CPS122 Lecture: Design Patterns **Last revised March 10, 2022**

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
   - Observer
   - Iterator
   - Command
   - Decorator (Wrapper/Filter)
   - Adapter
3. To demonstrate how several design patterns can be used together to solve a specific problem

Materials

1. Projectables
2. Answers to QC questions 9l-n
3. "Gang of Four" Design Patterns book to show
4. Demo and handout of Iterator demo
5. Spreadsheet with multiple charts demo
6. Demo and code to project for Command demo
7. Demo and handout of Observable demo
8. Demo of Patterns demonstration version of library showing multiple patterns use in solving a problem. This is mine Iteration 3 with
   - addObserver in CollectionPane initially commented out
   - delete button added to PatronPane

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

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 Joseph Biden. We may refer to him as "the president" - but his parents did not name him "President", they named him Joseph!

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 - including the following:

   (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.)

   1. Pattern Name and Classification

   2. Intent - what problem is it solving?

   3. Structure - what configuration of interacting objects can be used to solve the problem?

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 exactly one and only one copy of this object in existence.

   Where have you seen this pattern?

   ASK

B. What category does it belong to?.

   ASK

   Creational

C. The intent of this pattern is to provide a solution to a situation where there must be exactly one copy of an object in existence, which is accessible from many other places in the program.

   1. One apparent alternative would be to use some sort of globally-visible variable to refer to the single instance, but this would not guarantee that exactly one object of the class is created.

      For example, the object representing the LibraryDatabase must be created exactly once at program startup, either by reading previously-saved data from a file or some initial state. We must ensure that once an object of this class is created, it is the one that is used in all cases.

   2. Another apparent alternative would be to use a class with static methods - in which case no object per se would exist. This will work if all we need is behavior, but not existence or state.

For example, it does work for the controller for some of the use cases in the Library project such as Return, Renew, Status, or Add Patron, Book, or DVD - but not for one like Checkout that needs to keep track of the state of the list of items to be checked out. (In this case, a new object needs to be created for each checkout,)

In the case of the Library, the Singleton pattern has another advantage. The Java serialization method we are using to save the library data saves objects. Thus, for the LibraryDatabase, we need an object (we cannot just use a class with static methods.) Singleton class - but we must ensure there is only one such object. The Singleton pattern ensures this.

D. Structure - a singleton class has the following structure - illustrated here by code from LibraryDatabase.

PROJECT ALL THREE BELOW

1. The class has a private static variable that refers to the one and only instance of the class.

```
private static LibraryDatabase theLibraryDatabase;
```

2. The class has a private constructor - which means that an object of this class cannot be created except by a method of the class itself.

```
private LibraryDatabase()
{
   ...
}
```

3. The class as a public static method that provides access to this variable. (In the Library example, this is called getInstance(). If no copy exists as yet (the reference to the singleton copy is null), then a new copy is created and the variable is set equal to it. Then, in either case, the singleton copy is returned.

```
public static getInstance()
{
    if theLibraryDatabase == null
    {
        theLibraryDatabase = new LibraryDatabase();
        ....
    }
    return theLibraryDatabase;
}
```

III. The Abstraction-Occurrence Pattern

A. Another pattern you have been using in your library project is a pattern called the Abstraction-Occurrence pattern. Where have you used this pattern?

   ASK

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

   ASK

   Structural

C. Can anyone describe this pattern and how it relates to the library project?

   ASK

D. 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.

E. This particular situation is one to which it turns out there are a number of solutions that look workable but turn out to have significant problems. For illustration, we'll use the relationship between a course and its sections.

All sections of a course share certain common information such as the course number, its title, its number of credits, and its prerequisites. But each section has its own meeting time and location.

1. One solution is to store all the information about the course in each section:

```
class Section
{
    // Common to all sections of this course
    String courseNumber;
    String title;
    int credits;
    String [] prerequisites;
    // Unique to this section
    String meetingTime;
    String location;
    ...
}
```

   PROJECT

   This means that, whenever we create a new Section, we have to record all the other information as well. Moreover, if any of the common information is changed (e.g. changing the credits or prerequisites) we need to update all the sections - and if we forget to change one we have inconsistencies that can give rise to big problems.

