CS 535 Object-Oriented Programming & Design
Fall Semester, 2011
Doc 19 Some MVC Issues
Nov 29 2011

Model-View-Controller (MVC)

Model

Encapsulates

Domain information
Core data and functionality

Independent of

Specific output representations
Input behavior

View

Display data to the user

Obtains data from the model

Multiple views of the model are possible

Controller

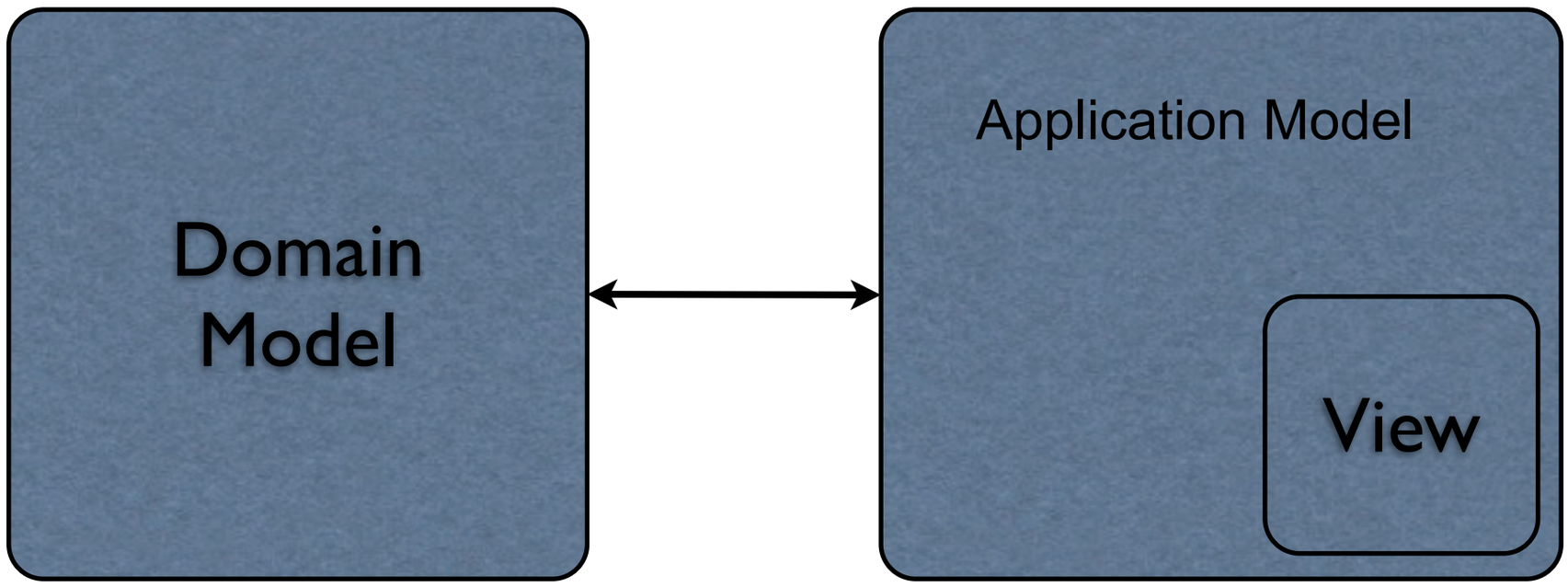
Handles input

- Mouse movements and clicks
- Keyboard events

Each view has it's own controller

Programmers commonly don't see controllers

Smalltalk Uses Application Model



Application Model

- Presentation of domain to user
- GUI + logic to present data from domain

Application Model becomes Controller

Handles interaction between View and Model

Main Points

Application Model is not the Model

The model should not know about the view

Application Model is not the Model

Why does it matter?

The model should not know about the view

Why does it matter?

