# Software Design Patterns

Alison Li

COMP 303 @ McGill University

## 1 Iterator Pattern

**Purpose.** *Provide a way to access the elements of an aggregate object in order without exposing its underlying representation.*

**Requirements.** *Using the* `Iterable` *interface:*

- *To enable iteration, have the class implement the* `Iterator` *interface.*
- `next()` *moves the iterator to the next element.*
- `hasNext()` *checks if the next element exists.*

```
public class CustomDataStructure implements Iterable<> {
    ...
    public Iterator<> iterator() {
        return new CustomIterator<>(this);
    }
}
```

When code gains access to a reference to an object of any sub-type of `Iterator`, the client code can iterate over it, regardless of the class of the object.

**Requirements.** *Using the* `Iterator` *interface:*

- *Make the class iterable by implementing the* `Iterable` *interface.*
- *The method* `iterator()` *which should return* `Iterator<T>`*, an iterator object.*

```
class ListIterator<T> implements Iterator<T> {
    Node<T> current;

    public ListIterator(List<T> list) {
        current = list.getHead();
    }

    public boolean hasNext() {
        return current != null;
    }

    public T next() {
        T data = current.getData();
        current = current.getNext();
    }
}
```

## 2 Strategy Pattern

**Purpose.** *When you want different variations of an algorithm. Define a family of algorithms, encapsulate each one, and make them interchangeable. Allow the ability for algorithms to vary independently from clients that use it.*

**Requirements.**

- *Implement the* `Comparator` *interface.*

- *The* `compare(Object o1, Object o2)` *method which should return an* `int`*.*
- *We can then sort with our desired Comparator by calling* `Collections.sort(List<T> list, Comparator<? super T> C)`*.*

One solution is to declare the comparator class as a nested class of the class being compared.

```
public class Card {
    public static class CompareBySuitFirst implements Comparator<Card> {
        @Override
        public int compare(Card pCard1, Card pCard2) {
            // Comparison code
        }
    }
}
```

The other solution is to define comparator classes as anonymous classes, which makes sense in cases where the comparator doesn't hold a state and is only referred to once.

```
Collections.sort(aCards, new Comparator<Card>() {
    @Override
    public int compare(Card pCard1, Card pCard2) {
        return pCard1.getRank().compareTo(pCard2.getRank());
    };
}
```

# 3 Flyweight Pattern

**Purpose.** *When you want to manage the creation of objects of a certain class, the Flyweight class, through a factory method. Provides a way to manage collections of immutable objects. Useful for ensuring the uniqueness of objects.*

**Requirements.**

- *A* `private` *constructor for the Flyweight class.*
- *A data structure that keeps a list of the Flyweight instances, stored in a* `static` *field.*
- *A* `static` *factory method that returns the unique Flyweight object that corresponds to the input parameter. A factory method allows the class to defer instantiation to a sub-class.*

In the example below that implements the creation of `Terrorists` and `Counter Terrorists` in the game of *Counter Strike*, we use the Flyweight Pattern since we need to reduce the object count for players.

```
interface Player {
    public void assignWeapon(String weapon);
    public void mission();
}
```

We implement the creation of `Terrorists` objects and `CounterTerrorists` objects with the `Player` interface.

```
class Terrorist implements Player {
    private final String TASK;
    private String weapon;

    private Terrorist() {
        TASK = "PLANT A BOMB";
    }

    public void assignWeapon(String weapon) {
        this.weapon = weapon;
    }

    public void mission() {
        ...
    }
}
```

```
class CounterTerrorist implements Player {
    private final String TASK;
    private String weapon;

    private CounterTerrorist() {
        TASK = "DIFFUSE BOMB";
    }

    public void assignWeapon(String weapon) {
        this.weapon = weapon;
    }

    public void mission() {
        ...
    }
}
```

Note that we could have also used a `static` block in `PlayerFactory` to pre-initialize all the flyweight instances, rather than waiting for the client to create them.

```
class PlayerFactory {
    private static HashMap <String, Player> hm = new HashMap<>();

    public static Player getPlayer(String type) {
        Player p = null;

        if (hm.containsKey(type)) {
            p = hm.get(type);
        } else {
            switch (type) {
                case "Terrorist":
                    p = new Terrorist();
                    break;
                case "CounterTerrorist":
                    p = new CounterTerrorist();
                    break;
                default:
                    ... // Unreachable code
            }
            hm.put(type, p);
        }
        return p;
    }
}
```

# 4 Singleton Pattern

**Purpose.** *Ensure that there is only one instance of a given class at any point in the execution of the program. Useful to simplify the access of stateful objects that have the role of a controller.*

**Requirements.**

- *A* `private` *constructor for the Singleton class.*
- *A* `static final` *field keeping a reference to the Singleton object.*
- *A* `static` *accessor method that returns the unique instance of the Singleton.*

Note that a Singleton attempts to guarantee a single instance of a class as opposed to unique instances of a class. The following example includes a `getRuntime()` method which returns the runtime object associated with the current Java application.

```
public class Runtime {
    private static Runtime currentRuntime = new Runtime();

```

```
    private Runtime() {}

    public static Runtime getRuntime() {
        return currentRuntime;
    }
}
```

# 5 Composite Pattern

**Purpose.** *Compose objects into tree structures to represent part-whole hierarchies. Composite lets you, or the client, manipulate a single instance of the object just as you would manipulate a group of them.*

