Lecture 8

Blackboard
MVC
Reflection

Blackboard Pattern

- Operative Metaphor:
  Patient Chart in an ICU
- Operative Image:
  MIT Math Session in the movie *Good Will Hunting*
- (aka Repository Systems)

Penny Nii Quote

- "The blackboard approach provides freedom from message-passing constraints. The message-passing paradigm, although modular, requires a recipient of the message as well as a sender. Often, the recipient is not known, or the recipient might have been deleted. In the blackboard approach, the "message" is placed on the blackboard, and the developer of the module is freed from worrying about other modules"

--Nii, *Blackboard Architectures and Applications*

Blackboard

- The Blackboard pattern uses opportunistic reasoning in solving problems for which no deterministic solution strategies are known before hand, or for problems too vast for a complete exhaustive search
- The Blackboard pattern provides a design in which several specialized subsystems utilize their partial knowledge bases and strategies to build an approximated solution to the original problem.
- Blackboard problems are often solved with solutions involving highly recursive languages such as Lisp, or backward-chaining engines such as Prolog
  - cf. Erman, et. al. 1980
  - cf. Nii, AI Magazine, 7(2) (1986), 38-53

Conceptual Entry Points

- *Nondeterministic* problem space
- Strategy of *approximation*
- Strategy of *experimentation*
- Strategy of *cooperation*
- *Opportunistic* solution strategy, along with forward/backward chaining engines
- Implied *intelligence* (learning, etc.)
- Use of *heuristics* (rule-based strategies)

Nondeterministic Problems

- Problems of this sort tend to be represented in domains which exhibit no predetermined strategies, where:
  - a complete search of the solution space is either not feasible or not practical
    - for example, an analysis of 10-word phrases using a vocabulary of 10,000 words would yield permutations of 1000^{10}
  - the data itself is suspect, or known to be partially in error
  - uncertain approximate solutions are acceptable, where certainty may be unlikely
  - at times radically different algorithms may be leveraged that span the same (competing) or different (coordinating) fields of expertise

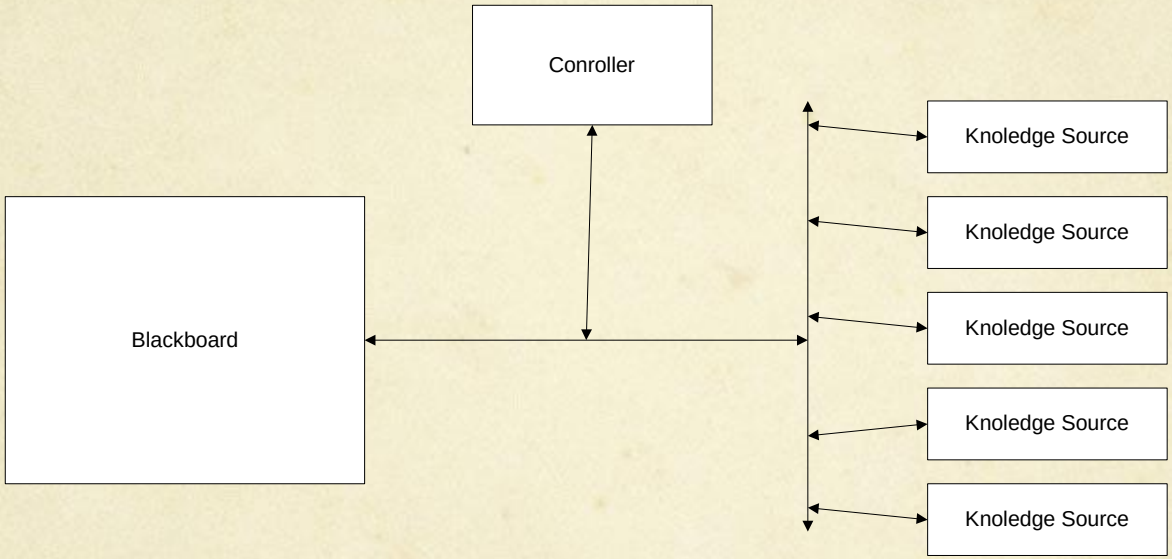
Pattern Space

- The Blackboard pattern is used to provide a modular solution approach to problems that do not have deterministic solutions
- Examples of non-deterministic problem spaces include:
  - speech/sound recognition
  - natural language processing
  - image recognition
  - real time signal processing
  - robotic vision
  - Hubble Space Telescope (OPUS Pipeline)

