The Smalltalk MVC paradigm with pluggable views

Tong Sin Yin
Chow Pui Yee

Preface

This paper is a revision of the technical report entitled *The Smalltalk-80 MVC paradigm with pluggable views* written by Duane Szafron and Brian Wikerson.

Summary

This paper describes the Smalltalk Model-View-Controller (MVC) user interface paradigm using three examples. It is intended for readers who are familiar with Smalltalk, but not with the user interface or graphics classes. It has been written so that readers can learn to build interface objects quickly and in a regular manner. Part of this paper may also be of interest to readers who have used the MVC paradigm already, since it describes a facet of the MVC paradigm which is not well documented in the lterature: pluggable views.

The first example is a simple but complete implementation of a dice using a standard MVC approach. It illustrates the communications which take place between the model, view and controller which represent an interface object. The second example is a re-implementation of a dice using a pluggable view. We also discuss the rationale behind and the importance of pluggable views. The third example is a dice with two different views. It shows how pluggable views can be used in more complex objects.

KEY WORDS: Smalltalk, Model-View-Controller, Pluggable View

1 Introduction

In the Model-View-Controller (henceforth called MVC) approach to user interfaces, user interface objects are represented as a triple of sub-parts: a model, a view and a controller. Each of the these sub-parts is responsible for a different aspect of the interface object's state and behavior.

In general, an interface object represents a real world entity. For example, a bitmap image on the computer screen can be used to represent one of the faces of a dice. In the terminology of Schneiderman's theory of user interfaces [Schneiderman], a user interface object contains computer semantics which relate directly to the task semantics of a real world entity. The task semantics in the dice example consist of a task object (the value of the dice) and a task action (rolling the dice). The computer semantics consist of a computer object (a region of the screen representing the dice, which has six different appearances depending on the result of rolling) and a computer action (changing the appearance of the region). In the MVC paradigm the computer semantics are subdivdied into: visual information, interaction information and non-visual, non-interaction information.

1.1 The Components of the MVC paradigm

The *model* of an interface object is the sub-part containing all of the non-visual and non-interactive computer semantic informaton. In general, the model of an interface object contains an abstract representation of the state of the object. Its protocol contains messages for changing and revealing its state.

In the case of the dice, the model would consist of a six-value state which ranges from one to six, a message to set the initial dice value, a message to roll the dice and a message to reveal the state. However, the model would not contain information pertaining to how the dice is displayed or how a user could initiate a change of its state.

The *view* of an interface object is the sub-part containing all of the visual computer semantic information. In general, a view might contain: display objects, size and location information, and highlighting information. Its protocol may contain messages for displaying itself, changing its size and location, determining if a point is within itself, and highlighting and de-highlighting aspects of itself.

In the case of the dice, the view would consist of: a set of images representing the six faces of a dice and messages for displaying itself.

The *controller* of an interface obejct is the sub-part that is responible for capturing the user's attmepts to manipulate the object. In general, a controller's protocol usually contains messages for asking if it would like to become active, asking if it would like to remain active, and for initiating and terminating control when user actions (mouse clicks or key pressed) are taken.

In the case of the dice, the controller would also contains protocol to roll the dice (change the state of the model) when the user clicks the left mouse button within its view.

1.2 The MVC Paradigm in Smalltalk

An instance of the class **ControlManager** is used to pass control among all of the existing controllers since in the Smalltalk implementation of MVC, only one controller can be in control at any one time. The controller which is in control is said to be active.

The MVC paradigm differs from the event driven paradigm in that, once a controller becomes active, control is not relinquished until the controller is finished or until a special action (for example the user typing a control-C) interrupts the controller. When a controller starts up, it usually performs some initial action. It then performs a control loop in which it repeatedly checks to see if it wants to remain active, and if so it

performs its control activity. When a controller exits its control loop, it generally performs some termination action.

The three sub-parts: model, view and controller are not really combined into a single interface object. In the most common situation, an interface object is represented by three seperate objects: one model, one view and one controller as shown in Figure 1.

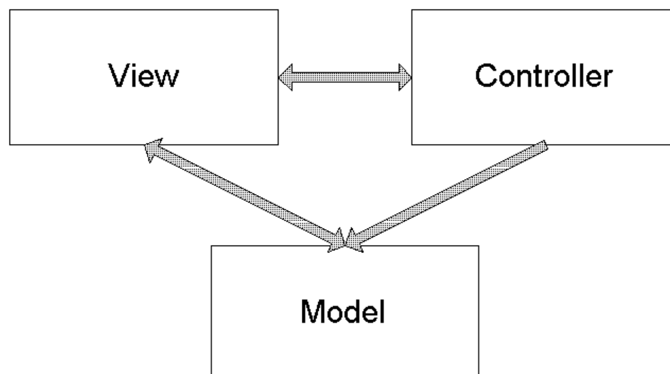

Figure 1: Three separate objects: model, view and controller

It should be noted that many of the interface objects in the Smalltalk image do not exactly divide their responsibilities in this way. For example, **TextEditorController** objects perform many of the tasks which should be performed by **TextEditorView** under this classification. However, these tasks have been moved to **TextEditorController** for efficiency reasons. A **TextEditorController** is a sub-class of the class **ParagraphEditor**, and a **ParagraphEditor** acts as both a view and controller. For example, it echoes its own characters to the screen as they are typed, and performs editing operations on its text, which it stores directly.

