 | Exactly one instance |
| 0..* | Zero or more instances |
| * | Zero or more instances |
| 1..* | One or more instances |

Access Modifiers



| | |
|---|---|
| + | Public |
| - | Private |
| # | Protected |

# Interfaces aka Protocol

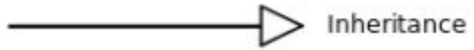


- “defines methods and values which the objects agree upon in order to co-operate”
- Cannot be instantiated
- Implementing an interface

## Abstract Class

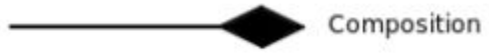
- “Abstract classes are classes that contain one or more abstract methods.”
- “An abstract method is a method that is declared, but contains no implementation. Abstract classes may not be instantiated, and require subclasses to provide implementations for the abstract methods.”
- Useful for allowing developers to customize

# Extension



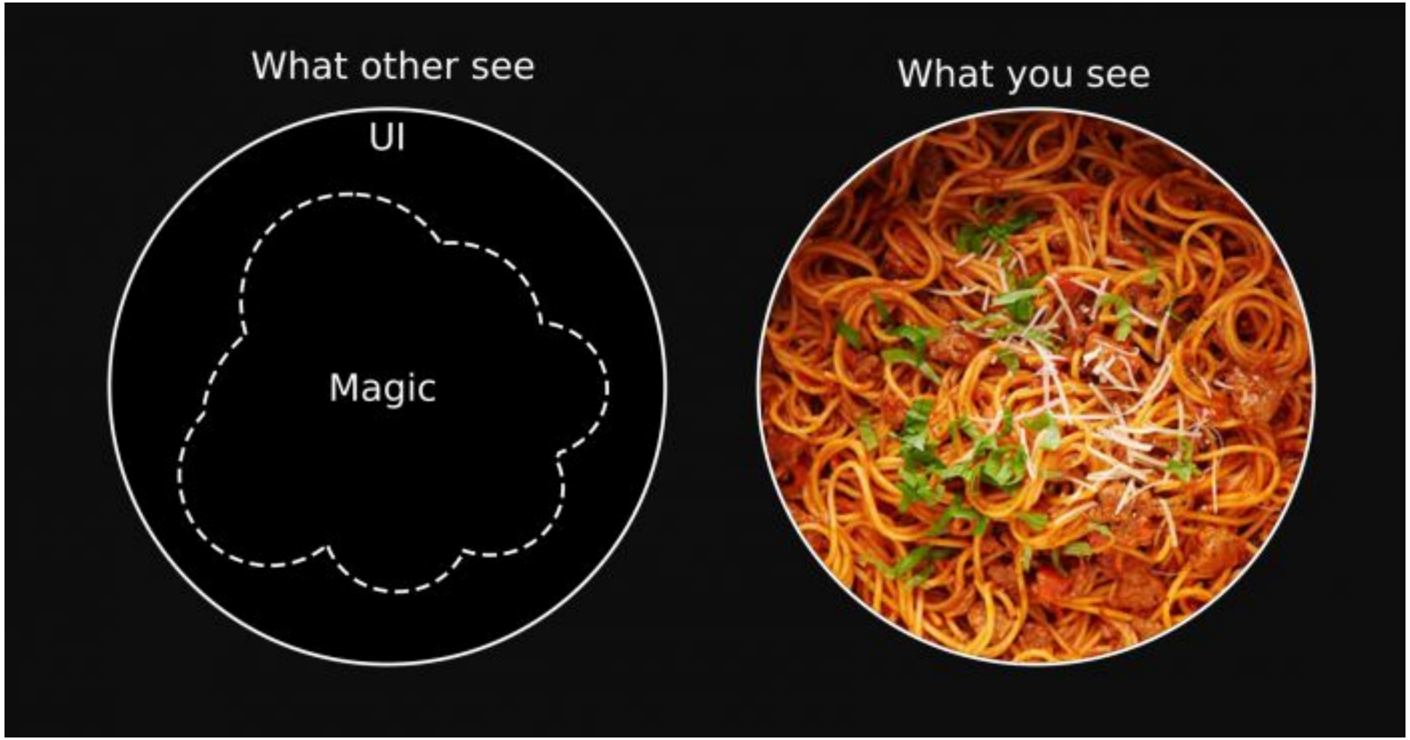
- Satisfies the “is a…” relationship
- Extend a base class
    - I.e. extending its functionality

# Composition



- Satisfies the “has a…” relationship
- Where one Class has an instance of another

# Why Do We Need Patterns?



## Pattern Types

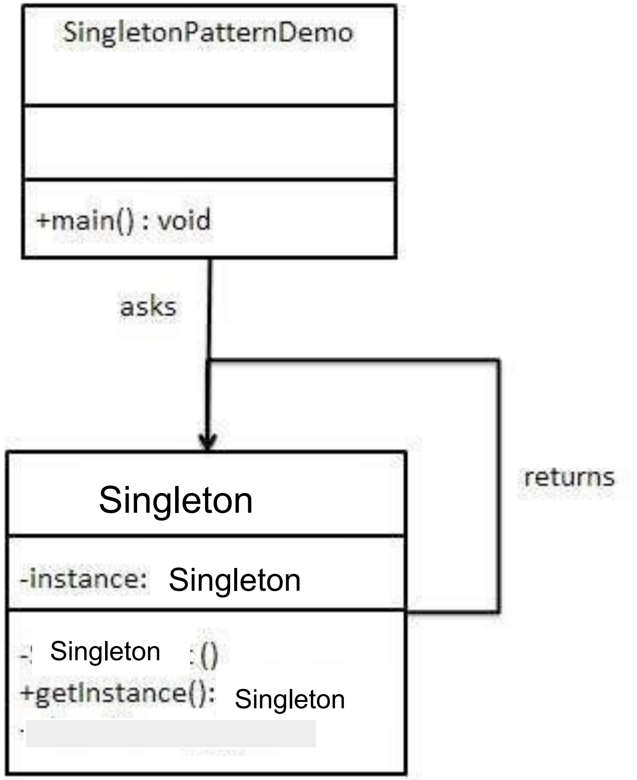
- Creational - “Deal with creation objects while hiding creation logic”
- Structural - “Deal with class and object composition”
- Behavioral - “Deal with communication between objects”

