Chapter 5
Building User Interfaces in Squeak

1 Issues in Building a User Interface

Designing user interface software really has two pieces to it.

- How do you create user interface software that you can maintain well, is truly object-oriented, is easy to build, and is easy to change in pieces without impacting everything?
- How do you create user interfaces that people can actually use?

This chapter addresses only the first question. The second question is much more complicated and will only be touched upon in the next chapter. This chapter deals with the characteristics of user-interface software that is easy to build, object-oriented, and maintainable.

These are not always goals that fit together easily. It's possible to build interfaces quickly, but in throwing the pieces together, the programmer might not also create a clean structure that is easy to manipulate years later. In terms of creating object-oriented and maintainable software, there is a mechanism called *model-view-controller (MVC) paradigm* that has served well over the years—but it's not at all easy to understand and use. MVC basically describes a user interface in terms of a *model* of the real world, which is presented in a *view*, with user input handled by one or more *controllers*.

Much of the work going on in user interface software today emphasizes the ease of use but not the engineering issues. In a lot of new user interface toolkits or prototyping tools, you embed the code that defines the model or application inside the view. For example, you may put a script behind a button that does something to a data structure when the button is pressed. Now, where's the model? Scattered all over the various UI components. Where's the view? Well, it's there, but it's completely and inextricably linked to the model. This is hard to maintain. Squeak offers a new mechanism called *Morphic* that provides both ease of use and the possibility of good engineering. This chapter presents both MVC and Morphic.

But what makes Squeak particularly effective for exploring UI construction is that one can use *either* MVC *or* Morphic—*or* something else entirely! Squeak provides all of the primitives for creating whatever UI paradigm or toolkit you might wish to explore, with MVC and Morphic as two examples. This chapter begins by doing exactly that—building piece of a UI toolkit from scratch. The reason is not that you will often build a UI toolkit, but if you see the issues underlying a structure like MVC, it can help in understanding any individual toolkit.

2 Developing Model-View-Controller

The core idea of MVC is that we need to separate our *model* of the real world from the user interface *view* . The reality of any application is that both the model and view are going to change over time. The model is going to change (e.g., we may be using a **Clock**, but decide later that we want to use an **AlarmClock**.) Our *view* is going to change: Between an analog and digital clock, between one kind of knob and another kind of button. We don't want to be tweaking the model every time that we change the view, and we don't want to have to change the view when we swap out the model.

How close can we get to this goal? Can the model know *nothing* of the view? Can the view know *nothing* of the model? We probably can't get both, but an MVC structure can get us as far as we can get towards that goal. In this section, we build a user interface for our clock. We're going to do it not once, not twice, but three times. Each time, the interface we build will be essentially the same, but we will try to improve on the maintainability and object-oriented-ness of the system. Each time, the user interface will look essentially the same (Figure 1), but the underlying mechanism will change fairly dramatically.

- The text at the top is the running clock.
- The buttons on the bottom increment or decrement the hours, and increment or decrement the minutes. The idea is to use these to set the time.

Figure 1: A Clock User Interface

2.1 Round 1: No Separation At All

Let's make the first pass by simply tacking on a user-interface onto our existing **Clock** structure. The code for this version of the user interface is on the CD, and is available at http://guzdial.cc.gatech.edu/st/clock-ui1.cs.

We begin by considering how we need to revise our existing design. We won't do a CRC Card analysis because, strange as it seems, we've already *decided* to do this wrong. We won't add any new classes. Instead, we'll simply throw everything into the **Clock** class. But in so doing, we might get a clearer idea of what we really do need in our analysis.

We can identify several pieces that we'll need to add to Clock in order to get the interface in Figure 1.

- We need some way to define all the pieces of our interface: The text are for the clock, the buttons for changing the time, and the blank area that handles everything. This will include adding one new instance variable: The upper-left hand corner **position** of the clock, in order to place all other interface components against a set point.
- We will need to do something to make sure that the time updates every second.
- We will also need to deal with the user's actions, which are typically called interface *events*, such as clicking on a button. Somehow, we must catch the event and handle it appropriately (e.g., incrementing the minute if the user clicks on the *Minutes+* button.)

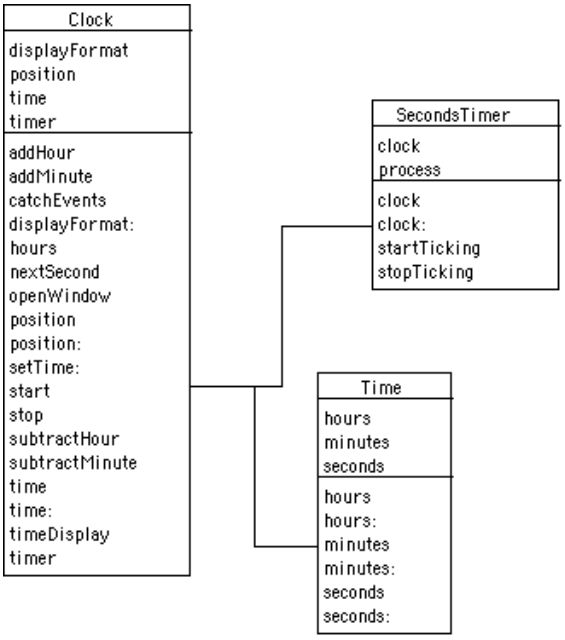

Figure 2: Class Diagram for Round One of Clock UI

All of this leads to the class diagram in Figure 2. (You *should* look at the long list on Clock and wonder “Do we really want all of that in one class? Does all of this belong here?”) We’ll add the position **instance** variable (and the accessor methods for it), and a method to **openWindow** that places all the interface components. We’ll add the methods **addHour**, **addMinute**, **subtractHour**, and **subtractMinute** to make it easier to build the buttons. We’ll have to modify **nextSecond** to send the message **timeDisplay** which will update the display. We’ll handle the user’s events in **catchEvents**.

We start implementing these changes by updating our definition of the **Clock** class.

```
Object subclass: #Clock
        instanceVariableNames: 'time timer displayFormat position '
        classVariableNames: ''
        poolDictionaries: ''
        category: 'ClockWorks'
```

The new position instance variable will track where the clock's window will go. We'll need accessor methods for the new variable.

```
position
        ^ position

position: aPoint
        position := aPoint.
```

Once the position is set, we can actually open a window on a clock.

```
openWindow
        | pen |
        "Open the blank frame"
        (Form extent: 200@200) fillWhite displayAt: position.

        "Draw the Buttons"
        pen := Pen new.
        pen up. pen goto: (position x) @ ((position y)+100). pen down.
        pen north. pen turn: 90.
        pen go: 200.
```

```
pen up. pen goto: (position x) @ ((position y)+150). pen down.
pen go: 200.
pen up. pen goto: ((position x)+100) @ ((position y)+100). pen down.
pen turn: 90.
pen go: 100.
'Hours +' displayAt: ((position x)+25) @ ((position y)+125).
'Hours -' displayAt: ((position x)+125) @ ((position y)+125).
'Minutes +' displayAt: ((position x)+25) @ ((position y)+175).
'Minutes -' displayAt: ((position x)+125) @ ((position y)+175).
```

This is a fairly complex piece of code, so let's walk it through a bit:

- The expression to create a blank (white) frame is one that we saw before, in Joe the Box.
- The **Pen** expressions are creating the boxes below the time display which suggest the "buttons".
- The final four lines create the labels on the buttons.
- Note that all the coordinates are offsets off of the position instance variable. The buttons start about 100 pixels below the upper left hand corner of the display.
- Also note that there is nothing in here that says anything about the text to appear in the upper part of this "window".

To make the text display work, we're going to need to modify **nextSecond** and create a method to actually paste the time display into our makeshift window.

nextSecond

```
time := time addTime: (Time fromSeconds: 1).
self timeDisplay.
```

timeDisplay

```
'          ' displayAt: position + (50@50).  "Erase whatever time was there before"
self display displayAt: position + (50 @ 50).
```

The code so far will make the window appear with the time display in the middle of it. As each second goes by, the time will update in the window. We have yet to deal with the buttons. Simply clicking inside our boxes won't do anything yet.

We need a piece of code called an *event loop*. An event loop waits for user interface events, figures out who needs what event, then passes the event to the appropriate interface component. An event loop is actually a

very important invention for user interfaces. Previous to having an event loop, interfaces would be written with the computer in charge, by dictating when input was to occur from what device (e.g, by providing a command line prompt to say “Okay, now *you* type something.”) An event loop changes everything: Now, the user is in control, and the computer waits for an event that it can handle, and then handles it when it arrives.

Here is an event loop for our first user interface system.

catchEvents

```
| hourPlus hourMinus minutePlus minuteMinus click |
"Define the regions where we care about mouse clicks"
hourPlus := (position x) @ ((position y)+100) extent: 100@50.
hourMinus := ((position x)+100) @ ((position y)+100) extent: 100@50.
minutePlus := (position x) @ ((position y)+150) extent: 100@50.
minuteMinus := ((position x)+100) @ ((position y)+150) extent: 100@50.

"Enter into an event loop"
[Sensor yellowButtonPressed] whileFalse: "Yellow button press ends the clock"
        ["Give other processes a chance, and give user a
        chance to pick the mouse button up."
        (Delay forMilliseconds: 500) wait.
        (Sensor redButtonPressed) ifTrue:
                "Red button press could go to a button"
                [click := Sensor mousePoint.
                (hourPlus containsPoint: click)
                        ifTrue: [self addHour].
                (hourMinus containsPoint: click)
                        ifTrue: [self subtractHour].
                (minutePlus containsPoint: click)
                        ifTrue: [self addMinute].
                (minuteMinus containsPoint: click)
                        ifTrue: [self subtractMinute].]].
```

Let's walk through this fairly lengthy code. The method starts out by defining four rectangles, one for each of our buttons. These are exactly the same regions that we defined when we drew the buttons. (You should be thinking, “Do I really have to do this twice?”) Then there’s a loop waiting for the yellow button to be pressed, which is the signal that we’ll decide indicates the end of the clock processing. Until the yellow mouse button is

pressed, we look for the red button to be pressed. If there is a red mouse button press, we get the mouse point position, and see if it's within one of our four regions. If the mouse press is in one of them, we execute the appropriate method for the clock to add or subtract time. The small **Delay** at the top of the loop is to prevent a single mouse click from being executed several times due to the loop being faster than the human.

> ***SideNote:*** This event loop triggers a button press upon clicking *down* with the mouse. Most interfaces actually trigger the event upon *releasing* the mouse button. This gives the user the opportunity to move the cursor after pressing down *before* triggering a button. The above event loop could be written to support mouse releases rather than mouse down events. The key is to track the state of the mouse so that the release can be detected.

We haven't actually created these four methods yet, but they are fairly straightforward.

addHour

```
time := time addTime: (Time fromSeconds: 60*60)
```

addMinute

```
time := time addTime: (Time fromSeconds: 60)
```

subtractHour

```
time := time subtractTime: (Time fromSeconds: 60*60)
```

subtractMinute

```
time := time subtractTime: (Time fromSeconds: 60)
```

To make our first user interface work, we execute code that looks like this:

```
c := Clock new.
c position: 100@10.
c setTime: (Time now printString).
c openWindow.
c start.
c catchEvents.
```

The clock will then start running. You can click on the buttons to change the displayed time. (Remember that you're modifying your **Clock** instance—you're not changing your system's time.) To stop the event loop, use your yellow button anywhere. The clock will still keep running. To stop the clock, use **c stop** in your workspace.

Exercises: Improving Round One

1. Change the event loop to process on mouse releases, not mouse downs.

2. Figure out why the clock is still running after the yellow button press, and change the code above (hint: in the event loop) so that the clock stops when the user interface stops.

2.2 Round 2: Separate Windows and Buttons

While this user interface has the advantage of working, it has a great many disadvantages. Here are a few.

- We've had to modify the **Clock** class a great deal. There's no separation between view and model here. To change the layout of the window, for example, we have to change the window drawing code as well as the event loop. To change the window from being a digital to an analog display, we might as well start from scratch.
- The clock suddenly has to know all kinds of things that it shouldn't care about: From the position of the window in space, to user interface events. Responsibilities are clearly wrong here.
- There is nothing reusable here. The next user interface is going to be just as hard as the first with this structure.

Let's start making it better by separating off the two most obvious components: The **ClockWindow** and the **ClockButtons**. It's pretty clear that we need these components. A CRC Card analysis would lead to determining these responsibilities.

- The **ClockWindow** should be responsible for assembling the user interface and displaying the time.
- The **ClockButtons** should handle displaying themselves and triggering the appropriate action in the clock.

We can move from these descriptions to a class diagram with more details of how to actually make this work (Figure 3). We can move the **position** instance variable and the **timeDisplay** method from **Clock** into **ClockWindow**. We're moving the window opening and event catching methods, too, but the names change. **ClockWindow** has an **openOn:** method that takes a parameter of a clock to use as its model. Instead of **catchEvents**, we'll have a **processEvents** method, which seems to be more accurate.

We need to add an instance variable to **Clock** that references the **ClockWindow**, **clock**. We need this because **ClockWindow** now knows how to do the **timeDisplay**, but we request the **timeDisplay** during **nextSecond**, which is in **Clock**. Thus, at least for this round, the **Clock** has to know the **ClockWindow**.

The **ClockWindow** knows its **position**, its **clock**, and its **buttons**. It needs to know about its buttons in order to check them and pass control

to them if they get a mouse click. The rest of **ClockWindow**'s methods are just accessors.

The **ClockButton** is another new class which handles drawing, checking for clicks, and executing some action for the on-screen button. We can talk about each responsibility separately.

- The **ClockButton** knows how to **draw** itself, and to do that, it knows its **frame** (the rectangle where it is displayed) and its **name** (to display its label).
- The **ClockButton** can respond whether it is **inControl:** of a given mouse click, by checking the position against its **frame**.
- If the **ClockButton** does have control, it's asked to **process**, which involves telling its **model** to execute a given **action**. We'll talk more about the **action** later, because it's quite important for having flexible MVC components.