Small Examples Hide the Issues

Clock App

Model

ButtonExample

View

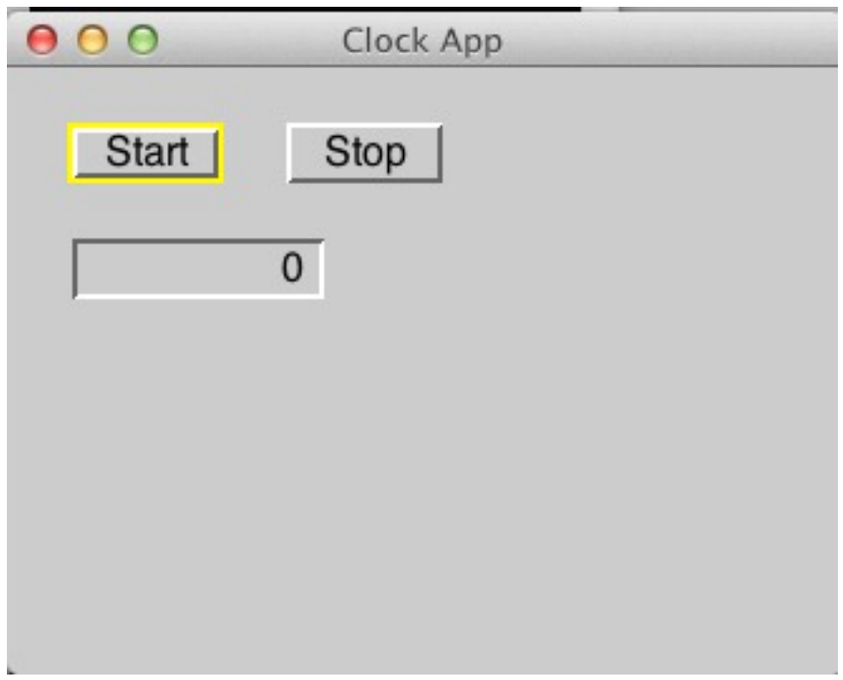
Created dynamically from window spec

Controller

Hidden

Clock App

View



Application Model Logic

```
startTimer
	clock startAfter: 0 seconds
```

```
stopTimer
	clock stop
```

```
timeDisplay
	^timeDisplay isNil
		ifTrue:
			[timeDisplay := 0
asValue]
		ifFalse:
			[timeDisplay]
```

Clock App - Where is the Domain Model?

```
initialize
        time := 0.
        clock := Timer new.
        clock
                period: 1 seconds;
        block:
                        [time := time + 1.
                        timeDisplay value: time]
```

time + clock = Domain Model

But Application Model contains code to make domain model work

Domain logic is in application model

So who cares?

Domain Logic in controller

- Can't reuse domain model - missing logic

Controller becomes more complex

- Does two different things

So create Domain Object - Clock

```
Smalltalk defineClass: #Clock
        superclass: #{Core.Object}
        instanceVariableNames: 'count timer '
```

Class Method

```
period: aDuration
        ^super new setPeriod: aDuration
```

Instance Methods

```
setPeriod: aDuration
        count := 0.
        timer := Timer new.
        timer period: aDuration.
        timer block: [timer := timer + 1]
```

```
start
        timer startAfter: 0 seconds
```

```
stop
        timer stop
```

```
time
        ^count
```

But how does view know clock change?

Clock as Subject

```
Smalltalk defineClass: #Clock
        superclass: #{Core.Object}
        instanceVariableNames: 'count timer '
```

Class Method

```
period: aDuration
        ^super new
          setPeriod: aDuration
```

Instance Methods

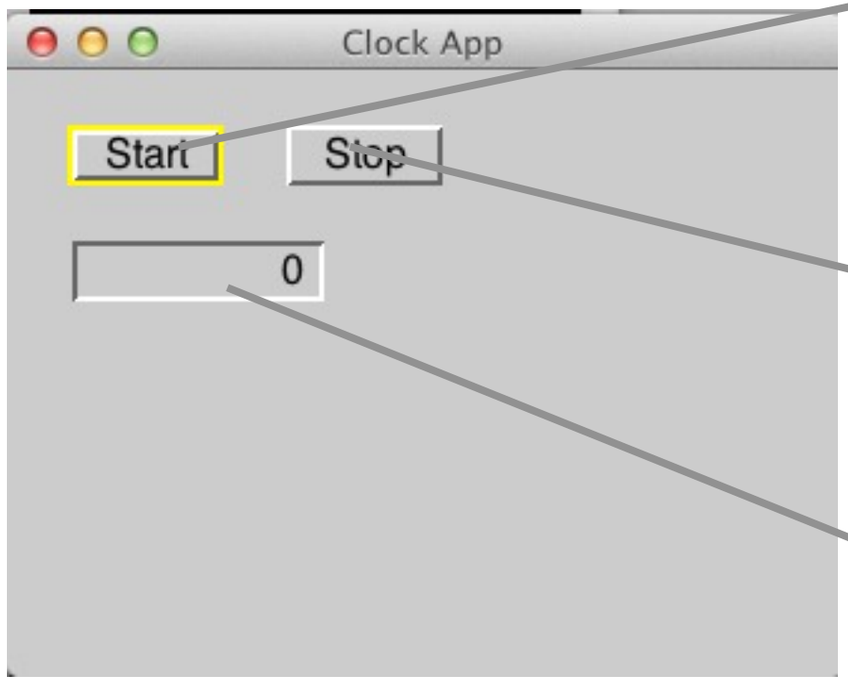
```
setPeriod: aDuration
        count := 0.
        timer := Timer new.
        timer period: aDuration.
        timer block:
                [count := count + 1.
                self changed]
```

```
start
        timer startAfter: 0 seconds
```

```
stop
        timer stop
```

```
time
        ^count
```

