COMP 110/401

Prasun Dewan[1]

Model-View-Controller (MVC) and Observer

How to break up a program into a set of interacting objects is challenging problem even for advanced programmers. Just as there are common types such as collections that are used in a variety of applications, there are also common object decomposition approaches, called design patterns, that are used in many applications. Two important patterns we will see here are MVC and Observer. MVC has been designed for interactive applications, and uses Observer as a sub-pattern, which is designed for a larger set of applications that are often also termed as "publish-subscribe" applications. We will see how these patterns support both reusability and multiple, concurrent, consistent user interfaces, possibly on distributed devices. Thus logical separation of components can also lead to physical separation on different computers.

Separation of Concerns

Before we look at different ways to decompose a program into classes (and associated interfaces), it is important to identify some general principles behind such decomposition. We know that decomposition leads to better program reusability and understandability. However, these benefits come from "good" or the "right amount of" decomposition. Intuitively, it is does not seem right or even possible to take each statement or even a method in an undecomposed program and put it in a separate class. Similarly, it does not seem right to create a monolithic single-class program. So what is the litmus test for deciding if two pieces of functionality, A and B, should be different classes?

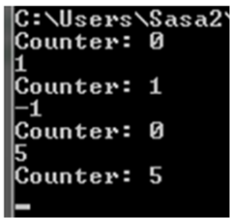
One answer may be that if A is independent of B, then they should be in separate classes. We can say A and B are independent if A is not needed to implement B, and vice versa. This principle tells us that lines and strings should be implemented in separate classes. But it does not tell us that a Cartesian plane and line should be implemented by separate classes, as a Cartesian plane consists of two perpendicular lines.

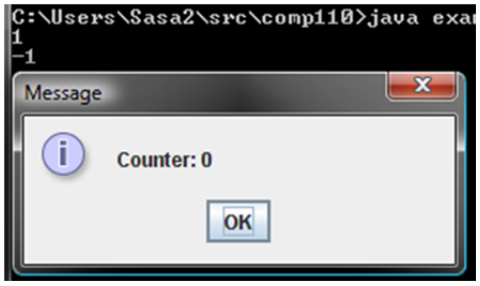
A more general principle, called separation of concerns, is that if A can exist without B, then put A and B in two different classes. We will say that A can exist without B if it is possible to write a "useful" program that consists of A but not B. The relationship is not commutative. It may not be possible for B to co-exist without A. For example, a line can co-exist without a Cartesian plane but not vice versa.

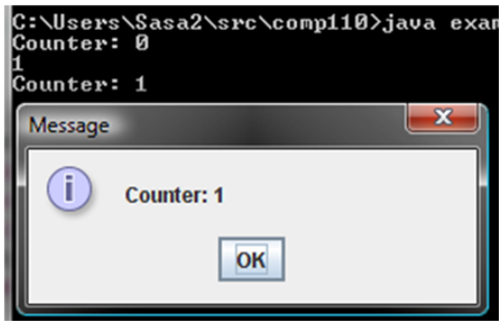
We can use this principle to decompose existing classes. Suppose an existing class C^1 implements two pieces of functionality, A and B. If we can imagine another useful class C^2 that consists of A and B', then we should put A, B and B' into separate classes, and recursively apply this principle to these three component classes.

Counter

To illustrate this principle more concretely and in detail let us take a simple example – that of a counter. The counter provides a (programming) interface that allows addition of an arbitrary positive/negative value to an interface. Figures ??? show three different user interfaces to manipulate the counter. Each user interface allows input of a value to add to the counter and displays the updated value to the user. The console-based user-interface of Figure ??? uses the console window for both input and output. First, it displays the initial value of the counter, and then it allows the user to enter the (positive or negative) values to be added to the counter, and after the input value, it shows the current value of the counter. The mixed-based user interface of Figure ??? retains the input mechanism to change the counter but uses an alternative way to display the counter value. Instead of printing it in the console window, it creates a new "message" window on the screen that shows the value. The multiple-user interface of Figure ??? shows that it is possible to combine elements of the other two user interfaces. It also retains console-based input but displays the counter in both the console window and the message window.







To implement the pure console-based interface of Figure ???, we can write the following code:

```
public class MonolithicConsoleUI {
        static int counter = 0;
        static Scanner scanner = new Scanner(System.in);
        public static void main(String[] args) {
                while (true) {
                        System.out.println("Counter: " + counter);
                        int nextInput = scanner.nextInt();
                        if (nextInput == 0) return;
                                counter += nextInput;
            }
      }
}
```

Similarly, to input the message-based user-interface, we can implement the following code:

```
public class MonolithicMixedUI {
      static Scanner scanner = new Scanner(System.in);
      public static void main(String[] args) {
            int counter = 0;
            while (true) {
                  JOptionPane.showMessageDialog(null, "Counter: " + counter);
                  int nextInput = scanner.nextInt();
                  if (nextInput == 0)
                        break;
                  counter += nextInput;
            }
      }
}
```

This code is identical to the previous one except that, instead of appending to the console, it uses the **JOptionPane** class of the Swing toolkit to display the value in a new message window.

