Design Patterns

CSC207 – Software Design

• Design pattern:
  – A general description of the solution to a well-established problem using an arrangement of classes and objects.

• Patterns describe the shape of code rather than the details.
  – There are lots of them in CSC 301 and 302.
Loop patterns from first year

• Loop pattern:
  – A general description of an algorithm for processing items in a collection.

• All of you (hopefully) have some loop patterns in your heads.
• You don’t really need to think about these any more; you just use them, and you should be able to discuss them with your fellow students.

• Some first-year patterns:
  – Process List
  – Counted Loop
  – Accumulator
  – Sentinel

Process list pattern

• **Purpose**: to process every item in a collection where you don’t care about order or context; you don’t need to remember previous items.

• **Outline**:
  ```java
  for (Object o : list) {
    // process o
  }
  ```

• **Example**: // Print every item in a list.
  ```java
  for (Object o : list) {
    System.out.println(o);
  }
  ```

• **Other example**: darken every pixel in a picture
Counted loop pattern

- **Purpose**: to process a range of indices in a collection.
- **Outline**:
  
  ```java
  for (int i = 0; i != max index; i++) {
    // process item at index i
  }
  ```

- **Example**:
  
  ```java
  // Bubble through a list: swap items that are out of order.
  for (int i = 0; i != list.size() - 1; i++) {
    if (list.get(i) < list.get(i + 1)) {
      swap(list, i, i + 1); // assuming helper function swap
    }
  }
  ```

- **Other example**: print indices of even-length string

Accumulator pattern

- **Purpose**: to accumulate information about items in a collection.
- **Outline**:
  
  ```java
  result = some appropriate base case, such as an empty list or 0
  for (Object o : list) {
    // Modify result with information from o.
  }
  ```

- **Example**:
  
  ```java
  // Find the longest String in a list.
  result = "";
  for (String s : list) {
    if (s.length() > result.length()) {
      result = s;
    }
  }
  ```

- **Other examples**: sum, min, accumulate a list of items meeting a particular criterion.
Sentinel pattern

- **Purpose**: to remove a condition in a loop guard.
- **Outline**:
  - add an item "sentinel" with a particular value at the end of a list
    int i = 0;
    while (list.get(i) != sentinel) {
      i++;
    }
    remove the sentinel from the end
- **Example**:
  - // find the index of o in list, if it's there.
    list.add(o); // make sure o is in list.
    int i = 0;
    while (!o.equals(list.get(i))) {
      i++;
    }
    list.remove(list.size() - 1); // remove the sentinel
    // if i == list.size(), o was not in the list.

Sentinel pattern, continued

- Here is the code that Sentinel replaces; note that
  \( i != \text{list.size()} \) is evaluated every time through
  the loop, even though it is false only once.

  ```java
  // find the index of o in list, if it's there.
  int i = 0;
  while (i != list.size() && !o.equals(list.get(i))) {
    i++;
  }
  // if i == list.size(), o was not in the list.
  ```
Design Pattern Categories

• Creational
  – **Purpose**: control the way objects are created
  – **Examples**: Singleton, Abstract Factory, Prototype

• Behavioural
  – **Purpose**: process a collection of items
  – **Examples**: Iterator, Visitor

• Structural
  – **Purpose**: store data in a particular way
  – **Examples**: Composite, Adapter

<table>
<thead>
<tr>
<th>Creational</th>
<th>Structural</th>
<th>Behavioural</th>
<th>Architecture</th>
</tr>
</thead>
<tbody>
<tr>
<td>Factory method</td>
<td><strong>Adapter</strong></td>
<td>Null Object</td>
<td>Layers</td>
</tr>
<tr>
<td>Abstract Factory</td>
<td>Bridge</td>
<td>Null Object</td>
<td>Presentation-abstraction-control</td>
</tr>
<tr>
<td>Builder</td>
<td>Composite</td>
<td>Command</td>
<td>Three-tier</td>
</tr>
<tr>
<td>Lazy instantiation</td>
<td>Decorator</td>
<td>Interpreter</td>
<td>Pipeline</td>
</tr>
<tr>
<td>Object pool</td>
<td>Façade</td>
<td>Iterator</td>
<td>Implicit invocation</td>
</tr>
<tr>
<td>Prototype</td>
<td>Flyweight</td>
<td>Mediator</td>
<td>Blackboard system</td>
</tr>
<tr>
<td><strong>Singleton</strong></td>
<td>Proxy</td>
<td>Memento</td>
<td>Peer-to-peer</td>
</tr>
<tr>
<td>Multiton</td>
<td></td>
<td><strong>Observer</strong></td>
<td>Model-View-Controller</td>
</tr>
<tr>
<td>Resource acquisition is initialization</td>
<td></td>
<td>State</td>
<td>Service-oriented architecture</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Chain of responsibility</td>
<td>Naked objects</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Strategy</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Specification</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Template method</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Visitor</td>
<td></td>
</tr>
</tbody>
</table>
Singleton Pattern

- **Purpose**: to ensure there is exactly one instance of a class.
- **Outline**:
  ```java
  // This was generated by NetBeans.
  public class NewSingleton {
    private NewSingleton() {} 

    public static NewSingleton getInstance() {
      return NewSingletonHolder.INSTANCE;
    }

    private static class NewSingletonHolder {
      private static final NewSingleton INSTANCE = new NewSingleton();
    }
  }
  ```

**Uses**: password verifier for a website, logger object for tracking events.

There are other options for an implementation. What are they? Why might there be an inner class here?

