Topic 7

MVC and Observer Patterns

Learning Outcomes

- **Identify** the impact of design decisions on software quality when developing an interactive system
- **Describe and Document Diagrammatically** the structure of the Model View Controller (MVC) pattern and how the different components interact
- **Design** MVC modules and correctly assign code and functionality to each module
- **Implement** MVC using Java UI components and Event management classes and interfaces
- **Describe and Document Diagrammatically** the structure of the Observer pattern and how it is used with MVC
- **Implement** Observer behaviour within an MVC structure using Java classes and interfaces

Implementing a GUI based system

- One of the primary goals of *implementing* a quality* interactive (GUI based) system is to separate the user interface code from the application code.
- Benefits:
  - –User Interface can be more easily modified (portability, accessibility etc.)
  - –Application code can change without affecting the interface (e.g. text file based data is replaced by DBMS)
  - –Project can be more easily split across teams (requires good management, communication)
  - –Provides a logical basis for modularising the application

* See next slide

Software Quality

Comprehensible (easy to understand code)

Maintainable (easy to repair, change, maintain)

Extensible (easy to extend/add new features)

Reusable (easy to reuse components in isolation)

Reliable (fault free, does not fail or crash)

Secure (generally independent of software structure)

Efficient (good performance with minimal resource usage i.e. performance/resource utilisation)
can be a trade-off with the other factors above

Model View Controller

One technique for modularising an interactive system is the Model View Controller approach:

- –Originally conceived for the Smalltalk environment
- –Can be applied to any UI based system (even when UI is non graphical!)
- –Used internally within AWT/Swing
- –Can be implemented easily and effectively in Java with communication between Packages, Classes and Listeners
- –Increases *cohesion* while managing *coupling* in a structured way

Model View Controller

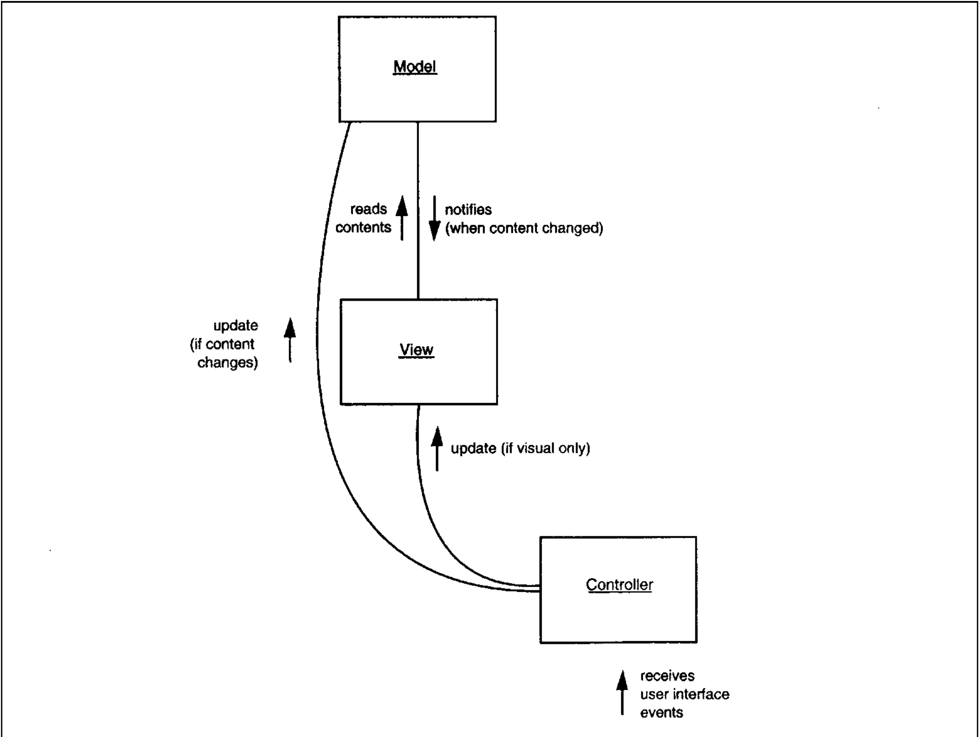

Model
reads contents
notifies (when content changed)
update (if content changes)
View
update (if visual only)
Controller
receives user interface events

Figure 9–3: Interactions between model, view, and controller objects

Model View Controller

Model: contains application code such as:

- data structures
- accessor methods
- I/O routines
- calculations etc.

