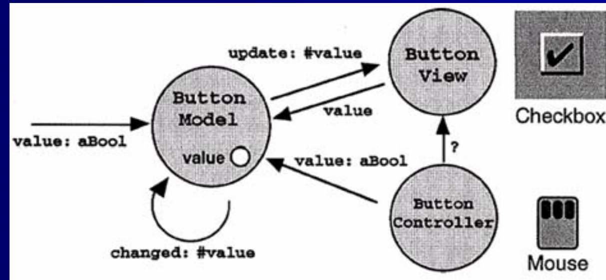
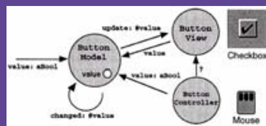
Inside Smalltalk MVC: Patterns for GUI Programming



Chien-Tsun Chen
Department of Computer Science and Information Engineering
National Taipei University of Technology, Taipei 106, Taiwan
ctchen@ctchen.idv.tw

Dec. 14 2004

This talk introduces Smalltalk MVC and presents patterns for programming GUI applications



MVC in Smalltalk: Something old, something new

Understanding the issues of pluggability and adaptors help us better applying MVC

References

[KP88] G. E. Krasner and S. T. Pope, “A Cookbook for Using the Model-View-Controller User Interface Paradigm in Smalltalk-80”, *JOOP*, 1988.

[Woolf94] B. Woolf, Understanding and Using ValueModels, 1994.

[Lwsie95] S. Lewis, *The Art and Science of Smalltalk*, Prentice Hall, 1995.

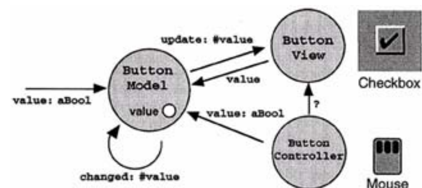
[VW95] VisualWorks Cookbook, Rev 2.0, ParcPlace-Digitalk, 1995.

[GoF95] E. Gamma et. al., Design Patterns: Elements of Reusable Object-Oriented Software, Addison-Wesley, 1995.

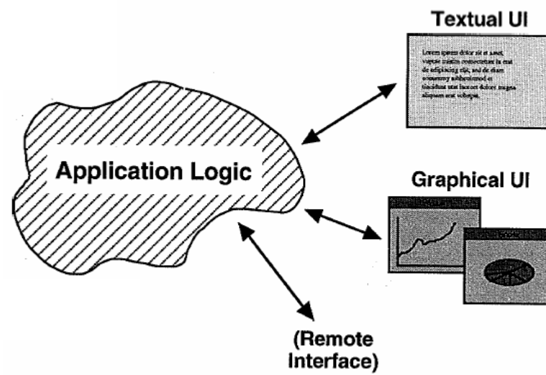
[Ducasse00] S. Ducasse, Object-Oriented Design with Smalltalk — a Pure Object Language and its Environment, 2000.

[Fowlwe04] Martin Fowler, Organizing Presentation Logic, 2004, http://www.martinfowler.com/eaaDev/OrganizingPresentations.html.

MVC in Smalltalk

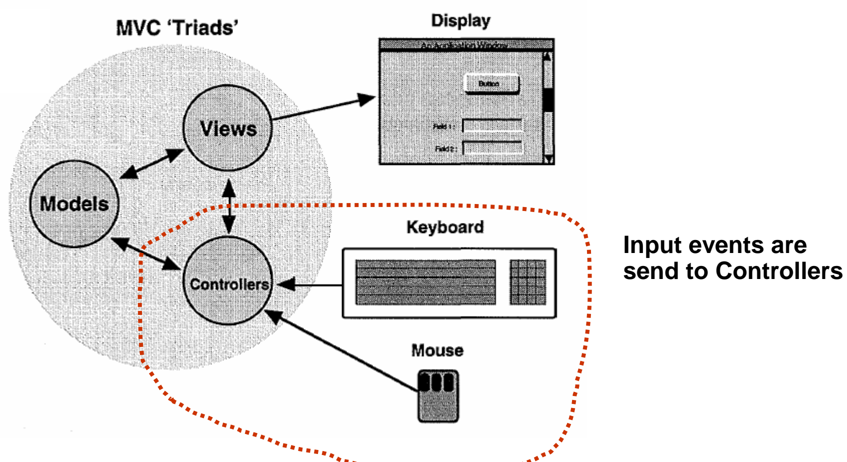


The basic concepts of MVC is separating the application logic from the user interface