2. Another "solution" is even worse - create a distinct class for each course, and store the common information as class constants.

```
class CPS122Section
{
    static final String COURSE_NUMBER = "CPS122";
    static final String TITLE = "OO Software Development"
    static final int CREDITS = 4;
    static final String [] PREREQUISITES = {"CPS121" };
    // Unique to this section
    String meetingTime;
    String location;
    ...
}
```

PROJECT

Now, when we create a new section, we only need to enter the unique information, and if we update any item of common information, we just update one constant. But what are the consequences of this "solution"?

ASK

3. We might try solving the problem using inheritance, by having a class that stores the common information and a subclass that stores the unique information for each section:

```
class Course
{
    String courseNumber;
    String title;
    int credits;
    String [] prerequisites;
    ...
}
class Section extends Course
{
    String meetingTime;
    String location;
    ...
}
```

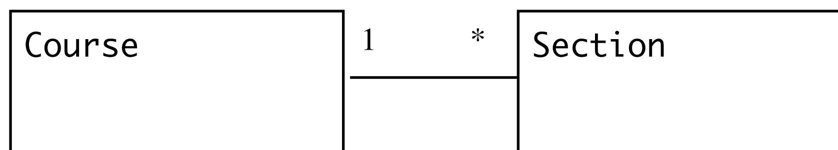
PROJECT

At first glance, this may appear to be a solution to our problem. But it's actually not.

a) Consider what happens where we add a new Section. Since it's a subclass of Course, it inherits all the fields of Course. Therefore, its constructor must include all the information, not just that unique to the Section.

b) Moreover, if we make a change to the common information, we need to change all the Sections, since each contains its own copy.

F. In general, the clean solution is to use two classes - an abstraction class (Course) and an occurrence class (Section)



PROJECT:

G. 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. (Sometimes, the solutions that appear to be solutions but actually are not are called antipatterns)

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;
```

PROJECT

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];
```

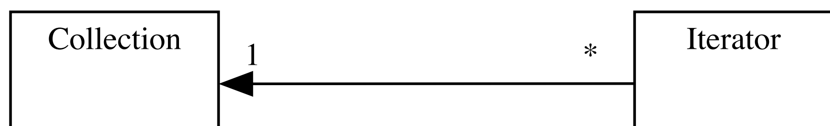
PROJECT

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.



PROJECT

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;
```

   PROJECT

   (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 Command Pattern

A. The next pattern addresses an issue that is quite common in GUI programs - support for do/undo/redo. It also can be used to handle a number of other issues we won't discuss.

We might illustrate this with a simple implementation of the "Towers of Hanoi" problem.

1. Describe problem

2. DEMO - show how Undo and Redo work with the various move operations and with Reset and ability to undo/redo operations.

3. Where else have you seen a facility like this?

   ASK for examples

4. At first glance, this may seem challenging to implement. But it turns out there is a design pattern that allows it to be implemented fairly straight-forwardly.

B. The heart of the idea is the notion of a Command object.

1. A Command object in a system offering undo has two behaviors:

   a) perform() - carry out the command

   b) undo() - undo the command

   In Java, this behavior can be specified by the following interface

```java
/** Interface implemented by Command objects
 */
public interface Command {

    /** Perform (do or redo) this command
     */
    void perform();

    /** Undo this command
     */
    void undo();
}
```

PROJECT

2. A Command object is created whenever an operation is initiated. For example, in a word-processing program

   a) Whenever a keyboard key is struck to insert a new character.

   b) Whenever the delete key is struck to delete a character.

   c) Whenever a the style of a range of text is changed (e.g. made boldfaced or italic) etc.

3. A Command object is a member of a class that implements the Command interface - with different classes used for different commands. The object that is created encapsulates the particular information that is needed to carry out the command e.g.

   a) What insertion character was struck

   b) The fact that the delete key was struck

   c) What range of text is to be restyled and what the new style is to be etc

4. As soon as a Command object is created, its perform() method is invoked. This method does two things

    a) It records the minimum amount of information needed about the state of affairs before the command is executed in order to allow the command to be undone.

        (1) For example, for a command to insert a character what is recorded is the position where it is being inserted.

        (2) For a command to delete a character what is recorded is the character that was deleted and the position from where it was deleted.

        (3) For a command to change the style of a range of text, what the range was and what the style(s) of the characters in the range were beforehand.

    b) It performs the actual operation

C. The program keeps track of Command objects using two stacks (last in first out lists) - both initially empty.

1. When a Command object is performed (and records the information needed for it to be undone), it is placed on top of the undo stack

2. When an undo operation is requested, instead of creating a Command object, the top item on the undo stack (the one most recently done) is removed and its undo() method is done. Then it is placed on top of the redo stack.

3. When a redo operation is request, instead of creating a Command object, the top item on the redo stack (the one most recently undone) is removed and its perform() method is done again. Then it is placed on top of the undo stack.

This can be realized by code in Java like the following:

a) Code executed when a command is initiated:

```
Command command = new ----
command.perform();
undoStack.push(command);
```

b) Code executed when an undo is performed:

```
Command command = undoStack.pop();
command.undo();
redoStack.push(command);
```

c) Code executed when a redo is performed:

```
Command command = redoStack.pop();
command.perform();
undoStack.push(command);
```

PROJECT

D. The basic mechanism for managing commands then, is quite simple. The critical part is creating the various classes that implement the Command interface.

SHOW Code for Towers of Hanoi problem

VI. 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.

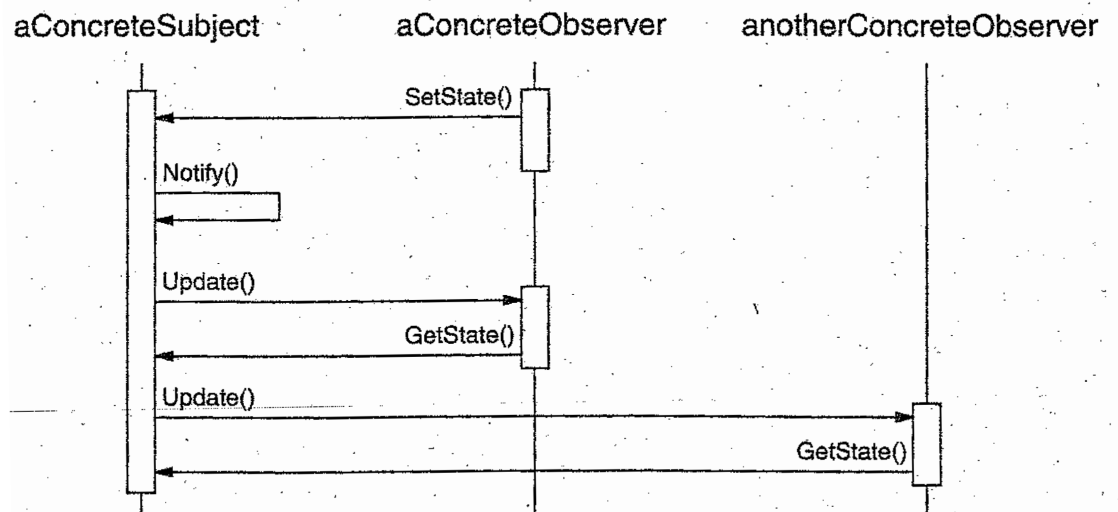
1. The observable maintains some information that is of interest to the observer(s), and is responsible to furnish that information to it/them upon request.

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. This sequence diagram from the "Gang of Four" book shows how the pattern works:
\



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: 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

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

A. The normal way to add an additional feature to a class is to use inheritance. But sometimes we want to add features to individual objects, rather than to entire classes to avoid proliferating classes. The Decorator (Wrapper) pattern is often a good way to do this - and is a pattern that is used in a wide variety of places.

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 or extend the source code for the class itself. We put the added functionality into a wrapper, that also forwards original requests to the object within.

C. To give an example of what this is and how it is used, we will use an example from the structure of the `java.io` package. We will discuss more examples when we cover Java input output at the end of the course, but for now we will use one specific example to illustrate this.

1. The `java.io` package includes a number of classes that support writing to a number of different places - all of which are derived from a base class `OutputStream`.

   a) There is a class `FileOutputStream` that supports writing to a file.

   b) There is a class `PipedOutputStream` which supports writing to another process running on the same system.

   c) Network connections furnish a system-dependent subclasss of `OutputStream` that supports writing to another system over the network.

   d) There is a class `ByteArrayOutputStream` which supports "writing" to an array of bytes.

2. All kinds of `OutputStream` furnish the same basic capability - the ability to write an individual byte to the appropriate destination.

3. But often we want to do more - e.g.

   a) Sometimes we want to write the binary representation of a primitive type as a sequence of bytes - e.g. an `int` can be written as a series of 4 bytes.

   b) Sometimes we want to write the text equivalent of a primitive type - e.g. the decimal representation of an `int` can be written as a sequence of characters, with the number needed being dependent on the value - e.g. 1 is written as one character, while 1234 requires 4.

