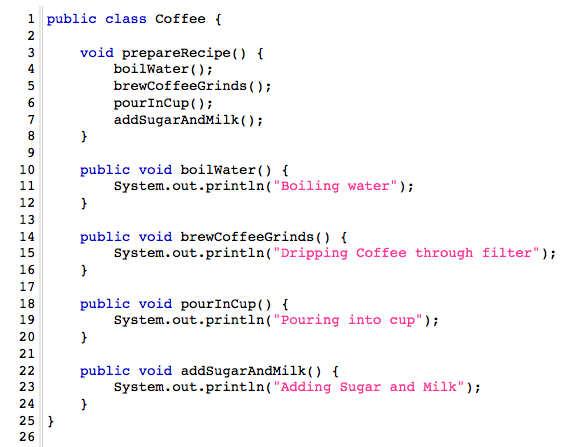
*객체지향 설계와 패턴*  
**LAB ASSIGNMENT #11**

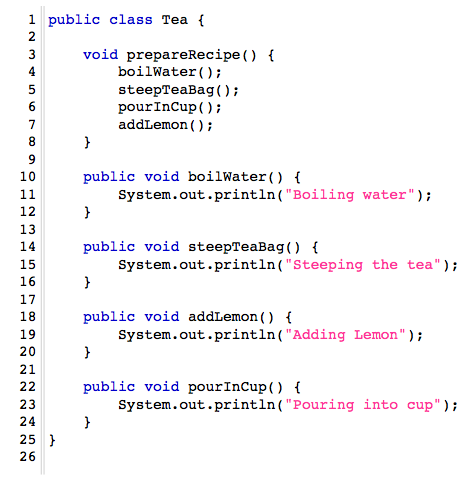
## Weight 10% 마감 5월 28일(금)

**목적: 오퍼레이션 패턴의 적용**

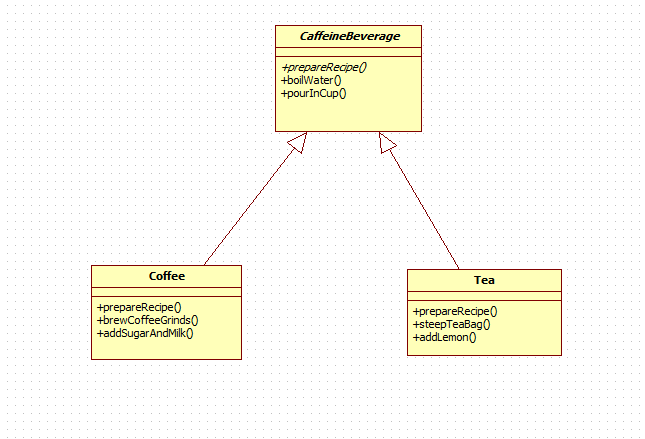
문제 #1: 템플릿 메소드 패턴

커피숍에 다음과 같은 두 가지 음료(커피, 차)에 대한 코드가 있다.





(1) coffee와 tea의 prepareRecipe()의 과정 중 boilWater() and pourInCup() 함수는 똑같다. 위 코드를 템플릿 메소드 패턴(공통 클래스를 CaffeineBeberage로 할 것)을 이용하여 설계하라.



(2) 위의 설계를 코딩하라.

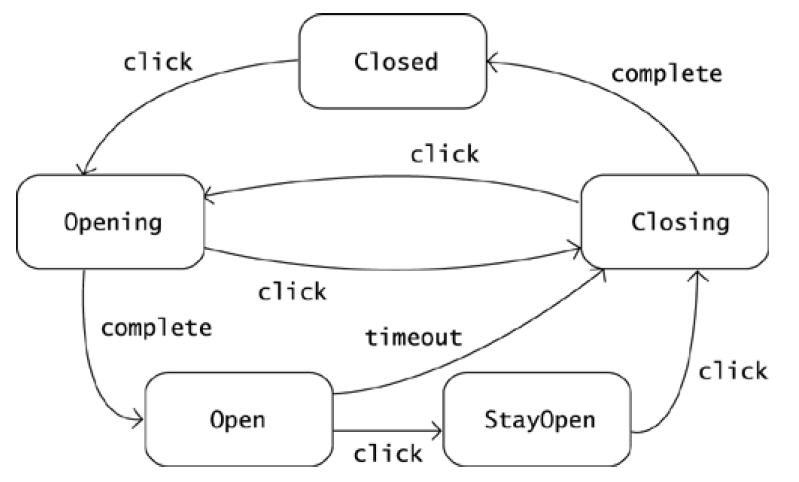
public abstract class CaffeineBeverage {  
 abstract void prepareRecipe();  
  
