CST8221 – Java Application Programming

Unit 6 – MVC Design Pattern

Object-Oriented Analysis and Design (OOAD) and Object-Oriented Programming (OOP) are all parts of Object-Oriented Software Development (or Engineering) where the object-oriented methodology (or strategy) is used throughout the development process:

- **Object-Oriented Analysis** is concerned with developing an object-oriented model of the problem or application domain. The identified objects reflect entities and operations that are associated with the problem to be solved.
- **Object-Oriented Design** is concerned with developing an object-oriented model of a software system to implement the identified requirements. The objects in an object-oriented design are related to the solution to the problem that is to be solved. There may be close relationship between some problem objects and some solution objects but the designer inevitably has to add new objects and to transform problem objects to implement the solutions. The object oriented design uses UML (Unified Modeling Language) to describe the design model. In the process of designing the model the designer looks for existing Object-oriented Design Patterns that can be use in the implementation.
- **Object-Oriented Programming** is concerned with realizing a software design model using an appropriate Object-Oriented Language. An appropriate Object-Oriented Language is a language which supports and implements the Object-Oriented Paradigm

Many object-oriented analysis and design methodologies have been proposed (Coad and Yourdon, 1990; Robinson, 1992; Jacobson et al., 1993; Booch 1994; Graham, 1994). A unification of the notations used in these methods has been defined and the Unified Modeling Language (UML) standard has been proposed. An associated Unified Modeling Process (Rational Unified Process) has also been developed (Rumbaugh, 1999).
In life we learn by imitation. Watching carefully, and with a little practice, we learn to distill (separate, extract) the essential parts from their examples. These "essential" parts form a "pattern." For example, a pattern for driving a car is: step on the break to stop, press (not hit -:) the gas pedal to go, turn the steering well to change directions and so on. Can we apply the same ideas to object-oriented programming? The answer is Yes.
A number of software visionaries found that in object-oriented programming there are many repeating situations (problems) and different object-oriented programmers developed similar solution (patterns) to them. These visionaries realized that, by summarizing and publishing descriptions of these patterns, they could provide an important service to the software tribe. Thus, **object-oriented design patterns** were born. Design patterns in object-oriented design play the same role data structures and algorithms play in procedural programming. Each helps you to avoid to "re-inventing the wheel."
The pioneers in the area of design patterns were actually inspired by the architectural design patterns of the architect Christopher Alexander. In his book, *The timeless Way of Building* (1979), Alexander gives a catalog of patterns for designing public and private living spaces. Each pattern in Alexander's catalog, as well as those in the catalog of software patterns, follows a particular format. The pattern first describes the context, a situation that gives rise to a design problem. Then the problem is explained, usually as a set of conflicting forces. Finally the solution shows a configuration and interaction that solves the problem.

Currently, the design patterns are classified into four main categories:

1. ***Creational patterns***, which deal with the process of object creation. Examples: Accessor / Mutator, Singleton, Factory, Abstract Factory, Prototype.
2. ***Structural patterns***, which deal primarily with the static composition and structure of classes and objects in a system. Examples: Decorator, Facade, Adapter, Proxy.
3. ***Behavioral patterns***, which deal primarily with dynamic interaction among classes and objects. Examples: Observer/Observable, Command.
4. **Architectural patterns**, which address the issues of designing a specific part of a system. Example: MVC

The description of each design pattern consists of some or all of the following sections [Gamma E.et al. 1995]:

Pattern name	The essence of the pattern.
Category	Creational, structural, or behavioral.
Intent	A short description of the design issue or problem addressed.
Also known as	Other well known names of the pattern.
Applicability	Situations in which the pattern can be applied.-
Structure	A class or object diagram that depicts the participants of the pattern and the relationships among them.
Participants	A list of classes

A design pattern has four essential elements:

- The **design pattern name**. It identifies the pattern. Naming things help us to think more effectively. Example of well known names: Decorators, Singletons, Factories, View-Model-Controller (VMC).
- The **problem** describes the situation in which the design pattern applies. It helps to understand when the design pattern might be useful.
- The **solution** describes the elements (classes and objects), and the relationships between elements (association, aggregation, inheritance, and instantiation), that solve the problem.
- The **consequences** help the programmer understand the tradeoffs (ramifications) of applying the solution. As usual in design, the choice of a solution involves tradeoffs. The consequences help the developer to decide which alternative fits best the specific solution.

Learning something about the design patterns is important since many of them are used in the Java API (C++ as well - STL). A detailed discussion of the design pattern subject is beyond the scope of this course. Here they are introduced briefly in order to give you better understanding of how Java API classes are designed and how they should be used. You have already used some design patterns when developing Java applications.