**Requirements.**

- *Component: Declares the interface for objects in the composition. Also implements default behaviour for the interface common to all classes.*
- *Leaf: Defines behaviour for primitives in the composition.*
- *Composite: An object designed as a composition of one or more similar objects, all exhibiting similar functionality. Stores child components and implements child related operations in the component interface.*

```
public interface Employee {
    public void showEmployeeDetails();
}
```

```
public class Developer implements Employee {
    private String name;
    private long empId;
    private String position;

    public Developer (long empId, String name, String position) {
        this.empId = empId;
        this.name = name;
                this.position = position;
    }

    @Override
    public void showEmployeeDetails() {
        System.out.println(empId+" " +name+);
    }
}
```

```
public class Manager implements Employee {
    private String name;
    private long empId;
    private String position;

    public Manager (long empId, String name, String position) {
        this.empId = empId;
        this.name = name;
                this.position = position;
    }

    @Override
    public void showEmployeeDetails() {
        System.out.println(empId+" " +name);
    }
}
```

```java
public class CompanyDirectory implements Employee {
    private List<Employee> employeeList = new ArrayList<Employee>();

    @Override
    public void showEmployeeDetails() {
        for (Employee emp:employeeList) {
            emp.showEmployeeDetails();
        }
    }

    public void addEmployee (Employee emp) {
        employeeList.add(emp);
    }

    public void removeEmployee (Employee emp) {
        employeeList.remove(emp);
    }
}
```

# 6 Decorator Pattern

**Purpose.** *When you want to optionally add some features to an object but still be able to treat the object like any other of the same type. A flexible alternative to sub-classing for extending functionality. Sub-classing adds behaviour at compile-time and the change affects all instances of the original class. Decorating can provide new behaviour at run-time for individual objects.*

**Requirements.**

- *Primitive: Declares the interface for objects that can have responsibilities added to them dynamically.*
- *Leaf: Defines the class to which additional responsibilities can be attached.*
- *Decorator: Maintains a reference to the object it decorates. Decorators have the same super type as the object they decorate. Since decorators have the same type as the object, we can pass around the decorated object instead of the original.*

*In this example, the primitive is* `Pizza`*, the leaf is the* `PlainPizza`*, and the decorators are the* `Mozarella` *and* `TomatoSauce`*.*

The `Pizza` interface is the blueprint for classes that will have decorators.

```java
public interface Pizza {
  public String getDescription();
  public double getCost();
}
```

`PlainPizza` implements the `Pizza` interface with only the required methods. Every `Pizza` made will start as a `PlainPizza`.

```java
public class PlainPizza implements Pizza {
  public String getDescription() {
    return "Thin dough";
  }

  public double getCost() {
    System.out.println("Cost of Dough: " + 4.00);
    return 4.00;
  }
}
```

`ToppingDecorator` has a "has a" relationship with Pizza. This is an aggregation relationship.

```java
abstract class ToppingDecorator implements Pizza {
  protected Pizza tempPizza;

```

```
  public ToppingDecorator(Pizza newPizza) {
    tempPizza = newPizza;

  }

  public String getDescription() {
    return tempPizza.getDescription();
  }

  public double getCost() {
    return tempPizza.getCost();
  }
}
```

Our decorators are `Mozarella` and `TomatoSauce`.

```
public class Mozzarella extends ToppingDecorator {
  public Mozzarella(Pizza newPizza) {
    super(newPizza);
    System.out.println("Adding Dough");
    System.out.println("Adding Moz");
  }

  // Returns the result of calling getDescription() for
  // PlainPizza and adds " mozzarella" to it
  public String getDescription() {
    return tempPizza.getDescription() + ", mozzarella";
  }

  public double getCost() {
    System.out.println("Cost of Moz: " + .50);
    return tempPizza.getCost() + .50;
  }
}
```

```
public class TomatoSauce extends ToppingDecorator {
  public TomatoSauce(Pizza newPizza) {
    super(newPizza);
    System.out.println("Adding Sauce");
  }

  // Returns the result of calling getDescription() for
  // PlainPizza, Mozzarella and then TomatoSauce
  public String getDescription() {
    return tempPizza.getDescription() + ", tomato sauce";
  }

  public double getCost() {
    System.out.println("Cost of Sauce: " + .35);
    return tempPizza.getCost() + .35;
  }
}
```

```
public class PizzaMaker {
  public static void main(String[] args) {
    // The PlainPizza object is sent to the Mozzarella constructor
    // and then to the TomatoSauce constructor
    Pizza basicPizza = new TomatoSauce(new Mozzarella(new PlainPizza()));

    System.out.println("Ingredients: " + basicPizza.getDescription());
    System.out.println("Price: " + basicPizza.getCost());
  }
}
```

# 7 Prototype Pattern

**Purpose.** *Hide the complexity of making new instances from the client. The concept is to copy an existing object rather than create a new instance. The existing object acts as a prototype and contains the state of the object. The newly copied object may change some properties only if required. Use when there are numerous potential classes that you want to only use if needed at run-time.*

**Requirements.**

- *Prototype: Declares an interface for cloning itself.*
- *Product: The concrete prototype that implements an operation for cloning itself.*
- *Client: Creates a new object by asking a prototype to clone itself.*

By making the `Animal` interface `Cloneable`, you are telling Java that it is OK to copy instances of this class. These instance copies have different hash codes.

```
public interface Animal extends Cloneable {
  public Animal makeCopy();
}
```

```
public class Sheep implements Animal {
  public Sheep() {
    System.out.println("Sheep is Made");
  }

  public Animal makeCopy() {
    System.out.println("Sheep is Being Made");
    Sheep sheepObject = null;

    try {
      // Calls the Animal super classes clone()
      // Then casts the results to Sheep
      sheepObject = (Sheep) super.clone();
    }

    // If Animal didn't extend Cloneable this error
    // is thrown
    catch (CloneNotSupportedException e) {
      System.out.println("The Sheep was Turned to Mush");
      e.printStackTrace();
     }
    return sheepObject;
  }

  public String toString() {
    return "Dolly is my Hero, Baaaaa";
  }
}
```