An application can have several user interfaces

The views and controllers work together to control the user interface to the models

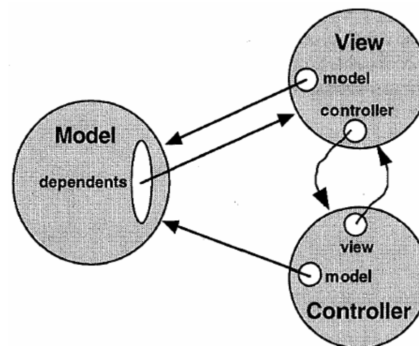


Input events are send to Controllers

Why Smalltalk separates views from controllers?

- To combine views and controllers in different ways to get different look & feel
- To allows views and controllers to inherit from different classes

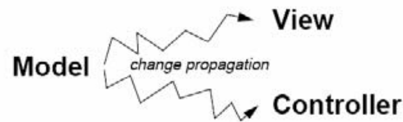
The relationships between model, view and controller

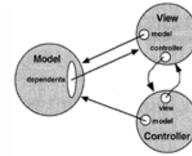


Controller does not “dependent on” Model

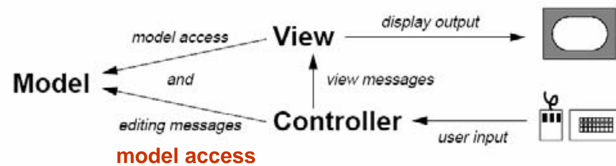
Another way to see the relationships between model, view and controller

Dependencies:



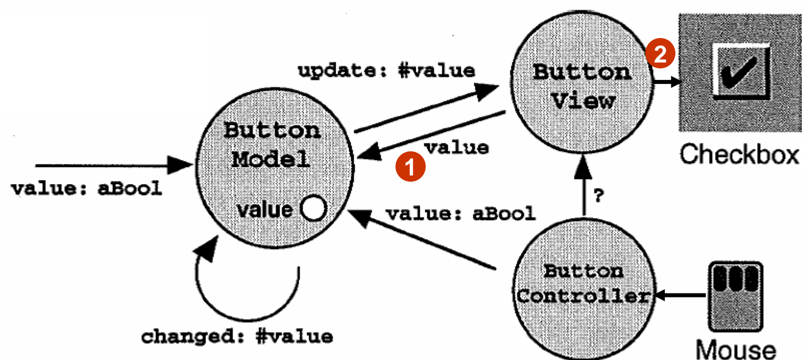


Other Messages:

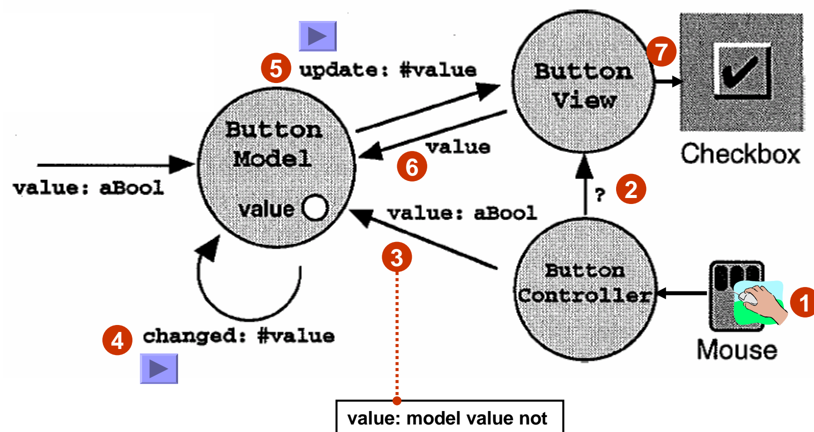


model access

MVC in action, phase 1: The window is opened and ButtonView send the message value to the ButtonModel