In many cases, the structure of an interface object is more complex than a simple triple: one model, one view and one controller. The simplest variation is a hierarchical interface object where one interface object contains others. When an interface object is composed of sub-objects in a hierarchical manner, the hierarchy is represented in the views of the objects. It is reasonable to use the views to present the hierarchy since the hierarchy is usually a visual one.

For example, consider a double-dice which is built by using two dice. We can either roll the dice separately or roll the two dice together. The views of the two dice are called *sub-views* of the double-dice view and the double-dice view is called the *super-view* of each dice view. Logically, we can regard such an object as three MVC triples: one triple for each dice and one triple for the composite object as shown in figure 2.

Although we could use three separate models for the double-dice and the composite object, it is also possible to use three view-controller pairs and a single model. The model would contain two six-value state representing the position of the double-dice. This alternate representation requires the use of pluggable views which will be discussed later in this paper.

1.3 A Simple Model for MVC Communication

Since the sub-parts of an interface object are represented by three distinct objects, they must communicate by sending messages to each other.

The standard communication model is shown in Figure 3. It consists of four separate communicatons.

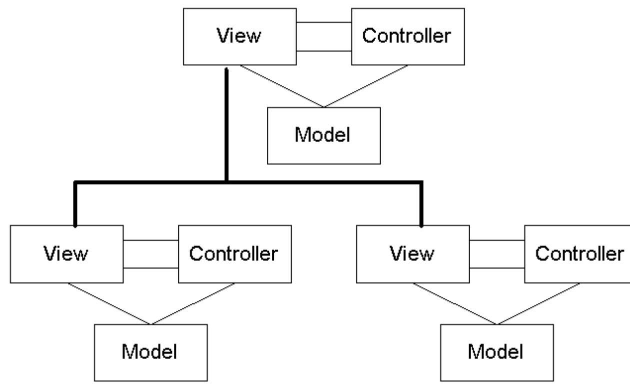

Figure 2: A hierarchical interface object

1. The user performs an action.
2. The controller interprets this action as a request to change the state of the model. The controller informs the model.
3. The model changes its state and then informs the view that it has changed.
4. The view then asks the model for its current state and updates itself accordingly.

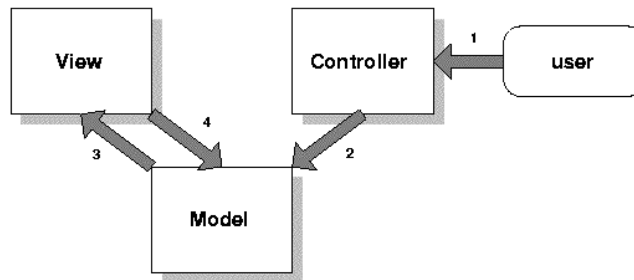

Figure 3: The Standard MVC communication model

When a model changes its state, it informs its view by using the concept of object dependency. Conceptually, if an object, A, is a dependent of an object, B, then whenever object B changes itself, it sends a message to object A to update itself. In this case, the view is a dependent of its model so the communication labelled, 3, in Figure 2 is achieved through this dependency.

This communication model will be used to implement MVCs which do not use pluggable views. When we introduce pluggable views later, it will be replaced by a different communication model.

2 Some Required classes and protocols

To understand how MVCs are implemented it is necessary to understand six classes: **Model**, **View**, **Controller**, **ScheduledWindow**, **CompositePart** and **Wrapper**. The classes **View** and **Controller** are important because all views and controllers are defined as subclasses of them respectively. The classes **ScheduledWindow**, **CompositePart** and **Wrapper** are used as container of the views.

2.1 Model

A **Model** is an object with one instance variable. The variable *dependents* stores a collection of objects which are dependents of the **Model**. In fact, in Smalltalk, every object can has its own dependents. In class **Object**, dependents are stored in the global variable, *DependentCollection*.

In the category 'private', two instance methods are provided to manipulate the dependents stored in a model. These two methods override the ones declared in class **Object**. The method `#myDependents:` sets the dependents of a **Model** while the method `#myDependents` returns the dependents of a **Model**.

2.2 View

Class **View** is the subclass of the class **DependentPart**. The instance variable *controller* is declared in class **View** and the instance variable *model* is declared in class **DependentPart**, which is the superclass of class **View**.

Class **View** does not have its own instance creation method. The one which is most commonly used is the class method `#model:`. This message takes a model as its argument. Subsequently, the instance variable *model* is set to this argument in the instance method `#setModel:` which is declared in the category 'private'. If a View is not created by the class method `#model:`, we need to explicitly set the variable *model* by sending the message `#model:` (which is declared in the category 'accessing' of class **DependentPart**) with a model as the sole argument. Besides setting the variable, the **View** is added as a dependent of **Model**. This is done in the instance method `#setModel` of class **DependentPart**. Simply setting a **Model** to the instance variable *model* directly results in failure to create the dependency between **View** and **Model**.