The Original Blackboard: Hearsay II (1977) Speech Recognition

- Rejected a linear, data-driven analysis model in favor of a more complex, dynamic system
- The blackboard, a global database of hypotheses, comprising the results of all inferences and predictions performed
- Hierarchic layers of hypotheses, linked by “supports” and “explains” relationships.
- Knowledge sources, or action modules, which dynamically create and modify hypotheses in the blackboard. Knowledge sources are independent domain-specific entities that embeds some particular “talent” or knowledge

The Components



The Process

- As analysis unfolds, each new path in the solution map can generate new alternative hypotheses, this is the essence of nondeterministic solutions
- These hypotheses are stored in the Blackboard
- In many cases, the solution will be described as “optimal”, or “partial”, or “limited”.
- Blackboard systems use various algorithms for deriving potential solution sets

Blackboard Component

- Provides a central data store for all hypotheses created during analysis
- Stores current state of the solution space
- Stores all control data
- Delivers requested data to the operating knowledge sources
- Since the Blackboard is essentially a shared memory repository, synchronization techniques may be required for multiple writers and readers (multiple knowledge sources)

Knowledge Source Component

- Knowledge Sources are separate and independent systems that individually solve parts of the problem in piecemeal fashion
- Each knowledge source has a certain area of "expertise", or at least "focus". In this they can be seen as "specialists".
- Knowledge sources obtain data from the blackboard, and return hypotheses developed from the data

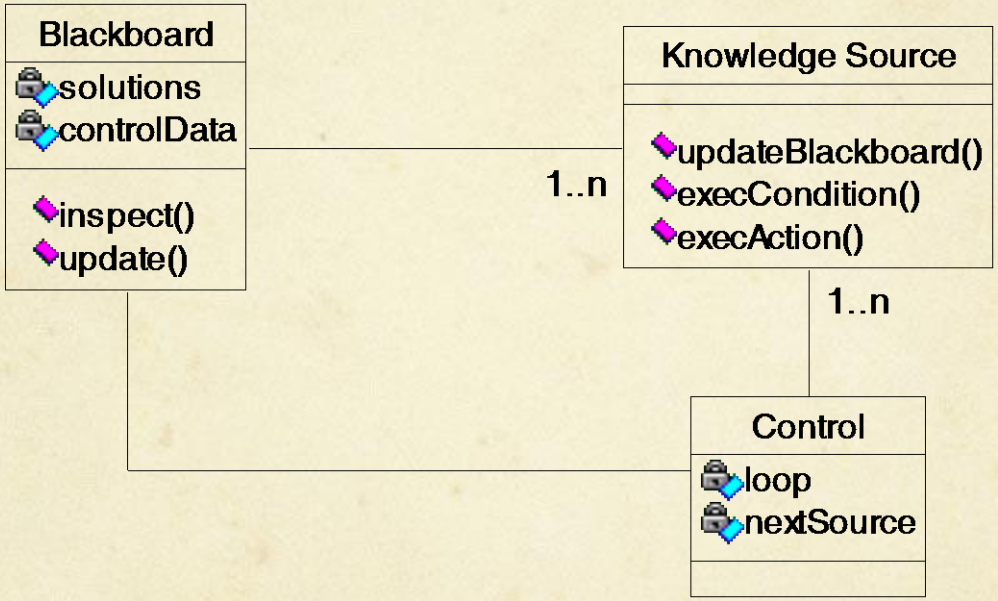
Knowledge Source Component

- Each hypothesis is made up of things like:
  - an abstraction level: the distance from the input
  - an estimated confidence factor
  - a time interval covered by the hypothesis
- Knowledge sources are reactive in the sense that they react to changes in the problem domain as rendered by the blackboard

Control Component

- The controller runs continually monitoring the state of the blackboard, and will end processing whenever an “acceptable” solution is found
- The controller uses heuristics to develop alternate strategies according to the current state of the blackboard, and schedules knowledge sources based on the current strategy
- The controller activates each knowledge source when its scheduled time has arrived
- The controller can query a knowledge source to determine its estimated probability of success or its estimated time of execution, and can update its strategy accordingly, often according to a user-delivered desired confidence level