Clock App with Clock subject




```
startTimer
        clock start
```

```
stopTimer
        clock stop
```

```
timeDisplay
        ^timeDisplay isNil
                ifTrue:
                        [timeDisplay := 0
asValue]
                ifFalse:
                        [timeDisplay]
```

```
initialize
        clock := Clock period: 1 seconds.
        clock addDependent: self
```

```
update: aSymbol
        timeDisplay value: clock time
```

Small Examples Hide the Issues

Hypothetical Situation

Player has to display what it holds

Rooms has to display what it contains

Trolls display actions

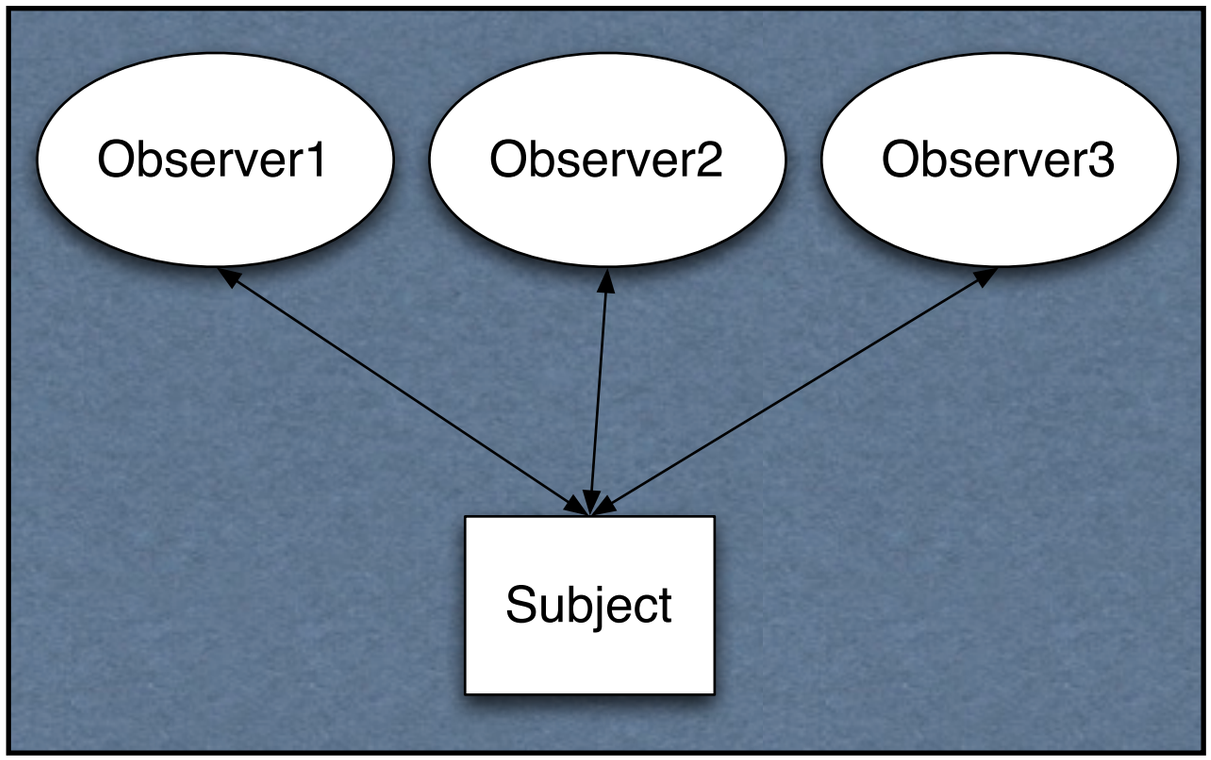
Does Application Model Know about

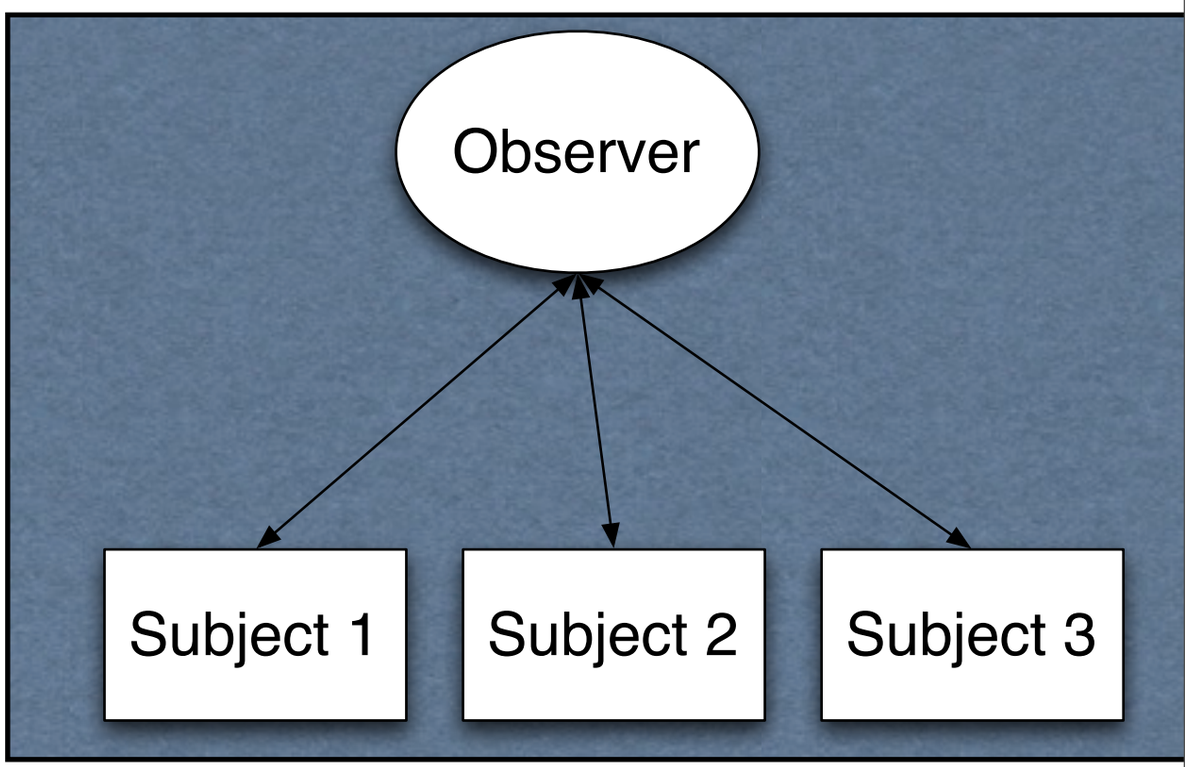
Player
Trolls
Rooms

```
Smalltalk defineClass: #Advententure
        superclass: #{UI.ApplicationModel}
        indexedType: #none
        private: false
        instanceVariableNames: 'player trolls rooms corridors '
        classInstanceVariableNames: ''
        imports: ''
        category: ''
```

What are the issues?