The Set/Get Pattern

- **Pattern name:** Set/Get, Mutator/Accessor. **Category**: Creational.
- **Problem:** How can data members (fields) be accessed and manipulated without breaking the encapsulation rules?
- **Solution:** For each of the fields, create a pair of methods using the prefixes **set** (mutator) and **get** (accessor) followed by the name of the field. In some cases only one of the methods is provided. Examples: getRuntime(), setOperand(), getResult() and so on.
- **Consequences:** Data members (fields) can be designated **private** thus enforcing the data hiding (encapsulation). Allows for building unified interfaces. Essential for building Java Beans and implementing RMI (Remote-Method-Invocation). On the negative side, creates larger classes and slows down the execution. Allows for implementing data verification and creating "virtual" fields.

The Decorator Pattern

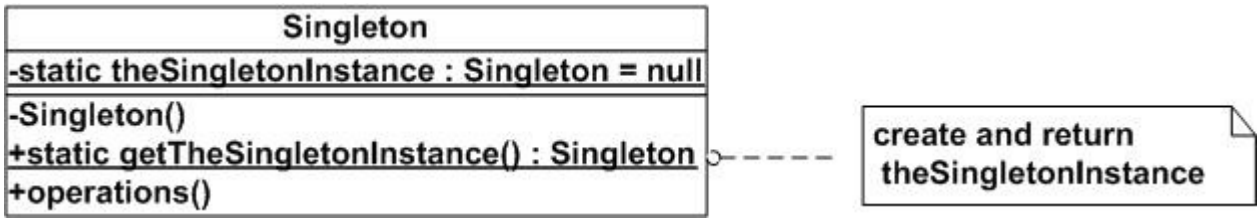
- **Pattern name:** Decorator, Wrapper, or Filter. **Category**: Structural
- **Problem:** How can capabilities of an object be extended at runtime?
- **Solution:** Create an object that contains (that is, decorates or wraps around) the existing objects. The containing object can forward messages (call methods) to the original object, preserving the original object's capabilities. It can also respond to messages not implemented by the original object, extending the original object capabilities.
- **Consequences:** Extending an object's capabilities at run time is more flexible than extending the object' class using inheritance, which must be done in compile time. The decorator pattern allows the programmer to add capability dynamically in small increments. Systems that support the decorator pattern can be very flexible.

Examples:

The Java wrapper classes: *Character*, *Integer*, *Double*, *String* and so on.

The Singleton Pattern

- **Pattern name:** Singleton. **Category**: Creational.
- **Problem:** How can the programmer ensure that a class has only one single, unique instance?
- **Solution:** Forbid access to the constructor (make it private or protected) and implement an accessor that returns a reference to the unique instance.
- **Consequences:** Control over access to the unique instance. Solution generalizes well to more than one (but limited in number) instances.



The Iterator Pattern

- **Pattern name:** Iterator, Enumerator, Cursor. **Category**: Behavioral.
- **Problem:** How can the elements of a collection be accessed sequentially without exposing its implementation?
- **Solution:** Provide a "mediator" ("go-between") object that is aware of the implementation of the collection, but can be used by client objects without exposing the implementation.
- **Consequences:** Support for variation in traversal, eliminate need for traversal interface in the collection (making it simpler), allow multiple traversals.

Example: The Iterator pattern is used in all of the Java collection classes such as ArrayList, Vector, List, HashTable, StringTokenizer and so on. The sequential access is provided by a method nextElement() (or nextToken() in StringTokenizer). The methods hasMoreElements() (hasMoreTokens()) are part of the pattern implementation.

The Model-View-Controller (MVC) Pattern

The Model-View-Controller (MVC) design paradigm is an incredibly useful tool in writing maintainable programs. MVC lets you separate your business logic from your Graphical User Interface (GUI), making it easier to modify either one without affecting the other. Initially, MVC requires a little extra planning and coding, but the long-term benefits are well worth it.