Blackboard UML



Heresay II Details

- Knowledge sources included tasks such as intelligently segmenting the raw signal into manageable chunks, identifying phonemes, generating word candidates, hypothesizing syntactic segments, and proposing semantic interpretations
- The control component comprised both a blackboard monitor (an event broadcaster when the blackboard contents changed) and a KS scheduler (priority-based)

Key Points

- Often, a complete search of the solution space is not feasible. In this case, algorithms must be developed that filter "significant" information from "noise"
- A data-directed (but not driven) mechanism of coordinating or non-coordinating subsystems uses opportunistic problem solving to provide some solution
- Such opportunistic analysis with multiple algorithms of various talents and biases may produce heuristics that can be used to refine searches
- Blackboards can be used in real-time systems, such as robotics, where different modules (vision, etc.) constantly produce new data for other modules (coordination, direction, activity) to work with, dynamically composing the robot's *Weltanschauung*

Advantages

- Fault tolerant
- Reusable
- Allows for experimentation
- Potential parallel execution, which can greatly enhance cost-effectiveness over traditional message-based systems

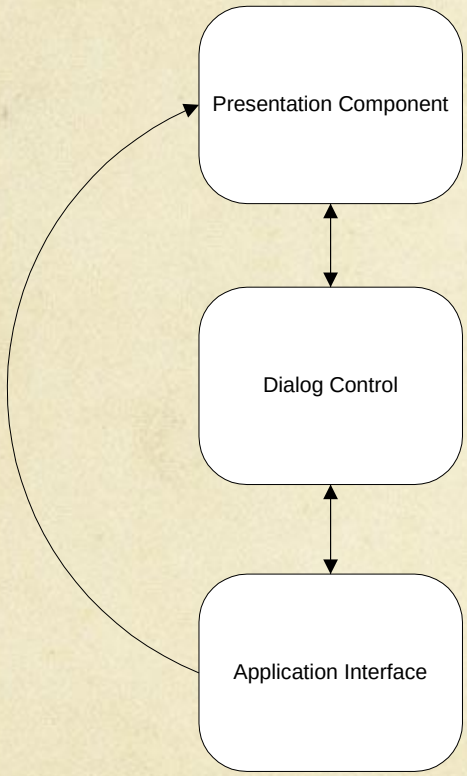
Disadvantages

- Low testability
- Solutions are variable in terms of quality and not guaranteed
- Usually inefficient, costly in terms of execution

Model View Controller (MVC) Pattern

Operative Metaphor:
Separation of Concern

A Bit of GUI History



- Early HCI (Human Computer Interaction) focused on two primary goals:
  - to minimize the effects of interface changes on the application as a whole
  - to promote portability among different windowing systems
- The Seeheim model (left) was the first model to focus on a solution to these problems
- Notice the bridging between layers

Problems

- The Seeheim model suffered from two main difficulties:
  - when replacing a presentation component, one usually had to rewrite the dialog to accommodate the new features of the new presentation component
  - Each dialog tended to need to be presentation based, requiring a change in the presentation component each time the dialog changed

Model-View-Controller

- MVC got its start with Smalltalk 80, and tried to address some of the same issues
- Spawned from early HCI (Human Computer Interaction) problems, where a need existed to *insulate* the "system" from the "fickle" user interface
- How do we update and support multiple renditions of data from a single set of data?
- The MVC pattern separates responsibilities so that:
  - multiple views may individually render the data
  - a single data model may be rendered in a variety of different ways by the multiple views
  - a controller controls the communication between the various views and the singular data model

Approaches

- The Seeheim model's guard against the impact of changes was *layering*.
- MVC's guard against the impact of changes was to encapsulate the functionality into distinct components

Domain-Application Segregation

- The complete segregation of domain information from the application model becomes essential when an application of any merit is intended. – Tim Howard *The Smalltalk Developer's Guide to VisualWorks*

Examples of Domain Adaptor Architecture

- Smalltalk (domain adaptors as part of application model providing a GUI for domain objects)
- J2EE (JavaBeans, EJBs)
- Simple example of MVC concepts: Excel spreadsheet with multiple graphs of data

Domain Adaptor Architecture

- Domain information belongs in a database
- Application information belongs in the application (GUI objects, dependencies, etc.)
- Domain information should be completely independent of the application displaying it, and should *feed* presentation objects their information, in a highly loosely-coupled manner
- Domain objects usually implement some business logic

