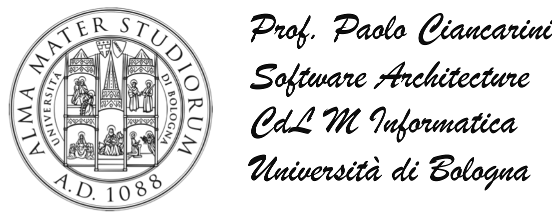
Architectural styles for user interface design



Prof. Paolo Ciancarini
Software Architecture
CdL M Informatica
Università di Bologna

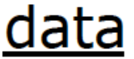
Agenda

- Motivation: user interfaces have components with several dependencies
- The Model View Controller architectural style and its variants
- The architectural styles of Web applications

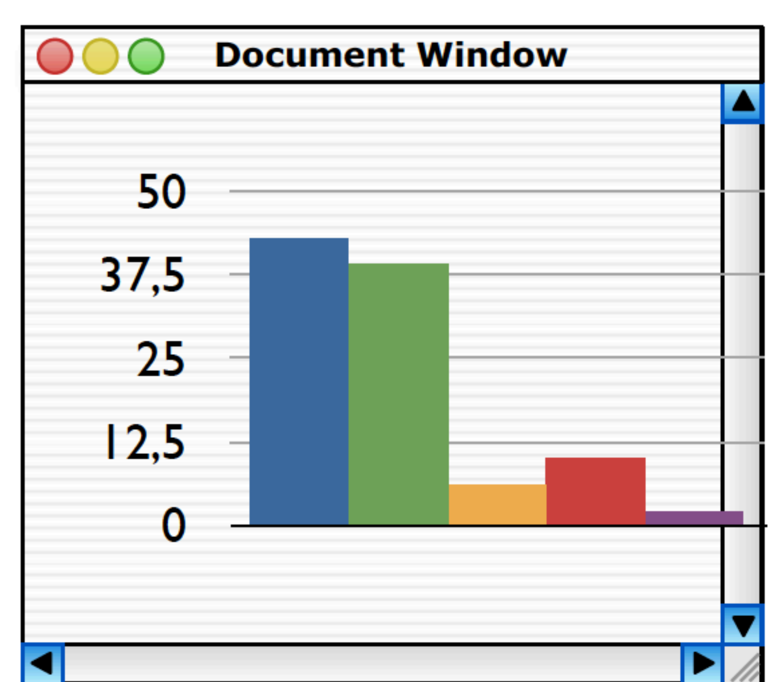
Problems with UI design

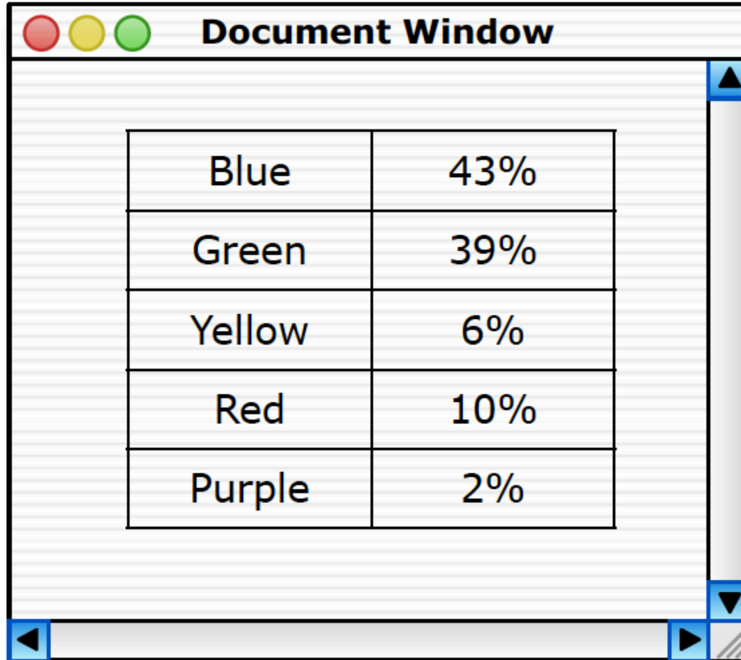
- User interfaces are often subject to changes in requirements, independent from the application
  - New types of input devices
    - Touch screen, mouse, special keyboards
  - New types of output
    - Porting to different “look-and-feel”
    - Alternative visualizations: charts, graphs, plots
    - Output heterogeneity: applets, Javascript, HTML, Swing
  - Prototyping the user experience
- The user interface changes more often than the business logic of a software product