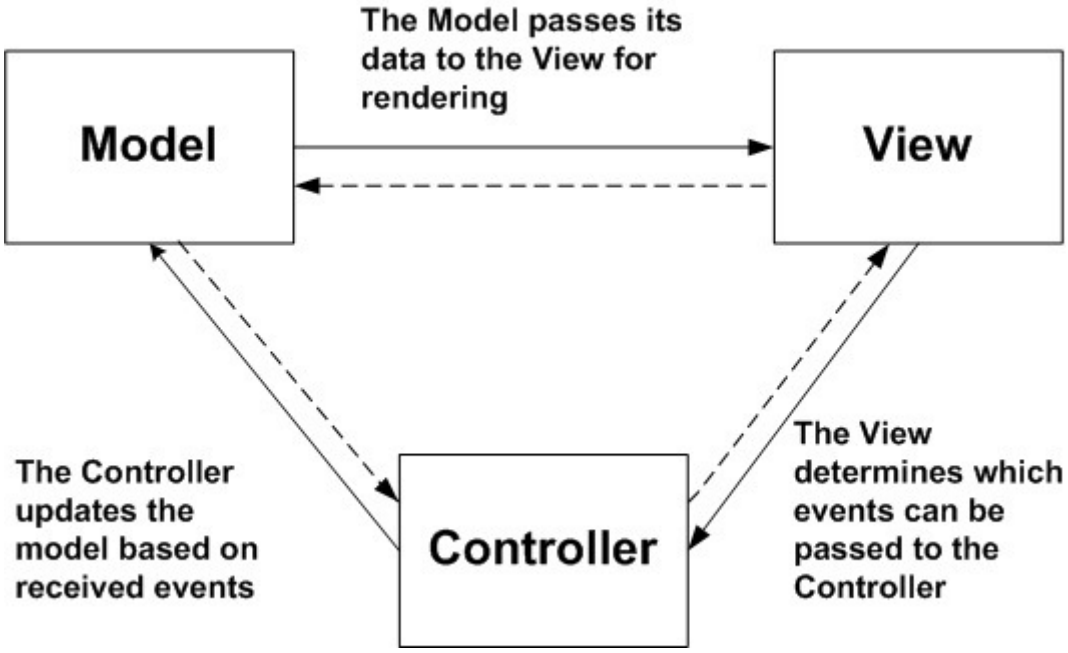
- **Pattern name:** MVC. **Category**: Architectural.
- **Problem:** How can applications with complex user interface (GUI) be created?
- **Solution:** Break the application into three elements. Each of these elements plays a specific role.

  Let us step back and think about the pieces that make up a user interface component such a button, a checkbox, or a text field. Every component has three essential elements:
  - Its ***content***, such the state of the button (pushed or not) or the text in a text field
  - Its ***visual*** appearance – color, size, borders, and so on
  - Its ***behavior*** – reactions to user induced events.

  How to make these three elements to work together and at the same time make them as independent as possible and adhere to the one of the main principles of object oriented programming: Do not make one object responsible for too much? The Model-View-Controller design pattern teaches how to accomplish this. The MVC paradigm was introduced by Smalltalk developers at Xerox PARC (Palo Alto Research Center) in the late 1970's. The basic idea is to split your application into three distinct parts, each of which can be replaced without affecting the others:
  - **The Model:** The model stores the content or in other words it encompasses the data of your application along with the logic that defines how to change or access the data.. It performs the calculations and the data manipulation. The model data always exists independent of the data representation to the user.
  - **The View:** The view determines how the data is represented to the user. This could take the form of a GUI, generated speech, audible tones, printouts, and so on.
  - **The Controller:** The controller reacts to the user actions and updates the model. It informs the view that the data is ready for display. It is possible to combine the view and the controller in one object (Swing uses such combination).
- **Consequences:** Allow for creation of flexible and reusable components. Changes in one component do not lead to changes in another component.

The MVC design pattern specifies precisely how these tree elements interact. The model stores the content and has *no user interface*. For a button, the content is pretty trivial – just a small set of flags which determine the state of the button. For a text field the content is a string object which holds the current text. This is not the same as the view of the content. The controller handles the user-input events such as mouse clicks and keystrokes. It then decides whether to translate these events into changes in the model or the view. The Figure 1 below shows the general interaction between the three elements:

Figure 1. Communication through the model-view-controller architecture



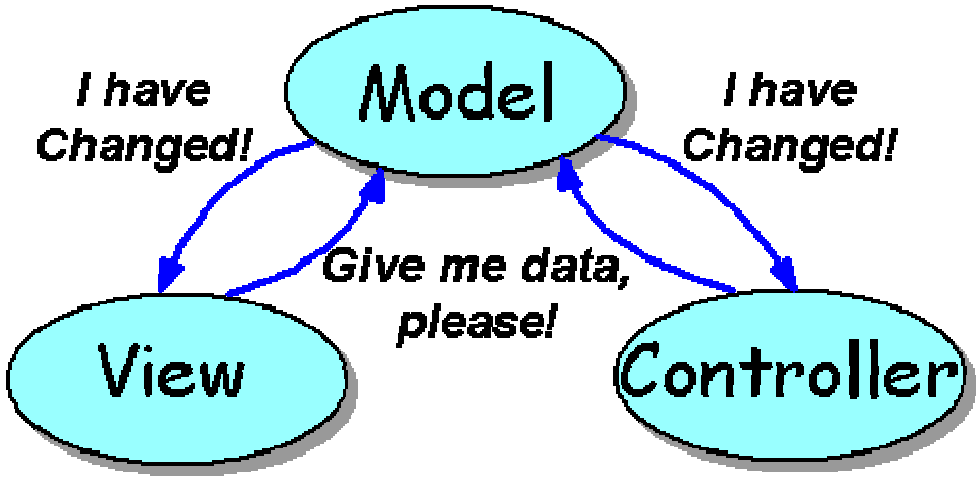
Interaction Between Elements

Let's examine the interaction between these three pieces. As you can see in Figure 1, there are many possible way to implement the interaction between the MVC elements.

In the MVC architecture the user input, the modeling of the external world, and the visual feedback to the user are explicitly separated and handled by three types of object, each specialized for its task. The **View** manages the graphical and/or textual output to the portion of the screen that is allocated to its application. The **Controller** interprets the mouse and keyboard inputs from the user, commanding the model and/or the view to change as appropriate. Finally, the **Model** manages the behavior and data of the application domain, responds to requests for information about its state (usually from the view), and responds to instructions to change state (usually from the controller).

Initially, the view, and possibly the controller, asks the model for its current state. The view may present data to the user, and the controller may check the data to help decide how to handle user interaction.

Figure 2. Model and user interface communication



As seen in Figure 2, the view and controller will typically "listen" for changes to the model. The Java language implements this notification using events. Whenever the model says "I have changed," the view and controller can ask for the new state of the model's data, then update their presentation to the user.

Figure 3. Controller to model communication

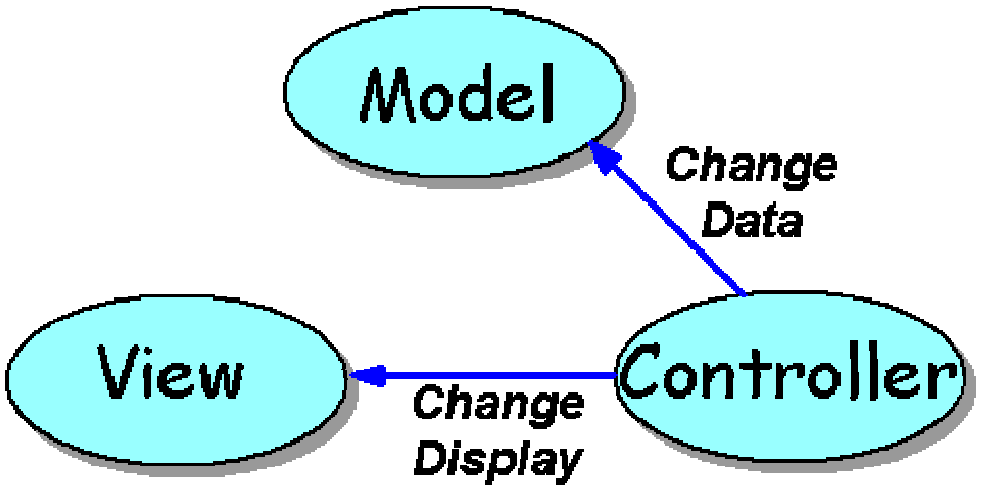
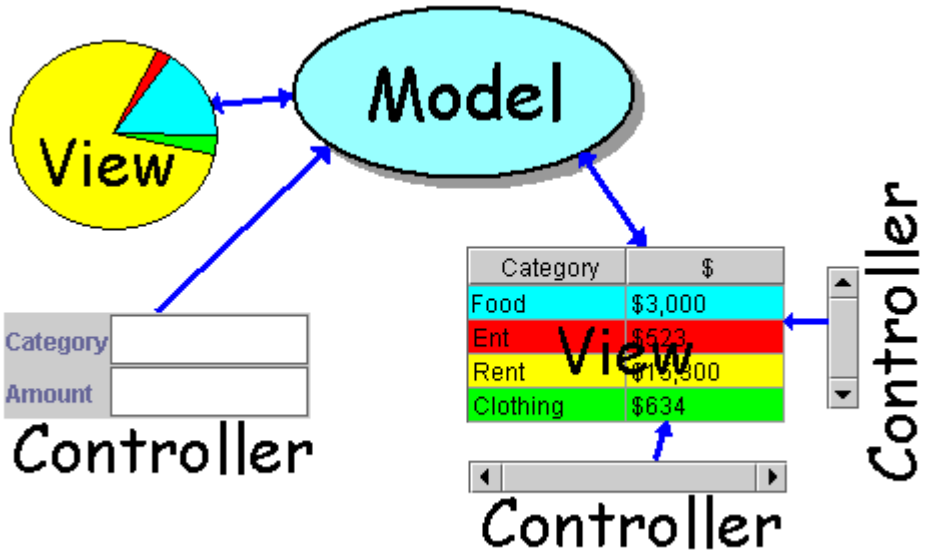