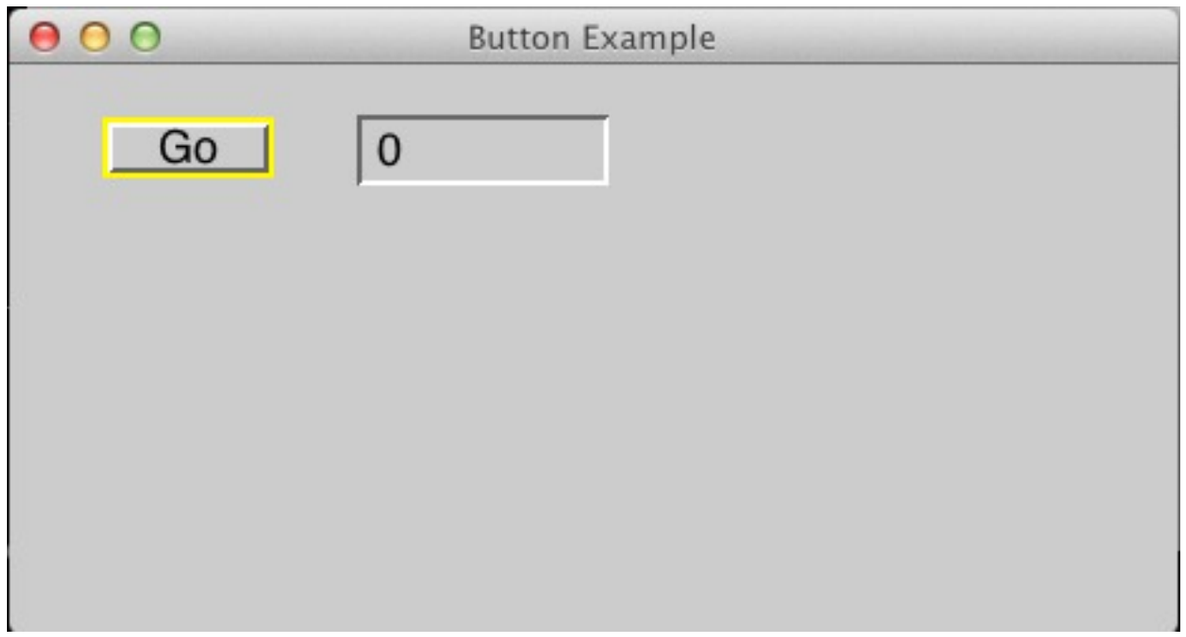
Observer pattern





Subject notifies all observers when it changes

Button Counter Example



```
ButtonExample>>count

        ^count isNil
                ifTrue:
                        [count := 0 asValue]
                ifFalse:
                        [count]
```

```
ButtonExample>>go

        self count value: (self count value  + 1) .
        ^self
```


250V

```
ButtonExample>>countAdapter

	| countAdapter |
	countAdapter := AspectAdaptor subject: self.
	countAdapter
		forAspect: #count;
		subjectSendsUpdates: true.
	^countAdapter
```

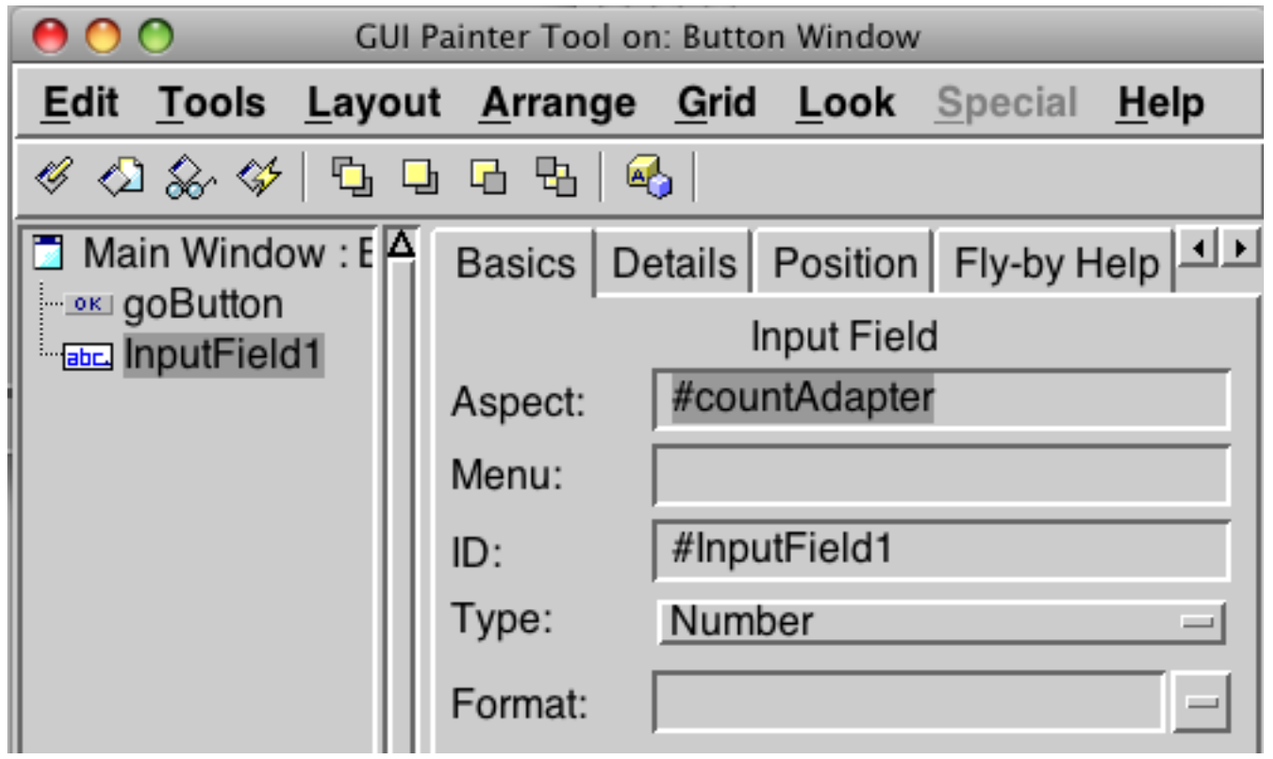
```
ButtonExample>>initialize

	count := 0
```

```
ButtonExample>>count

	^count
```

```
ButtonExample>>go

	count := count + 1.
	self changed: #count.
	Dialog warn: 'Time to go'.
	^self
```

```
ButtonExample>>count: anInteger

	count := anInteger
```