`CloneFactory` receives any `Animal`, or `Animal` subclass and makes a copy of it and stores it in its own location in memory.`CloneFactory` has no idea what these objects are except that they are subclasses of `Animal`.

```
public class CloneFactory {
  public Animal getClone(Animal animalSample) {
    // Because of Polymorphism the Sheep makeCopy() is called here instead of Animal
    return animalSample.makeCopy();
  }
}
```

```
public class TestCloning {
  public static void main(String[] args){
    // Handles routing makeCopy method calls to the right subclasses of Animal
    CloneFactory animalMaker = new CloneFactory();
```

```
5
6       // Creates a new Sheep instance
7       Sheep sally = new Sheep();
8
9       // Creates a clone of Sally and stores it in its own memory location
10      Sheep clonedSheep = (Sheep) animalMaker.getClone(sally);
11
12      // These are exact copies of each other
13      System.out.println(sally);
14      System.out.println(clonedSheep);
15    }
16  }
```

# 8 Command Pattern

**Purpose.** *When you need to issue requests to objects without knowing anything about the operation being requested or the receiver of the request. Encapsulate a request as an object, thereby letting you parameterize clients with different requests, queue or log requests, and support undo-able operations.*

**Requirements.**

- *Command: Declares an interface for executing an operation.*
- *Concrete Command: Implements* `execute()` *by invoking the corresponding operation(s) on Receiver.*
- *Receiver: Knows how to perform actual operation.*
- *Invoker: Execute the operation through functional calls declared in the Command interface.*

In this example, `Television` is the receiver and `DeviceButton` is the invoker. The concrete command is `TurnTVOn` and you can imagine other concrete commands like `TurnTVOff`.

```
1  public interface ElectronicDevice {
2    public void on();
3    public void off();
4    public void volumeUp();
5    public void volumenDown();
6  }
```

```
1  public class Television implements ElectronicDevice {
2    private int volume = 0;
3
4    public void on() {
5      System.out.println("TV is on");
6    }
7
8    public void off() {
9      System.out.println("TV is off");
10   }
11
12   public void volumeUp() {
13     volume++;
14     System.out.println("TV Volume is at: " + volume);
15   }
16
17   public void volumenDown() {
18     volume--;
19     System.out.println("TV Volume is at: " + volume);
20   }
21 }
```

Each command you want to issue will implement the `Command` interface:

```java
public interface Command {
  public void execute();

  // You may want to offer the option to undo a command
  public void undo();
}
```

```java
public class TurnTVOn implements Command {
  ElectronicDevice theDevice;

  public TurnTVOn(ElectronicDevice newDevice) {
    theDevice = newDevice;
  }

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

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

You can imagine how `TurnTvOff` would look like. Next is our invoker. It has a method `press()` that, when executed, causes the `execute()` to be called. The `execute()` method for the `Command` interface then calls the method assigned in the class that implements that interface.

```java
public class DeviceButton{
  Command theCommand;

  public DeviceButton(Command newCommand){
    theCommand = newCommand;
  }

  public void press(){
    theCommand.execute();
  }

  // Now the remote can undo past commands
  public void pressUndo(){
    theCommand.undo();
  }
}
```

We create `TVRemote` to return the type of device we're going to use. You could program this to work with multiple devices.

```java
public class TVRemote {
  public static ElectronicDevice getDevice(){
    return new Television();
  }
}
```

```java
public class PlayWithRemote {
  public static void main(String[] args) {
    // Gets the ElectronicDevice to use
    ElectronicDevice newDevice = TVRemote.getDevice();

    // TurnTVOn contains the command to turn on the tv
    // When execute() is called on this command object
    // it will execute the method on() in Television
    TurnTVOn onCommand = new TurnTVOn(newDevice);

    // Calling the execute() causes on() to execute in Television
    DeviceButton onPressed = new DeviceButton(onCommand);

    // When press() is called theCommand.execute(); executes
    onPressed.press();
```

```
16
17      //-------------------------------------------------------------
18
19      // Now when execute() is called off() of Television executes
20      TurnTVOff offCommand = new TurnTVOff(newDevice);
21      onPressed = new DeviceButton(offCommand);
22      onPressed.press();
23
24      //-------------------------------------------------------------
25
26      // Now when execute() is called volumeUp() of Television executes
27      TurnTVUp volUpCommand = new TurnTVUp(newDevice);
28      onPressed = new DeviceButton(volUpCommand);
29      // When press() is called theCommand.execute(); executes
30      onPressed.press();
31      onPressed.press();
32      onPressed.press();
33    }
34  }
```

# 9 Template Method Pattern

**Purpose.** *Avoid re-implementing common behaviour by putting all the common code in the superclass and define some "hooks" to allow sub-classes to provide specialized functionality where needed. The common method in the superclass is a "template" that gets instantiated differently for each subclass.*

**Requirements.**

- *Abstract Class: Define a method (algorithm) in an* `abstract` *class. It contains both* `abstract` *and non-abstract methods.*
- *Concrete Class: The sub-classes that extend the abstract class then override those methods that don't make sense for them in the default way.*

In this example, `makeSandwich()` is the template method, and is declared `final` to keep sub-classes from changing the algorithm. The concrete classes are the different types of sandwiches like `ItalianHoagie` and `VeggieHoagie`.