View: the representation of the data i.e. how it is presented to the user

- (may be devices other than simple display terminal e.g. accessibility devices for disabled persons)
- multiple views of the same data (e.g. a spreadsheet has cell and chart views)

Controller: handles user interaction and mediates between the Model (data) and The View (representation)

- e.g user drags mouse in a drawing program. The controller modifies the series of points or vectors (data structures) in the model as the mouse is moved (user interaction) and instructs the view to redraw the data in the model (alternatively the model may automatically instruct it's registered views to redraw its data - example of Observer pattern [Design Patterns, Gamma et al. 1995])

• ***Consider Microsoft Excel as an Example***

Model View Controller

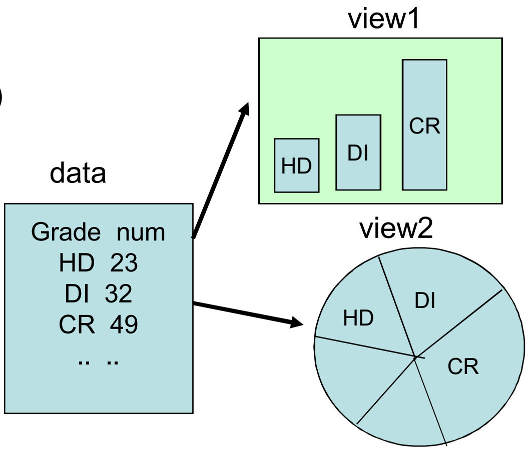
The Model, View and Controller are treated as separate <u>modules</u>

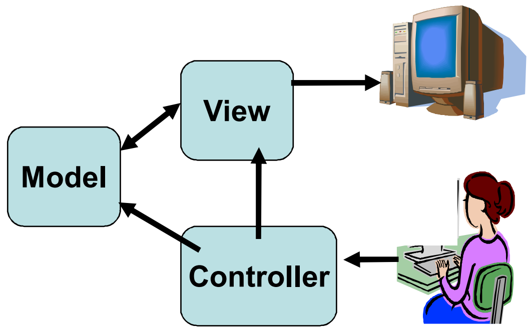
- may be classes or packages depending upon their complexity
- a system may have multiple views, controllers and models
- multiple views are quite common but not required
- nearly always a separate controller for each view (for cohesion) although these are often implemented as inner classes in Java and thus not entirely separate
- multiple models are usually used when there exist groups of independent program data

Model-View Controller Pattern

- Problem – Ensure any changes to data (model) is reflected by all observers (views)
- Solution – separate problem into three separate components
- Model provides the data and determines the current state of application
- View determines the appearance
- Controller interprets the events (mouse, keystroke)
- Commonly used in Web applications:

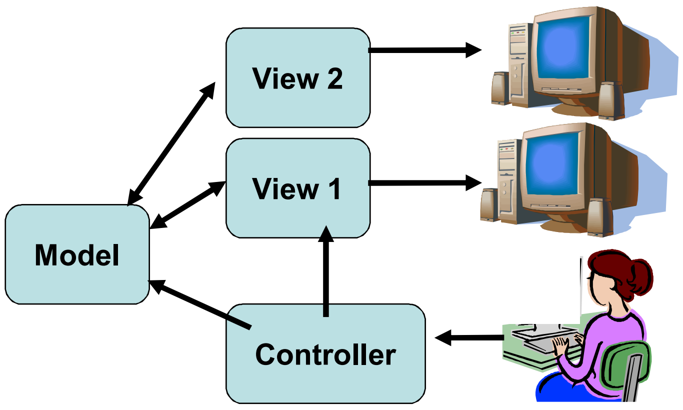
Model	->	Database or xml
View	->	HTML page
Controller	->	Script handling the events





MVC Events Sequence

- User performs action (using keyboard, mouse) and controller is passed the relevant data
- Controller may simply update view or specify changes to model
- Model may notify all attached views if it has changed
- View may query model to retrieve data and updates display