Finally, to implement the multiple user-interface version, we can write code that combines the output of the two previous classes.

How can we remove the code duplication in these three classes?

Combining Classes and Method Decomposition

We could create a single class that has code for both console-based and message-based output. A main argument could specify which version of the user-interface is needed. This value would be used by the main method to determine which output schemes would be used. In this example, it is difficult to see how we can create multiple methods. But in general, we can use method decomposition to separate the code for different kinds of nput and output in different methods.

This kind of approach was encouraged prior to the concept of class-based modularization. The problem here is that a single class is doing multiple things – in this example, input, multiple kinds of output, and counter manipulation. The same class has to be changed to add a new output style (such as the multiple user interface of Fugure???), change the input prompts, or modify the counter operations. This means a programmer assigned to change one of these aspects of the class might have to examine other aspects

also and cooperate with programmer working on other aspects of the class, possibly concurrently. These problems existed in the two individual classes also – input, output, and counter manipulation was combined in each of the two classes. By combining the three ain classes, we exacerbate these problems, even if we use method decomposition. This is the reason for the separation of concerns principle, which suggests class rather than method decomposition in such situations.

Inheritance

Perhaps inheritance is the solution, then, which does provide class decomposition. We can write the code to output the counter value in a separate method that can be overridden by a subclass. This leads to the question: should the console-based user-interface be a subclass of the mixed user-interface, or vice versa. What happens when we add he multi-user interface version? This question shows that there isn't fundamental IS-A relationship between the three applications, and thus, inheritance is the wrong answer.

IS-A vs. HAS-A

Inheritance is not the only relationship among classes. Before we studied inheritance, we did write multi-class programs in which instances of a class referred to instances of other classes via instance variables and/or properties. For example, an instance of a CartesianPlane has references to instance of two Line objects. Such references create a different relationship among types, called HAS-A, which is often a competitor of IS-A, though both relationships can exist between a pair of classes. A type T^1 HAS-A T^2 if it has a property or (instance or local) variable of type T^2.

When we study delegation, we will better understand the relationship between IS-A and HAS-A.

Models and Interactors

Thus, we must look for HAS-A relationships among the classes into which we decompose our two programs. Each of the two programs can be decomposed into two functionalities – (a) counter semantics: storage and manipulation of an integer counter and a (b) counter user-Interface: user-interface to display and change the counter. In the three programs, the counter functionality is identical, and the counter user-interface is different. Thus, the separation of concerns principle says that each user interface and the counter semantics should be in a separate class.

Let us start with the class of the counter semantics, as the user-interface classes depend on it. This class mist allow an integer to be incremented or decremented by an arbitrary value and provides a method to access the current counter value. Here is an implementation that meets these requirements:

```
public class ACounter implements Counter {
      int counter = 0;
      public void add (int amount) {
            counter += amount;
      }
      public int getValue() {
```

```
            return counter;
      }
}
```

As required, the class has no user-interface code. In fact, it is oblivious to even the fact that a user-interface could be attached to it. We will refer to any class that can be attached to a user-interface but is oblivious to the user-interface as a *model*.

We can now define a class that uses this class and its associated interface to implement the console user-interface.

```
public class AConsoleUIInteractor implements CounterInteractor {
    static Scanner scanner = new Scanner(System.in);
      public void edit(Counter counter) {
            while (true) {
                  System.out.println("Counter: " + counter.getValue());
                  int nextInput = scanner.nextInt();
                  if (nextInput == 0)
                        return;
                  counter.add(nextInput);
            }
      }
}
```

This class accepts an instance Counter as a parameter, displays its initial value to the user, and then executes a loop that calls the add method of the counter with each input value and then displays the new counter. This class does not know how the counter is implemented but is aware of the methods provided by it. We will refer to any class that implements the user-interface of a model without knowing its implementation as an *interactor*.

Now that we have these two classes, we need a third class that connect them. That task is done by a main class:

```
public class InteractorBasedConsoleUI {
      public static void main(String args[]) {
            (new AConsoleUIInteractor()).edit(new ACounter());
      }
}
```

The main class instantiates the model and interactor, and connects them together by informing the interactor about the model by passing the model to the interactor edit() method,

(Why not make the interactor class create the model?)

