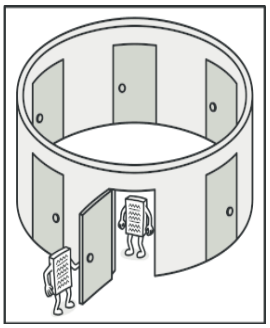
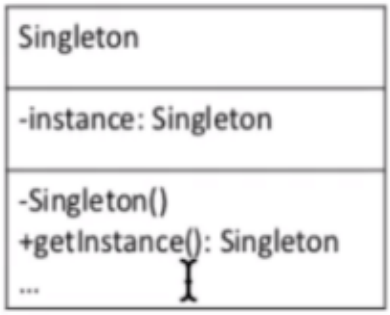
BY GABRIEL VAINER

1. Singleton-CREATIONAL





This pattern involves a single class which is responsible for creating an object while making sure that only a single object gets created.

Every Singleton implementation will all have three key factors in common:

1. All fields must be Private
2. The constructor must be Private and not do anything
3. Must contain a method to get the instance of the private information. Better implementations involve a check to see if an instance was already made

Driver.java | Captain.java

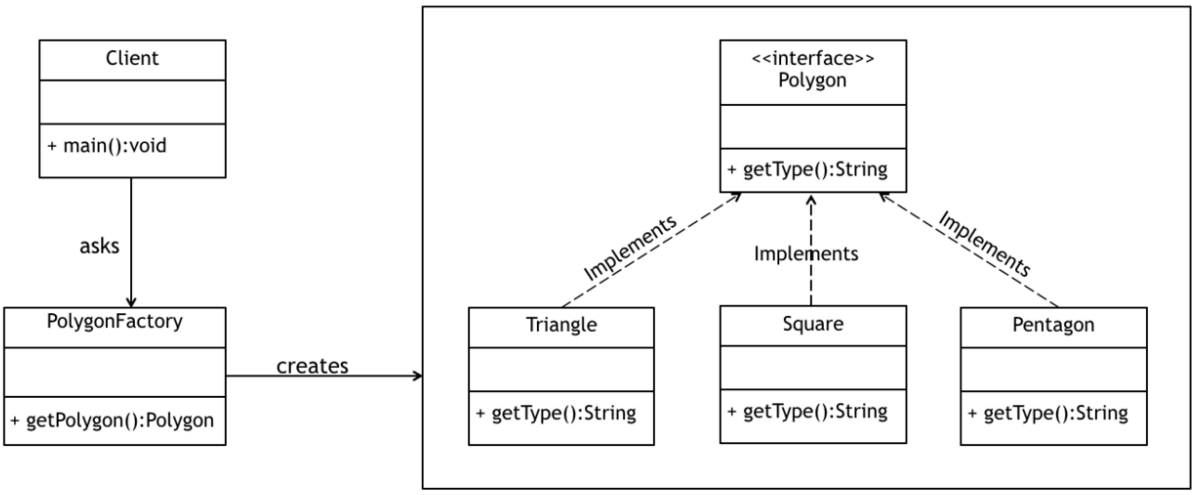
```
1  package example;
2
3  public class Captain {
4
5      private static Captain captain;
6
7      private Captain() {
8
9      }
10
11     public static Captain getCaptain() {
12         if(captain == null) {
13             captain = new Captain();
14             System.out.println("New captain is created");
15         }
16         else{
17             System.out.println("Captain already created");
18         }
19         return captain;
20     }
21
22 }
```

```
public class SingleObject {

   //create an object of SingleObject
   private static SingleObject instance = new SingleObject();

   //make the constructor private so that this class cannot be
   //instantiated
   private SingleObject(){}

   //Get the only object available
   public static SingleObject getInstance(){
      return instance;
   }

   public void showMessage(){
      System.out.println("Hello World!");
   }
}
```

This pattern is relatively straightforward. We want to make our software in such a way that there only exists one instance for our class. For example, if we were to take Captain.java(top right picture) and put it under a test driver, we would have to design it in such a way where we cannot create multiple Captains. With that being said, the most common solution is within the get instance method. As displayed on Captain.java, we only instantiate a new captain if the instance has not been created (if there is a captain that already exists).

