CS130: Software Engineering

Lab 4

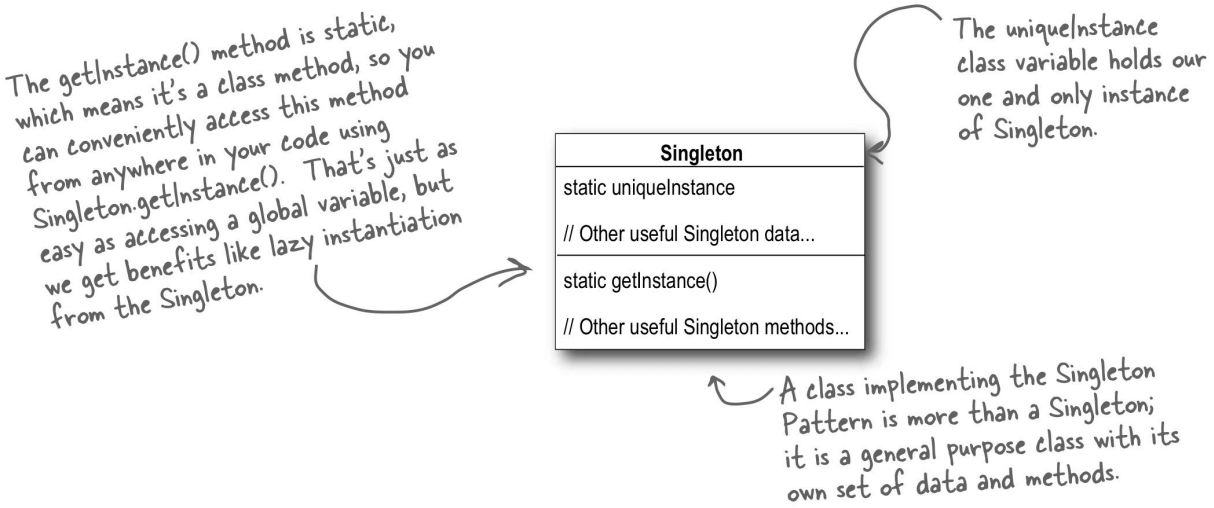
Agenda

1. Design Pattern Discussion
    a. Singleton
    b. Command
    c. Adapter
    d. Facade
    e. Template Method
    f. State
2. Project Part A Presentations

Design Pattern Discussion

Singleton

Definition: The Singleton Pattern ensures a class has only one instance and provides a global point of access to it.

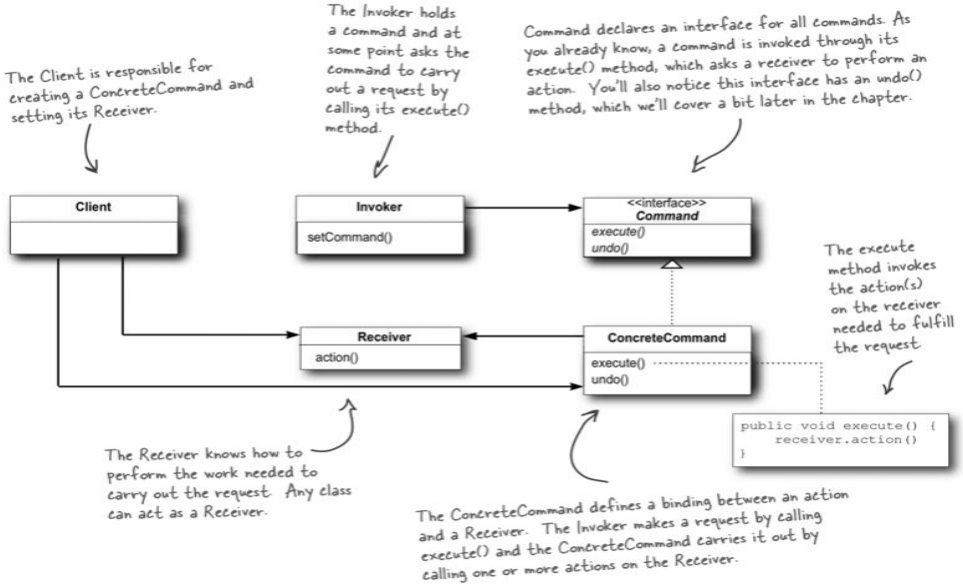


Implementation

```
public class GameBoard {
  // volatile ensures that the GameBoard instance is always retrieved from
  // main memory instead of cache; ensures thread safety
  private volatile static GameBoard board;
  private GameBoard() {}

  public static GameBoard getBoard() {
    if (board == null) {
      synchronized(GameBoard.class) {
        // Double-checked locking provides two benefits:
        // 1. board instance creation is thread-safe
        // 2. subsequent board instance retrievals don't apply a lock, thus
        //    making retrievals performant
        if (board == null) {
          board = new GameBoard();
        }
      }
    }
    return board;
  }
  }
```