Figure 3 shows user interaction in the application. If the user decides to interact, the controller takes charge. It watches for user input, such as clicking or moving the mouse or pressing keyboard keys. It decides what the interaction means, and asks the model to update its data and/or the view to change the way it displays the data. For example, you could have a view that displays a set of data in a Swing *JList* component. You add a scrollbar as a controller, which directs the view to change which items are displayed. Further, you could add *another* controller, perhaps a Swing *JTextField*, to take user input and ask the model to add the new value to the set. Of course this would cause the model to shout "I have changed!", to which the view responds by asking for the set of data. The above scenario has two controllers. One watches for mouse interaction with a scrollbar, while the other accepts user input to add to the model. You can have any number of controllers and any number of views in your application.

Suppose you have an application that presents information from a database in a table and pie chart. The table can be scrolled using horizontal and vertical scroll bars, and new data can be entered via a pair of text fields. The MVC pattern might be applied as shown in Figure 4.

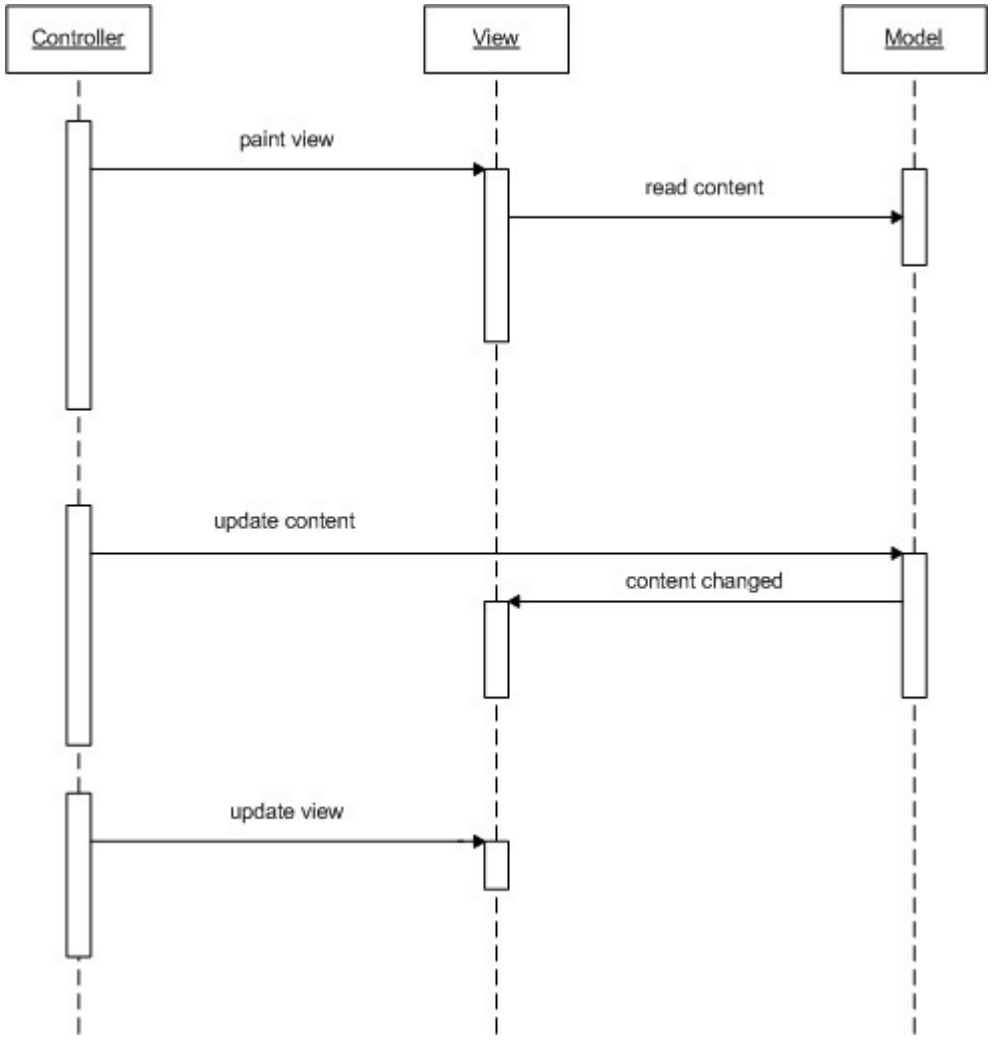
Figure 4. Multiple views and controllers



This example includes a model, two views and three controllers. The scrollbar controllers update only the table view, while the text field controller updates the model.

