CPS122 Lecture: Design Patterns   Last revised March 4, 2019

Objectives

1. To introduce and illustrate the idea of design patterns
2. To introduce some key design patterns students have used or will use:
    • Singleton
    • Abstraction-Occurrence
    • Iterator
    • Observer
    • Factory
    • Decorator (Wrapper/Filter)
    • Adapter
3. To demonstrate how several design patterns can be used together to solve a specific problem
4. To introduce the DisplayableCollections package

Materials

1. "Gang of Four" Design Patterns book to show
2. Projectable of pp. 6-7 of book (general description of "pattern language"
3. Projectable of p. 127-128 (the Singleton pattern)
4. Projectable of figures 6.1 and 6.2 from Lethbridge/Langaniere
5. Projectable of String array operations not using an iterator
6. Demo and handout of Iterator demo
7. Spreadsheet with multiple charts demo
8. Projectable of abstract sequence diagram for Observer pattern (p. 295)
9. Projectable of simple Observer demo (tic-tac-toe)
10. Demo and handout of Observable demo
11. Demo and handout of swing program with buttons with different looks and feels
12. Projectable of "Gang of Four" book page 175 (Decorator pattern)
13. Demo of my version of library showing multiple patterns use in solving a problem
14. Excerpts from source code for AbstractDisplayableMap and ListModelAdapter.
15. Documentation (online) for DiplayableCollections package
16. Handout of demonstration use of DisplayableCollections for an address book

I. **Introduction**

   A. One of the characteristics of an expert in many fields is that the expert has learned to recognize certain *patterns* that characterize a particular problem or call for a particular approach to a solution.

   Examples:

   1. A civil engineer uses certain structural patterns when designing bridges, highways, etc.

   2. A Medical Doctor recognizes certain patterns of symptoms as indicative of certain diseases.

   B. In the world of OO software, one key concept is the concept of *design patterns* - standard patterns of relationships between objects in a system (or portion of a system) that constitute good solutions to recurring problems.

   QC Question l

   1. A key book in this regard is the book *Design Patterns* by Gamma, Helm, Johnson, and Vlissides (known in OO circles as "the Gang of Four").

      a) SHOW

      b) This book classifies the patterns it discusses into three broad categories, which were also discussed in the book. What are they?

      ASK - QC Question m

   2. One important characteristic of the study of patterns is the idea of giving each pattern a name, so that when people talk they can use the name of the pattern and others will know what they are talking. (Unfortunately, some key patterns have more than one name; but at least we don't have the issue of the same name referring to different patterns!)

      QC Question n - then pick up

3. In addition to giving the overall pattern a name, we also give a name to the various objects that participate in the pattern. Here, the name given to each object is mean to describe its *role* and *responsibilities*. The object does not have to actually be called by this name - the purpose of the name in the pattern is simply to help us understand how responsibilities are apportioned.

   EXAMPLE: At the present time, the person who has the role and responsibilities of President of the United States is Donald Trump. We may refer to him as "the president" - but his parents did not name him "President", they named him Donald!

C. In this lecture, we will try to illustrate the concept of design patterns by introducing several examples of design patterns - each of which is a good way to solve a particular kind of problem in a software system.

   1. Obviously, one only wants to use the pattern if it meets a real need, of course!

   2. Most actually are patterns you have already used in labs or your project.

D. We will begin with an example you've already used, and show how to describe it using "pattern language".

   In standard discussions of patterns (including the "Gang of Four" book), it is common to describe patterns in a standard way.

   PROJECT: pp. 6-7 of "Gang of Four" book

   (Note: we will discuss patterns from other sources as well - but the "style" of this book is an accepted standard regardless of where the pattern descriptions come from.)

II. **The Singleton Pattern**

A. We will begin by talking about a pattern that you are already familiar with. One of our goals will be to use it as an example of how patterns are described. The singleton pattern is applicable wherever there is a kind of object where it is necessary that there be one and only one copy of this object in existence.