Think of Singleton like a president's relationship to a government. During a presidential term, there can only be one active president at a time, meaning there is not more than one person with the same job. Shifting this into java terminology, the president would act as a class, and the singleton implementation would ensure that there will not be more than one instance of president.

2. **Factory-CREATIONAL** (does the picture below remind you of something similar we've done before?)



This pattern offers a unique way to create a new object such that we can create an object without exposing the creation logic to the client and refer to newly created objects using a common interface. This also implies that the subclasses determine the type of objects being made when invoked.

There are many ways to implement a factory pattern, but generally as long as **(1)**you're using some kind of interface for your subclasses as well as **(2)** a factory class that generates an object of a concrete class based on given information, you have a factory. Let's use the map above to create a polygon factory.

Step 1: Set up your interface and subclasses

Lets define the Polygon interface and the 3 shapes that will implement it.\

```
1  package example;
2
3  interface Polygon{
4      String getType();
5  }
6
7  class Circle implements Polygon{
8
9      @Override
10     public String getType() {
11         // TODO Auto-generated method stub
12         return "Circle";
13     }
14
15 }
16
17 class Triangle implements Polygon{
18
19     @Override
20     public String getType() {
21         // TODO Auto-generated method stub
22         return "Triangle";
23     }
24
25 class Pentagon implements Polygon{
26
27     @Override
28     public String getType() {
29         // TODO Auto-generated method stub
30         return "Pentagon";
31     }
32
33 }
```

Nothing new right? We've done this before, no new skills that we have to apply

Step 2: Set up your factory

Lets make a factory class that generates an object of a concrete class based on given information. We will call it PolygonFactory.

```
class PolygonFactory{

    public Polygon getPolygon(String s) {

        if(s.equals("Circle")){
            return new Circle();
        }
        if(s.equals("Triangle")){
            return new Triangle();
        }
        if(s.equals("Pentagon")){
            return new Pentagon();
        }
    }
}
```

Option #1

This is not the only way of implementing a factory. As I said in the last page, there's many ways to implement a factory pattern

**Here's an alternative factory approach. This one is very similar to what Rawad did in lecture 5. Kindly note that both methods are valid approaches, the only real difference behind them is how the client is made (the driver.java file to see the type of output), as we will see in a moment.

OBJ

Let's say we wanted to make a driver that gives us all of the polygons(we want to have an output of **Circle /n Triangle /n Pentagon**). As stated previously, these options, while having the same functionality, will produce different drivers depending on how you make it. Let's observe:

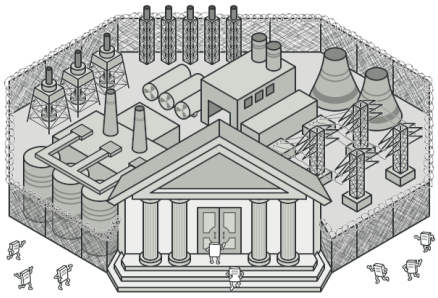
Driver.java for option 1

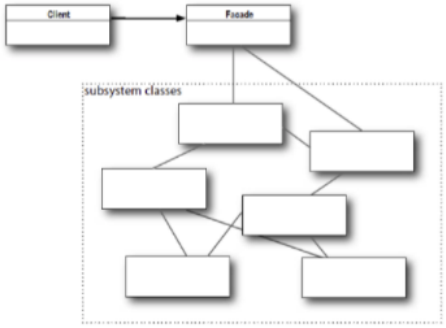
```
package example;

public class Driver {

    public static void main(String args[]) {

        PolygonFactory factory = new PolygonFactory();
        Polygon c = factory.getPolygon("Circle");
        Polygon t = factory.getPolygon("Triangle");
        Polygon p = factory.getPolygon("Pentagon");

    }

}
```

Driver.java for option2

```
1  package example;
2
3  public class Driver {
4
5      public static void main(String args[]) {
6          PolygonFactory c = new CircleFactory();
7          Polygon p1 = c.getPolygon();
8          System.out.println(p1.getType());
9
0          PolygonFactory t = new TriangleFactory();
1          Polygon p2 = t.getPolygon();
2          System.out.println(p2.getType());
3
4          PolygonFactory p = new PentagonFactory();
5          Polygon p3 = p.getPolygon();
6          System.out.println(p3.getType());
7      }
8
9  }
```