We can similarly write an interactor for the mixed user-interface, AMixedUIInteractor,

```
public class AMixedUIInteractor implements CounterInteractor {
    static Scanner scanner = new Scanner(System.in);
      public void edit(Counter counter) {
            while (true) {
                  JOptionPane.showMessageDialog(null,
                              "Counter: " + counter.getValue());
                  int nextInput = Console.readInt();
```

```
                    if (nextInput == 0)
                        return;
                    counter.add(nextInput);
                }
            }
}
```

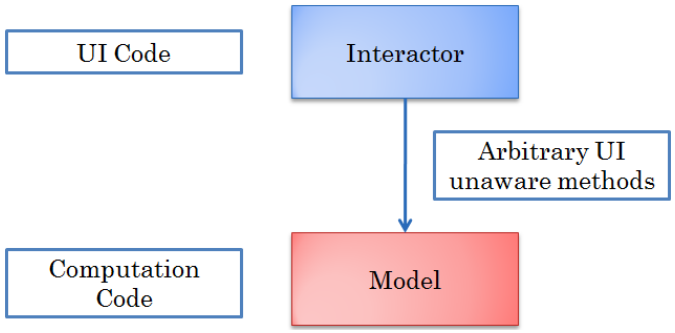
Both interactors implement the same interface, which seems intuitively right as they provide the same abstract functionality – interactive manipulation of a counter. As in the case of the addElement() on AStrngHistory and AStringSet, the behavior of the edit() method in the two cases is different.

Similarly, we can write an interactor class for the multi-user interface application.

By creating a model class, three interactor classes, and three main classes, we have met, to some degree, the separation of concerns principle. We will see later how we can meet it to a higher degree by performing even more decomposition. Before we do that, let us reflect on this exercise.

UI-Computation Separation

The class decomposition was not difficult once we knew the criterion to apply – keep the semantics and user-interface code of an interactive application separate. This principle is a special case of the separation of concerns principle in which the separated concerns are semantics and user-interface. This principle leads to a general pattern for decomposing code, shown in Figure ???. In this pattern, the interactor and model implement user-interface and semantics/computation code and the interactor can invoke arbitrary model methods, which are completely unaware of the user-interface code.



This pattern can be applied to any interactive application, though what is computation/semantics code and what is user-interface code is sometimes tricky to determine. It is often answered by taking two user-interfaces for the same application and abstracting what is common in them. For example, in the case of PowePoint, what is common in the slider sorter, normal, and outline view can be called the PowerPoint model; and in the case of Facebook, what is common in the desktop and mobile user-

interfaces can be called the Facebook model. Once we decide on what the model is, the Interactor is easy to define. It is a translator between model methods and application-independent I/O. It calls model methods in response to application-independent input such as scanInt() and reacts to changes in the model by calling application-independent output such as println(). Thus, the mature of the interactor depends on the underlying I/O library. A corollary of this definition is that a model should not contain any code that interacts directly with an I/O library.

Thus, what we have learnt from the counter exercise can be applied to any interactive application. In other words, given any interactive application, we have a pattern for dividing it into at least three classes, a model class, an interactor class, and a main class that puts them together. The idea of ObjectEditor is based on this model-pattern. ObjectEditor acts as an application-independent interactor to create the user-interface of any model.

This pattern has used beyond allowing reuse of code among different user-interfaces of a model. The same model can be attached to multiple interactors simultaneously. Moreover, the interactors and model can be on separate computers. For example, in the Facebook case, one interactor can be on mobile device and another on our laptop, and they can be communicating with a model on a Facebook server.

Design Patterns

The term pattern is used in several contexts in computer science. A pattern some recurring them in programs. In CS-1, you might have seen two kinds of loops – those controlled by a counter whose value is known before the loop is entered, and those controlled by some event that occurs while the loop is executing. These two are examples of loop patterns. When we studies Beans, shapes, and collections, we saw conventions that could be used to determine the logical components of structured objects. These conventions are examples of method header or signature patterns, as they define names of methods, their return types, their parameters, and the order of the parameters.

The interactor pattern is an example of a design pattern – a recurring theme for meeting the separation of concerns principle. A design patterns involves two or more kinds of classes (and associated interfaces), and describes ways in which these classes communicate with each other, that is, access each other's members. It is associated with a context in which it is applied. It is more general than a specific class or interface and, in fact, applied to an infinite number of classes and interfaces. In the interactor pattern, the context is interactive applications, the class kinds are UI-dependent interactor and UI-independent model, and the communication involves the interactor calling arbitrary (UI-unaware) methods in the model. One can imagine an infinite number of model classes and for each model, an infinite number of interactor classes.

The idea of design patterns in computer science was invented by a group led by Ralph Johnson. It was inspired by the notion of architectural patterns invented by Christopher Plummer to capture commonalities in different kinds of building architectures. Apparently, referring to a catalog of architectural patterns never caught on. Referring to a catalog of (software) design patterns to design

software is, on the other hand, common today. The first and still most popular such catalog is the book: Design Patterns, elements of reusable object-oriented software, by Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides.

Purists distinguish between three kinds of recurring themes in object-oriented software – architectures, design patterns, and frameworks. The differences between them are so subtle that we will refer to all three as design patterns.