4. One way to do this would be to create two subclasses of each of the basic types of `OutputStream` having the desired capability.

   For example, a subclass that of `FileOutputStream` allows writing the binary format of a primitive type to a file, and another subclass that allows writing the textual representation of a primitive type to a `FileOutputStream`.

   The same would have to be done for each of the other subclasses of `FileOutputStream`. So the end result would be lots of additional classes.

5. The Java io package avoids this proliferation of classes by defining just two more classes which can be wrapped around any kind of `OutputStream.`

   a) A `DataOutputStream` can be wrapped around any kind of `OutputStream` to support converting a primitive type to a sequence of bytes, each of which is then written to the inner output stream (So when writing an `int`, four bytes are written to the wrapped stream.).

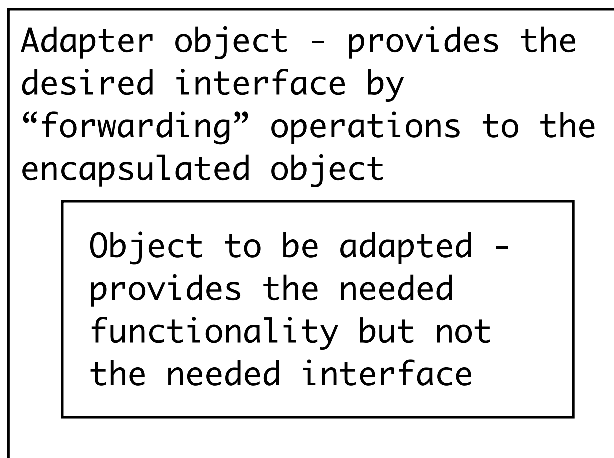
It has one constructor which takes a single parameter that can be any type of `OutputStream`.

b) In similar fashion, a `PrintStream` can be wrapped around any kind of `OutputStream` to support converting a the textual representation of primitive type to a sequence of characters, bytes, each of which is then written to the inner output stream (So when writing an `int`, the number of characters written to the wrapped stream depends on the value).

c) Both wrappers also support the basic method for writing a byte to an `OutputStream` which is simply passed through to the wrapped stream.

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.



PROJECT

B. We will illustrate this shortly

IX. Applying Design Patterns to the Library Project

A. Several of the patterns we have discussed are used in the library project. Which ones - and where used?

   ASK

   1. Singleton is used for `LibraryDatabase` and several others.

   2. Abstraction-Occurrence is used for the relation between Items and Copies

   3. Iterator will be used for producing reports and for accessing all the copies that a given patron has checked out.

B. One pattern will not be used in the library project as assigned, but could be used if you wanted to incorporate an undo facility. Which one?

   ASK

   Command

C. One pattern we have already discussed is actually used in the library project skeleton code, though you're probably not aware of it yet. To see the issue, consider the following demo:

   1. DEMO Version Design Patterns version of Library project, **with `addObserver()` code in `CollectionPane` commented out.**

      a) Need to add a book, so go to CollectionsPane
      b) Note Add Book button initially grayed-out
      c) Login as manager, button still grayed-out because code to enable manager buttons is in code executed when pane is shown
      d) Need to leave pane and come back to make button enabled

e) Now log out as manager - button still enabled until you leave pane and return

2. Two problems result from this:

   Inconvenience
   Possible security issue - non-manager can sometimes do manager functions