Dependencies among windows



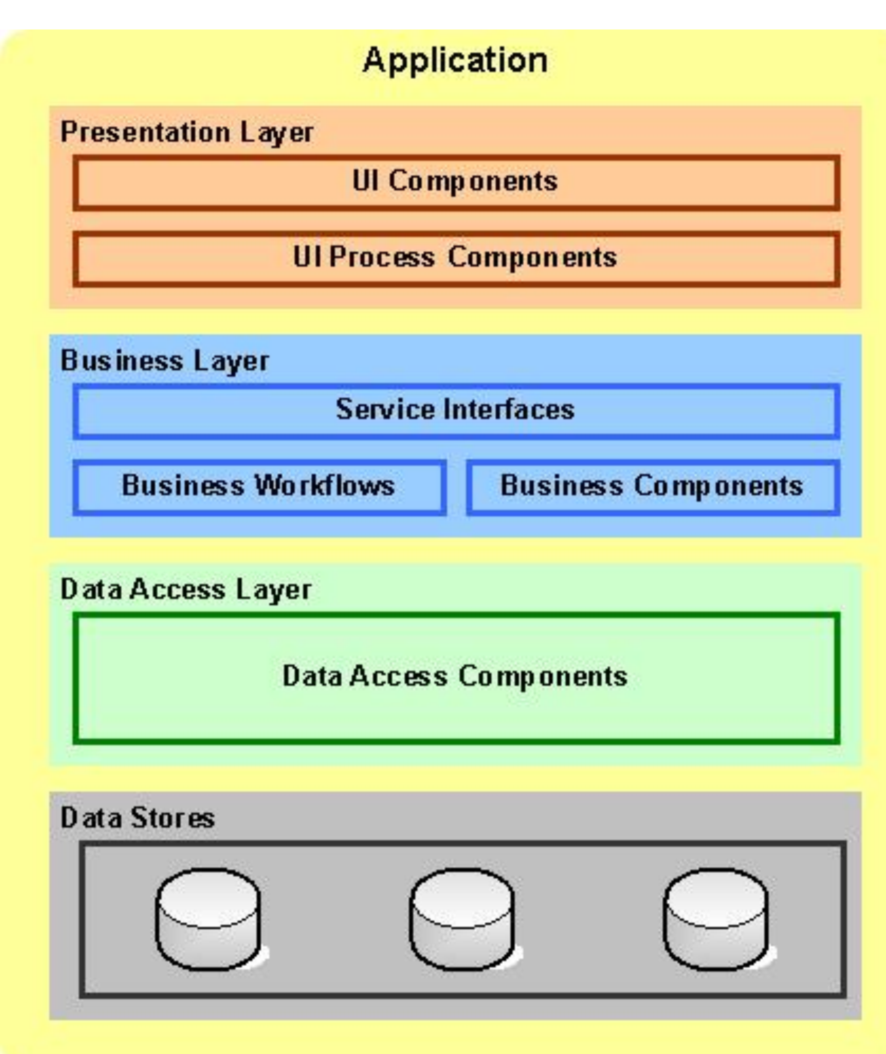
Blue: 43%
Green: 39%
Yellow: 6%
Red: 10%
Purple: 2%





Blue	43%
Green	39%
Yellow	6%
Red	10%
Purple	2%

Separation of the user interface



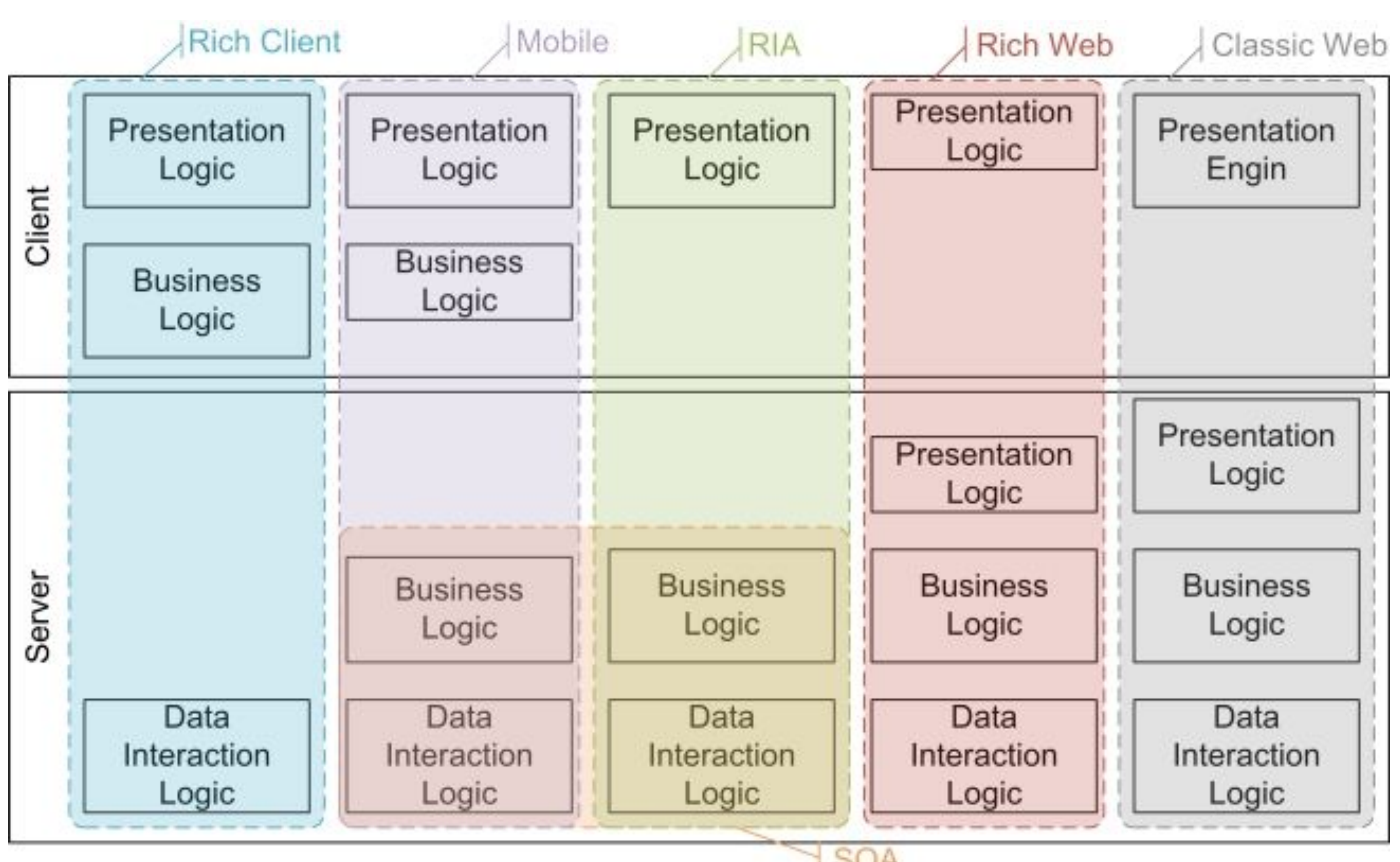
Web applications:

- The presentation tier changes often
- The business tier does not

Separation of the user interface: why it is convenient

- Market pressure
  - New technologies
  - Fashionable “Look and feel”
- Modify the ‘way of working’
  - E.g. form-based to task-based
- Extending an application architecture
  - E.g. “Webify” a rich client application

From a rich client to a classic web application



Design of the user interaction

Separate modeling of domain, presentation, and actions based on user input into three subsystems:

- Model
  - Manages behavior and data of application domain
- View
  - Manages display of information
- Control
  - Processes user input and informs model and/or view of change if appropriate

Burbeck "Applications programming in smalltalk-80 (tm): How to use model-view-controller" *Smalltalk-80 v2. 5. ParcPlace,* 1992.

The model, the view, and the controller

In a typical application we find three fundamental parts:

- Data (Model)
- An interface to view and modify the data (View)
- Operations that can be performed on the data (Controller)

- The model represents the data, and does nothing else; it does NOT depend on the controller or the view
- The view displays the model data, and sends user actions (e.g. button clicks) to the controller. The view can be independent of both the model and the controller; or actually be the controller, and therefore depend on the model
- The controller provides model data to the view, and interprets user actions such as button clicks. The controller depends on the view and the model.
- In some cases, the controller and the view are the same object

```
//Example 1 (no MVC):
void Person::setPicture(Picture pict){
    m_picture = pict; //set the member variable
    m_listView->reloadData(); //update the view
}


//Example 2 (with MVC):
void Person::setPicture(Picture pict){
    m_picture = pict; //set the member variable
}

void PersonListController::changePictureAtIndex(
                                   Picture newPict, int personIndex){
    m_personList[personIndex].setPicture(newPict); //modify model
    m_listView->reloadData(); //update the view
}
```