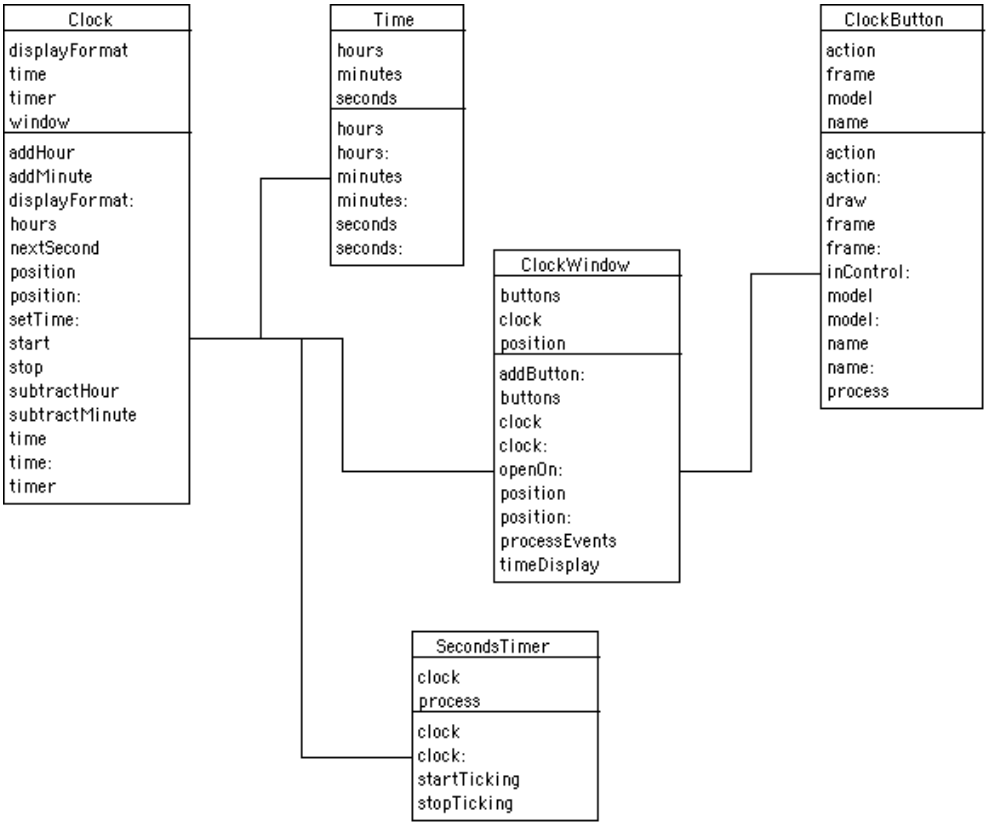

Figure 3: Class Diagram for Round Two User Interface

The code in this section is available on the CD and at http://guzdial.cc.gatech.edu/st/clock-ui2.cs. We'll start out by redefining the **Clock** class, and defining **ClockButton** and **ClockWindow**.

```
Object subclass: #Clock
```

```
        instanceVariableNames: 'time timer window displayFormat '
        classVariableNames: ''
        poolDictionaries: ''
        category: 'ClockWorks'
Object subclass: #ClockButton
        instanceVariableNames: 'model frame action name '
        classVariableNames: ''
        poolDictionaries: ''
        category: 'ClockWorks'
Object subclass: #ClockWindow
        instanceVariableNames: 'position clock buttons '
        classVariableNames: ''
        poolDictionaries: ''
        category: 'ClockWorks'
```

The **Clock** still knows about its **window**, which is unfortunate, but we'll clear this up later. At least, it no longer has to know about a **position**. The **Clock** is nearly back to its basic responsibilities, with the **ClockWindow** taking care of the user interface issues.

Let's start by looking at how we create windows in this version of the user interface code.

```
openOn: aModel
        | button |
        position isNil ifTrue: [self error: 'Must set position first.'].

        "Set this model as this window's clock"
        clock := aModel.

        "Open the blank frame"
        (Form extent: 200@200) fillWhite displayAt: position.

        "Draw the Buttons"
        button := ClockButton make: 'Hours +'
                at: ((position x) @ ((position y)+100) extent: 100@50)
                for: aModel
                triggering: #addHour.
```

```
        self addButton: button.
        button := ClockButton make: 'Hours -'
                at: (((position x)+100) @ ((position y)+100) extent: 100@50)
                for: aModel
                triggering: #subtractHour.
        self addButton: button.
        button := ClockButton make: 'Minutes +'
                at: ((position x) @ ((position y)+150) extent: 100@50)
                for: aModel
                triggering: #addMinute.
        self addButton: button.
        button := ClockButton make: 'Minutes -'
                at: (((position x)+100) @ ((position y)+150) extent: 100@50)
                for: aModel
                triggering: #subtractMinute.
        self addButton: button.
```

This is a significant piece of code, so let's walk through the pieces.

- The method starts out by checking to make sure that the **position** is set. If it's not, we will not be able to position everything else, so it's worth checking.
- Next, the argument **model** is set to be the window's **clock**.
- The method clears a frame, as before.
- Each button is created as an instance of **ClockButton**. We specify a name for each button, its frame (a rectangle is specified as an upper left corner and a distance to the lower right, the extent), and the model and action message that it sends. The pound sign is necessary for defining a *symbol*. **#addHour** is a `symbol`. A symbol is a kind of special **String** that the Smalltalk knows which can be the name of a method. Internally, there can only be one instance of each kind of symbol, so all references point to the same thing. That makes lookup especially fast.
- As the buttons are created, they are added into the button list.

While most of the accessors of **ClockWindow** are fairly straightforward, it's worth taking a peek at **addButton**:

```
addButton: aButton
        buttons isNil ifTrue: [buttons := OrderedCollection new].
        buttons add: aButton.
```

Notice the first line: We check if **buttons** is already set up as an **OrderedCollection**, and if not, we set it. This isn't the best way of initializing an instance variable. It's better to do it in an **initialize** method. This is called *lazy initialization*. There is an advantage to use this method if it's difficult to initialize an object or if the class variable is not used often and is huge. In general, though, it's not the cleanest way to initialize a variable.

Processing the event loop becomes a very different activity when the window and buttons are all separate:

```
processEvents
    "Enter into an event loop"
    | click |
    [Sensor yellowButtonPressed]
    whileFalse: "Yellow button press ends the clock"
        ["Give other processes a chance,
        and give user a chance to pick up."
        (Delay forMilliseconds: 500) wait.
        (Sensor redButtonPressed)
        ifTrue: "Red button press could go to a button"
            [click ← Sensor mousePoint.
            buttons do: [:b |
                (b inControl: click) ifTrue: [b process]].]].
```

The main loop here is the same, but the body of that loop is different. Now, we simply ask each button "Do you want control of this mouse click?" and if so we tell the button to **process**. It's a very simple structure which distributes responsibility from the window into the buttons.

Obviously, we'll have to handle that responsibility in the buttons. Let's begin going through the **ClockButton** to see how it's implemented. The basic creation method for buttons is a class method. This means that you create ClockButton instances with a specialized send to the class itself.

```
make: aName at: aRect for: aModel triggering: aMessage
    | newButton |
    newButton ← self new.
    newButton name: aName.
    newButton frame: aRect.
    newButton model: aModel.
```

```
newButton action: aMessage.
newButton draw.
^newButton.
```

Drawing a button is pretty straightforward: We simply use the **Pen** code we wrote earlier, but parameterize the positions differently.

draw

```
| pen |
pen := Pen new.
pen color: (Color black).
pen up. pen goto: (frame origin).
pen north. pen turn: 90. pen down.
pen goto: (frame topRight).
pen turn: 90. pen goto: (frame bottomRight).
pen turn: 90. pen goto: (frame bottomLeft).
pen turn: 90. pen goto: (frame origin).
name displayAt: (frame leftCenter) + (25@-10).
        "Offset in a bit, and up a bit for aesthetics"
```

This code is pretty self-explanatory because **Rectangles** know a lot of nice methods for getting their coordinates, such as **origin**, **topRight**, and **bottomLeft**. Basically, we just move the pen around the points of the frame, starting at the **origin** (top left). The little fudge factor in the positioning of the label, **name**, is just to make the label look a bit better. Try it with and without the fudge factor to see why it's there.

inControl: and **process** are both one-liners. **inControl:** is simply testing whether the click point is within the **frame** of the button. If it is, **process** tells the model to perform the given **action**.

inControl: aPoint

```
^frame containsPoint: aPoint
```

process

```
model perform: action
```

Before we explain how **process** *does* work, let's consider how it *might* work. Imagine that **process** does nothing at all, by default. Instead, you create a separate subclass for the **HoursPlusButton**, for the **HoursMinusButton**, and so on. In each subclass, you override the default **process** method in the superclass, and provide a **process** method which does the appropriate action for *Hours+*, *Hours-*, and so on. What would each of these subclass **process** methods look like? For

HoursPlusButton, it would just say **model addHour**. That's it, just two words.

The original user interface components for MVC in Smalltalk-80 *did* work like this—you subclassed components for each specific use. But since the subclasses were *so* similar, it became clear that it would be possible to parameterize the superclass so that each use would simply be an instance with different instance variables. We call these new kinds of user interface components *pluggable*, because they have plugs (parameters) for the various important parts of the user interface. Our **ClockButton** is pluggable. The **action** is a plug.

What makes pluggable work, in general, are blocks and the **perform**: message. **process** uses an interesting message, **perform**:. **perform**: takes a symbol as an argument, then sends the symbol as a message to the receiving object. In a sense, this is writing code on the fly. The action message could be anything, even input from the user translated into a symbol. (**String**s understand **asSymbol** to convert.) Asking an object to perform whatever message we want a powerful structure that dynamic languages like Smalltalk provide. This flexibility is important in order to create pluggable components.

We'll move the text display into the **ClockWindow**. The **ClockWindow** will clear the existing text, then ask the **clock** what the **display** time is, then display it.

```
timeDisplay
        '          ' displayAt: position + (50@50).  "Erase whatever time was there before"
        (clock display) displayAt: position + (50 @ 50).
```

And we'll change **Clock** to ask the **ClockWindow** to do the text display.

```
nextSecond
        time ← time addTime: (Time fromSeconds: 1).
        window timeDisplay.
```

We can create this clock with workspace code like the below.

```
c := Clock new.
w := ClockWindow new.
w position: 100@10.
c setTime: (Time now printString).
w openOn: c. c window: w.
c start.
```

w processEvents.

Notice that the user interface *looks* exactly the same, but we know that the underlying mechanism is now radically different. To stop this click, you click with the yellow button to stop the event loop. Then do **c stop** to stop the clock.

2.3 Round 3: Separating Even the Text

Round Two is clearly a much nicer user interface model, but it's still not as good as it could get. Let's consider its strengths and weaknesses.

- Clearly, we have a much nicer separation between user interface components (views) and the model. Except for handling the text display, the clock doesn't know anything about its window or the buttons that impact it. The window only needs to know about the clock with respect to getting the time to display.
- Though they're named **ClockWindow** and **ClockButton**, these are fairly generic pieces now. Those buttons could appear part of anything, and send any message to any object. The window isn't a window in the sense of being draggable nor integrated with other windows, but it is a frame that things can be placed in. Both are nice starts toward generic UI components.
- The text is a real problem. Not only does it force us to spread around more information than we might like (Why should the clock know its window? Why should the window know about anything other than its components? Why should it have to know about the clock?), but it also deeply constrains the UI structure. Imagine converting all of this into an analog clock, with a dial and two hands. The window and buttons would work almost as-is—*almost*. But the text display of the clock is hard-wired in.

Separating the text display of the time is going to be tricky. We have to have some way for the clock to tell its view (whether a textual display or an analog dial) when the time has changed, so that the view can update. But we don't really want the clock to know about its view, or even its *views*—plural. If we do this right, one could imagine having two (or more) different displays on the same clock. But in order to do this right, we certainly don't want the window to be hard-wired to display the clock. How then does the window find out when it has to update?

The original Smalltalk developers had this same problem. They wanted a mechanism for building user interfaces that was efficient, flexible, and yet maintained an object-oriented nature. What they came up with was the rest of the Model-View-Controller paradigm. We've already met two of the three pieces, and the third one isn't all that critical for what we're doing.

- The Model is the object from the problem domain. It's where the services and attributes core to the problem are stored.
- The Views are the objects that the user interacts with. Buttons, text areas, graphs, and other kinds of interaction objects are views.
- The Controller is essentially the event loop. It is the controller that collects user interface events (through the **Sensor**), then hands them off to the view (e.g., mouse clicks on a button) or to the model (e.g., keyboard input). The controller is probably the most complicated piece of the three, but fortunately, is one least often requiring changes. Typically, a mouse click is a mouse click, and only unless you want something to happen only upon a more unique combination, like control-a-shift-mouse click, do you care about modifying the controller.

By separating these three pieces, we can modify any one without modifying the others. The model can change, but the user interface remain the same. The interface can change without changing the underlying model. Changing the controller apart from the others is also useful—you can decide to trigger something upon mouse click or upon control key. The decision can be made later.

The MVC structure alone doesn't solve the text update problem. The text update problem is handled by the *dependency* mechanism that was built into Smalltalk to make this kind of update work within MVC. The dependency mechanism allows views to **update** upon **changes** in the model. The dependency mechanism can actually be used to handle any kind of dependency relationships—not just between views and models. It certainly works really well for this connection.

- Views make themselves *dependent* on a model object. Literally, the code is **model addDependent: view**. How the dependency is recorded is unimportant for the paradigm. Suffice to say that there is more than one way that it happens, depending on the superclass of the model, but in no case does the programmer of the model class ever have to maintain a record of its dependents. For the most part, the model can ignore that any views exist.
- Models simply announce when something has changed. Literally, the code is **self change: #something** where **#something** should inform the view of the *aspect* of the model that changed. (**#something** doesn't have to be a symbol — it could be a string or something else. Using a symbol is efficient and keeps the information passing small.) If a model (say, a **Passenger**) has many attributes, one view may only care about one aspect of the model (say, its destination), while another view may only care about another (say, its payment type). Announcing that something has changed is a pretty

lightweight operation. It's reasonable to sprinkle them liberally throughout the model wherever a view *might* care about a change.

- Behind the scenes, the change: method informs all the dependent views to **update:** with the aspect information as an argument. The author of a view needs to create an appropriate **update:** method for her view. The update methods can decide if they care about that aspect, and if so, they can ask their model for whatever information they need.

Let's use this structure to create our Round #3 user interface for the **Clock**. We need to introduce a **ClockText** class, that's clear. It must respond to **update:**. In order to **update:**, it needs to know the **position** it must draw to, the **model** that it has to get information from, and the **query** message that it must send for the information to display. The **Clock** no longer needs to know about the **ClockWindow**. And the **ClockWindow** doesn't need the **Clock** anymore—only the **ClockText** will be requesting information from the **Clock**. All of this leads to the class diagram in Figure 4.