MVC in action, phase 2: The user clicks the mouse on the area of the screen and causes the reactions



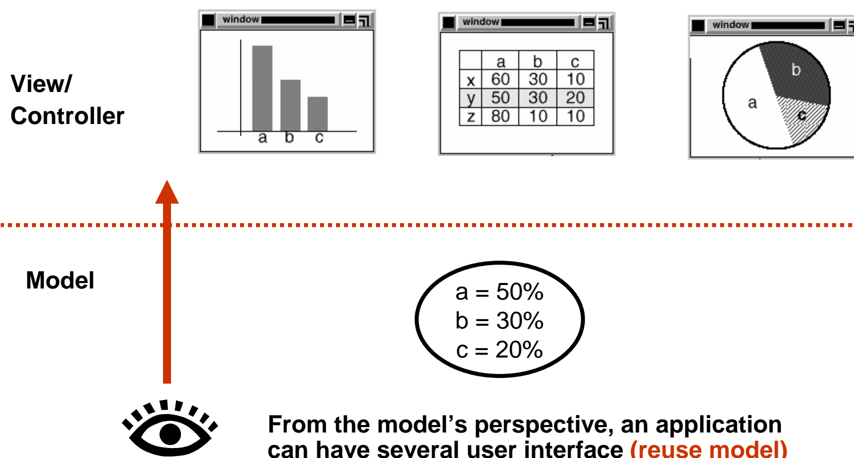
value: model value not

Remember the following import points of classic MVC architecture of Smalltalk [Lwsie95]

1. Neither the view nor the controller hold onto the model's state.
2. The controller doesn't know anything about the *visual layout* of the widget. When it needs that information, it asks the view.
3. When the controller changes the state of the model, it doesn't directly tell the view.
4. The model doesn't know about the view. When its state is changed by the controller, it's only because of the *dependency* mechanism that the view gets to know about the change.
5. When the model tells the view that it's changed, it doesn't tell the view the new state-it only tells it what *aspect* has changed.

Pluggability and Adaptors

The problem comes from the original idea of MVC: separating application logic (model) from UI