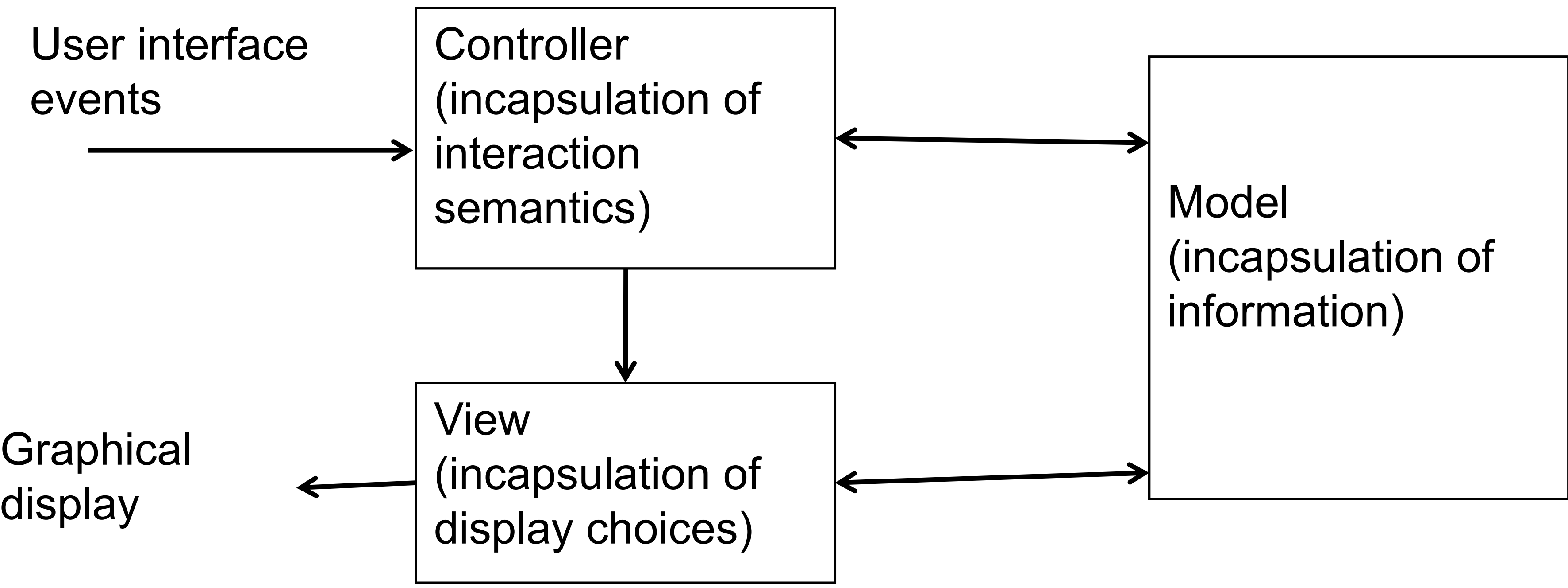
Problems

- Business logic need knowledge of the UI
  - E.g. coloring a field when given a value
- Different presentation tiers have different capabilities
  - E.g. can we still color a field when given a value
- UI performance can influence the implementation
  - E.g. gather more data before sending

Typical requirements

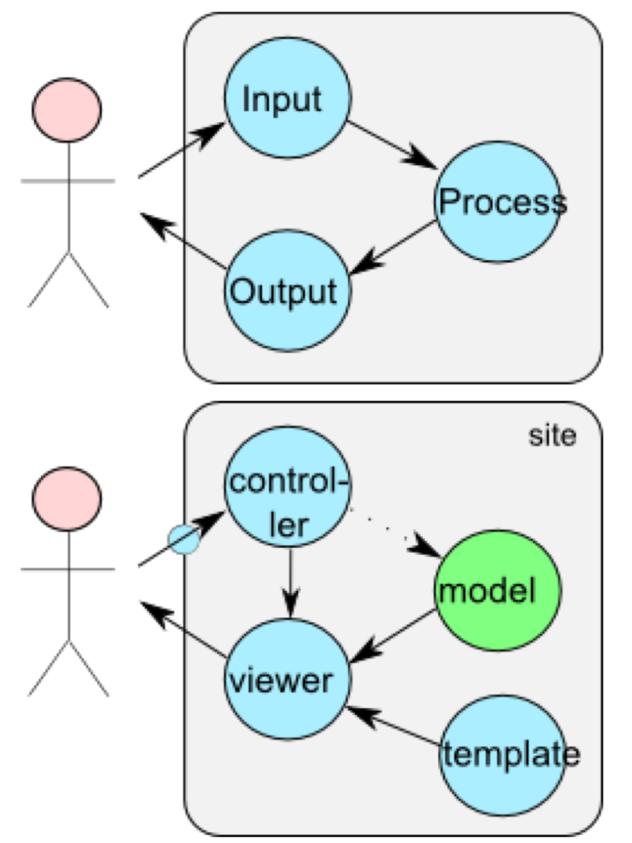
- Decouple presentation from business logic
  - Define service interface & data requirements
- Integrate local and remote data sources prior to display
- Enable connected and disconnected use

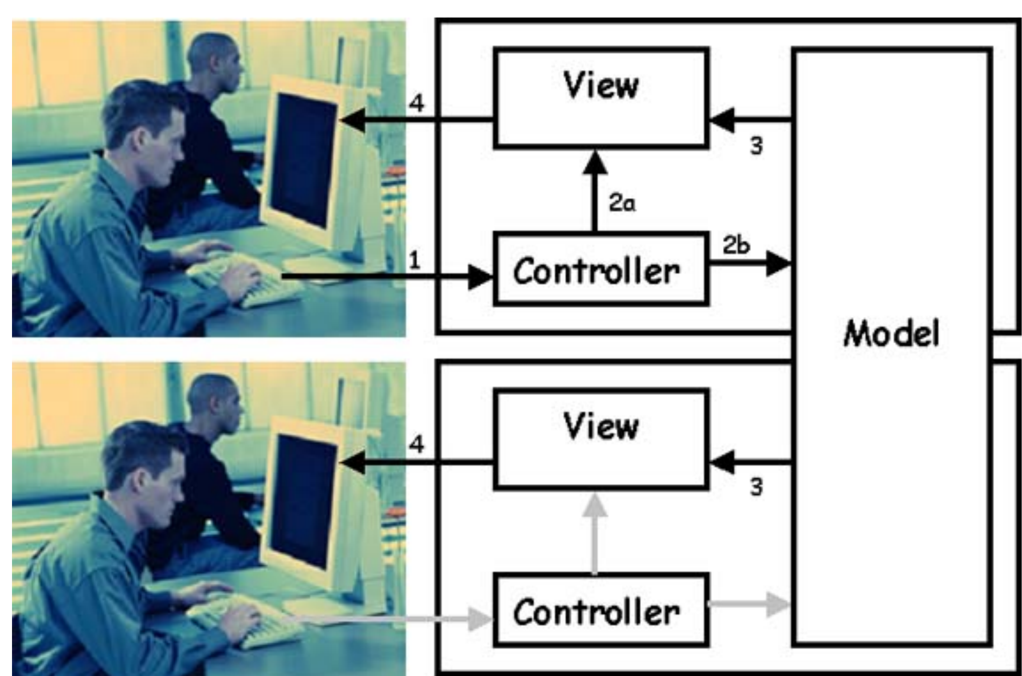
Design of user interactions



MVC goal

- The goal of MVC is, by decoupling models and views, to reduce the complexity in architectural design and to increase flexibility and maintainability





MVC participants

The MVC style has three participants:

•The *Model* is a representation of the application data.

