Building User Interfaces

Chapter 5

Squeak: Object-oriented design with multimedia applications

Story

- What a UI toolkit does: Iteratively building a Clock UI without one
- Pluggable UI in Squeak: MVC and Morphic
- Using Morphic

Challenges of O-O UI Design

- Two key questions:
  - How do you create user interface software that you can maintain, is well object-oriented, is easy to build, and is easy to change?
  - How do you create user interfaces that people can actually use?
- The first is our focus here, and is MUCH easier than the second

MVC: Model-View-Controller

- Key idea in UI Software
  - Models define the world
  - Views are what the users see
  - Controllers handle user input (low-level: mouse, keyboard, etc.)
- Hard to use, but good for engineering
- Other models: Merge all three
  - Easier to build, harder to maintain

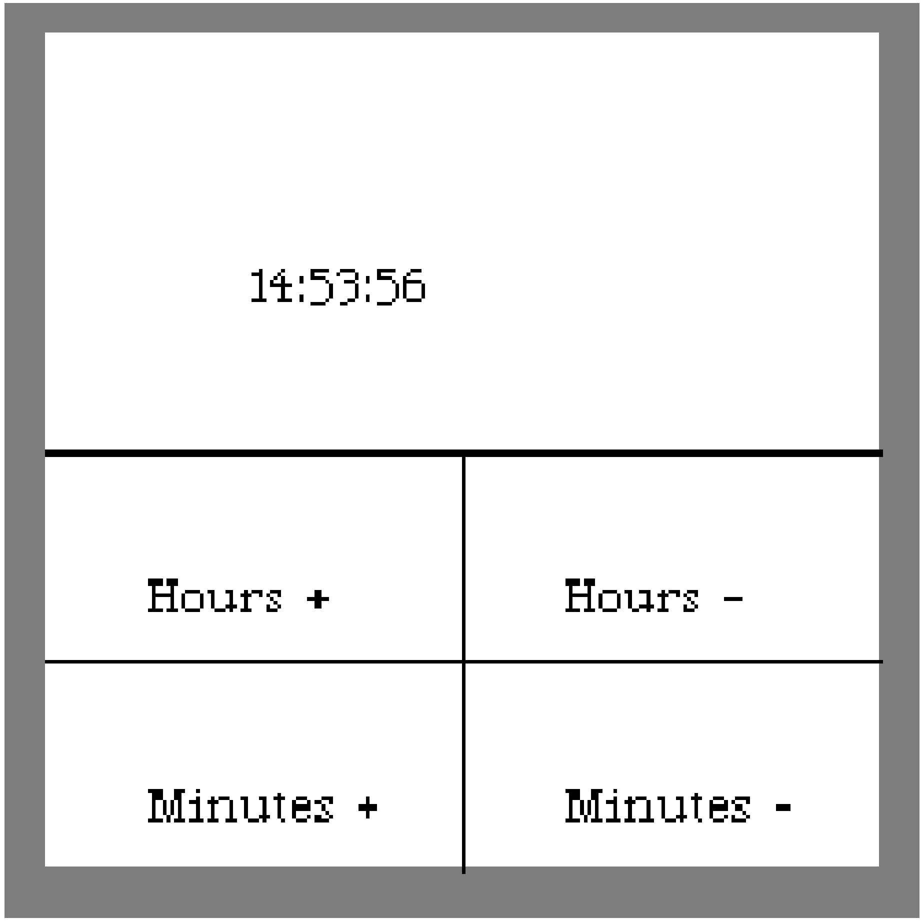
MVC and Morphic

- Squeak supports multiple models of UI building
  - Can do raw polling of Sensor and posting to Display
  - Can code in basic MVC structure
  - Can code in pluggable structure for both MVC and Morphic
  - Can code in Morphic structure
    - Controller embedded in the World.
    - Models and Views can be merged

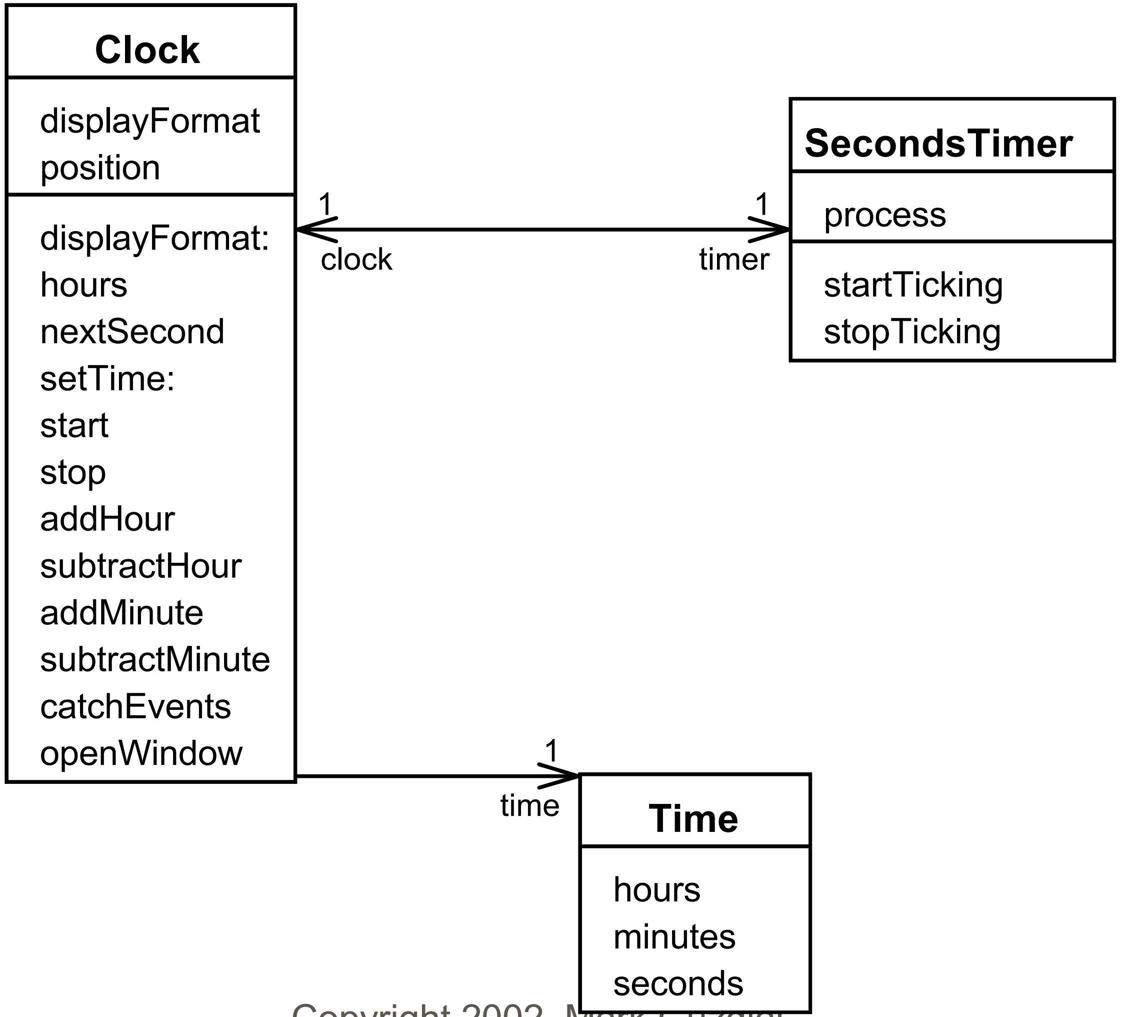
Why we want MVC

- What if you build an interface for Clock, and later want it to be an AlarmClock?
- What if you build a digital clock face, but later want the analog form?
- Can we create a system where:
  - We can swap the model out, and the view stays the same
  - We can change the view, and the model remains the same
- How little do the Model and View have to know of each other?

Clock UI we're going to build



Round #1: Munge it all



Opening a Window

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

Opening a Window, Part 2

```
"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.
```

Opening a Window, Part 3

```
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).
```

Displaying Time

```
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).
```

An Event Loop

- Core to most modern user interfaces
- Basically
  - Is there a user event? If so, get it.
  - Who needs it? (focus of control)
  - Pass on the event
- Absolutely critical to shift agency from computer to human

Our First Event Loop

```
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.
```

Our First Event Loop, Part 2

```
"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 up."
    (Delay forMilliseconds: 500) wait.
```

Our First Event Loop, Part 3

```
      (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].]].
```