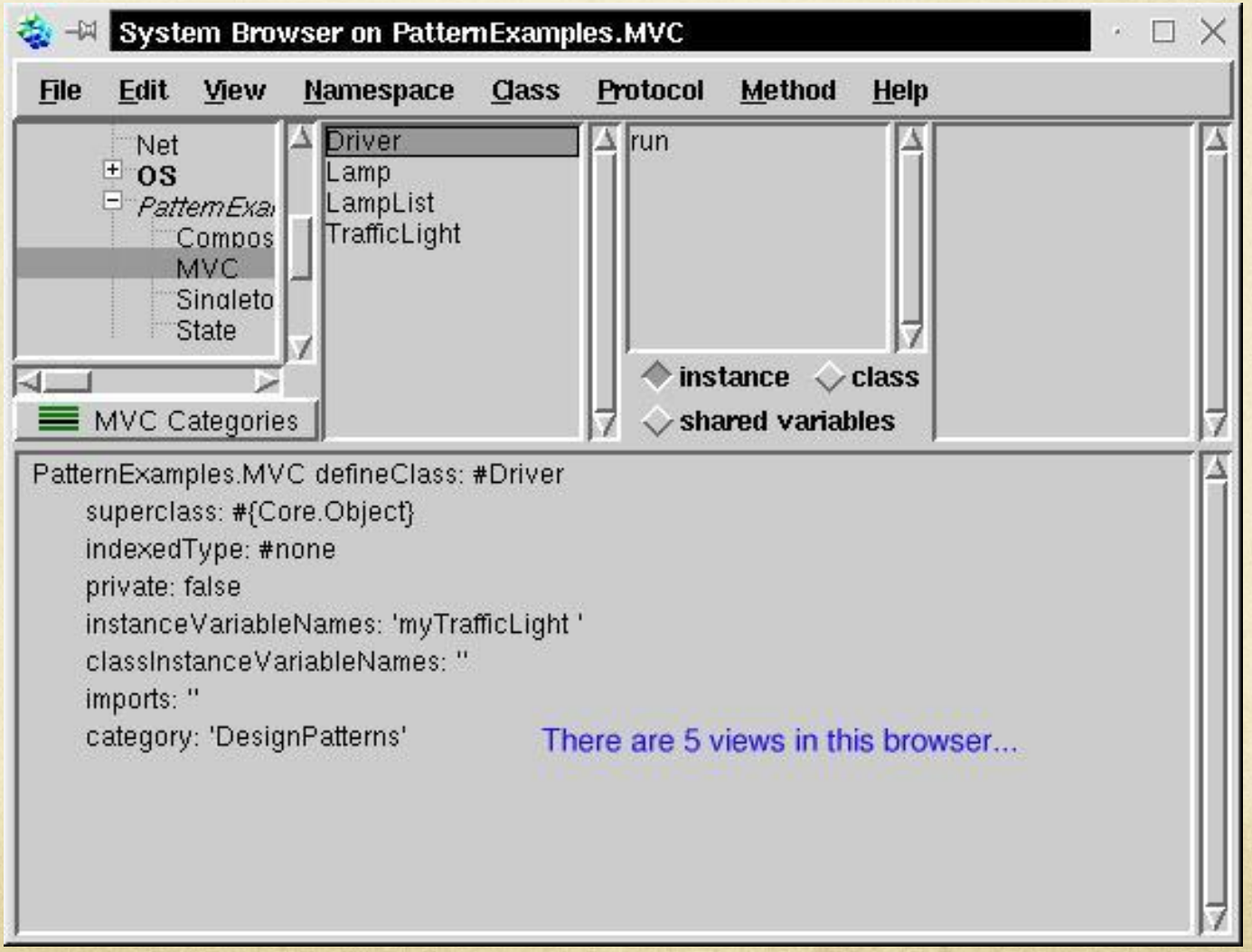
The Model

- The model encapsulates and manipulates the domain data to be rendered
- The model has no idea how to display the information is has nor does it interact with the user or receive any user input
- The model encapsulates the functionality necessary to manipulate, obtain, and deliver that data to others, *independent* of any user interface or any user input device

The Model

- Each model has a list of *dependents*, about which is knows nothing: the dependents are interested in the model, the model is not interested in its dependents
- Whenever the model changes, it first notifies itself of the change, and then notifies all of its dependents of the change
- In Smalltalk, each piece of information in a domain object (model implementation) is called an *aspect*.