```
1  public abstract class Hoagie {
2    boolean afterFirstCondiment = false;
3
4    final void makeSandwich() {
5      cutBun();
6      if(customerWantsMeat()){
7        addMeat();
8        afterFirstCondiment = true;
9      }
10
11     if (customerWantsCheese()) {
12       if (afterFirstCondiment) { System.out.print("\n"); }
13       addCheese();
14       afterFirstCondiment = true;
15     }
16
17     if (customerWantsVegetables()) {
18       if (afterFirstCondiment) { System.out.print("\n"); }
19       addVegetables();
20       afterFirstCondiment = true;
21     }
22
23     if (customerWantsCondiments()) {
24       if (afterFirstCondiment) { System.out.print("\n"); }
25       addCondiments();
26       afterFirstCondiment = true;
```

```
    }
    wrapTheHoagie();
  }

  // These methods must be overridden by the extending sub-classes
  abstract void addMeat();
  abstract void addCheese();
  abstract void addVegetables();
  abstract void addCondiments();

  public void cutBun() {
    System.out.println("The Hoagie is Cut");
  }

  // These are called hooks, which the user can override
  // Use abstract methods when you want to force the user
  // to override and use a hook when you want it to be optional
  boolean customerWantsMeat() { return true; }
  boolean customerWantsCheese() { return true; }
  boolean customerWantsVegetables() { return true; }
  boolean customerWantsCondiments() { return true; }

  public void wrapTheHoagie() {
    System.out.println("\nWrap the Hoagie");
  }

  public void afterFirstCondiment() {
    System.out.println("\n");
  }
}
```

Here is the basis of our `ItalianHoagie`.

```
public class ItalianHoagie extends Hoagie{
  String[] meatUsed = { "Salami", "Pepperoni", "Capicola Ham" };
  String[] cheeseUsed = { "Provolone" };
  String[] veggiesUsed = { "Lettuce", "Tomatoes", "Onions", "Sweet Peppers" };
  String[] condimentsUsed = { "Oil", "Vinegar" };

  public void addMeat() {
    System.out.print("Adding the Meat: ");
    for (String meat : meatUsed){

      System.out.print(meat + " ");
    }
  }

  public void addCheese() {
    System.out.print("Adding the Cheese: ");

    for (String cheese : cheeseUsed){
      System.out.print(cheese + " ");
    }
  }

  public void addVegetables() {
    System.out.print("Adding the Vegetables: ");

    for (String vegetable : veggiesUsed){
      System.out.print(vegetable + " ");
    }
  }

  public void addCondiments() {
    System.out.print("Adding the Condiments: ");

    for (String condiment : condimentsUsed){
      System.out.print(condiment + " ");
    }
```

```
  }
}
```

Next, our `VeggieHoagie` will be dramatically different.

```
public class VeggieHoagie extends Hoagie{
  String[] veggiesUsed = { "Lettuce", "Tomatoes", "Onions", "Sweet Peppers" };
  String[] condimentsUsed = { "Oil", "Vinegar" };

  boolean customerWantsMeat() { return false; }
  boolean customerWantsCheese() { return false; }

  public void addVegetables(){
    System.out.print("Adding the Vegetables: ");

    for (String vegetable : veggiesUsed) {
      System.out.print(vegetable + " ");
    }
  }

  public void addCondiments() {
    System.out.print("Adding the Condiments: ");

    for (String condiment : condimentsUsed){
      System.out.print(condiment + " ");
    }
  }
  void addMeat() {}
  void addCheese() {}
}
```

```
public class SandwichSculptor {
  public static void main(String[] args){
    ItalianHoagie cust12Hoagie = new ItalianHoagie();
    cust12Hoagie.makeSandwich();
    VeggieHoagie cust13Hoagie = new VeggieHoagie();
    cust13Hoagie.makeSandwich();
  }
}
```

# 10 Observer Pattern

**Purpose.** *Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.*

- *Push Model: The subject sends the observer a notification about all the data it will need. The observer doesn't need to query the subject for information.*
- *Pull Model: The subject merely notifies the observer that something happened, and the observer queries the subject to get the information it needs.*

Note that there is a push model and a pull model of the Observer Pattern.

**Requirements.**

- *Subject: Contains* `register()`, `unregister()`, *and* `notifyObserver()`. *The object that implements this interface will be in charge of keeping state and is alternatively called the "observable" in the observer pattern.*
- *Observer: An interface for the main observer. Classes interested in observing the state of the subject will implement this interface and only be interested in their specific domains.*

This interface handles adding, deleting, and updating all observers:

```
public interface Subject {
  public void register(Observer o);
  public void unregister(Observer o);
  public void notifyObserver();
}
```

The Observers update method is called when the Subject changes.

```
public interface Observer {
  public void update(double ibmPrice, double aaplPrice, double googPrice);
}
```

`StockGrabber` uses the Subject interface to update all Observers:

```
public class StockGrabber implements Subject {
  private ArrayList<Observer> observers; // An ArrayList to hold all observers
  private double ibmPrice;
  private double aaplPrice;
  private double googPrice;

  public StockGrabber() {
    observers = new ArrayList<Observer>();
  }

  public void register(Observer newObserver) {
    observers.add(newObserver);
  }

  public void unregister(Observer deleteObserver) {
    int observerIndex = observers.indexOf(deleteObserver);
    System.out.println("Observer " + (observerIndex+1) + " deleted");
    observers.remove(observerIndex);
  }

  public void notifyObserver() {
    // Cycle through all observers and notifies them of price changes
    for(Observer observer : observers){
      observer.update(ibmPrice, aaplPrice, googPrice);
    }
  }