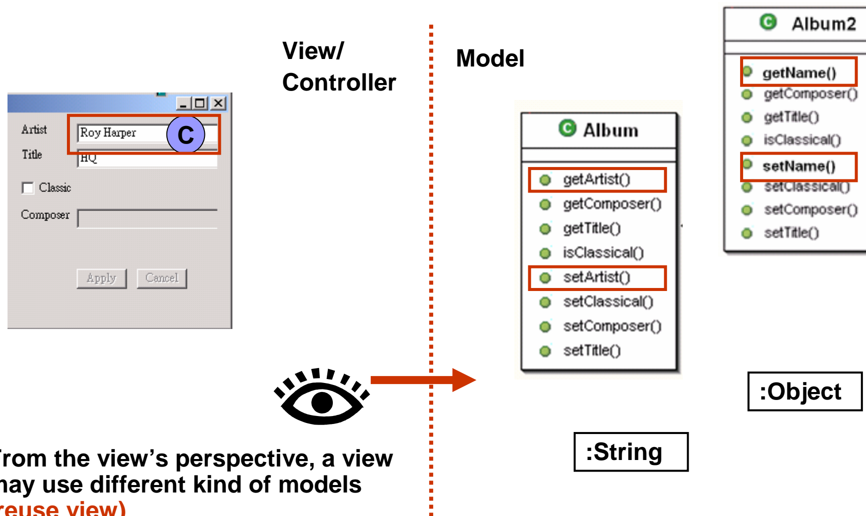


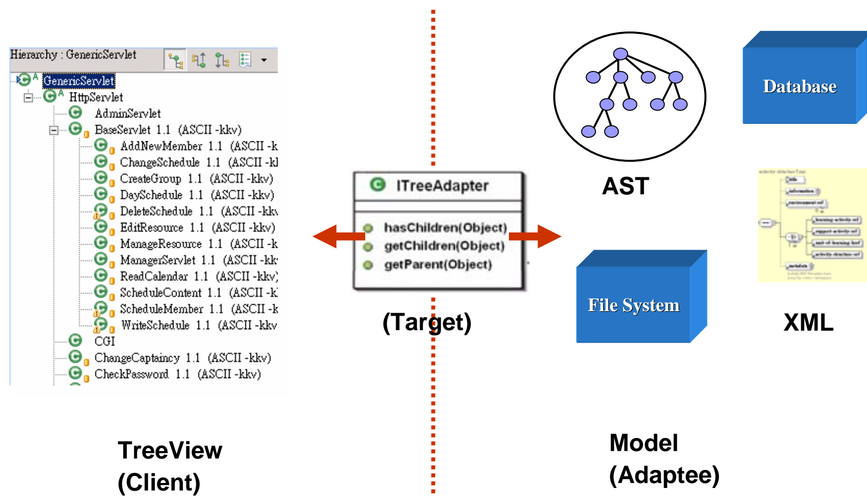
View/
Controller

Model

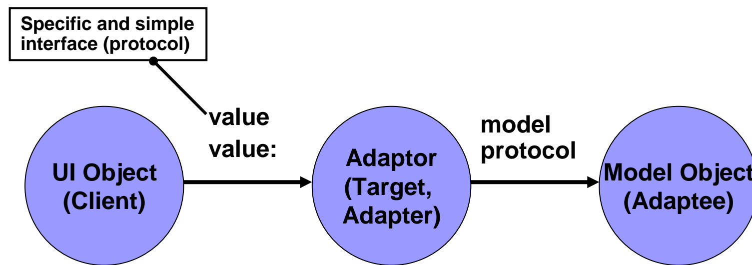
a = 50%
b = 30%
c = 20%

From the model's perspective, an application can have several user interface (reuse model)

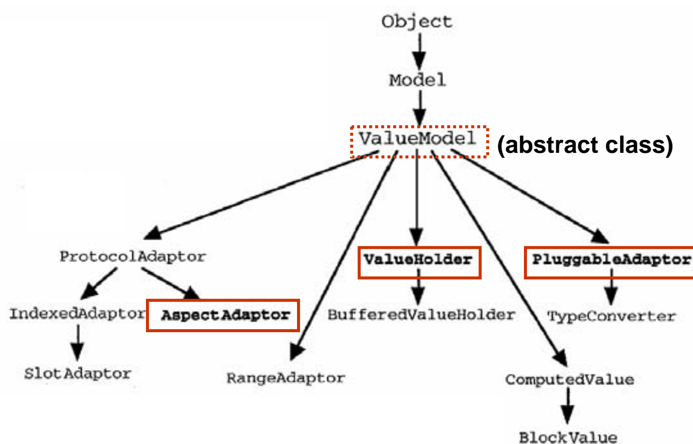

If we look at MVC from the View's perspective…
View/
Controller
Model
Artist
Roy Harper
C
Title
HQ
Classic
Composer
Apply
Cancel
Album
getArtist()
getComposer()
getTitle()
isClassical()
setArtist()
setClassical()
setComposer()
setTitle()
Album2
getName()
getComposer()
getTitle()
isClassical()
setName()
setClassical()
setComposer()
setTitle()
:Object
:String
From the view's perspective, a view
may use different kind of models
(reuse view)


Connecting a widget to a model by (Pluggable)
Adapter pattern (to find a "narrow" interface
for Adaptee)
Hierarchy : GenericServlet
GenericServlet
HttpServlet
AdminServlet
BaseServlet 1.1 (ASCII -kkv)
CGI
ChangeCaptaincy 1.1 (ASCII -kkv)
CheckPassword 1.1 (ASCII -kkv)
ITreeAdapter
hasChildren(Object)
getChildren(Object)
getParent(Object)
(Target)
AST
Database
File System
XML
TreeView
(Client)
Model
(Adaptee)