MVC in Smalltalk



The View

- A View is a specific visual rendering of the information contained in the model.
- A view may be graphical or text based
- Multiple views may present multiple renditions of the data in the model
- Each view is a *dependent* of a model
- When the model changes, all dependent views are updated (using a publish-subscribe pattern)

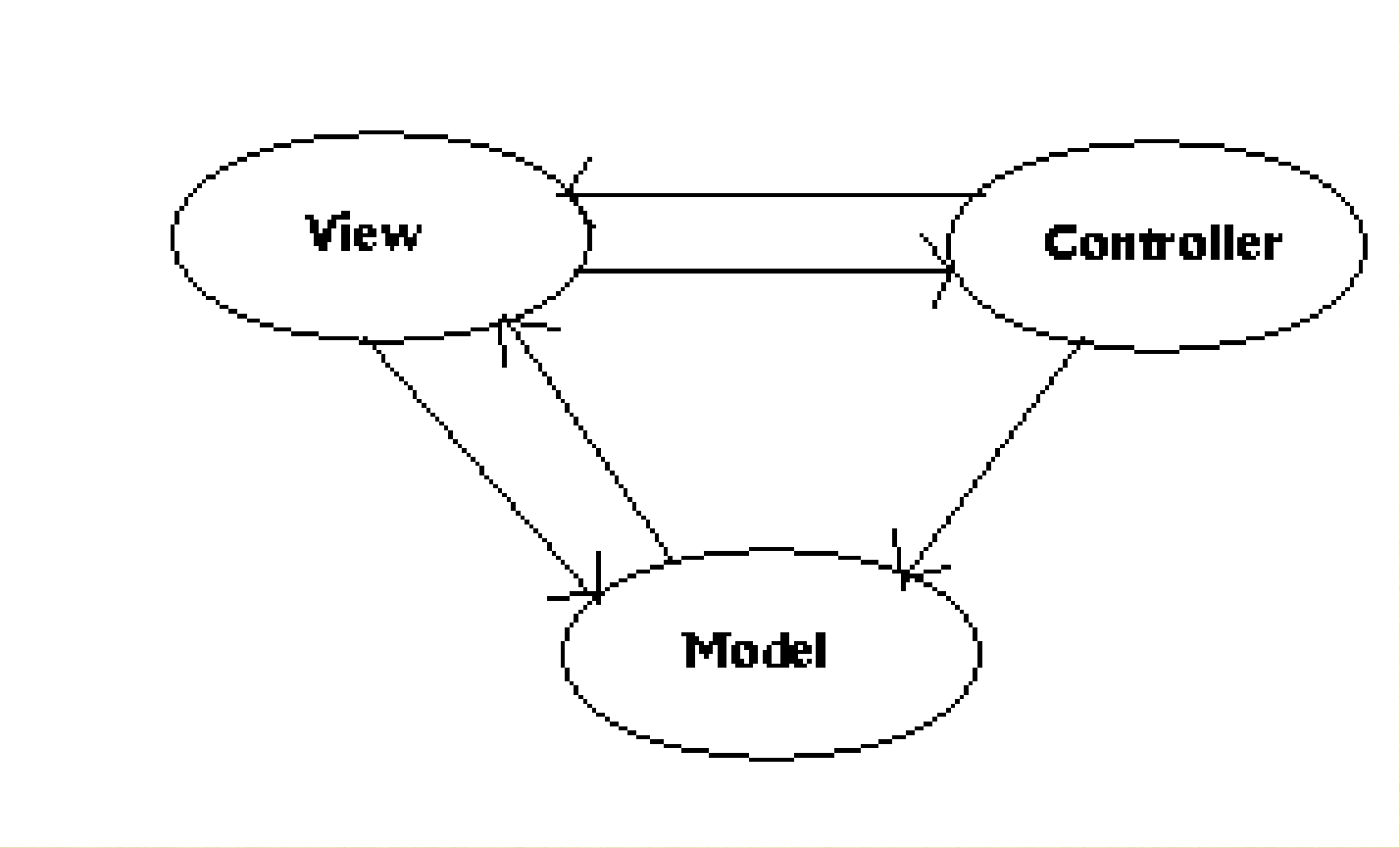
Model-View Connection

- Implemented via the Observer pattern (cf. GoF)
- Earlier Smalltalk solutions involved the use of *holder* objects which would wrap an individual widget's value (copy from domain object to holder to GUI, and vice versa)
- AspectAdaptors are *observers* that hold information for a specific field in the domain object (the *aspect*) of the *subject* to which they are attached (over a *subject channel*)