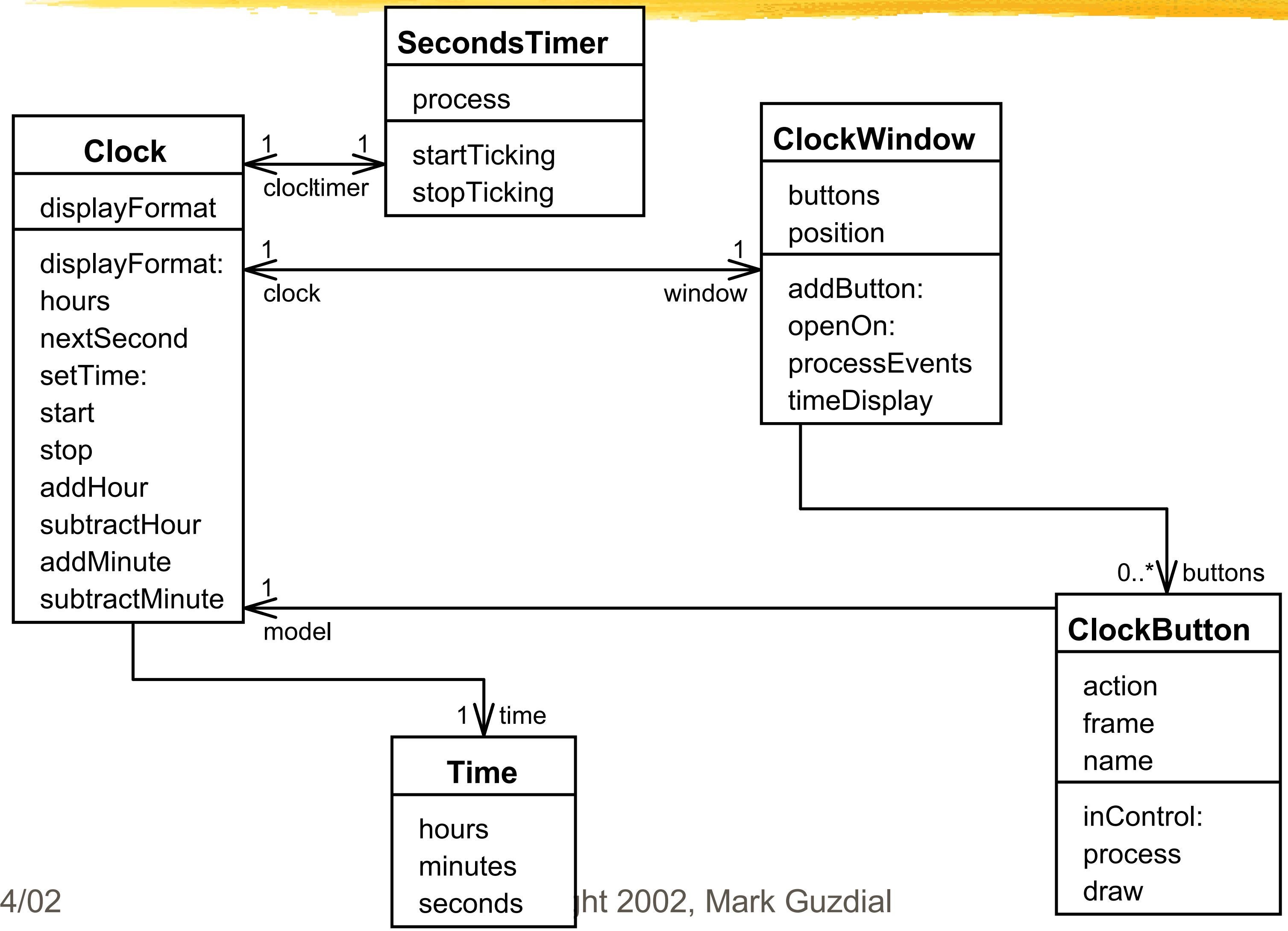
Running the Code

```
c := Clock new.
c position: 100@10.
c setTime: (Time now printString).
c openWindow.
c start.
c catchEvents.
“Don’t forget c stop!”
```

Critique of Round #1

- Modified Clock to add user interface
  - Do real clocks have positions on the screen?
- It's impossible to maintain
  - Swap from digital to analog? Start over!
- Clock has too much responsibility
- Absolutely nothing reusable here

Round 2:
ClockWindow and ClockButton



SecondsTimer

process

startTicking
stopTicking

Clock

displayFormat

displayFormat:
hours
nextSecond
setTime:
start
stop
addHour
subtractHour
addMinute
subtractMinute

ClockWindow

buttons
position

addButton:
openOn:
processEvents
timeDisplay

ClockButton

action
frame
name

inControl:
process
draw

Time

hours
minutes
seconds

Details

- ClockWindow
  - Handles *position*, *timeDisplay*, and *processEvents* from Clock
  - Still needs to know Clock for displaying
- ClockButton
  - Knows its *model*, *action*, *frame*, and *name*
  - Knows how to *draw*, *process*, and whether its *inControl*.

Opening Windows in Round #2

```
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.
```

Opening Windows in Round #2, Part 2

```
"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.
```

Adding a Button, lazily

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

Processing UI Events

```
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]].]].
```

Making Buttons

```
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

```
draw
    "Just like Round #1, but now in ClockButton"
    | 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"
```

inControl and process

- The "hardest" parts are actually the smallest and easiest

inControl: aPoint

"If the point is in the frame, have control"

^frame containsPoint: aPoint

process

"Tell the model to do the action"

model perform: action

Displaying Time is still Yucky

```
timeDisplay "In ClockWindow"
    "ClockWindow asks Clock for time"
    '          ' displayAt: position + (50@50).  "Erase"
    (clock display) displayAt: position + (50 @ 50).

nextSecond "In Clock"
    "Clock tells ClockWindow when"
    time := time addTime: (Time fromSeconds: 1).
    window timeDisplay.
```

Running Round #2

```
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.
```

Critiquing Round #2

- Clearly, much nicer separation between view and model
- ClockWindow and ClockButton (except for the name) are pretty darn generic
- But text update is still a problem
  - Why does Clock need to know its view?
  - Why should the window have hard-coded a request to its clock?

Solution: Dependents and change/update

- A view becomes **dependent** on its model
  - model addDependent: view
- A model can announce a **change** in some *aspect* of itself
  - self changed: #aspect
- Dependent views are asked if they would like to **update** based on the given *aspect*
  - dependents do: [:each |
    each update: #aspect].

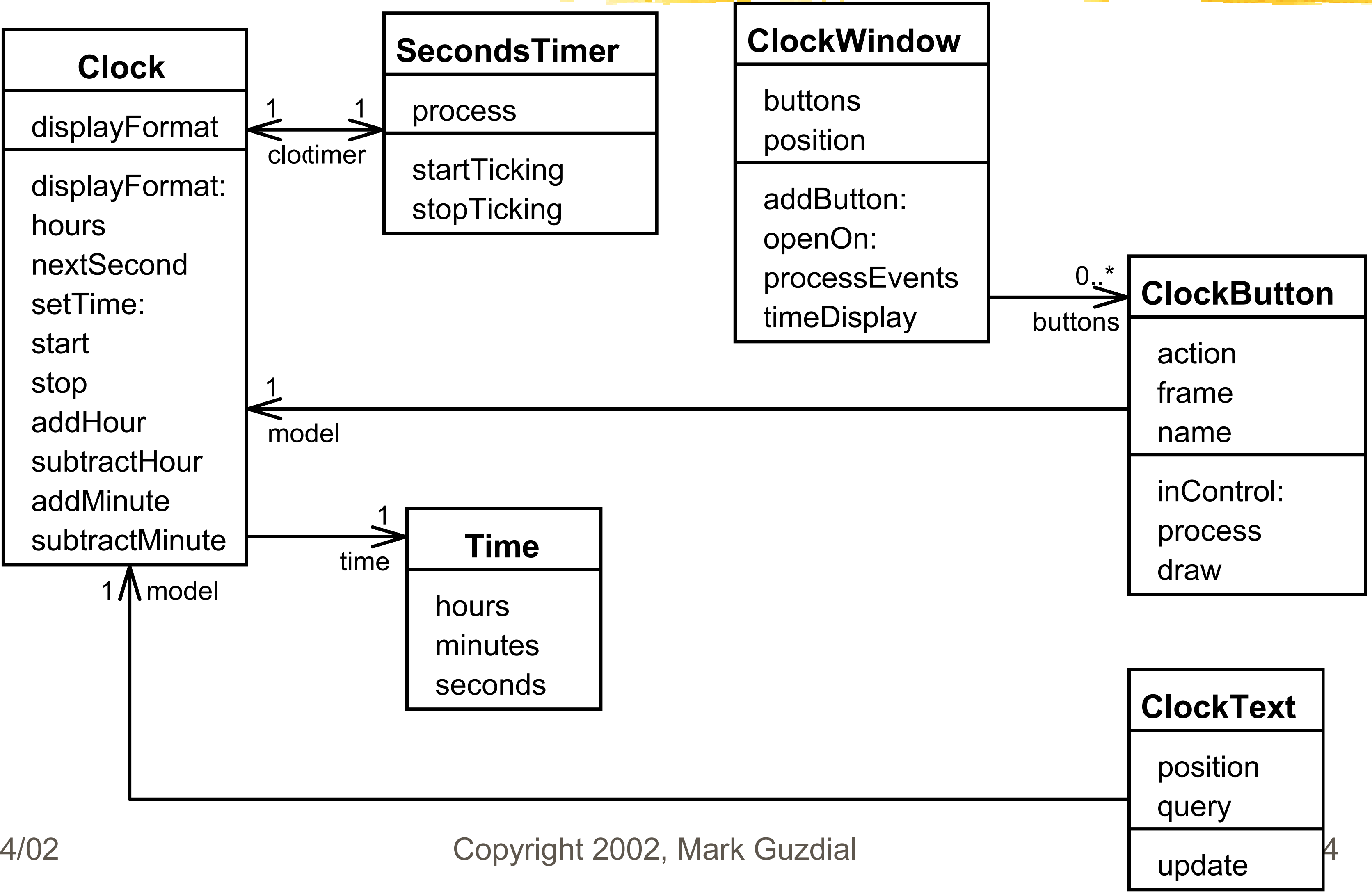
Change/update and Dependents buys us Flexibility

- Can have any number of views on same model
  - E.g., views for Doctors, Nurses, Billing Office all on same Patient model
- Views can update only on aspects they care about
  - self changed: #testResults vs. self changed: #prescription

Decreases information sharing

- Announcing a changed: is cheap
  - Do it often, whether or not a view may care about that aspect
- Models don't have to manage their dependents
  - A general dependents dictionary stored in the system
  - Can subclass Model instead of Object for more efficiency
- Views need to know their model and the aspect of the model that they care about

Round #3: Adding ClockText



Clock
displayFormat
displayFormat:
hours
nextSecond
setTime:
start
stop
addHour
subtractHour
addMinute
subtractMinute

1
1
clotimer

SecondsTimer
process
startTicking
stopTicking

ClockWindow
buttons
position
addButton:
openOn:
processEvents
timeDisplay

0..*
buttons

ClockButton
action
frame
name
inControl:
process
draw

1
model

1
time

Time
hours
minutes
seconds

1
model

ClockText
position
query
update

Round #3: Notable for What's Gone

- Clock doesn't know window
- ClockWindow doesn't know its clock
  - ClockWindow won't even know its text!
- ClockText and ClockButtons know their models

How a Clock does nextSecond

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

ClockText is dependent on its Clock

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

ClockText handles update:

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

Creating a ClockText

- ClockText class method

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

Round #3: Opening a 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.
```