The text widget wants a ValueHolder, we want to have an integer. So we use an adapter that looks like a ValueHolder to the text widget. However when it needs to access the value it uses the count and count: methods.


GUI Painter Tool on: Button Window
Edit Tools Layout Arrange Grid Look Special Help
Main Window : E
OK goButton
abc InputField1
Basics Details Position Fly-by Help
Input Field
Aspect: #countAdapter
Menu:
ID: #InputField1
Type: Number
Format:

ButtonExample

- Simple example
- Designed to show how to use a widget

It handles both view logic and domain logic

```
Smalltalk defineClass: #Counter
	superclass: #{Core.Object}
	instanceVariableNames: 'count '

Counter class>>new
	^super new initialize

Counter>>count
	^count

Counter>>count: anInteger
	count := anInteger

Counter>>increment
	self count: count + 1

Counter>>initialize
	count := 0
```

```
Smalltalk defineClass: #ButtonExample
        superclass: #{UI.ApplicationModel}
        instanceVariableNames: 'count '
```

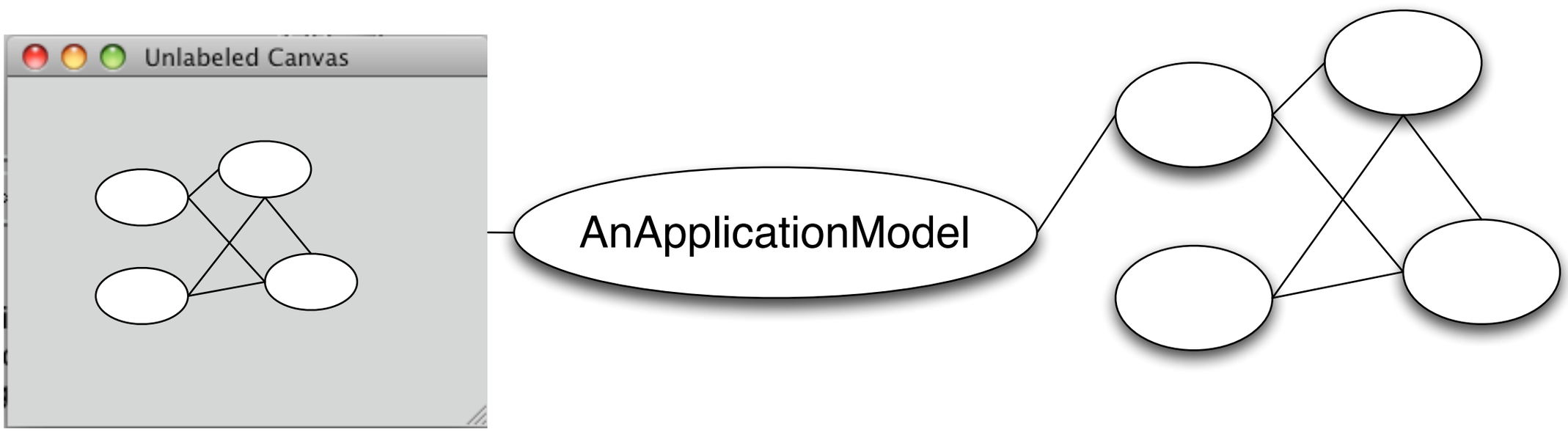
```
initialize
        count := Counter new
```

```
countAdapter
        | countAdapter |
        countAdapter := AspectAdaptor subject: count.
        countAdapter
                forAspect: #count;
                subjectSendsUpdates: true.
^countAdapter
```

```
go
        count increment.
        count changed: #count.
        Dialog warn: 'Time to go'.
        ^self
```

