Prasun Dewan[1]

3. Shared Objects

In the previous chapter, we saw how WYSIWIS applications can be supported by shared window systems. Here we will focus on non-WYSIWIS interfaces. We will define them using the model-view-controller framework, which allows multiple views to share the state of a single object. This framework will be extended to the distributed MVC, which allows the views and model to be distributed. We will see how this architecture can be implemented using the concept of remote method invocation. Distributed MVC centralizes the model. An alternative is to replicate the model. We will look at techniques for implementing replicated objects.

Non WYSIWIS User Interfaces

As we saw in the previous chapter, different window systems can support sharing of different properties of a window. While these systems are free to determine if they share exposed regions and window state, such as window positions and stacking order of windows, they must share window sizes and contents. This means that they cannot, for instance, allow one user to see a pie chart view and another to see the bar chart view of some data, as shown below.

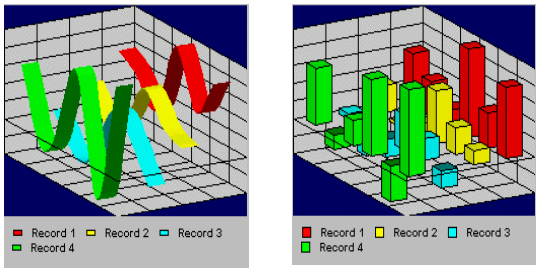


Figure 1 Multiple Views of a Shared Model

Multiple views of application data are not unusual. For instance, the popular PowerPoint application allows multiple views of a slide deck, and one can imagine different users looking at different views.

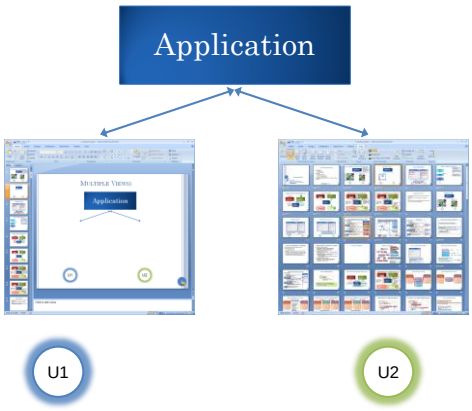


Figure 2 Multiple views of a PPT slide deck

An even more common use of multi-view collaboration is shown in the figure below.

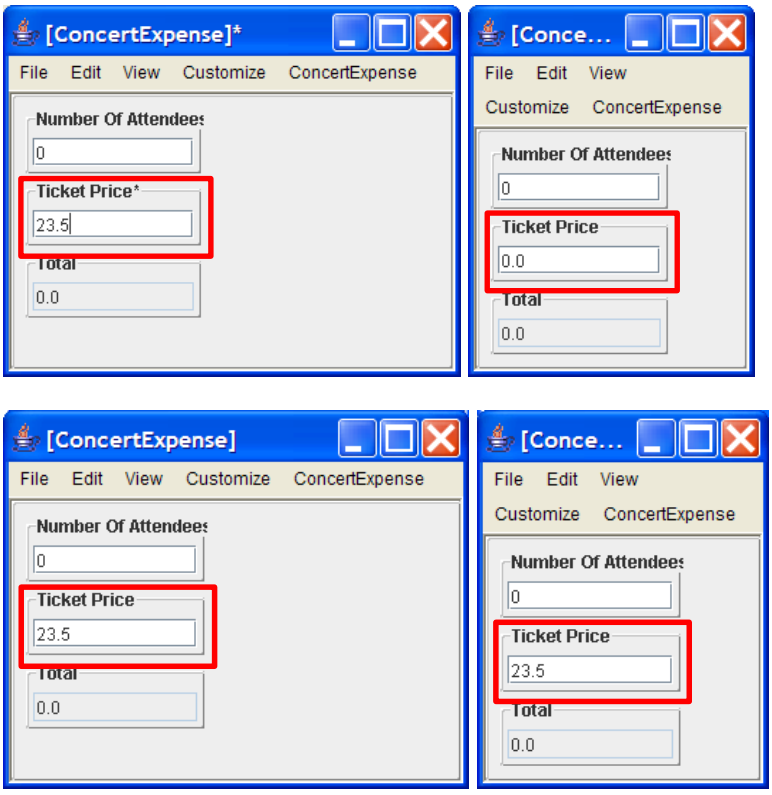


Figure 3 Multi-view, multi-user interaction for managing concert expenses

Here two users are collaboratively interacting with a custom spreadsheet for managing expenses for a group of friends going to a concert together. This application could be used in different ways. The two users may be responsible for determining and entering the ticket price and number of attendees, respectively, or they may not have a division of responsibility, where instead whoever has the latest information enters it. As we see above, the sizes of the windows of the two users is different. Moreover, there is a level of asynchrony in the collaboration. The ticket price entered by the left user is not seen by

the right user as it is entered. When this value is committed (by hitting Enter), it is seen by the right user. We will use this application as a running example to concretely illustrate various concepts.

We see above three examples of non-WYSIWIS interfaces which cannot be supported by shared window systems. To determine how they can be supported, we need to identify not what they *are* not (WYSWIS), but rather what they *are*. In fact, the single-user I/O architecture we saw earlier, reproduced below, does not have a way of expressing the coupling between the user-interfaces. The problem is that this framework assumes the application creates a user-interface through input and output operations. What we need is an architecture that captures the initutive notion of multiple views of application data.

Model View Controller

The Smalltalk model view controller (MVC) framework is such an architecture. It assumes that the semantic state of an interactive application is maintained by an object, called the model. One or more view objects output the state to the user, and one or more controller objects change this state in response to user input. The view (controller) objects call read (write) methods in the model to display (update) its state. Whenever the model is changed, autonomously or by a controller, it informs all of its views that it has changed. As a view is dynamically created/destroyed, it registers/unregisters itself with the model so that the model knows who it should notify.

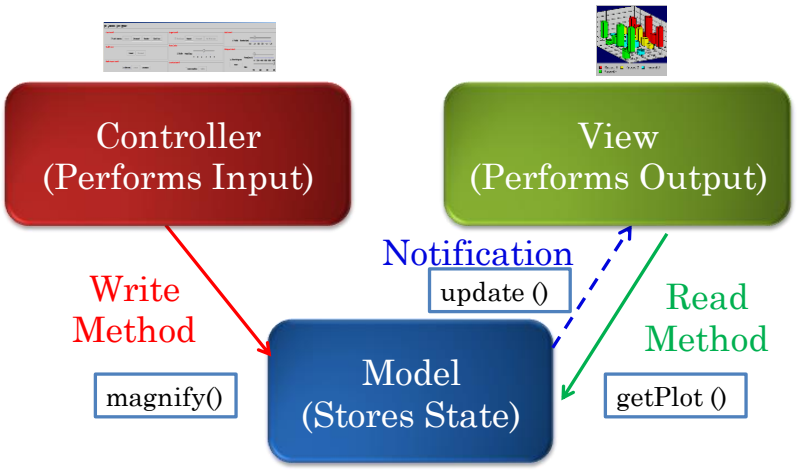


Figure 4 Model View Controller (MVC) Framework

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. We will see the use of this feature later.

In general, a notifying object is called an observable and a notified object is called its observer. In the figure, the dashed lines from the models to its views indicate that its awareness of its observers is *notification awareness*: it knows which observers need to be notified about changes to it but does not

know any aspect of their behavior. This is in contrast to the awareness in the views and controllers of the model, which know about the model's getter and setter methods. If the model changes, the views and controllers typically change in response because of this awareness. Even though the observable/observer notion appeared first with the context of the larger MVC pattern, it is now recognized as an independent pattern.

In addition to objects, humans regularly use this idea of notification. 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 another model) 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 identifies 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.

The observer pattern allows models and views to be bound to each other. A separate mechanism is needed to bind a controller to one or more models. A controller can provide a setter method for doing so, and either it or some other object can call this method. A view with a single model can also provide such a method so that it does not have to rely on the notification method providing the model reference.

Let's use the concert expense application to concretely understand MVC.

The code below implements the model of the application.