The category 'controller accessing' contains four methods: `#controller`, `#controller:`, `#defaultControllerClass` and `#defaultController`. Each view should have its corresponding controller. The first two methods simply return the controller of a **View** and set its controller respectively. The third method returns the class of the default controller. The forth method returns an instance of the default controller class. The default instance creation method used in `#new`.

2.3 Controller

A **Controller** is an object with three instance variables. The first two are *model* and *view* which are used to communicate with the other sub-parts. The variable *sensor* is an instance of **InputSensor** which is used to handle user action such as mouse click and keyboard pressed. Usually, the initialization (sometimes also the instance creation) of a **Controller** is done by the view.

The categories 'model access' and 'view access' provide methods to set and return the value of *model* and *view* respectively. The category 'cursor' contains the method `#viewHasCursor` only. This method returns true if the cursor point of the receiver's sensor (usually, we are referring to the mouse) is within the boundary of the receiver's view.

The category 'control defaults' contains four methods: `#controlActivity`, `#controlToNextLevel`, `#isControlActive` and `#isControlWanted`. The method `#controlActivity` is called repeatedly by `#controlLoop` (see next paragraph) when the controller is active. This method is usually overrode to perform appropriate actions. For example, we can detect which mouse button is being pressed and ask the model to change some

of its states or perform an action. The method `#controlToNextLevel` ask the controller's view if any of its subviews wants control. If so, the control is passed on the appropriate one. The method `#isControlActive` returns true if the receiver is active. This method can be overrode to change the condition to determine whether a controller is active or not. The method `#isControlWanted` returns true if the controller wants control. The default behavior is to answer true if `#viewHasCursor` returns true. Again, we can override it to suit our application.

The following is the method `#controlToNextLevel` :

```
controlToNextLevel

     | aView |
     aView := view subViewWantingControl.
     aView == nil ifFalse: [aView startUp]
```

The method `#subViewWantingControl` is defined in the category 'control' in class **VisualPart**. The default behavior of this method is to return *nil* as we assume that the view does not have any subview. Therefore, if the view needs to pass control to its subviews, we have to override this method in the subclasses of **View**.

The category 'basic control sequence' contains six methods: `#poll, #startUp, #checkForEvents, #controlLoop, #controlInitialize` and `#controlTerminate`. The method `#poll` looks for any activity. If there is no activity for a certain period of time, it will wait on semaphore. The method `#checkForEvents` asks the window in which is the container of the view to check for events. The method `#startUp` gives control to the controller. It then sends three messages to itself in the following order: `#controlInitialize, #controlLoop` and `#controlTerminate`. The method `#controlInitialize` initializes the controller. Sometimes it is re-defined in subclasses to perform specific action when the controller gains control. In the method `#controlLoop`, the controller sends `#isControlActive` to itself. If true is returned, the controller sends itself the message `#controlActivity`. It is repeated until `#isControlActive` returns false. The method `#controlTerminate` is sometimes re-defined if some specific actions are needed to be performed when the controller gives up control.

The default behavior of the method `#isControlActive` is that the controller still wants control if the cursor is in its view and the blue mouse button is not pressed.

```
isControlActive

    ^self viewHasCursor and: [self sensor blueButtonPressed not]
```

In **Smalltalk**, the mouse buttons are referred as <red>, <yellow> and <blue>. The reason is that we are not always working with a 3-button mouse. Therefore, using notation like <left>, <middle> and <right> does not make sense all the time. The following table shows the naming convention of the mouse button in **Smalltalk**.

color	*position*	*function*
red	left	select
yellow	middle	operate
blue	right	window

The <`select`> button is used to *select* a window location or a menu item, position the text cursor, or highlight text. The <`operate`> button brings up a menu of *operations* that are appropriate for the current view or selection. This menu is referred to as the <operate> menu. The <`window`> button brings up the menu of actions that can be performed on any **Smalltalk** window, such as move and close.

2.4 ScheduledWindow

A **ScheduledWindow** is an object with thirteen instance variables. The six of them which are the most important will be discussed. *label* is a String. It is the title of the window which appears at the top of it. *minimumSize* and *maximumSize* are the minimum and maximum size respectively of the display box of a ScheduledWindow. *component* is a visual component, an object that is installed and displayed in the **ScheduledWindow**. The visual component of a **ScheduledWindow** can be an instance of **DependentPart**, **CompositePart** , **Wrapper** or their subclasses (e.g. **View** is a subclass of **DependentPart**. We can also install an object such as **Image** or a **ComposedText** as its component. Each **ScheduledWindow** has its own controller and it is stored in the variable *controller* (an instance of **StandardSystemController** or its subclasses). A **ScheduledWindow** may have a model associated with it, and it is stored in *model*.