  // Change prices for all stocks and notifies observers of changes
  public void setIBMPrice(double newIBMPrice){
    this.ibmPrice = newIBMPrice;
    notifyObserver();
  }

  public void setAAPLPrice(double newAAPLPrice){
    this.aaplPrice = newAAPLPrice;
    notifyObserver();
  }

  public void setGOOGPrice(double newGOOGPrice){
    this.googPrice = newGOOGPrice;
    notifyObserver();
  }
}
```

`StockObserver` represents each Observer that is monitoring changes in the subject:

```
public class StockObserver implements Observer {
  private double ibmPrice;
  private double aaplPrice;
  private double googPrice;

  // Static used as a counter
  private static int observerIDTracker = 0;

  // Used to track the observers
  private int observerID;
```

```
11
12    // Will hold reference to the StockGrabber object
13    private Subject stockGrabber;
14
15    public StockObserver(Subject stockGrabber) {
16
17      // Store the reference to the stockGrabber object so
18      // I can make calls to its methods
19      this.stockGrabber = stockGrabber;
20
21      // Assign an observer ID and increment the static counter
22      this.observerID = ++observerIDTracker;
23
24      // Message notifies user of new observer
25      System.out.println("New Observer " + this.observerID);
26
27      // Add the observer to the Subjects ArrayList
28      stockGrabber.register(this);
29    }
30
31    // Called to update all observers
32
33    public void update(double ibmPrice, double aaplPrice, double googPrice) {
34      this.ibmPrice = ibmPrice;
35      this.aaplPrice = aaplPrice;
36      this.googPrice = googPrice;
37      printThePrices();
38    }
39
40    public void printThePrices(){
41      System.out.println(observerID + "\nIBM: " + ibmPrice + "\nAAPL: " +
42          aaplPrice + "\nGOOG: " + googPrice + "\n");
43    }
44  }
```

Now for testing the pattern.

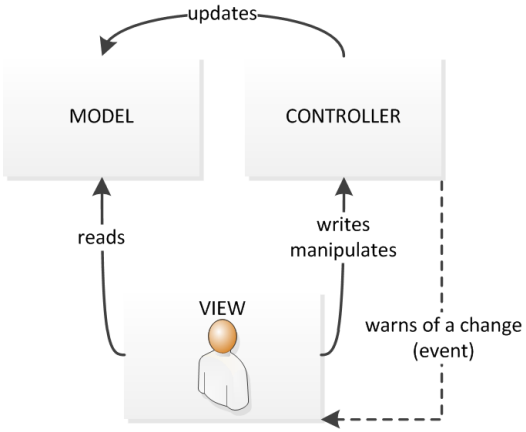
```
1   public class GrabStocks{
2     public static void main(String[] args){
3       // Create the Subject object
4       // It will handle updating all observers as well as deleting and adding them
5       StockGrabber stockGrabber = new StockGrabber();
6
7       // Create an Observer that will be sent updates from Subject
8       StockObserver observer1 = new StockObserver(stockGrabber);
9
10      stockGrabber.setIBMPrice(197.00);
11      stockGrabber.setAAPLPrice(677.60);
12      stockGrabber.setGOOGPrice(676.40);
13
14      StockObserver observer2 = new StockObserver(stockGrabber);
15
16      stockGrabber.setIBMPrice(197.00);
17      stockGrabber.setAAPLPrice(677.60);
18      stockGrabber.setGOOGPrice(676.40);
19
20      // Delete one of the observers
21      // stockGrabber.unregister(observer2);
22      stockGrabber.setIBMPrice(197.00);
23      stockGrabber.setAAPLPrice(677.60);
24      stockGrabber.setGOOGPrice(676.40);
25    }
26  }
```

# 11 MVC Pattern

**Purpose.** *Specifies that an application consist of a data model, presentation information, and control information. The pattern requires that each of these be separated into different objects. MVC is more of an architectural pattern.*

**Requirements.**

- *Model: Contains only the pure application data, it contains no logic describing how to present the data to a user.*
- *View: Presents the model's data to the user. The view knows how to access the model's data, but it does not know what this data means or what the user can do to manipulate it.*
- *Controller: Exists between the view and the model. It listens to events triggered by the view (or another external source) and executes the appropriate reaction to these events. In most cases, the reaction is to call a method on the model. Since the view and the model are connected through a notification mechanism, the result of this action is then automatically reflected in the view.*




The Model performs all the calculations needed and that is it. It doesn't know the View exists.

```
public class CalculatorModel {
  private int calculationValue;

  public void addTwoNumbers(int firstNumber, int secondNumber){
    calculationValue = firstNumber + secondNumber;
  }

  public int getCalculationValue() {
    return calculationValue;
  }
}
```

This is the View. Its only job is to display what the user sees. It performs no calculations, but instead passes information entered by the user to whomever needs it.

```
import java.awt.event.ActionListener;
import javax.swing.*;

public class CalculatorView extends JFrame {
  private JTextField firstNumber  = new JTextField(10);
  private JLabel additionLabel = new JLabel("+");
  private JTextField secondNumber = new JTextField(10);
  private JButton calculateButton = new JButton("Calculate");
  private JTextField calcSolution = new JTextField(10);

  CalculatorView(){
```

```
    // Sets up the view and adds the components
    JPanel calcPanel = new JPanel();

    this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    this.setSize(600, 200);

    calcPanel.add(firstNumber);
    calcPanel.add(additionLabel);
    calcPanel.add(secondNumber);
    calcPanel.add(calculateButton);
    calcPanel.add(calcSolution);

    this.add(calcPanel);
  }

  public int getFirstNumber(){
    return Integer.parseInt(firstNumber.getText());
  }

  public int getSecondNumber(){
    return Integer.parseInt(secondNumber.getText());
  }

  public int getCalcSolution(){
    return Integer.parseInt(calcSolution.getText());
  }

  public void setCalcSolution(int solution){
    calcSolution.setText(Integer.toString(solution));
  }

  // If the calculateButton is clicked execute a method
  // in the Controller named actionPerformed
  void addCalculateListener(ActionListener listenForCalcButton){
    calculateButton.addActionListener(listenForCalcButton);
  }

  // Open a popup that contains the error message passed
  void displayErrorMessage(String errorMessage){
    JOptionPane.showMessageDialog(this, errorMessage);
  }
}
```

The Controller coordinates interactions between the View and Model.