 public void boilWater() {  
 System.*out*.println("Boiling water");  
 }  
  
 public void pourInCup() {  
 System.*out*.println("Pouring into cup");  
 }  
}

public class Coffee extends CaffeineBeverage {  
 void prepareRecipe() {  
 boilWater();  
 brewCoffeeGrinds();  
 pourInCup();  
 addSugarAndMilk();  
 }  
  
 public void brewCoffeeGrinds() {  
 System.*out*.println("Dripping Coffee through filter");  
 }  
  
 public void addSugarAndMilk() {  
 System.*out*.println("Adding Sugar and Milk");  
 }  
}

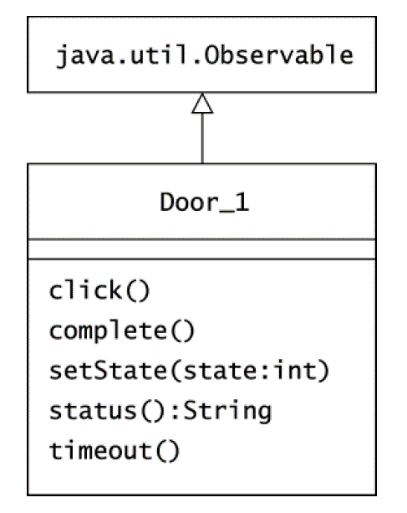
public class Tea extends CaffeineBeverage {  
 @Override  
 void prepareRecipe() {  
 boilWater();  
 steepTeaBag();  
 pourInCup();  
 addLemon();  
 }  
  
 public void steepTeaBag() {  
 System.*out*.println("Steeping the tea");  
 }  
  
 public void addLemon() {  
 System.*out*.println("Adding Lemon");  
 }  
}

문제 #2: 상태 패턴

다음은 출입구의 상태 변화를 나타낸 다이어그램이다.



(1) Door1을 다음과 같이 설계하여 코딩하였다. Door1 클래스를 위한 Complete()와 timeout() 메소드를 작성하라.



package com.oozinoz.carousel;

public class Door\_1 extends Observable

{

public static final int CLOSED = -1;

public static final int OPENING = -2;

public static final int OPEN = -3;

public static final int CLOSING = -4;

public static final int STAYOPEN = -5;

private int state = CLOSED;

//

}

public String status()

{

switch (state)

{

case OPENING :

return "Opening";

case OPEN :

return "Open";

case CLOSING :

return "Closing";

case STAYOPEN :

return "StayOpen";

default :

return "Closed";

}

}

public void click()

{

if (state == CLOSED)

{

setState(OPENING);

}

else if (state == OPENING || state == STAYOPEN)

{

setState(CLOSING);

}

else if (state == OPEN)

{

setState(STAYOPEN);

}

else if (state == CLOSING)

{

setState(OPENING);

}

}

private void setState(int state)

{

this.state = state;

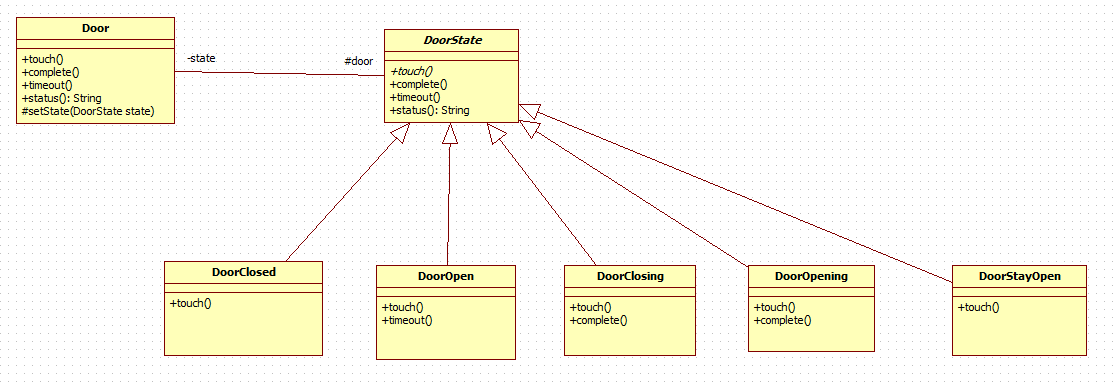
setChanged();