The category 'accessing' contains many methods to set the attributes of a **ScheduledWindow**. The category 'scheduling' provides various methods to open a **ScheduledWindow**. The simplest way to create and open a window is shown in the following code:

```
ScheduledWindow new open
```

This window is created with default inside color, no label, default minimum size and no restriction of its maximum size. We can create a window with more features by the following code:

```
| aWindow |

aWindow := ScheduledWindow new.
aWindow label: 'Hello World!'.
aWindow minimumSize: 200@100.
aWindow maximumSize: 200@100.
aWindow insideColor: ColorValue white.
aWindow open.
```

Figure 4: An instance of ScheduledWindow

2.5 CompositePart and Wrapper

The instance variable *component* of class **ScheduledWindow** must be a single visual components. Thus, we need an intermediate object in order to hold two or more components of the same window. The instance variable *component* of class **CompositePart** is an ordered-collection of **Wrapper**. Each wrapper holds a visual component.

Several methods are provided to add a visual component to an instance of **CompositePart** (or one of its subclasses). One of them is `#add:at:`. The first argument is an instance of any subclasses of **VisualComponent** (e.g. **View**) and the second argument is a **Point**. The visual component is added on a **TranslatingWrapper**. The second argument of `#add:at:` specifies the position of the top left corner of

the **TranslatingWrapper**. Other methods for adding components can be found in the category 'adding-removing'.

A **Wrapper** holds a **VisualComponent** to which it forwards messages. Although class **Wrapper** is not an abstract class, instances of it are seldom used. Usually, we use its subclasses such as **Translating-Wrapper**, **BoundedWrapper** and **BorderedWrapper**. An instance of **Wrapper** is created by the class method #on: with a **visual component as the argument. For the subclasses of Wrapper, other methods are used.**

class	*instance creation method*
Wrapper	**#on:**
Translatingwrapper	**#on:at:**
BoundedWrapper	**#on:in:**
BorderedWrapper	**#on:in:**

2.6 Protocol for dependent objects

The concept of dependent objects is implemented in Smalltalk by providing various instance methods in the protocol for class Object.

The method `#addDependent` adds the argument object as a dependent of the receiver while the method `#removeDependent:` removes the argument object from the dependents list of the receiver. The method `#dependents` returns an OrderedCollection (a collection of objects that can be accessed individually by an integer index) of objects dependent on the receiver. The above methods are defined in the category 'dependent access'.

Each object should send itself the message, `#changed:` if some aspect of itself changes, where the aspect which changed is the argument. The default behavior of this message is to send `#update:` messages to each of its dependents with the aspect as the argument. The message `#update:` should be implemented by all dependents to take whatever action is necessary when the given aspect of the object on which they depend has changed.

When the changes of state of the model do not need to be parameterized by any aspect, we can use the method `#changed`. The method `#changed:` and its variation is defined in the category 'changing' while the method #update: and its variation are defined in the category 'updating'.

2.7 Redisplaying a view

There are two basic situations in which a view must redisplay itself:

- Model updates – the model changes in a way that affects the view
- Window damage repair – the window is refreshed, an overlapping window is moved ... etc.

It's very simple to equip a view with the means to handle either kind of event:

- Make sure the view responds to `#update:`.
- Make sure the view responds to `#displayOn:` by displaying itself on the graphics context that is passed as an argument. The view should, of course, use the current state of the model.

The default behavior of the method `#update:` (which is defined in the category 'updating' of the class DependentPart) is to send the message `#invalidate` to itself. To express its effect in

simplest terms, this method simply redisplay the area spanned by a visual component.

It should be noted that we should not send the message `#displayOn:` to the view itself in the method `#update:`. By doing this, the view will end up displaying itself twice.

3 MVC for a Dice

In this section we will describe the model, view and controller of a simple dice. The dice can have value from 1 to 6 inclusively. If the dice is pressed, the dice is rolled and a new value will be displayed. The value of the dice is represented by the same number of black spots. Figure 5 shows the dices of six different values embedded in ScheduledWindows.

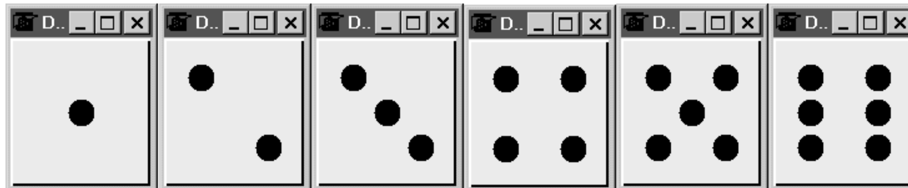

Figure 5: A visual representation of the six states of a dice

Following convention, the model, view and controller classes are named: Dice, DiceView and DiceController respectively. To open the dice, try this:

```
DiceView openOn: Dice new.
```

3.1 Class Dice

The class Dice is a subclass of the class Model. It has a single instance variable, *value*, which has an integer value between 1 and 6.

```
Model subclass: #Dice
     instanceVariableNames: 'value '
     classVariableNames: ''
     poolDictionaries: ''
     category: 'Example-Dice'
```

3.1.1 Class Messages for Dice

There is a single class message, `#new`, which creates a new instance and initializes its value with 1.