•The *View* is the collection of visual elements presented to the end user

•The *Controller* coordinates changes to the model and the view based on business logic execution.

This specification is sufficiently vague and allows for many possible designs and different behaviors.

In particular, it does not specify how the three participants interact with each other.

Model: CRC

Class Model	**Collaborators**
Responsibilities • Provides the functional core • It is a Repository for persistent data • Registers Views and Controller interest in data • Notifies registered Views or Controllers about data changes	• View • Controller

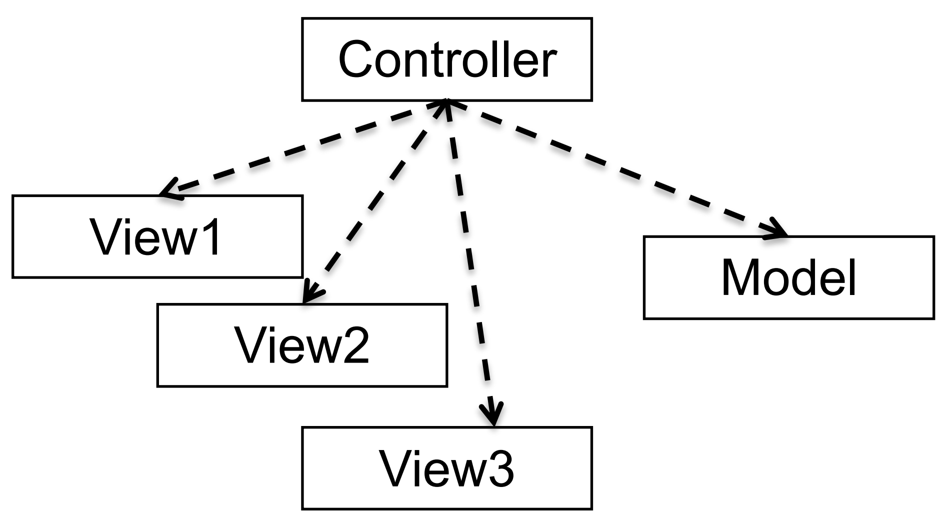
View: CRC

Class View	**Collaborators**
Responsibilities • Displays information to the user • Creates and starts its Controller • Updates when new data arrive from the Model	• Model • Controller

Controller: CRC

Class Controller	**Collaborators**
Responsibilities • Handles input events from the user • Translates an event in a query for the Model or View • Updates when new data arrive from the Model	• Model • View