3. What pattern could we use to address these issues?

   ASK

   Observer

4. This has actually been incorporated to code - LoginOutUseCase has been made Observable, and PatronPane and CollectionPane have been made Observers of it so button enabled state is updated when manager login state changes.

   a) Remove comment-out and demo how it now works
   b) Show update code in CollectionPane

D. Another issue has been addressed using the Observer pattern and two more we haven't yet discussed

   DEMO display of patrons list in Patrons Pane

   Note how list is updated in proper order when adding or deleting a patron

X. **Putting Several Together - an Illustration of Patterns Use**.

A. A number of years ago, a student asked me in lab how co build an interface of this sort. Unfortunately, it actually turns out that this is much more easily said than done! Here's why.

1. What if the list changes? (E.g. if we add or delete a patron?) We would like the displayed list to be updated instantly. How might we do this?

   ASK

2. Here, we could use the Observer pattern - i.e.the list's model is made an observer of the map - so that whenever the map is changed, the displayed list is updated.

3. Unfortunately, this is not easily possible, because a `TreeMap` is not a subclass of `Observable`! Instead, it is a subclass of another class called `AbstractMap`. Since Java does not allow multiple inheritance, we would have to redo the whole inheritance structure to make it an `Observable` - as well as modifying the source code in many places to notify observers when an operation is done that changes the content.

   There is a pattern we have discussed that presents a way to handle this without modifying the source code that is part of the Java standard libraries (which we couldn't do and wouldn't want to do in any case!) What is the pattern we are looking for?

   ASK

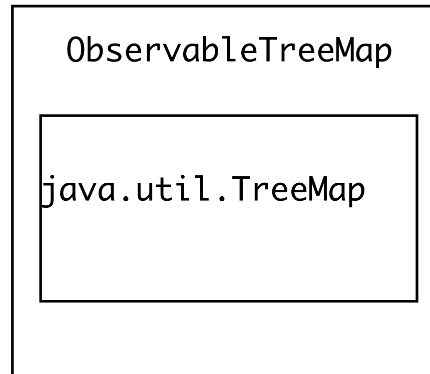
4. We can make use of the Wrapper pattern. We wrap a `TreeMap` in an observable class that:

   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.

d) The result looks like this:



PROJECT

B. Unfortunately, we're not there yet.

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.

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

2. However, maps do <u>not</u> provide the ability to access items by position, but only by key. And the internal structure of a `TreeMap` does not make doing this feasible - even if we could modify the code, which we can't!.

3. What pattern that we have discussed would be appropriate here?

   ASK

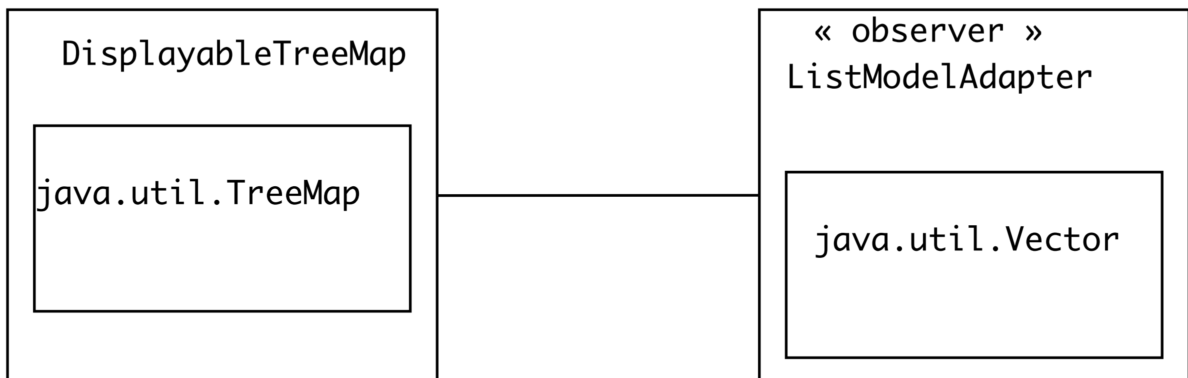
C. We can begin to solve the problem by creating an <u>adapter</u> that allows a `TreeMap` to be used as the `ListModel` for a `JList`. To do this, it can

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. We then get the following:



PROJECT

E. This has been built in to the starter code for your project, and will be used for Iteration 2. Doing this will require you to change the way the collections of patrons and items are created in the LibraryDatabase - so follow instructions in the implementation notes carefully!