3. Facade-STRUCTURAL





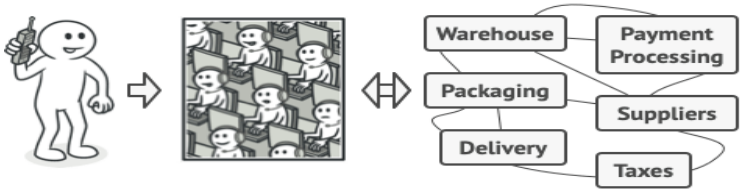
This pattern provides convenient access to a particular part of the subsystem's functionality. It knows where to direct the client's request and how to operate all the moving parts. This allows the client experience to become easier as well as allowing the programmer to isolate their code from the entire complexity of a sub-system.

Every Facade Implementation has 5 key components:

1. A facade class
2. Fields within the class include the concrete classes to be unified/put into one.
3. A constructor consisting of filling in the required fields for the facade
4. A method of some sort that incorporates the functionality of all the concrete classes into one operation.
5. The Client uses the facade instead of calling the subsystem objects directly.

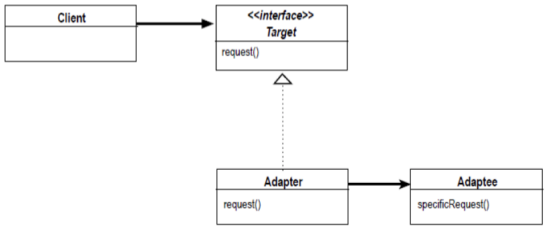
```
class HomeTheatherFacade{
    Screen s;
    Light l;
    Projector p;

    public HomeTheatherFacade(Screen s, Light l, Projector p) {
        this.s = s;
        this.l = l;
        this.p = p;
    }

    public void watchMovie() {
        s.down();
        l.dim(10);
        p.on();
    }

}
```

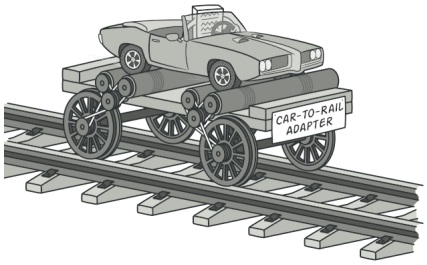
```
public class Driver {


    public static void main(String [] args) {
        HomeTheaterFacade h = new HomeTheaterFacade(,,);
        h.watchMovie();
    }
}
```



Think of Facade as placing a phone order. The operator acts as the facade to all of the departments and services. The operator conveniently combines all of the services and departments needed to make an order. Without the operator, you would need to call all of these services just to make one order (you gotta be kidding me)! With the help of the operator, you only have to call once to make the order. Shifting this into java terminology, the operator is the Facade class that combines the sub-system and the sub-system of departments and services to represent the concrete classes that are needed to perform a unified operation.

4. Adapter-STRUCTURAL





This pattern lets classes with incompatible interfaces work together.

This pattern involves a single class which is responsible for joining functionalities of independent or incompatible interfaces. Rawad's lecture notes demonstrate a great and classic implementation of the adapter pattern. Let's observe:

Step 0: Bruh

```
interface Duck{
    public void quack();
}

class DomesticDuck implements Duck{
    public void quack() {
        System.out.println("quack");
    }
}

interface Turkey{
    public void gobble();
}

class WildTurkey implements Turkey{
    public void gobble() {
        System.out.println("gobble");
    }
}
```

Oh no! Looks like we have two distinct concrete classes with incompatible interfaces, despite the fact that they have the same functionality! We should find a way to combine these two. Let's try to make a turkey quack.

Step 1: "QUACK"- Turkey

```
class TurkeyAdapter implements Duck{
    Turkey turkey;

    public TurkeyAdapter(Turkey turkey) {
        this.turkey = turkey;
    }

    public void quack() {
        turkey.gobble();
    }
}
```

Take a moment to understand what's happening over here. Our "adapter" is in the form of a class. It has a field of one of the concrete classes and implements its non corresponding interface (A **turkey** adapter that has a field of **Turkey** (concrete class) implementing **Duck** (turkey's non corresponding interface)). This is where the juicy stuff happens. Because we implemented Duck, we need to implement all of its methods. We know that the turkey-equivalent of quacking is gobbling, so when we override the quack method for turkey, we will make the turkey gobble.