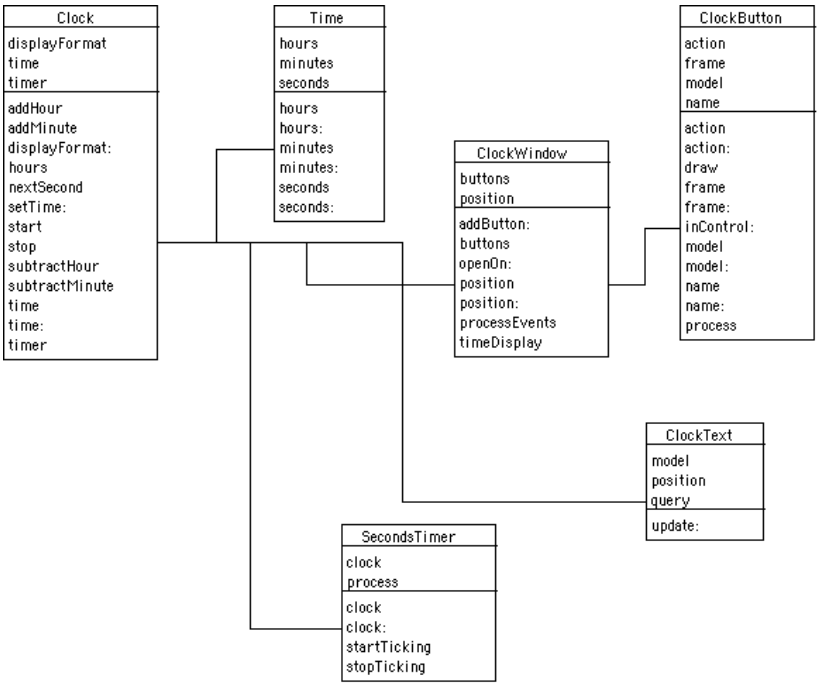

Figure 4: Class Diagram for Round Three of Clock UI

The code for Round #3 is available on the CD and at http://guzdial.cc.gatech.edu/st/clock-ui3.cs. We begin by redefining the classes **Clock** and **ClockWindow**, and adding our new **ClockText**.

```
Object subclass: #Clock
        instanceVariableNames: 'time timer displayFormat '
```

```
        classVariableNames: ''
        poolDictionaries: ''
        category: 'ClockWorks'
Object subclass: #ClockText
        instanceVariableNames: 'model position query '
        classVariableNames: ''
        poolDictionaries: ''
        category: 'ClockWorks'
Object subclass: #ClockWindow
        instanceVariableNames: 'position buttons '
        classVariableNames: ''
        poolDictionaries: ''
        category: 'ClockWorks'
```

We need to modify the **Clock** to announce a change in the time upon **nextSecond**. Unlike the previous versions of nextSecond, this change is quite simple, can be added or changed easily (e.g., involves no new instance variables), and doesn't really impact the design of our problem domain object.

```
nextSecond
        time ← time addTime: (Time fromSeconds: 1).
        self changed: #time.
```

The **ClockText** object maintains a **model** instance variable. When the model is set, we create the dependency between the text area and the clock.

```
model
        ^model


model: aModel
        model := aModel.
        aModel addDependent: self.
```

When **nextSecond** occurs, every view (dependent) on the clock is asked to update. Our text area only cares about the time changing, even if there were other aspects of the clock to care about. The text area remembers the **query** message to ask the model in order to get whatever it needs to display. We can ask the model to **perform:** the **query** message, and tell the result to display itself.

```
update: anEvent
```

```
anEvent = #time ifTrue: [
'          ' displayAt: position .  "Erase whatever time was there before"
(model perform: query) displayAt: position.]
```

To make it easier to set up a text area, we create a class method that creates a text area and sets it up appropriately.

at: aPosition on: aModel for: aQuery

```
| text |
text := self new.
text position: aPosition.
text model: aModel.
text query: aQuery.
^text
```

Finally, we can write a new opening method for our window that sets up the text area. The **openOn**: method is very similar to the one in Round Two, but we don't need to set the **ClockWindow**'s **clock** variable and we do have to set up the **ClockText**. Notice that the clock doesn't retain any connection to the text area. It doesn't need one—once the text area is set up, the window doesn't need to deal with it at all. (In a *real* window, the window would care about its subviews for things like updating upon moving or resizing, but it's not an issue with our pretend window.)

openOn: aModel

```
| button  |
position isNil ifTrue: [self error: 'Must set position first.'].

"Open the blank frame"
(Form extent: 200@200) fillWhite displayAt: position.

"Setup the textArea"
ClockText at: (position + (50@50)) on: aModel for: #display.

"Draw the Buttons"
button := ClockButton make: 'Hours +'
        at: ((position x) @ ((position y)+100) extent: 100@50)
        for: aModel
        triggering: #addHour.
self addButton: button.
button := ClockButton make: 'Hours -'
```

```
                at: (((position x)+100) @ ((position y)+100) extent: 100@50)
                for: aModel
                triggering: #subtractHour.
        self addButton: button.
        button := ClockButton make: 'Minutes +'
                at: ((position x) @ ((position y)+150) extent: 100@50)
                for: aModel
                triggering: #addMinute.
        self addButton: button.
        button := ClockButton make: 'Minutes -'
                at: (((position x)+100) @ ((position y)+150) extent: 100@50)
                for: aModel
                triggering: #subtractMinute.
        self addButton: button.
```

Running the Round Three version of the code is pretty similar to Round Two. Remember, still, to stop the clock with **c stop** after ending the event loop.

```
c := Clock new.
w := ClockWindow new.
w position: 100@10.
c setTime: (Time now printString).
w openOn: c.
c start.
w processEvents.
```

2.4 Strengths and Weaknesses of Model-View-Controller

MVC is the dominant metaphor for UI construction today. It's at the heart of how we think about user interface toolkits today. Even the latest user interface toolkits, like the Java Swing toolkit, are essentially MVC-based systems. Let’s consider the strengths and weaknesses of MVC.

Strengths

- Clean object-oriented structure that minimizes information sharing. The model knows essentially nothing of the views. The views don’t need to poll the model, and as we’ve seen, can be designed to be quite generic.

- Can support multiple views on the same model. An **update**: message goes to *all* dependents, even if they're in different windows. One could imagine having a single **Patient** class, for example, with separate views for doctors (who need to see test results and specify diagnoses and treatments), nurses (who need to see and implement treatments), and billing office (who doesn't need to see the diagnosis, but does need to know the costs of tests and treatments). The model doesn't have to be changed at all to support any number of views.

Weaknesses

- Inefficient. The view gets told to update, and then it has to ask the model for the value, and then it updates. Having the model *tell* the view what to change would be more efficient, but less object-oriented. If the model knew what the view wanted, then there would be an information dependency between them where changing one might require changes in the other.
- Especially inefficient for multiple views. Let's say a doctor changes something on a patient's record (adds a test, for example). That view changes the model, which then triggers an update, and all views (say, the laboratory's for what tests to run, the nurse's for what tests to check on, and the billing office for what to charge) now get the update, and request what is, probably, the same piece of information. Why couldn't the doctor's view tell all the others' views directly? To do that would require views to know something about each other, which is a less clean structure, but more efficient.
- Gets very complicated if you want to have a view dependent upon multiple models. Imagine that you have a nurse's view that wants to show all of the status information for the patients in a three-person room. When the view gets informed that the patient's prescription has changed, *which* model does the view request information from? You can handle this by creating *application models*—models that are dependent upon a set of problem domain models. The views get built on the application models. In the example, you'd create a **Room** object, that is itself dependent on the three patients, and whose responsibility it is to figure out which patient announced the update. The nurse's view is dependent on the **Room**.

The problem is that it's hard to deal with MVC's weaknesses without destroying its object-oriented clarity. That's an open research problem today. Some UI researchers are exploring ways to allow the programmer to work in pure MVC, while improving the efficiency at compile time. Other researchers are looking for alternatives to MVC, such as constraint-based systems.

Exercises: Rebuilding the Clock Interface

3. Now, try writing a graphic text area that displays an analog clock, then update the ClockWindow to use that.

4. Find a user interface toolkit and figure out how it handles model-view communication. Odds are good that it is using a change-update mechanism.

3 Building Pluggable User Interfaces in Squeak

Since the previous sections shows that it is clearly possible to build generic user interface components, it should come as no surprise that there are pluggable user interface components built into Squeak. The first versions of Squeak did not—they were based on Smalltalk-80 *before* pluggable interface components were created. Pluggable UI components were introduced soon after the Morphic user interface was added to Squeak. (More on Morphic later in this section.)

Building your interface with pluggable components has its strengths and weaknesses. An important strength is that, for many common interfaces, using pluggable components means that creating a user interface is *much* simpler. You may need *no* additional classes besides your model classes.

A weakness of pluggable components is that *only* standard interfaces can be created with them. If you wanted to have something animate inside a pluggable interface, or have scrollbars change their shape or color dependent on their content, you can't do it with pluggable interfaces. To do more unusual things, you'll need to either build your own pluggable interfaces—or do it in Morphic.

There are basically three kinds of pluggable components in Squeak. Each component comes in both a View (to be used in MVC) and a Morph (to be used in Morphic) variations. All three require a model to be specified to use them. Any selector can be **nil** (unspecified) which means that that specific functionality is not enabled.

- **PluggableButtonView** (**PluggableButtonMorph**):
  PluggableButtons normally keeps track of a state selector and an action selector. The action selector is the message to send the model when the button is selected. The state selector is used to ask if model if the button should currently be on or off. There are also selectors for asking the model for a yellow-button menu to display and for asking the model for a label (if the button's label needs to dynamically update). There are options to make the button work on mouse down rather than the standard mouse up, to ask the user if they're sure, and to use a shortcut character for triggering the button. PluggableButtons are often used with instances of **Switch** which can handle tracking

boolean state. An example of use is the code in the **Browser** for creating the class switch button:

```
aSwitchView ← PluggableButtonView
        on: self  “The browser is the model”
        “It’s ‘on’ if the class messages are being shown”
        getState: #classMessagesIndicated
        “When triggered, class messages should be shown”
        action: #indicateClassMessages.
aSwitchView
        label: 'class';      “Label”
        window: (0@0 extent: 15@8); “Size of view”
        “Make sure that no text gets whumped”
        askBeforeChanging: true.
```

- **PluggableTextView (PluggableTextMorph):** PluggableText areas can keep track of up to four selectors. One selector retrieves the text to be displayed from the model. Another submits new text to the model when the user accepts the text. (Setting this selector to **nil** makes the text essentially read-only.) There are also selectors for getting the current text selection and for a yellow-button menu. PluggableText areas are often used with instances of **StringHolder** which can handle model-like access to a string. An example of use can be found in Celeste, the email reader in Squeak.

```
        “Set up a StringHolder as a model”
        textHolder ← StringHolder new .
        textHolder contents: initialText.  “Set the initial value”

        textView ←PluggableTextView
        on: textHolder  “The textHolder is the model”
        text: #contents “Ask for #contents when need the text”
        “Send #acceptContents: with the text as an argument to save”
        accept: #acceptContents:.
```

- **PluggableListView (PluggableListMorph):** PluggableLists can keep track of up to five selectors. The main three get the contents of the list, get the currently selected item, and set the currently selected item. There is also a selector for a yellow-button menu. The fifth selector processes a keystroke typed in the list, and the selector must take an argument of the keystroke. There is also an option to **autoDeselect** which allows you to turn off selection by clicking on

an item twice. The code that creates the message category list in the **Browser** looks like this:

```
                                        "Browser is the model"
    messageCategoryListView ← PluggableListView on: self
    "messageCategoryList returns the categories in an array"
            list: #messageCategoryList
    "messageCategoryListIndex returns an Integer of the current sel"
            selected: #messageCategoryListIndex
    "when the user changes the selection, messageCategoryListIndex is
sent"
            changeSelected: #messageCategoryListIndex:
    "MessageCategory has its own menu"
            menu: #messageCategoryMenu:.
```

3.1 Creating Pluggable Interfaces in MVC

Let's use the pluggable user interface components in Squeak to create an MVC-based interface for the **Clock**. We'll put the method in **ClockWindow**, though it really could go into **Clock**—nothing of the **ClockWindow** will be used anymore. The code for creating the window will look like this:

```
w := ClockWindow new.
w openInMVC.
```

CautionaryNote: Be sure that you are in an MVC project, or at the top-level (i.e., not in any project), when running this example. You won't see the MVC window if you do this from Morphic.

This is a very long method because of all the pieces that need to be created. In order to describe it, text will appear in the middle of the method, in this normal text font.

```
openInMVC
        | win component clock textArea |


        "Create the clock"
        clock := Clock new.
        clock setTime: (Time now printString).
        clock start.
```

Because the window is being opened separately from the **Clock**, it will be the window opening method's responsibility to create the clock, set the time, and get it started.

```
"Create a window for it"
win := (StandardSystemView new) model: self.
win borderWidth: 1.
```

StandardSystemView is the main window class in MVC. A **StandardSystemView** takes care of things like displaying a title bar and close box—and handling them appropriately. It also manages interactions with the main window scheduler that creates the illusion of overlapping windows. The above is necessary to create the window itself.

```
"Set up the text view and the various pieces"
textArea := PluggableTextView on: clock text: #display accept: nil.
textArea window: (0@0 extent: 100@100).
win addSubView: textArea.
```

Here's the PluggableText area code. The model will be the **Clock** instance that was created earlier. The text to display will be whatever the clock responds from the message **display**. We do not want the user to be able to edit the text, so we set the accept selector to **nil**. We specify the size of the text area to be 100 pixels horizontal by 100 pixels vertical. Finally, the window adds the new text area into it as a sub-view.

```
component := PluggableButtonView new
        model: clock;
        action: #addHour;
        label: 'Hours +';
        borderWidth: 1.
component window: (0@100 extent: 100@50).
win addSubView: component.
```

The button for incrementing the hours is created here. Its model is the **Clock** instance, with the action method to add an hour. It has a label, and a border will be displayed one pixel wide. The position of the button is

specified to be essentially the same as what it was in our previous user interface. The window is told to add the button.

We create the other three buttons similarly.

```
component := PluggableButtonView new
        model: clock;
        action: #subtractHour;
        label: 'Hours -';
        borderWidth: 1.
component window: (100@100 extent: 100@50).
win addSubView: component.
component := PluggableButtonView new
        model: clock;
        action: #addMinute;
        label: 'Minutes +';
        borderWidth: 1.
component window: (0@150 extent: 100@50).
win addSubView: component.
component := PluggableButtonView new
        model: clock;
        action: #subtractMinute;
        label: 'Minutes -';
        borderWidth: 1.
component window: (100@150 extent: 100@50).
win addSubView: component.
```

We need an additional button in this interface that wasn't in the previous ones. Since the clock is no longer accessible once the window is created, we need some way to stop it. We'll build a stop button for stopping the clock.

A better way to do this is to stop the clock upon closing the window. The model of the **StandardSystemView** is sent the message **windowIsClosing** when it is to be closed. The message breaks MVC in some ways: What if there are multiple views open on the same model? Which one is closing? For now we'll just create a stop button.

```
component := PluggableButtonView new
        model: clock;
```

```
            action: #stop;
            label: 'STOP';
            borderWidth: 1.
        component window: (0@200 extent: 200@100).
        win addSubView: component.
```

The below code sets the label for the window, and defines a minimum size. The window is opened by asking its controller to open.

```
    win label: 'Clock'.
    win minimumSize: 300 @ 300.
    win controller open
```

While this is a pretty long method, it is a *single* method. No new classes are needed to implement this user interface. It also has a good bit of flexibility built into it. The window can be dragged around, and even resized, and it will work just fine.

We do have to make one change to **Clock**. Pluggable components don't allow us to use any changed aspect. We have to do something that they expect. For PluggableText areas, the text knows to care about the **update:** message if the aspect is the same as the get-text selector. This means that **nextSecond** has to announce a change to **display** in order to get the text to update appropriately.

```
nextSecond
    time ← time addTime: (Time fromSeconds: 1).
    self changed: #display.
```

The completed window looks like Figure 5. Go ahead and try it from the code. Be sure to hit the Stop button before closing the window.

