CS 130 Software Engineering – University of California, Los Angeles –(Professor Kim)
Class Activity: Design Pattern Skeletons

* Strategy *

```
public class Thing {
        Strategy strategy;
        public void action(){
                strategy.do();
        }
}

public interface Strategy {
        public void do();
}
public class WaysofDoingThings implements Strategy {
        public void do {
        // implement a specific way of doing things
        }
}

public class AnotherWayofDoingThings implements Strategy {
        public void do { …}
}
public class ConcreteThing extends Thing {
        public ConcreteThing () {
                strategy = new WaysOfDoingThings();
        }
}
```

Observer Design Pattern *

```
public interface Observer (
        public void update (State st1, State st2, ...);
}

public interface Subject {
        public void attach (Observer o);
        public void detach (Observer o);
        public void notify ();
}

public class SpecificSubject implements Subject {
        private ArrayList observers;
        private State state1;
        private State state2;
```

```
	public SpecificSubject () {
		observers = new ArrayList();
	}
	public void attach (Observer o) {
		observers.add(o);
	}
	public void detach (Observer o) {
		int i = observers.indexOf(o);
		if (i>=0) observers.remove (i);
	}
	public void notify() {
		for (int i = 0; i < observers.size(); i++) {
			Observer observer = (Observer)observers.get(i);
			observer.update(state1, state2);
		}

	}
	public void setStates(State st1, State st2, State st3) {
		this.state1 = st1; this.state2= st2;
		notify();
	}
}
public class OneObserver implements Observer {
	private Subject subject;
	public OneObserver(Subject s) {
		this.subject = s;
		subject.attach(this);
	}
	public void update (State s1, State s2) { // do specific reaction to the subject's state change
	}

}
public class AnotherObserver implements Observer {
	private Subject subject;
	public AnotherObserver(Subject s) {
		this.subject = s;
		subject.attach(this);
	}
	public void update (State s1, State s2) { // do specific reaction to the subject's state change
	}

}

public class Client {
	public static void main (String[] args) {
		SpecificSubject subject = new SpecificSubject();
		Observer o1 = new OneObserver(subject);
		Observer o2 = new AnotherObserver(subject);
		// create states
		subject.setStates(...);
```

```
        }
}
```

* Factory Method Pattern *

```
public abstract class Creator {
        abstract Product createMethod (string item);
        // This is a factory method.

        public Product otherMethodsThatUseCreation(String type) {
                Product product = createMethod(type);
                product.doCommonOperation();  // the assumption is that operations/ actions after the
object construction is not going to change very much.
                return product;
        }
}

public class ConcreteCreator extends Creater {
        // ConcreteCreator must implement how to create concrete products.
        Product createMethod (string item); {
                Product product = null;
                if (item) {
                        product = new ConcreteProduct1();
                }else if (...) {
                        product = new ConcreteProduct2();
                }
        }
}

public abstract class Product {

}

public class ConcreteProduct extends Product {}
```

* Abstract Factory Pattern * --- Think of AbstractFactory as AbstractIngredientCollection

```
public abstract class Creator {
        IngredientCollection collection = null;

        abstract Product createMethod (string item);
        // This is a factory method.
```

```
	public Product otherMethodsThatUseCreation(String type) {
		Product product = createMethod(type);
		product.doCommonOperation();
		return product;
	}
}

public class ConcreteCreator extends Creator {

	// ConcreteCreator must implement how to create concrete products.
	Product createMethod (string item); {
		AbstractIngredientCollection collection = new ParticularIngredientCollection();
		// The above statements constrains that "ConcreteCreator" is associated with
"ParticularIngredientCollection".
		Product product = null;
			product = new ConcreteProduct(collection);
		}else if (...) {
		}
	}
}

public abstract class Product {
	IngredientA iA;
	IngredientB iB;

	public void doCommonOperation() {...}
}
public class ConcreteProduct extends Product {
	IngredientCollection collection;
	public ConcreteProduct (AbstractIngredientCollection collection ) {
		this.collection = collection;
		this.iA = collection.createIngredientA();
		this.iB = collection.createIngredientB();

	}
}

public interface AbstractIngredientCollection{
	IngredientA createIngredientA();
	IngredientB createIngredientB();
}
public class ParticularCollection implements AbstractIngredientCollection {
	IngredientA createIngredientA () {
		return new ParticularIngredientA();
	}
	IngredientB createIngredientB () {
		return new ParticularIngredientB();
	}
}
```

```
}
```