ButtonExample class controls when count changes

ButtonExample can then inform window of changes
Keeps Counter class independent of GUI



What if other objects can change count?

ButtonExample will not be able to inform window of changes

When we have multiple domain objects a message to one may result in changes to one or more objects. If these objects contribute to the display it may not be possible for the application model to notify the view how to update itself correctly.

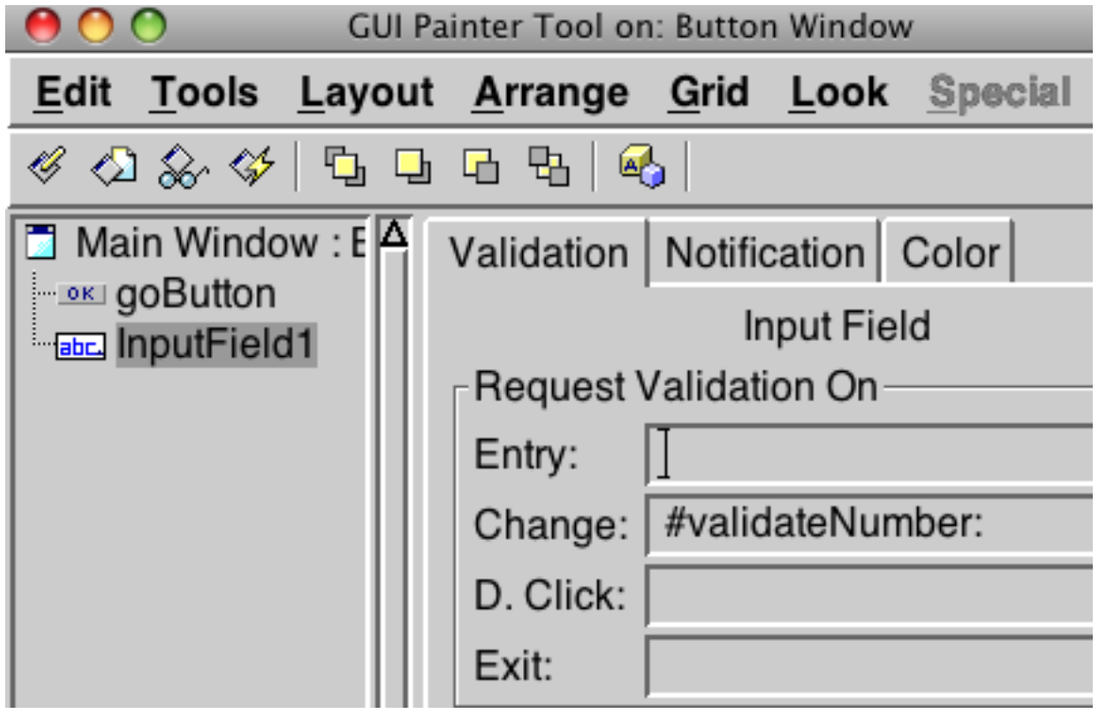
```
Smalltalk defineClass: #Counter
        superclass: #{Core.Object}
        instanceVariableNames: 'count '

Counter class>>new
        ^super new initialize

Counter>>count
        ^count

Counter>>count: anInteger
        count := anInteger

Counter>>increment
        self count: count + 1.
    self changed: #count

Counter>>initialize
        count := 0
```

```
Smalltalk defineClass: #ButtonExample
        superclass: #{UI.ApplicationModel}
        instanceVariableNames: 'count '
```

```
initialize
        count := Counter new
```