## Use Design Patterns When It Makes Sense

Cautionary story

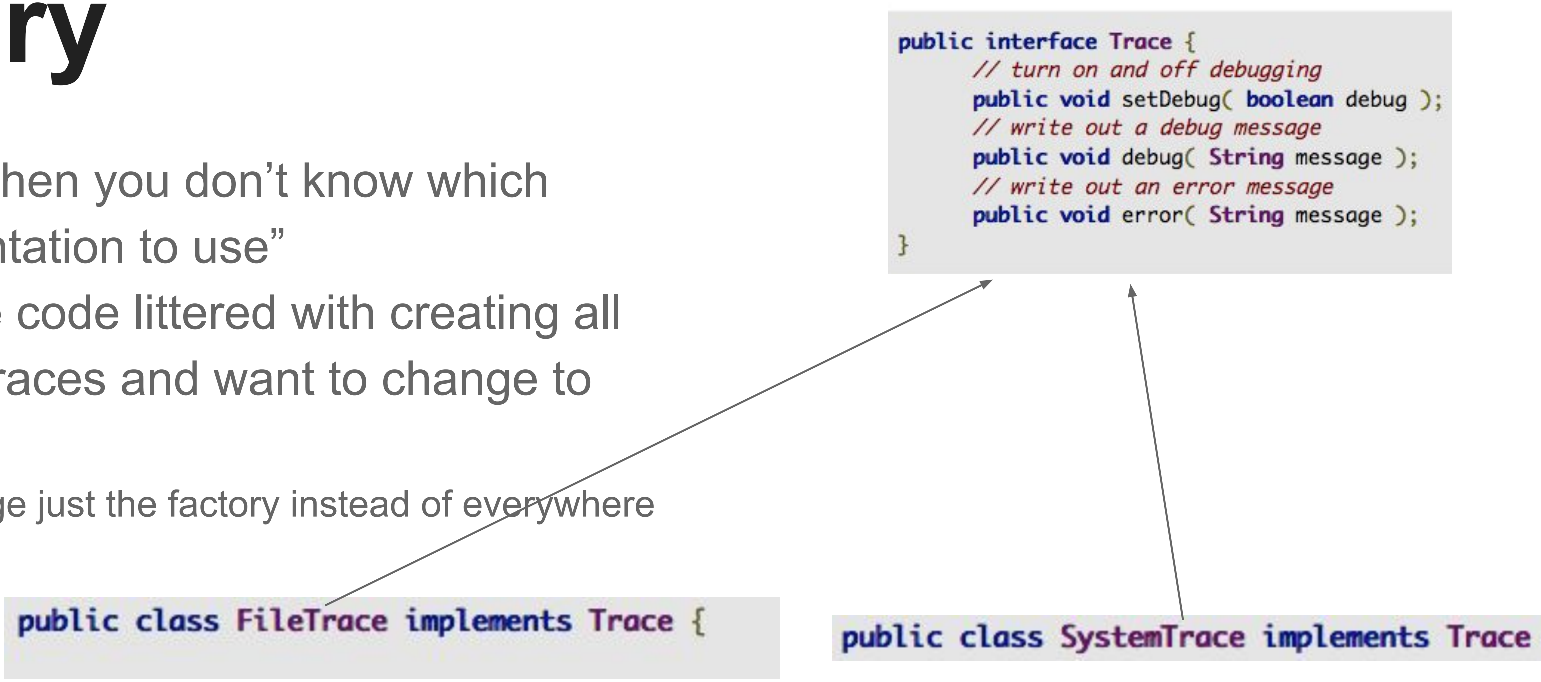
# Singleton

- Description: “Provides one, and only one, instance of an object.”
- Use Cases:
  - Thread pools
  - Caches
  - Logging
  - Device drivers



# Factory

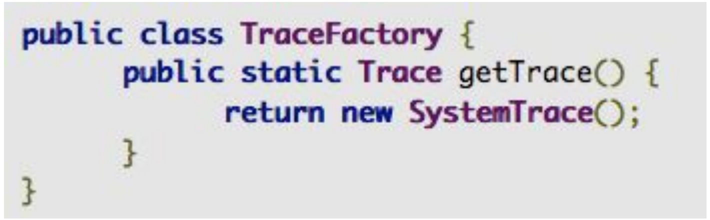
- “Useful when you don’t know which implementation to use”
- You have code littered with creating all SystemTraces and want to change to FileTrace
    - Change just the factory instead of everywhere



```java
public interface Trace {
        // turn on and off debugging
        public void setDebug( boolean debug );
        // write out a debug message
        public void debug( String message );
        // write out an error message
        public void error( String message );
}
```

```java
public class FileTrace implements Trace {
```

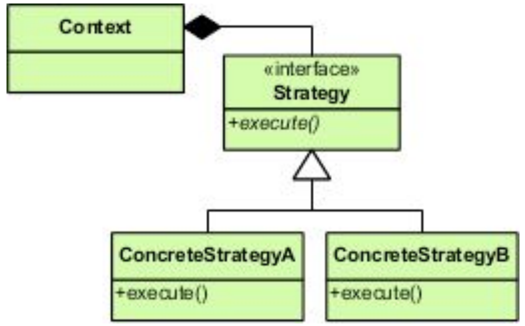
```java
public class SystemTrace implements Trace
```



```java
public class TraceFactory {
        public static Trace getTrace() {
                return new SystemTrace();
        }
}
```

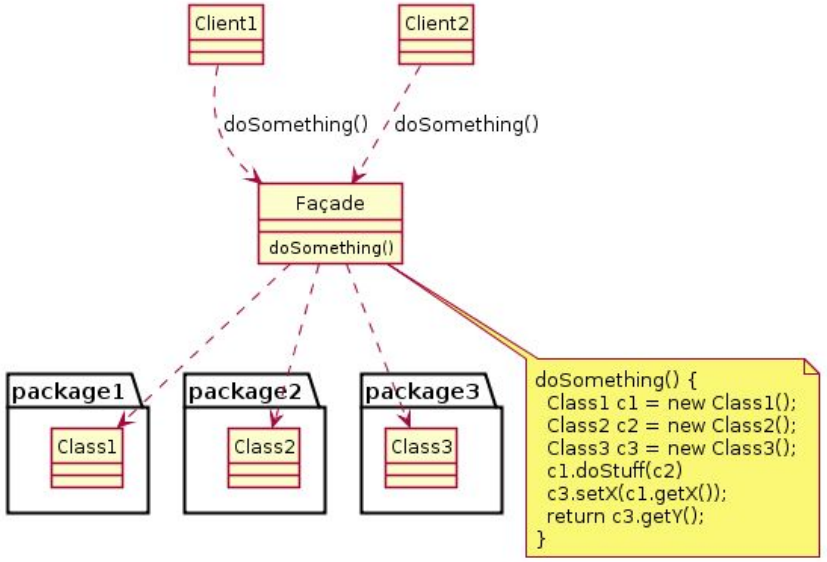
# Strategy

- “Easily swap out algorithms”
- Use Cases:
  - Game logic
  - Different calculation implementations