Category 'instance creation'

```
new
     "Answer with an initialized instance of the receiver"

     ^super new initialize
```

3.1.2 Instance Messages for Dice

There are three categories of instance messages: 'instance-release', 'accessing' and 'modifying', each containing a single instance message. The messages are: `#initialize`, `#roll` and `#value`, which initializes an instance to have value 1, changes the value randomly and answers the current value respectively.

Category 'initialize-release'

```
initialize
     "Initialize the variables of the receiver"

     value := 1
```

Category 'accessing'

```
value
     "Answer with the current value held by the receiver"

     ^value
```

Category 'modifying'

```
roll
     "Set my value to be a random integer between 1 and 6"

     | rand |
     rand := Random new.

     value := (rand next * 6 + 1) truncated.

     self changed
```

In the method for `#roll`, the receiver sends itself the message `#changed` so that all of its dependents (in this case its view(s)) will be sent the message `#update:`.

3.2 Class DiceView

The class DiceView is a subclass of the class View with one new instance variables *images*, which stores the six different value images of dice.

```
View subclass: #DiceView
     instanceVariableNames: 'images '
     classVariableNames: ''
     poolDictionaries: ''
     category: 'Example-Dice'
```

3.2.1 Class Messages for DiceView

There are two class messages for DiceView and both are instance creation messages.

Category 'instance creation'

The first instance creation message is a general message which is used for creating a DiceView. It simply creates a new instance of DiceView and sends it an initialization message which passes on the model.

```
model: aDice
     "Answer and initialize a new instance of DiceView."

     ^super new model: aDice.
```

The second instance creation message creates an instance of the class ScheduledWindow, creates a DiceView, and embeds the DiceView in the ScheduledWindow. It then sends the message #open to the ScheduledWindow so that the view is displayed on the screen.

```
openOn: aDice
     "Create an instance of the receiver in a ScheduledWindow"

     | diceView window |
     window := ScheduledWindow
             model: aDice
             label: 'Dice'
             minimumSize: 100@120.
     window maximumSize: 100@120.

     diceView := self model: aDice.
     window component: diceView.
     window open
```

3.2.2 Instance Messages for DiceView

There are three categories of instance messages defined in the class DiceView: 'initialize-release', 'controller access' and 'displaying'.

Category 'initialize-release'

Category 'initialize-release' contains a single initialization message which sets the instance variable *model* (in super class) to the argument passed and stores the six images in the instance variable *images* in the form of a dictionary. Each integer value has its corresponding image.

```
model: aDice
     "Initialize the image of dice."

     super model: aDice.

     images := Dictionary new.
     images at: 1 put: (ImageReader fromFile: 'one.bmp' asFilename) image.
     images at: 2 put: (ImageReader fromFile: 'two.bmp' asFilename) image.
     images at: 3 put: (ImageReader fromFile: 'three.bmp' asFilename) image.
     images at: 4 put: (ImageReader fromFile: 'four.bmp' asFilename) image.
     images at: 5 put: (ImageReader fromFile: 'five.bmp' asFilename) image.
     images at: 6 put: (ImageReader fromFile: 'six.bmp' asFilename) image.
```

Category 'controller access'

Category 'controller access' contains a single message which returns the class of controller associated with this view class. In this case, the default controller class is DiceController.

```
defaultControllerClass
     "Answer the class of the controller associated with me."

     ^DiceController
```

Category 'displaying'

Category 'displaying' contains a single message which displays a suitable image according to the value of the dice.

```
displayOn: aGraphicsContext
     "Display an image."

     | currentValue |
     currentValue := model value.
     (images at: currentValue) displayOn: aGraphicsContext.
```

3.3 Class DiceController

The class DiceController is a subclass of the class Controller with no new instance variables or class messages. There is only one instance message which is a refinement of a message in class Controller.

```
Controller subclass: #DiceController
     instanceVariableNames: ''
     classVariableNames: ''
     poolDictionaries: ''
     category: 'Example-Dice'
```

3.3.1 Instance Messages for DiceController

Category 'control defaults'

Category 'control defaults' contains a single message which defines the control activity when the controller is active. If the red button [1] is pressed, it asks the model to roll and then waits for the button to be released.

```
controlActivity
     "If my <select> button is pressed, roll the model,
      then wait for the button to be released"

     self sensor redButtonPressed
             ifTrue: [
                     model roll.
                     self sensor waitNoButton.
             ].

     ^super controlActivity
```

[1] Red button is the left mouse button in UNIX and Microsoft environment.

4 Pluggable Views

A *pluggable* view is a view which communicates with its model through messages whose selectors are stored in the view. There are two reasons for using pluggable views, one pragmatic and one aesthetic.

4.1 The Rationale for Pluggable Views

4.1.1 Pragmatic Reason

The pragmatic reason for using pluggable views is that if they are used, then *multiple* view-controller pairs of the same type can be used on *different aspects* of the same model.

For example, consider a two-dice where the state of each dice is independent. A single model can be used, whose state consists of two, six value variables. Each dice can be represented by the view-controller pair of a dice. When one of the dices is rolled, its view must inform the model. However, if both dices sent the *same* message to the model, then the model would not be able to tell which dice was rolled. If however, each dice sent a *different* message then the model would be able to differentiate between the two dice and updates its state correctly.