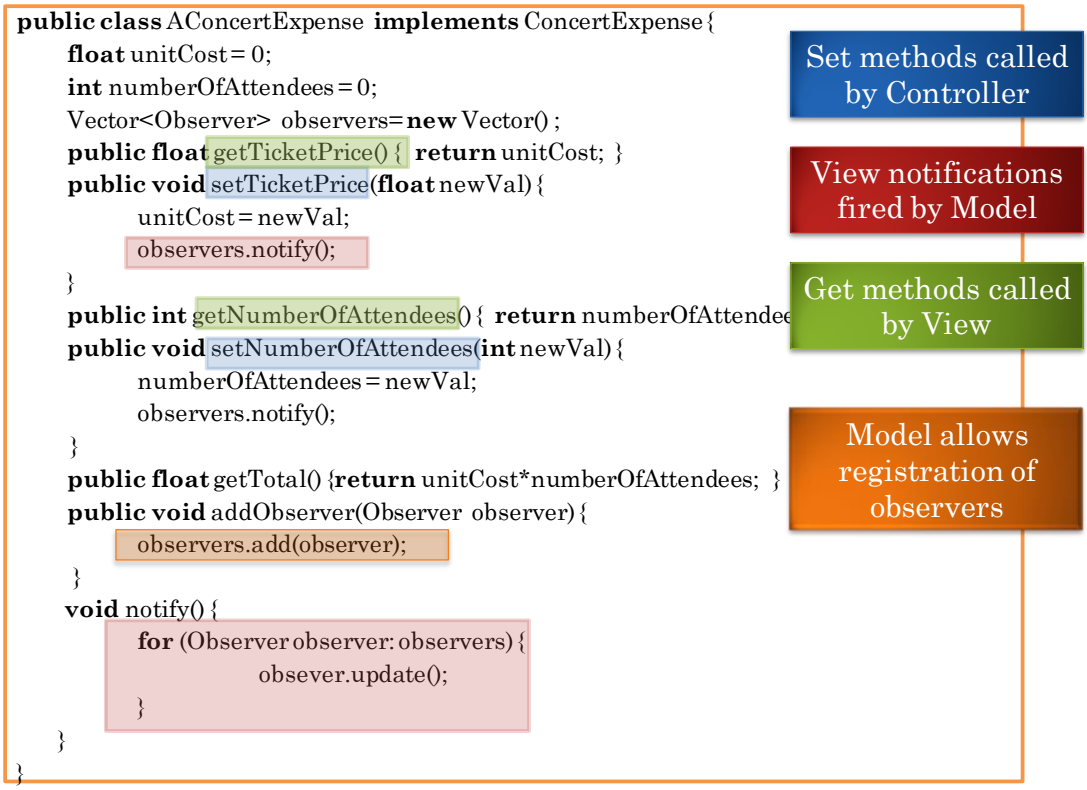


Figure 5 Model Code for Concert Expense

The list of registered observers is kept in the observer history, `observers`. The method `addObserver` adds elements to this list. In general, an observable also defines a method to remove a registered observer. The method `notifyObservers` retrieves each element of `observers`, invoking the `update` method defined by the Observer interface, shown below.

```
public interface Observer {
        public void update();
}
```

Figure 6 Observer Interface

All views must implement this observer interface implemented by the model.

To understand the relationship between this model and its views and controllers, let us see what happens when a user edits the ticket price and hits Enter:

- The controller converts the text entered by the user into a float value (checking for errors), and calls setTicketPrice(float) in the model with this value.
- The model updates an instance variable and calls `notifyObservers`, which in turn, invokes `update` on all the observer views in the order in which they were registered.

- The views call all of the read methods of the model, getTicketPrice(), getNumberOfAttendees(), and getTotal() to determine the new state of the model, and update their display.

Generic View and Controller

To complete the implementation of our running example, we should show the implementation of the view and controller. Usually, these user-interface modules are implemented using either Console I/O routines, which create Console user-interfaces, or (user-interface) toolkits, which create GUIs. In either case, surveys show, about fifty percent of the total code of the application deals with tedious I/O tasks, and this code is messy. More important, from the point of view of this course, creating GUIs such as the concert application user interface requires learning a large number of toolkit details, which are beyond the scope of this course.

Therefore, we will use an alternative approach to creating MVC-based GUIs, which involves the use of a generic tool, called ObjectEditor, that can automatically create user-interfaces of models. ObjectEditor provides a generic view and controller to interact with the model, as shown below.

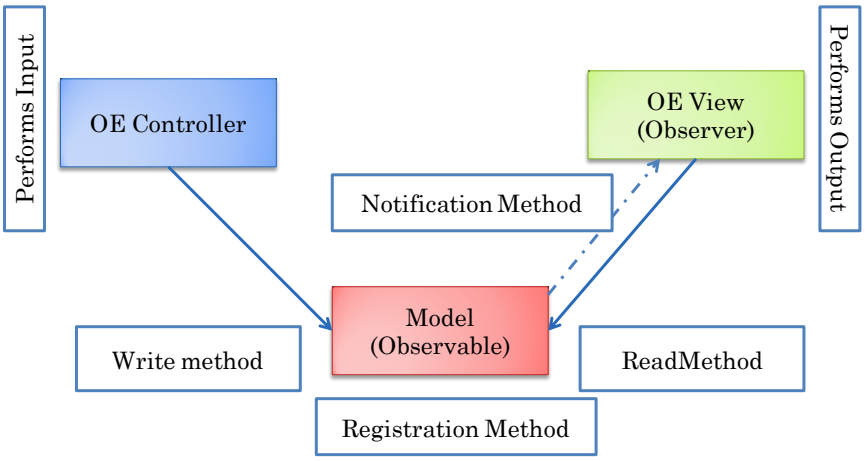


Figure 7 MVC-based ObjectEditor

As programmers, we must bind the generic view and controller to the model. This is done by calling the edit() method of ObjectEditor, as shown below.

```
package main;
import budget.ConcertExpense;
import budget.AConcertExpense
import bus.uigen.ObjectEditor;
public class MVCBudget {
    public static void main(String[] args) {
            ConcertExpense model = new AConcertExpense ();
            ObjectEditor.edit(model);
    }
}
```

Figure 8 Binding a model to the ObjectEditor View and Controller

Thus, we have completed the ObjectEditor-based implementation of the concert expense application. This implementation supports single-user interaction. How should we support multi-user interaction?

Distributed MVC

MVC is particularly well suited for multi-user interaction. In the single-user case, it allows a single user to create multiple views and controllers to be attached to a model.

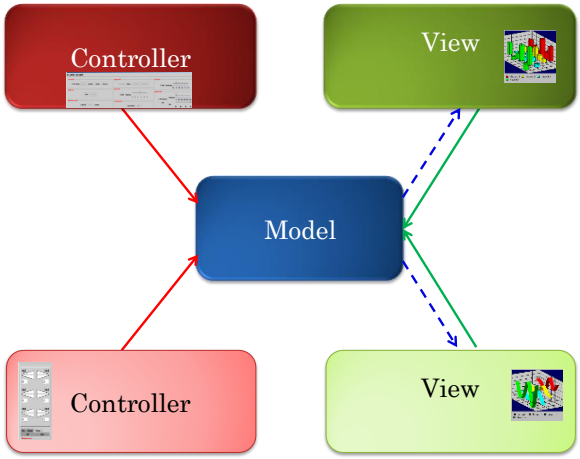


Figure 9 Multiple views and controllers of a single user attached to a model

In the multi-user case, "all" we have to do is make sure that views and controllers, residing on different users' computers, attach to a common model which may be located on one of the collaborators' computers, or some special computer.

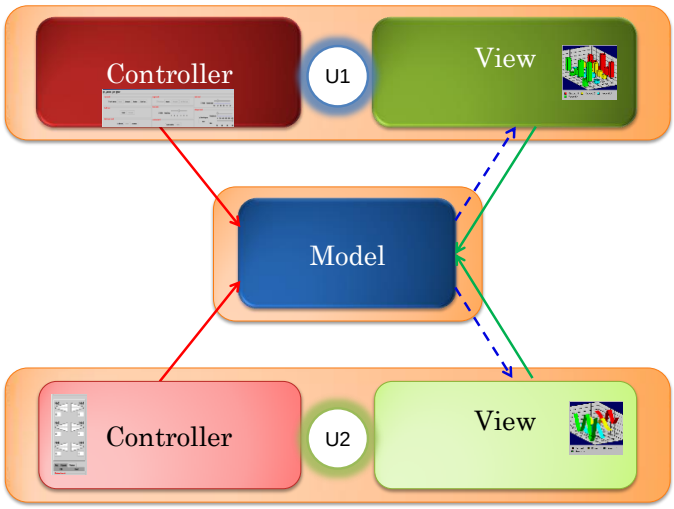


Figure 10 Multiple views and controllers of different users attached to a common model

The word “all” is in quotes because allowing a model to communicate with remote views and controllers of different users is not trivial. It raises several issues, especially when we try to automate collaboration functions:

1. Identifying model components: How does an infrastructure provide fine-grained collaboration functions, that is, concurrency control, coupling, access control, and merging mechanisms that depend on the components of the model?
2. Efficient communication: How do we minimize the number of messages that are sent over the network to process input and update the views?
3. Communication mechanism: How do the controllers, views, and models invoke operations in remote objects?