Opening a Window, Part 2

```
"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.
```

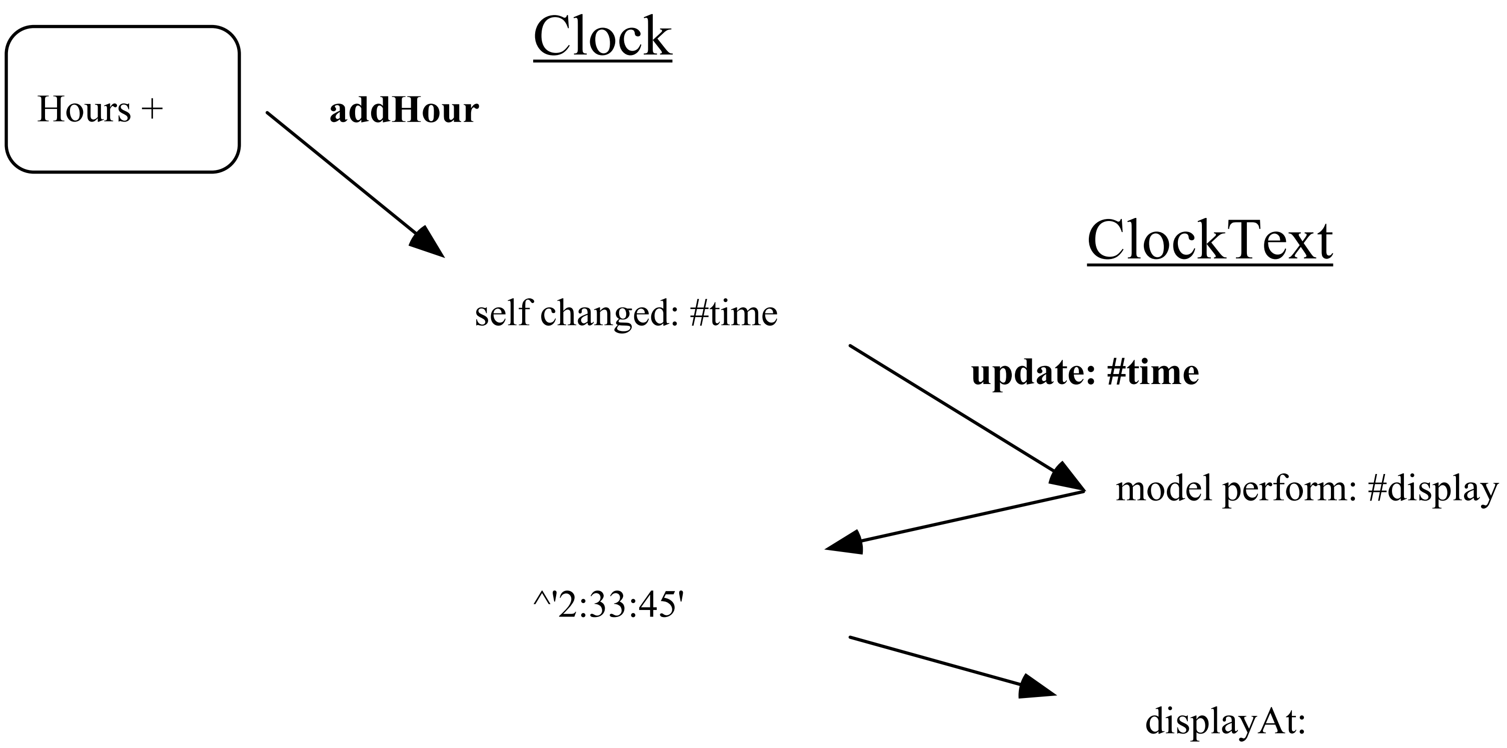
Done with Clock UI Rounds!

- Note: YOU WILL PROBABLY NEVER NEED TO WRITE CODE LIKE THIS!
  - No update:, but probably changed:
  - Probably never write your own event loop
- But now you know what's *inside* the toolbooks you use

Strengths and Weaknesses of MVC

- Strengths
  - Clean O-O structure: Minimizes information sharing, easy to maintain
  - Can support multiple views on same model
- Weaknesses
  - Inefficient: Trace how an update occurs
  - Especially inefficient for multiple views
  - One view on multiple models breaks down
    - Introduce ApplicationModel
- Research: Maintain the good parts, optimize in the system

Tracing an MVC Interaction



Clock

Hours +

addHour

self changed: #time

ClockText

update: #time

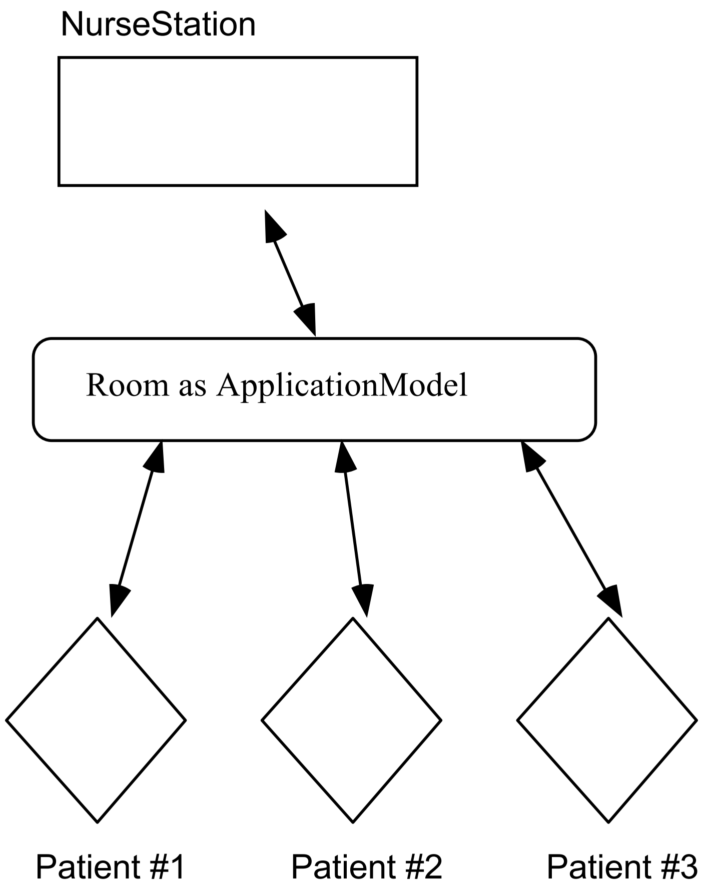
model perform: #display

^'2:33:45'

displayAt:

The Need for ApplicationModels

- When you have a view drawing from multiple models, managing which model did the update is a Responsibility.
- Delegate the Responsibility to a new Model whose role is just that



Pluggable User Interfaces

- ClockButton and ClockText are "pluggable"
  - We simply set the *model* and the *query* and use *perform:*
  - The key parts have become "plugs"
- Pluggable interfaces are easier to use, but less flexible
  - The decisions of what can be sent between the view and the model have been made for you

Alternative to Pluggable

- The class Button knows how to *draw*, respond if *inControl*, and *process*
  - But *process* does nothing in the superclass
  - AddHourClockButton defines *process* as:

```
process
    model addHour
```

  - SubtractMinuteClockButton defines *process* as:

```
process
    model subtractMinute
```