Where have you seen this pattern?

ASK

B. It belongs, therefore, to the category of creational patterns.

C. Project, go over pages 127-128 of the "Gang of Four" book

III. **The Abstraction-Occurence Pattern**

A. Another pattern you have been using in your library project is a pattern called the Abstraction-Occurrence pattern. Can anyone describe this pattern and how it relates to the library project?

ASK

B. In terms of the three categories of pattern we discussed earlier, to which category would this pattern belong?

ASK

Structural

C. This pattern actually doesn't appear in the "Gang of Four" book. The author of one book that does discuss it introduces this pattern this way: "Often in a domain model you will find a set of related objects that we will call *occurrences*; the members of such a set share common information but also differ from each other in important ways.

Examples: *ASK*

D. This particular situation is one to which it turns out there are both good and bad solutions. The idea is to represent the various occurrences in some way that avoids duplicating the common information.

1. In general, the clean solution is to use two classes - an « abstraction » class and an « occurrence » class.

   PROJECT: Figure 6.1 of Lethbridge/Langaniere

2. The author of this book also notes that there are a number of possible "antipatterns" (or poor solutions.) Let's look at several and discuss why each is not a good way to solve the problem.

   PROJECT: Figure 6.2. of Lethbridge/Langaniere

   Discuss why each "solution" is not good

   a) This example is undesirable because of replication of information. It would become even worse if some of the common information were subject to change, since it would have to be updated multiple times.

   b) This example is undesirable because each new instance of the abstraction (e.g. new book title) would require creation of a new <u>class</u> - necessitating recompilation of the software and resulting in a proliferation of classes.

   c) This example is undesirable because it involves replication of information again - even though the occurrence class does not have attributes for the information, its base class (and therefore every one of its objects) does.

E. This particular problem illustrates one of the virtues of studying design patterns - you can find a clean solution to a problem, while avoiding mistakes that would otherwise be easy to fall into.

IV. **The Iterator Pattern**

A. Another pattern you have used is the Iterator pattern. The Iterator pattern prescribes three roles: a collection, an iterator over the collection, and an object that uses the iterator to systematically visit all the items in the collection.

Where have you seen this pattern before?

ASK

B. What category of patterns would this belong to?

ASK

Behavioural

C. To see the motivation for the pattern, suppose that we had a collection of Strings, and we want to perform various operations on all the strings in the collection at various points in the program. Suppose, further, that the collection of Strings were stored in an array.

```
String [] someCollection;
```

1. Now, we could systematically print all the strings by using a loop with an index into the array:

```
for (int i = 0; i < someCollection.length; i ++)
   System.out.println(someCollection[i]);
```

PROJECT

2. Suppose, at some other point in the program, that we want to print out the shortest string in the collection (and suppose that they are not necessarily stored in order of length.) The following code would work (assuming the collection contains at least one string.)

```
String shortest = someCollection[0];
for (int i = 1; i < someCollection.length; i ++)
  if (someCollection[i].length()< shortest.length())
     shortest = someCollection[i];
```

PROJECT

3. In similar fashion, we could write code to find the alphabetically-first (or last) string, or to print out all the strings that begin with the letter 'A', or whatever.

4. Now suppose we decide to use a different collection to store the strings - perhaps a Set or a List or whatever. All of the code would need to be modified to change the way it accesses the strings; further, the code would need to know the details of how the collection is stored.

D. A better approach is to separate the notion of "iterating over all the elements in the collection" from the details of how the collection is stored. This decoupling is accomplished by an *iterator*.