Step 2: Why did we do this?

```
public class Driver {

    public static void main(String [] args) {
        WildTurkey w = new WildTurkey();
        TurkeyAdapter t = new TurkeyAdapter(w);
        t.quack();
    }
}
```

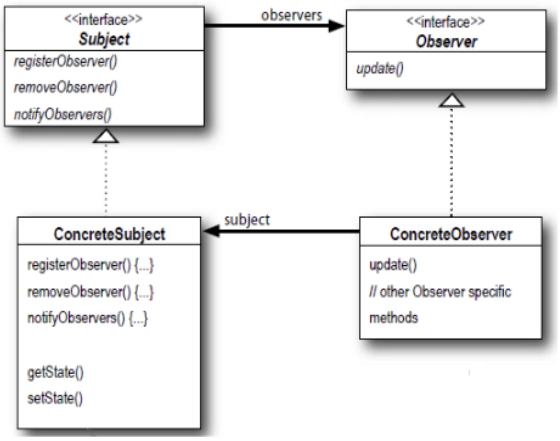
Now look what we did over here! By overriding the Duck**(non-corresponding)** interface to a class that takes in turkey**(concrete class)** as a field, we allowed a concrete class to work with an incompatible interface.

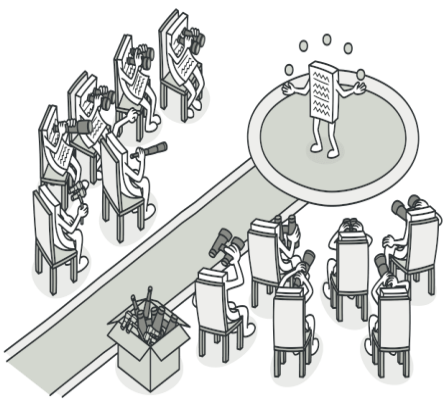
Bonus Step : Food for thought.

So we made a turkey quack. Very nice. While we haven't technically fully adapted both classes to work together by just adapting turkey, we could fully adapt by adapting duck, which would be just as simple. How would we do it? Using the exact same logic as the turkey adapter

We would define a duckAdapter class such that it would take in a field of **Duck**(concrete class) implementing **Turkey**(Ducks's non corresponding interface). When we override gobble, we will make the duck quack (duck.quack()). Ezpz

5. Observer-BEHAVIOURAL





This pattern is used when there is one-too-many relationships between objects such as if one object is modified, its dependent objects are to be notified automatically. Typically, the Observer pattern is used when changes to the state of one object may require changing other objects, and the actual set of objects is unknown beforehand or changes dynamically.

Every Observer implementation will all have eight key factors in common:

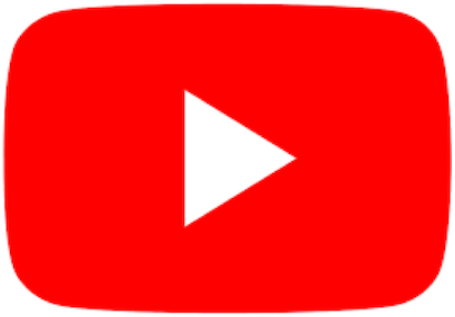
```
1  package example;
2
3  import java.util.List;
4  import java.util.ArrayList;
5
6  class Subject{
7      List<Observer> observers = new ArrayList<Observer>();
8      int state;
9
10     public int getState() {
11         return state;
12     }
13
14     public void setState(int n) {
15         state = n;
16         notifyObservers();
17     }
18
19     public void notifyObservers() {
20         for(Observer obs:observers) {
21             obs.update();
22         }
20     }
21
22     public void attach(Observer obs) {
23         observers.add(obs);
24     }
25 }
abstract class Observer{
    Subject subject;
    public abstract void update();
}
```