  - Observation: Only difference is in action message

Pluggable UI Objects in Squeak

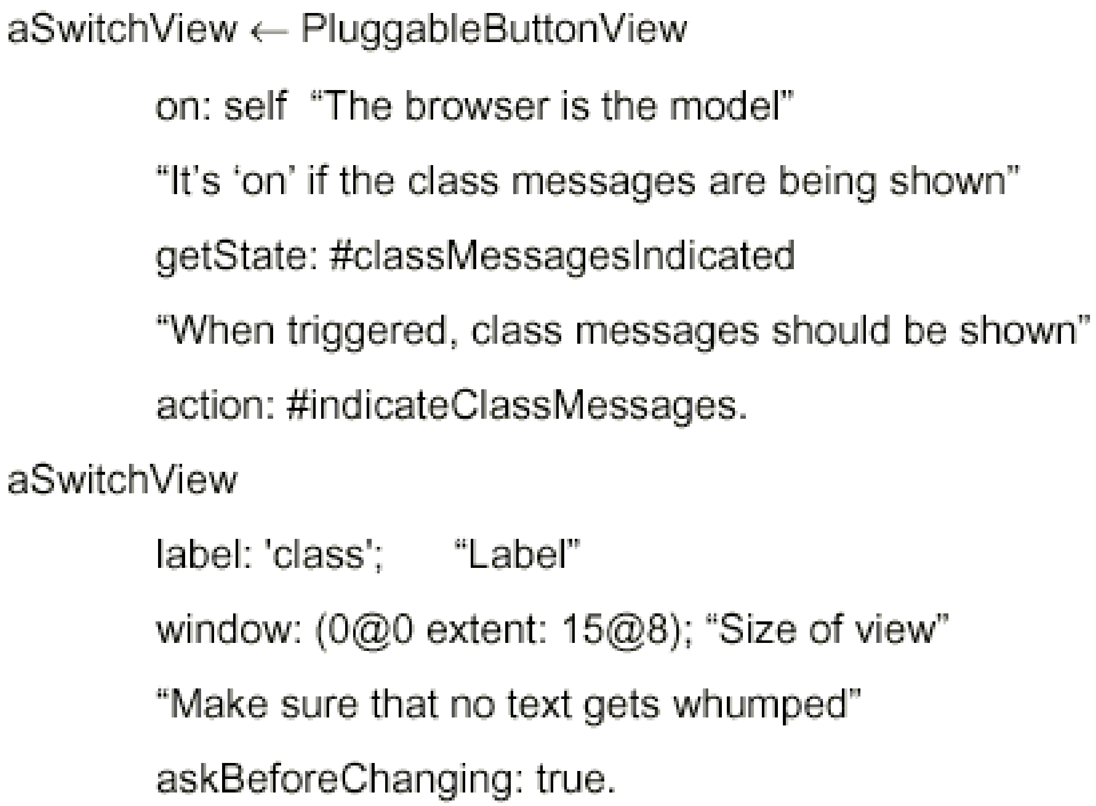
- Three key ones: Buttons, Text, and List
- Each defines a set of *selectors* that can be sent from the view to the model
- Each works in both MVC and Morphic
- All limit you to announce *changed*: to only **defined** selectors.

PluggableButtonView (PluggableButtonMorph)

- Selectors/aspects: state and action
  - Am I on or off?
  - Here's what you should do when I get clicked.

PluggableButtonView Example

- From Browser's class button



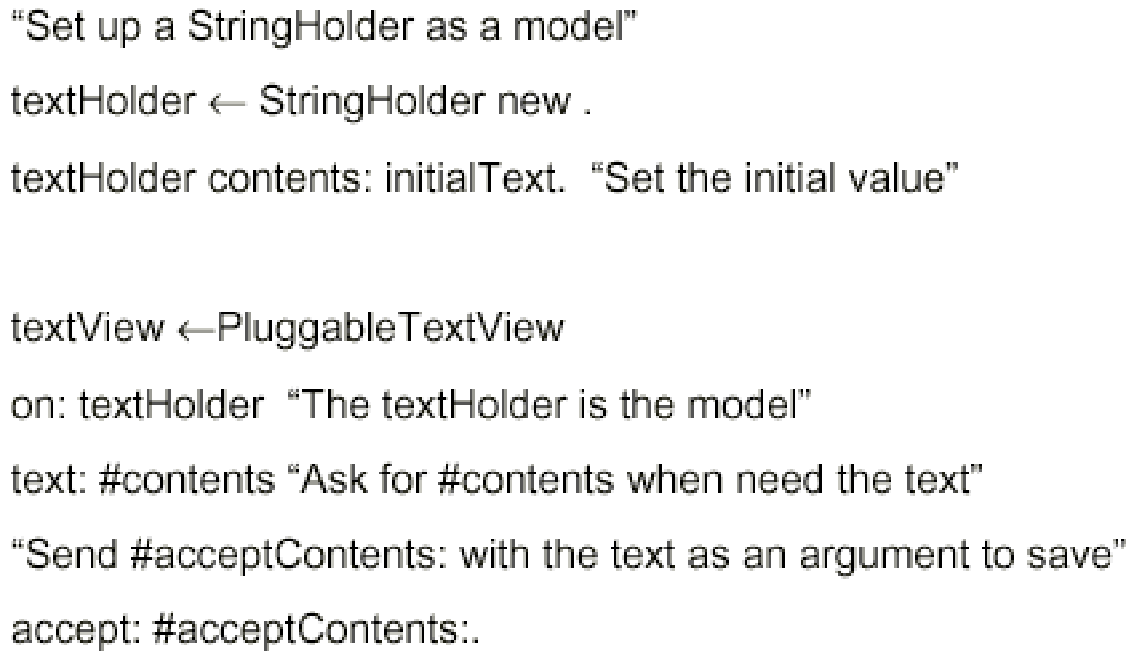
```
aSwitchView ← PluggableButtonView
        on: self  "The browser is the model"
        "It's 'on' if the class messages are being shown"
        getState: #classMessagesIndicated
        "When triggered, class messages should be shown"
        action: #indicateClassMessages.
aSwitchView
        label: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

PluggableTextView/Morph

- Four selectors/aspects:
  - Retrieve text from model
  - Submits new text to model (nil = Read Only)
  - Current text selection
  - Yellow-button menu

PluggableTextView example

- From Celeste:



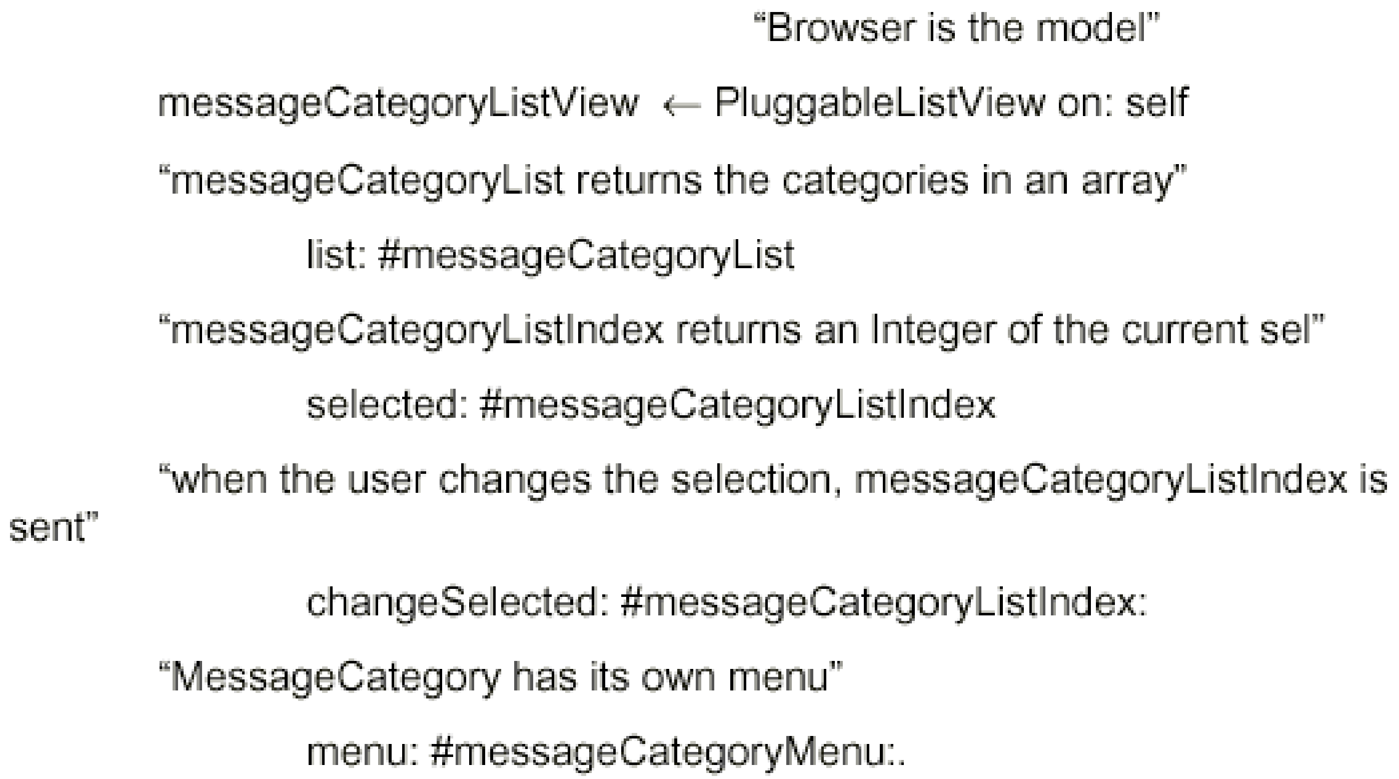
```
"Set up a StringHolder as a model"
textHolder ← StringHolder new .
textHolder contents: initialText.  "Set the initial value"

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/Morph

- Selectors/aspects:
  - Contents of list
  - Currently selected item
  - Set current selection
  - Yellow-button menu
  - Keystroke handler

PluggableListView example

- Browser message category list:



```
                                        "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:.
```

Simple Text Example Here

- m := MyModel open.
- m gobbledygook "'Here is some text.'"
- m add: 'Here is MORE text.'.
- m changed.

