Design patterns

Lecture 9

EE 564

Daqing Hou, Winter 2007

Outline

- Motivation for design patterns
- Composite and visitor (+interpreter)
- Indirection (proxy, decorator, and adapter)
- Procedures as objects (strategy, command)
- Observer pattern
- Flyweights and singletons
- Bridge and state

Composite and visitor

- Widget hierarchy (AWT/Swing)
- ASTNode in JDT DOM
- Resources in Eclipse workspace

- Base type hierarchy and additional behavior
- Traversal of composite
- Interpreter versus visitor
- Eclipse AST as an example

Indirection in patterns

- Obtain interposed object from original object, which
  - In adapter, exhibits different behavior
  - In proxy, exhibits identical behavior and preserves original type
  - In decorator, exhibits additional behavior and preserves original type
- These patterns can be implemented with composition or inheritance. But composition and inheritance are not only means of implementation.
- Eclipse o.e.core.runtime.IAdaptable

Procedures as objects

- Values can be stored in data structures, passed into procedures, and returned from function calls.
- Sometimes need to treat procedures as values.
- If languages do not support function pointers, can create objects for procedures.
- Two patterns (strategy & command) use this technique.

examples

- java.lang.Comparable
- java.util.Comparator
- javax.swing.tree.TreeCellEditor

- o.e.jface.action.IAction
- java.lang.Runnable

Comparable

- public interface Comparable {

```
/**
 *Effects: if this is less than x, returns -1;
 * if this is greater than x, returns 1;
 * otherwise, returns 0.
 *Exceptions:…
 */
public int compareTo(Object x) throws
   ClassCastException, NullPointerException
}
```

- **java.lang.Comparable**

  All Known Implementing Classes: BigDecimal, BigInteger, Byte, ByteBuffer, Character, CharBuffer, Charset, CollationKey, Date, Double, DoubleBuffer, File, Float, FloatBuffer, IntBuffer, Integer, Long, LongBuffer, ObjectStreamField, Short, ShortBuffer, String, URI

Using Comparable to sort

```
void sort(Object v[]){
   for (int i=v.length-1; i>-1; --i){
        for (int j=0; j<i; ++j){
           vj = (Comparable)v[j];
           if (vj.compareTo(v[j+1])>0)
              // swap v[j], v[j+1]
        }
   }
}
```

Comparator

- public interface Comparator {

```
/**
 *Effects: if x is less than y, returns -1;
 * if x is greater than x, returns 1;
 * otherwise, returns 0.
 *Exceptions:…
 */
public int compare(Object x, Object y) throws
   ClassCastException, NullPointerException
}
```

- Java.util.Comparator

Using Comparator to sort

```
void sort(Object v[], Comparator comp){
   for (int i=v.length-1; i>-1; --i){
        for (int j=0; j<i; ++j){
           if (comp.compare(v[j],v[j+1])>0)
              // swap v[j], v[j+1]
        }
   }
}
```

IAction

```
package org.eclipse.jface.action;
/**
 * An action represents the non-UI side of a command which can be triggered
 * by the end user. Actions are typically associated with buttons, menu items,
 * and items in tool bars. The controls for a command are built by some
 * container, which furnished the context where these controls appear and
 * configures them with data from properties declared by the action.
 * When the end user triggers the command via its control, the action's run
 * method is invoked to do the real work.
 * …
 * Clients should subclass the abstract base class <code>Action</code> to define
 * concrete actions rather than implementing <code>IAction</code> from scratch.
*/
public interface IAction {
 …
 public void run();
 …
}
```

Strategy versus command

- With the strategy pattern, the using context expects a certain behavior from the procedure.
- With the command pattern, the context expects only an interface.
- In either pattern, the procedure may make use of data provided by the context.
- The strategy that an object uses can vary. The possible ways of changing is open ended.

Objects can see and listen too

- A change in one object is of interest to several other objects.
- Examples:
  - document changes
  - email arrival
  - file cache in distributed systems
  - EventListener in JDK
- Observer pattern; subject and observer
- Observer pattern is also known as **publish/subscribe**, or by another name **listener** in certain cultures (e.g., JFC Swing).