Logical vs. Physical Structure

Consider first the issue of identifying model components. Unlike a shared window system, distributed MVC can allow multiple users to interact simultaneously with a model without causing assertion violations. In other words, any sequence of method calls invoked by controllers and views of different users can also be invoked by controllers and views of a single-user. However, it is possible for users to step on each other’s toes. For example, if two users concurrently edit the ticket price in our example, one of them will overwrite the input of the other. Moreover, it is possible to violate access control policies. For example, a user not authorized to edit the ticket price might do so. Furthermore, if the two users are disconnected, and one of them edits the ticket price and another edits the number of attendees, it should be possible to safely merge their changes. Finally, if we extend the example to allow the addition of a comment, it may be useful to not couple it while coupling the other components of the model such as ticket price.

We could implement coupling, merging, concurrency, and access control in the model by making it collaboration-aware. However, ideally, we would like these collaboration functions to be implemented automatically in a collaboration infrastructure. This implies that the infrastructure must be able to decompose the model into parts to, for instance, allow only some users to edit the ticket price. Thus, we need a general way to derive the structure of a model.

It is possible to decompose an object based on its instance variables. The object is decomposed into the values stored in its instance variables, and each of these values is further decomposed in this fashion, until we reach primitive values. We will refer to this structure as the physical structure of the object, as it is essentially the structure of its physical representation in memory.

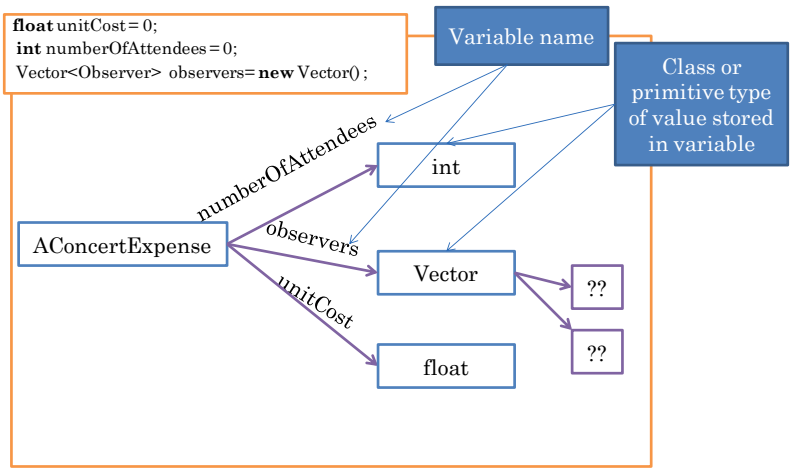
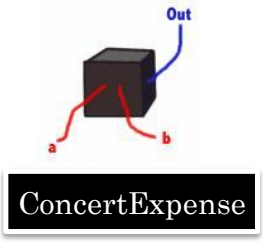


Figure 11 Physical Structure

The "??" in the boxes for the components of a Vector indicate that we cannot complete the structure without knowing the instance variables defined by class Vector. This information is deliberately kept from the classes that use a Vector. In fact, even in our implementation of AConcertExpense, we did not make the instance variables public. Thus, unless the infrastructure is tied to the language compiler, it cannot access the structure above.

We could imagine giving it the same access as a debugger, but that would mean that what the end-user sees, such as the components that can be locked, is tied to the implementation of the model, which violates the principle of encapsulation in object-oriented programming languages. Thus the idea of encapsulation seems to be at odds with our goal of automatically supporting fine-grained collaboration functions in a collaboration infrastructure. Encapsulation hides information, treating the object as a black box whose internal details are hidden from external observers. As the figure shows, the only visible components are its "input" and "output", which in the case of an object are the signatures of its public methods, giving the names, parameter types, and result types of these methods. An infrastructure, on the other hand, needs information about the structure of the object.




```
public interface ConcertExpense {
        public float getTicketPrice();
        public int getNumberOfAttendees();
        public void setNumberOfAttendees(int newVal);
        public void setTicketPrice(float newVal);
        public float getTotal();
        public void addObserver(Observer observer);
}
```

Figure 12 A black box object exposing only the signatures of its public methods

The only way out of this impasse is for the infrastructure to somehow derive the structure of an object from these signatures. In the case of our example, as humans we are, in fact, able to use this information to map the object to the components shown in the figure below.

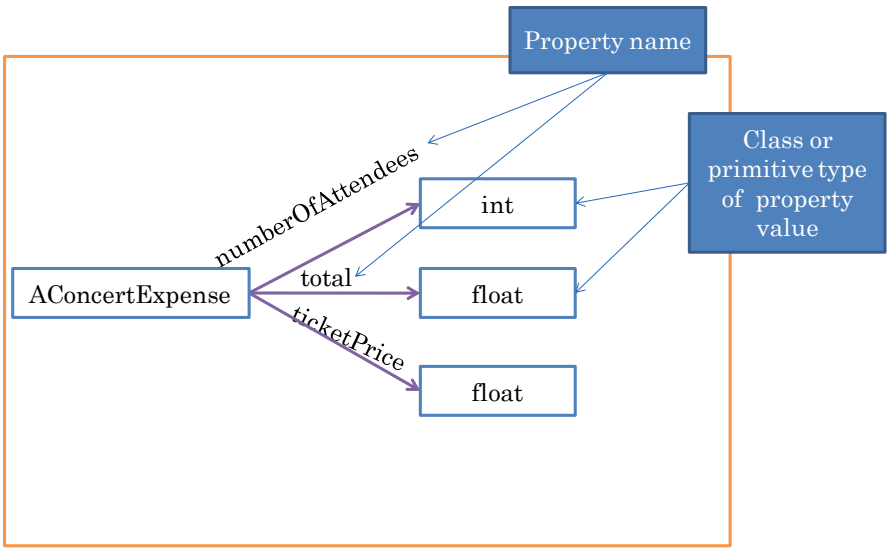


Figure 13 Logical structure derives from signatures of the public methods of an object

Here we are identifying typed "properties" of an object. The object is decomposed into the values stored in its properties, and each of these values is further decomposed in this fashion, until we reach primitive values. We will refer to this structure as the logical structure of an object, as it corresponds, not to how Java physically lays out the object, but how we logically view the object. The physical structure of an object depends on its implementation – specifically, its instance variables – while its logical structure depends only on its public methods. We will use the term physical (logical) component to refer to an instance variable (property). Logical components are units of the external state of an object, while the physical components are units of the internal state of the object.

As the figures above show, to draw the physical (logical) structure of a value, we start with the name of the primitive type or class of the value. For each physical (logical) component of each instance of the object, we draw an edge from the name. The label of the edge is the name of the physical (logical) component and its end point is the physical (logical) structure of the value of the physical (logical) component. We stop at values that cannot be decomposed.

The term "bean" symbolizes a reusable component that can easily work together with other beans to create applications, much as a coffee bean can be seamlessly mixed with other coffee beans to create a coffee flavor.

Even classes that do not follow these conventions qualify as beans as long as they provide some way of specifying the getter and setter methods of properties. Naming conventions are one way of doing so. The Java beans framework provides other mechanisms to do so, which are much more complicated and meant mainly meant for classes that were written before these conventions were defined. When you are creating new classes, as in this book, it is best to simply follow these naming conventions, which have the important effect of make making them easier to understand.

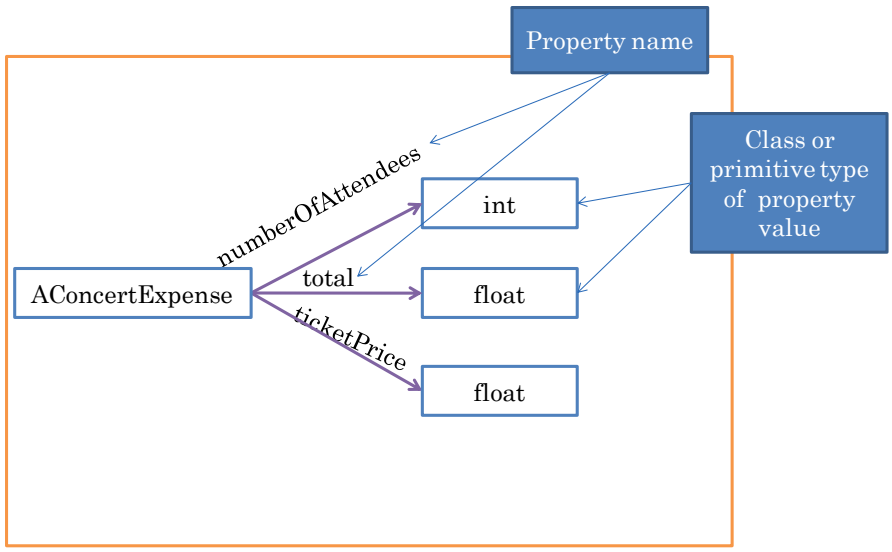


Figure 14 Logical Structure

As the example shows, some properties, in particular numberOfAttendees and ticketPrice, correspond to instance variables, numberOfAttendees and unitCost. However, some instance variables (such as observables) are not exported as a logical component. Conversely, it is possible to export a logical component such as total that is not directly stored in memory but computed by the object.

