CMSC 132: OBJECT-ORIENTED PROGRAMMING II

Design Patterns II

Department of Computer Science
University of Maryland, College Park

Final Exam

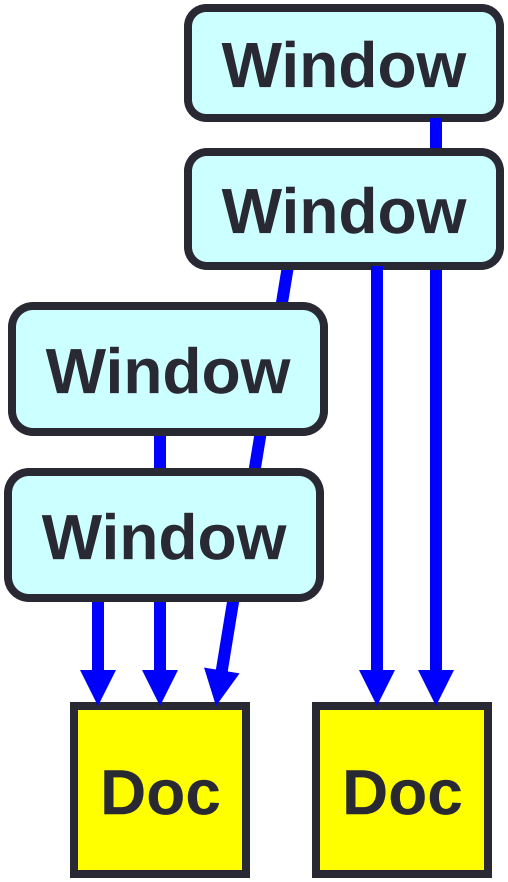
- **Date:** Friday, Aug 8
- **Time:** **10:30am to 12:30pm (NOTICE IS 30 MINUTES BEFORE USUAL CLASS/LAB TIME)**
- Additional information on the class web page

Observer Pattern

- **Definition**
  - Updates all dependents of object automatically once object changes state
- **Where to use & benefits**
  - One change affects one or many objects
  - Many object behavior depends on one object state
  - Need broadcast communication
  - Maintain consistency between objects
  - Observers do not need to constantly check for changes

Observer Pattern

- **Example**
  - Multiple windows (views) for single document
  - Original
    - Each window checks document
    - Window updates image if document changes
    - Think of window as asking “Are we there yet?”
  - Using pattern
    - Each window registers as observer for document
    - Document notifies all of its observers when it changes



Observer Example

```
public interface Observer {
     // called when observed object o changes
    public void update(Observable o, Object a)
}

public class Observable {
    protected  void setChanged()
    protected void clearChanged()
    boolean hasChanged()

    void addObserver(Observer o)
    void notifyObservers()
    void notifyObservers(Object a)
}
```

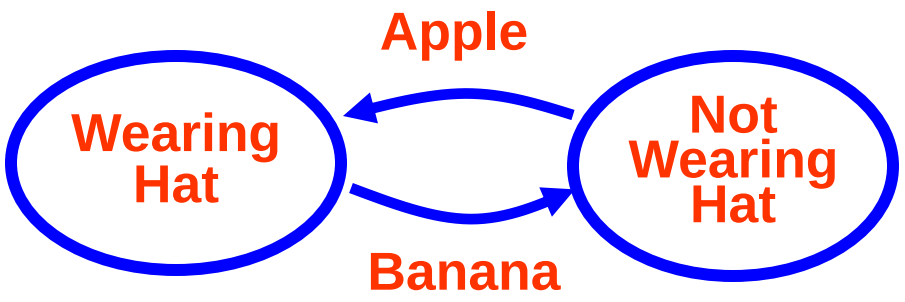
```
public class MyWindow implements Observer {
    public openDoc(Observable doc) {
        doc.addObservers(this); // add window to list
    }
    public void update(Observable doc, Object arg)  {
        redraw(doc);  // display updated document
    }
}

public class MyDoc extends Observable {
    public void edit() {
        …                                  // edit document
        setChanged();                      // mark change
        notifyObservers(arg);   // invokes update()
    }
}
```