Adaptors connect UI objects (widgets) which speak only value/value: to model objects with their own protocol

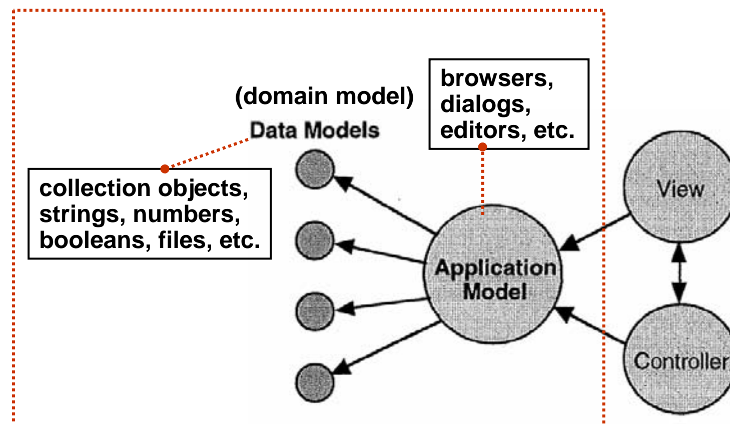


ValueModel solves the problem of change notification, sharing, and adaptation



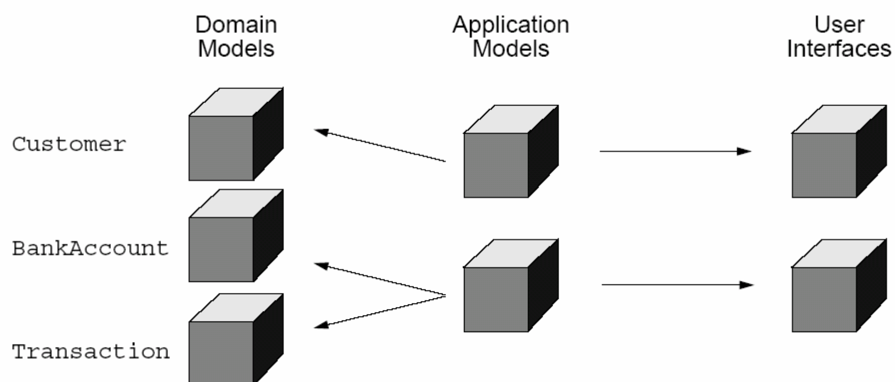
A portion of the class hierarchy showing subclasses of ValueModel

VisualWorks extends MVC by splitting the model into application model and data model

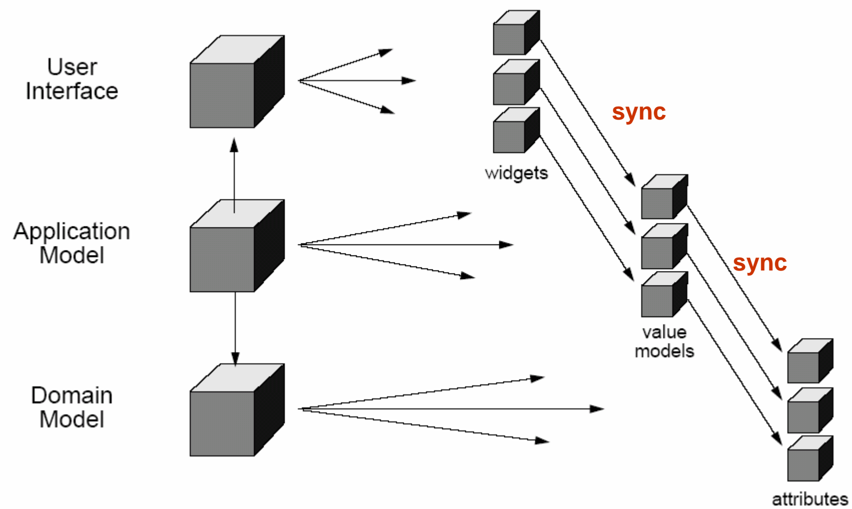


Dual role of model objects: They act as a store for the application's data, and they act upon that data in application specific ways