Building a Pluggable Clock: Clock must change slightly

- Aspect symbol must equal query message

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

ClockWindow openAsMorph for Morphic

```
openAsMorph
    | win component 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.
```

openAsMorph, Part 2

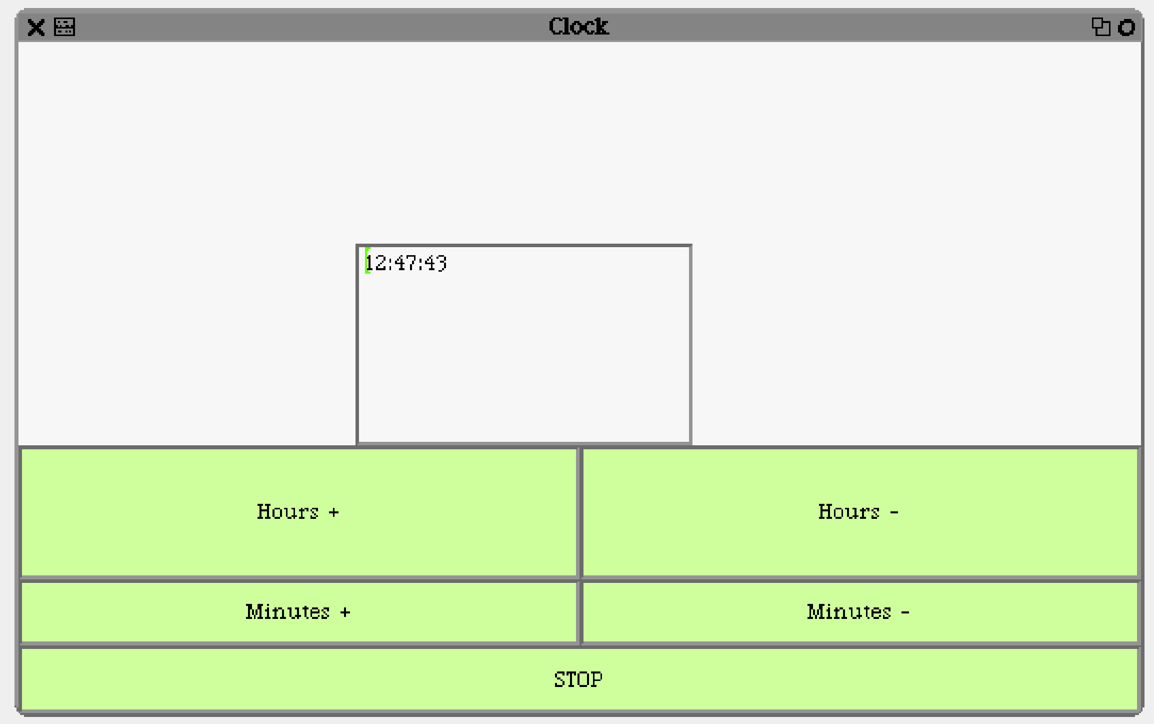
```
"Set up the text view and the various pieces"
component := PluggableTextMorph on: clock text: #display accept: nil.
win addMorph: component frame: (0.3@0.3 extent: 0.3@0.3).

component := PluggableButtonMorph new
        model: clock;
        action: #addHour;
        label: 'Hours +';
        borderWidth: 1.
win addMorph: component frame: (0@0.6 extent: 0.5@0.2).
```

openAsMorph, part 3

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

    win openInWorld.
    ^win
```

Pluggable Clock UI in Morphic

- w := ClockWindow new.
- w openAsMorph.

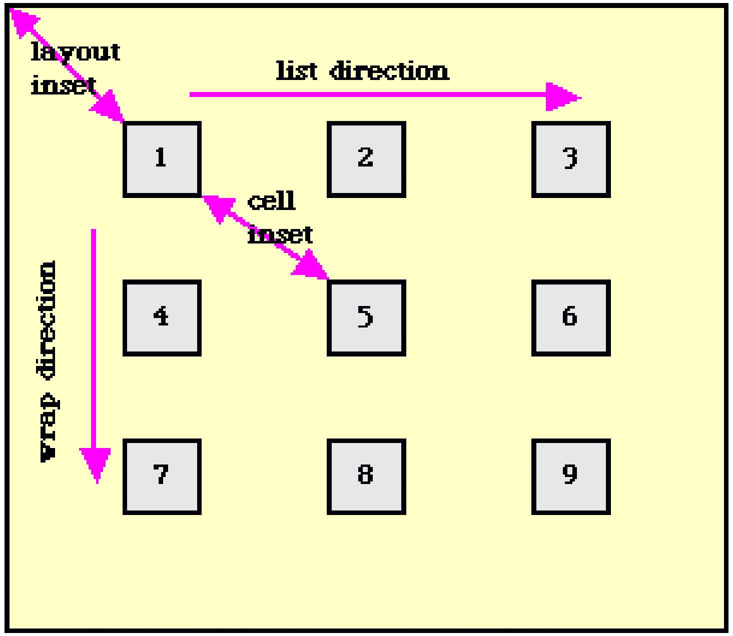


What if you want to control where things go?

- Every Morph has a LayoutPolicy (set with `layoutPolicy:`)
  - SystemWindows by default use a ProportionalLayout (`layoutPolicy: (ProportionalLayout new)`) which allows for fractional positioning
  - Any Morph can also use a TableLayout (`layoutPolicy: (TableLayout new)`) which can lay things out dynamically.
  - AlignmentMorphs provide some default class methods for creating well-formed layouts, like columns and rows.

More on TableLayouts

- Table layouts dynamically position things as they're added
- They are inset from the edges (`layoutInset:`) and from each other (`cellInset:`)
- They define adding in one-dimension (`listDirection: #leftToRight`) and two-dimensions (`wrapDirection: #topToBottom`)



Picture by Andreas Raab

Sizing in TableLayouts

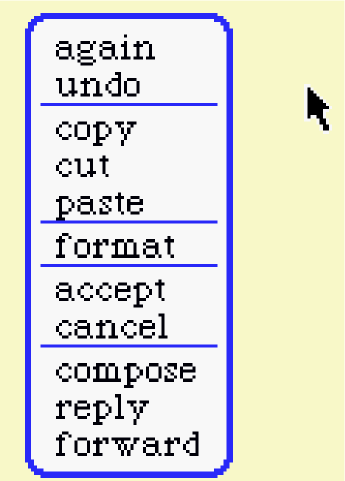
- Sizing of objects is controlled by vResizing: and hResizing:
- Most common options:
  - #shrinkWrap—fit tightly around submorphs
  - #spaceFill—take up as much space as owner allows
  - #rigid—no automatic resizing
- Can also control listCentering: (#topLeft, #center, etc.)
- Lots of other options, e.g., spaceFillWeight which gives one object precedence over others

Using an AlignmentMorph for positioning

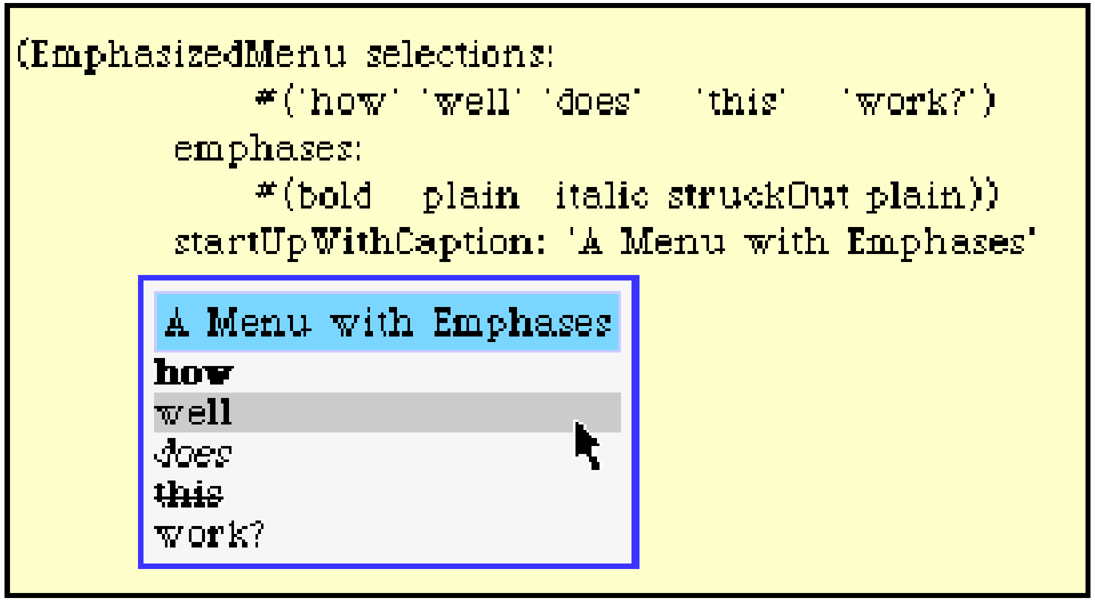
```
openAsMorph2
    | win component filler clock |
    "Create the clock"
    clock := Clock new.
    clock setTime: (Time now printString).
    clock start.
    "Create a window for it"
    win := SystemWindow labelled: 'Clock'.
    win model: self.