```
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;

public class CalculatorController {
  private CalculatorView theView;
  private CalculatorModel theModel;

  public CalculatorController(CalculatorView theView, CalculatorModel theModel) {
    this.theView = theView;
    this.theModel = theModel;

    // Tell the View that when ever the calculate button
    // is clicked to execute the actionPerformed method
    // in the CalculateListener inner class
    this.theView.addCalculateListener(new CalculateListener());
  }

  class CalculateListener implements ActionListener {

    public void actionPerformed(ActionEvent e) {
      int firstNumber, secondNumber = 0;

      // Surround interactions with the view with
      // a try block in case numbers weren't
```

```
      // properly entered
      try {

        firstNumber = theView.getFirstNumber();
        secondNumber = theView.getSecondNumber();

        theModel.addTwoNumbers(firstNumber, secondNumber);

        theView.setCalcSolution(theModel.getCalculationValue());

      }

      catch(NumberFormatException ex){
        System.out.println(ex);
        theView.displayErrorMessage("You Need to Enter 2 Integers");
      }
    }
  }
}
```

Here in `MVCCalculator` we create all the objects.

```
public class MVCCalculator {
    public static void main(String[] args) {
      CalculatorView theView = new CalculatorView();
      CalculatorModel theModel = new CalculatorModel();
        CalculatorController theController = new CalculatorController(theView,theModel);
        theView.setVisible(true);
    }
}
```

# 12 Visitor Pattern

**Purpose.** *The visitor represents an operation to be performed on the elements of an object structure. It is used when you have to perform the same action on many objects of different types. Lets you define a new operation without changing the classes of the elements on which it operates.*

**Requirements.**

- *Visitor: This is an interface or an abstract class used to declare the* `visit()` *operations for all the types of visitable classes. Created to automatically use the right code based on the Object sent.*
- *Concrete Visitor: For each type of visitor, all the* `visit()` *methods declared in the abstract visitor must be implemented. Each Visitor will be responsible for different functions.*
- *Visitable: This is an interface which declares the* `accept()` *operation. Allows the Visitor to pass the object so the right operations occur on the right type of object.*
- *Concrete Visitable: These classes implement the* `Visitable` *interface or class and defines the* `accept()` *operation. The visitor object is passed to this object using the* `accept()` *operation.*

Here is our `Visitor` interface:

```
interface Visitor {
    public double visit(Liquor liquorItem);
    public double visit(Tobacco tobaccoItem);
    public double visit(Necessity necessityItem);
}
```

Our Concrete Visitor classes are `TaxVisitor` and `HolidayTaxVisitor`.

```
class TaxVisitor implements Visitor {
    // This formats the item prices to 2 decimal places
    DecimalFormat df = new DecimalFormat("#.##");
```

```
    // This is created so that each item is sent to the // right version of visit() which is
     required by the
    // Visitor interface and defined below
    public TaxVisitor () {}

    // Calculates total price based on this being taxed
    // as a liquor item
    public double visit(Liquor liquorItem) {
        System.out.println("Liquor Item: Price with Tax");
        return Double.parseDouble(df.format((liquorItem.getPrice() * .18) + liquorItem.
    getPrice ()));
    }

    // Calculates total price based on this being taxed // as a tobacco item
    public double visit(Tobacco tobaccoItem) {
        System.out.println("Tobacco Item: Price with Tax");
        return Double.parseDouble(df.format((tobaccoItem.getPrice() * .32) + tobaccoItem.
    getPrice()));
    }

    // Calculates total price based on this being taxed // as a necessity item
    public double visit(Necessity necessityItem) {
        System.out.println("Necessity Item: Price with Tax");
        return Double.parseDouble(df.format(necessityItem.getPrice())); }
    }
```

```
class TaxHolidayVisitor implements Visitor {
    DecimalFormat df = new DecimalFormat("#.##");

    public TaxHolidayVisitor() {}

    public double visit(Liquor liquorItem) {
        System.out.println("Liquor Item: Price with Tax");
        return Double.parseDouble(df.format((liquorItem.getPrice() * .10) + liquorItem.
    getPrice()));
    }

    public double visit(Tobacco tobaccoItem) {
        System.out.println("Tobacco Item: Price with Tax");
        return Double.parseDouble(df.format((tobaccoItem.getPrice() * .30) + tobaccoItem.
    getPrice()));
    }

    public double visit(Necessity necessityItem) {
        System.out.println("Necessity Item: Price with Tax");
        return Double.parseDouble(df.format(necessityItem.getPrice()));
    }
}
```

Here is our `Visitable` interface. `accept()` is passed the same visitor object but then the method `visit()` is called using the visitor object. The right version of `visit()` is called because of method overloading.

```
interface Visitable {
    public double accept(Visitor visitor);
}
```

Our concrete visitable objects are `Liquor`, `Necessity`, and `Tobacco`.