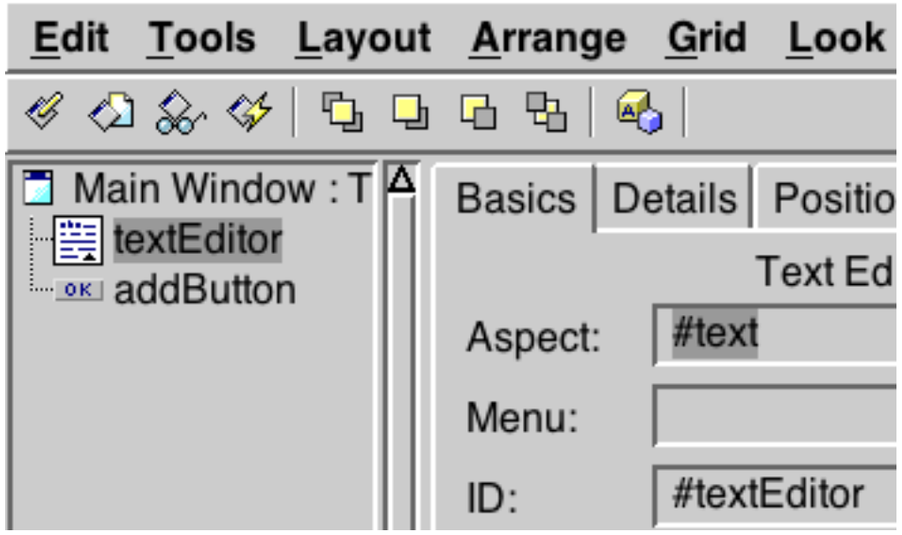
```
countAdapter
        | countAdapter |
        countAdapter := AspectAdaptor subject: count.
        countAdapter
                forAspect: #count;
                subjectSendsUpdates: true.
        ^countAdapter
```

```
go
        count increment.
        Dialog warn: 'Time to go'.
        ^self
```

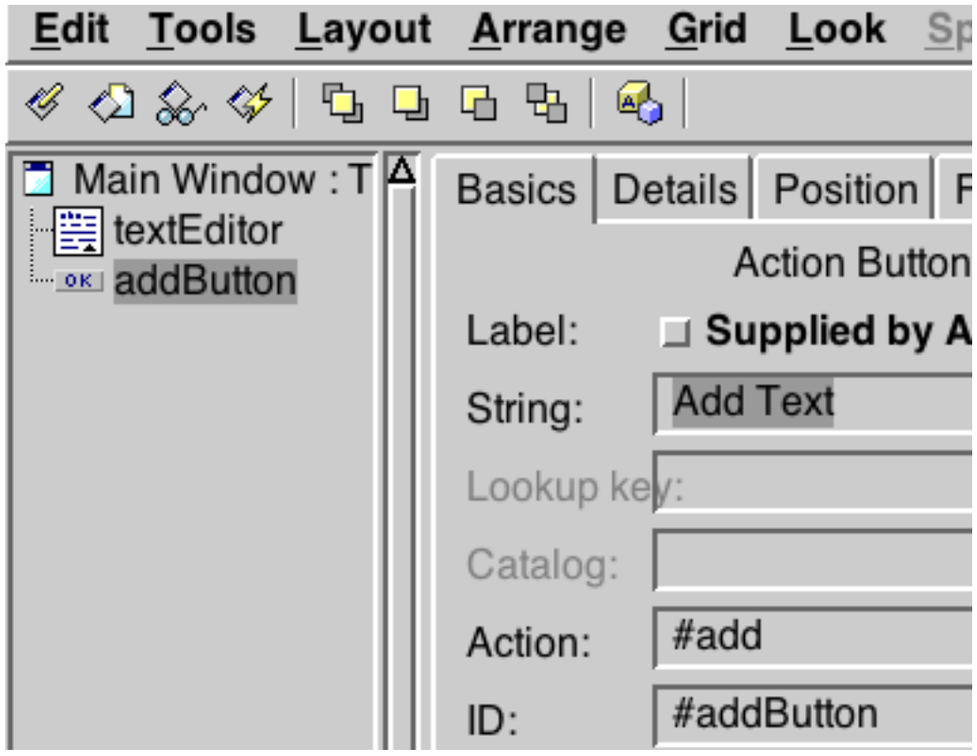



```
ButtonExample>>validateNumber: aController
	| entry |
	entry := aController editValue.
	^entry >= 0
```

Text Editor
widget




```
TextExample>>text
	^text isNil
		ifTrue:
			[text := 'Hi Mom' asValue]
		ifFalse:
			[text]
```




```
TextExample>>add
	self text
	value: self text value , '\Add more text\' withCRs
```