notifyObservers();

}

public void complete() {  
 if (state == *CLOSING*) {  
 setState(*CLOSED*);  
 } else if (state == *OPENING*) {  
 setState(*OPEN*);  
 }  
}  
  
public void timeout() {  
 if (state == *OPEN*) {  
 setState(*CLOSING*);  
 }   
}

(2) 위 프로그램을 상태 패턴을 적용하여 설계를 개선하고 코딩하라.



import java.util.Observable;  
  
public class Door extends Observable {  
 public final DoorState CLOSED = new DoorClosed(this);  
 public final DoorState CLOSING = new DoorClosing(this);  
 public final DoorState OPEN = new DoorOpen(this);  
 public final DoorState OPENING = new DoorOpening(this);  
 public final DoorState STAYOPEN = new DoorStayOpen(this);  
  
 private DoorState state = CLOSED;  
  
 public void touch() {  
 state.touch();  
 }  
  
 public void complete() {  
 state.complete();  
 }  
  
 public void timeout() {  
 state.timeout();  
 }  
  
 public String status() {  
 return state.status();  
 }  
  
 protected void setState(DoorState state) {  
 this.state = state;  
 setChanged();  
 notifyObservers();  
 }  
}

public class DoorClosed extends DoorState {  
 public DoorClosed(Door door) {  
 super(door);  
 }  
  
 public void touch() {  
 door.setState(door.OPENING);  
 }  
}

public class DoorClosing extends DoorState {  
 public DoorClosing(Door door) {  
 super(door);  
 }  
  
 @Override  
 public void touch() {  
 door.setState(door.OPENING);  
 }  
  
 @Override  
 public void complete() {  
 door.setState(door.CLOSED);  
 }  
}

public class DoorOpen extends DoorState {  
 public DoorOpen(Door door) {  
 super(door);  
 }  
  
 @Override  
 public void touch() {  
 door.setState(door.STAYOPEN);  
 }  
  
 @Override  
 public void timeout() {  
 door.setState(door.CLOSING);  
 }  
}

public class DoorOpening extends DoorState {  
 public DoorOpening(Door door) {  
 super(door);  
 }  
  
 @Override  
 public void touch() {  
 door.setState(door.CLOSING);  
 }  
  
 @Override  
 public void complete() {  
 door.setState(door.OPEN);  
 }  
}

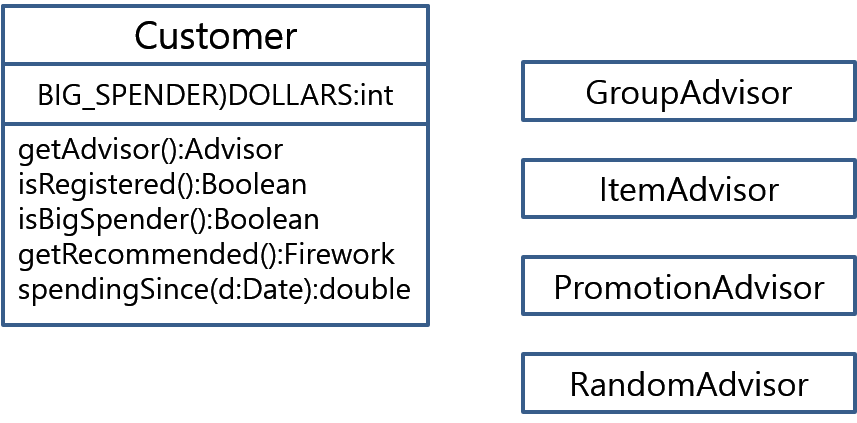
public abstract class DoorState {  
 protected Door door;  
  
 public DoorState(Door door) {  
 this.door = door;  
 }  
  
 public abstract void touch();  
  
 public void complete(){}  
  
  
 public void timeout(){}  
  
 public String status() {  
 String s = getClass().getName();  
 return s.substring(s.lastIndexOf('.') + 1);  
 }  
}