1. An iterator is always attached to some collection. Usually, a collection has some method that creates an iterator for the collection - so the way that an iterator is constructed is by asking the collection to create one. At any time, a given collection may have any number of iterators in existence.

```
┌─────────────┐                              ┌─────────────┐
│ Collection  │ 1                          * │  Iterator   │
│             │◀─────────────────────────────│             │
│             │                              │             │
└─────────────┘                              └─────────────┘
```
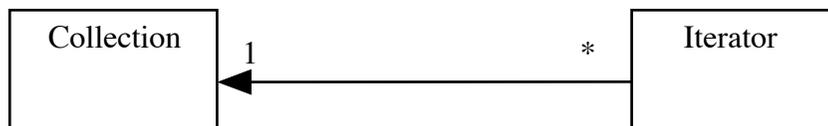
2. Moreover, an iterator always either refers (implicitly) to some element in the collection, or is at the end of the collection. If a collection has several iterators, each has its own position relative to the collection.

3. An iterator has three basic responsibilities:

   a) Report whether or not it currently refers to an element of the collection.

   b) If it does refer to an element of the collection, provide access to that element

   c) Advance to the next element of the collection

4. An iterator provides access to the elements of the collection is some order that is defined by the underlying collection - but which always satisfies certain properties.

   a) A newly-constructed iterator always refers to the "first" element of the collection. (Where "first" is defined by the underlying collection - e.g. if the collection is a Set, the choice may appear arbitrary but actually obeys some consistent rule that the user of the iterator need not be aware of.)

   b) If the iterator is used to systematically visit each element of the collection (by repeatedly accessing the current element and advancing to the next), every element of the collection will be visited exactly once in some collection-specified order, and then the iterator will become past the end of the collection.

5. If code is written to access all the elements of a collection through an iterator, and the kind of collection is changed, the only other code that may (or may not) need to be changed is the code that asks the collection to create the iterator.

E. The `java.util` package defines an Iterator interface, and each of the collections it supports have an iterator() method that creates a new iterator and returns it.

   1. The Java iterators differ slightly from the responsibilities I have presented above:

      a) The operations of accessing the current element and moving on to the next element are combined in a single method.

      b) An iterator can also support a method for removing the last element visited from the collection.

      c) Thus, the Java Iterator interface contains three methods

        (1) `hasNext()` - true unless past the end of the collection

        (2) `next()` - combines accessing the current element with moving on to the next

        (3) `remove()` - remove from the collection the last element that was returned by next(). [An iterator for a particular type of collection is not required to actually support this operation.]

  2. As you discovered in lab, you cannot create an iterator for a Map directly. That's because a Map actually involves *two* collections - a collection of keys, and a collection of values - plus an association between members of the two collections. Thus, to iterate over a Map, you must use the Map's `keySet()` or `values()` method to get access to the appropriate collection, and then get an iterator from it.

F. An example of making use of the Iterator pattern

  1. Handout Discuss Iterator demo code

  2. Run it

  3. Go over it

  4. **AN IMPORTANT NOTE:** Unless one is building a collections package, one normally doesn't have to actually implement iterators - just use them. The implementation is include here so you can see how iterators actually work.

G. A year from now, you will be learning how collections are actually implemented, and will use C++. The C++ standard library also defines iterators, which work the same way, though the names of the methods are different:

1. A collection will have a method called `begin()` to create an iterator that refers to the start of the collection.

2. A collection will have a method called `end()` to create an iterator that is one past the end of the collection. Two iterators to the same collection can be compared to see if they refer to the same point by using ==.

3. The element an iterator refers to is accessed by using the * operator.

4. An iterator is advanced to the next element by using the ++ operator.

   Thus, C++ code for printing all the strings in a collection of strings similar to the example we have done in Java would look like this:

   ```
   for (someCollection::iterator iter =
                            someCollection.begin();
        iter != someCollection.end();
          iter ++)
     -- code to write out * iter;
   ```

   (I've shown you this C++ code now, because you'll see a lot of code like this next semester, and it will be helpful to recognize then what pattern is being used.)

5. Actually, C++ iterators can be bidirectional - i.e. allowing one to either move forward or backward within the collection. Some collections support a reverse iterator that allows you to get an iterator that refers to the last item in the collection and then work backwards from there.