The Controller

- Controllers handle user input. They "listen" for user direction, and *handle* requests using the model and views
- Controllers often watch mouse events and keyboard events
- The controller allows the decoupling of the model and its views, allowing views to simply render data and models to simply encapsulate data
- Controllers are "paired up" with collections of view types, so that a "pie graph" view would be associated with its own "pie graph" controller, etc.
- The *behavior* of the controller is dependent upon the *state* of the model

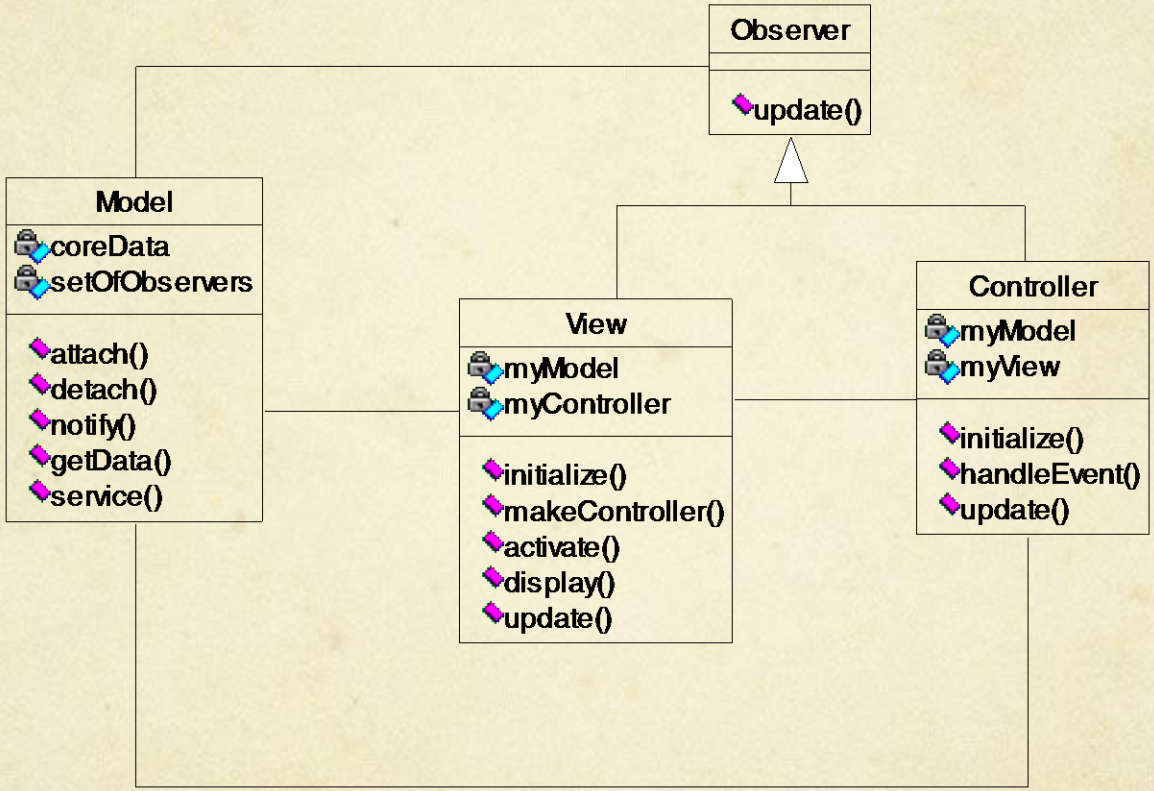
MVC Communication