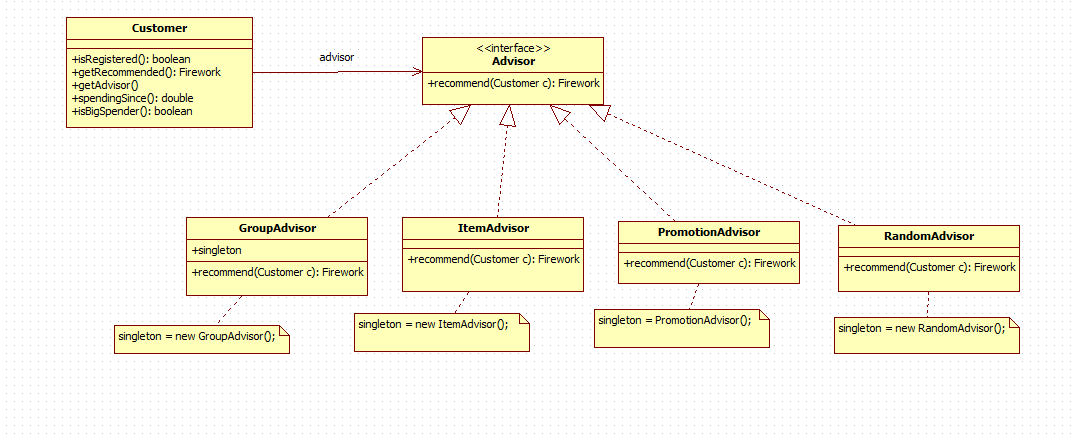
public class DoorStayOpen extends DoorState {  
 public DoorStayOpen(Door door) {  
 super(door);  
 }  
  
 @Override  
 public void touch() {  
 door.setState(door.CLOSING);  
 }  
}

문제 #3: Strategy패턴

축제의 하일라이트인 불꽃놀이를 위한 폭죽을 판매하는 회사가 있다. 고객이 폭죽판매 Oozinoz 웹 사이트를 방문하거나 콜 센터에 전화하면 구매할 불꽃 놀이를 제안하는 프로그램을 만들려 한다. Oozinoz는 두 개의 상용 기성 추천 엔진을 사용하여 고객에게 제공할 적합한 불꽃 놀이를 선택하는 데 도움을 준다. Customer 클래스는 이러한 엔진 중 하나를 선택하고 적용하여 고객에게 추천할 불꽃 놀이를 결정한다고 하자.

1. 공통되는 인터페이스를 구현함으로써 나타낼 수 있는 전략(폭죽을 추천)들을 가지고 폭죽을 추천해주는 소프트웨어를 제공하는 결과를 보여주도록 설계를 완성하라.  
   각각의 Advisor 구현은 recommend() 오퍼레이션을 제공하는 singleton을 제공해야 한다.





1. 아래 코드는 Customer 클래스의 getAdvisor()의 구현이다.   
   추가로 Customer 클래스의 getRecommended()와 ItemAdvisor 클래스를 간단하게 구현하라.

**private** Advisor getAdvisor(){

**if**(advisor==**null**)

{

**if**(PromotionAdvisor.singleton.hasItem())

{

advisor = PromotionAdvisor.singleton;

}**else** **if**(isRegistered()){

advisor = GroupAdvisor.singleton;

}**else** **if**(isBigSpender()){

advisor = ItemAdvisor.singleton;

}**else**{

advisor = RandomAdvisor.singleton;

}

}

**return** advisor;

}

private boolean isBigSpender()

{

Calendar cal = Calendar.getInstance();

cal.add(Calendar.YEAR, -1);

return spendingSince(cal.getTime()) > BIG\_SPENDER\_DOLLARS;