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

Menus in Pluggable Interfaces

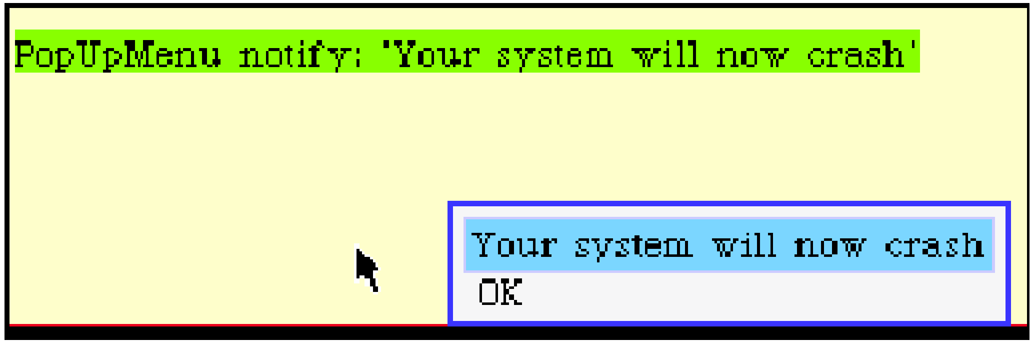
```
CustomMenu "From Celeste"
                    labels:
    'again\undo\copy\cut\paste\format\accept\cancel
compose\reply\forward' withCRs "Turn $\ into CR"
                    lines: #(2 5 6 8)
                    selections: #(again undo
copySelection cut paste format accept
cancel compose reply forward)
"Send startUp to get it to appear.
Selection is returned."
```



Other and Simpler Menus



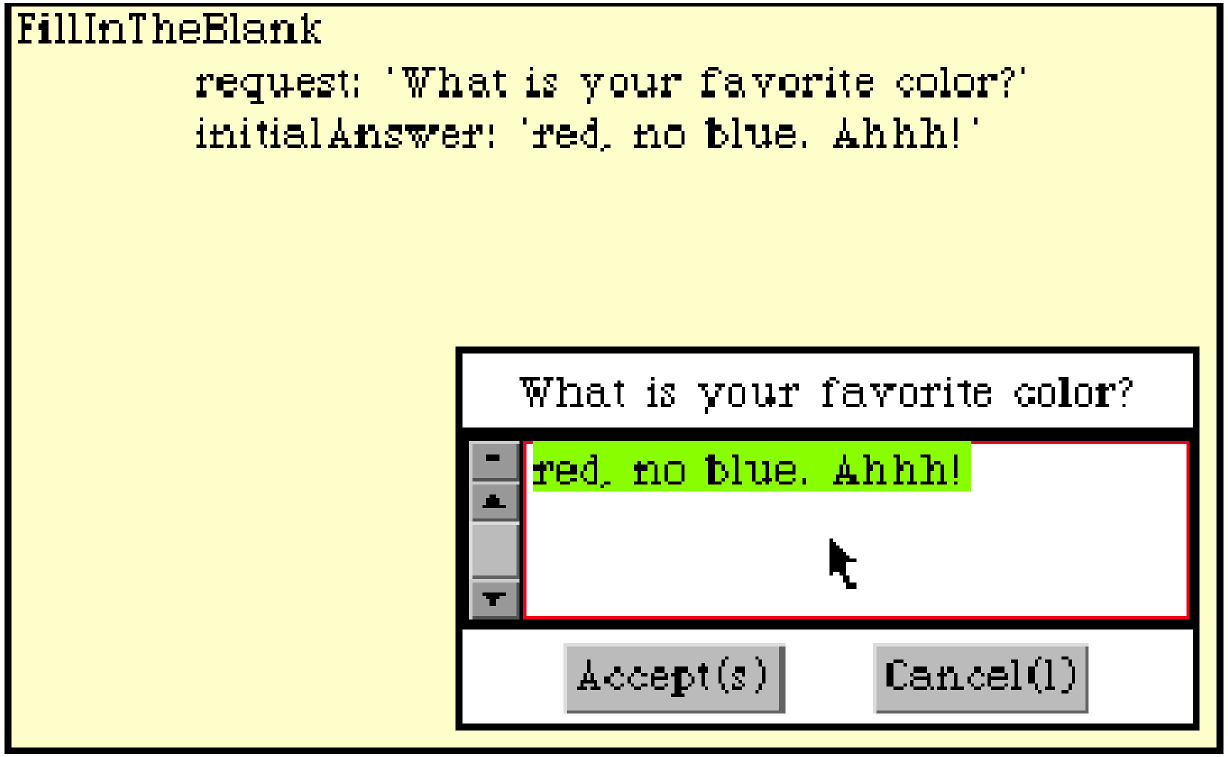
```
(EmphasizedMenu selections:
        #('how' 'well' 'does'  'this'  'work?')
    emphases:
        #(bold  plain  italic struckOut plain))
    startUpWithCaption: 'A Menu with Emphases'
```



```
PopUpMenu notify: 'Your system will now crash'
```

Simple Dialog

- FillInTheBlank



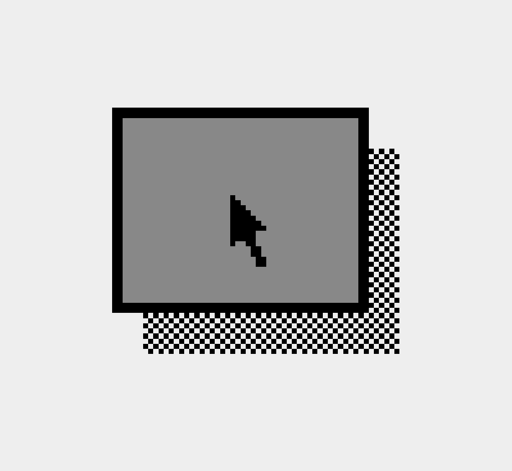
```
FillInTheBlank
        request: 'What is your favorite color?'
        initialAnswer: 'red, no blue. Ahhh!'
```

What is your favorite color?

red, no blue. Ahhh!

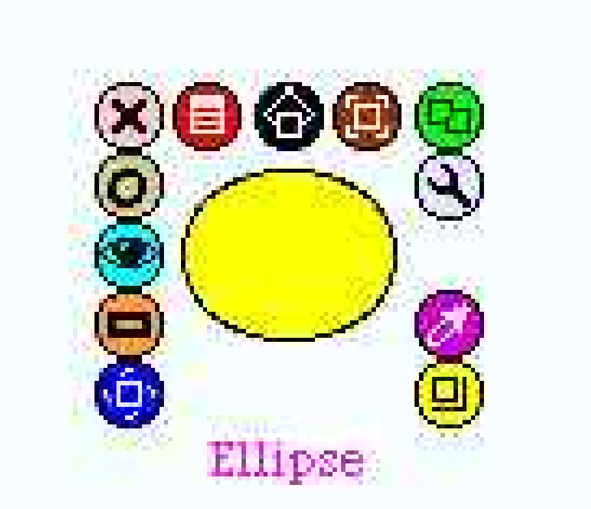
Introduction to Morphic

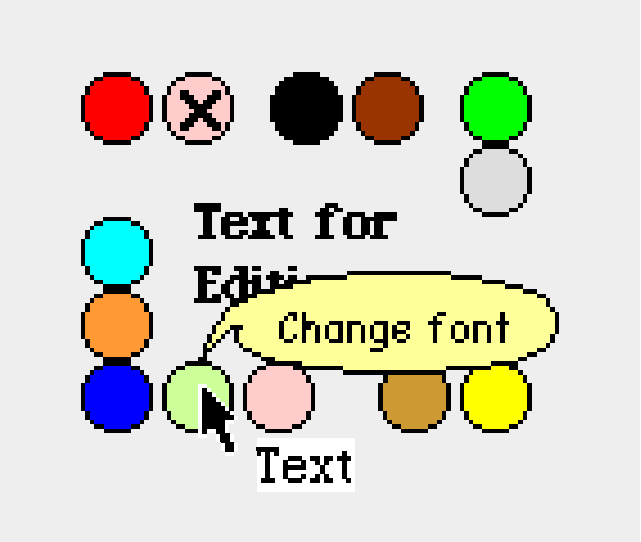
- Any object can be a window
  - All on-screen objects are subclass of *Morph* so common behavior is assured
  - For example, moving things leaves a shadow
  - Morphic objects are:
    - Concrete
    - Uniform
    - Flexible