# Facade

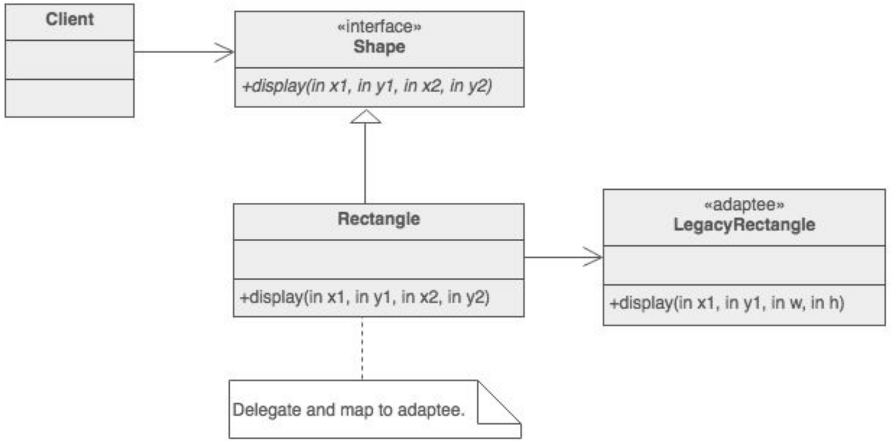
- “Provides a simplified interface”
- Use cases:
  - reduce dependencies of outside code on the inner workings of a library
  - Make a library more readable
  - Wrap poorly designed APIs in a single well-designed API



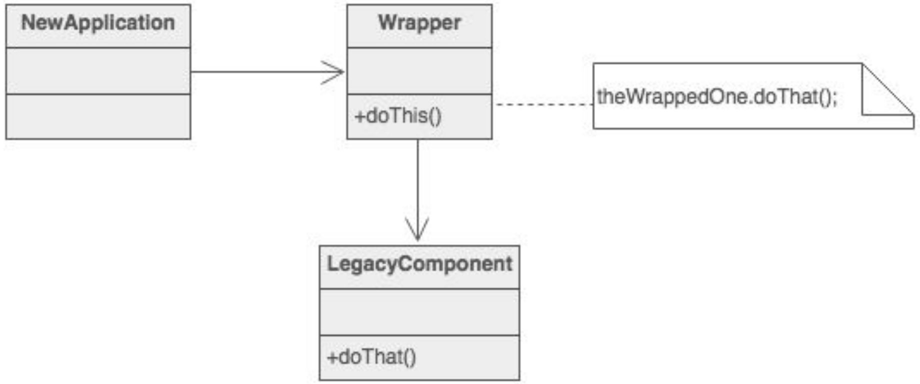
```
doSomething() {
  Class1 c1 = new Class1();
  Class2 c2 = new Class2();
  Class3 c3 = new Class3();
  c1.doStuff(c2)
  c3.setX(c1.getX());
  return c3.getY();
}
```

# Adapter

- "Wraps an existing class with a new interface"
- Use Cases:
  - Interfacing with legacy code
  - Interfacing with other libraries, packages or systems

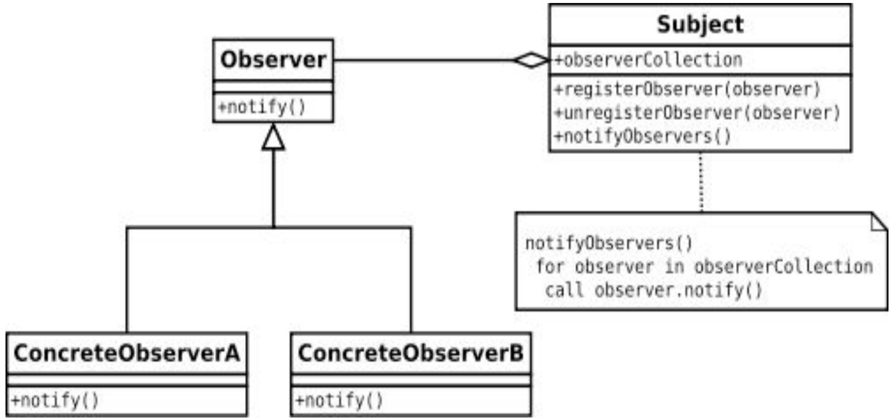

Client
«interface»
Shape
+display(in x1, in y1, in x2, in y2)
Rectangle
+display(in x1, in y1, in x2, in y2)
«adaptee»
LegacyRectangle
+display(in x1, in y1, in w, in h)
Delegate and map to adaptee.

The Adapter could also be thought of as a "wrapper".

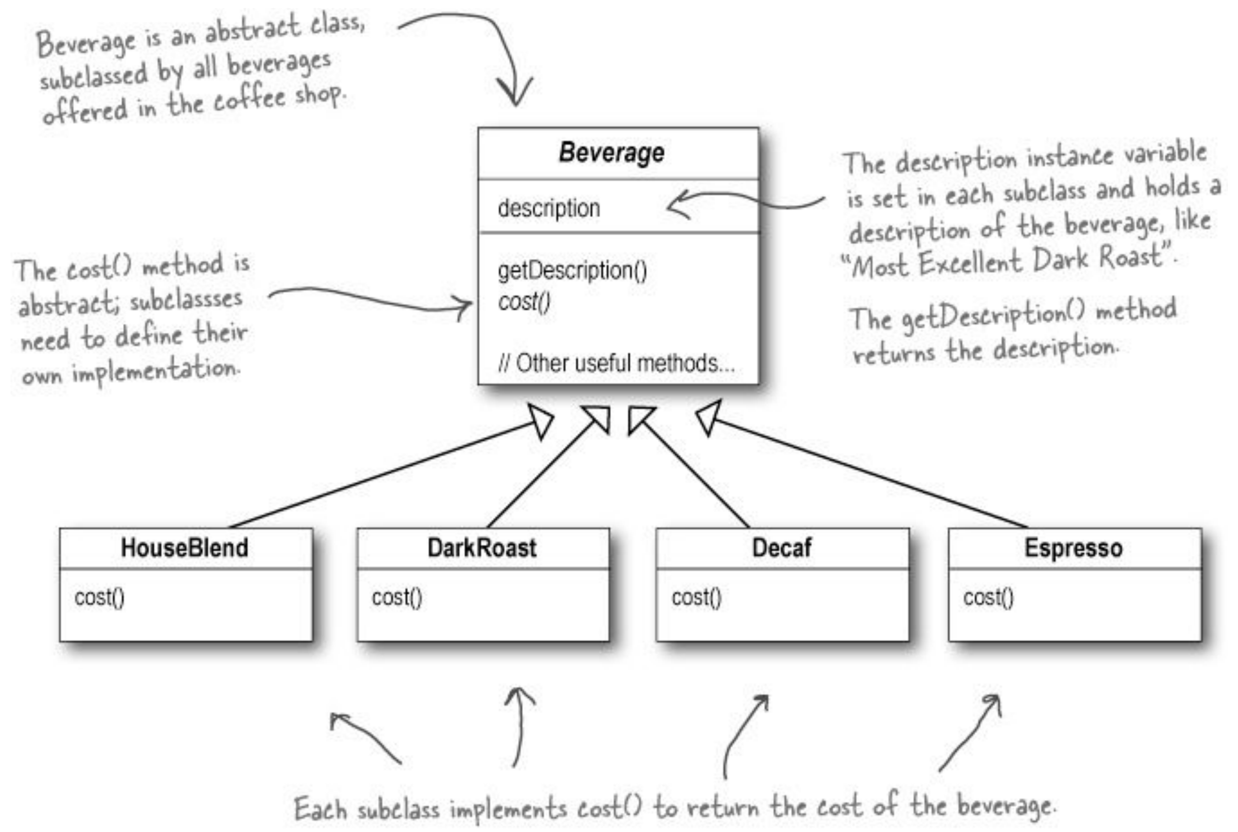


# Observer / Observable

- “Enables subscription based notifications to be registered to”
- Use Cases:
  - Notify on state changes
  - Useful in the MVP pattern
  - Helpful in event handling systems

