Using Design Patterns to Elaborate upon Design Models

Moving from Analysis to Design

Examine Design models and initial code to:

• Improve cohesion
• Reduce coupling
• Enhance Reusability

GoF Design Patterns should be used to rewrite code to promote these goals
Concordance Example

A Concordance consists of an alphabetized list of words appearing in a short document together with an ordered list of the distinct line numbers of lines in which each word appears. Design and implement a short document concordance program. Your program should be able to accept any document, generate a Concordance for the document, and display (or print) the document with its concordance.

The domain model will contain the following classes:

- Concordance
- Document
- Word
- WordList
- Line
- LineNumberList
Concordance Domain Model

Concordance makes/displays WordList

Document parses *

Line contains *

Word forms *

LineNumberList
Implementation

Consider class WordList. It has the following requirements:

Quick Access -- HashTable $O(1)$, BinarySearchTree $O(lg(n))$, Vector $O(n)$

Alphabetically ordered display -- BinarySearchTree $O(ln)$, Vector $O(n)$, HashTable

Choose a BinarySearchTree as the most appropriate data structure

Next, consider class LineNumberList.

Line numbers are entered in the order they appear, and are read from front to back.

Consider using either a Vector or a Queue to list line numbers. Choose the Vector for simplicity and because it can be read non-destructively.
Class Diagram – First Iteration

Concordance

Document

Line
number

BinSTree
root

BinSTreeNode
children
parent

Word

Vector

holds

0..2
Consequences of Implementation 1

Class Interfaces

class Word {
    private String word;
    private Vector theLines;

    public Word (String s, int lineNum) { …. }
    public void addLine(int num) {
        theLines.add(new Integer(num));
    }
    public boolean equals(Word w) { … }
    public String toString( ) { … }
    public String getWord( ) { … }
    public String getLineNums( ) { … }
}

The statements highlighted in blue indicate attributes and operations flagged by a “House Unwordy Activities Committee” as not properly belonging to class Word.

In the interest of high cohesion we will redesign the class
Modified Class Diagram

TreeIterator
  root

BinSTree
  root

BinSTreeNode
  children
  parent

holds

Association
  key

Word

Vector
Who is responsible for creating Associations? Concordance?

Better Solution – use a Builder to construct Association objects

Builder Pattern

Builder

Director

ConcreteBuilder

Product

(Concordance)

(Concource)

(Association)
Implementation of Builder Pattern

```java
public class Concordance {
    private Builder assBuilder = new AssocBuilder();
    private Builder docBuilder = new DocumentBuilder();
    private BinsSTree bst = new BinsSTree();

    public void readLines(Document doc) {
        String delims = " \n\t,.?;:");
        for (int line = 1; true; line++) {
            try {
                String text = doc.readLine();
                if (text == null) return;
                text = text.toLowerCase();
                Enumeration e = new StringTokenizer(text, delims);
                while (e.hasMoreElements())
                    enterWord((String)e.nextElement(), line);
            } catch (IOException e) {System.err.println(e.toString());}
        }
    }

    private void enterWord(String word, int line) {
        assBuilder.buildPart(word);
        Association ass = assBuilder.getPart();
        if (!bst.contains(ass)) {
            ass.addLine(Integer(line));
            bst.add(ass);
        } else {
            boolean flag = false;
            Iterator itr = bst.iterator();
            while (itr.hasNext() && !flag) {
                Association visit = (Association) itr.next();
                if (visit.equals(ass)) {
                    flag = true;
                    visit.addLine(Integer(line));
                }
            }
        }
    }
}
```
Implementation of Builder Pattern

class AssocBuilder implements Builder{
    private Association theProduct;

    public void buildPart() {
        theProduct = new Association();
    }

    public void buildPart(String word) {
        Word w = new Word(word);
        theProduct = new Association()
        theProduct.addWord(w);
    }

    public Association getPart() {
        return theProduct;
    }
}

class Association implements Comparable{
    private Word word;
    private Vector v;
    private String key;

    public Association() {
        word = null;    v = new Vector();    key = null;
    }

    public addWord(Word w) {
        word = w;    key = w.getWord();
        theProduct.addWord(w);
    }

    public addLine(Integer lineNum) {
        v.add(lineNum);
    }

    public Association getPart() {
        return theProduct;
    }

    //methods equals, compareTo, toString, etc.
}
Builder Pattern

Intent – Separate the construction of a complex object from its representation, so that the same construction process can create different representations (in our example – Association and Document)