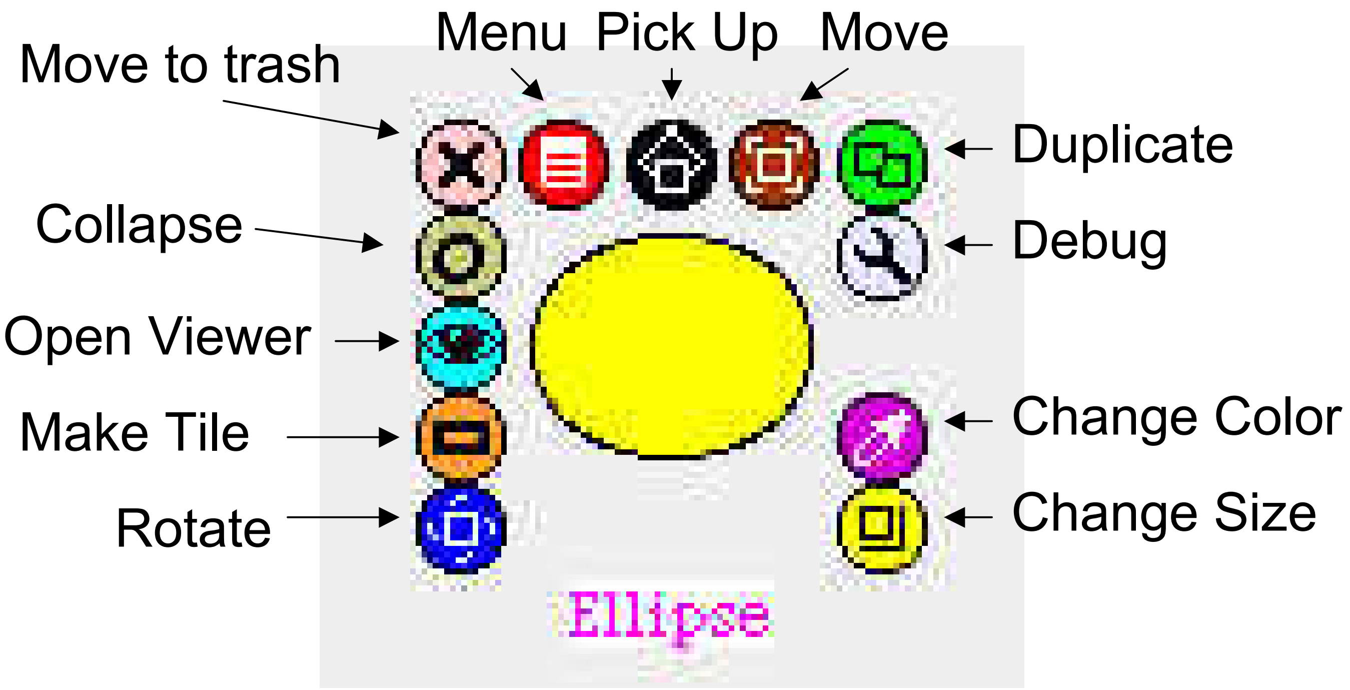
Select Any Morph to Manipulate In Standard Ways

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





Description



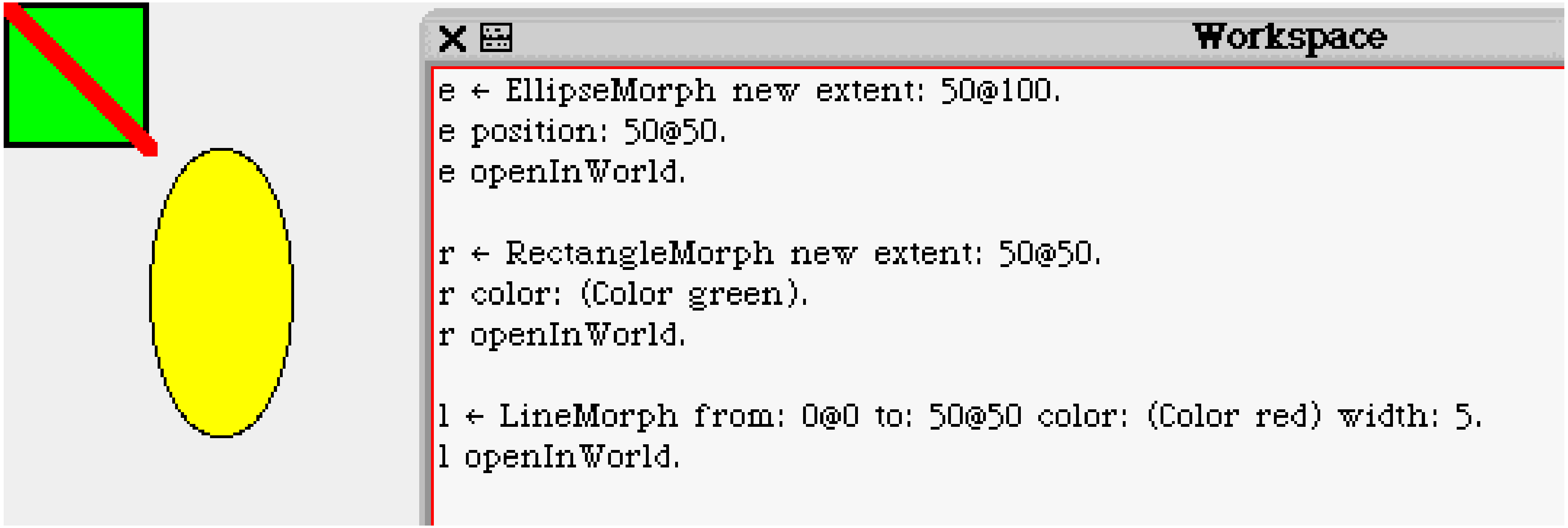
Morphs can be composed

- Morphic-select something.
  - Repeat to walk through owners and submorphs
  - Shift-Morphic-Select to go inner-to-outer
- Find submorphs also by inspecting or exploring
  - Available through debug options in Red-Halo Menu

Creating Morphs

- Use the Objects menu from World menu
- Grab them from the flaps (Supplies, Widgets)
- Use the New Morph menu
- Send “new openInWorld” to Morph class

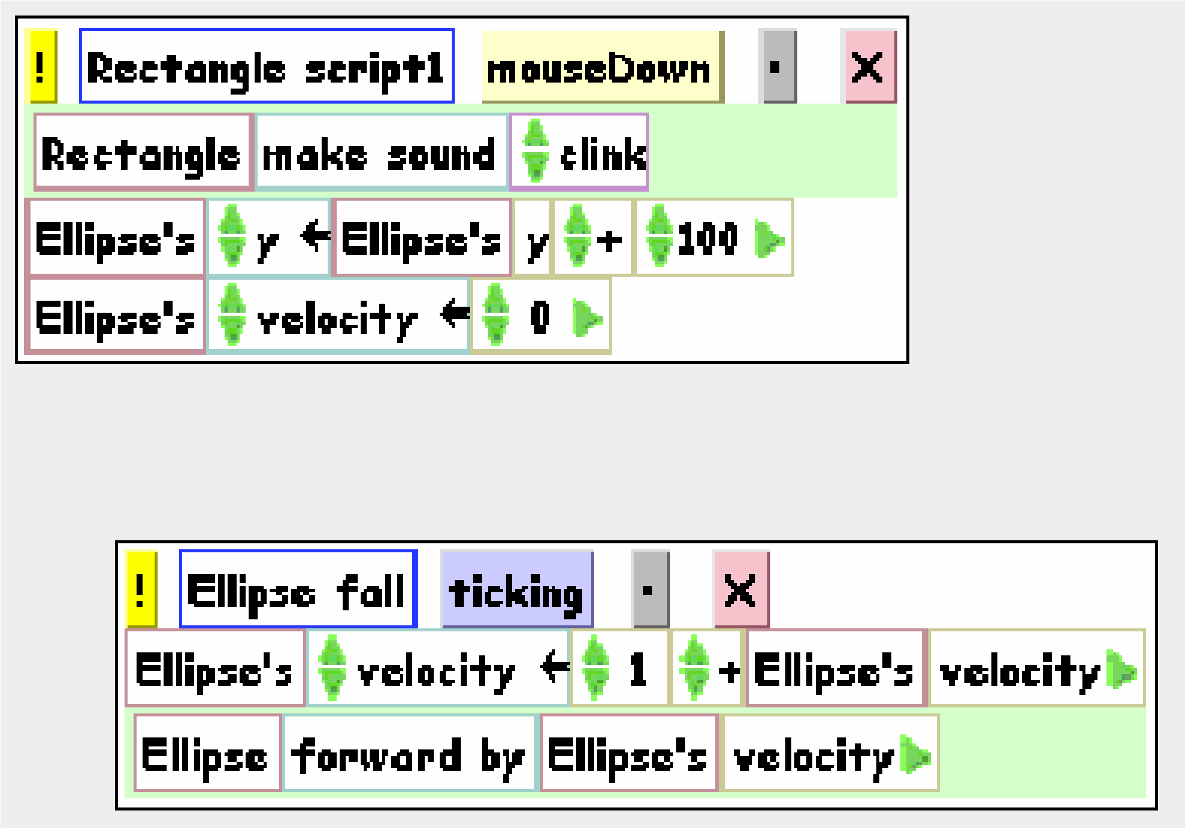
Some Example Morphs



```
e ← EllipseMorph new extent: 50@100.
e position: 50@50.
e openInWorld.

r ← RectangleMorph new extent: 50@50.
r color: (Color green).
r openInWorld.

l ← LineMorph from: 0@0 to: 50@50 color: (Color red) width: 5.
l openInWorld.
```