Command Design Pattern

Definition: The Command Pattern encapsulates a request as an object, thereby letting you parameterize other objects with different requests, que or log requests, and support undoable operations.



Implementation

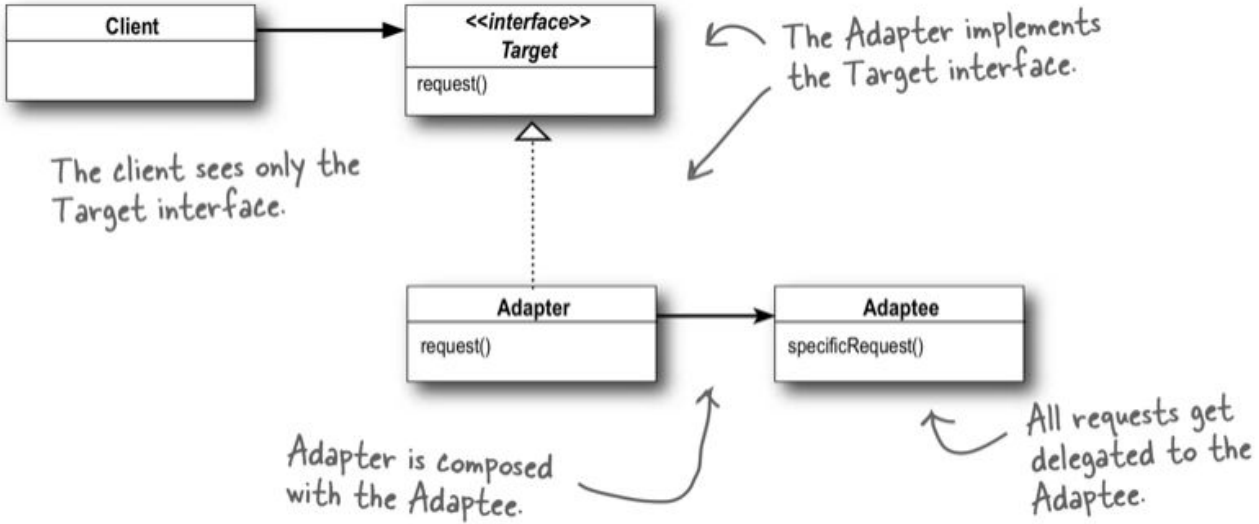
```
public interface Order { // Command
  public void makeOrder();
}

public interface Cook(){ // Receiver
  makeSoup();
  makeToast();
}

public class Waitress { // Invoker
  Order currentOrder;
  public Waitress() {}

  public void tellOrder(Order o) {
    this.currentOrder = o;
  }

  public void submitOrder() {
    currentOrder.makeOrder();
  }
}
```

```
public class SoupOrder implements Order {
  Cook cook;

  public SoupOrder(Cook c) {
    this.cook = c;
  }

  public void makeOrder() {
    cook.makeSoup();
  }
}
```

```
public class Client {
  public static void main(String[] args) {
    Waitress thisWaitress = new Waitress();
    Cook thisCook = new Cook();

    // One quirk about this example is that we nee
    // order objects that know the cook at creatio
    thisWaitress.tellOrder(new SoupOrder(thisCook)
    thisWaitress.submitOrder();
  }
}
```

Adapter Design Pattern

Definition: The Adapter Pattern converts the interface of a class into another interface the clients expect. Adapter lets classes work together that couldn't otherwise because of incompatible interfaces.