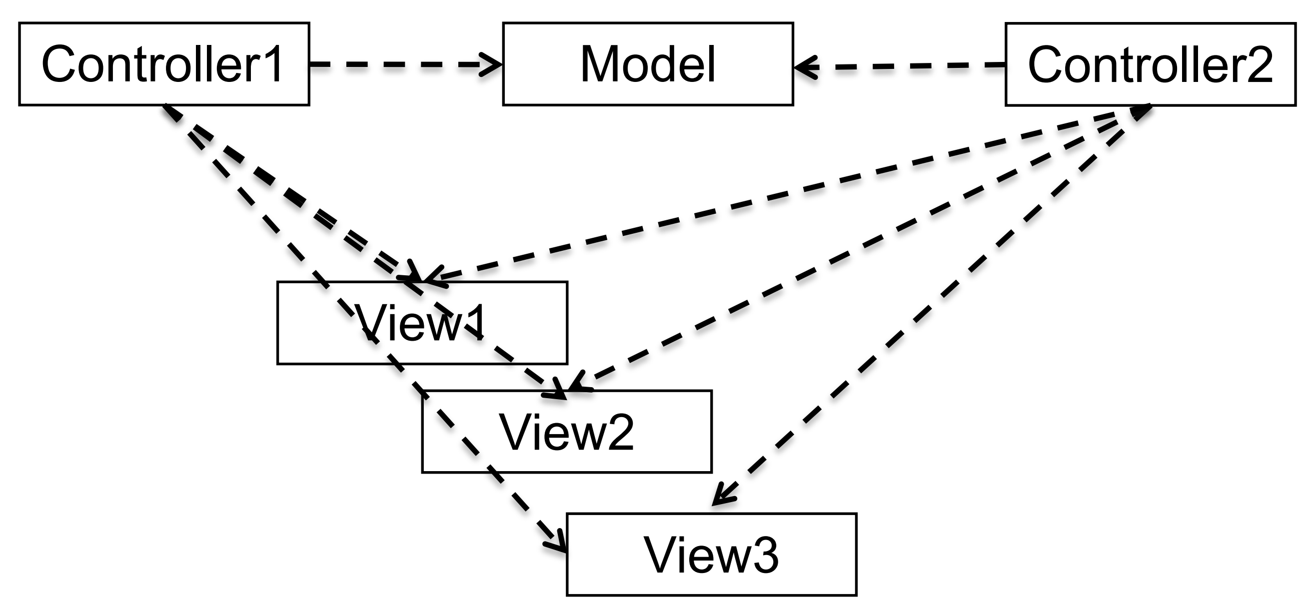
Unique controller



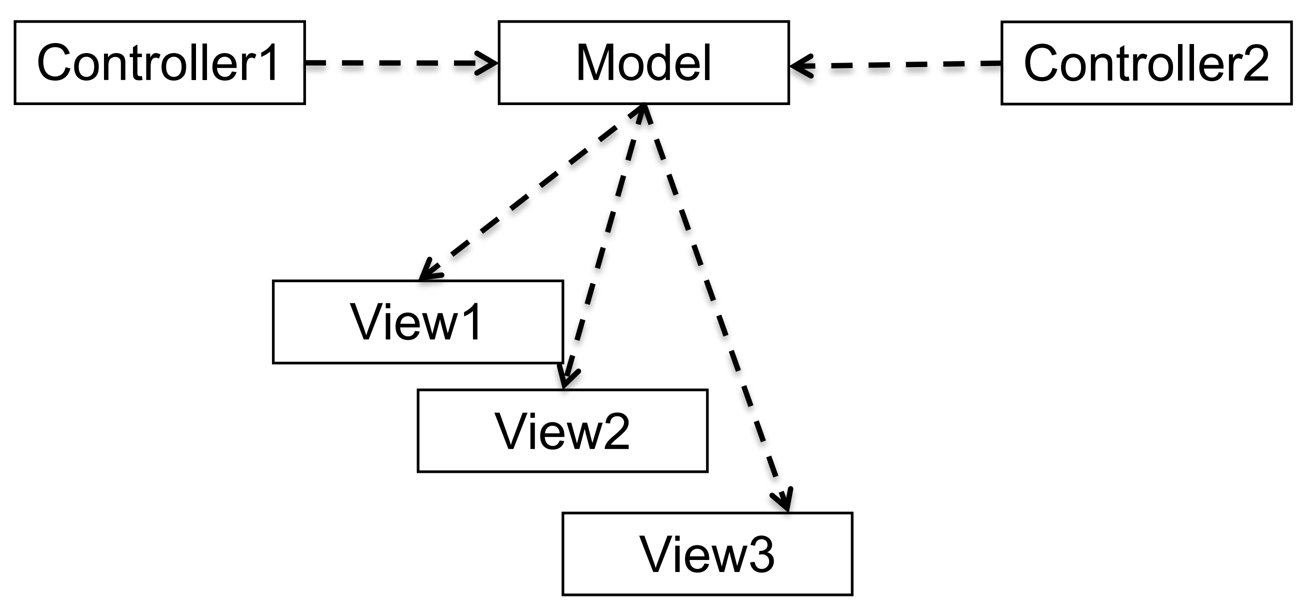
This architecture is problematic if requests to change the model come from many different sources (such as GUI components, keyboard