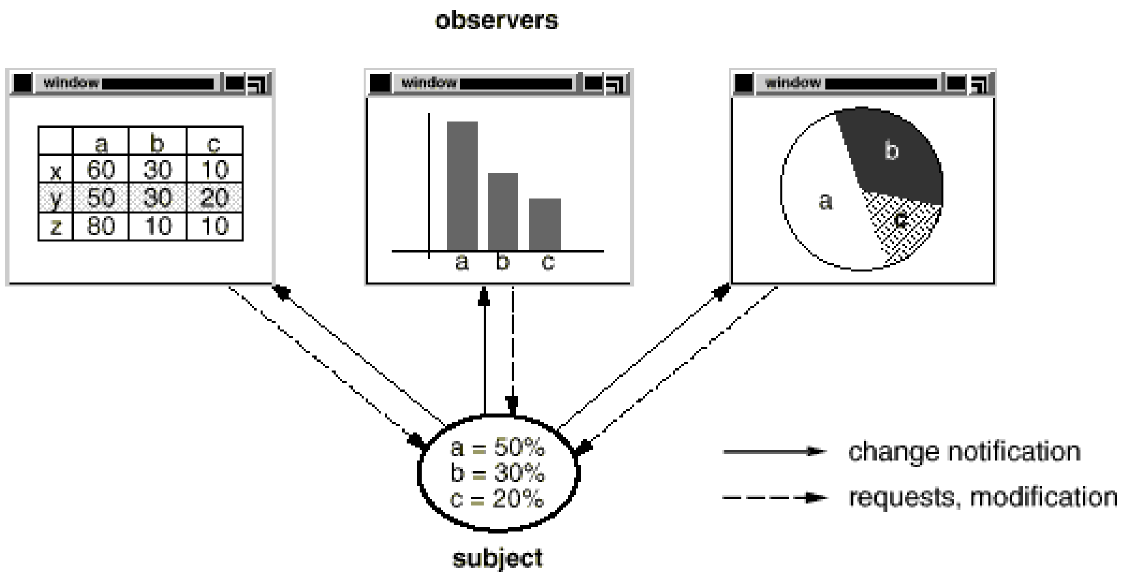
Benefits of MVC

- Results in less coupling between functionality and presentation
  - Easier to maintain (cohesive modules)
  - Separates roles clearly e.g. DB programmer, application programmer, usability expert, graphic design and GUI programmer etc.
- Ensures all views are consistent with current model
- Allows different types of views for same document
- Allows different user interface technologies for the same application
  - e.g. Java desktop app, web app, android app etc.
- Provides a well known pattern or structure

Observer: Motivation

- A spreadsheet view and bar chart are different presentations of the same application data object.
- The data objects are independent of the presentation layer.
- The different presentations do not know about each other.
- The presentations should be **notified** about changes in the data object.

Observer: Solution

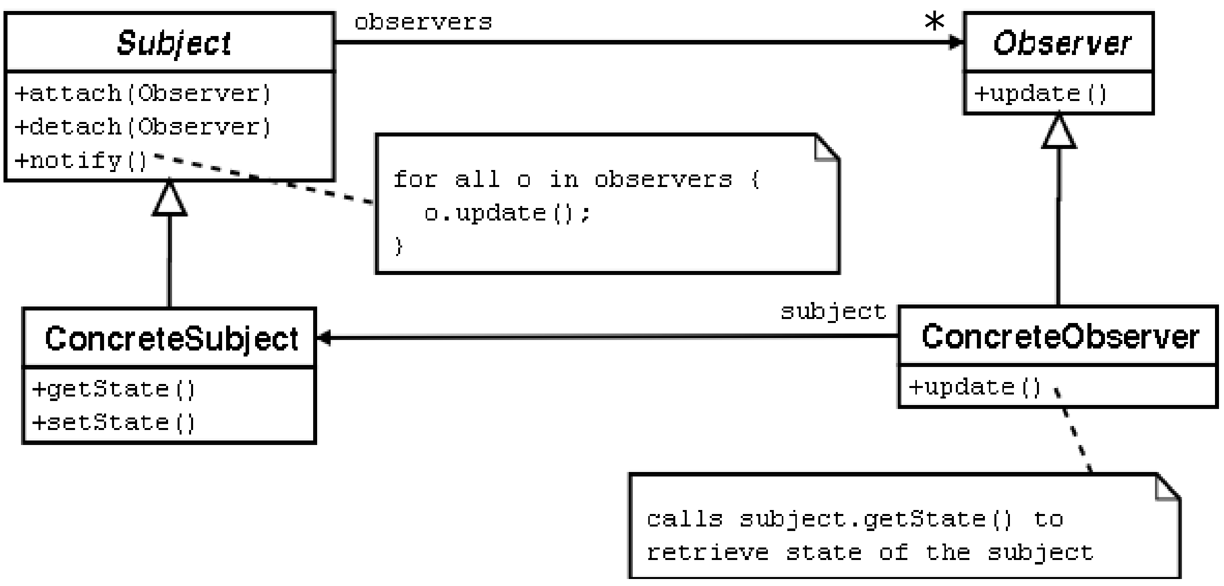


- Key objects: *subject* (Observable) and *observer*.
- A subject may have any number of dependent observers.
- All observers are notified whenever the subject changes its state.
- Each observer can query the subject to synchronize their states.