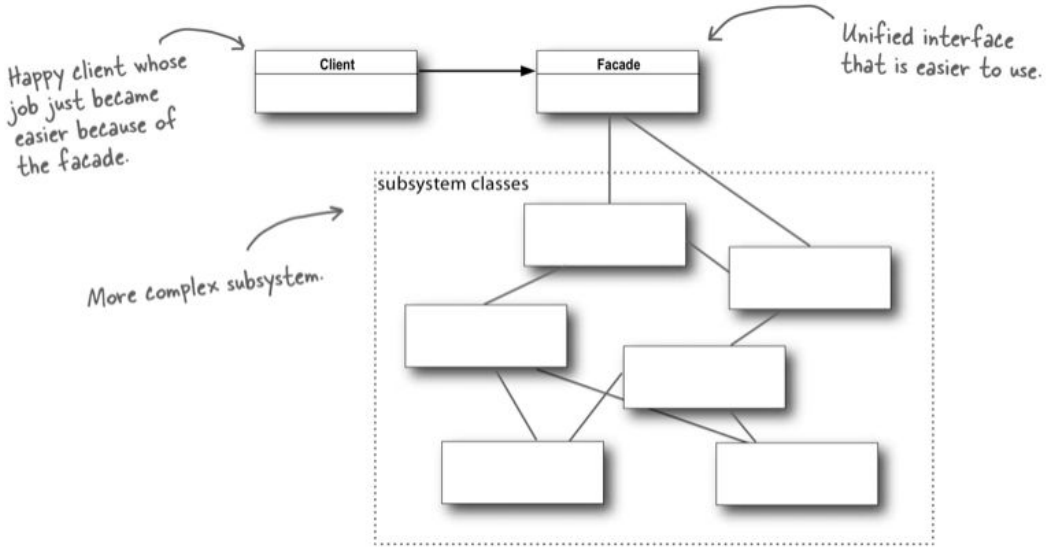
Implementation

```
public class Adaptee {
  public Adaptee() {}
  public void undesiredInterface() {
    System.out.println("I'm doing something!");
  }
}

public interface Target {
  public void desiredInterface();
}

public class Adaptor implements Target {
  Adaptee adaptee;
  public Adaptor(Adaptee a) {
    this.adaptee = a;
  }

  public void desiredInterface() {
    adaptee.undesiredInterface();
  }
}
```

```
public class Client {
  public static void main(String args[]) {
    Adaptee adaptee = new Adaptee();
    Target adaptor = new Adaptor(adaptee);
    adaptor.desiredInterface();
  }
}
```

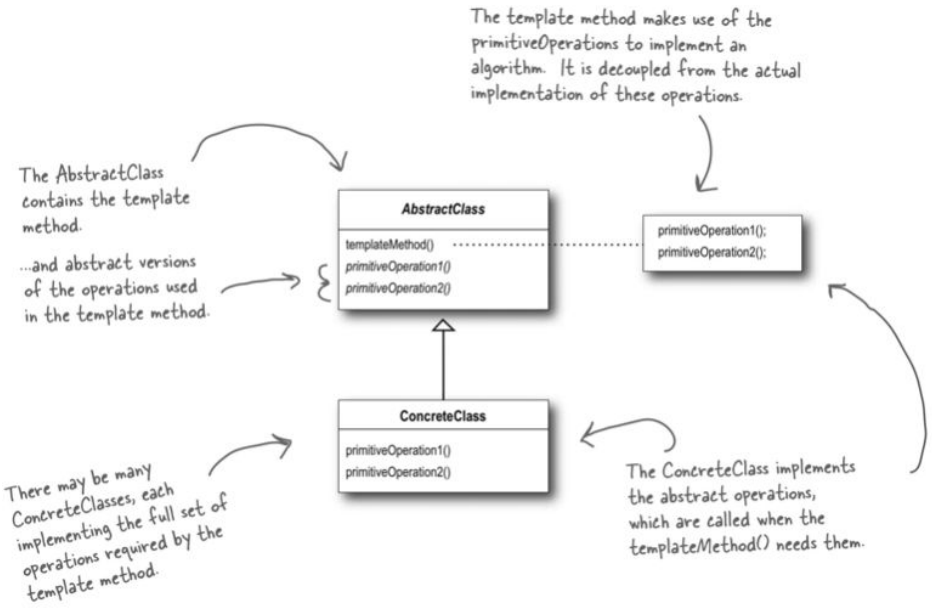
Facade Design Pattern

Definition: The Facade Pattern provides a unified interface to a set of interfaces in a subsystem. Facade defines a higher-level interface that makes the subsystem easier to use.



Template Method

Definition: The Template Method Pattern defines the skeleton of an algorithm in a method, deferring some steps to subclasses. Template Method lets subclasses redefine certain steps of an algorithm without changing the algorithm's structure.