Bean Conventions and Programming Patterns

For an infrastructure to automatically derive the logical structure of an object, we need to define more precisely the notion of a property.

A class defines a property `P` of type T if it declares a getter method for reading the value of the property, that is, a method with the following header:

```
public T getP()
```

If it also declares a setter method to change the property, that is, a method with the header

```
public void setP (T newP)
```

then the property is *editable*; otherwise it is *read-only*.

As we see from these definitions, the getter and setter methods of a property must begin with the word "get" and "set", respectively. Of course, names do not affect the semantics of these methods. For instance, had we instead named `getBMI` as `obtainBMI`, we would not

change what the method does. However, in this case, we would be violating the *bean conventions* for naming getter and setter methods. The words “get” and “set” are like keywords in that they have special meanings. While keywords have special meanings to Java, “get” and “set” have special meanings to those relying on bean conventions. Under these conventions, the names of both kinds of methods matter, but not the names of the parameters of the setter methods.

On the other hand, the number and types of parameters and results of the methods matter. The getter method must be a function that takes no parameter, while the setter method must be a procedure that takes exactly one parameter whose type is the same as the return type of the corresponding getter method.

These conventions, like any other programming conventions, are useful to (a) humans trying to understand code so that they can maintain or reuse it, and (b) tools that manipulate code. A class that follows these conventions is called a *bean*.

In the definition of properties, we used the relationships among the *signatures of the methods* of *a specific class*, as shown in the figure. We will refer to these relationships as *programming patterns*. These are to be distinguished from *design patterns*, which describe the *protocol* used by (instances of) multiple classes to communicate with each other. MVC is an example of a design pattern, as it describes how models, views, and controllers, which are instances of different classes, communicate with each other.

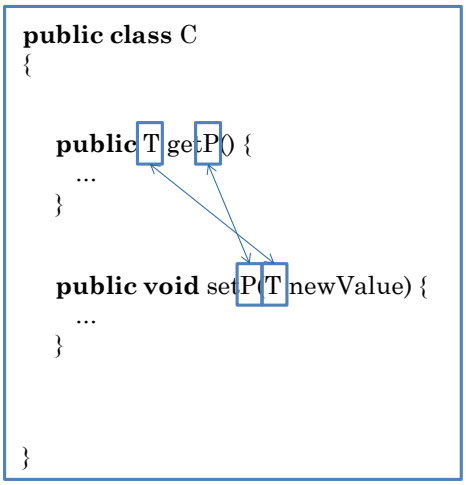


Figure 15 Relationships among method signatures in the Bean programming pattern

Programming Patterns for Collections and Tables

While all properties are logical components, not all logical components are properties. The reason is that an object can have only a static number of properties, defined by the getter and setter methods of its class. It is not possible to dynamically add properties as an object executes. Consider the following modification to the ConcertExpense interface in which the list of observers is exported by a getter method. This list dynamically grows as observers register with the model.

```
public interface ConcertExpense {
        public float getTicketPrice();
        public int getNumberOfAttendees();
        public void setNumberOfAttendees(int newVal);
        public void setTicketPrice(float newVal);
        public float getTotal();
        public void addObserver(Observer  observer);
        public Vector getObservers();
}
```

Figure 16 A Collection Property

Previously, we could not identify the physical components of this list, as we did not know the internal structure of Vector. Now we cannot identify the logical components of this collection, as we have no programming patterns for identifying dynamic logical components.

If we are building a Java-based infrastructure, we could assume that all dynamic lists are instances of the Collection types defined by Java. However, this does not allow us to support programmer-defined dynamic lists such as the one shown below.

```
public interface  StringHistory  {
        public void addElement(String element);
        public int size();
        public String elementAt(int index);
}
```

Figure 17 A String History

This type is a history in that once an element is added, it cannot be deleted or modified. No existing Java Collection type directly supports histories – we must create them ourselves, possibly using some existing Java Collection type.

Thus, it would be useful to support programmer-defined types such as Vector and StringHistory that support a dynamic indexable number of logical components. As there are no standard programming patterns for such types, let us define our own, based on the conventions used in class Vector.

We will say that a type is an indexable collection of logical components of type T if it provides (a) a public method named `elementAt` taking a single `int` parameter returning an element of type T, and (b) a public parameter-less method named `size` to determine the number of elements currently in the collection. This is in the spirit of looking for methods whose names start with `get` to determine and access the static properties of an object. Like the getters, these are read methods called by views.

An indexable collection can provide additional optional methods to insert, delete, or modify its components. We will assume the patterns shown below for defining these methods. These methods are like the optional setter methods in a Bean. Like the setter methods, these are write methods called by controllers.

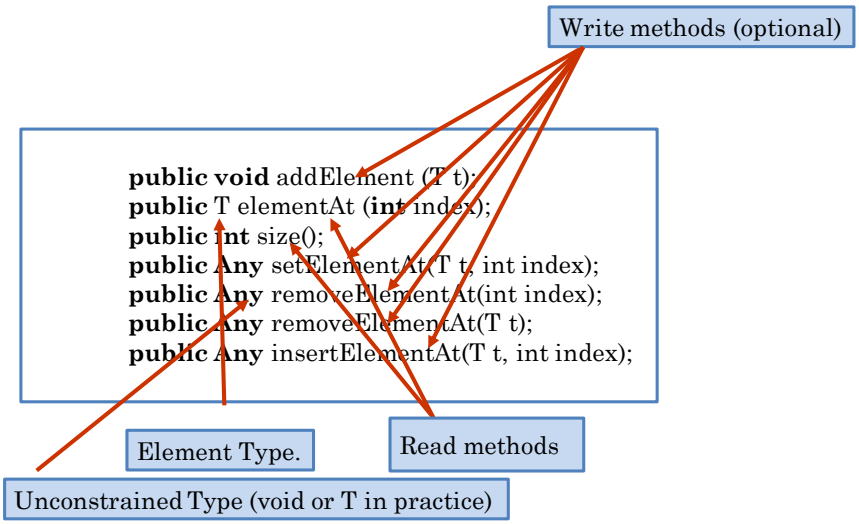


Figure 18 Vector-based Programming Pattern

The programming pattern above is based on the Java class Vector, which was the first indexable collection type offered by Java. Some of the infrastructures you will use in this class were built before other Java Collection types were introduced and thus assume these patterns. It is, of course, possible to define other conventions, based, for instance on the Java ArrayList class.

Another important logical structure is a table. The figure below defines the programming patterns assumed here for defining a table that maps keys of type K to elements of type E. The type Enumeration is the precursor of the more popular type Iterator.

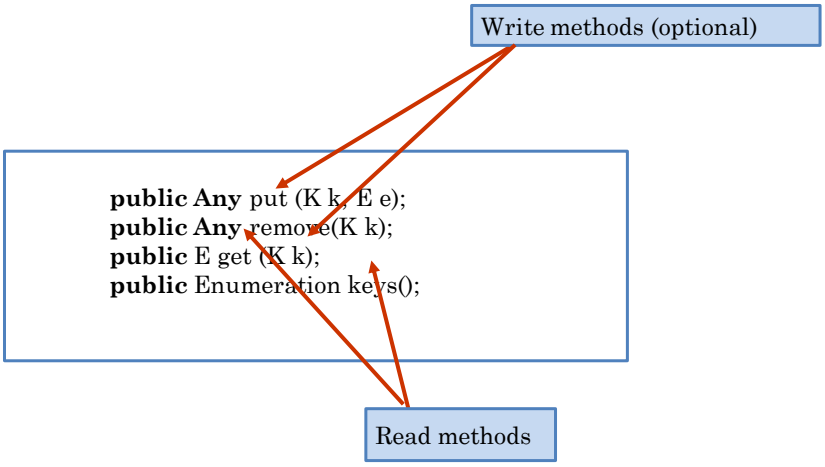


Figure 19 Table Programming Patterns

We see below an example of this pattern, which does not define any remove method.

```
public interface  StringToConcertExpense {
        public  ConcertExpense get(Key string);
        public  void put(Key string, ConcertExpense expense) ;
        public Enumeration<String> keys();
}
```

Figure 20 Implementation of the Table Pattern

Composing Patterns

These Bean, collection, and table programming patterns allow us to compose a variety of programmer-defined classes whose logical structure is exposed to an infrastructure. The figure below shows an example.