Views and Controllers

The model-interactor pattern allows sharing of model code in the various user-interfaces. However, it does not provide a way for sharing code among the various user interfaces. In our example, all three user interfaces provide the same way to input the counter increment. It should be possible to share its implementation between the implementation of the mixed and console-based user interface. Moreover, the multi user interface combines the mechanisms of displaying output in the other two approaches. There should be a way to share the implementation of these mechanisms.

The above discussion suggests that we should recursively apply the separation of concerns principle to an interactor class to separate its input and output functionality into different classes. The (Smalltalk) MVC design pattern does exactly this. It is an extension of the interactor pattern in which the input and output are performed by classes called the controller and view, respectively. When the user inputs a new command the controller calls some write method in the model such as **`add`** in our example. Conversely, to display a result, the view calls a read method such as **`getValue`** in the counter example.

Thus, given a model and an application-independent I/O library, we can define a controller and view. The controller translates application-independent input provided by the library into calls to write methods of the model. A view reacts to changes in the model by calling application-independent output provided by the library. A corollary of these definitions is that a view (controller) should not contain any code that interacts directly with the input (output) primitives in the library,

Let us see how we can use this pattern to configure the three applications.

Console-based UI: We keep the model from the interactor-based implementation, and replace the console interactor with a console controller and console view.

Mixed-UI: We reuse the model and console controller and create a new message view to display JOption messages.

Multiple User Interfaces: We do not create any new model, view, or controller. We reuse the model, console controller, console view, and message view. In this application, the model is connected to both views and the console controller.

There need not be a one-to-one correspondence between a controller and view. The whole point of MVC is that these two kinds of components are independent of each other. In the multiple user-interface application, a single controller is associated with two views (through the model, of course). It is

more usual for a view to be associated with multiple controllers as often the same model operation is invoked my different kinds of inputs. For example, an email may be sent by pressing a button, selecting from a menu, or entering a key shortcut. The methods for handling these three kinds of input are independent of each other, and so, by the separation of concerns principle, should be in separate controller classes sharing a single model and perhaps, also a single view.

Are the model, views, and controllers different classes or instances? In general, a program may simultaneously create multiple models with the same behavior but different state. For example, a program may create multiple counters. This implies that the model must be an instance. Different models may simultaneously have independent controllers and views with the same behavior. More interesting, a particular model may simultaneously have multiple independent controllers and views with the same behavior. Consider a game program that allows multiple users on different computers to independently interact with an object using the same kind of user interface. Or a distributed presentation that can be viewed simultaneously by distributed users. This implies that, in general, controllers and views must also be instances. For the same reason, interactors should also be instances.

Synchronization and Observer Pattern

Two questions we need to answer to complete the description of the interactor and MVC design patterns have to do with giving feedback to a user command. In the interactor design pattern, when one interactor changes a model, how do other interactors update their displays? Similarly, in the MVC pattern, when a controller processes an input command, how do all the views know they should update their displays? For instance, if the console controller is used to increment a counter, how do all the views know that they should display the new counter value? The answers to both questions are the same, so we will consider only the MVC feedback question.

These questions were trivial to answer in the monolithic implementation as a single class had all of the input, output, and computation code. Inputting code in the class could call all of the output code. Now we need coordination among the various objects in our program, which could even be on separate computers.

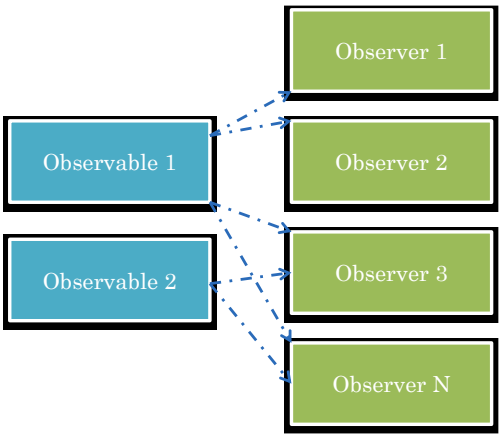
We can make the controller notify all the other views after it has invoked a write method in a model. This means that each controller must be aware of every view, which means the pieces of code that create views and controllers must coordinate with each other, making them complex. Moreover, this approach does not allow views to be updated autonomously, that is, without user input. Imagine a counter being updated every second or a temperature sensor object being updated when a new thermometer reading is sensed. Despite these two shortcomings, this is a practical approach in a Web context, as we see below.

Fortunately, in many situations, a simpler, centralized approach, shown in Figure ???, can be taken that addresses both problems. Whenever the model history is changed, autonomously or by a controller, it informs all of its views that it has changed. Thus, our write methods become more complicated - they not only change the state of the object but also notify all of its views.

Thus, now the model is aware of its views. One of the advantages of the previous version of the pattern was that model was completely independent of its user-interface components – both views and controllers. As a result, these components could change without requiring any changes to the model. Now that the model is aware of its views, would we ever have to change it if we decide to create some new kind of view for it?