Figure 5 below shows the interacting among Model, View, and Controller objects using UML sequence diagram.

Figure 5. Interaction among Model, View, and Controller objects



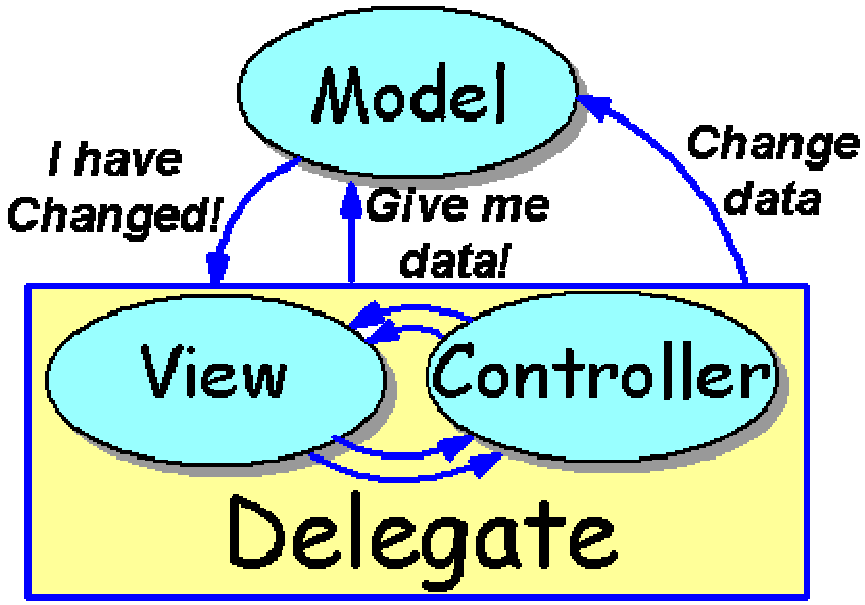
Delegates - Combining View and Controller

You may be concerned at this point about separation between the scrollbars and the table they scroll. In theory this separation is good, but in practice it can make life much more difficult:

- You need to separate the GUI among multiple elements, and provide variables and events to communicate between the controller GUI and the view GUI
- Where does an Swing *TextField* go? It acts like a view to display existing data, and like a controller to change the data.
- Many components have built-in support for user interaction such as scrolling.
- Interaction between multiple controllers and views can be quite heavy and complex.

Because of such issues, the MVC paradigm is often simplified by *combining* views and controllers. There are several names for this approach - the Swing designers use the name ***UI delegate*** to refer to a combined view/controller. This combination into a delegate is shown in Figure 6.

Figure 6. Combining view and controller into a delegate



In a delegate, the view and controller communicate as necessary to perform their duty. It is a good idea to keep this communication separate when possible, but often it's impossible to break a component into a view or a controller. And, in practice, the separation doesn't provide nearly as significant a benefit as separation of the Model and Delegate.

The delegate *as a whole* communicates with the model in the same way described earlier for the view and controller. The separation between the model and the user interaction of the delegate is the key to the success of this model. Although this version of MVC is somewhat simplified by the combination of the view and controller into the delegate, the design is still MVC-based.

In Java Swing each component contains a model and a UI delegate. The model is responsible for maintaining information about the component's state. The UI delegate is responsible for maintaining information about how to draw the component on the screen. In addition, the UI delegate (in conjunction with AWT) reacts to various events that propagate through the component. Note that the separation of the model and the UI delegate in the MVC design is extremely advantageous. One unique aspect of the MVC architecture is the ability to tie multiple views to a single model. For example, if you want to display the same data in a pie chart and in a table, you can base the views of two components on a single data model. That way, if the data needs to be changed, you can do so in only one place—the views update themselves accordingly. In the same manner, separating the delegate from the model gives the user the added benefit of choosing what a component will look like without affecting any of its data. By using this approach, in conjunction with the lightweight design, Swing can provide each component with its own pluggable look-and-feel.
As a programmer using Swing components, you generally do not need to think about the MVC architecture. Each user interface component has a wrapper class (such as *JButton* or *JTextField*)

that stores the model and the view. When you want to find out something about the content (for example, the text in the text field) the wrapper class asks the model and returns the answer to you. When you want to change the view, the wrapper class forwards the request to the view. So what is a *JButton*? It is simply a wrapper class inheriting from *JComponent* that holds the *DefaultButtonModel* object, some view data (such as the button label and icon) and a *BasicButton* object that is responsible for the button view.