BY GABRIEL VAINER

```
class ObserverA extends Observer{
    public ObserverA(Subject subject) {
        this.subject = subject;
        this.subject.attach(this);
    }

    public void update() {
        System.out.println("A observed " + subject.state);
    }
}

class ObserverB extends Observer{
    public ObserverB(Subject subject) {
        this.subject = subject;
        this.subject.attach(this);
    }

    public void update() {
        System.out.println("B observed " + subject.state);
    }
}
```

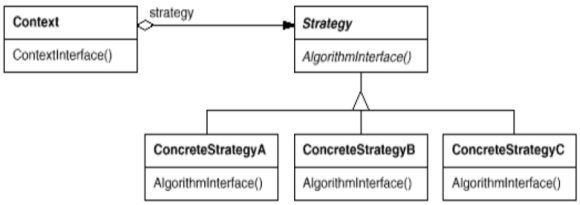
```
public class Driver {

    public static void main(String [] args) {
        Subject subject = new Subject();
        new ObserverA(subject);
        new ObserverB(subject);
        subject.setState(100);
    }
}
```

1. Subject Class- typically has two private fields: an int representing the state, and a set of observers.
2. getState()- because state is a private field that cannot be accessed by the client. Retrieves the state.
3. setState()- updates the state and notifies all of the observers immediately.
4. notifyObservers()- for every observer within the set of observers, update the information and show proof
5. attach()- add an observer to the set of observers
6. Observer Class- an abstract class - typically has a protected field of type Subject. Contains an abstract method update()
7. Concrete Observer Classes- extends Observer. Overrides update().
8. When we run this particular driver, the output should show that both ObserverA and ObserverB's state has changed.



Think of Observer as a Youtube subscription. The benefit of subscribing to a channel is that you don't have to manually check when a content creator releases a new video, you'll automatically be sent a notification saying that x channel has released a new video. Shifting this idea to java terminology, when you subscribe to a channel, you are an observer. Your account gets added(attatch()) to a set of all subscribers (other observers) and when a new video gets released(setState()), all of the subscribers are sent a notification (notifyObservers()) and are updated accordingly (update()).

6. Strategy-BEHAVIOURAL



This pattern lets you define a family of algorithms, put each of them into a separate class, and make their objects interchangeable.In Strategy pattern, a class behavior or its algorithm can be changed at run time, which is given by some sort of context.

There are many ways to implement a strategy pattern. Rawad's lecture notes demonstrate a great and classic implementation of the adapter pattern. Let's observe:

Step 1: Make an interface

```
interface Strategy{
    public int doAlgorithm(int a, int b);
}
```

Step 2: Create concrete classes implementing the same interface

```
class StrategyAdd implements Strategy{
    public int doAlgorithm(int a, int b) {
        return a+b;
    }
}

class StrategySubtract implements Strategy{
    public int doAlgorithm(int a, int b) {
        return a-b;
    }
}
```

You can create as many concrete classes (strategies) as you'd like, as long as they implement the interface

Step 3: Create a Context Class

```
class Context{
    Strategy strategy;

    public Context(Strategy strategy) {
        this.strategy = strategy;
    }

    public int executeStrategy(int a, int b) {
        return strategy.doAlgorithm(a, b);
    }
}
```

Things are gettin a lil' **crunchy** (Paco 3:16). Over here, we created a new class Context that has (usually a private) Strategy field. The constructor takes in a strategy and assigns it to the field. Then, we have a concrete method execute(Strategy) that takes in two integers and performs the expected strategy algorithm.

Step 4 : Game Ending Credits

```
public static void main(String [] args) {
    Context c = new Context(new StrategyAdd());
    System.out.println(c.executeStrategy(5, 4));
    c = new Context(new StrategySubtract());
    System.out.println(c.executeStrategy(5, 4));
}
```

You did it: you've successfully added a strategy that works such that the algorithm is changed by runtime given by the context. Notice how we call a new strategy. We initialized a new context and assigned its strategy field to strategyadd. So, when we executed the strategy, we performed the chosen algorithm given by our context, which in this case was adding, so therefore we add the two numbers.

Mess around with Context in the driver a little bit and see what happens. Try adding some more strategies.

Good luck on your midterm!