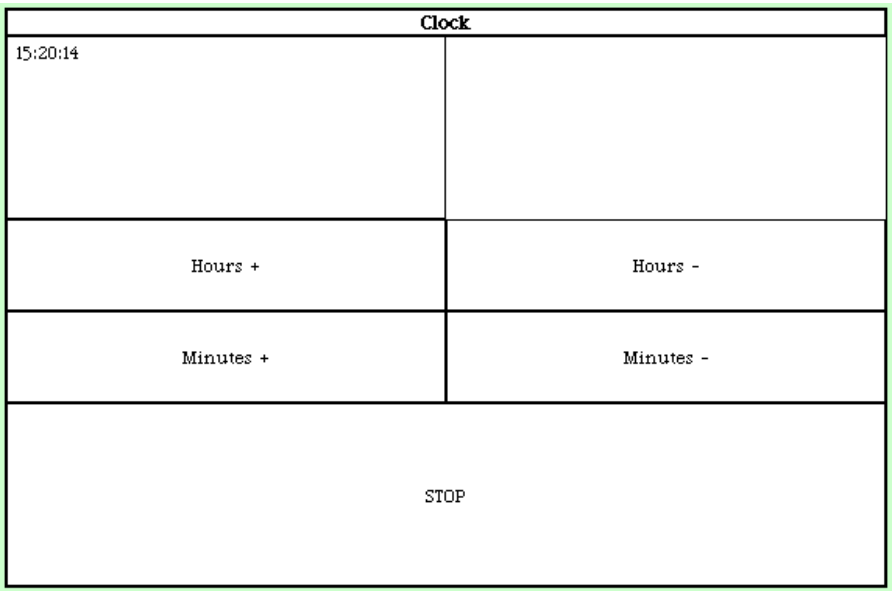

Figure 5: Clock UI from Pluggable Components in MVC

3.2 Creating Pluggable Interfaces in Morphic

Morphic is a very different model for doing user interfaces. In this section, we'll do the same interface, using the same pluggable components with an MVC architecture, but in Morphic. But first, we'll try out and introduce Morphic.

> ***CautionaryNote:*** There is a mixed use of terms in Squeak that may be confusing. The original window model (the structure by which all interfaces were built) in Smalltalk was called MVC (for Model-View-Controller). MVC is also a *paradigm*, a way of thinking, about building user interfaces. It is possible to use the MVC *paradigm* in a Morphic project, but objects that rely on the MVC *window model* must be run at the top-level or in an MVC project. We'll try to make it clear as we go along.

3.2.1 Introducing Morphic

At this point, even if you have a slow computer, try out Morphic. From the Desktop Menu, choose *Open* and *New Project (Morphic)*. Click in the new Project Window. Use the red button to open a World menu, where you can access tools via the *Open* window (just as in the Desktop menu).

You might also choose *Authoring* (which wasn't in the Desktop Menu) and open the Standard Parts Bin. You'll see a window like in Figure 6. Click on any of the components of this window, drag it out, and drop it on your desktop.

You start to see immediately why Morphic is different than the MVC window model. *Any* object can be a "window" in Morphic—even stars

and ellipses. Anything can lay on the desktop, can be resized, can be dragged around, can be laid on top of another window or Morphic object.

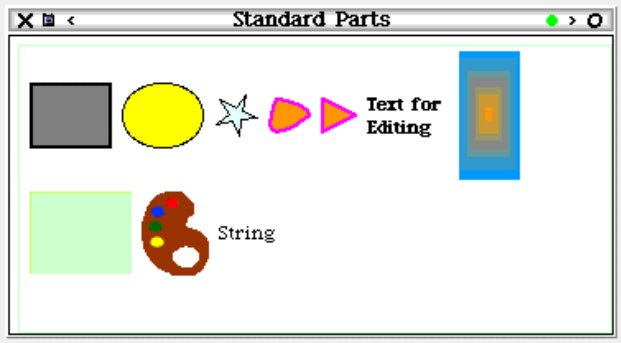

Figure 6: Standard Parts Bin in Morphic

Morphic was invented for the Self programming language. Self is, in several senses, a successor language to Smalltalk. Self explored just how efficient an object-oriented programming language could be without sacrificing pure object-oriented semantics, that is, *everything is an object*. Self was a project at Sun Microsystems Laboratories by David Ungar and Randall Smith. The Morphic user interface was developed by Randall Smith and John Maloney in Self, but John left that project to join Apple as Squeak was just getting started, and he brought Morphic with him.

Every display object in Morphic is a Morph. That's a key design element in Morphic. Every object in Morphic—every window, menu, and graphical widget—is a subclass of the class **Morph**, and inherits a set of standard behaviors and interfaces.

Morphic objects are *concrete*, *uniform*, and *flexible*, as the original design goals for Morphic stated. Morphic objects are concrete in that they can be moved around and manipulated in ways similar to the real world. Click down on a Morphic object and drag it. Notice the drop shadow behind the object (Figure 7). That's the kind of concrete realism that the Morphic designers wanted.

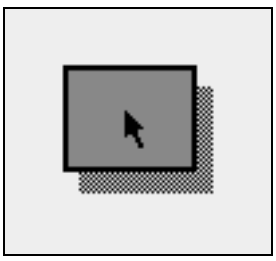

Figure 7: Moving with Drop Shadow

Morphic objects are uniform in that they all have a basic structure and can be manipulated in the same basic ways. Each Morphic object can be

selected (Table 1) to bring up a set of *halos* (colored dots) that allow the selected Morphic object to be manipulated in a standard set of ways, including resizing, rotations, and dragging. A standard set of halos is shown in Figure 8. You can get help on the meaning of any halo by simply resting your cursor above the colored dot (don't click down) and waiting a moment. A pop-up balloon explains the halo (as is seen for the red Menu halo in Figure 8.) Not all of the halos may make sense right now (like making a tile or opening a viewer), but those will be explained later.

System	Morphic Selection
Macintosh	Command-Click
Windows	Control-Alt-Click
UNIX	???-Click

Table 1: Morphic Selection in Various Systems

Go ahead and try resizing or rotating some morphs. You may be surprised to find that everything responds to those halos. Try keeping a menu up (by clicking on *keep this menu up* on any menu), selecting it, then rotating it.

Note that sometimes when you select (depending on where you click) you may click on a sub-element of the object, like a menu item inside of the menu. *Morphs can be composed to create new morphs*. That's another key design feature of Morphic. When you Morphic-select on a morph, you get the bottommost morph first. If you keep repeating the selection, though, you select the parent morph, and its parent, and then cycle around back to the bottommost morph.

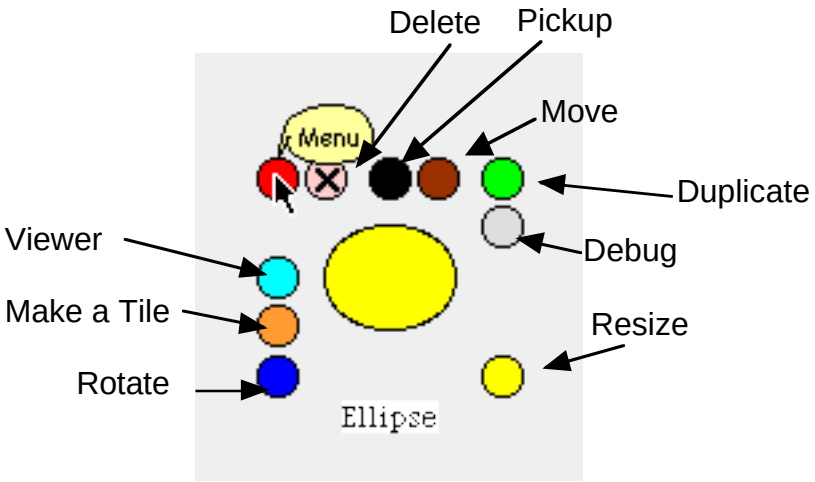

Figure 8: Standard Morphic Halos

Not all objects have the same halos. As is seen in Figure 9, editable strings have some extra halos that do morph-specific things. But the main

halos are uniform, and manipulation with halos is a constant across Morphic.

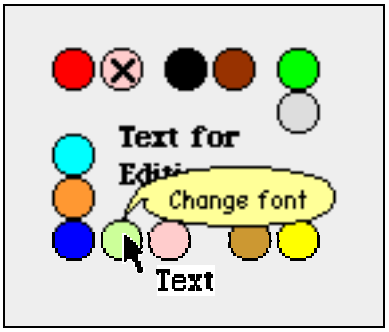

Figure 9: Halos on an Editable String Morph

There are two morph-specific menus associated with any morph. One is accessed from the red (upper left) halo. The second is accessed by using control-click (the same for all platforms) on the morph itself. In general, the red halo menu tends to have more programmer-specific commands (e.g., a debug menu, items to control the **Pen** that draws the morph), while the control-click menu tends to have more end-user facilities (e.g., features to name the object, save the object).

We have already seen some of the flexibility of Morphic. Every object can be resized and rotated—and most morphs still work in a rotated form! As we'll see in Section 4, the flexibility (and uniform structure) of Morphic extends into every morph.

There are many other morphs built into Squeak than just those few in the Standard Parts Bin. The way to get to all of them is via the *New Morph* menu, available from the World Menu (Figure 10). Basically all morphs are available through this window. The sub menus in the New Morph menu are the names of the class categories for the morph's classes, e.g., the class category *Morphic-Books* becomes the *Books* sub-menu on the New Morph menu, and the classes defined in that category become the morphs available in the sub-menu.

Figure 10: New Morph Menu

Another way to access morphs is through *flaps*. There is a Preference available (from the Help menu) to enable global flaps, *useGlobalFlaps* (Figure 11). (Help is available from pop-up balloons here, as for halos.) When the flaps are enabled, they appear along the edges of the screen in a Morphic project. The bottom flap is called *Supplies* and it contains standard morphs, like those in the Standard Parts Bin. By default, the flap will open when the cursor passes over the flap tab (Figure 12). Morphs can then be dragged and dropped onto the desktop.

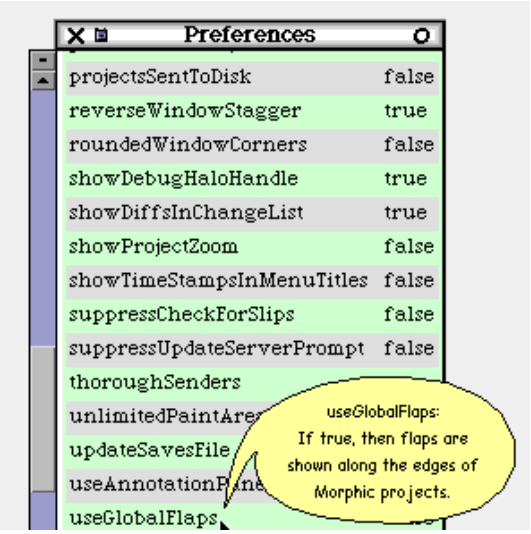

Figure 11: Preferences window, with useGlobalFlaps help available

SideNote: The other flaps contain tools, menus, and some useful buttons and menus in the *Squeak* flap. As everything else in Squeak, everything about flaps is completely malleable. Try Morphic-selecting a flap tab, then

choosing the red halo menu. You'll find that flaps tabs are positionable, can be triggered on something other than mouse-over, and can change their names or colors. The *About Flaps* button in the *Squeak* flap gives more information on using and creating flaps.

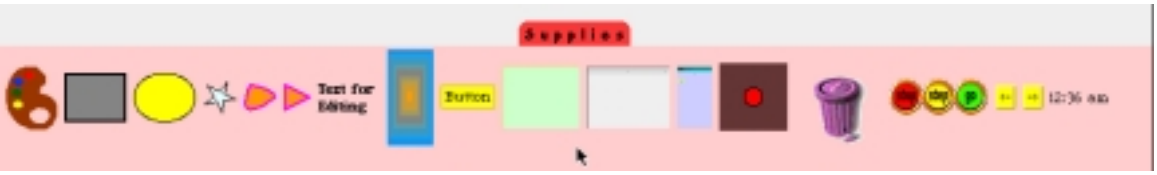

Figure 12: Supplies Flap in Morphic

3.2.2 Doing Pluggable Components in Morphic

An important aspect of Morphic is that there is an easy learning curve from the MVC window model, where the same kinds of pluggable components programming that we were doing in the older UI structure is still available in Morphic. You can bring up the Morphic version of our ClockWindow right now with these workspace expressions.

```
w := ClockWindow new.
w openAsMorph.
```

Notice that this will work in an MVC project, as well as in Morphic! When you open a morph from MVC, a miniature Morphic world (a Morphic window) is opened, with the morph inside it. This provides for a lot of flexibility in moving into Morphic.

Let's walk through the **openAsMorph** method for **ClockWindow**.

```
openAsMorph
    | win component clock |

    "Create the clock"
    clock := Clock new.
    clock setTime: (Time now printString).
    clock start.
```

We start out the same way: Creating the clock, setting its time, and starting the clock.

```
    "Create a window for it"
    win := SystemWindow labelled: 'Clock'.
    win model: self.
```

Instead of creating a **SystemView** as we did in the MVC model, we create a **SystemWindow**. A **SystemWindow** is a morph that provides all the standard window functionality: A close box, a collapse box, and a title bar. Note that you do not *have* to use a **SystemWindow** in Morphic—anything can be a window. But if you like the basic window structure, **SystemWindow** is a good starting place.

```
"Set up the text view and the various pieces"
component := PluggableTextMorph on: clock text: #display accept: nil.
win addMorph: component frame: (0.3@0.3 extent: 0.3@0.3).
```

Creating the **PluggableTextMorph** is obviously *very* similar to the **PluggableTextView** that we saw earlier. The same **on:text:accept:** message is used to create the instance. There are three significant differences:

- Notice that we don't define a **window:** for the view (that is, the frame where the component will be displayed). Instead, we specify the frame when we add the morph into the **SystemWindow**.
- Instead of **addView**:, we use **addMorph:frame:** to add the morph into a specific place in the window. Note that we don't *have* to specify a frame. You can just use **addMorph:** to add the morph in. You use tools such as **AlignmentMorph** in order to get the structure that you want if you just toss the morph in without specifying a frame. (We'll talk more about **AlignmentMorph** later in this section.)
- Notice that the frame is not specified in terms of a rectangle made up of points on the window. Instead, the frame is defined in terms of *relative* positions, where **0@0** is the upper-left hand corner and **1.0@1.0** is the lower-right hand corner. The rectangle defined for the PluggableTextMorph starts 1/3 of the window's horizontal and vertical size, and extends for another 1/3 (to **2/3@2/3**). The relative size will be respected through all resizing