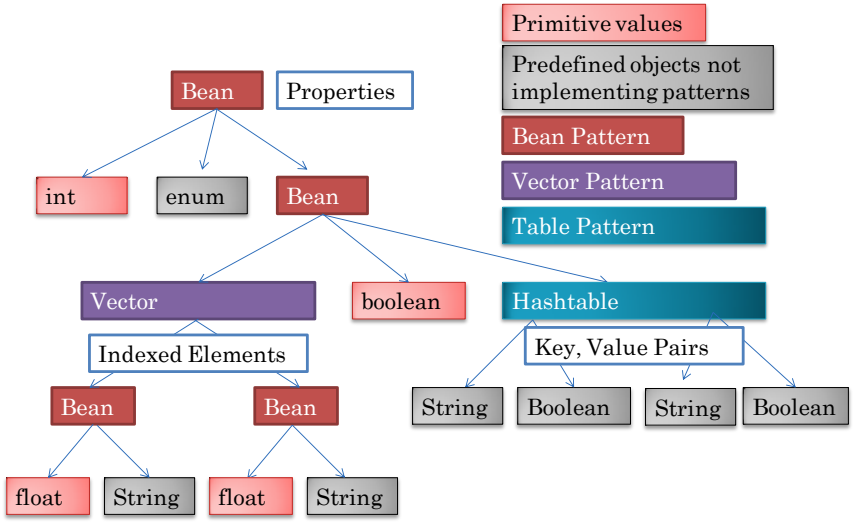


Figure 21 Pattern Composition

Here we have used color coding to identify the pattern used in the definition of each structured logical component. The leaf components of this structure are either primitive types, such as int and boolean, or predefined object types, such as String and Boolean that are well known to the infrastructure.

It is possible for a logical component to be a programmer-defined type that does not follow one of these three programming patterns. The modified ConcertExpense type given below is an example of such a type, as it no longer follows Bean conventions. An infrastructure would regard such as type as a leaf node in the logical structure, and would be unable to, for instance, lock components of the type.

```
public interface ConcertExpense {
        public float obtainTicketPrice();
        public int numberOfAttendees();
        public void writeNumberOfAttendees(int newVal);
        public void ticketPrice(float newVal);
        public float computeTotal();
        public void addObserver(Observer  observer);
}
```

Figure 22 A Programmer-defined type not following any programming pattern

Efficient Communication in Distributed MVC

Programming patterns offer a solution to the problem of automatically providing fine-grained collaboration functions in an infrastructure that allows multiple users to use different views to manipulate shared models. Let us next consider the issue of efficient distributed communication.

In distributed MVC, a model is on a different machine from the view and controller of a user. This means passing parameters to it and receiving results from it involves sending messages over the network, which can be costly. To illustrate this cost, consider what happens when a user makes a new input.

1. The controller of the user sends the model a message with the new input.
2. The model processes the input, and then sends a message to each view informing them that it has changed.
3. Each view responds by invoking one or more read methods in the model to get the current state of the model.
4. The model responds to each read method by sending a message containing the returned value.

Some of these steps such as 1 cannot be optimized. However, it is possible to make this communication more efficient by having the model send, with the notification to the views, the parts of its logical structure that have changed. As a result, steps 3 and 4 are not necessary.

Thus, the notion of the logical structure is necessary to support not only fine-grained collaboration functions but also fine-grained notifications. The nature of the notification sent when some component of a logical structure changes is tied to the programming pattern defining the structure.

The following modified version of AConcertExpense illustrates the nature of the notifications defined for the Bean pattern by Java.

```
public class AConcertExpense implements  ConcertExpense  {
    float unitCost = 0;
    int numberOfAttendees = 0;
    PropertyChangeSupport propertyChange = new PropertyChangeSupport(this);
    public float getTicketPrice() {  return unitCost;  }
    public void setTicketPrice(float newVal) {
            if (newVal == unitCost) return;
            float oldVal = unitCost; int oldTotal = getTotal();
            unitCost = newVal;
            propertyChange.firePropertyChange("ticketPrice", oldVal,  newVal);
            propertyChange.firePropertyChange("total", oldTotal,   getTotal());
    }
    public int getNumberOfAttendees() {  return numberOfAttendees;   }
    public void setNumberOfAttendees(int  newVal) {
            if (numberOfAtendees == newVal) return;
            int oldVal = numberOfAttendees; int oldTotal = getTotal();
            numberOfAttendees = newVal;
            propertyChange.firePropertyChange("numberOfAttendees", oldVal, newVal);
            propertyChange.firePropertyChange("total", oldTotal, getTotal());
    }
    public float getTotal() {return unitCost*numberOfAttendees;   }
    public void addPropertyChangeListener(PropertyChangeListener l) {
       propertyChange.addPropertyChangeListener(l);
    }
}
```

Figure 23 Property Notifications

This model does not directly store or notify observers in a list. Instead, it uses an instance of PropertyChangeSupport, provided by Java, to do so. As we see above, a notification indicates name of the property that changed, its old value, and its new value. By sending the old values, the observable frees the observables from remembering these values. The firePropertyChange() method of PropertyChangeSupport converts these three values into an instance of the standard Java PropertyChangeEvent and passes this value as a parameter to the notification method invoked in a PropertyChangeListener.

The following code gives the exact details of the notification method and event.

```
public interface java.beans.PropertyChangeListener
            extends java.util.EventListener {
      public void propertyChange(PropertyChangeEvent evt);
}

public class java.beans.PropertyChangeEvent
            extends java.util.EventObject {
     public PropertyChangeEvent (
            Object source, String propertyName,
            Object oldValue, Object newValue) {…}
     public Object getNewValue() {…}
     public Object getOldValue() {…}
     public String getPropertyName() {…}
```

```
        …
    }
```

An observable for such observers must define the following method to add an observer:

```
addPropertyChangeListener(PropertyChangeListener l) {…}
```

Our example model implements this method by simply asking PropertyChangeSupport to keep track of the listener.

In addition, an observable can optionally define the following method to remove an observer:

```
removePropertyChangeListener(PropertyChangeListener l) {…}
```

Our example model does not define this method.

ObjectEditor is an example of a PropertyChangeListener, as shown below.

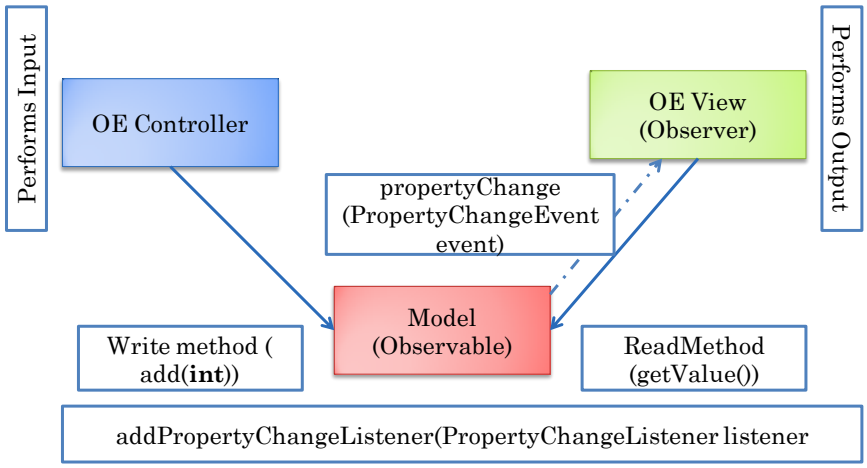


Figure 24 ObjectEditor as a PropertyChangeListener

If a model defines the method:

```
addPropertyChangeListener(PropertyChangeListener l) {…}
```

the ObjectEditor view automatically calls it to register itself with the object. It reacts to the notification about a property change by updating display of property, as shown below:

```
public class ObjectEditorView implements
        java.beans.PropertyChangeListener {
    public void propertyChange (PropertyChangeEvent arg) {
        // update display of property arg.getPropertyName()
        // to show arg.getNewValue()
        …
    }
}
```

Collection and Table Notifications

Property notifications work only for objects implementing the Bean pattern. They do not work for changes to a variable sized collection, such as a AStringHistory. When we invoke a write method (like addElement()) on such a collection, we do not change any property. There is no standard interface for communicating information about collection changes, so we will define one which is summarized in the figure below.