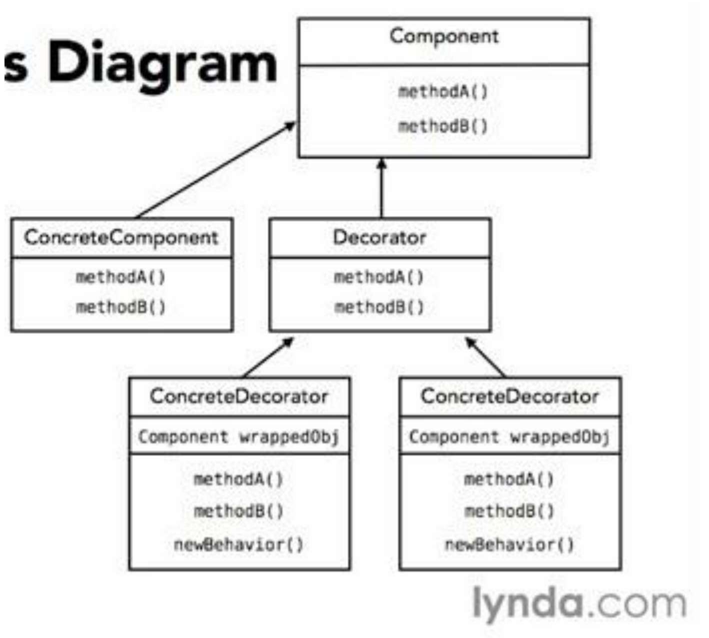


# Decorator - Code Smell

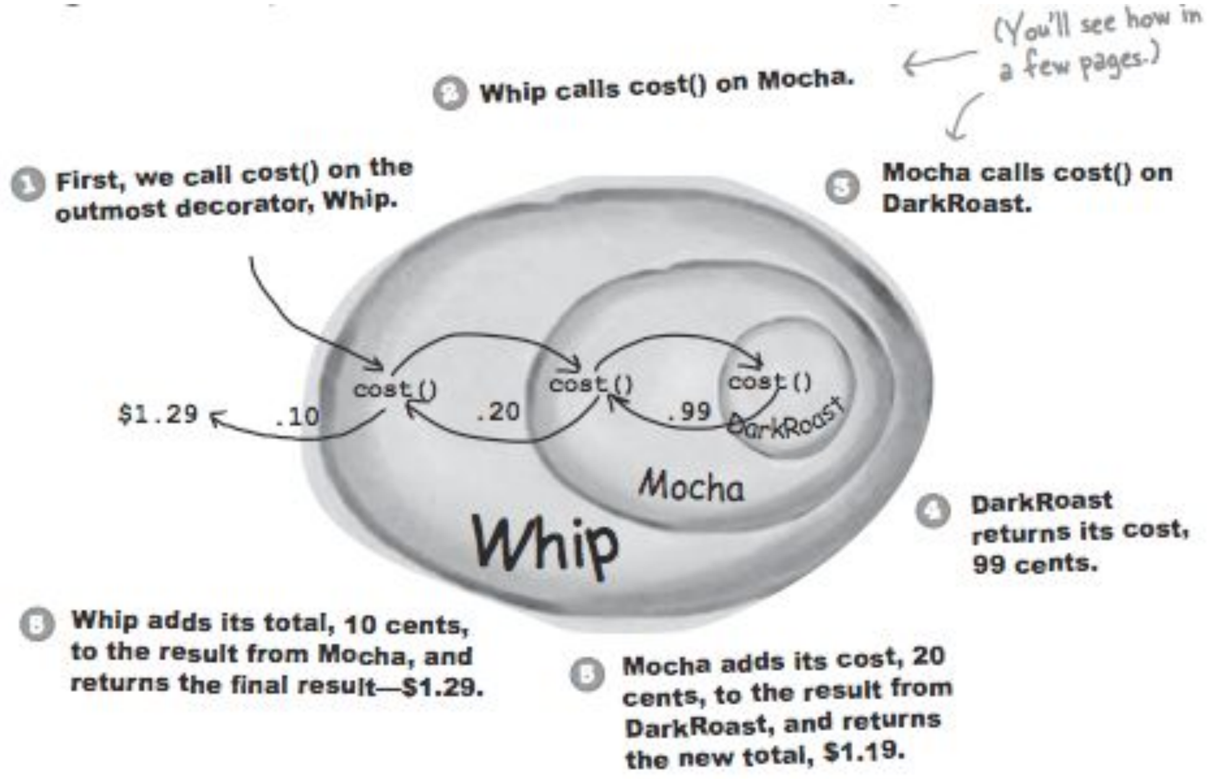

Beverage is an abstract class, subclassed by all beverages offered in the coffee shop.
Beverage
description
getDescription()
cost()
// Other useful methods...
The description instance variable is set in each subclass and holds a description of the beverage, like "Most Excellent Dark Roast".
The getDescription() method returns the description.
The cost() method is abstract; subclassses need to define their own implementation.
HouseBlend
cost()
DarkRoast
cost()
Decaf
cost()
Espresso
cost()
Each subclass implements cost() to return the cost of the beverage.

# Decorator

- “Simplifies many objects with optional decorations”
- Use Cases
    - Menus
    - Drinks
    - Enemies in games



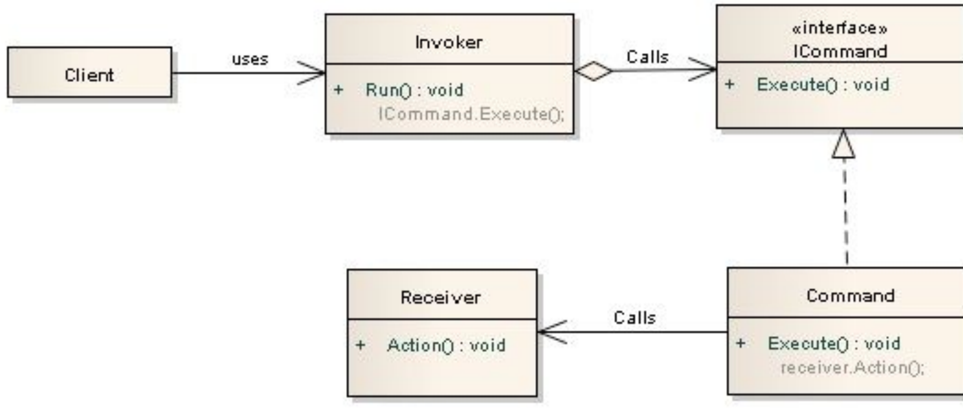
# Decorator



1. First, we call cost() on the outmost decorator, Whip.
2. Whip calls cost() on Mocha.
3. Mocha calls cost() on DarkRoast.
4. DarkRoast returns its cost, 99 cents.
5. Mocha adds its cost, 20 cents, to the result from DarkRoast, and returns the new total, $1.19.
6. Whip adds its total, 10 cents, to the result from Mocha, and returns the final result—$1.29.