and mouse event listeners, and timers), so that it is not practical to combine all of them in one class.

Multiple controllers

- Strong coupling!



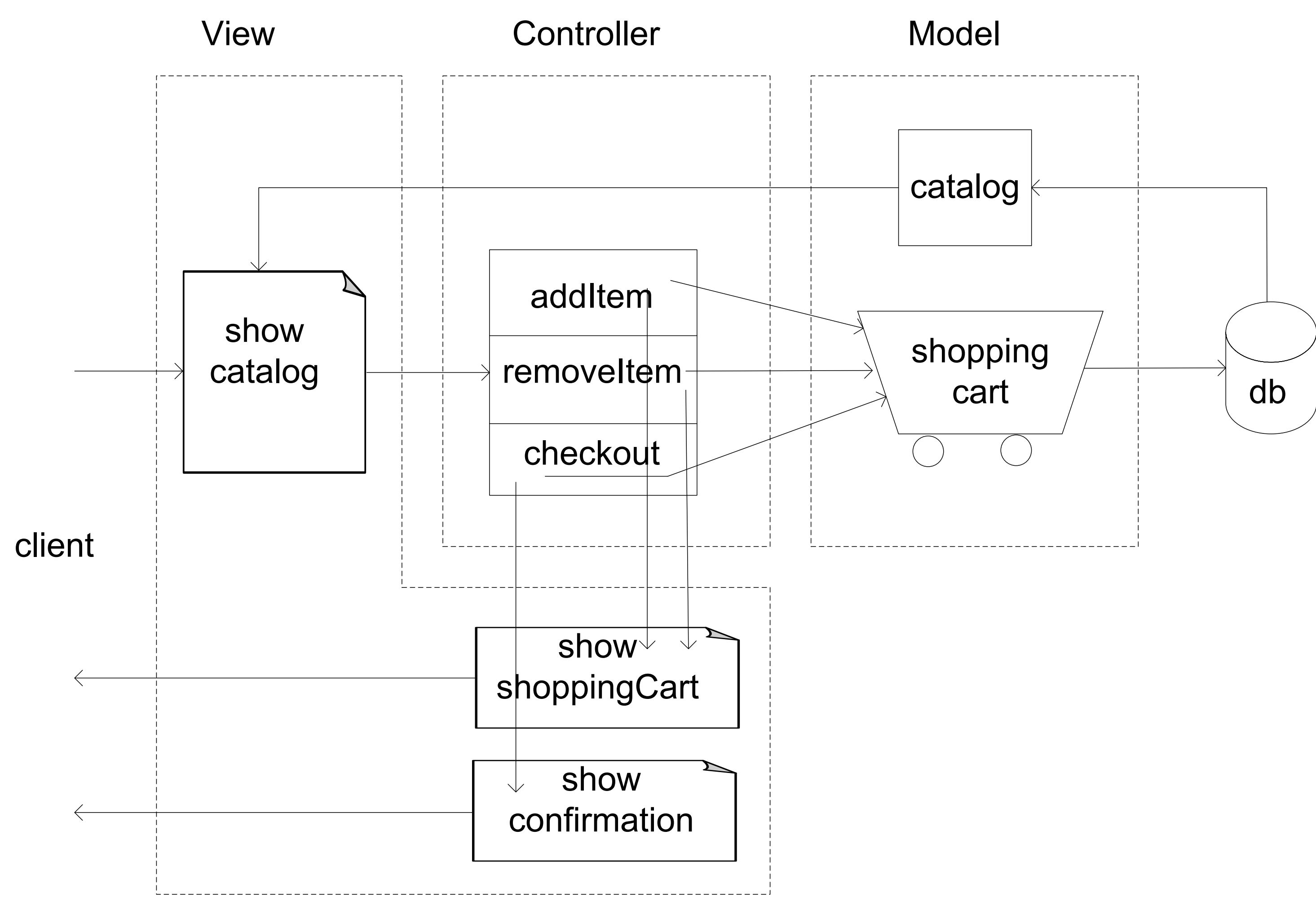
Model as “repository”