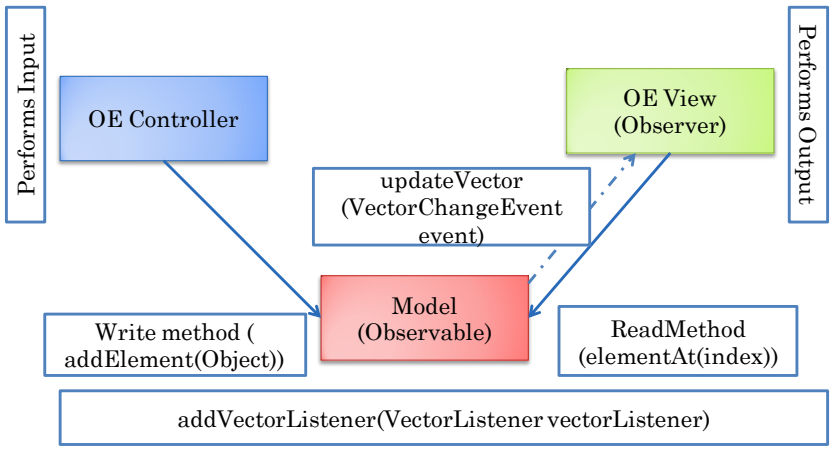


Figure 25 Collection notifications

To support this architecture, we define the following VectorListener and VectorChangeEvent interfaces, which correspond to the PropertyChangeListener and PropertyChangeEvent interfaces.

```
package util.models;
public interface VectorListener {
        public void updateVector(VectorChangeEvent evt);
}
```

```
package util.models;
public class VectorChangeEvent {
       // constants to be used for event type
       public static final int AddComponentEvent = 1,
                               DeleteComponentEvent = 2,
                               ChangeComponentEvent = 3,
                               InsertComponentEvent = 4,
                               CompletedComponentsEvent = 5,
                               ClearEvent = 6,
                               UndefEvent = 1000;
       // constructor, oldObject can be null when no value  is
replaced
       public VectorChangeEvent(Object theSource, int type,
int posn, Object oldObject, Object newObject, int newSize) {

       …
       }
       …
}
```

The word, Vector, in these interfaces reflects the fact that they were invented for Vectors. In fact, they can be used for any variable sized collection implementing the collection pattern given before. VectorChangeEvent recognizes four important different changes to such a collection: adding a new element to the end of the collection, deleting an existing element, inserting a new element at an arbitrary position, and clearing of the collection. It defines integer constants for each of these event types. The event indicates the source object that fired it, its type (defined using one of the constants mentioned above), the position of the element added/inserted/deleted by the write operation, the old object (if any) at that position, the new object (if any) inserted by the operation at that position, and the new size of the collection after the operation finishes execution. Knowing the new size will be important when we create replicated models.

We also define a class VectorChangeSupport, which is the analogue of the Java PropertyChangeSupport class. The following code shows its use in the StringHistory example.

```
import util.models.VectorChangeSupport;
import util.models.VectorListener;
public class AnObservableStringHistory extends AStringHistory {
VectorChangeSupport vectorChange = new
AVectorChangeSupport(this);
    public void addVectorListener(VectorListener vectorListener) {
        vectorChange.addVectorListener(vectorListener);
    }
    public void addElement(String element) {
        super.addElement(element);
         // vectorChange constructs VectorChangeEvent
        vectorChange.elementAdded(element);
    }
}
```

Figure 26 Notifying String History

Assuming a working implementation of the StringHistory interface, AStringHistory, we have created above a subclass that notifies observers about the additions to the collection. For each kind of write operation such as addElement(), VectorChangeSupport defines a corresponding method, such as elementAdded(), that can be invoked to inform all observers about the operation. This method creates an appropriate change VectorChangeEvent and passes it to the updateVector() method of the observer.

The updateVector() method must decode the update operation by looking at the type property of the event. We also provide a more direct way to get collection notifications, which does not involve such decoding. We define another listener interface, called VectorMethodsListener, which unlike PropertyChangeListener and VectorListener, is not event based. Instead, it defines a separate method for each kind of write operation.

```
package util.models;
public interface VectorMethodsListener<ElementType> {
  public void elementAdded(Object source, ElementType
element, int newSize);
  public void elementInserted(Object source, ElementType
element, int pos, int newSize);
  public void elementChanged(Object source, ElementType
element, int pos);
  public void elementRemoved(Object source, int pos, int
newSize);
  public void elementRemoved(Object source, ElementType
element, int newSize);
  ...
}
```

Figure 27 VectorMethodsListener

As we see above, the interface defines a large number of methods, which must be implemented by every implementation of it. Thus, whether a collection observer implements VectorMethodsListener or VectorListener depends on how many different kinds of update operations the observer is interested in.

VectorChangeSupport handles notifications to both kinds of observers, as illustrated by the following variation of AnObservableStringHistory.

```
import util.models.VectorChangeSupport;
import util.models.VectorListener;
import util.models.VectorMethodsListener;
public class AnObservableStringHistory extends AStringHistory {
VectorChangeSupport vectorChange = new
AVectorChangeSupport(this);
    public void addVectorListener(VectorListener vectorListener) {
        vectorChange.addVectorListener(vectorListener);
    }
    public void addVectorMethodsListener(VectorMethodsListener
    vectorListener) {
        vectorChange.addVectorMethodsListener(vectorListener);
    }
    public void addElement(String element) {
        super.addElement(element);
         // vectorChange notifies both kinds of listeners
        vectorChange.elementAdded(element);
    }
}
```

Figure 28 Adding both kinds of listeners

The model simply needs to define an extra method that allows registration of VectorMethodsListener instances. The notification methods such as elementAdded() defined by VectorChangeSupport notify both kinds of listeners.

Table Notifications

There are only two kinds of table notifications: (1) a key, value pair has been put, or (2) a key has been removed. Therefore, we support only one kind of table listener, based on VectorMethodsListener rather than event-based VectorListener.

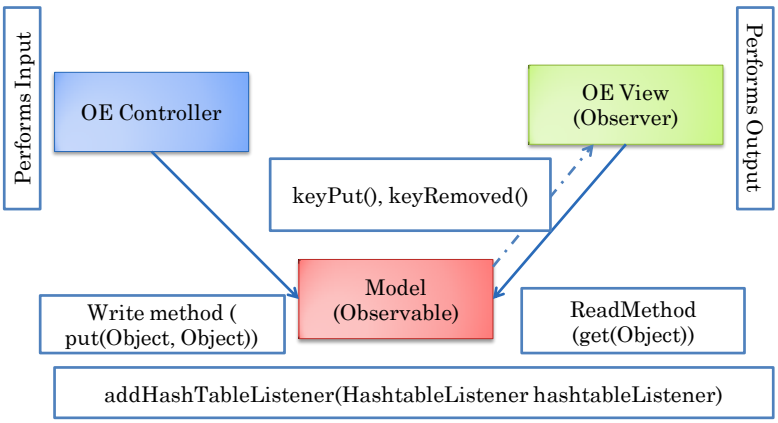


Figure 29 Table Listener Architecture

The interface, HashtableListener, defines the two table notifications.

```
package util.models;
public interface HashtableListener {
    public void keyPut(Object source, Object key, Object
value, int newSize);
    public void keyRemoved(Object source, Object key, int
newSize);
}
```

Figure 30 Table Listener

We also provide a predefined class, HashtableChangeSupport, that keeps track of table listeners and notifies them as illustrated in the code below.

```
import util.models.HashtableChangeSupport;
import util.models.HashtableListener;
public class AnObservableStringToConcertExpense implements
StringToConcertExpense {
Hashtable contents = new Hashtable();
HashtableChangeSupport tableChange = new
AHashtableChangeSupport(this);
    public void addHashtableListener(HashtableListener
    tableListener) {
         tableChange.addHashtableListener(tableListener);
    }
    public void put (String element, ConcertExpense expense) {
        contents.addElement(element);
        tableChange.elementPut(element, expense);
    }
    public void get(String element) {
        return contents.get(element);
    }
    public Enumeration keys() {
        return contents.keys();
    }
}
```

Figure 31 Notifying Table

Predefined Notifying Dynamic Structures

Typically, we do not need to define our own tables and collections – the predefined Java classes Vector and Hashtable suffice. However, these classes do not send notifications. Therefore, we have defined notifying versions of these classes, called AListenableVector and AListenableHashtable, respectively. These two classes support the collection and table notifications described above.

Often we need to create a notifying dynamic sequence of characters. For example, to create a synchronous collaborative editor, we would need to trap each user's insertion, deletion, and modification of a character, and send it to other users. We could use AListenableVector to store this sequence of characters, but that would be inefficient. Therefore, we define another class,

AListenableString, for such a sequence. Unlike a Java String, but like a StringBuffer, AListenableString is mutable. In fact, the implementation of AListenableString uses a StringBuffer. It sends the collection notifications described earlier. We will see below its use.

A distributed model should create instances of these three classes when its logical structure consists of dynamic tables, collections and text.