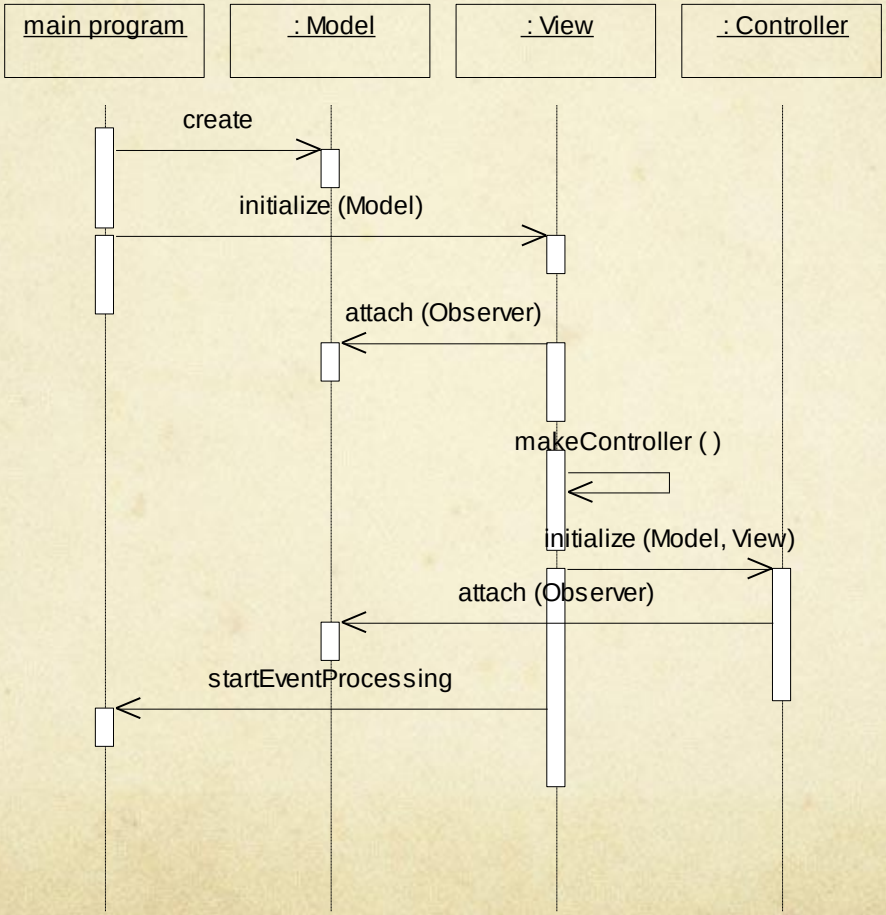
How Does It Work?

- The model has a list of views it supports
- Each view has a reference to its model, as well as its supporting controller
- Each controller has a reference to the view it controls, as well as to the model the view is based on. However, models know nothing about controllers.
- On user input, the controller notifies the model which in turn notifies its views of a change
- Changing a controller's view will give a different *look*.
  Changing a view's controller will give a different *feel*.
- In this way, the model is decoupled from its views in terms if *management*