Creating the buttons is very much the same in the Morphic version of the **Clock** user interface, modulo those same three changes as described above. In each case, we define a morph, we *don't* define a view-window, but we *do* define a frame when we add the morph to the whole window.

```
component := PluggableButtonMorph new
        model: clock;
        action: #addHour;
        label: 'Hours +';
```

```
        borderWidth: 1.
win addMorph: component frame: (0@0.6 extent: 0.5@0.2).
component := PluggableButtonMorph new
        model: clock;
        action: #subtractHour;
        label: 'Hours -';
        borderWidth: 1.
win addMorph: component frame: (0.5@0.6 extent: 0.5@0.2).
component := PluggableButtonMorph new
        model: clock;
        action: #addMinute;
        label: 'Minutes +';
        borderWidth: 1.
win addMorph: component frame: (0@0.8 extent: 0.5@0.1).
component := PluggableButtonMorph new
        model: clock;
        action: #subtractMinute;
        label: 'Minutes -';
        borderWidth: 1.
win addMorph: component frame: (0.5@0.8 extent: 0.5@0.1).

component := PluggableButtonMorph new
        model: clock;
        action: #stop;
        label: 'STOP';
        borderWidth: 1.
win addMorph: component frame: (0@0.9 extent: 1@0.1).
```

Opening the Clock window is even easier in Morphic than it is in the MVC window model. All morphs understand how to **openInWorld**. We don't have to mess with controllers. We simply tell the window to open.

```
win openInWorld.
^win
```

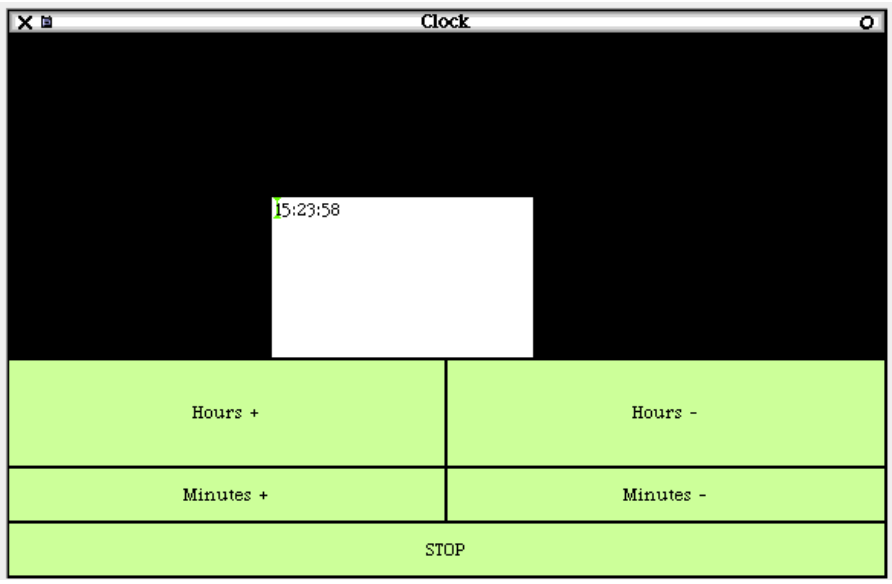

Figure 13: First Version of Pluggable Clock UI in Morphic

The resulting window appears as in Figure 13. Notice that the basic structure is exactly the same as in Figure 5, the MVC window model version. It's exactly the same *except* for the big black stuff around the text area. That is actually *nothing*. There is no morph there, and that's what the SystemWindow shows when there is no morph to display.

We'd like a better looking window than that. What we need is some kind of filler, which is where **AlignmentMorph** comes in. An **AlignmentMorph** is especially designed to fill in spaces and to align things nicely within that space.

The method below differs from **openAsMorph** only in that it fills *all* the top 2/3 of the **ClockWindow** with an **AlignmentMorph**. The **PluggableTextMorph** is then added to the **AlignmentMorph**. We tell the fill to center the morphs placed into it from the #**bottomRight**. The result is in Figure 14.

openAsMorph2

```
| win component filler clock |


"Create the clock"
clock := Clock new.
clock setTime: (Time now printString).
clock start.


"Create a window for it"
win := SystemWindow labelled: 'Clock'.
```

```
win model: self.

"Set up the text view and the various pieces"
filler := AlignmentMorph newRow.
filler centering: #bottomRight.
win addMorph: filler frame: (0@0 extent: 1.0@0.6).
component := PluggableTextMorph on: clock text: #display accept: nil.
filler addMorph: component.
```

"ALL OF THE REST IS JUST LIKE **openAsMorph**"

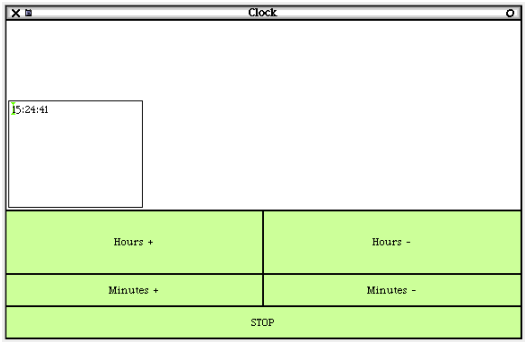

Figure 14: Cleaning up the Pluggable Morphic UI

We can get a different position for the text area by telling it to use center as #**center**. The result is in Figure 15. We can also get a different look by changing the *orientation* of the **AlignmentMorph**. Try **filler orientation: #horizontal** (or change it dynamically by Morphic-selecting the **AlignmentMorph** in the **ClockWindow**, then using the red-halo-menu to change the orientation.)

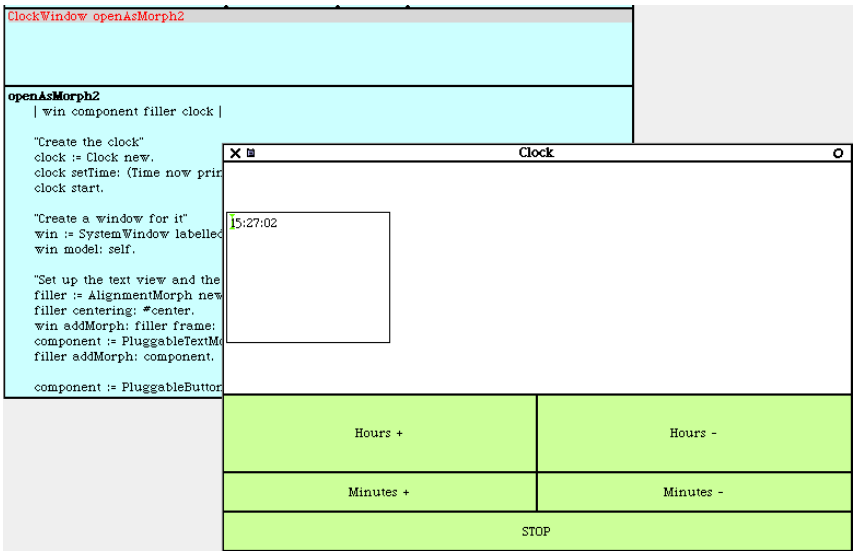

Figure 15: Exploring Variations on the AlignmentMorph

Basically, an AlignmentMorph has two roles:

- To lay out the component morphs (submorphs) in a row or column, possible resizing the submorphs as necessary.
- To possible resize itself based both on the sizes of the submorphs and whether or not it's contained in *another* **AlignmentMorph**. You can tell an **AlignmentMorph** to be **rigid** (never resize), **spaceFilling** (make yourself as big your enclosing **AlignmentMorph** will allow) and **shrinkWrap** (make yourself as big as you can be), and each of these can apply to the horizontal or vertical dimensions.

The options are laid out in the initialize method of **AlignmentMorph**, where the below is a quote from that:

```
orientation ← #horizontal.          "#horizontal or #vertical  or #free"
centering ← #topLeft.               "#topLeft, #center, or #bottomRight"
hResizing ← #spaceFill.             "#spaceFill, #shrinkWrap, or #rigid"
vResizing ← #spaceFill.             "#spaceFill, #shrinkWrap, or #rigid"
```

You can use multiple **AlignmentMorphs** to get the effect that you want. To center something in the middle, simply put **AlignmentMorphs** to either size and let them be space-filling. To force something to the right, put a space-filling **AlignmentMorph** on the left.

3.3 Menus and Dialogs in a Pluggable World

Menus can actually be handled exactly the same way in MVC and Morphic models, and they're very easy. While there are a wide variety of Menu classes, there are a couple of classes that serve as programmer's tools. You can set them up (even on-the-fly upon a button press), and then

open them up. They will return a value when selected. For example, the menu that pops up over the messages pane in Celeste (the email reader in Squeak) is created like this:

```
CustomMenu
        labels: 'again\undo\copy\cut\paste\format\accept\cancel
compose\reply\forward' withCRs
        lines: #(2 5 6 8)
        selections: #(again undo copySelection cut paste format accept cancel
compose reply forward)
```

The labels are the words in the menu. Labels can be specified as an array of strings, or a single string with carriage return (CR, ASCII 13) characters separating the items. **withCRs** translates back slashes into CRs for you. The lines are where lines should go in the menu, e.g., after items 2, 5, 6, and 8. The selection symbols match up with the labels and define the symbol to be returned when selected.

When this menu is sent the message **startUp**, the menu is opened up, and the user makes a selection. The selection symbol is then returned for later processing. One can also **startUp: initialSelection** so that a given item starts out being selected. If no item is selected, the menu returns **nil**.

You don't have to create all of the labels, lines, and selection symbols in one fell swoop. There is also an **add:action:** method for adding a word and a corresponding symbol to a menu, and an **addLine** method for inserting lines.

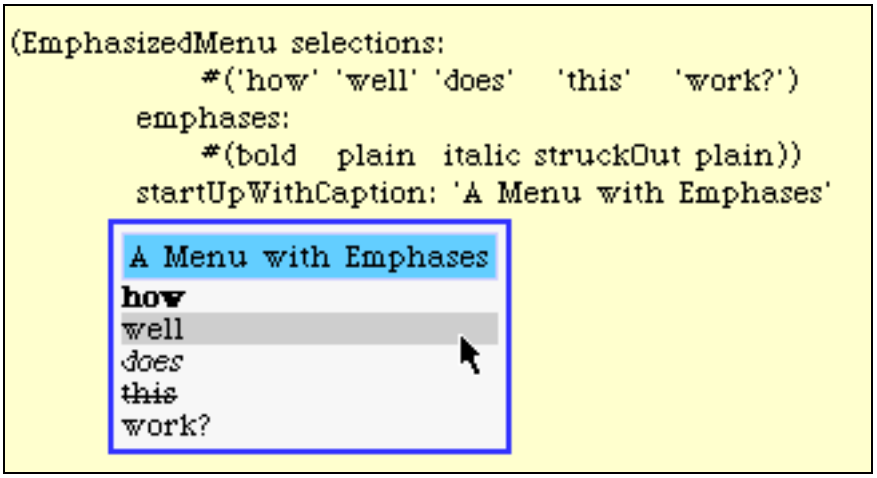

Figure 16: A Menu with Emphases

There are several useful menu classes that provide particular kinds of menus. An **EmphasizedMenu**, for example, allows you to add emphases to your menus, such as bold and italics, like in Figure 16. **PopUpMenu** provides some of the default dialogs that you might expect

to be able to inform the user of important events (Figure 17). **FillInTheBlank** is classified as a menu, but it's really the provider of various query-the-user dialogs, such as **request:** (Figure 18).

If you know that you will only be using your menu in Morphic, you can use the class **MenuMorph** and **GraphicalDictionaryMenu**. **MenuMorph**s understand some Morphic-specific features, like **addStayUpItem** (which allows a menu to stay available for later mouse clicks). When a **MenuMorph** is being constructed, it is also possible to specify **balloonTextForLastItem**: to set up help for users. **GraphicalDictionaryMenu** knows how to display forms for items, which can be a useful alternative in many situations.

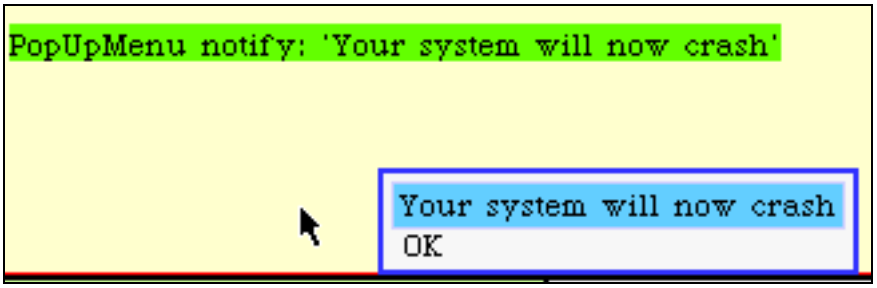

Figure 17: Using a PopUpMenu to Inform the User

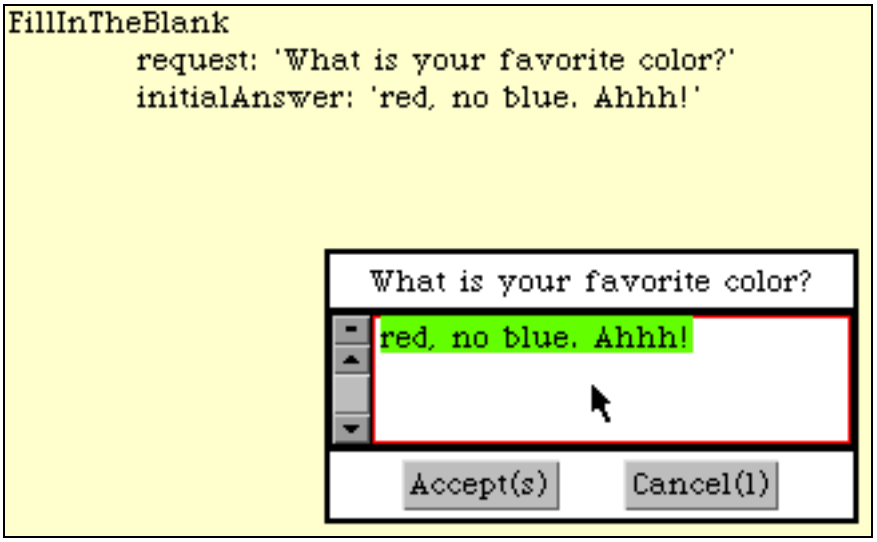

Figure 18: Querying the User with FillInTheBlank

Exercises: Working with Pluggable Interfaces

5. Redesign the **ClockWindow** so that there is no Stop button, and the **Clock** is stopped as soon as the **ClockWindow** is closed.

6. Get rid of the **ClockWindow** class and make the user interface work from **Clock**.

7. Use pluggable components to make a simple Rolodex. Have Rolodex cards containing name, address, and phone number information. Provide a scrolling list of names, and when one is selected, display the information in a text area.

8. Use pluggable components to make a simple calendar system. Provide a multi-pane list browser for picking a year (within, say, a ten year range), a month, and a date. (Be sure to fill in the date pane only when the year and month are selected!) Allow the user to fill in text pane notes for the given date. Use a Dictionary to store the text information, with the dates as the indices.

4 Building Morphic User Interfaces

The real strength of Morphic lies in creating Morphic interfaces within Morphic. Morphic interfaces don't necessarily have to follow the MVC paradigm, but they can. Morphic interfaces can also be assembled rapidly by simply dragging and dropping them. We have already seen that one morph can be *added* to another. From within Morphic, we say that one morph can be *embedded* within another.

In this section, we'll explore how to work with morphs from the user interface perspective, and then from the programmer's perspective. We'll use the same example, a simple simulation of an object falling, to explore both sides. Along the way, we'll describe the workings of Morphic.