Fortunately, the methods invoked by an object on its views can be independent of the behavior of the view. All the object has to do is notify its views that it has changed. This information is independent of the kind of display created by the view. In fact, it can be sent not only to views of an object but also other objects interested in monitoring changes to it, as we see below.

A notifying object is called an observable and a notified object is called its observer. The dashed lines from the models to its views indicate that the awareness in it of its observers is *notification awareness*: it knows which observers needs to be notified about changes to it but does not know any aspect of their behavior. Even though the observable/observer notion appeared first with the context of the larger MVC pattern, it is now recognized as an independent pattern. Figure ???? shows the pattern independent of MVC.



The figure shows that, an observable can be attached to multiple observers. The multiple user-interface application motivates this configuration, as a model is attached to both the console and message views, both of which are observers. Does it make sense to attach multiple observables to one observer, as the figure suggests? In the context of MVC, this would mean a single view showing multiple models. An example is a single battle-simulation view showing updates of multiple tank and plane models. In your project, you will create single view that shows multiple avatar objects.

How does an observable know which observers to notify? It defines a special registration method, which is called to associate it with an observer. The method keeps track of all of the registered observers in a collection. It also sends a notification to the newly registered observer so that the latter knows it has been registered. If the observer is a view, it would use the notification to display the state of the model when the registration took place. Usually, an observable also defines a method to unregister observables who wish to not observer all changes to the observable.

In the observer pattern, somehow observers have to get registered with their observables. In MVC, this means that somehow views have to get registered with their models. Also in MVC, a controller needs to know about its models so it can invoke write methods on them. Design patters are silent on how the connection among objects are made. In other words, these are application-specific decisions. A controller-model connection could be made by the controller or some other external code such as a main class. The external code can call a setter method in the controller or pass the model reference in a constructor. Similarly, an observer-observable connection can be made by the observer or some external program.

When an observer can receive notifications from multiple observables calling the same notification method, the notification method identifies the observable through an argument. This argument is then used to call read methods in the observable. Notification methods can also have arguments describing the kind of change, so read methods need not be called, saving a roundtrip message in distributed computing. If a notification method does not identify the observable, then somehow the observer has to get a reference to the unique observable that calls the method. Again, the MVC design pattern is silent about how this reference is made.

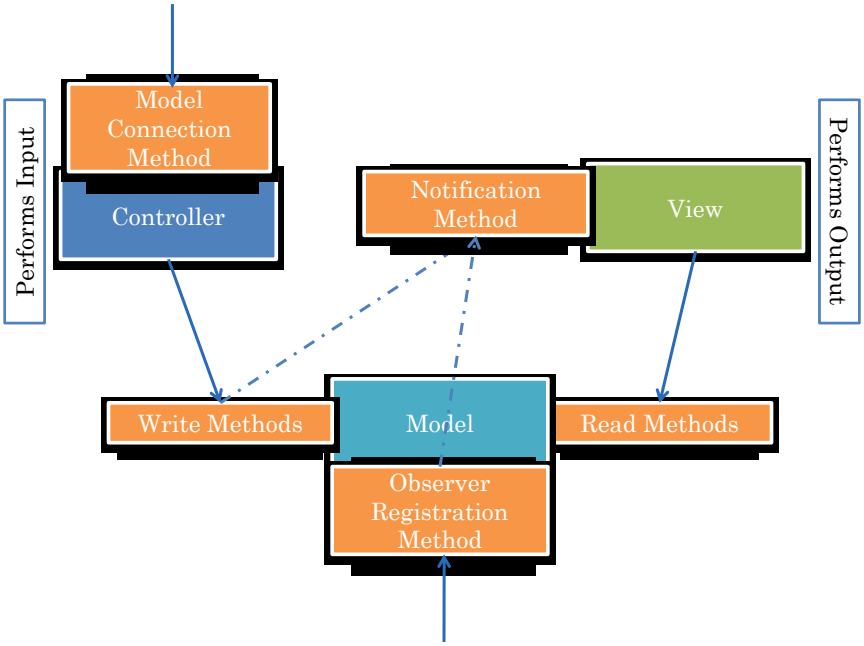


Figure ??? shows the complete MVC model, leaving unspecified who calls the model connection method/constructor in the controller and the observer registration method in the model.

Not just objects, but also humans regularly use this idea of notification in the observer pattern. For instance, students in a class are often notified when the web page for the class is changed. Consider the various steps that take place to make such interaction possible:

- The professor for a course provides a way to add and remove students from a mailing list.
- Students may directly send the professors add and remove requests or some administrator may do so on their behalf as they add/drop the course.

- When a professor modifies the information linked from the web page, he/she sends mail to the students in the mailing list. On the other hand, when he simply reads the information, he does not send any notification.
- The notification tells the student which professor sent it so that they know which web page to look at.
- The students access the page to read the modified information.