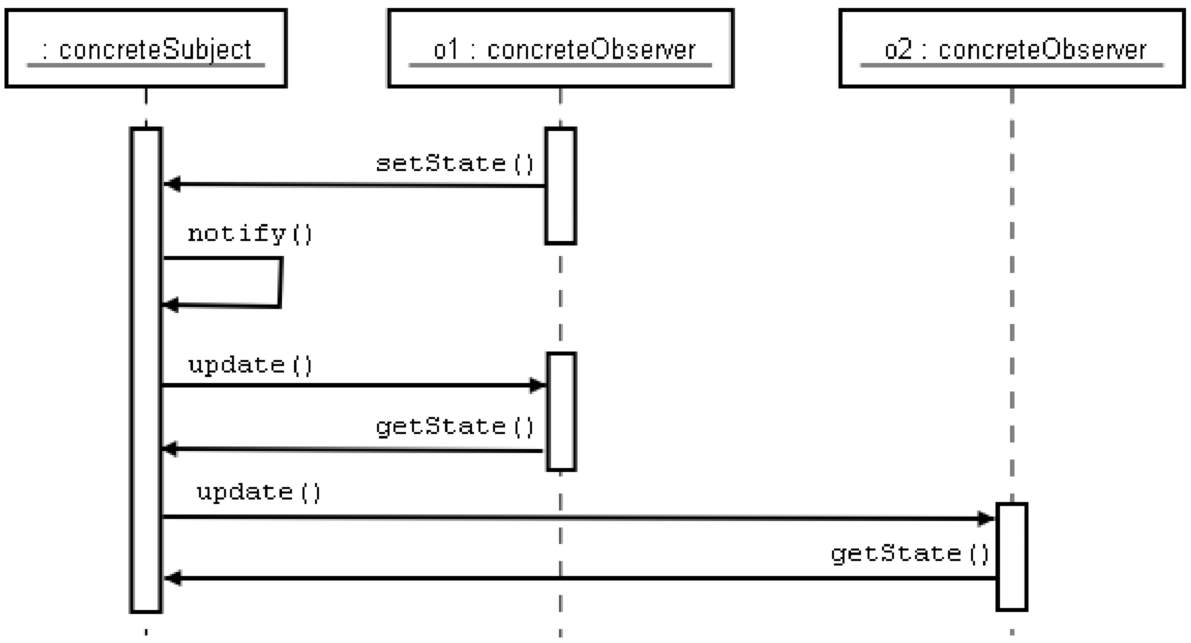
Observer: Participants

- *Subject*
  - knows its observers. any number of observers may observe a subject
  - provides an interface for attaching/detaching observers
- *Observer*
  - defines an updating interface for objects that should be notified of changes
- *ConcreteSubject*
  - stores state of interest to ConcreteObserver objects
  - sends a notification to its observers when state changes
- *ConcreteObserver*
  - maintains reference to a ConcreteSubject object
  - stores state that should stay consistent with subject's
  - implements the Observer notification interface to keep its state consistent with the subject's

Observer: Class Diagram



Observer: Sequence Diagram



Observer: intent and context

- Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically
- apply *Observer* when
  - an abstraction has two aspects, one dependent on the other.
  - a change to one object requires changing others
  - an object should be able to notify other objects without making assumptions about the identity of these objects.

Observer Example

- example that takes keyboard input and treats each input line as an event.
  - It uses the Java API classes Observer and Observable
  - When a string is entered in a system.in the method *notifyObservers*() is called by invoking their update() methods
  - The program reads the input in a separate thread
  - see `observer.*;` example

Observer: Considerations (1)

- Sometimes *observers* need to observe more than one *subject.*
- who triggers the update?
  - state-changing *subject* methods call *notify()* method, or
  - make clients responsible for calling *notify().*
- avoid dangling references when deleting *subjects*

  -- *Subject* notifies its observers about its deletion.
- make sure *Subject*'s state is self-consistent before calling *notify* (the *observers* will query the state).

Observer: Considerations (2)

- avoiding observer-specific update protocols
  - push model: subject sends its observers detailed information about the changes
  - pull model: subject only informs observers that state has changed; observers need to query subject to find out what has changed
- specifying modifications of interest explicitly
  - of interest when *observer* are interested in only some of the state-changing events:
    - *Subject.attach(Observer, interest)*
    - *Observer.update(Subject, interest)*
- encapsulating complex update semantics
  - when there is a highly complex relationship between subject and observer, introduce a ChangeManager class to reduce the amount of work.