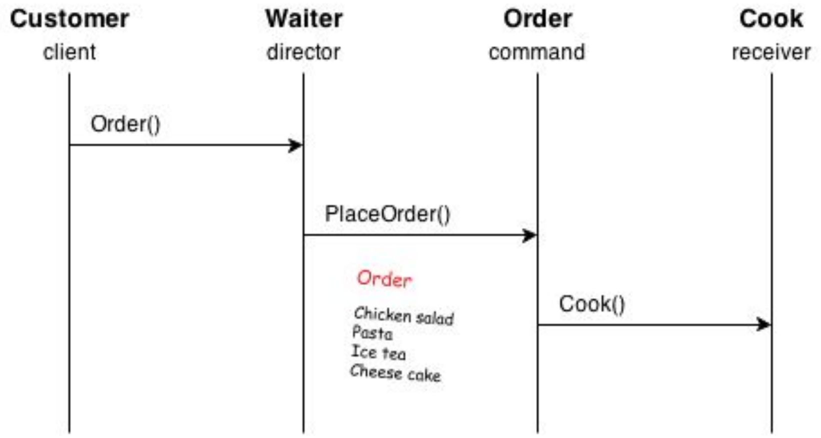
# Command

- "Encapsulates a command as an object"
- Use Cases:
    - Separate invoker and receiver



## Command Example

The Command pattern allows requests to be encapsulated as objects, thereby allowing clients to be parametrized with different requests. The "check" at a diner is an example of a Command pattern. The waiter or waitress takes an order or command from a customer and encapsulates that order by writing it on the check. The order is then queued for a short order cook. Note that the pad of "checks" used by each waiter is not dependent on the menu, and therefore they can support commands to cook many different items.

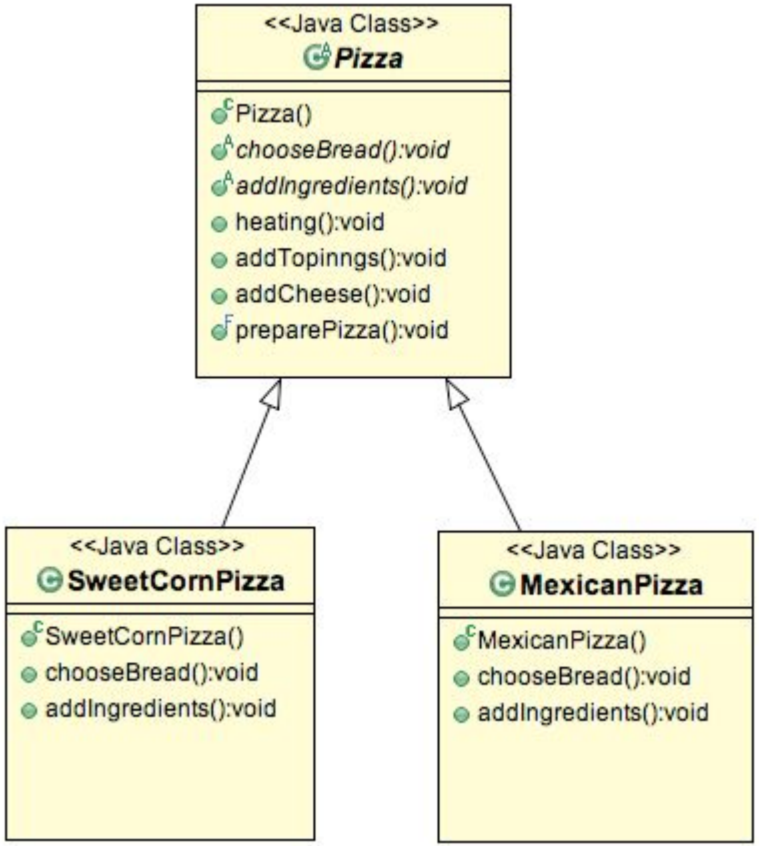


### Check list

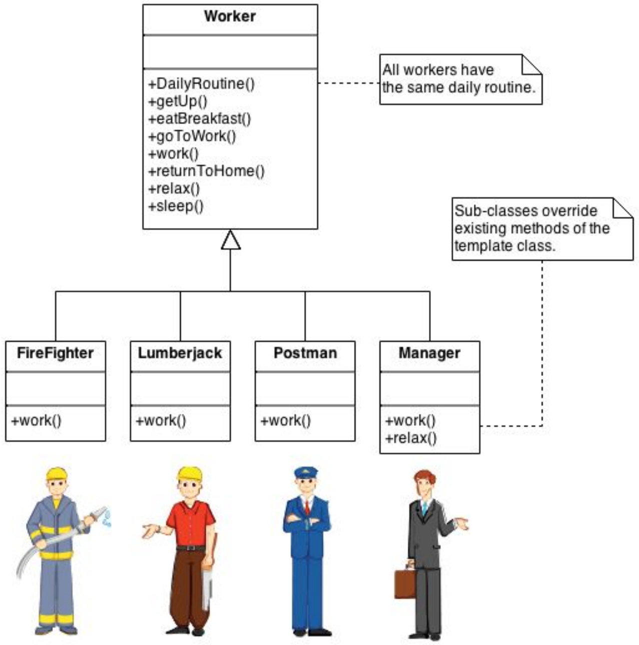
1. Define a Command interface with a method signature like `execute()`.
2. Create one or more derived classes that encapsulate some subset of the following: a "receiver" object, the method to invoke, the arguments to pass.
3. Instantiate a Command object for each deferred execution request.
4. Pass the Command object from the creator (aka sender) to the invoker (aka receiver).
5. The invoker decides when to `execute()`.