An Application Model is a model that is responsible for creating and managing a runtime UI, usually consisting of a single window. It manages only application information, leaving the domain information to its aspect models

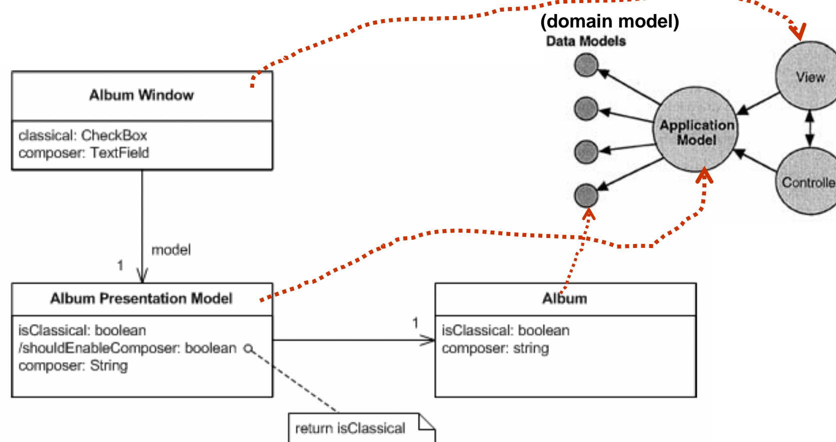


The fine-grained structure of an Application



Review of the Presentation Model Pattern (1/2)

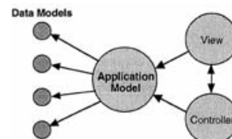
Represent the state and behavior of the presentation independently of the GUI controls used in the interface



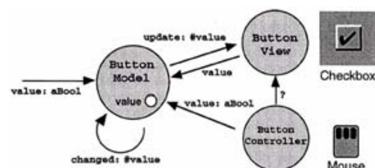
Review of the Presentation Model Pattern(2/2)

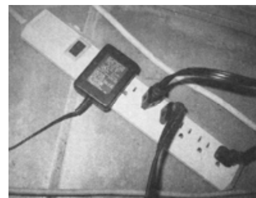
GUIs consist of widgets that contain the state of the GUI screen. Leaving the state of the GUI in widgets makes it harder to get at this state, since that involves manipulating widget APIs, and also encourages putting presentation behavior in the view class.

Presentation Model pulls the state and behavior of the view out into a model class that is part of the presentation. The Presentation Model coordinates with the domain layer and provides an interface to the view that minimizes decision making in the view. *The view either stores all its state in the Presentation Model or synchronizes its state with Presentation Model frequently*.



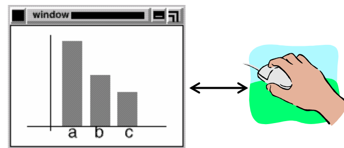
Conclusion: Remember the two pictures

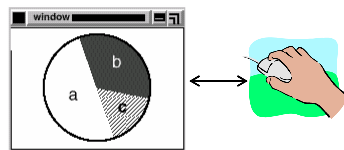




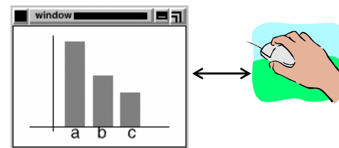
Question?

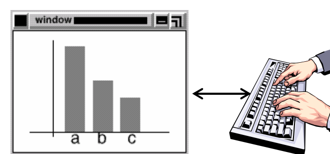
Combine views & controllers in different ways to get different Look & Feel





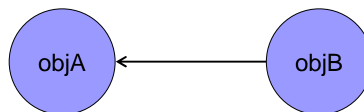
Using different views to get a different "look"





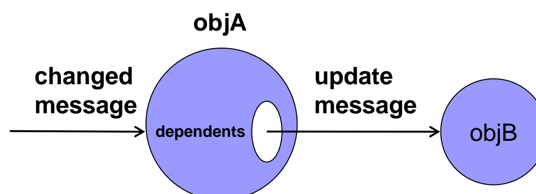
Using different controllers to get a different "feel"