Implementation

```
public abstract class CaffeineBeverage {
  final void prepareRecipe() {
    boilWater();
    brew();
    pourInCup();
    addCondiments();
  }

  // Left to concrete classes to implement
  abstract void brew();

  // Hook, definition left empty but can be over
  // add desired functionality
  void addCondiments() {}
}
```

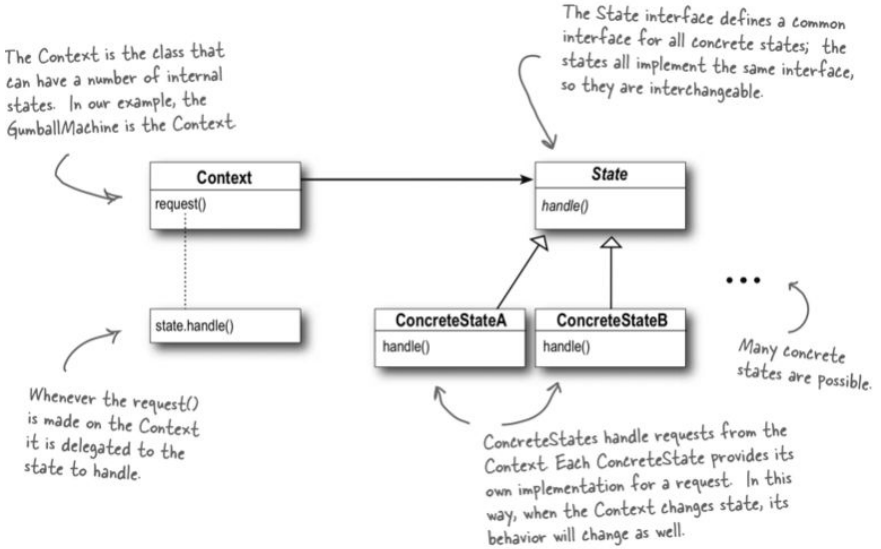
```
public class Tea extends CaffeineBeverage {
  public void brew() {
    System.out.println("Steeping tea");
  }
}
```

```
public class SweetTea extends Tea {
  // Implementing the optional hook
  public void addCondiments() {
    System.out.println("Adding honey");
  }
}
```

```
public class Coffee extends CaffeineBeverage {
  public void brew() {
    System.out.println("Dripping coffee through f
  }
  public void addCondiments() {
    System.out.println("Adding sugar and milk");
  }
}
```

State Method

Definition: The State Pattern allows an object to alter its behavior when its internal state changes. The object will appear to change its class.



Implementation

```
public class CeilingFan { // Context
  State offState = new CeilingFanOffState();
  State lowState = new CeilingFanLowState();
  State highState = new CeilingFanHighState();
  State state = offState;

  public CeilingFanState getOffstate() { return offState; }
  public CeilingFanState getLowState() { return lowState; }
  public CeilingFanState getHighState() { return highState; }

  public setState(CeilingFanState s) {
    this.state = s;
  }
}
```

```
public interface CeilingFanState { // State
  turnOff();
  setToLow();
  setToHigh();
}

public class CeilingFanOffState implements CeilingFanState {
  CeilingFan ceilingFan;
  public CeilingFanOffState(CeilingFan c) { this.ceilingFan = c; }

  turnOff() {}
  setToLow() { ceilingFan.setState(ceilingFan.getLowState()); }
  setToHigh() { ceilingFan.setState(ceilingFan.getHighState()); }
}

public class CeilingFanLowState implements CeilingFanState {
  CeilingFan ceilingFan;
  public CeilingFanLowState(CeilingFan c) { this.ceilingFan = c; }

  turnOff() { ceilingFan.setState(ceilingFan.getOffState(); )}
  setToLow() {}
  setToHigh() { ceilingFan.setState(ceilingFan.getHighState()); }
}

public class CeilingFanHighState implements CeilingFanState {
  CeilingFan ceilingFan;
  public CeilingFanHighState(CeilingFan c) { this.ceilingFan = c; }

  turnOff() { ceilingFan.setState(ceilingFan.getOffState(); )}
  setToLow() { ceilingFan.setState(ceilingFan.getLowState()); }
  setToHigh() {}
  }
```

Project Presentations

Peer Feedback form: http://bit.ly/peer-feedback-A
Team Feedback form: http://bit.ly/teams-feedback-A