The Swing MVC architecture has some side effects which is important to be mentioned. One of the important recommendations to the developers is that developers should not use independent threads to change model states in components. Once a component has been painted to the screen (or is about to be painted), updates to its model state should only occur from the *event-dispatching queue* . The event-dispatching queue is a system thread used to communicate events to other components. It handles the posting of GUI events, including those to repaint components. The issue here is an artifact of the MVC architecture and deals with performance and potential race conditions. As it is mentioned above, a Swing component draws itself based on the state values in its model. However, if the state values change while the component is in the process of repainting, the component can repaint incorrectly—this is unacceptable. To compound matters, placing a lock on the entire model, as well as on some of the critical component data, or even cloning the data in question, could seriously hamper performance for each refresh. The only feasible solution, therefore, is to place state changes in serial with refreshes. This ensures that modifications in component state do not occur at the same time that Swing is repainting any components, and no race conditions will occur.

Why is MVC So Important?

By this point you may be thinking, "Sounds like neat theory, but it also sounds like a lot of work!" Development typically occupies 10% or less of a program's life cycle. Therefore, program maintenance should be a developer's *number one* concern. A clean, understandable design is a good start, but to be really effective, the design should separate business logic from user interface. Think about some of the possible ways a program changes over time:

- The current state of GUI design evolves, and a company decides to update their GUIs.
- A company changes network architectures, and wants to port their applications.
- A company decides to create a limited-feature demo of their application.
- A company wants to add a new way of examining existing data.

Changes like these often involve *either* the GUI *or* the business logic (model) of an application. If the GUI code and business logic are tightly coupled, making these changes can be quite difficult. Using an MVC-based design makes the changes less extensive, and more importantly, more isolated, reducing the chance of introducing bugs in unrelated code.
Using an MVC-based design, the above changes are fairly simple:

- Updating the GUI requires *only* changing GUI code.
- Updating network architectures, perhaps changing from a two-tier to a three-tier database architecture requires modifying only part of the model. The stable GUI is not touched.
- Creating a limited feature demo might merely be a matter of subclassing the model to block access to some features. Again, no change to the GUI.
- Adding a new way to examine data is simply a matter of adding a new view. Often *no* change to the model is necessary, nor is it necessary to change other views!

A final thought on the importance of MVC: by separating the model (business logic) from the GUI, you can also separate the coding tasks. Separate developers can work on each part, for instance -- a GUI specialist coding the GUI and a domain expert coding the business logic.

The Observer/Observable Pattern

- **Pattern name:** *Observer/Observable*. **Category**: Behavioral.
- **Problem:** What if a group of objects needs to update themselves when some object changes state? This can be seen in the "model-view" aspect of the MVC (model-view-controller) architecture, or the almost-equivalent "Document-View Architecture." Suppose that you have some data (the "document") and more than one view, say a plot and a textual view. When you change the data, the two views must know to update themselves, and that's what the observer facilitates. It's a common enough problem that its solution has been made a part of the standard **java.util** library.
- **Solution:**

  There are two types of objects used to implement the observer pattern in Java. The ***Observable*** class keeps track of everybody who wants to be informed when a change happens, whether the "state" has changed or not. When someone says "OK, everybody should check and potentially update themselves," the **Observable** class performs this task by calling the **notifyObservers( )** method for each one on the list. The **notifyObservers( )** method is part of the base class **Observable**. There are actually two "things that change" in the observer pattern: the quantity of observing objects and the way an update occurs. That is, the observer pattern allows you to modify both of these without affecting the surrounding code.

  Observer is an "interface" class that only has one member function, **update( )**. This function is called by the object that's being observed, when that object decides if it is time to update all its observers. The arguments are optional; you could have an **update( )** with no arguments and that would still fit the observer pattern; however this is more general—it allows the observed object to pass the object that caused the update (since an **Observer** may be registered - using the ***addObserver()*** method - with more than one observed object) and any extra information if that's helpful, rather than forcing the **Observer** object to hunt around to see who is updating and to fetch any other information it needs. The "observed object" that decides when and how to do the updating will be called the **Observable**.