V. The Observer Pattern

A. The Observer pattern is useful when we have two kinds of objects called an observable (or subject) and an observer, that are connected in such a way that the observer needs to know about changes in the observable, but we want to minimize the coupling between these objects.

(The view and model classes in an MVC system are a good example of this - when the model changes, the view(s) may need to change - but we don't want to tightly couple the model and view. Thus far, we have not used the Observer pattern to implement an MVC system, largely because our example systems have been very simple.)

B. An example: spreadsheet with chart(s) based on data

DEMO

C. In the Observer pattern, we have two kinds of classes: an observable class, and one or more observer classes, with specific responsibilities.

  1. The observable maintains some information that is of interest to the observer(s), and is responsible to furnish that information to it/them upon request.

  2. Each observer is responsible to *register* itself with the observable, by calling some registration method.

  3. When the observable changes, it is responsible to notify each of its registered observers about the change. The notification may include some indication as to what has changed (though the pattern does not mandate this.)

  4. Each observer is responsible to have an appropriate method which the observer can invoke when it changes.

  5. Each observer, in turn, is responsible to look at the notification it received and - if something of interest to it changed - request appropriate updated information from the observable.

     PROJECT - Abstract sequence diagram for this pattern from "Gang of Four" book p. 295

D. The Java standard library provides support for this pattern

1. The API defines a <u>class</u> called `java.util.Observable` and an <u>interface</u> called `java.util.Observer`.

   a) `Observable` is a class because it implements behaviors that any observable object needs; but if an observable inherits them, then it does not need to implement them itself.

   b) `Observer` is an interface because all that is required to be an observer is that one have a method called `update()` that allows the observable to inform it of changes. What this method actually does varies greatly from situation to situation, so there is no benefit to inheriting any implementation.

2. An observer registers itself as being interested in being notified of changes to an observable by calling the method `addObserver()` of the observable.

3. Any code that changes an observable calls a method of the observable called `setChanged()` and then calls `notifyObservers()` to actually report changes to its observers. (Usually, these calls are part of the code of a method of the observable object.)

   a) `notifyObservers()` can be called without the object having changed, in which case nothing happens. Once `notifyObservers()` has been called the observable is considered unchanged until `setChanged()` is called again.

   b) Several changes can be made before observers are notified to reduce overhead, if desired.

4. When an observable has changed and `notifyObservers()` is called, the `update()` method of each registered observer is called.

a) The first parameter to `update()` is the observable that has changed.

b) The second parameter is an optional parameter to `notifyObservers()` that can specify the *nature* of the change. It is often null.

5. When an observer's `update()` method is called, it is responsible to use an appropriate method or methods to access the observable to get at the new information, and then to take appropriate action.

E. A simple example: A program that plays tic-tac-toe might have a model class called Board and a view class called BoardDisplay

PROJECT Sample code and go over use of pattern

F. Another example: Observer Demo

1. The program consists of an observable object that records a temperature, and three views that report the temperature using three different scales (Celsius, Fahrenheit, and Kelvin.) A new temperature can be typed in any view, and all three views are updated to reflect the new temperature.

2. *DEMO*

3. *HANDOUT* - source code - go over

VI. **The Factory Pattern**

A. The factory pattern is a fairly sophisticated pattern that is a bit hard to get a hold of. Perhaps the best way to get a handle on it is to show an example of a place where it is used. The illustration we will use is not one where the factory pattern is visible on the surface, but it plays a key role behind the scenes.

B. As you recall, earlier in the course we discussed the Java swing package, which incorporates the notion of "pluggable look and feel". In essence, what this means is that, when a GUI component (e.g. a button) is created, it displays itself using in the way appropriate to its look and feel. It turns out that how a component displays itself is determined when it is created.

1. Example - it is possible to create a swing program with several buttons that have different looks and feels. (Not a good idea in general, but we do this here for illustration.)

2. Demo program.

3. Handout / discuss code

C. Behind the scenes, this functionality is implemented by using *factories*.

1. A factory is an object that creates other objects. It is possible to have different factories that have the same interface - i.e. produce objects of the same general kind - but each produce their own kind of object.