}

public Firework getRecommended() {  
 Firework recommend = getAdvisor().recommend(this);  
 return recommend;  
}

public class ItemAdvisor implements Advisor {  
 public static ItemAdvisor *singleton* = new ItemAdvisor();  
 private ItemAdvisor(){}  
 public Firework recommend(Customer customer) {  
 return (Firework) LikeMyStuff.*suggest*(customer);  
 }  
}

문제 4: Strategy Pattern

다음은 학생의 리스트를 받아 여러 가지 방법으로 순서 정렬하기 위한 프로그램의 일부이다.

public class SortApp {

public static void main(String[] args) {

// Two contexts following different strategies

SortedList studentRecords = new SortedList();

studentRecords.add("Samual");

studentRecords.add("Jimmy");

studentRecords.add("Sandra");

studentRecords.setSortStrategy(new QuickSort());

studentRecords.sort();

studentRecords.setSortStrategy(new ShellSort());

studentRecords.sort();

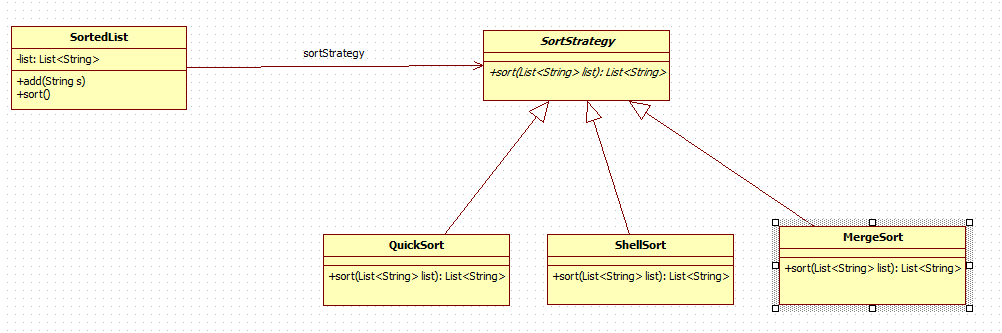
studentRecords.setSortStrategy(new MergeSort());

studentRecords.sort();

}

}

(1) 위 프로그램이 실행되기 위하여 Strategy 패턴을 적용하여 QuikSort, ShellSort, MergeSort 방법을 필요에 따라 대체할 수 있도록 SortedList, SortStrategy 클래스를 설계하라.



(2) 위 설계를 코딩하고 실행하라.

import java.util.List;  
  
public abstract class SortStrategy {  
  
 public abstract List<String> sort(List<String> list);  
  
}

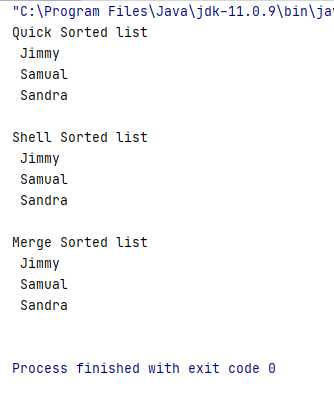
import java.util.ArrayList;  
import java.util.List;  
  
public class SortedList {  
 private List<String> list = new ArrayList<>();  
 private SortStrategy sortStrategy;  
  
*// public SortedList(SortStrategy sortStrategy) {  
// this.sortStrategy = sortStrategy;  
// }* public void setSortStrategy(SortStrategy sortStrategy) {  
 this.sortStrategy = sortStrategy;  
 }  
  
 public void add(String name) {  
 list.add(name);  
 }  
  
 public void sort() {  
 list = sortStrategy.sort(list);  
  
 for (String name : list) {  
 System.*out*.println(" " + name);  
 }  
 System.*out*.println();  
 }  
}

import java.util.Arrays;  
import java.util.List;  
  
public class ShellSort extends SortStrategy {  
  
 public static void shellSort(String[] arr) {  
 int n = arr.length;  
  
 for (int gap = n / 2; gap > 0; gap /= 2) {  
 for (int i = gap; i < n; i++) {  
 String key = arr[i];  
 int j = i;  
 while (j >= gap && arr[j - gap].compareTo(key) > 0) {  
 arr[j] = arr[j - gap];  
 j -= gap;  
 }  
 arr[j] = key;  
 }  
 }  
  