4.1 Programming Morphs from the Viewer Framework

The Viewer framework (sometimes called *etoys system*) has been developed by Scott Wallace of the Disney Imagineering Squeak team as an easy-to-use programming environment for end users. It's not a finished item, and it may change dramatically in future versions of Squeak. But as-is, it provides us a way of exploring Morphic before we dig into code.

We're going to create a simulation of an object falling. Our falling object will be a simple EllipseMorph. Our falling object will have a velocity (initially zero) and a constant rate of acceleration due to gravity. We'll just use pixels on the screen as our distance units.

If you recall your physics, the velocity increases at the rate of the acceleration constant. For our simulation, we'll only compute velocity and position *discretely* (i.e., at fixed intervals, rather than all the time the way that the real world works). Each time element, we'll move the object the amount of the velocity, and we'll increment the velocity by the amount of the acceleration. This isn't a very accurate simulation of a falling object, but it's enough for demonstration purposes.

For example, let's say that we would run our discrete simulation every second. Let's say that velocity was currently 10 and the acceleration

was 3. We say that the object is falling 10 pixels per second, with an acceleration of 3 pixels per second per second (that is, the velocity increases by 3 pixels per second at each iteration, which occurs every second). When the next second goes by, we add to the velocity so that it's 13 pixels per second, and we move the object 13 pixels (because that's the velocity). And so on.

We'll also create a *Kick* object. When the object is kicked, we'll imagine that the object has been kicked up a few number of pixels, and it's velocity again goes back to zero. Strictly speaking, an upward push on the falling object would result in an upward velocity that would decrease as gravity pulled the object back down. Again, we're simplifying for the sake of a demonstration.

Create three morphs (from the *New Morph* menu, or from the Standard Parts bin, or from the Supplies flap): A **RectangleMorph** (default gray), an **EllipseMorph** (default yellow), and a **TextMorph** (appears in Supplies and Parts as "Text for Editing"). We're going to use the rectangle and text as our Kicker, and the ellipse as our falling object.

We'll start out by creating our Kicker button. Click on the text so that you can edit it, and change it to say "Kick." Now Morphic-select it, and drag it (via the black *Pick Up* halo) into the rectangle (Figure 19). Use the control-click menu to *embed* the text into the rectangle. After you choose the *embed* menu item, you will be asked to choose which morph you want to embed the text into. Choose the **RectangleMorph**. (As we'll see later in this chapter, the other option, a **PasteUpMorph**, is actually the whole Morphic world. It is possible to embed morphs into the desktop of a Morphic World.) Once embedded, they move as one morph (Figure 20).

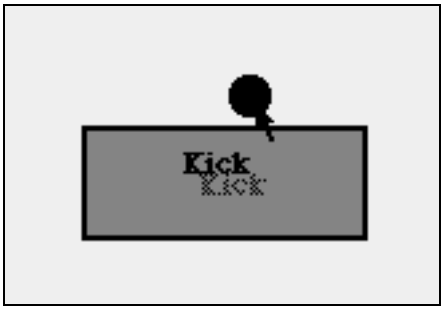

Figure 19: Dragging the TextMorph into the RectangleMorph

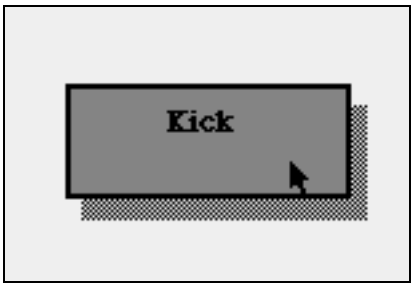

Figure 20: Once Embedded, They Drag Together

Now, let's start programming our two morphs. Morphic-select the ellipse and choose the center left (turquoise) halo, the *View me* halo. When you do, a *Viewer* for the ellipse will open (Figure 21).

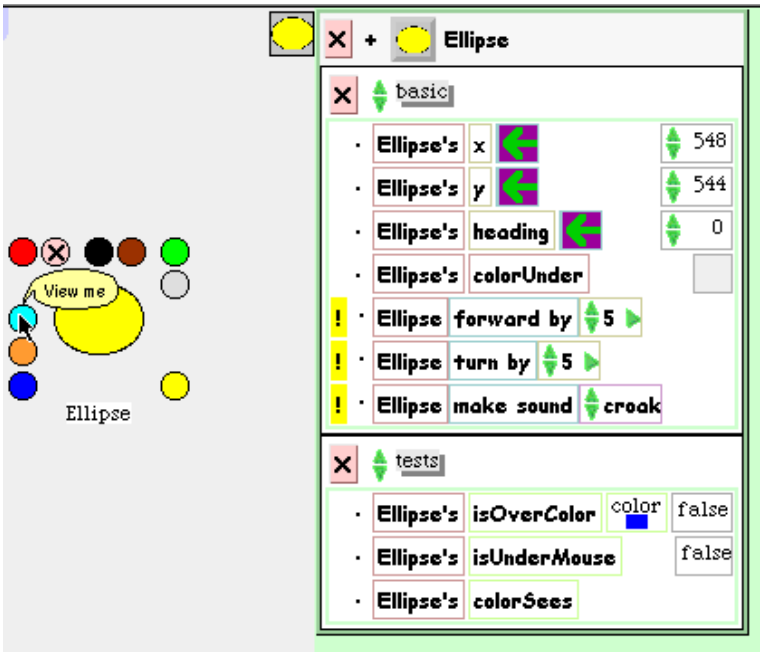

Figure 21: Opening a Viewer on the Ellipse

The Viewer is a kind of browser on a morph. It allows you to create methods for this morph, instance variables for the given morph, and to directly manipulate the morph. Click on one of the yellow exclamation points—whatever the command is (say, *Ellipse forward by 5*) will be executed, and the morph will move five pixels. Directly change the number of the x or y coordinate, and the morph will move.

For what we want to do, change the heading of the ellipse to 180. That means, it's heading will be straight down. That's important because objects fall down. If the heading were zero, our object would fall up.

4.1.1 Adding an Instance Variable

We are going to need a velocity for our falling object, so let's add an instance variable to our ellipse. Click on the small tile of the ellipse inside the viewer itself. (The leftmost tile of the ellipse in Figure 21 is actually a tab. Click on it, and the viewer will slide to the right. Click it again to open the viewer back up.) A pop-up menu will provide a number of programming items, including adding a new instance variable (Figure 22). Choose *add a new instance variable* and enter the name as *velocity.*

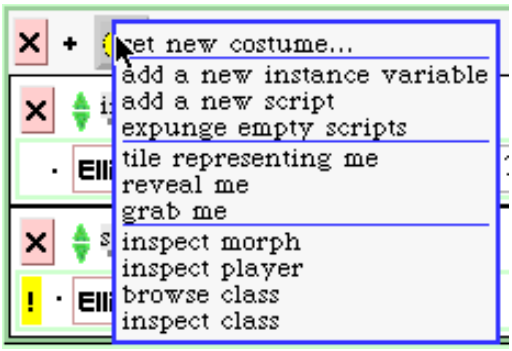

Figure 22: Adding an Instance Variable to a Morph

> ***SideNote***: Take note of what we're doing here: We're adding an instance variable *directly to an instance*, not to the *class*. The Viewer system offers a different kind of object-oriented programming, called *Prototype-based objects*. Each of the morphs is a prototype that can be given variables and methods *directly*. It is possible to then create new instance morphs from these prototypes, and the new morphs will inherit the variables and methods (called *scripts* in the Viewer system). We won't be going that far into Viewers in this book.

The viewer will then update to show the new instance variable (Figure 23). This instance variable can be accessed or set, just like any other instance variable. In a few steps, we'll use it in an equation for changing the velocity by the amount of a gravitational constant.

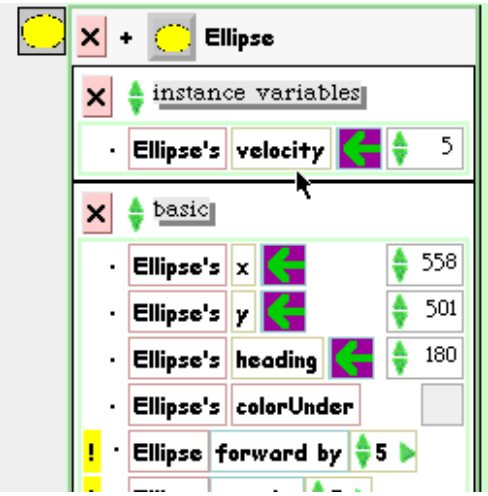

Figure 23: Ellipse's Viewer with the new Velocity Instance Variable

4.1.2 Making our Ellipse Fall

We can then begin to program our falling object. Click on the "forward by" tile and drag it off the viewer.

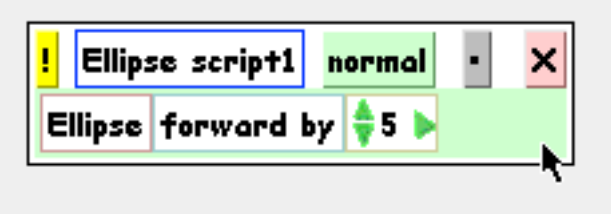

Figure 24: Creating Our First Viewer Script

Let's make this script run all by itself. We'll trigger it upon clicking the mouse down upon the ellipse. Click and hold on the word *normal.* You'll get a pop-up menu of the conditions on which the script should run (Figure 25). Choose *mouseDown* (Figure 26).

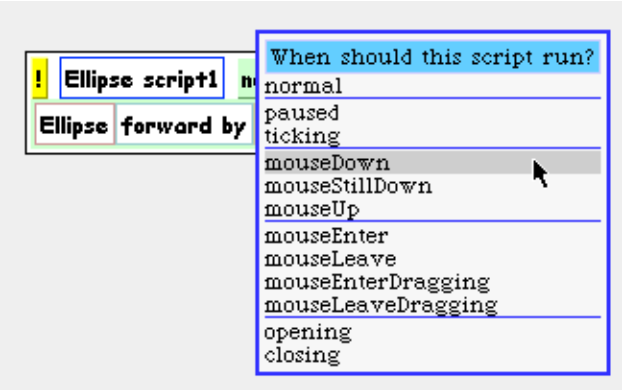

Figure 25: Changing the Conditions of the Script

Figure 26: How the Script Window Changes

Now, click on the ellipse. Each time that you click on it (actually, as soon as you click down on it), it should jump forward five steps. You can play with the amount of the jump in the script1 window to get different amounts of jump.

When an object falls, it should move as much as its velocity, using the simplified model of physics that we're using. So, instead of the constant in the script, we need to reference the velocity instance variable that we've built. That's fairly easily done. Click on the velocity tile in the ellipse's Viewer, and drag it over the constant in the script (Figure 27). Now, when you click down on the ellipse, it moves forward as much as the value of the velocity.

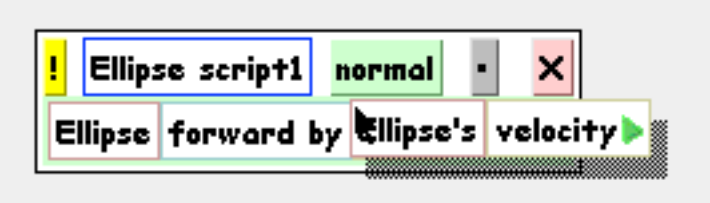

Figure 27: Dragging the Velocity over the Constant

The next step is to make the velocity increase at each time interval. Go back up to the Viewer and click-and-drag on the arrow next to the velocity. You're now grabbing a set of tiles for *setting* the velocity. Drag them into your script window, just above the *forward by* tiles. (You'll find that the other tiles literally move out of your way as you drag in your tiles.) You'll now be setting the velocity to 1 (Figure 28). Now click on

the little green arrow next to the 1. The line will expand to *1 + 1* (Figure 29). Go back up the Viewer and drag the velocity instance variable tile over the second 1 (Figure 30). You've now constructed the falling script. Your rate of acceleration is 1, and velocity will increase by it at each time interval.

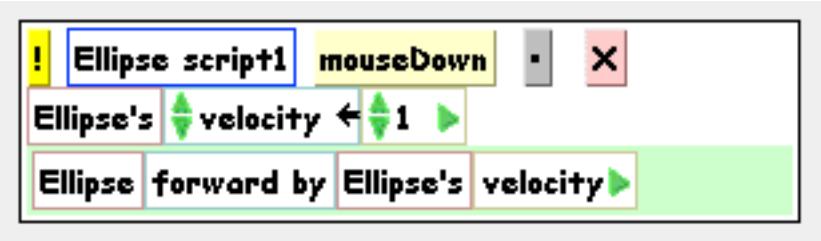

Figure 28: Setting Velocity to 1

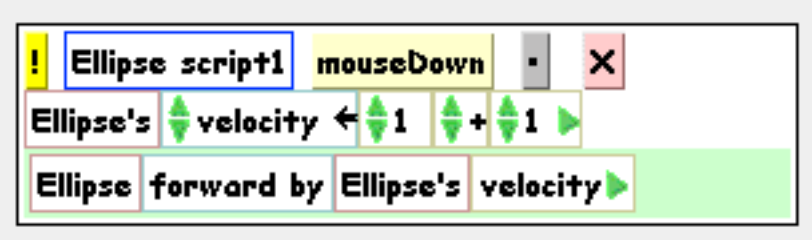

Figure 29: Setting Velocity to 1 + 1

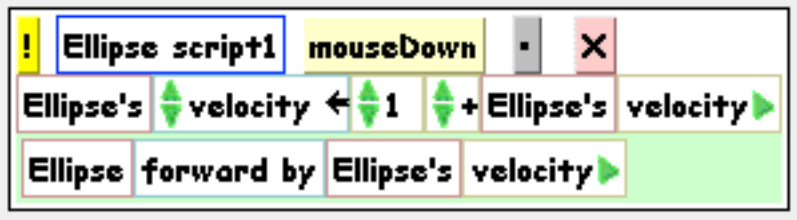

Figure 30: Setting Velocity to 1 + Velocity

You can really make this work now. Change the *mouseDown* trigger on the script to *ticking*. A ticking script fires continuously at a regular interval. (You can change the interval by clicking on the *Ellipse script1* tile and choosing the menu item there.) You will find your ellipse falling ever more rapidly toward the bottom, and then bounce when it gets to the bottom. (That's default Viewer behavior.) You can set the script back to triggering *normal* (which means that it just sits) to stop the falling and to be able to move the ellipse elsewhere.

Feel free to explore different values than 1 for the acceleration constant. You can make small changes by clicking on the up or down arrows next to the 1, or click right on the 1 and type whatever you want. Be careful how large you make it, though! Remember that this value is the amount of change of the *velocity*, so it compounds quickly.

If you want, you can now *name* your script. Click on the *Ellipse script1* tile, and choose *Rename this script* (Figure 31). You might call it *Fall*.