2. In essence, Swing uses factories to produce the various components that can displayed, which are different for different looks and feels. Each factory has methods for producing each of the various kinds of components. (The explanation given here is vastly simplified.)

3. The `setLookAndFeel()` method chooses a particular factory, and when components are subsequently created, their visible representation is "manufactured" by the current factory

VII.The Decorator Pattern (also known as Wrapper, Filter)

A. PROJECT "Gang of Four Book" page 175.

B. In general, this decorator (wrapper) approach is useful whenever we have a class whose functionality we want to extend, but we have good reason not to modify the source code for the class itself. We put the added functionality into a wrapper, that also forwards original requests to the object within.

    1. We will look at a major example of this in the structure of the `java.io` package in a couple of weeks.

    2. We will also look at an example that makes use of this at the end of this lecture.

VIII.The Adapter Pattern

A. In general, we use the Adapter pattern when we have an object that provides the basic functionality we need, but doesn't have the interface we need. To do this, we encapsulate the object in an adapter object that provides the needed interface by "forwarding" operations to the encapsulated object.

```
Adapter object - provides the
desired interface by
"fowarding" operations to the
encapsulated object

    Object to be adapted -
    provides the needed
    functionality but not
    the needed interface
```

B. We will illustrate this shortly

IX. Putting it all Together - an Illustration of Patterns Use

A. To illustrate how patterns can be used, consider the following practical problem that could arise if fully implementing your Library.

It would be nice if various panes of the GUI could display lists of patrons and items, the former in alphabetical order of name and the latter in call number order.

1. To handle a patron's fines, we could go to the patron list, select the patron, and click a "Fines" button.

   DEMO in my version - note that supporting code is not implemented, but this is how interface would work if it were

2. To add copies of an item, we could go to the item list, select the item, and click an "Add Copy" button.

   Log in as manager on the manager pane, then DEMO in my version

3. To deal with reservations, we could go to the item list and click a "Reservations" button to deal with the reservations for that item.

   DEMO in my version - note that supporting code is not implemented, but this is how interface would work if it were

4. To keep our subsequent discussion simple, we will focus on dealing with the display of the patrons list.

B. It actually turns out that this is much more easily said than done! Here's why.

1. A `JList` (which is the GUI component that is used to display the patrons list) is backed by an object called the list model that keeps track of the items displayed in the list. A list model must implement the `ListModel` interface.

   SHOW `javax.swing.ListModel` documentation.

Note that the key functionality is the ability to access an element by position - which is needed to support operations like scrolling the list. (The `getElementAt(int)` method).

2. However, we want to store the patrons list in the library using some form of `Map` - and maps do <u>not</u> provide the ability to access items by position, but only by key.

3. Further, a sorted map (`TreeMap`) keeps items in key order (e.g. phone number); but we would prefer to have the list be displayed in alphabetical order of name.

C. We can begin to solve the problem by creating an <u>adapter</u> that allows a map to be used as the `ListModel` for a `JList`. To do this, it can

1. Maintain an internal vector of elements

2. At creation, get the list of elements from the appropriate map, sort them into the order they will be displayed in, and store them in the internal vector.

3. Provide access to the vector by position through the `getElementAt(int)` method required by the `ListModel` interface - since a `Vector` does support accessing elements by position.

D. However, this is not a complete solution. What if the list changes? (E.g. if we add or delete a patron?) Here, we could use the Observer pattern - i.e. the `ListModel` adapter is made an observer of the map - so that whenever the map is changed, the `ListModel` adapter updates its list.

1. Unfortunately, this is not easily possible, because maps do not implement the Observable interface!

2. So we make use of a third pattern - the Wrapper pattern. We wrap a `TreeMap` in an observable class that:

a) Implements the same interface as a `TreeMap` (`java.util.SortedMap`)

b) Forwards `SortedMap` operations that don't change the map to the encapsulated map

c) Forwards any operation that changes the map to the the encapsulated map and then notifies its observers that the map has changed.