Smalltalk "dependency mechanism": I want to know whenever your values have been changed



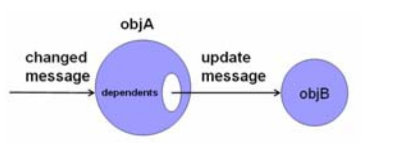
objB is dependent on objA

Static structure



Dynamic behavior

The changed message and the update message of the "dependency mechanism"




```
size: aNumber
  size := aNumber.
  self changed: #size.
```

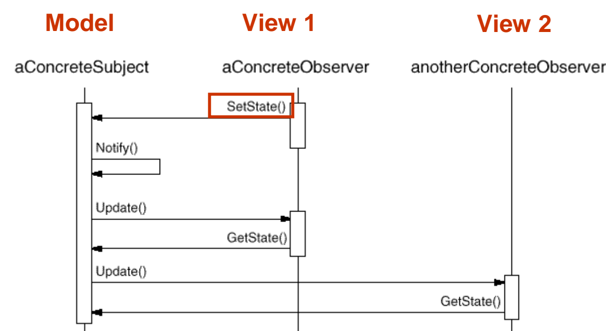
objA changed: anAspect with: aParm (two parameters)
objA changed: anAspect (one parameter)
objA changed. (no parameters)

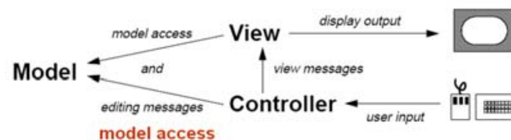
dependent update: anAspect with: aParm from: anObj. (three parameters)
dependent update: anAspect with: aParm. (two parameters)
dependent update: anAspect. (one parameter)

```
update: anAspect
  anAspect = #color ifTrue: [self redraw].
  anAspect = #size ifTrue: [self redraw].
```

Comparing the Observer sequence diagram with the MVC pattern (1/3)

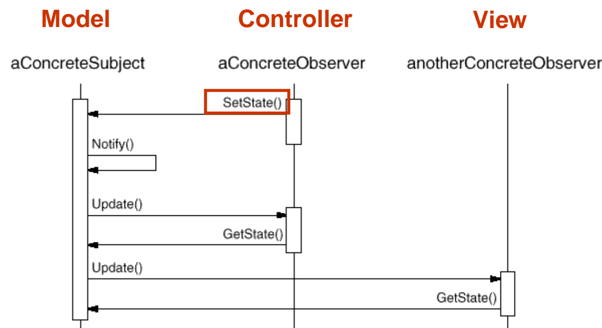
Model | View 1 | View 2

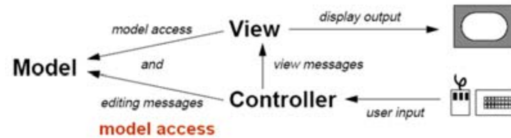




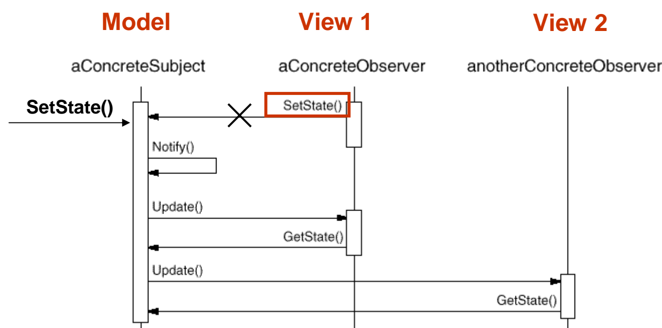
model access

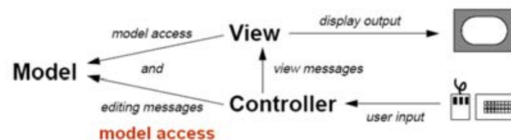
Comparing the Observer sequence diagram with the MVC pattern (2/3)