 }  
  
  
 @Override  
 public List<String> sort(List<String> list) {  
 String[] arr = list.toArray(new String[0]);  
  
 *shellSort*(arr);  
 list = Arrays.*asList*(arr);  
  
 System.*out*.println("Shell Sorted list");  
 return list;  
 }  
}

public class QuickSort extends SortStrategy {  
  
 public static int partition(String[] arr, int start, int end) {  
 String pivotValue = arr[start];  
 while (start < end) {  
 String value1;  
 String value2;  
 while ((value1 = arr[start]).compareTo( pivotValue ) < 0) {  
 start = start + 1;  
 }  
 while ((value2 = arr[end]).compareTo( pivotValue ) > 0) {  
 end = end - 1;  
 }  
 arr[start] = value2;  
 arr[end] = value1;  
 }  
 return start;  
 }  
 public static void quickSort(String[] arr, int start, int end) {  
 if (start >= end) {  
 *// we are done* return;  
 }  
 int pivotIndex = *partition*(arr, start, end);  
 *quickSort*(arr, start, pivotIndex);  
 *quickSort*(arr, pivotIndex+1, end);  
 }  
  
 @Override  
 public List<String> sort(List<String> list) {  
 String[] arr = list.toArray(new String[0]);  
  
  
 *quickSort*(arr,0,arr.length-1);  
 list = Arrays.*asList*(arr);  
  
 System.*out*.println("Quick Sorted list");  
 return list;  
 }  
}

public class MergeSort extends SortStrategy {  
 public static void merge(String[] arr, int start, int mid, int end) {  
 int n = end - start + 1;  
 String[] b = new String[n];  
 int i1 = start;  
 int i2 = mid + 1;  
 int j = 0;  
  
  
 while (i1 <= mid && i2 <= end) {  
 if (arr[i1].compareTo(arr[i2]) < 0) {  
 b[j] = arr[i1];  
 i1++;  
 } else {  
 b[j] = arr[i2];  
 i2++;  
 }  
 j++;  
 }  
  
  
 while (i1 <= mid) {  
 b[j] = arr[i1];  
 i1++;  
 j++;  
 }  
  
  
 while (i2 <= end) {  
 b[j] = arr[i2];  
 i2++;  
 j++;  
 }  
  
  
 for (j = 0; j < n; j++) {  
 arr[start + j] = b[j];  
 }  
 }  
  
  
  
  
 public static void mergeSort(String[] arr, int start, int end) {  
 if (start == end) {  
 return;  
 }  
 int mid = (start + end) / 2;  
  
 *mergeSort*(arr, start, mid);  
 *mergeSort*(arr, mid + 1, end);  
 *merge*(arr, start, mid, end);  
 }  
  
  
 @Override  
 public List<String> sort(List<String> list) {  
 String[] arr = list.toArray(new String[0]);  
  
 *mergeSort*(arr,0,arr.length-1);  
 list = Arrays.*asList*(arr);  
  
 System.*out*.println("Merge Sorted list");  
 return list;  
 }  
}

public class SortApp {  
 public static void main(String[] args) {  
 SortedList studentRecords = new SortedList();  
 studentRecords.add("Samual");  
 studentRecords.add("Jimmy");  
 studentRecords.add("Sandra");  
*// studentRecords.setSortStrategy(new QuickSort());  
// studentRecords.sort();* studentRecords.setSortStrategy(new ShellSort());  
 studentRecords.sort();  
*// studentRecords.setSortStrategy(new MergeSort());  
// studentRecords.sort();* }  
}



**제출물**

제출하여야 할 것:

1. UML로 표현한 설계. 어디가 변경되었는지 표시하고 패턴이 적용된 이유와 MakeAGraph설계에 그 패턴을 적용하였을 때 어떤 효과가 있었는지 기술하라.

2. 구현한 결과:

\* 원시코드

\* 실행 한 후의 결과 화면

**평가**

과제 #8은 다음과 같은 기준으로 평가할 것임:

1. 패턴의 효과성

2. 설계의 타당성

3. 설계, 구현 결과물에서 볼 수 있는 설계에 대한 이해 수준