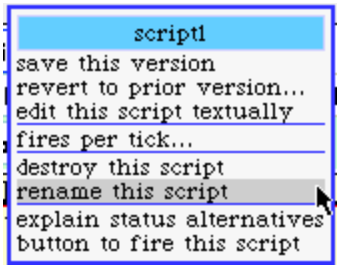

Figure 31: Changing the Name of a Script

4.1.3 Building the Kicker

Now let's build the kicker. Open up a Viewer on your kicker rectangle. Drag out the tile that has the rectangle making a sound, and drop it to make a new script. With this start, whenever we "kick" the ellipse, a sound will be made. Feel free to use the up and down arrows on the sound tile to explore other sounds, and pick the one that makes sense as the "kick" sound to you. (Next chapter, we'll talk about how to record new sounds to use in the sound tile.) Go ahead and make this script work on *mouseDown*. You can click the kick rectangle to hear the sound.

When we kick the object, we should move the object up a few pixels (the effect of our kick), and we should set the velocity to zero. Your final script should look like the top of Figure 32. Set the kicker's script to fire on *mouseDown* and the falling object's script to fire on *ticking*, and you should have a working simulation of a falling object that you can kick.

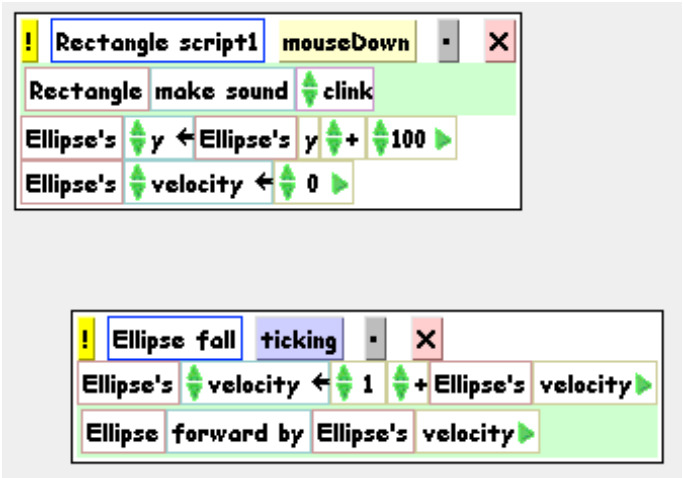

Figure 32: The Final Scripts

You can save these morphs and share them with others as-is. Control-click on any of the morphs and choose *Save morph in file*. You can name

the file, and its file extension will be ".morph". You can send this file to others (via email or even on the Web). Others can load it back in to their image. From the file list, when you select a morph file, your yellow button menu will let you file in the morph and recreate it—scripts and all.

The references between objects may get messed up in this process. For example, the kicker's script will probably need to remapped to the falling object. That's what the *Make A Tile* halo (just under the Viewer halo) is good for. Simply make a tile and drag it into each of the "Ellipse" tiles in the kicker's script.

Exercises: Improving the Viewer Falling Object

9. Should the kick script belong to the kicker or the falling object? We currently have it as the kicker, but maybe the falling object should figure out how it should fall, and the kicker should just tell the falling object to fall. Rebuild the system that way.

10. Our velocity is really the *vertical* velocity. Add *horizontal* velocity to the object. Create a launcher that fires out the falling object at a given vertical and horizontal velocity. If you do it right, the object should fall in an arc. (Remember why from your physics?)

11. Remembering your physics, figure out how you need to set things up, without changing the kicker, such that kicking the object stops it dead.

12. How would you make the falling object fall *up*, that is, fall as if the gravitation pull was from the top of the screen rather than the bottom? (Hint: The gravity's impact in our simplistic simulation is through the acceleration on the object.)

13. Brainstorm a bit over class-based versus prototype-based object systems. When is one an advantage over the other? Consider at least these two scenarios: (1) When prototyping a new object and (2) when maintaining objects that were designed five years ago.

4.2 Programming Basic Variables and Events of Morphs

The previous section gave you a sense of how easy it can be to manipulate morphs. For working through how you want your interface to work, this is a great process. You can quickly assemble a morph that you want, and even test out functionality. However, it gets hard to make many of them, or to create abstractions over them (e.g., subclasses, abstract classes), or to control things like connections between objects. Also, the Viewer system doesn't yet provide all the tools of the text-based programming, such as a debugger.

Typically, you still want to use text to build your more complex systems. The transition between the tiling world and the scripting world isn't as complex as you might think. If you click on the *script1* tile, you

get a pop-up menu that allows you to view your script textually (Figure 33). This provides you the opportunity to see what the mapping is from the Viewer system into the text world.

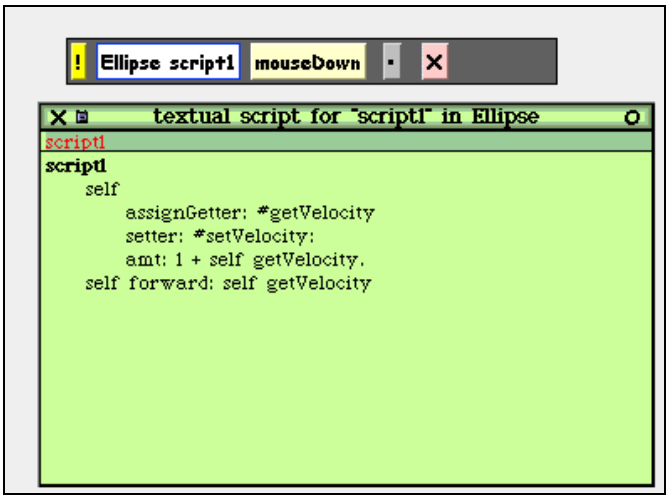

Figure 33: Viewing a Tile Script as Text

But the text world is clearly more complicated than the tile world. We need to know some more things about Morphic in order to dig into programming there. This section introduces the key instance variables, events, and methods needed to program in Morphic.

4.2.1 Instance Variables and Properties

The below table summarizes the main instance variables that are common to every morph. Each of these can be set and accessed using the normal Smalltalk conventions. The **bounds** is accessed using the **bounds** method and set using the **bounds:** method. One of the interesting thing about Morphic is that any change is immediately apparent in the system. Changing the **bounds** makes the morph change its size immediately. You don't have to do any kind of refresh to make it happen.

Instance variable	Meaning
bounds	The rectangle defining the shape of this morph. Change the bounds to resize or move the morph. (**fullBounds** is the bounds of the morph and all of its submorphs. They're most often the same.)
owner	The containing morph. It's **nil** for the World, but is otherwise the morph in which **self** is embedded.
submorphs	The morphs inside me, typically changed

	with **addMorph**:
color	The main color of the morph.
name	Morphs can be named, and that's what shows up at the bottom of the halows when you Morphic-select an object.

The last instance variable, **name**, is a bit of a trick. Yes, you can use **name**: on any morph, but if you look at the definition of the class **Morph**, you won't find **name** there. Instead, there is another instance variable named **extension** that refers to an instance of **MorphExtension**, and it is the **MorphExtension** that knows how to be named.

What's going on here is a cost savings technique. Every object on the screen in Morphic is a morph. Morphs must therefore be cheap to have around. Thus, extra things like **name** (not every morph needs a name) are *extensions*. If you set the name of a morph, it will check to see if it has an extension, and create one if it doesn't (see the **Morph** method **assureExtension**), then set the name in the extension. The name accessor asks the extension for the name. This is an example of the *delegation* introduced in Chapter 2.

MorphExtension provides many other instance variables, some of which are:

Instance variable	Meaning
balloonText, balloonTextSelector	Any morph can do **self extension balloonText: 'This is all about me…'** and will set the balloon help for themselves. A morph can also set its **balloonTextSelector** which will be used to access balloon text dynamically.
visible	Determines whether a morph is visible or not
locked	Manipulate with **lock** and **unlock**. A locked morph can't even be selected.
sticky	A sticky morph can't be moved. Change it with **toggleStickiness**.

There are other interesting instance variables in **MorphExtension**, but these are the most critical, save one: **otherProperties**. There is built-in space for additional properties in **MorphExtension**, without having to add additional instance variables.

otherProperties is a **Dictionary**. You can add properties with **setProperty:toValue:** and retrieve them with **valueOfProperty:** and ask if a property were there with **hasProperty:**. The name of a property is typically a symbol, and the value can be anything you want. The properties won't be as fast to access as an instance variable, but this allows for great expandability without ever changing the basic structure of **MorphExtension** instances.

4.2.2 Morphic Events

Programming user interfaces in Morphic is much easier than under the MVC window model. Conceptually, the complicated controller part is built into the toolkit. A handful of predefined user interface events are passed on to morphs that want them. The basic model is that a morph is asked if it would like to handle a particular kind of event, and if so, the event is sent by calling a predefined method in your morph. (**Morph**, of course, defines all of these and will catch them if your subclass doesn't override them.)

The object passed around is a **MorphicEvent**. A **MorphicEvent** understands many of the same things as **Sensor**, but encapsulates the event into an object. You don't poll **MorphicEvent** the way that you do **Sensor**. Instead, you can ask a **MorphicEvent** whether **redButtonPressed** is **true** if it's a mouse event (**isMouse** would return true), or you can ask the **MorphicEvent** what the **keyCharacter** is (if **isKeystroke** is true).

The below table summarizes how to handle the most common kinds of events.

Event you want your morph to handle	How to handle it
MouseDown	Have a method **handlesMouseDown**: which takes a **MorphicEvent** as input, and return **true**. Have a method named **mouseDown**: which takes a **MorphicEvent**, and deal with the mouse down as you wish.
MouseUp and MouseOver (mouse passes over the object)	Similarly, have a **handlesMouseUp**: or **handlesMouseOver**: method, then a **mouseUp**: and **mouseOver**: method.
MouseEnter and MouseLeave	Return **true** for **handlesMouseOver**:, then define **mouseEnter**: and **mouseLeave**:

MouseMove (within the morph)	Return **true** for **handlesMouseDown**: then implement **mouseMove**:
Key Strokes	When your morph should capture keystrokes, return **true** for **hasFocus**, then accept events in **keyStroke**: When the focus is changing, your morph will be sent **keyboardFocusChange**:, **true** for receiving and **false** for losing.

There are more subtleties to the Morphic event handling model. For example, if a morph's extension defines an **eventHandler**, then your events can be delegated to the object referenced by the **eventHandler**. There are also events associated with mouse clicks starting text entry or not, accepting drag-and-drop, and catching whether the mouse is already carrying an object when it enters the bounds of the morph. More details on these can be found in the event handling category of **Morph** instance methods, but the above are the most common cases.

4.2.3 Animation

One of the most interesting things about Morphic is that it makes animated user interfaces very easy to build. To make your morph animate, you need to implement just one method, **step**, and optionally one other method, **stepTime**.

- At regular intervals, the method **step** is called on all morphs. In your morphs' **step** methods, you can change the appearance, update the display, poll a model to ask for its current values, or do whatever else you'd like.
- The default step interval is once a second. **stepTime** can return a different value, which is the number of milliseconds between each time you want **step** to be called.

An easy-to-understand example of using **step** and **stepTime** is the **ClockMorph**. The **ClockMorph** is a subclass of **StringMorph**, and all it does is display the time. The **stepTime** method simply returns 1000—the clock updates once a second (1000 milliseconds). The **step** method simply sets the contents of the string (**self**) to the current time. That's all that's needed to create an updating string with the time.

4.2.4 Custom menus

There is a custom menu associated with each morph, available from the control-click menu and from the red halo menu. You can easily add morph-specific items to this menu, by overriding the method

addCustomMenuItems: aCustomMenu hand: aHandMorph. This method is called whenever the menu is requested by the user (via control-click or red-halo click). Simply use **add:action:**, **add:target:action:**, and **addLine** methods to add additional items to the menu being handed to the method.

Most of the time, you will want to allow your morph's superclass a chance to add its menu items, via **super addCustomMenuItems: aCustomMenu hand: aHandMorph**. But if you'd like to limit the menu items that a user sees, you don't need to call the superclass. The menu will still have many generic **Morph** items in it, though.

For an example menu customization, **ImageMorphs** provide user-accessible manipulations through this method.

```
addCustomMenuItems: aCustomMenu hand: aHandMorph


    super addCustomMenuItems: aCustomMenu hand: aHandMorph.

    aCustomMenu add: 'choose new graphic...' target: self action:
#chooseNewGraphic.

    aCustomMenu add: 'read from file' action: #readFromFile.

    aCustomMenu add: 'grab from screen' action: #grabFromScreen.
```

4.2.5 Structure of Morphic

The Morphic world may be clearer if some of the internal structure is described. It's important to realize that, just as everything in Squeak is an object, everything in Morphic is a morph (i.e., an instance of a subclass of **Morph**). This includes the desktop itself and even the cursor.

The desktop itself, the World, is an instance of the class **PasteUpMorph**. There are many **PasteUpMorph**s around. The Standard Parts Bin and the flaps are also **PasteUpMorph**s. **PasteUpMorph**s are general "playfields" (as some of them are named) which can hold other morphs.

The World **PasteUpMorph** does something very important: It runs **doOneCycleNow** repeatedly. This method updates the cursors, processes user interface events for the given cursor, runs step methods, and updates the display. The method **doOneCycleNow** appears below:

```
doOneCycleNow

    "Do one cycle of the interactive loop. This method is called repeatedly
when the world is running."


    "process user input events"
```

```
self handsDo: [:h |
        self activeHand: h.
        h processEvents.
        self activeHand: nil].

self runStepMethods.
self displayWorldSafely.
StillAlive ← true.
```

Notice that the above paragraph (and above code) make it clear that events are handled *for each cursor*. A Morphic world can have multiple cursors at once. Each is an instance of **HandMorph**. It is **HandMorph** that sends the events to morphs. Because of this implementation, it is possible to have multiple users interacting in the same Morphic world. There is an option under the *Help* menu from the World Menu called *Telemorphic* which lets you connect multiple users to the same image each with their own cursor.

The **HandMorph** provides many core behaviors to Morphic. As can be seen in the above code, it's the **processEvents** method in **HandMorph** which deals with sending the appropriate messages to the appropriate morphs when user input comes in. It's also the **HandMorph** which creates the control-click menu, in the method **buildMorphMenuFor**:. The **HandMorph** puts up the halos, builds the halo menus, and even builds the World Menu. So, if you want to change the halos or core menus of the system, you start by modifying or subclassing **HandMorph**.

The process of displaying the world safely (**displayWorldSafely**) leads to asking each submorph of the world to **drawOn**: the world's **Canvas**. The **drawOn**: method is the hook for creating your own look to Morphs, if you want something different than a composition or slight modification to the base morphs. **drawOn**: takes a **Canvas** object as its argument. An instance of **Canvas** knows how to draw basic objects (like rectangles and ovals) as well as draw arbitrary **Form**s.

4.3 Programming A Morphic Falling Object

Let's re-do the falling object simulation, but this time, from textual Squeak. The idea is to create the same kind of interaction as the Viewer version, but using the Morphic programming structure described in Section 4.2. By creating a textual version, we have objects that we can later build upon in other contexts. This code is on the CD as **programmedFall.cs**.

We won't go through a CRC Card analysis here, because we already know what basic objects we want. We need a kicker and a falling object. We will shift responsibilities a bit from the Viewer version: It's the falling object that knows how to be kicked. The kicker just tells the falling object to kick.