# Template

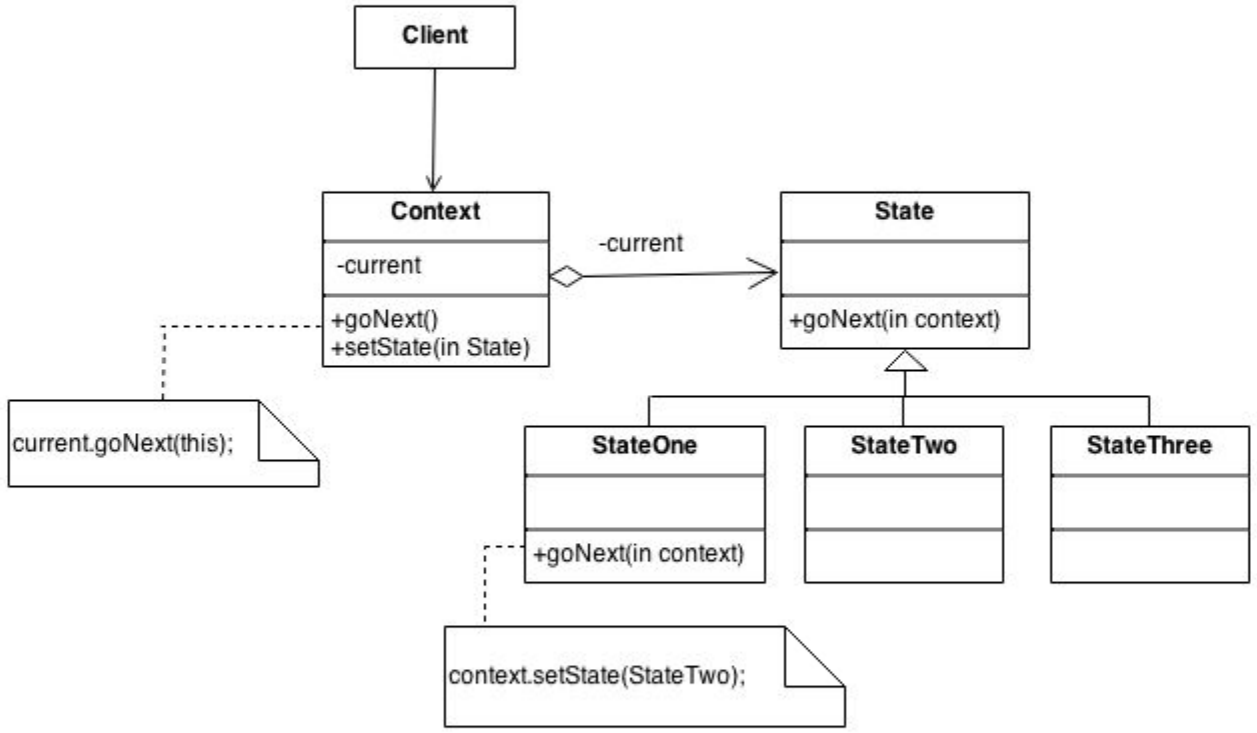
- “Factors out commonality between similar classes”
- Use Cases
  - Initial design and notice similarity
    - Useful for writing code once



## Template Example



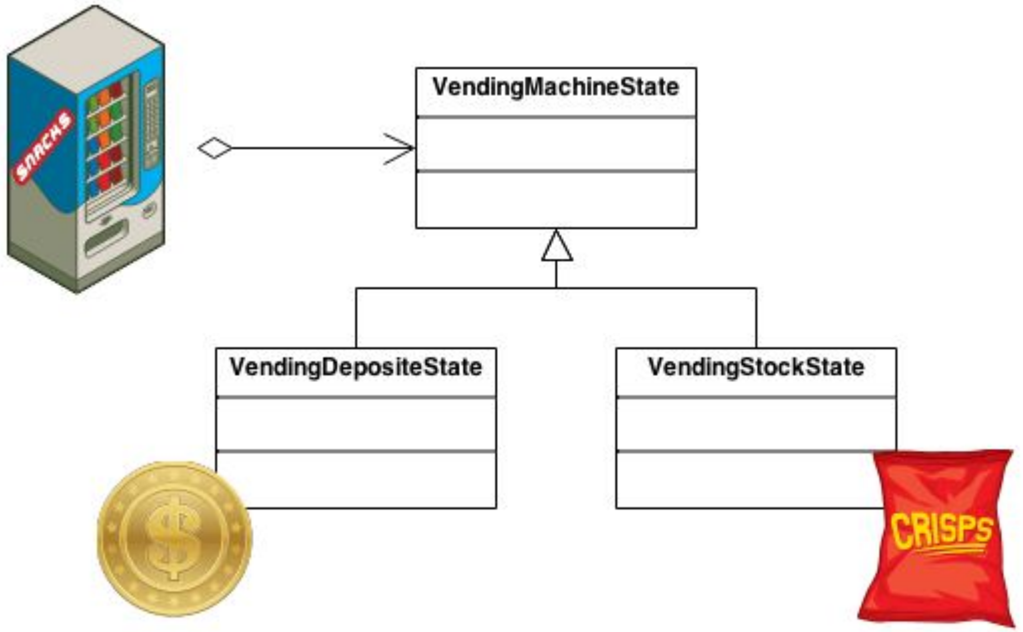
# State



current.goNext(this);

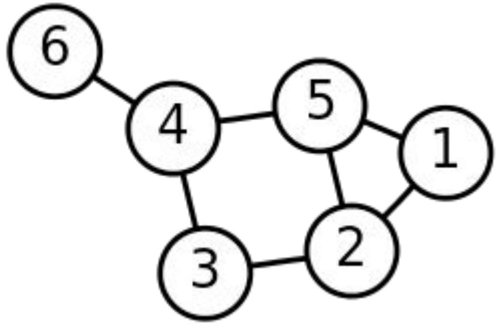
context.setState(StateTwo);