In order for this technique to work, it is necessary that the view-controller pair communicates with the model using message selectors which are stored in instances of the view-controller pairs. Since both the view and the controller communicate with the model, it seems necessary to store some of these message selectors in the view and some in the controller. Instead, we localize the pluggability of the view-controller pair in the view. We do this by modifying the communications model shown in Figure 3 so that all communication with the model is done by the *view*. For this reason, we use the term *pluggable view* instead of a term like pluggable view-controller pair. The communication model for pluggable views is shown in Figure 6.

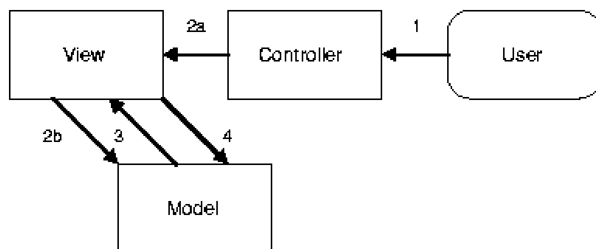

Figure 6: The pluggable MVC communication model

1. The user performs an action.
2. (a) The controller interprets this action as a request to change the state of the model. The controller informs the view.
   (b) The view informs the model using the *action selector* stored in itself.
3. The model changes its state and then informs the view that it has changed.
4. The view then asks the model for its current state using the *state selector* stored in itself and updates itself accordingly.

As a second example where pluggable views are advantageous, consider a browser which contains two lists, each in a separate view. If a user adds an element to the first list, then the controller could send a message like:

```
model addElement: newElement
```

to its model. However, if the user added an element to the second list, then the same message could not be used. If it was, then the model would not be able to determine which list to use.

One way to solve this problem is to use view-controller pairs from different classes for each list. However, this would result in a needless proliferation of classes if the view-controller pairs were identical except for this one difference. A better solution is to use two different instances of the same pluggable view class.

For example, if there are two lists in a programming environment browser that represent modules which are organized into categories, then the message

```
model perform: actionMsg with: element
```

could be invoked by both views. In one view, *actionMsg* and *element* would be bound to `#addCategory:` and an object representing a new category respectively. In the other view, they would be bound to `#addModule:` and an object representing a new module respectively.

4.1.2 Aesthetic Reason

There is an aesthetic advantage to use pluggable views, even if multiple instances of the same view class are not going to be used by one model. The advantage occurs when a view is built for one model, but then is used by another model at a later date. This *re-usability of classes* is one of the major goals for object-oriented design.

In this case, if a pluggable view is not used, the second model to use the view must use the message selector which was defined when the original MVC was constructed. However, this message selector may not be appropriate for the new model, either because it is already being used for another purpose by the model, or because its meaning was specialized to apply to the original model.

4.2 Adaptors

There is an alternative to pluggable views referred to as *adaptors*. It involves the creation of a new object called an adaptor which is responsible for communications between the model and the view-controller pair. In this case, the view would be a dependent of the adaptor instead of the model. The communication model shown in Figure 7 uses an adaptor.

If an adaptor is used, the pluggable information is encapsulated in the adaptor. Because both the view and controller want to communicate with the model directly, neither is really appropriate for maintaining the pluggable information. The role of the adaptor is to act as if it were the *model*, while adapting the protocol sent by the view and controller to that understood by the real model. The adaptor encapsulates the pluggable information.

Adaptors have two distinct advantages over pluggable views. First, the presence of the adaptor can be ignored when understanding the roles of the model, view and controller. The mechanism for making the view-controller pair pluggable can also be ignored, simplifying the

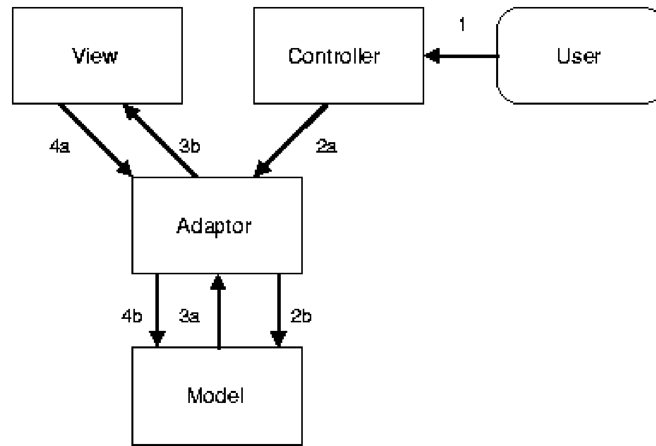

Figure 7: The pluggable MVC model with an adaptor

MVC paradigm.

Second, because adaptors act just like the models they represent, the concept of adaptors can be applied to any existing view-controller pair without rewriting the pair. Pluggable views require at least the creation of two subclasses, which usually involves a great deal of overriding of methods to use the pluggable selectors rather than accessing the model directly.

5 A Dice with a Pluggable View

A pluggable implementation of a dice can be constructed by replacing the explicit messages `#value` and `#roll` in the classes DiceView and DiceController by messages which use selectors stored in instances of DiceView. The selectors are stored in two new instance variables, *actionSelector* and *stateSelector*. That is, the non-adaptor form of pluggable views shown in Figure 6 is used.