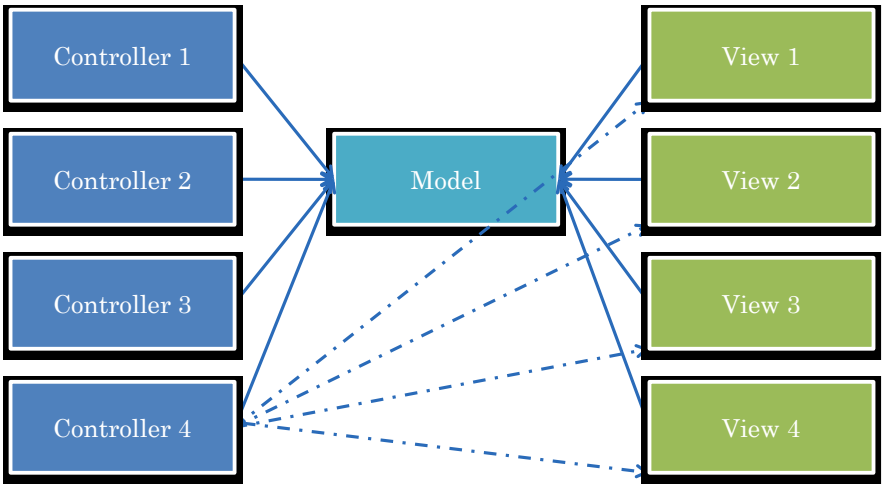
The observable/observer pattern has analogous steps:

- An observable provides methods to add and remove observers from an observer list.
- Observers may directly invoke these methods or some other object, such as a man method, may do so on their behalf.
- When an observable modifies its state, that is, invokes a write method, it calls a method in each observer to notify it about the change. On the other hand, when it executes a read method, it does not call this notification method.
- The call to the notification method can identify the observable doing the notification. The reason is that an object may observe more than one observable. For example, a battle simulation user interface may observe the positions of multiple tanks and planes.
- Each notified observer calls read methods in the observable to access the modified state.

Web MVC = (SmallTalk) MVC – Observer

The MVC pattern described here is the one developed in Smalltalk-80. The number 80 indicates the year in which this version of Smalltalk was released. So this is a very old idea, though it did not get much traction until the 1990s when researches in collaborative computing found that it could be used to create multiple views for the same model on the computers of different users simultaneously manipulating the model.

Recently, the term MVC has been popularized by Web-based interactive applications. In these applications, the models reside in the server and the views and models reside in browsers. One of the fundamental properties of Web-based applications is that the server cannot initiate communication to a browser, it can only respond to communication initiated by the browser. This means that the model cannot send notifications to the views. Therefore, in Web-based MVC, a controller in a browser, after invoking a model method in the server, notifies all the views in the browser, as shown in Figure ???.



This is an alternative we considered and dismissed earlier as it requires each controller to know about all the views. In Web-based MVC, a single controller exists, and all views in a browser register with it, instead of with a model. As described above, there is no way for an input in one browser to trigger a view update in another browser. So this "modern" version of MVC is arguably inferior to the one developed by 1980! Those who have grown up with the Web and not seen the original MVC use the Web-based MVC in contexts in which models can push changes to views, without realizing the inherent limitations of doing so.

The inability to push changes from a Web server to a browser prevents several applications that we see today in the so-called Web 2.0 applications. There have been several attempts to simulate/implement such communication in the Web. As a result, there has been a movement to use these solutions to go back to the original MVC version.

This course is not about the Web, so we will not discuss Web-based MVC in more depth. It has been presented here only to show that there is more than one MVC pattern.

Observer without MVC

As we saw above, the observer pattern can be used without MVC. To illustrate this point, here are some examples: A counter could notify an object keeping a log of all changes to its counter. A temperature sensor object could notify an object that alarms users if the temperature goes beyond some threshold value. The Java toolkit allows an object representing a button or text field to also send notifications to objects that wish to react to button presses and text field edits, respectively. The interaction in Figure ??? shows an example of a non-view observer of our counter object. A "rocket launcher" observer "launches" a rocket by printing a message when the counter value goes to zero Figure ???. The fact that its message is printed after the console views prints the zero value of the counter indicates that it was notified after the console view.

For instance, a counter could notify an object keeping a log of all changes to its counter. A temperature sensor object could notify an object that alarms users if the temperature goes beyond some threshold

value. The Java toolkit allows an object representing a button or text field to also send notifications to objects that wish to react to button presses and text field edits, respectively. The following figure shows an example of a non-view observer of our counter object. A “rocket launcher” observer “launches” a rocket by printing a message when the counter value goes to zero (Figure ???):

```
Counter: 5
-1
Counter: 4
-1
Counter: 3
-1
Counter: 2
-1
Counter: 1
-1
Counter: 0
LIFT OFF!!!
```

The fact that its message is printed after the console views prints the zero value of the counter indicates that it was notified after the console view.

The notion of observers is pervasive in object-oriented software. Different implementations of this idea differ in the syntax they use (e.g. the term Listener instead of Observer) and the kind of information an observer is passed about the changed object.