Participants

- **Builder** – an Abstract Interface for creating parts of a Product object.
- **ConcreteBuilder** –
  - Implements the Builder Interface
  - Defines and keeps track of the representation it creates
  - Provides an interface for retrieving the product
- **Director (Concordance)**
  - Constructs a Product object using the Builder Interface
- **Product (Association)**
  - Includes classes that define the constituent parts, including interfaces for assembling the parts into the final result.
Builder Pattern

Collaborations

• The client creates the Director object and configures it with the desired Builder object.
• The Director notifies the ConcreteBuilder whenever a part of the Product should be built.
• The Builder handles requests from the Director and adds parts to the Product
• The client retrieves the Product from the Builder
Alternative Design

Follow the pattern more closely and create a separate Director that directs the construction of the BinSTree and the Document.
public class Concordance {
    private Builder treeBuilder = new TreeBuilder();
    private Builder docBuilder = new DocumentBuilder();
    private DocParser director = new DocParser(treeBuilder);
    private Document theText;

    public Concordance() {
        director = new DocParser(treeBuilder);
    }

    public void makeConcordance(String filename) {
        docBuilder.buildPart(filename);
        theText = docBuilder.getPart();
        director.constructConcordance(theText);
        bst = treeBuilder.getResult();
    }

    public void printConcordance() {
        // ...
    }
}
class DocParser {
    private BinsSTree bst = new BinSTree( );
    private Association ass;
    private Document theText;
    private TreeBuilder treeBuilder;
    public constructConcordance(Document doc) {
        theText = doc;
        readLines(theText);
    }
    public void readLines(Document doc) {
        String delims = " 	
.,!?;:";
        for (int line = 1; true; line++) {
            try {
                String text = doc.readLine( );
                if (text == null) return;
                text = text.toLowerCase( );
                Enumeration e = new StringTokenizer(text, delims);
                while (e.hasMoreElements( ))
                    enterWord((String)e.nextElement( ),line);
            } catch (IOException e) {System.err.println(e.toString( ));}
        }
    }
    private void enterWord(String word, int line) {
        Word w = new Word(word);
        treeBuilder.buildPart(w, line);
    }
}

Alternative Design Classes
class TreeBuilder {

    private BinSTree bst = new BinSTree();
    private Association ass;

    public void buildPart(Word w, int line) {
        ass = new Association(w, new Integer(line));
        if (!bst.contains(ass)) {
            bst.add(ass);
        } else {
            boolean flag = false;
            Iterator itr = bst.iterator();
            while (itr.hasNext() && !flag) {
                Association visit = (Association) itr.next();
                if (visit.equals(ass)) {
                    flag = true;
                    visit.addLine(Integer(line));
                }
            }
        }
    }

    public BinSTree getResult() {
        return bst;
    }
}
Additional Patterns

Singleton

Adapter

Composite
Singleton Pattern

Intent

Ensure a class only has one instance, and provide a global point of access to it.

Applicability

Use Singleton pattern when

• There must be exactly one instance of a class, and it must be accessible to clients from a well-known access point.

• When the sole instance should be extensible by sub-classing, and clients should be able to use an extended instance without modifying their code.

Consequences

1. Controlled access to sole instance. Singleton class encapsulates its sole instance and has strict control over how and when clients access it.

2. Reduced name space. It avoids polluting the name space with global variables that store sole instances.

3. Permits refinement of operations and representation. It can be subclassed.
Singleton Pattern – Example code

Singleton class declaration

class Singleton {

public:
    static Singleton* Instance( );

protected:
    Singleton( );

private:
    static Singleton* _instance;

}

Implementation of Singleton

Singleton * Singleton:: _instance = 0;
Singleton * Singleton::Instance( ) {
    if (_instance == 0)
        _instance = new Singleton( );
    return _instance;
}

Adapter Pattern

Class Adapter

Adapter lets classes work together that couldn’t otherwise because of incompatible interfaces.
Adapter Pattern

Client → Target
  request( )
  adaptee

Adaptee
  specialRequest( )

Adapter
  request( )
  adaptee → specialRequest( )

Object Adapter

Adapter can add additional functionality to the Adaptee object that the Adaptee object lacks but that Target requires.
Adapter (a.k.a. Wrapper)

Applicability

Use the Adapter pattern when

• You want to use an existing class and its interface does not match the one you need.

• You want to create a reusable class that cooperates with unrelated or unforeseen classes, that is, classes that don’t necessarily have compatible interfaces.