Open the dice with pluggable view by:

```
PluggableDiceView openOn: (Dice new) action: #roll state: #value.
```

To make things clear, we call the updated DiceView and DiceController as PluggableDiceView and PluggableDiceController respectively. These new classes are the same as DiceView and DiceController respectively except for the changes described below.

5.1 Changes in PluggableDiceView

Two new instance variables *actionSelector* and *stateSelector* are added. Also the instance creation messages and the initialization message must be changed to account for these two new instance variables.

```
View subclass: #PluggableDiceView
     instanceVariableNames: 'images actionSelector stateSelector '
     classVariableNames: ''
     poolDictionaries: ''
     category: 'Example-Dice'
```

5.1.1 Changes in Class Messages for PluggableDiceView

Category 'instance creation'

The messages for instance creation are changed. Two more arguments of action selector and state selector are passed.

```
model: aModel action: actionMsg state: stateMsg
     "Create a PluggableDiceView instance."

     ^self new model: aModel action: actionMsg state: stateMsg.


openOn: aDice action: actionMsg state: stateMsg
     "Create an instance of the receiver in a ScheduledWindow"

     | diceView window |
     window := ScheduledWindow
             model: aDice
             label: 'Dice'
             minimumSize: 100@120.
     window maximumSize: 100@120.

     diceView := self model: aDice action: actionMsg state: stateMsg.

     window component: diceView.
     window open
```

5.1.2 Changes in Instance Messages for PluggableDiceView

Category 'instance-release'

Two more arguments are passed to this message. It initializes the instance variables *actionSelector* and *stateSelector*.

```
model: aDice action: actionMsg state: stateMsg
     "Initialize the instance variables of receiver."

     self model: aDice.

     images := Dictionary new.
     images at: 1 put: (ImageReader fromFile: 'one.bmp' asFilename) image.
     images at: 2 put: (ImageReader fromFile: 'two.bmp' asFilename) image.
     images at: 3 put: (ImageReader fromFile: 'three.bmp' asFilename) image.
     images at: 4 put: (ImageReader fromFile: 'four.bmp' asFilename) image.
     images at: 5 put: (ImageReader fromFile: 'five.bmp' asFilename) image.
     images at: 6 put: (ImageReader fromFile: 'six.bmp' asFilename) image.

     actionSelector := actionMsg.
     stateSelector := stateMsg.
```

Category 'displaying'

The displaying message must be modified to use the message selector stored in the instance variable, *stateSelector*.

```
displayOn: aGraphicsContext
     "Display an image."

     | currentValue |
     currentValue := model perform: stateSelector.
     (images at: currentValue) displayOn: aGraphicsContext.
```

Category 'updating state'

A message must be added which the controller can use it to relay the fact that the model should change its state.

```
changeState
     "Ask model to change state."

     model perform: actionSelector.
```

Category 'controller access'

The default controller class in pluggable view is changed to PluggableDiceController.

```
defaultControllerClass
     "Answer the class of the controller associated with me."

     ^PluggableDiceController
```

5.2 Changes in PluggableDiceController

Category 'control defaults'

Finally, the method in class DiceController for `#controlActivity` must be modified to use the new message `#changeState` in PluggableDiceView to initiate a state change instead of informing the model directly.

```
controlActivity
     "If my <select> button is pressed,
      tell pluggable view to inform model changing state,
      then wait for the button to be released"

     self sensor redButtonPressed
             ifTrue: [
                     view changeState.
                     self sensor waitNoButton
             ].

     ^super controlActivity
```

6 A Double-Dice

To illustrate how pluggable views are used, we present a double-dice which uses two view-controller pairs which are dice and a single model. The double-dice contains two independent dice view placed in a ScheduledWindow. There are totally thirty six (6x6) different states.

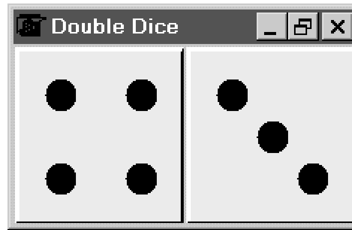

Figure 8: An instance of a double-dice

Following convention, the model and view classes are named: DoubleDice and DoubleDiceView. However, there is no DoubleDiceController since the class Controller is sufficient. Note that although instances of the class DoubleDice are being used as models, instances of other classes could be used as well since the DoubleDiceView is pluggable. The only real requirement of the model is that it can represent thirty six distinct states.

With the use of pluggable views, the double-dice can be operated in two modes. The two dice can roll independently or roll at the same time.

To open a double-dice which the two dice roll independently:

```
DoubleDiceView
     openOn: (DoubleDice new)
     action: #(#leftRoll #rightRoll)
     state: #(#leftValue #rightValue).
```

To open the double-dice which the two dice roll at the same time:

```
DoubleDiceView
     openOn: (DoubleDice new)
     action: #(#bothRoll #bothRoll)
     state: #(#leftValue #rightValue).
```

6.1 Class DoubleDice

The class DoubleDice is a subclass of the class Model. It has two instance variables, *leftValue* and *rightValue*, which store the value of left and right dice respectively.

```
Model subclass: #DoubleDice
     instanceVariableNames: 'leftValue rightValue '
     classVariableNames: ''
     poolDictionaries: ''
     category: 'Example-Dice'
```

6.1.1 Class Messages for DoubleDice

There is a single class message, `#new`, which creates a new instance and initializes its state.