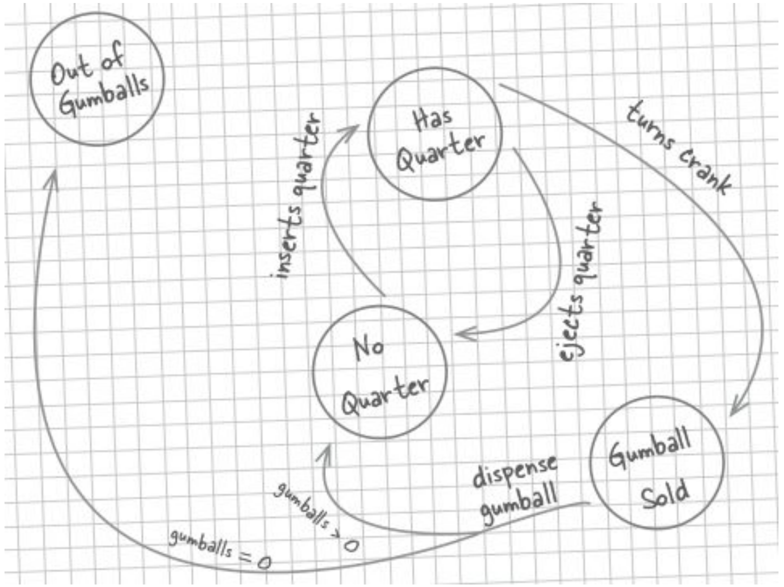
# State Example



## State Machine

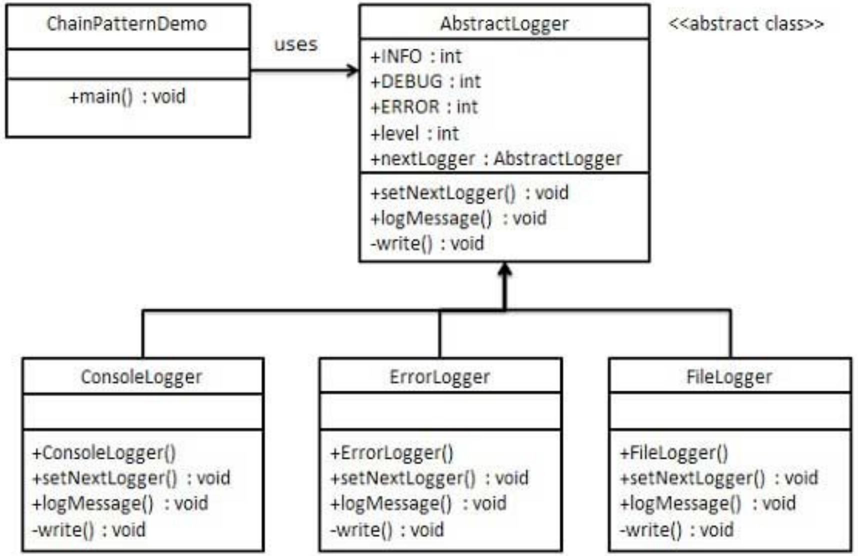
- "Allows an object to alter its behavior when its internal state changes"
- Use Cases:
  - Useful by anything that can be modeled by a state graph
- Graph = State Machine
  - Vertices = state
  - Edges = actions (each pair of vertices)
- Reflexive Edges
- Excellent for testing





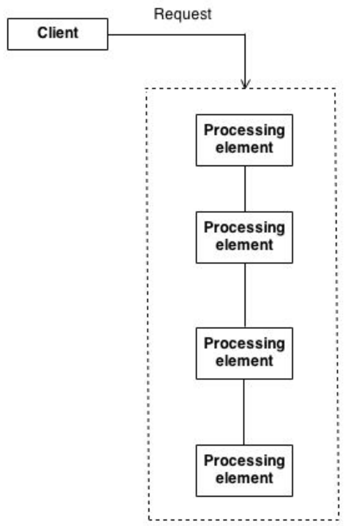
# Chain of Responsibility

- “Allows multiple changes at resolution”
- Use Cases:
  - Cache implementations
  - Thread Pool Manager
  - Implementing your own logger



# Chain of Responsibility Example

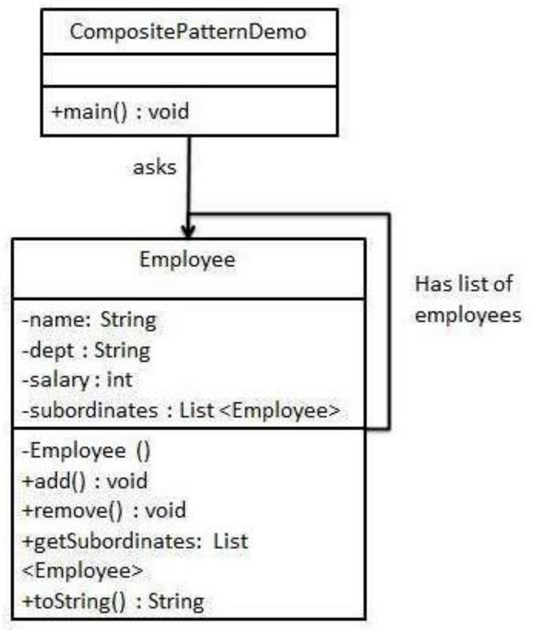
Encapsulate the processing elements inside a "pipeline" abstraction; and have clients "launch and leave" their requests at the entrance to the pipeline.



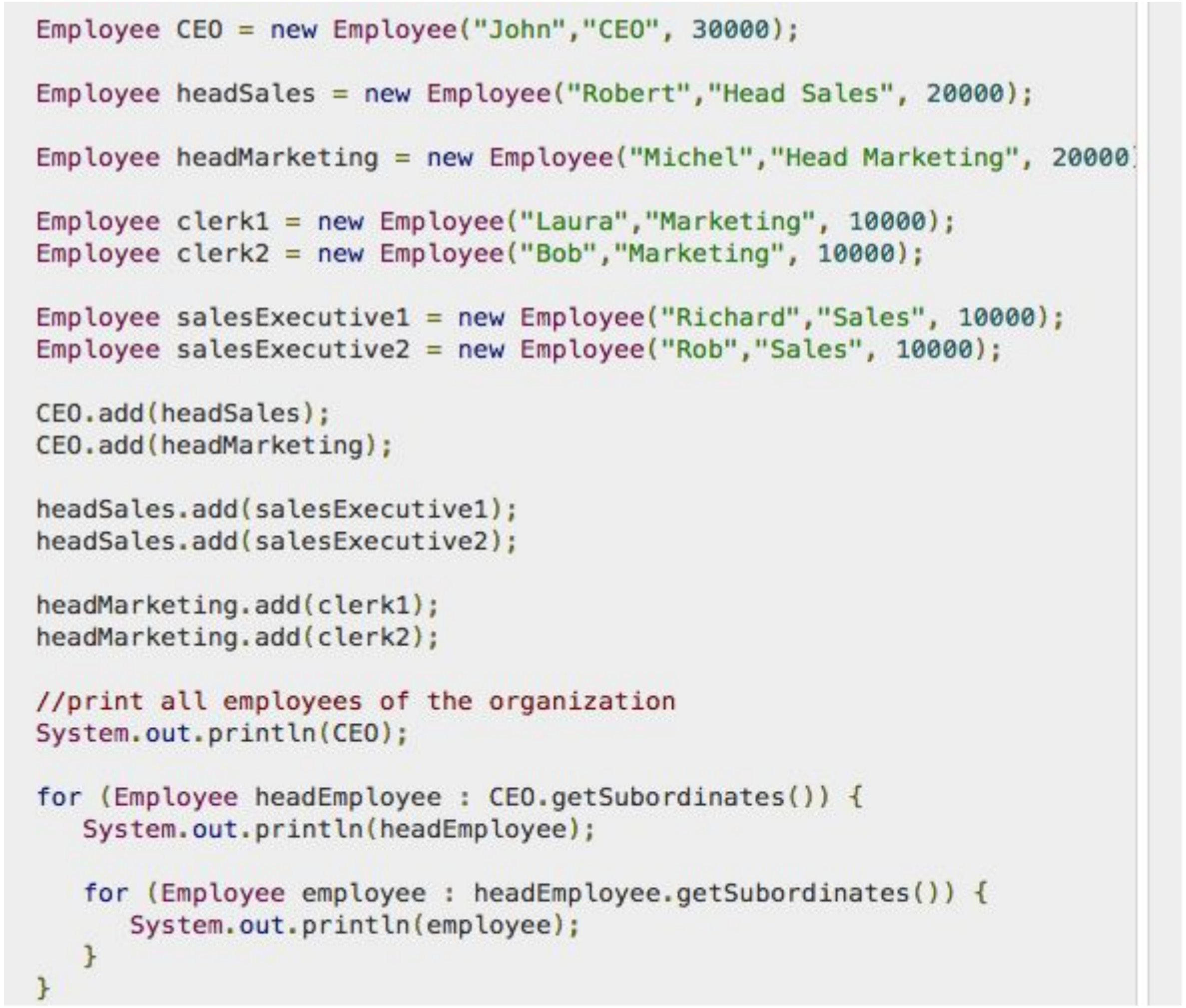
The pattern chains the receiving objects together, and then passes any request messages from object to object until it reaches an object capable of handling the message. The number and type of handler objects isn't known a priori, they can be configured dynamically. The chaining mechanism uses recursive composition to allow an unlimited number of handlers to be linked.