*Singleton *

```
public class Singleton {
        private static Singleton instance;
        private Singleton () { }
        public static getInstance () {
                if (instance ==null ) {
                        instance = new Singleton();
                }
                return instance:
        }
}

public class Singleton {
 private static Singleton uniqueInstance;
 // the single object to be constructed, but it is constructed the first time it is needed.
 private Singleton() {}
 // constructor is private so that other classes cannot use it directly
 public static Singleton getInstance() {
 // this is a getter method.
  if (uniqueInstance == null) {
   uniqueInstance = new Singleton();
        // it is constructed the first time it is used.
  }
  return uniqueInstance;
 }
 // ...
}
```

Adaptor Pattern

```
public class Client {
        public static void main (String args[]) {
                Adaptee adaptee = new Adaptee();
                Target adaptor = new Adaptor (adaptee);
                adaptor.idealInterface();
                /// this is a long code and you really don't want to change this.
        }
}
public interface Target {
        public void idealInterface();
}
```

```
public class Adaptor implements Target {

	Adaptee adaptee = null;
	public Adaptor (Adaptee a) {
		this.adaptee = a;
	}
	public void idealInterface () {
		// do some extra work and wrap the notIdealInterface
// code here.
		adaptee.notIdealInterface();
// code here
	}

}
```

* Command Pattern *

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

public class SpecificCommand implements Command{
	Receiver receiverObject;
	public SpecificCommand (Receiver r) {
		this.receiverObject = r;
	}
	public void execute () {
		receiverObject.action();
	}
}

public interface Receiver () {
	action();
}
public class Invoker {
	Command slot;
	public Invoker () { }
	public void setCommand (Command c) {
		this.slot = c;
	}
	public void activate() {
		slot.execute();
	}
}

public class Client {
	public static void main (String [] args) {
```

```
                Invoker invoker = new Invoker();
                Receiver receiver = new Reciever();
                Command cmd = new SpecificCommand(receiver);
                invoker.setCommand(cmd);
                invoker.activate();
        }

}
```

* State *

```
public interface State {
        transition1();
        transition2();
}
public class State1 implements State {
        Context machine;
        public State1 (Context m) {
                this.machine = m;
        }
        transition1() {
                // encode a specific effect of doing transition 1 on State 1.
                machine.setState(machine.getState2());
        }
        transition2() {
                // encode a specific effect

        }
}
public class State2 implements State {
        // similar to the class above.
        // implement the effect of transitions on State 2.
        transition1() {...}
        transition2() {...
                machine.setState(machine.getState1());
        }
}
public Context () {
        State state1 = new State1();
        State state2 = new State2();
        State state3 = new State3();
        // Getter methods for each state
}
```

Template Method Skeleton

```
public abstract class Thing { }
        final void templatemethod()
                commonstep1();
                varyingstep2();
                commonstep3();
        }
        final void templatemethodslightlydifferent (slight different input arguments)
                commonstep1();
                varyingstep2();
                commonstep3();
                commonstep4()
        }

        abstract void varyingstep2();
        void commonstep1() {
          // do the identical step 1
        }
        void commonstep3() {
          // do the identical step 3
        }
}
public class OneThing extends Thing {
        void varyingstep2() {
         // I am doing step 2 in my way
        }
void varyingstep2() {
        "HelloWorld"
        }

}
public class ThingA extends OneThing {

}

public class ThingB extends OneThing {


}


public class AnotherThing extends Thing {
        void varyingstep2() {
        // I am doing step 2 in another way
        }
}
```