E. We then get the following:

```
┌─────────────────────────┐        ┌─────────────────────────┐
│   DisplayableTreeMap     │        │     « observer »         │
│                          │        │   ListModelAdapter       │
│  ┌────────────────────┐  │        │                          │
│  │ java.util.TreeMap   │──┼────────┼──┐                       │
│  │                    │  │        │  │ java.util.Vector │    │
│  └────────────────────┘  │        │  └──────────────────┘    │
└─────────────────────────┘        └─────────────────────────┘
```
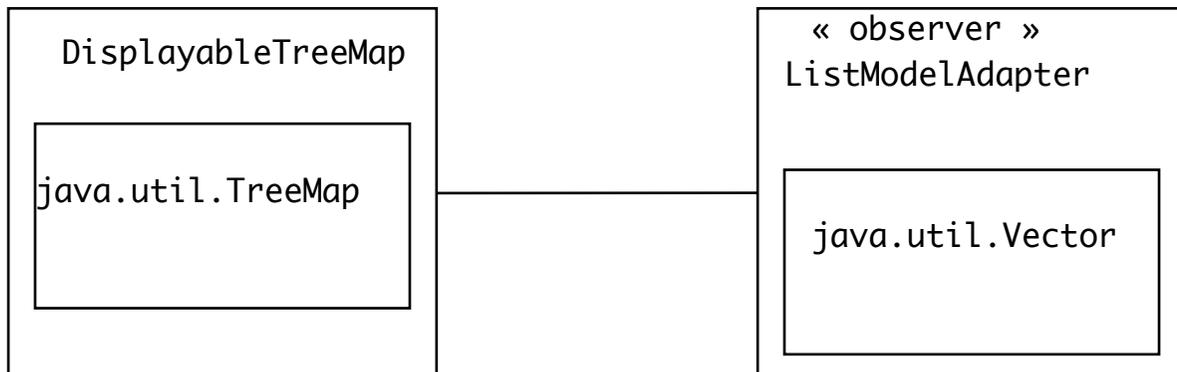
F. DEMO

1. Add a new patron

2. Edit a patron, changing the name

3. Delete a patron

G. Show Code

1. `AbstractDisplayableMap`

a) Variable `encapsulatedMap`; constructor,

b) `containsKey()`, `get()`, `put()`, `remove()`

c) `changed()`

2. `ListModelAdapter.ForCollection`

a) Variable `elements`

b) `getElementAt()`, `getSize()`

c) `update()` [in `ListModelAdapter`]

X. Using the DisplayableCollections pattern in your project

A. You don't need this package for Iteration 1. You may want to use it for Iteration 2 in connection with representing the lists of patrons, items, etc. However, use of this package is entirely optional,

B. All the code for this package is included in compiled form in the starter code you have been given for the project. Complete documentation for the package is available online.

SHOW

C. We will look briefly at a demonstration of how this can be used in code - a simplified version of the address book modified to make use of this package.

HANDOUT

1. Note imports for classes needed from displayablecollections package

2. Constructor:

a) The address book is stored in a `DisplayableTreeSet`

b) The GUI component displaying the list of persons is called `personList`. Its model is set equal to an object created by the `orderedListModel()` method of the address book.

c) The `Person` class defines a `toString()` method that determines what is displayed for a person. (End of handout)

   Note: `toString()` is part of the standard Object interface in Java, and defines how an object is to be displayed as a character string.

d) The `Person` class also implements the `Comparable` interface and defines a `compareTo()` method that specifies how persons are to be ordered in the display.

e) Whenever the address book is changed (by the New or Open use cases), the list model must also be updated - note how this is done in the code.

f) That's all that's needed - the list will display whatever is in the address book, and whenever the address book is updated, the address book will change.

   Methods that modify the address book (such as `doAdd()` do not need to include any code to update the GUI - this is done automatically by the observer pattern whenever the book is changed.