Example MVC Implementation

We are ready to see how the observer-based MVC pattern can be implemented in our sample application. Let us first define the interfaces of an observable counter:

Observable/Observer Interfaces

```
	public interface ObservableCounter {
	public void add(int amount);
	public int getValue();
	public void addObserver(CounterObserver observer);
	public void removeObserver(CounterObserver observer);
}
```

The interface defines the same methods as the Counter interface, for reading and writing the counter, and methods to add and remove counter observers, which are defined by the following interface:

```
public interface CounterObserver {
	public void update(ObservableCounter counter);
}
```

The update method is the notification method called each time an observable changes. Its argument is the observable that changed.

Observable Implementation

The implementation of the observable interface supports two operations defined by it. The write method, **add**, notifies the observers, as shown below:

```
public class AnObservableCounter implements ObservableCounter {
	int counter = 0;
	ObserverList observers = new AnObserverList();
	public void addObserver(CounterObserver observer) {
		observers.addElement(observer);
		observer.update(this);
	}
	public void removeObserver(CounterObserver observer) {
		observers.removeElement(observer);
	}
	public void add(int amount) {
		counter += amount;
		notifyObservers();
	}
	public int getValue() {
		return counter;
	}
	void notifyObservers() {
	   for (int observerNum = 0; observerNum < observers.size();
 observerNum++)
		observers.elementAt(observerNum).update(this);
	}
}
```

The list of registered observers is kept in the observer history, **observers**. The methods **addObserver** and **removeObserver** simply call existing methods to add and remove elements from the history.

The method **notifyObservers** retrieves each element of **observers**, invoking the **update** method defined by the observer interface.

As we have seen earlier, the keyword **this** in a method refers to the object on which the method is called. Thus, the program fragment

```
	update(this)
```

passes **update** a reference to the observable so that it can invoke getter methods on it to display the counter.

As we see here, the notification method is called not only in the write method but also in the registration method.

Observer Implementations

The console and message views implement the generic interface for observing a counter. The console view implements the update method by simply appending a display of the counter to the console.

```
public class ACounterConsoleView implements CounterObserver {
	public void update(ObservableCounter counter) {
		appendToConsole(counter.getValue());
	}
```

```
	void appendToConsole(int counterValue) {
		System.out.println("Counter: " + counterValue);
	}
}
```

The message view provides a different implementation of the method, creating a message window displaying the counter value:

```
public class ACounterJOptionView implements CounterObserver {
	public void update(ObservableCounter counter) {
		displayMessage(counter.getValue());
	}
	void displayMessage(int counterValue) {
		JOptionPane.showMessageDialog(
null, "Counter: " + counterValue);
	}
}
```

Since it used the Swing class, **JOptionPane**, to create the message view, it is called **ACounterJOptionView**.

In Figure ???, we saw a non-non view "rocket launching" observer of the model. Here is its implementation.

```
public class ARocketLaunchingCounterObserver implements CounterObserver {
	public void update(ObservableCounter counter) {
		if (counter.getValue() == 0)
			launch();
	}
	public void launch() {
		System.out.println("LIFT OFF!!!");
	}
}
```

It takes an action only when the notifying counter goes to zero.

Console Controller

Finally, let us consider the controller object. Its interface is given below:

```
public interface CounterController {
	public void setModel(Counter theCounter);
	public void processInput();
}
```

The first method can be called by an external object, such as the main class, to bind it to a model. The second method starts its input processing loop. Because of the loop, this method should be called after all other actions have been taken to create the model view controller pattern.

The class of the console controller is given below:

```
public class ACounterController implements CounterController {
	ObservableCounter counter;
	Scanner scanner = new Scanner(System.in);
	public void setModel(ObservableCounter theCounter) {
		counter = theCounter;
	}
	public void processInput() {
```

```
            while (true) {
                  int nextInput = scanner.nextInt();
                  if (nextInput == 0) break;
                  counter.add(nextInput);
            }
      }
}
```

After each input, it calls the add method of the counter with the value. It does not have to worry about the output – that is the concern of the views.

Composing MVC

We have the individual pieces of our design patterns. These need to be now connected. We will do so in main methods. Here is an example main method that connects the model to its controller and all of its observables:

```
public class MultipleUIMVCComposerWithRocketLauncher {
      public static void main(String args[]) {
            ObservableCounter model = new AnObservableCounter();
            model.add(5);
            model.addObserver(new ACounterJOptionView());
            model.addObserver(new ACounterConsoleView());
            model.addObserver(new ARocketLaunchingCounterObserver());
            CounterController controller = new ACounterController();
            controller.setModel(model);
            controller.processInput();
      }
}
```

The method creates the multiple user-interface version of the application along with the "rocket launcher". By deleting appropriate lines, we can remove the launcher, the message view, and/or the console view.