```
class Liquor implements Visitable {
    private double price;

    Liquor(double item) {
        price = item;
    }

    public double accept(Visitor visitor) {
        return visitor.visit(this);
    }

```

```
    public double getPrice () {
        return price;
    }
}
```

```
class Necessity implements Visitable {
    private double price;
    Necessity(double item) {
        price = item;
    }

    public double accept(Visitor visitor) {
        return visitor.visit(this);
    }

    public double getPrice () {
        return price;
    }
}
```

```
class Tobacco implements Visitable {
    private double price;

    Tobacco(double item) {
        price = item;
    }

    public double accept(Visitor visitor) {
        return visitor.visit(this);
    }

    public double getPrice () {
        return price;
    }
}
```

```
public class VisitorTest {
    public static void main(String[] args) {
        TaxVisitor taxCalc = new TaxVisitor();
        TaxHolidayVisitor taxHolidayCalc = new TaxHolidayVisitor();

        Necessity milk = new Necessity(3.47);
        Liquor vodka = new Liquor(11.99);
        Tobacco cigars = new Tobacco(19.99);

        System.out.println(milk.accept(taxCalc) + "\n");
        System.out.println(vodka.accept(taxCalc) + "\n");
        System.out.println(cigars.accept(taxCalc) + "\n");

        // TAX HOLIDAY PRICES
        System.out.println(milk.accept(taxHolidayCalc) + "\n");
        System.out.println(vodka.accept(taxHolidayCalc) + "\n");
        System.out.println(cigars.accept(taxHolidayCalc) + "\n");
    }
}
```

# 13 Factory Method Pattern

**Purpose.** *When you want to define the class of an object at runtime, i.e. perhaps you don't know ahead of time what class object you need. It also allows you to encapsulate object creation so that you can keep all object creation code in one place.*

A method returns one of several possible classes that share a common superclass. The class is chosen at runtime.

**Requirements.**

- *Factory: Includes a factory method for making objects.*
- *Products: The object we are going to produce through the factory,*
- *Product Superclass: The supertype of the products.*

```
abstract class Plan {
    protected double rate;
    abstract void getRate();

    public void calculateBill(int units) {
        System.out.println(units*rate);
    }
}
```

```
class DomesticPlan extends Plan {
    @Override
    public void getRate() {
        rate = 3.50;
    }
}
```

```
class CommercialPlan extends Plan {
   @Override
    public void getRate() {
        rate = 7.50;
   }
}
```

```
class InstitutionalPlan extends Plan {
    @Override
    public void getRate() {
        rate = 5.50;
}
```

```
class PlanFactory{
    public Plan getPlan(String planType) {
        if (planType.equalsIgnoreCase("DOMESTICPLAN")) {
            return new DomesticPlan();
        }

        else if (planType.equalsIgnoreCase("COMMERCIALPLAN")) {
            return new CommercialPlan();
        }

        else if (planType.equalsIgnoreCase("INSTITUTIONALPLAN")) {
            return new InstitutionalPlan();
        }

        return null;
   }
}
```

Below is how we would get an object of type `Plan` depending on the input to the factory method `getPlan()`:

```
class TestPlanFactory {
    public static void main(String args[]) {
      PlanFactory planFactory = new PlanFactory();
      Plan p = planFactory.getPlan("DOMESTICPLAN");
    }
}
```

# 14 Abstract Factory Pattern

**Purpose.** *Another layer of abstraction over the factory method pattern. This pattern works around a super factory which creates other factories. Provides a framework that allows us to create objects that follow a general pattern.*

**Requirements.**

- *Abstract Factory: Declares an interface that is used to create abstract product objects.*
- *Concrete Factory: Implements the operations declared in the Abstract Factory to create concrete product objects.*
- *Abstract Product: Declares an interface for a specific type of product object.*
- *Concrete Product: Provided by its Concrete Factory and defines the individual implementation details for the specified product through the implementation of its Abstract Product interface.*

We are going to create a `Shape` interface and concrete classes implementing this interface.

```
public interface Shape {
   void draw();
}
```

```
public class RoundedRectangle implements Shape {
   @Override
   public void draw() {
      System.out.println("Inside RoundedRectangle::draw() method.");
   }
}
```

```
public class RoundedSquare implements Shape {
   @Override
   public void draw() {
      System.out.println("Inside RoundedSquare::draw() method.");
   }
}
```

```
public class Rectangle implements Shape {
   @Override
   public void draw() {
      System.out.println("Inside Rectangle::draw() method.");
   }
}
```

We create an abstract factory class `AbstractFactory` as the next step. The factory classes `ShapeFactory` and `RoundedShapeFactory` are defined where each factory extends `AbstractFactory`.

```
public abstract class AbstractFactory {
   abstract Shape getShape(String shapeType);
}
```

```
public class ShapeFactory extends AbstractFactory {
   @Override
   public Shape getShape(String shapeType){
      if(shapeType.equalsIgnoreCase("RECTANGLE")){
         return new Rectangle();
      }else if(shapeType.equalsIgnoreCase("SQUARE")){
         return new Square();
      }
      return null;
   }
}
```

```
public class RoundedShapeFactory extends AbstractFactory {
   @Override
   public Shape getShape(String shapeType){
      if(shapeType.equalsIgnoreCase("RECTANGLE")){
         return new RoundedRectangle();
      }else if(shapeType.equalsIgnoreCase("SQUARE")){
         return new RoundedSquare();
      }
      return null;
   }
}
```

A factory generator class `FactoryProducer` is created.

```
public class FactoryProducer {
   public static AbstractFactory getFactory(boolean rounded){
      if(rounded){
         return new RoundedShapeFactory();
      }else{
         return new ShapeFactory();
      }
   }
}
```

Use the `FactoryProducer` to get factories of concrete classes by passing information of such type.