Programming Morphic from the Viewer Framework

- Develop the Falling Object Simulation to these:



```
Rectangle script1   mouseDown
Rectangle make sound clink
Ellipse's y ← Ellipse's y + 100
Ellipse's velocity ← 0
```

```
Ellipse fall   ticking
Ellipse's velocity ← 1 + Ellipse's velocity
Ellipse forward by Ellipse's velocity
```

Class-based vs. Prototype-based Inheritance

- Class-based
  - You create a class that defines data structure and behavior
  - Instances are made of that class
- Prototype-based
  - You create an instance and give it data and behavior
  - You can create instances off the instance
  - Some things get inherited, others may not

Prototype-based Inheritance

- Strengths
  - Easier to get started and build something
  - Works well for rapid prototyping
- Weaknesses
  - Harder to maintain

Programming in Morphic

- Key instance variables and properties that Morphs share
  - Both Morph and MorphExtension
- Handling Morphic Events
- Animating Morphs
- Providing menus to Morphs
- Structure of Morphic
- Programming a Morphic Falling Object

Overall Structure of Morphic

- All morphs in a project live in the World (instance of PasteUpMorph)
  - Worlds have a Canvas that handles display of all morphs
- The World contains one or more Hands (cursor)
  - Hands interpret user events and pass them on to morphs (e.g., event loop)
  - Hands also deal with generation of menus as needed
- The World sends *step* messages at regular intervals to morphs to allow updating over time

Instance Variables and Properties

- bounds: Rectangle defining shape of the morph. Change it resize or move.
- owner: Containing morph.
- submorphs: Contained morphs (addMorph: to change)
- color
- name
  - Well, not actually...

The Morph annex: extension

- MorphExtension knows
  - balloonText, balloonTextSelector
  - visible
  - locked: Locked morphs can't be selected (*lock* and *unlock*)
  - sticky: Sticky morphs can't be moved (*toggleStickiness*)
  - otherProperties: A Dictionary to store more

Morphic Events

- When the Hand Morph detects an event:
  - Create a MorphicEvent
    - Can't poll it, but can ask it *redButtonPressed*
  - Appropriate MorphicEvent is passed to object under the Hand by sending the corresponding message

Handling Morphic Events

- To handle mouse down:
  - Have a method *handlesMouseDown* which inputs MorphicEvent and returns *true*
  - Have a method named *mouseDown:* which takes a MorphicEvent and processes it
- MouseUp/MouseOver
  - *handlesMouseUp:/mouseUp:*
  - *handlesMouseOver:/mouseOver:*

More Event Handling

- MouseEnter/MouseLeave
  - *handlesMouseOver:* returns *true*
  - *mouseEnter:/mouseLeave:*
- MouseMove (within the morph)
  - *handlesMouseDown:*
  - *mouseMove:*
- Keystrokes
  - Return *true* for *hasFocus*
  - Accept events in *keyStroke:*
  - *keyboardFocusChange:* will tell you of change

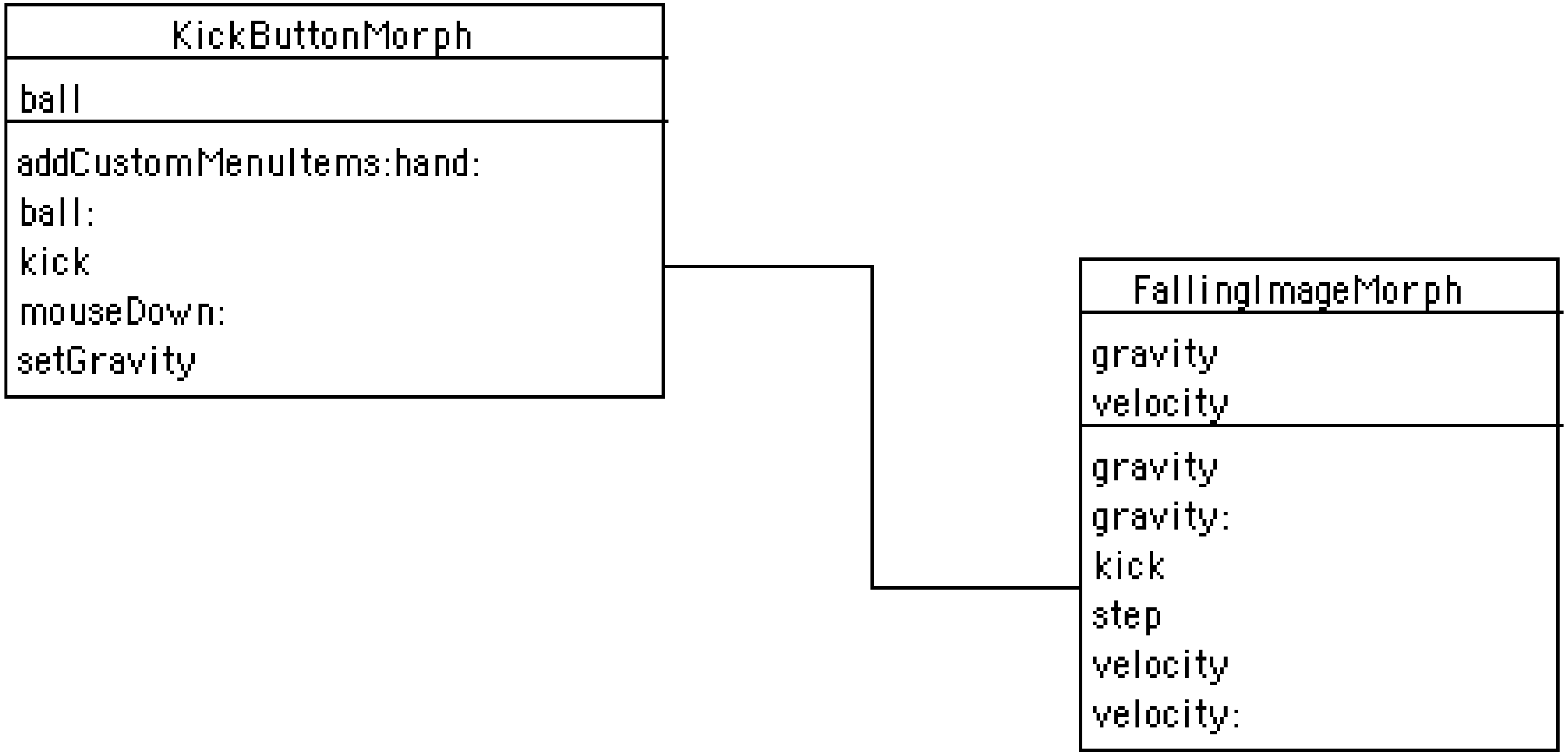
Animation

- Morphic interfaces are designed to animate
- *step* is sent to all morphs
- *stepTime* is interval for *step* (number in milliseconds)

Custom Menus in Morphic

- addCustomMenuItems: aCustomMenu hand: aHandMorph
  - Called when red-halo (menu halo) or control-click menu is requested
  - You can add with *add:action:* or others
  - First, do *super addCustomMenuItems: aCustomMenu hand: aHandMorph*

Falling Object in Morphic



KickButtonMorph

ball

addCustomMenuItems:hand:
ball:
kick
mouseDown:
setGravity

FallingImageMorph

gravity
velocity

gravity
gravity:
kick
step
velocity
velocity:

Subclassing

- (Could use SimpleButtonMorph, but too easy)

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

Making the Falling Object fall

```
step
    velocity := velocity + gravity. "Increase velocity
    by gravitational constant"
    self bounds: (self bounds translateBy:
    (0@(velocity))).
stepTime
    "Amount of time in milliseconds between steps"
    ^1000
```

Kicking the object

```
kick
   velocity := 0. "Set velocity to zero"
   self bounds: (self bounds translateBy:
   (0@(100 negated))).
```

Initializing the Falling Object

```
initialize
   super initialize. "Do normal image."
   velocity := 0. "Start out not falling."
   gravity := 1. "Acceleration due to gravity."
```

Implementing the Kicker

```
handlesMouseDown: evt
   "Yes, handle mouse down"
   ^true
mouseDown: evt
   self kick.
kick
   ball kick.
```

Initialize to make it Button-ish

```
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."
```

Allowing changing gravity

```
addCustomMenuItems: aCustomMenu hand:
    aHandMorph
    super addCustomMenuItems: aCustomMenu hand:
    aHandMorph. "Do normal stuff"
    aCustomMenu add: 'set gravity' action: #setGravity.
setGravity
    "Set the gravity of the ball"
    | newGravity |
    newGravity := FillInTheBlank request: 'New gravity'
                            initialAnswer: ball gravity printString.
    ball gravity: (newGravity asNumber).
```

Running the Simulation

```
aBall := FallingImageMorph new initialize.
aBall newForm: (Form fromUser).
aKicker := KickButtonMorph new initialize.
aKicker ball: aBall.
aBall openInWorld.
aKicker openInWorld.
```

Morphic vs. MVC

- MVC (world-view, not paradigm)
  - Is faster than Morphic
  - Is less elegant
  - Doesn’t support multimedia like Morphic
- Morphic
  - Is slower
  - Is better looking, more flexible, more powerful
  - Can do multimedia

Can we do MVC (paradigm) in Morphic?

- In terms of changed-update and dependencies, SURE!
- We can’t really do controllers in Morphic
  - They’re built-in to the World
- But most Morphic interfaces either:
  - Combine model and view
  - Or use *step* to poll the model