• (object adapter) you need to use several existing subclasses, but it’s impractical to adapt their interface by subclassing every one. An object adapter can adapt the interface of its parent class.

Participants

Target
-- defines the domain-specific interface that Client uses.

Client
-- collaborates with objects conforming to the Target interface.

Adaptee
-- defines an existing interface that needs adapting.

Adapter
-- adapts the interface of Adaptee to the Target interface
Composite Pattern

**Component**
- Operation()
- Add(Component)
- Remove(Component)
- GetChild(int)

**Leaf**
- Operation()

**Composite**
- Operation()
- Add(Component)
- Remove(Component)
- GetChild(int)

For all g in children:
  g.Operation();
Composite Pattern

Applicability

You want to be able to ignore the difference between compositions of objects and individual objects. Clients will treat all objects in the composite structure uniformly.

Participants

• Component
  -- declares the interface for objects in the composition
  -- implements default behavior for the interface common to all classes, as appropriate
  -- declares an interface for accessing and managing its child components.

• Leaf
  -- defines behavior for primitive objects in the composition.

• Composite
  -- defines behavior for components having children.
  -- stores child components
  -- implements child-related operations in the Component interface.
Composite Pattern

Collaborations

Clients use the Component class interface to interact with object in the composite structure. If the recipient is a Leaf, then the request is handled directly. If the recipient is a Composite, then it usually forwards request to its child components, possibly performing additional operations before and/or after forwarding.

Consequences

• Defines class hierarchies consisting of primitive objects and composite objects. Wherever client code expects a primitive object, it can also take a composite object.

• Makes the client simple. Clients can treat composite structures and individual objects uniformly.

• Makes it easier to add new kinds of components. Newly defined Composite or Leaf subclasses work automatically with existing structures and client code.

• Can make your design overly general. It makes it harder to restrict the components of a composite.
Example of Composite Pattern

- Client
- Equipment
  - Operation()
  - Add(Component)
  - Remove(Component)
  - GetChild(int)
- *
- For all g in children
  - g.Operation();

FloppyDisk
  - Operation()

CDDrive
  - Operation()
Sample Code

class Equipment {
    public:
    virtual ~Equipment( );
    const char* name() { return _name; }
    virtual Watt Power( );
    virtual Currency NetPrice( );
    virtual Currency DiscountPrice( );
    virtual void Add(Equipment *);
    virtual void Remove(Equipment *);
    virtual Iterator<Equipment*> * CreateIterator( );

    protected:
    Equipment (const char *);

    private:
    const char* _name;
};

class FloppyDisk : public Equipment {
    public:
    FloppyDisk(const char*);
    virtual ~FloppyDisk( );
    virtual Watt Power( );
    virtual Currency NetPrice( );
    virtual Currency DiscountPrice( );
};

class Chassis : public CompositeEquipment{
    public:
    Chassis (const char* );
    virtual ~Chassis( );
    virtual Watt Power( );
    virtual Currency NetPrice( );
    virtual Currency DiscountPrice( );
};
Sample Code for Composite Example

class CompositeEquipment: public Equipment {
public:
    virtual ~CompositeEquipment( );
    virtual Watt Power( );
    virtual Currency NetPrice( );
    virtual Currency DiscountPrice( );
    virtual void Add(Equipment *);
    virtual void Remove(Equipment *);
    virtual Iterator<Equipment *> * CreateIterator( );

protected:
    CompositeEquipment(const char *);

private:
    List<Equipment *> _equipment;
}

An implementation of NetPrice( )

Currency CompositeEquipment::NetPrice( ) {
    const_iterator i = CreateIterator( );
    Currency total = 0;
    for (i -> first( ); i -> isDone( ); i -> next( ) ) {
        total += i -> currentItem( ) -> NetPrice( );
    }
    delete i;
    return total;
}
Assume we have additional Equipment classes such as Bus, Cabinet, etc. We can assemble equipment into a (simple) computer (CompositeEquipment object).

```cpp
Cabinet * cabinet = new Cabinet("PC Cabinet");
Chassis * chassis = new Chassis("PC Chassis");
Cabinet -> Add(chassis);
Bus * bus = new Bus("MCA Bus");
bus -> Add(new Card("100 Mbs Ethernet");
chassis -> Add(bus);
chassis -> Add (new FloppyDisk("3.5 in Floppy");
cout << " the net price is " << chassis -> NetPrice( ) << endl;
```