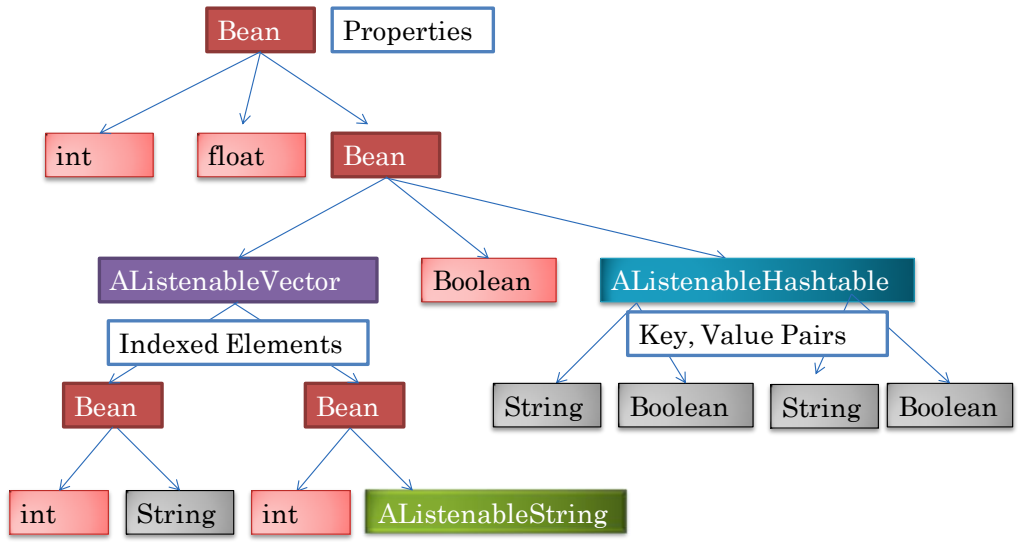


Figure 32 Using predefined dynamic classes

Visualization Examples

To better understand logical structures, notifications, the predefined notifying classes, and ObjectEditor behavior, let us consider some additional examples.

ObjectEditor, in fact, is a tool based on logical structures. Given a model, it creates a widget structure that is isomorphic to the logical structure of the model. This is shown in the figure below.

```
import java.beans.PropertyChangeListener;
public interface ConcertExpense {
     public float getTicketPrice();
     public void setTicketPrice(float newVal);
     public int getNumberOfAttendees();
     public void setNumberOfAttendees(int newVal);
     public float getTotal();
     public void addPropertyChangeListener(PropertyChangeListener l);
}
```

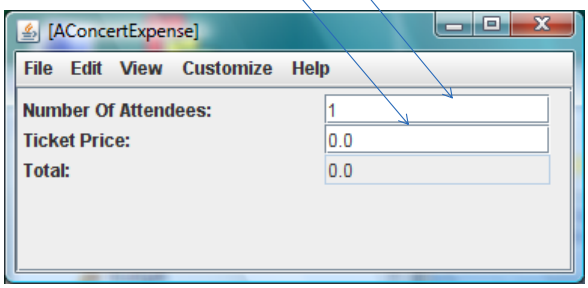


Figure 33 Correspondence between logical and widget structure

To create the widget structure, ObjectEditor must map each logical component to an appropriate widget. It allows the programmer to influence this mapping. Here we will consider some aspects of the default mapping supported by it, which is based on the type of the logical component.

Strings, and as we see in the figure, numbers, are mapped to textboxes. Enums are mapped to combo-boxes, as shown below.

```
public enum ConcertType  {
     Country, Rock, Jazz, Classical
};
```

```
public interface TypedConcertExpense extends ConcertExpense {
    public ConcertType getConcertType();
    public void setConcertType(ConcertType newVal);
}
```

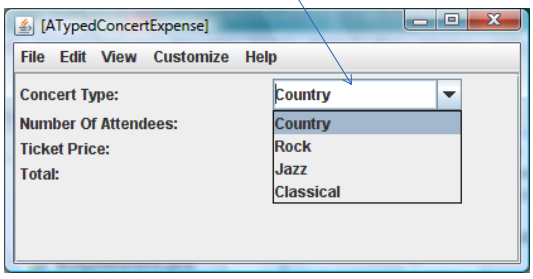


Figure 34 Enums-->ComboBoxes

Here we see the visualization of an instance of the class, ATypedConcertExpense, which implements the interface TypedConcertExpense. This interface defines an enum property, concertType, displayed as a combobox by ObjectEditor, with each enum literal corresponding to a constant defined by the enum.

Boolean values are mapped to checkboxes, as shown in the figure below.

```
public interface CommentedConcertExpense extends TypedConcertExpense {
     public boolean getLongComment();
     public AListenableString getComment();
     public void setComment(AListenableString newVal);
}
```

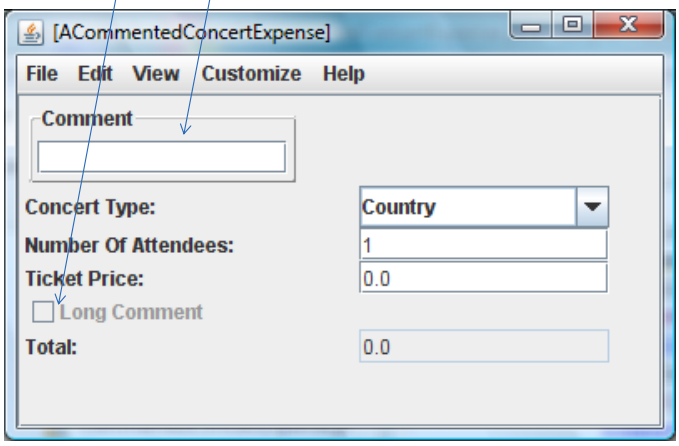


Figure 35 bool--> check box

The object above defines two additional properties: comment of type AListenableString, which allows a user to add a textual comment, and longComment of type bool, which determines if the comment is too long. AListenableString, like String, is mapped to a textbox.

The figure below shows how collections and tables are mapped to widgets. A collection is mapped to a panel, whose components are widgets to which the logical components of the collection are mapped. ObjectEditor chooses a default scheme for aligning the component widgets.

A table's logical structure is mapped to a visual table, with each key, value pair being displayed as a separate row of the visual table. Thus, the logical structures of the key and value are flattened into the row.

```
public static void main(String[] args) {
    TypedConcertExpense firstConcert =  new ACommentedConcertExpense();
    TypedConcertExpense secondConcert = new ACommentedConcertExpense();
    AListenableVector<TypedConcertExpense> list = new AListenableVector();
    list.addElement(firstConcert);
    list.addElement(secondConcert);
    ObjectEditor.edit(list);
    Hashtable<String, TypedConcertExpense> table = new AListenableHashtable();
    table.put("Concert 1", firstConcert);
    table.put("Concert 2", secondConcert);
    ObjectEditor.edit(table);
}
```

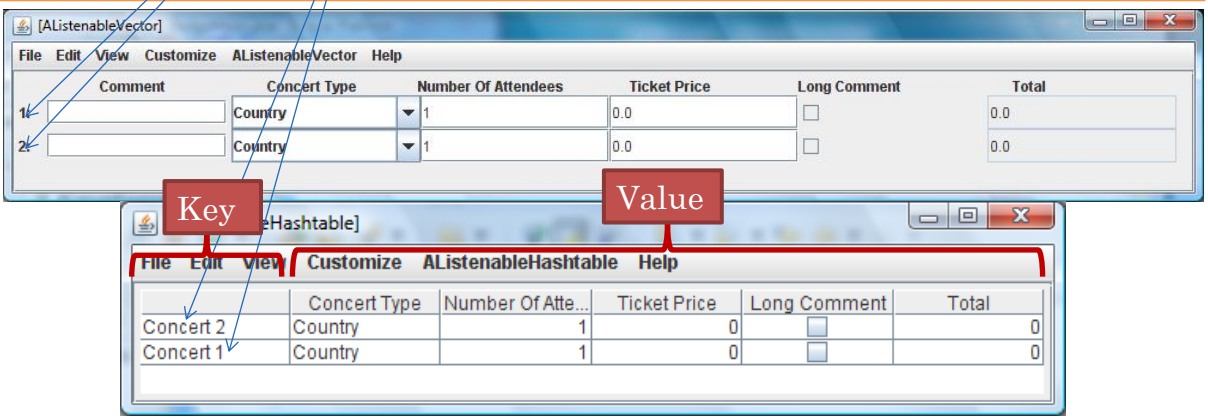


Figure 36 Collection, Table --> Container, Table Widget

In addition to mapping a logical structure to a widget structure, ObjectEditor registers view objects in it as listeners of all notifying nodes in the logical structure, as shown in the figure below. These nodes must, of course, implement standard patterns so the registration methods can be found in them and notifications sent by them can be processed.