# Composite

- “Composite pattern is used where we need to treat a group of objects in similar way as a single object.”



# Composite Example



```java
Employee CEO = new Employee("John","CEO", 30000);

Employee headSales = new Employee("Robert","Head Sales", 20000);

Employee headMarketing = new Employee("Michel","Head Marketing", 20000)

Employee clerk1 = new Employee("Laura","Marketing", 10000);
Employee clerk2 = new Employee("Bob","Marketing", 10000);

Employee salesExecutive1 = new Employee("Richard","Sales", 10000);
Employee salesExecutive2 = new Employee("Rob","Sales", 10000);

CEO.add(headSales);
CEO.add(headMarketing);

headSales.add(salesExecutive1);
headSales.add(salesExecutive2);

headMarketing.add(clerk1);
headMarketing.add(clerk2);

//print all employees of the organization
System.out.println(CEO);

for (Employee headEmployee : CEO.getSubordinates()) {
   System.out.println(headEmployee);

   for (Employee employee : headEmployee.getSubordinates()) {
      System.out.println(employee);
   }
}
```

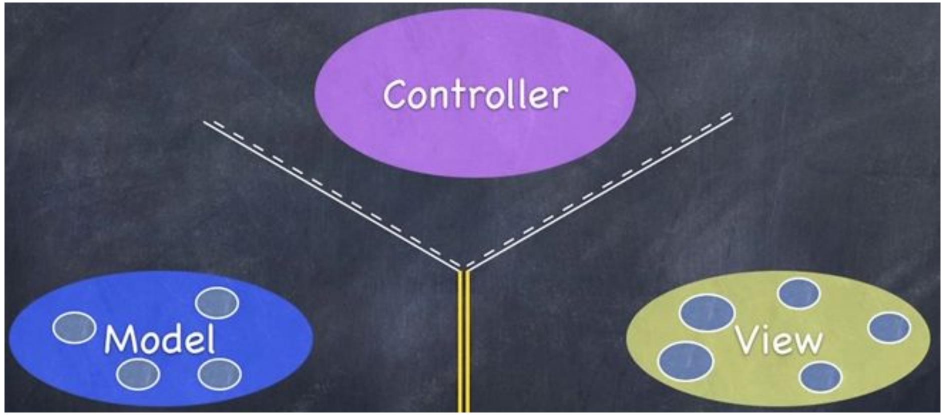
# Antipatterns

- God Class - One class with all responsibilities
- Vendor lock in - Excessively dependent on an externally supplied component
- Magic numbers

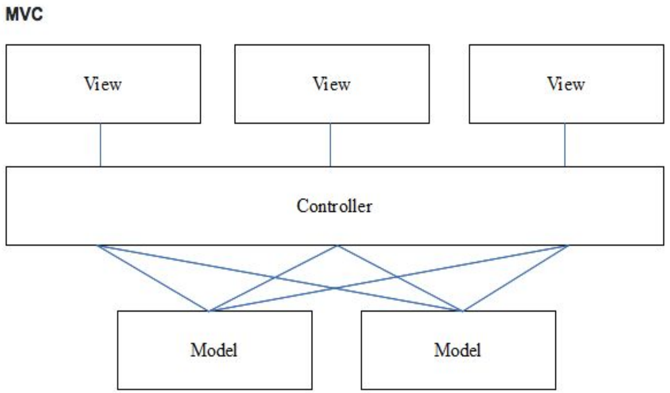
## Backup Slides

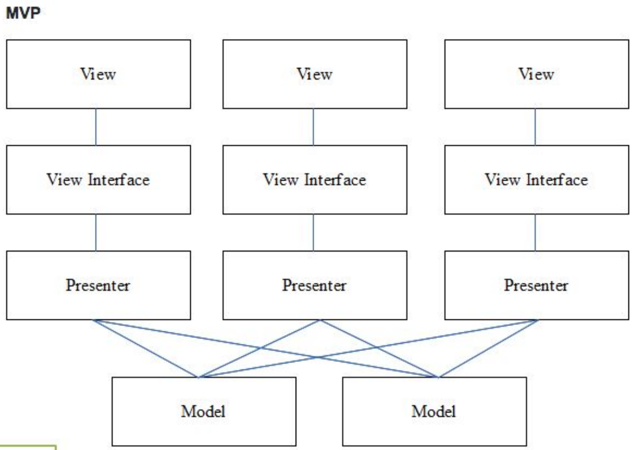
## Model View Controller / Presenter

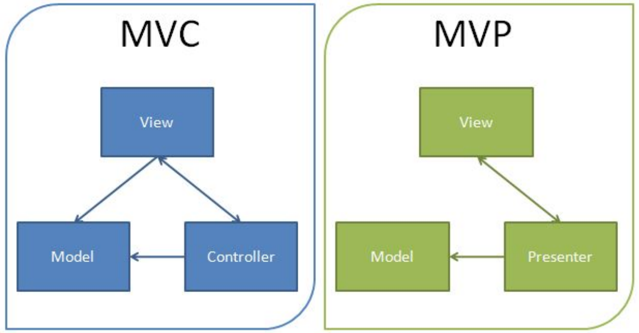
- “Separation of concerns”
  - UI
  - Formatting
  - Backend
- Use Cases:
  - Everything
    - Mobile apps
    - Web sites



# MVP vs MVC


MVC
View
View
View
Controller
Model
Model


MVP
View
View
View
View Interface
View Interface
View Interface
Presenter
Presenter
Presenter
Model
Model


MVC
View
Model
Controller
MVP
View
Model
Presenter