```
public class AbstractFactoryPatternDemo {
   public static void main(String[] args) {
      // Get a regular shape factory
      AbstractFactory shapeFactory = FactoryProducer.getFactory(false);

      Shape rectangle = shapeFactory.getShape("RECTANGLE");
      rectangle.draw();

      Shape square = shapeFactory.getShape("SQUARE");
      square.draw();

      // Get a rounded shape factory
      AbstractFactory roundedShapeFactory = FactoryProducer.getFactory(true);

      Shape roundedRectangle = roundedShapeFactory.getShape("RECTANGLE");
      roundedRectangle.draw();

      Shape roundedSquare = roundedShapeFactory.getShape("SQUARE");
      roundedSquare.draw();
   }
}
```

# 15 Adapter Pattern

**Purpose.** *When you want to translate one interface of a class into another interface. Lets classes work together that could not otherwise because of incompatible interfaces.*

**Requirements.**

- *Target: The interface we want to adapt new classes to.*
- *Adapter: Implements the target interface. This is the class we will use to adapt the adaptee to the target.*
- *Adaptee: The class we want to adapt to the target.*

This is the target interface. This is what the client expects to work with. It is the adapter's job to make new classes compatible with this one.

```
public interface EnemyAttacker {
  public void fireWeapon();
  public void driveForward();
}
```

`EnemyTank` implements `EnemyAttacker` perfectly. Our job is to make classes with different methods work with the `EnemyAttacker` interface.

```
public class EnemyTank implements EnemyAttacker {
  Random generator = new Random();

  public void fireWeapon() {
    int attackDamage = generator.nextInt(10) + 1;
    System.out.println("Enemy Tank Does " + attackDamage + " Damage");
  }

  public void driveForward() {
    int movement = generator.nextInt(5) + 1;
    System.out.println("Enemy Tank moves " + movement + " spaces");
  }
}
```

This is the Adaptee. The Adapter sends method calls to objects that use the `EnemyAttacker` interface to the right methods defined in `EnemyRobot`.

```
public class EnemyRobot {
  Random generator = new Random();

  public void smashWithHands() {
    int attackDamage = generator.nextInt(10) + 1;
    System.out.println("Enemy Robot Causes " + attackDamage + " Damage With Its Hands");
  }

  public void walkForward() {
    int movement = generator.nextInt(5) + 1;
    System.out.println("Enemy Robot Walks Forward " + movement + " spaces");
  }
}
```

The Adapter must provide an alternative action for the the methods that need to be used because `EnemyAttacker` was implemented. This Adapter does this by containing an object of the same type as the Adaptee, `EnemyRobot`. All calls to `EnemyAttacker` methods are sent instead to methods used by `EnemyRobot`.

```
public class EnemyRobotAdapter implements EnemyAttacker {
  EnemyRobot theRobot;

  public EnemyRobotAdapter (EnemyRobot newRobot) {
    theRobot = newRobot;
  }

  public void fireWeapon() {
    theRobot.smashWithHands();
  }

  public void driveForward() {
    theRobot.walkForward();
  }
}
```

The Adaptee is sent as a parameter argument to the Adapter's constructor. Methods from the target interface can then be called on the Adapter object that contains the Adaptee.

```
public class TestEnemyAttackers {
  public static void main(String[] args){
    EnemyTank rx7Tank = new EnemyTank();
    EnemyRobot fredTheRobot = new EnemyRobot();
    EnemyAttacker robotAdapter = new EnemyRobotAdapter(fredTheRobot);

    // The Robot
```

```
    fredTheRobot.walkForward();
    fredTheRobot.smashWithHands();

    // The Enemy Tank
    rx7Tank.driveForward();
    rx7Tank.fireWeapon();

    // The Robot with Adapter
    robotAdapter.driveForward();
    robotAdapter.fireWeapon();
  }
}
```

# 16 Proxy Pattern

**Purpose.** *Provides a class which will limit access to another class. Think of it as a gatekeeper that will block access to another object.*

In this example, we throw a proxy object `ATMProxy` in between the client and the `ATMMachine` object. We will limit access to fields in the `ATMMachine` to just getting the ATM's current state and getting cash in the machine.

```
public class ATMMachine implements GetATMData{
  public ATMState getYesCardState() { return hasCard; }
  public ATMState getNoCardState() { return noCard; }
  public ATMState getHasPin() { return hasCorrectPin; }
  public ATMState getNoCashState() { return atmOutOfMoney; }
  public ATMState getATMState() { return atmState; }
  public int getCashInMachine() { return cashInMachine; }
}
```

This interface will contain just those methods that you want the proxy to provide access to.

```
public interface GetATMData{
    public ATMState getATMState();
    public int getCashInMachine();
}
```

In this situation the proxy both creates and destroys an `ATMMachine` object.

```
public class ATMProxy implements GetATMData {
  // Allows the user to access getATMState in the object ATMMachine
  public ATMState getATMState() {
    ATMMachine realATMMachine = new ATMMachine();
    return realATMMachine.getATMState();
  }

  // Allows the user to access getCashInMachine in the object ATMMachine
  public int getCashInMachine() {
    ATMMachine realATMMachine = new ATMMachine();
    return realATMMachine.getCashInMachine();
  }
}
```

Note that the user will not be able to perform `atmProxy.setCashInMachine(10000)` because `ATMProxy` does not have access to that potentially dangerous method. Performing `realATMMachine.setCashInMachine(10000)` will not work either because `realATMMachine` is of type `GetATMData` where we defined limited access.

```
public class TestATMMachine {
  public static void main(String[] args){
    // The interface limits access to just the methods you want made accessible
    GetATMData realATMMachine = new ATMMachine();
    GetATMData atmProxy = new ATMProxy();

    System.out.println("Current ATM State " + atmProxy.getATMState());
    System.out.println("Cash in ATM Machine $" + atmProxy.getCashInMachine());
  }
}
```