Observer pattern

- ```
  class Subject{
  addObserver(Observer)
  notify(Change c) // calls update() on each
  observer
  … // state info
  }
  ```
- ```
  class Observer {
   update() // Observer-specific response
  }
  ```

- Variations in practice

- Two key aspects to consider:
  state (change&context) + control

Observer pattern: state

- What constitutes a change in the subject?
- What information an observer needs in order to respond to a change in the subject?

Observer pattern: control

push versus pull

- Push:
  When a change happens, subject will send observers all state information.
- Pull:
  Subject only notifies observers about a change; observers need to query subject subsequently for the nature of the change.
- Make a difference on performance in distributed computing.

Observer pattern: control

Further decouple from subject order of notification & observers

- Mediator:
  - observers register interest to mediator;
  - subject announces changes to mediator;
  - mediator delivers changes to observers.
  - uses push model.
- Mediator localizes communication so that it
  - can be symmetric among the `colleagues'.
  - can be prioritized by the mediator.
- Mediator can be generalized to a *white board.*

Observer pattern: control

- Mediator can be generalized to a *white board,* which is less restrictive on the kinds of information to be exchanged.
- White board can also be used as an alternative to direct calls on methods.
  - callers separated from callees.
  - no need for synchornization: caller can come back later to the white board to check for callees' response.

Observer pattern: Discussion

When is it appropriate to use this pattern?

Flyweights

- If a program creates too many objects, it may consume too much memory.
- Flyweight pattern helps avoid creating logically equivalent objects more than once.
- The name flyweight indicates that too many objects, even if they are small, can cause trouble in memory usage.
- How severe a problem?
  - identifiers in a program. 1000 classes, 10 vars each, each var used 10 times, 10 char's (2 bytes each char), each String object 44+20 bytes. 6.4 MB
  - how many classes does Eclipse have?

Flyweights

- ```
  class IdTable {
  private Hashtable words;
  String makeWord(String id){
      word = words.get(id);
      if (word!=null) return word;
      words.add(id); return id;
  }
  }
  ```
- This example used String. In general the pattern applies to any object.
- The flyweight object must define equals() for object equivalence.
  ```

Flyweights

Eclipse example:
StringPool, IStringPoolParticipant

Singletons

```
class Single {
private Single(){…}
private static Single instance=null;
public static Single getInstance(){
 if (instance!=null) return instance;
 instance = new Single();
}
}
```

- `clientMethod(){s= Single.getInstance();`
  `s.doSomething();…}`
  versus
  `clientMethod(Single s){ s.doSomething();…}`
- Eclipse examples: Workspace, BuilderManager (most managers?)

State

- Varies rep. of a context type at runtime.

```
class Set {
private SetRep rep;
int size;
int low, high;
void insert(Object o){
    if (rep.contain(o)) return;
    ++size; rep.insert(o);
    if (size>low) { rep=BigSet(rep);}
}
}
```

- SetRep has two subclasses: BigSet and SmallSet.
  - BigSet uses Hashtable
  - SmallSet uses Vector

Bridge

- Bridge connects a **type** hierarchy with an **implementation** hierarchy.

```
class Set {
private SetRep rep;
}
class ExtSet extends Set {
void union(Set);
void diff(Set);
}
```

- SetRep has two subclasses: BigSet and SmallSet.
  - BigSet uses Hashtable
  - SmallSet uses Vector
- Implementation hierarchy should avoid referencing the type hierarchy.
- Note the similarity between state and bridge.

List of patterns discussed

1. Iterator
2. Observer
3. Mediator
4. Composite
5. Interpreter
6. Visitor
7. State
8. Strategy
9. Command
10. Adapter
11. Proxy
12. Decorator
13. Bridge
14. Flyweights
15. Singletons
16. Template pattern*
17. Factory methods*
18. Factory*

patterns not discussed

1. Prototype
2. Builder
3. Façade
4. Chain of responsibility
5. momento