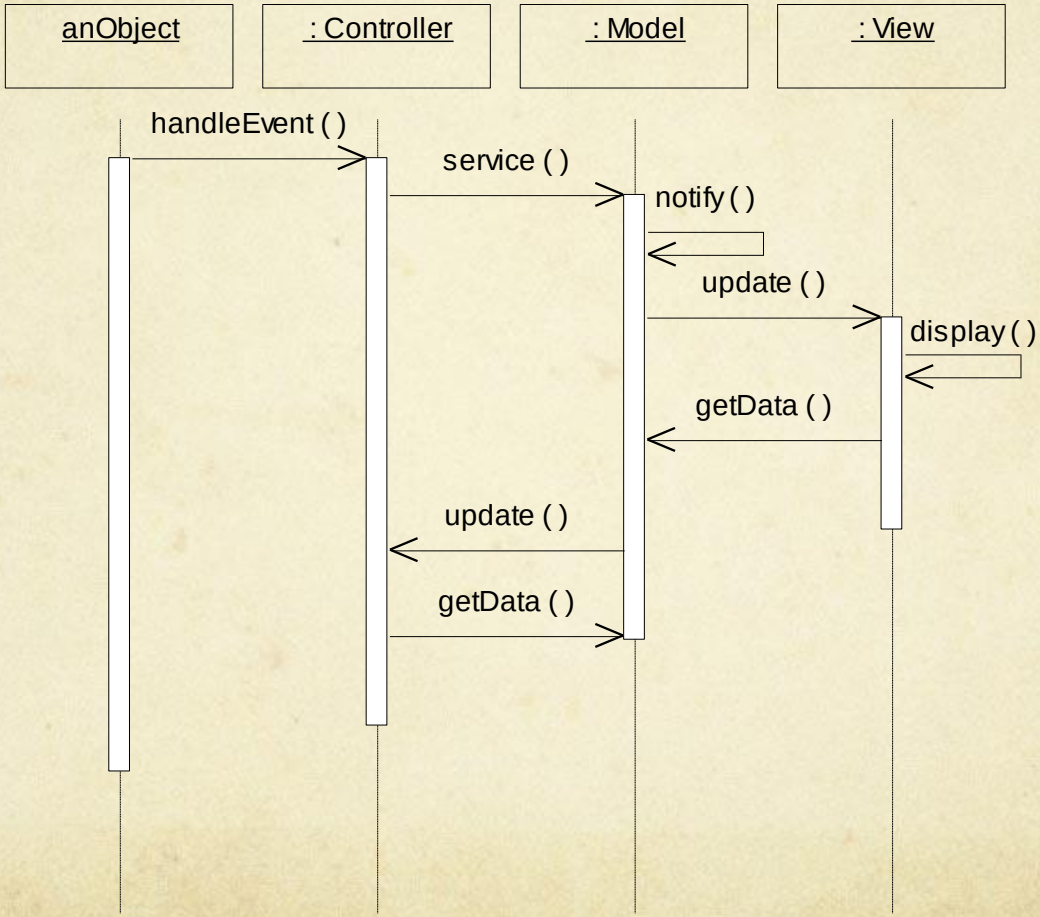
MVC UML


Observer
update()
Model
coreData
setOfObservers
attach()
detach()
notify()
getData()
service()
View
myModel
myController
initialize()
makeController()
activate()
display()
update()
Controller
myModel
myView
initialize()
handleEvent()
update()

MVC Scenario



User Input Scenario



Reflection Pattern

Operative Metaphor:
I'm Ok, You're Ok

Reflection

- The Reflection pattern allows runtime discovery of interfaces and dynamic calling of discovered interfaces

Known Examples

- CLOS (Common Lisp Object System) Metaobject Protocol (MOP)
- Java offers Reflection through capabilities provided in the Java Reflection API defined in the java.lang.reflect package, which provides classes for the runtime analysis of class files
- CORBA offers Reflection in terms of its Dynamic Invocation Interface (DII) and Interface Repository

CLOS

- Although Lisp began with McCarthy's research on recursive descriptive languages in 1956-58, Lisp did not entertain object oriented concepts until 1986 (Symbolics, Xerox, Lucid (Gabriel))
- The Common Lisp Object System (CLOS) offered a new and intriguing concept: Metaobjects
- CLOS has now been incorporated into Common Lisp

Metaobjects

- For each kind of *base* lisp element, there is a corresponding metaobject class (class, slot-definition, generic function, method, etc.)
- A metaobject class encapsulates information about a class itself: it's methods and properties
- Metaobjects can be used to *interrogate* a class as well as to dynamically *modify* it, as well as *instantiate* a new class from a textual model
- Metaobjects control the behavior of CLOS itself, and as CLOS is a program written in CLOS, CLOS is inherently self modifying

CLOS Class Metaobjects

- A class metaobject defines the structure and *default* behavior of its instances, specifically:
  - Class name (runtime mutability)
  - Immediate base and derived classes
  - Slots (attributes with optional default initializers/getters/setters)
  - Class documentation (what is this?)
  - Class methods

Metaobject Protocols

- Metaobject protocols are interfaces to programming environments via programming APIs that give programmers the ability to modify the language's behavior, in addition to providing the ability to program "normally" within that environment.
- Language semantics become blurred in that the very semantics themselves are modifiable by the programmer/user based on generic OO principles
- MOPs allow users to create variant derivative languages easily

Java Reflection

- Reflection was added to Java to support Remote Method Invocation, object serialization, and JavaBeans
- Reflection is used by JavaBeans to determine the properties, events, and methods that are supported by a particular bean.
- The Introspector class (static getBeanInfo()) will return a BeanInfo object which encapsulates the reflection information

.NET Reflection

- Reflection is provided in .NET via the metadata stored in the .NET assembly (portable executable format)
- Reflection on the metadata in the assembly is how, for example, MS Visual Studio provides detailed type-as-you-go prompting for method calls and their parameters
- Examples: reflect.exe, reflection.exe

CORBA Reflection

- The CORBA Interface Repository is simply another CORBA object that stores the contents of IDL defined types
- Once you add an IDL type to the repository, any other CORBA object can query the repository at runtime and:
  - *dynamically* understand an interface and make calls on it
  - *dynamically* manipulate an interface by *adding* or *deleting* interface methods (the adapter pattern on speed)