Category 'instance creation'

```
new
     "Answer with an initialized instance of the receiver"

     ^super new initialize
```

6.1.2 Instance Messages for DoubleDice

There are three categories of instance messages: 'initialize-release', 'accessing' and 'modifying', where the first category contains one message and the other two categories contain two messages each. The messages are: #initialize, #leftValue, #rightValue, #leftRoll and #rightRoll.

Category 'initialize-release'

The first message initializes the values of two dice to be 1.

```
initialize
     "Initialize the variables of the receiver"

     leftValue := 1.
     rightValue := 1.
```

Category 'accessing'

This category contains two messages which answer the value of the left and right dice respectively.

```
leftValue
     "Answer with the current value of the left dice held by the receiver"

     ^leftValue


rightValue
     "Answer with the current value of the right dice held by the receiver"

     ^rightValue
```

Category 'modifying'

This category contains three messages which roll the left, right, and both dice. The left or right dice will then have a new value between 1 to 6. Its dependents are asked to update.

```
leftRoll
     "Set my first value to be a random integer between 1 and 6"

     | rand |
     rand := Random new.

     leftValue := (rand next * 6 + 1) truncated.

     self changed
```

```
rightRoll
     "Set my second value to be a random integer between 1 and 6"

     | rand |
     rand := Random new.

     rightValue := (rand next * 6 + 1) truncated.

     self changed


bothRoll
     "Set the two values to be a random integer between 1 and 6"

     | rand |
     rand := Random new.

     leftValue := (rand next * 6 + 1) truncated.
     rightValue := (rand next * 6 + 1) truncated.

     self changed
```

6.2 Class DoubleDiceView

The class DoubleDiceView is a subclass of the class View with no new instance variables.

```
View subclass: #DoubleDiceView
     instanceVariableNames: ''
     classVariableNames: ''
     poolDictionaries: ''
     category: 'Example-Dice'
```

6.2.1 Class Messages for DoubleDiceView

There is only one class message for DoubleDiceView. It is an instance creation message. It creates the two dice-views and put them in the ScheduledWindow.

Category 'instance creation'

```
openOn: aDice action: actionMsgArray state: stateMsgArray
     "Create an instance of the receiver in a ScheduledWindow"

     | window leftView leftWrapper rightView rightWrapper composite |
     leftView := PluggableDiceView
             model: aDice
             action: (actionMsgArray at: 1)
             state: (stateMsgArray at: 1).
     leftWrapper := BorderedWrapper on: leftView in: (0@0 corner: 0.5@1.0).

     rightView := PluggableDiceView
             model: aDice
             action: (actionMsgArray at: 2)
             state: (stateMsgArray at: 2).
     rightWrapper := BorderedWrapper on: rightView in: (0.5@0 corner: 1@1).

     composite := CompositePart new.
```

```
composite addWrapper: leftWrapper.
composite addWrapper: rightWrapper.

window := ScheduledWindow
        model: aDice
        label: 'Dice'
        minimumSize: 230@120.
window maximumSize: 230@120.
window component: composite.
window open
```

In DoubleDiceView, there are two dice-views in the same window. Both of them are PluggableDiceView. In order to place them in the same window, we need to add them in wrappers which will become the component of a CompositePart. This CompositePart is embedded in the ScheduledWindow as a component.

7 Conclusion

We have described the Smalltalk Model-View-Controller (MVC) user interface paradigm using three examples. The first example is a simple but complete implementation of a dice. It illustrates the basic communication pattern of the MVC paradigm with non-pluggable views.

The second example re-implements the dice using a pluggable view. It illustrates how the communication pattern changes when pluggable views are used.

The third example consists of a double-dice which uses the pluggable view implementation of the dice. It shows how pluggable views can be used to associate multiple view-controller pairs with different aspects of a common model.

In addition to the three examples, we have discussed the rationale for pluggable views, the reasons why they are important and two different communication patterns for implementing them. It is our belief that all views should be pluggable so that they may be easily re-used in multiple applications.

References

[Goldberg, Robson] Goldberg, A. and D. Robson, *Smalltalk-80: The Language and its Implementation*, Addison-Wesley, Reading 1983.

[Goldberg] Goldberg, A., *Smalltalk-80: The Interactive Programming Environment*, Addison-Wesley, Reading 1984.

[Schneiderman] Schneiderman, B., *Designing the User Interface*, Addison-Wesley, Reading 1987.

[Szafron] Szafron, D., J. Adria and B. Wilkerson, *GUIDE: An Environment for Software Design*, INFOR, January 1985, pp 31-52.

[Trevor] Trevor Hopkins and Bernard Horan, *Smalltalk: an introduction to application development using VisualWorks*, Prentice Hall 1995.