State Pattern

- **Definition**
  - Represent change in an object's behavior using its member classes
- **Where to use & benefits**
  - Control states without many if-else statements
  - Represent states using classes
  - Every state has to act in a similar manner
  - Simplify and clarify the program
- **Example**
  - States representing finite state machine (FSM)
  - Original
    - Each method chooses action depending on state
    - Behavior may be confusing, state is implicit
  - Using pattern
    - State interface defines list of actions for state
    - Define inner classes implementing State interface
    - Finite state machine instantiates each state and tracks its current state
    - Current state used to choose action
- **Example:** StateCode

State Example – Original Code

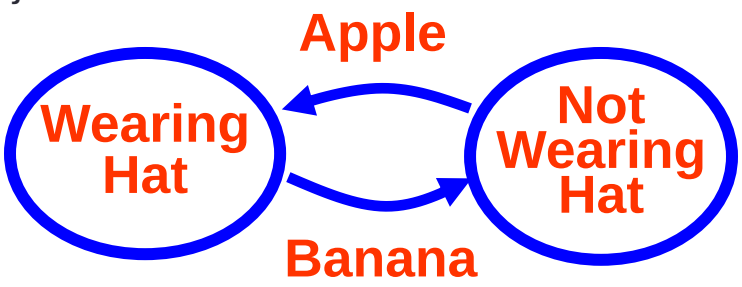
```
public class FickleFruitVendor {
    boolean wearingHat;
    boolean isHatOn() { return wearingHat; }
    String requestFruit() {
        if (wearingHat) {
            wearingHat = false;
            return "Banana";
        } else {
            wearingHat = true;
            return "Apple";
        }
    }
}
```



State Example

```
public interface State {
    boolean isHatOn();
    String requestFruit();
}

public class FickleFruitVendor {
    State wearingHat = new WearingHat();
    State notWearingHat = new NotWearingHat();

    // explicitly track current state of Vendor
    State currentState = wearingHat;

    // behavior of Vendor depends on current
    state
    public boolean isHatOn() {
        return currentState.isHatOn();
    }
    public String requestFruit() {
        return currentState.requestFruit();
    }
}
```

```
public class WearingHat implements State {
    boolean isHatOn() { return true; }
    String requestFruit() {
        // change state
        currentState = notWearingHat;
        return "Banana";
    }
}

public class NotWearingHat implements State {
    boolean isHatOn() { return false; }
    String requestFruit() {
        // change state
        currentState = wearingHat;
        return "Apple";
    }
}
}
```



Visitor Pattern

- **Definition**
  - Define operations on elements of data structures without changing their classes
- **Where to use & benefits**
  - Add operations to classes with different interfaces
  - Can modify operations on data structure easily
  - Encapsulate operations on elements of data structure
  - Decouples classes for data structure and algorithms
- **Example**
  - Print elements in collection of objects
  - Original
    - Iterator chooses action based on type of object
    - Many if-else statements
  - Using pattern
    - Visitor interface defines actions during visit
    - Visitable interface allow objects to accept visit
    - Action automatically selected by polymorphic functions through double dispatch
- **Example:** VisitorCode

Visitor Example – Original Code

```
public void messyPrintCollection(Collection c) {
     for (Object o : c) {
          if (o instanceof String)
               System.out.println(“{"+o.toString()+“}"); // add { }
          else if (o instanceof Float)
               System.out.println(o.toString()+"f");       // add f
          else
               System.out.println(o.toString());
     }
}
```

Visitor Example

```
public interface Visitor {
    public void visit(VisitableString s);
    public void visit(VisitableFloat f);
}
public interface Visitable{
    public void accept(Visitor v);
}

public class VisitableString implements Visitable {
    private String value;
    public VisitableString(String s) { value = s; }
    public String toString( ) { return value.toString( ); }
    public void accept(Visitor v) { v.visit(this); }
}

public class VisitableFloat implements Visitable {
    private Float value;
    public VisitableFloat(Float f) { value = f; }
    public String toString( ) { return value.toString( ); }
    public void accept(Visitor v) { v.visit(this); }
}
```

```
public class PrintVisitor implements Visitor {
    public void visitCollection(Collection c) {
        for (Object o : c) {
            if (o instanceof Visitable)
                ((Visitable) o).accept(this);
            else
                System.out.println(o.toString());
        }
    }
    public void visit(VisitableString s) {
        System.out.println("{"+s.toString( )+"}");
    }
    public void visit (VisitableFloat f) {
        System.out.println(f.toString( )+"f");
    }
}
```