Because the textual version will not have the code as accessible as the Viewer version, we'll need to add some user interface to do the kind of exploration that a user might want to do. Probably the most common manipulation will be to change the gravitational acceleration constant. In terms of responsibility, it seems natural to let the falling object hold a menu item for allowing the user to change the gravitational constant. But given that our falling object will be moving constantly, it's easier on the user to stick it in the kicker.

Just to make the falling object a little more interesting, we'll create it as a subclass of ImageMorph. An ImageMorph can hold any kind of Form, which means that we can have any kind of falling object we may wish. Think about what kind of images you might want to have crashing on your screen, with a clear user interface for kicking those objects.

A UML diagram of our classes appears in Figure 34. We'll create a **KickButtonMorph** as our kick button, and a **FallingImageMorph** as our falling object. The **KickButtonMorph** will keep track of the **ball** that it kicks. It will have hooks into the user interface, for the gravity-setting menu item (**addCustomMenuItems:hand:**) and for capturing button clicks (**mouseDown**:). The **FallingImageMorph** will keep track of its **gravity** (more correctly, the constant acceleration due to gravity) and **velocity**, provide setters and getters for these, and implement a **kick** method. It will have a **step** method where it will implement falling.

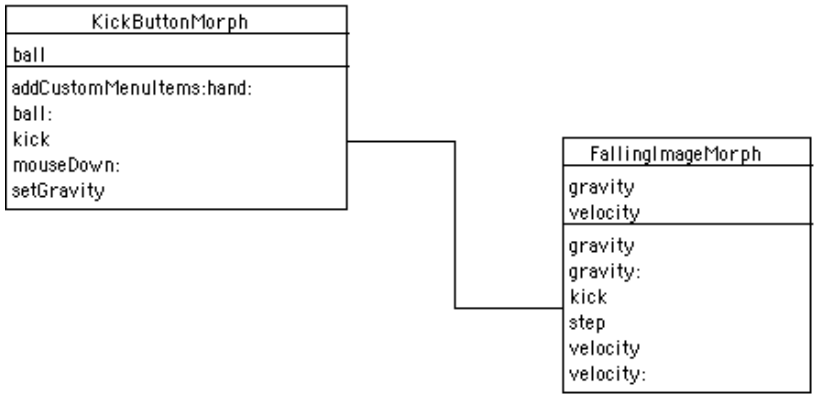

Figure 34: UML Diagram for Textual Falling Object Simulation

We can now begin implementing our classes with some class definitions. While we said that the falling object would be a subclass of

ImageMorph, we didn't talk yet about what the kicker would be subclassed from. A good solution is to do in code just what we did via direct manipulation of the morphs: We'll start from a **RectangleMorph**. We'll override **initialize** so that our **KickButtonMorph** gets the label that we want.

Notice that there *is* a **SimpleButtonMorph** that would make sense to subclass from. Similarly, there are many button subclasses that would be useful to explore and subclass. However, they make it *too* easy—if we used one of those, we would never deal with mouseDown or setting our own label. We would only provide an action method. While that's what you'll do in normal practice, we'll unpack the details a bit here to show better how the button is constructed.

```
ImageMorph subclass: #FallingImageMorph
        instanceVariableNames: 'velocity gravity '
        classVariableNames: ''
        poolDictionaries: ''
        category: 'Morphic-Demo'
RectangleMorph subclass: #KickButtonMorph
        instanceVariableNames: 'ball '
        classVariableNames: ''
        poolDictionaries: ''
        category: 'Morphic-Demo'
```

4.3.1 Implementing the Falling Object

Let's start out by implementing the basic falling procedure. We know what this looks like from our Viewer implementation, and we know from our discussion of Morphic animation that we fall in a **step** method. Falling is a process of incrementing the velocity by the acceleration due to gravity, and then moving the object down by the amount of its velocity.

```
step
        velocity ← velocity + gravity. "Increase velocity by gravitational constant"
        self bounds: (self bounds translateBy: (0@(velocity))).
```

As mentioned earlier, the position and size of a morph is determined by its **bounds**. If we move the **bounds**, we move the object. The **bounds** is a **Rectangle**. To move a rectangle is to *translate* it, and the method **translateBy:** handles the translation. The amount of translation is a **Point**: The amount of horizontal translation and the amount of vertical translation. To move an object down, then, we translate it by **0 @ velocity**.

We don't want the step to happen too often, so we'll provide a **stepTime** method. We'll use one second as the step interval, so that our velocity is in the simple units of pixels per second, and our gravity constant is pixels per second per second.

```
stepTime

    "Amount of time in milliseconds between steps"

    ^1000
```

Next, we need the ability to kick the object. Kicking, as we defined it earlier, sets the velocity back to zero and moves the object back up 100 pixels. Again, this is a translation, where the vertical coordinate is negative because it's a move up.

```
kick

    velocity ← 0. "Set velocity to zero"

    self bounds: (self bounds translateBy: (0@(100 negated))).
```

Finally, let's provide an initialize method that sets the velocity and acceleration to a reasonable state.

```
initialize

    super initialize. "Do normal image."

    velocity ← 0. "Start out not falling."

    gravity ← 1. "Acceleration due to gravity."
```

We will need methods for getting and setting the gravity, if not the velocity, too. Those are left as an exercise for the reader.

4.3.2 Implementing the Kicker

The main requirement for the kicker is that it be able to kick an object, so let's begin with that. We'll trigger the kicking action on mouse down, which means that we have to announce that our morph will handle mouse down, then provide a **mouseDown**: method.

```
handlesMouseDown: evt

    "Yes, handle mouse down"

    ^true

mouseDown: evt

    self kick.
```

Kicking is pretty easy when the kicked object implements the kicking.

```
kick

    ball kick.
```

That's enough to allow for kicking. We'll need an ability to set the ball to be kicked (**ball**:), but that's actually enough to start our simulation.

However, if we created our objects right now, our kicker would only be a raw rectangle without a label. If we want to have a different look, we should override the **initialize** method.

The initialize method first does whatever rectangles do for initialization, then sets up a label. Our label will be a string (StringMorph) saying “Kick the Ball.” StringMorph’s know their size (extent), so we’ll set the kicker’s extent to match it. Then we’ll add the string into the rectangle, and place the center of the button wherever the mouse is.

```
initialize
        | myLabel |
        super initialize. "It's a normal rectangle plus..."

        myLabel ← StringMorph new initialize.
        myLabel contents: 'KickTheBall'.
        self extent: (myLabel extent). "Make the rectangle big enough for the label"
        self addMorph: myLabel.

        self center: (Sensor mousePoint). "Put it wherever the mouse is."
```

4.3.3 Running the Text Falling Simulation

In a workspace, we can now run our simulation. We need to create each object, initialize it, and open it in the world. We need to tell the kicker what its ball is. We’ll set the form for the falling object to be selected by the user, so when you execute the below code, you’ll have to click and drag a rectangle of interesting display before it’ll run. (Feel free to replace that with a form of your own choosing.)

```
aBall ← FallingImageMorph new initialize.
aBall newForm: (Form fromUser). “Here’s where you select a form”
aKicker ← KickButtonMorph new initialize.
aKicker ball: aBall.
aBall openInWorld.
aKicker openInWorld.
```

With this, you can bounce the ball around (Figure 35). (Though, it probably doesn’t look like a ball, unless you selected one.) However, all you can do is bounce the ball here—not much more exploration than that.

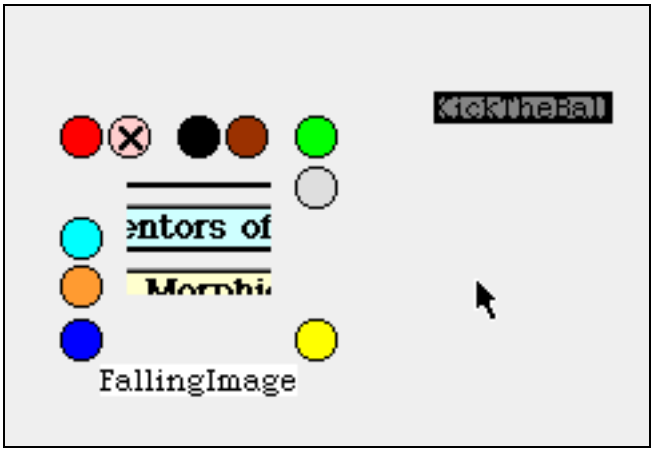

Figure 35: FallingImageMorph and KickButtonMorph

4.3.4 Changing the Gravitational Constant

As seen in our original design, we plan to make a menu item available for changing the gravitational constant for the falling object. We can do that pretty easily. First, we add it to the control-click menu.

```
addCustomMenuItems: aCustomMenu hand: aHandMorph
    super addCustomMenuItems: aCustomMenu hand: aHandMorph. "Do normal stuff"

    aCustomMenu add: 'set gravity' action: #setGravity.
```

Then, we provide a method for setting the gravity. Setting the gravity will use a FillInTheBlank to let the user know what the current gravity is and to input a new gravity. The gravity is a number, but FillInTheBlank accepts an initial answer and returns a string, so we need to convert.

```
setGravity
    "Set the gravity of the ball"
    | newGravity |
    newGravity ← FillInTheBlank request: 'New gravity'
                                initialAnswer: ball gravity printString.
    ball gravity: (newGravity asNumber).
```

Now, try control-clicking on the kicker and changing the gravity for the falling object.

5 Generating: Using Morphs that You Haven't Met Yet

Squeak 2.6 contains 225 ancestors of the class Morph (determined by PrintIt on **Morph allSubclasses size**), and more appear with every new fileIn, new release, and new update. Documentation for each and

every morph will fill a book of this size—and would be obsolete almost as soon as it was published. Therefore, it's important to figure out how to use morphs that you haven't used before.

Here is one useful strategy:

- Just like for any other object, start out by checking out its class comment (if any) and class methods. Example methods help a lot, but even class methods that create useful instances can tell you a lot about using a morph.
- Find someplace where the morph is being used. A good way to do that is to figure out an instance or class method that would almost certainly be necessary to use the morph well (an accessor method, perhaps), then find all of its Senders.
- Finally, just try it. Send the class **new initialize openInWorld** and see what happens. If you get an error message, it will probably relate to something missing. By tracking back through the errors, you can probably figure out what it expects.

Let's use this strategy on a couple of morphs and see how well it works. We'll figure out how to use the **PolygonMorph**. There is no class method, and there are no example methods. But there are two instance creation methods. One of them looks like this:

```
vertices: verts color: c borderWidth: bw borderColor: bc

    ^ self basicNew vertices: verts color: c borderWidth: bw borderColor: bc
```

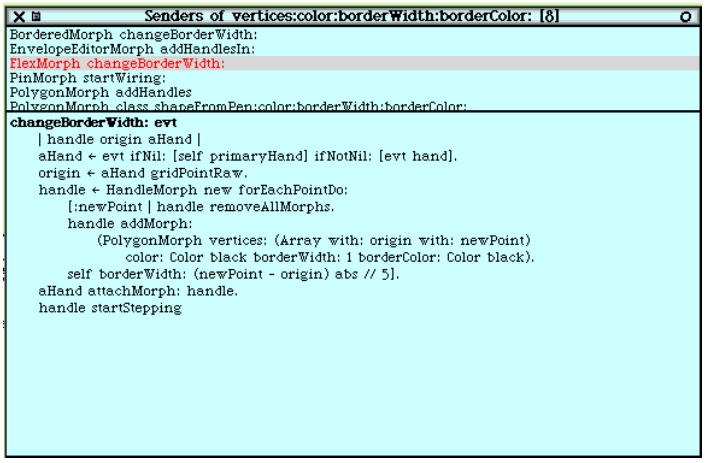

Figure 36: Senders of PolygonMorph Instance Creation Message

By requesting the senders of this message, we can find examples of objects using **PolygonMorphs** (Figure 36) It looks like the vertices are an array (probably of Points), the colors are just instances of Color, and the border width is an integer. So we can go ahead and try it with some workspace code, just making up values for the parameters. If we get it

wrong, we'll get a debugger that will let us play with values. But even the first try creates a real polygon (Figure 37).

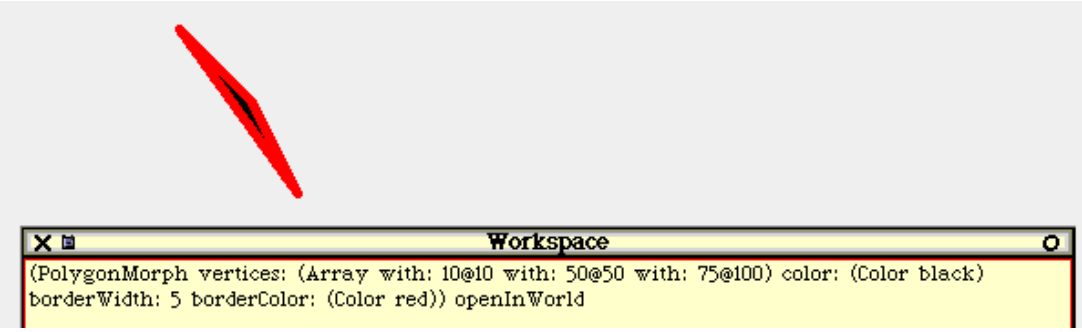

Figure 37: Testing Out PolygonMorph

Exercises

14. Without changing any of the code (that is, using only the interface provided for the user), how do you make the falling object in the text version completely stop?

15. Extend the text falling object so that objects have horizontal velocity, too. Add a Cannon button that fires the falling object with a specified horizontal and vertical velocity.

16. In the real world, each object does not have its own gravitational constant. A famous example was created by Randy Smith (who later co-developed Self and Morphic!) called *ARK*, Alternate Reality Kit. In the ARK, all objects had the same gravitational constant, as well as constants relating to friction with the desktop. All of the constants could be lowered or raised at once. Try implementing a piece of ARK: Create global gravitational constants, with several falling objects that manipulate them. Perhaps you might also allow kickers to manipulate falling objects that they are touching or they are nearest to. (Hint: **World submorphs** lists all the objects in your current world. Their **bounds** gives you their locations.)

17. Create an object that continually updates (via **step**) but responds to user actions. Create an eyeball (a dark ellipse inside a light ellipse) whose position moves the pupil toward the current position of the mouse. (Hint: Sensor still works in Morphic.)

18. Try to figure out some of the other useful morphs built-in to Morphic, such as **GraphMorph** and **JoystickMorph**. Create a simple tool that will take a minute of positions from the **JoystickMorph** and plot them in the **GraphMorph**.

19. There already is a **SimpleSliderMorph** in Squeak, but it's only vertical and doesn't allow us to easily change the look of the slider (e.g.,

have the slider wider than the track). Create a more powerful **SliderMorph**.

References

John Maloney recommends this paper as a good description of the design philosophy of Morphic.

The Self-4.0 User Interface: Manifesting a System-wide Vision of Concreteness, Uniformity, and Flexibility, 1995, Randall B. Smith, John Maloney, and David Ungar
http://self.sunlabs.com/papers/self4.0UserInterface.html