  Observable has a flag to indicate whether it's been changed. In a simpler design, there would be no flag; if something happened, everyone would be notified. The flag allows you to wait, and only notify the **Observers** when you decide the time is right. Notice, however, that the control of the flag's state is **protected**, so that only an inheritor can decide what constitutes a change, and not the end user of the resulting derived **Observer** class.

  Most of the work is done in **notifyObservers( )**. If the **changed** flag has not been set, this does nothing. Otherwise, it first clears the **changed** flag so repeated calls to **notifyObservers( )** won't waste time. This is done before notifying the observers in case the calls to **update( )** do anything that causes a change back to this **Observable** object. Then it moves through the **set** and calls back to the **update( )** member function of each **Observer**.

  At first it may appear that you can use an ordinary **Observable** object to manage the updates. But this doesn't work; to get an effect, you *must* inherit from **Observable** and somewhere in your derived-class code call **setChanged( )**. This is the member function that sets the "changed" flag, which means that when you call **notifyObservers( )** all of the observers will, in fact, get notified. *Where* you call **setChanged( )** depends on the logic of your program.
- **Consequences:** The Observer object and the Observer can work asynchronously and independantly.

Figure 7 and 8 show the UML class diagrams of the Observer/Observable design pattern.

Figure 7. The classic Observer pattern UML diagram

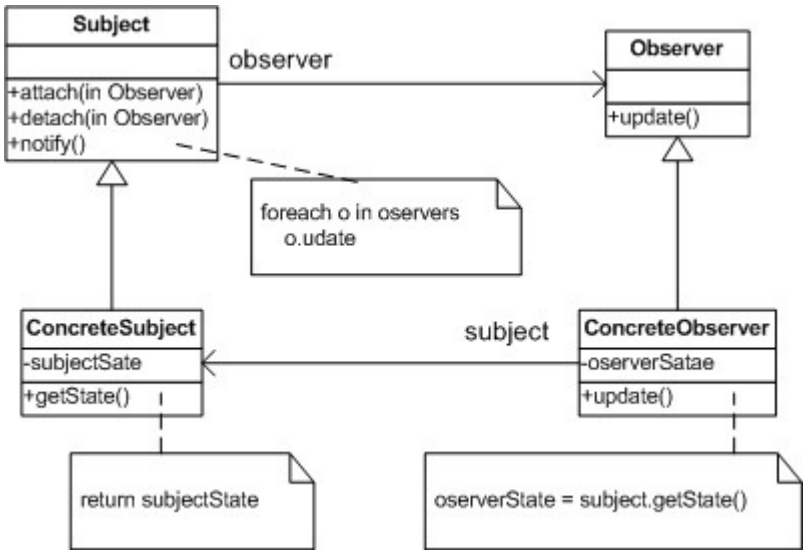
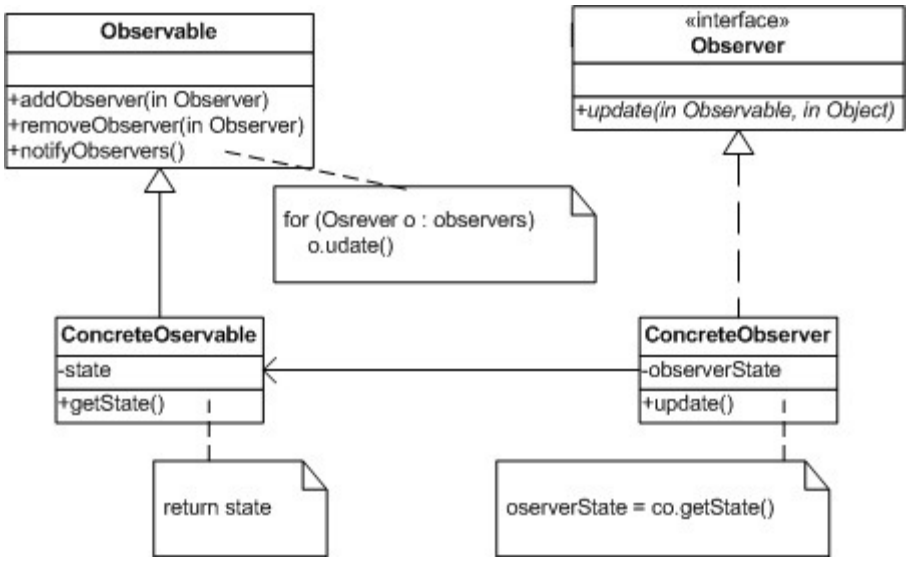


Figure 8. The Java Observer pattern implementation UML diagram



Refer to the code examples to see how the Observer pattern is implemented in Java.