**UML**: Singleton Pattern

- `-` means private
- `+` means public
- Only one is ever created.

**Examples**:
- interface to a database
- logging system
Iterator Pattern

- **Purpose**: to separate the list contents from the object that iterates over them so that multiple iterators can be used.
- **Outline**:

  ```java
  interface java.util.Iterable: the collection of information.
      
      One method: Iterator<T> iterator()
  
  interface java.util.Iterator: an object that knows the internals of that collection and can give them back one by one.
      
      Methods: Object next(), boolean hasNext(), void remove()
  
  Uses:
      
      Iterator itr = aList.iterator();
      while(itr.hasNext())
          // process itr.next()
      
      This also allows you to plug into the Java foreach loop:
      
      for (Object o : list) ...
  ```

Implementing the Iterator Pattern

```java
public class MyCollection<T> implements Iterable<T> {
    private int size;
    private T[] list = ...;
    public Iterator<T> iterator() {
        return new MyIterator<T>();
    }
}

private class MyIterator<T> implements Iterator<T> {
    int current = 0;
    public boolean hasNext() { return current < list.size(); }
    public T next() {
        T res = list[current];
        current++;
        return res;
    }
    // optional operation; what are the difficulties?
    public void remove() {}
}
```

Use:
- Given m, a variable of `type MyCollection<String>`,
- ```java
   Iterator itr = m.iterator();
   while(itr.hasNext()) {
       String s = itr.next();
   }
   ```
- For `String s : m` {
- ```java
   // do something with s
   ```
- ```java
   // do something with s
   ```
UML: Iterator Pattern

Aggregate
+createIterator()

Client

Iterator
+first()
+next()
+isDone()
+currentItem()

ConcreteAggregate
+createIterator()

ConcretelIterator

{ return ConcretelIterator(this); }

UML: Iterator Pattern

Client

Collection
+createTraversalObject(): TraversalAbstraction

TraversalAbstraction
+first()
+next()
+isDone()

MapTraversal

ListTraversal

ListCollection
+createTraversalObject()

MapCollection
+createTraversalObject()

return new ListTraversal(this);
Observer Pattern

• **Purpose:** to allow multiple objects to observe when another object changes.
• **Outline:**

```java
Class `java.util.Observable`: the item being watched.
Classes to be watched extend this class.
Methods (the most important ones):
    void addObserver(Observer o), boolean hasChanged(), void notifyObservers()
interface `java.lang.Observer`: an object that wants to know when the watched item changes.
    Methods: void update(Observable o, Object arg)
```

• **Uses:**
  – As an alternative (or enhancement) to MVC, where each view observes the model.
  – RSS

---

**UML: Observer**

```plaintext
Subject
    +Attach(in Observer)
    +Detach(in Observer)
    +Notify()

observer
```

```plaintext
Observer
    +Update()
```

```plaintext
ConcreteSubject
    -subjectState
    +GetState()

subject
```

```plaintext
ConcreteObserver
    -observerState
    +Update()
```

```plaintext
foreach o in observers
    o.Update()
```

```plaintext
return subjectState
```

```plaintext
observerState = subject.GetState()
```
Sample Code

• How can the **Observer pattern** improve the design of *Fraud Detection* system?

Adapter Pattern

• Intent:
  – implement an interface known to one set of classes so that they can communicate with other objects that don't know about the interface

• Context:
  – want to use a class in a way that its original author didn't anticipate
    • E.g. write data to a string instead of to a file
    • Or apply regular expressions to streams instead of to strings
Adapter (cont'd)

• Motivation:
  – You want to use a class as though it implemented an interface that it doesn't actually implement
  – You do not want to modify or extend that class
  – You can translate the operations you want to perform to the ones the class actually implements

• Solution: create an adapter that implements the interface you want, and calls the methods the class has

UML: Adapter Pattern
Adapter examples

A legacy Rectangle component's display() method expects to receive "x, y, w, h" parameters, but the client wants to pass "upper left x and y" and "lower right x and y". This incongruity can be reconciled by adding an additional level of indirection – i.e. an Adapter object.

![Diagram showing the Adapter pattern with interfaces and classes related to shape display.]