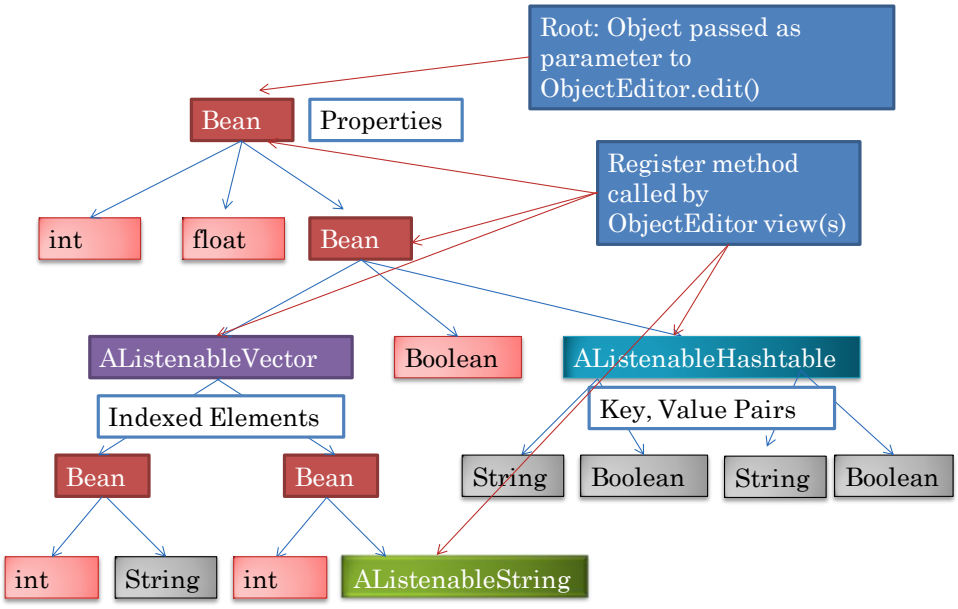


Figure 37 Registering with Notifying Nodes

Let us look at the dynamic behavior of one of the final user-interfaces of the concert expense example to better understand notifications.

Suppose a user edits the numberOfAttendees textbox but does not hit Enter. As shown below, a star is shown next to the textbox, and the total textbox is not updated. This is because the ObjectEditor controller processing this textbox waits for Enter before calling setNumberOfAttendees.

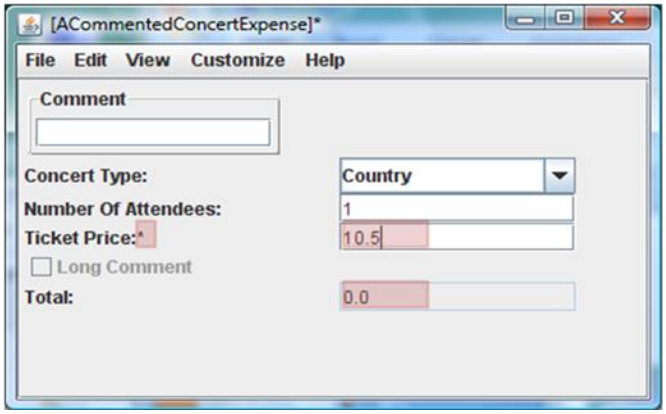


Figure 38 Write method not called until user pressed Enter

As we see below, when the user does hit Enter, the star disappears and the total field is updated by the ObjectEditor view responsible for this field in response to the property change notification sent by the setter method.

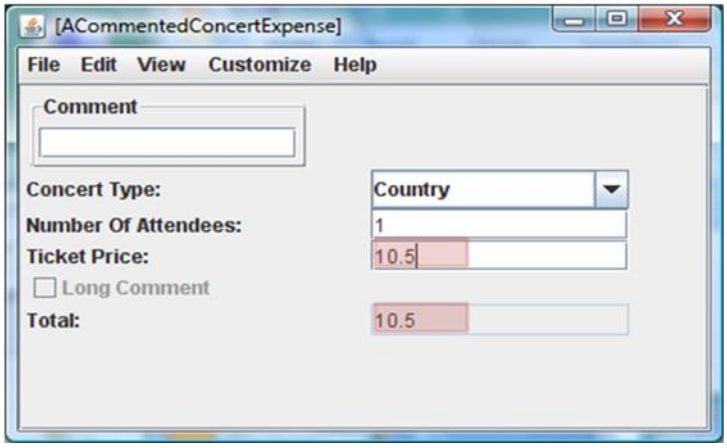


Figure 39 Controller calls setter and view processes notification by updating total

Sometimes the model must be involved as each character is entered by a user. In our example, it incrementally calculates the longComment property as each character is entered. To ensure that it gets these notifications, it made this property an instance of AListenableString and registered itself as a listener of this object. ObjectEditor processes each incremental edit to the textbox to which the instance is mapped by invoking an appropriate write method in it, which in turn, notifies its listeners. Because these write methods are called incrementally, a "*" is not shown next to the edited textbox.

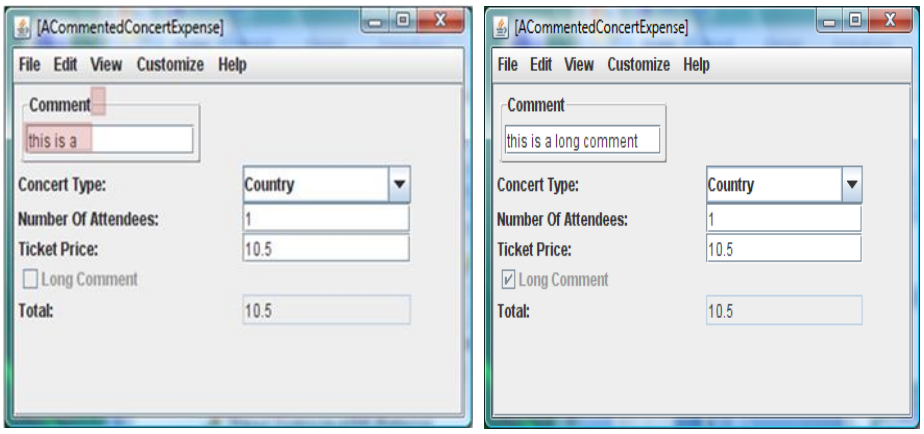


Figure 40 Write methods in AListenableString called on each incremental edit to its text box

The figure below shows what happens when the user presses Enter after having finished editing the comment. At this point the setComment() method in the model is called, which simply clears the comment value. ObjectEditor, which registered itself as a listener of the comment, responds to this event by clearing the text box.

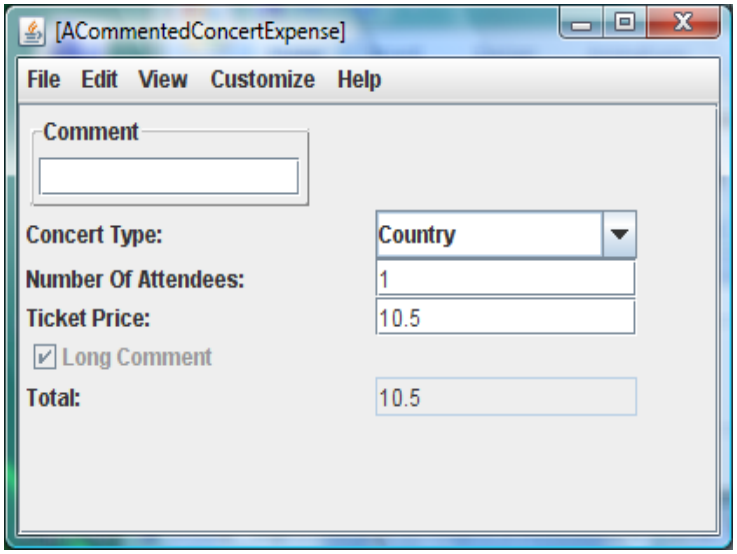


Figure 41 The enter key calls the property setter which clears the comment but not longComment

The following implementation of the class of the (root) model shows how this behavior is implemented.

```
public class ACommentedConcertExpense extends ATypedConcertExpense implements
VectorListener, CommentedConcertExpense {
   AListenableString comment = new AListenableString();
   static int LONG_COMMENT_SIZE = 10;
   boolean longComment = false;
   public ACommentedConcertExpense() {
      comment.addVectorListener(this);
   }
   public boolean getLongComment(){return comment.size() > LONG_COMMENT_SIZE;}
   public AListenableString getComment() {return comment;}
   public void setComment(AListenableString newVal) {
      comment = newVal;
      comment.clear();
   }
   boolean oldLongComment = false;
   public void updateVector(VectorChangeEvent evt) {
      boolean newLongComment = getLongComment();
      propertyChange.firePropertyChange("longCommentType", oldLongComment,
                    newLongComment);
      oldLongComment = newLongComment;
   }
}
```

Figure 42 Implementation of the root model

The model assigns a new instance of AListenableString to the comment instance variable and, in its constructor, registers itself as a VectorListener of this value. Whenever the instance is updated, it calls the updateVector() method in the model, which fires a property change event informing its own listener(s), the ObjectEditor, about the new value of the longComment property. As mentioned later, the setComment() method clears the instance of AListenableString. While ObjectEditor listens to this event, the Sync infrastructure, described later, does not. Therefore, it is more general to call the removeAllElements() method, which is more inefficient as it sends a series of elementRemoved() notifications processed by both ObjectEditor and Sync.

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 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.