More Design Patterns

- Marker interface
  - Label semantic attributes of a class

- Observer
  - A way of notifying change to a number of classes

- State
  - Alter an object's behavior when its state changes

- Visitor
  - Defines a new operation to a class without changing class
Marker Interface Pattern

Definition

- Label semantic attributes of a class

Where to use & benefits

- Need to indicate attribute(s) of a class
- Allows identification of attributes of objects without assuming they are instances of any particular class
Marker Interface Pattern

Example

- Classes with desirable property GoodProperty
- Original
  - Store flag for GoodProperty in each class
- Using pattern
  - Label class using GoodProperty interface

Examples from Java

- Cloneable
- Serializable
public interface GoodProperty { }  // no methods

class A implements GoodProperty { … }  
class B { … }

class A goodObj = new A();
class B badObj = new B();

if (goodObj instanceof GoodProperty) …  // True  
if (badObj instanceof GoodProperty) …  // False
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

Using pattern
- Each window registers as observer for document
- Document notifies all of its observers when it changes
public interface Observer {
    public void update(Observable o, Object a)
        // called when observed object o changes
}

public class Observable {
    protected void setChanged()       // changed
    protected void clearChanged()     // no change
    boolean hasChanged()              // return changed?
    void addObserver(Observer o)       // track observers
    void notifyObservers()            // notify if changed,
    void notifyObservers(Object a)    // then clear change
}
Observer Example

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
State Pattern

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

State Example – Original Code

Wearing Hat

Not Wearing Hat

Apple

Banana
State Example

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

public class WearingHat implements State;
public class NotWearingHat implements State;
```
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();
    }
}
State Example

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

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

} // end class
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
- Crossing class hierarchies may break encapsulation
Visitor Pattern

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
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);
}
Visitor Example

```java
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); }
}
```

Double dispatch
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("{")
        System.out.println(""+s.toString()+"}"");
    }

    public void visit(VisitableFloat f) {
        System.out.println(f.toString()+"f");
    }
}
UML Class Diagram of Abstract Visitor
Callback

Definition

Executable code passed as argument to other code

Approach

1. Higher-level code passes function as argument to lower-level code
   - In Java, pass object implementing interface
   - In C/C++, pass pointer to function

2. Lower-level code invokes callback function to perform desired task
Callback (cont.)

Motivation

- Keeps code separate
  - Clean division between higher & lower-level code

- Promotes code reuse
  - Lower-level code supports different callbacks

- Supports event-driven programming
  - Lower-level code registers function as handler

Examples

- Observer pattern  → Observer.update( )
- Visitor pattern  → Visitor.visit( )
Design Patterns – Summary

- Can be useful for designing quality software
- Successful use requires familiarity & experience
- Treat as examples of well-written code
  - Can learn how to program …
  - …by studying how good programmers write code