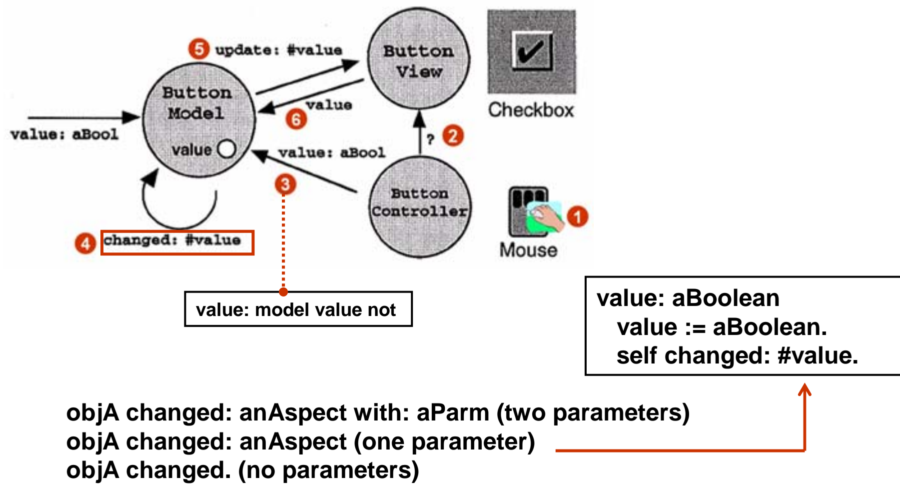


Comparing the Observer sequence diagram with the MVC pattern (3/3)





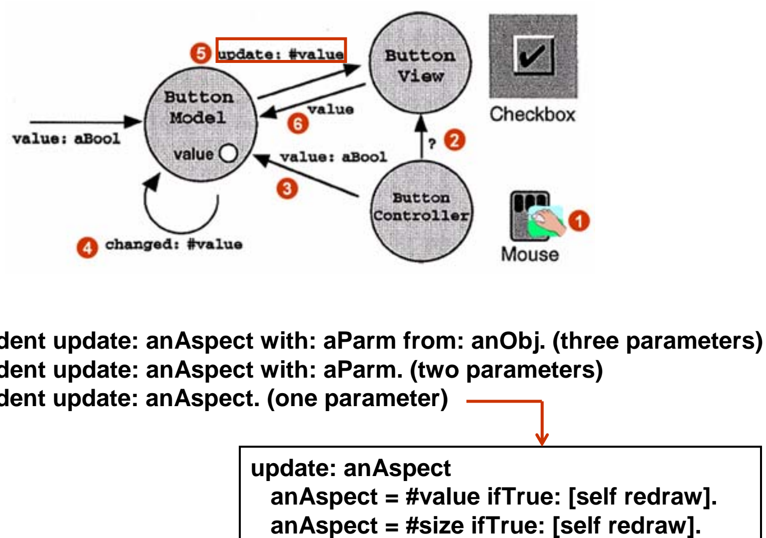
The MVC uses the one parameter version of changed




```
value: aBoolean
  value := aBoolean.
  self changed: #value.
```

objA changed: anAspect with: aParm (two parameters)
objA changed: anAspect (one parameter)
objA changed. (no parameters)

The MVC uses the one parameter version of update



dependent update: anAspect with: aParm from: anObj. (three parameters)
dependent update: anAspect with: aParm. (two parameters)
dependent update: anAspect. (one parameter)

```
update: anAspect
  anAspect = #value ifTrue: [self redraw].
  anAspect = #size ifTrue: [self redraw].
```

Why split the model into two pieces?

Splitting the model in this way removes application specific processing from the data model, making it much more reusable. It also provides something, of a justification for putting some user-interface functionality in the application model. After all, in some cases an application is nothing more than a particular set of operations with a particular user-interface. Consequently, it doesn't matter *too much* if the application model knows some things about the user-interface. In this case the application *is* the interface [1, p. 100].