Control Flow

To understand how the three kinds of objects, model, view, and controller, interact with each other, assume that the model is bound to a console view, a message view, and the rocket launcher, as in the main above. Let us see what happens when a user enters a new value:

- The controller gets the new value **`int`** and invokes **`add`** on the model.
- The model calls **`notifyObservers`**, which in turn, invokes **`update`** on all the observers in the order in which they were registered.

The console view shows the new value in the console window, the message view creates a message window, and the rocket launcher prints its message if the counter has reached 0.

Change Description Notification

Consider the implementation of our observer interface:

```
public interface CounterObserver {
      public void update(ObservableCounter counter);
```

```
}
```
Here the notification method, **update**, takes a single parameter indicating the source of the notification. If an observer has only one observable during its life time, there is no need to pass the observable to the notification method. Instead, the observer can be informed about the observable in a constructor or in a separate method:

```
public interface CounterObserver {
      public void setObservable(ObservableCounter counter);
      public void update();
}
```

Now imagine a distributed observable/observer scenario, where the two objects are located, say, in the U.S. and China, respectively. The above scheme is inefficient. The observable sends a message to China saying that it has changed. The observer must now send a message back to the US to get the changed value in which it is interested. If the observable knew the kind of change in which an observer was interested, it could have sent the change with the notification. Thus the message back to the U.S. would not have been necessary. The trick is to know in what the observer is interested.

In our example, the model can send the new value of the counter.

```
public interface CounterObserver {
      public void update(ObservableCounter counter, int newCounterVal);
}
```
This, the update method has an extra parameter representing the change.

Java "Application-Independent" Observer Implementation

What if we wish to display our Counter model using ObjectEditor? ObjectEditor can certainly display the counter value. But can it receive notifications? To do so, it would have to implement the CounterObserver interface, which is application-specific and unknown to an application-independent tool such as ObjectEditor.

This limitation motives Observer interfaces that are application independent. There are many such interfaces in Java. The earliest one is java.util.Observer:

```
public interface Observer {
      public void update(Observable o, Object arg);
}
```
It is like the version of the CounterObsever that takes an extra argument describing the change. The main differences are is the (a) change is an arbitrary object rather than an int value, and (b) observable is of type java.util.Observable rather than CounterObservable.

Java Observable is much like CounterObservable. The main difference is that it expects its Observers to be of type Observer rather than CounterObsever:

```
public class java.util.Observable {
      public void addObserver(Observer o) { … };
```

```
        public void notifyObservers() { … };
        public void setChanged() {…};
        public void clearChanged();
        public boolean hasChanged()
        ….
}
```

Thus, by generalizing from integer updates to object updates, we can create an application-independent implementation of the Observer pattern. Even though all design patterns are application-independent, their implementation is usually application-dependent. Observer is one of the few patterns that have an application-independent implementation.

The Java version of this implementation, however, unlike ours, does not define an interface for an observable object. This means that an observable object must be a subclass of this class it is to have observers of the Java type Observer. With single inheritance in Java, this is not always desirable. ObjectEditor does support this interface. If the object passed to the edit() call is an Observable, it listens to updates from it.

One other difference between ours and Java's implementation of the Observer pattern is that notifyObservers() does not call the Observer notification methods unless, since the last time it was called, the setChanged() method was called. The motivation for this extra step is to efficiently support a single notification for a series of zero or more calls to write methods. Each write method calls setChanged(). When the notification method is finally called after zero or more writes, it checks hasChanged() to determine if notifications should be sent. If the object has changed, it sends the notifications and then calls clearChanged(). This efficiency comes at the cost of increasing the chance of making programming mistakes in which the setChanged() method is accidentally not called.

Finally, consider the arg argument to the Observer update method. It is meant to describe the kind of change so that read methods do not have to be called. As it is an instance of Object, any kind of change can be encoded. This creates a problem for an application-independent tool such as ObjectEditor, which does not know how to decode the change. For a single-component object, this is not so much a problem, as it can contain the value of the property. For the counter example, it can contain the integer value of the counter. A tool such as ObjectEditor, which can determine the names and types of the components of an object, then knows how to interpret the value. Multi-component structured-objects pose a problem.

In the next chapter, we will see an alternative, provided by the Java Beans framework, that does not have these limitations of the Java Observer/Observable types.

Summary

The MVC framework defines three kinds of objects: models, views, and controllers, which are connected to each other dynamically. When a user enters a command to manipulate a model, a controller processes it and invokes an appropriate method in the model, which typically changes the state of the model. If the method does changes the model state, it notifies the views and other observers of the model about the change. A view responds to the notification by redisplaying the model.

Exercises

1. What advantages does the interactor pattern approach offer over the monolithic main approach?

2. What advantages does the MVC framework offer over the interactor approach? Do you see any disadvantages?

3. In the MVC pattern, describe the roles of the controller(s), model, and view(s). Classify all of the classes you created for Assignment 9 as a controller, a model, a view, or none of the above.