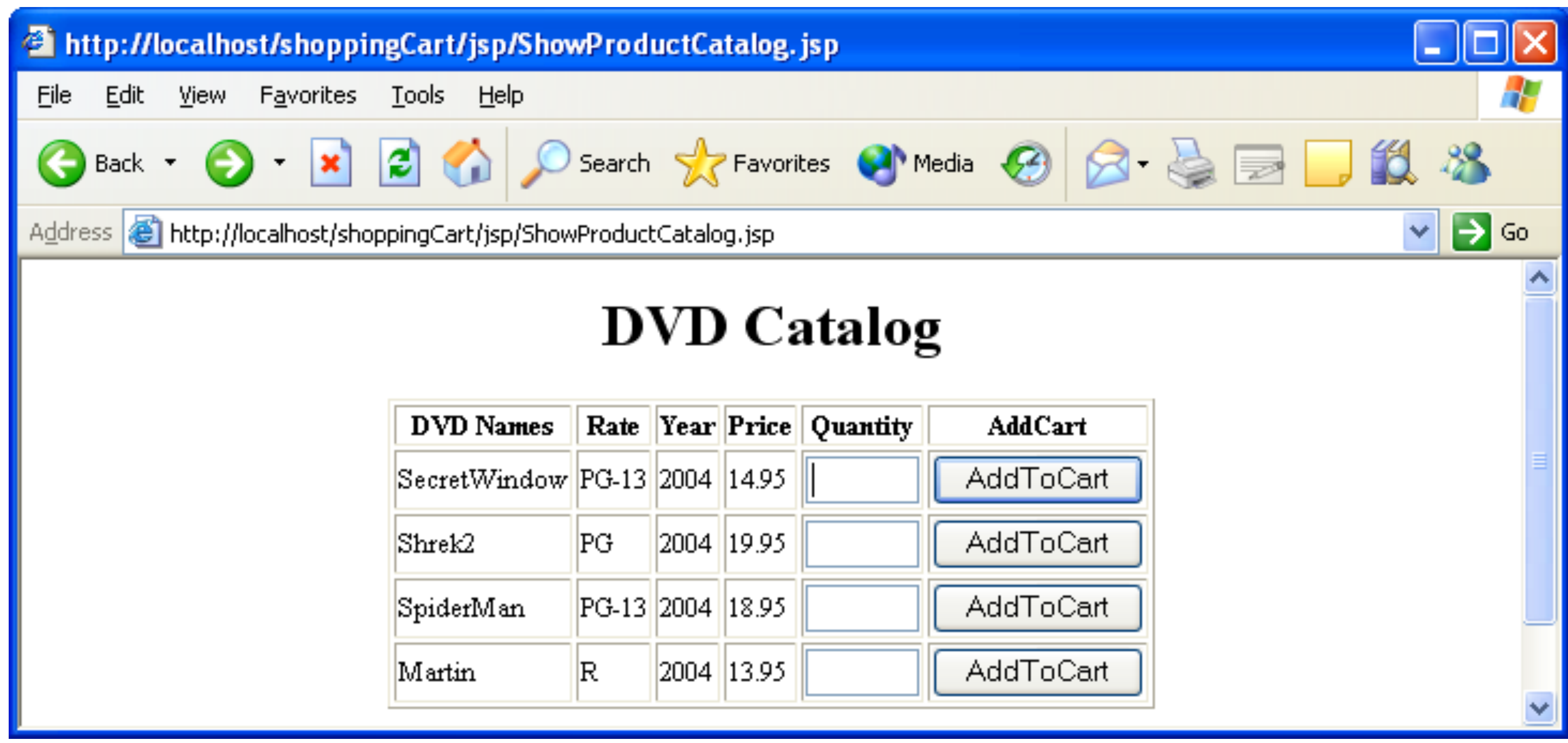
Example: online shopping cart

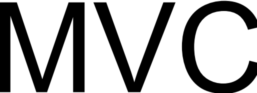
- There are a variety of ways to implement an online store
- The next slide shows a possible solution based on MVC
- The slide shows the Model, View, and Controller modules, the connections between them, and some back-end database support
- This is a simplified design: many features of an online store are not included in this implementation such as customer information processing, shipping and handling processing, accounting processing, etc.

Online store: architecture



Online store: A view





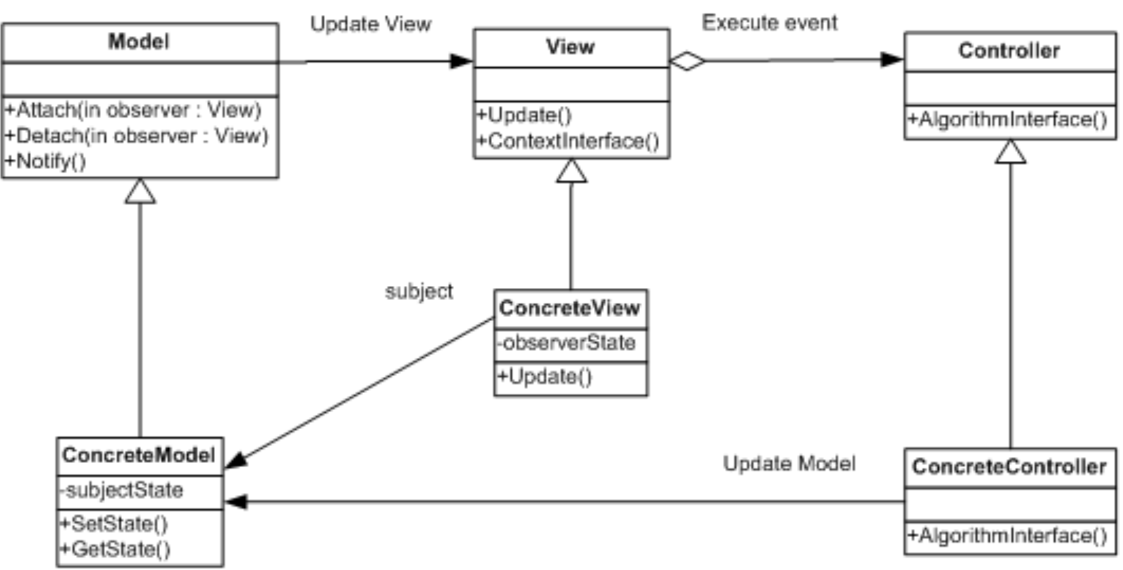
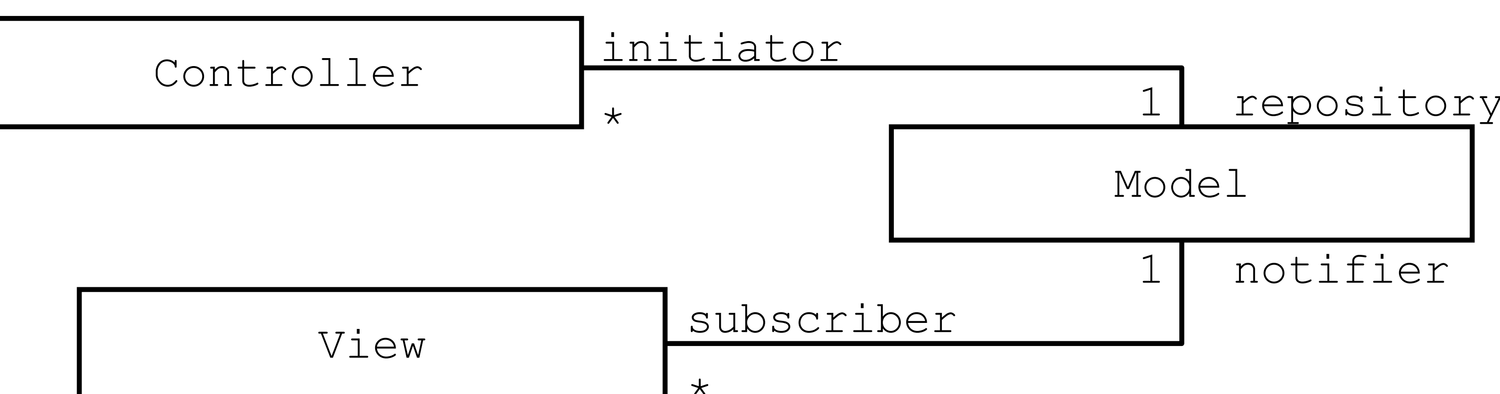


Figure 3: MVC

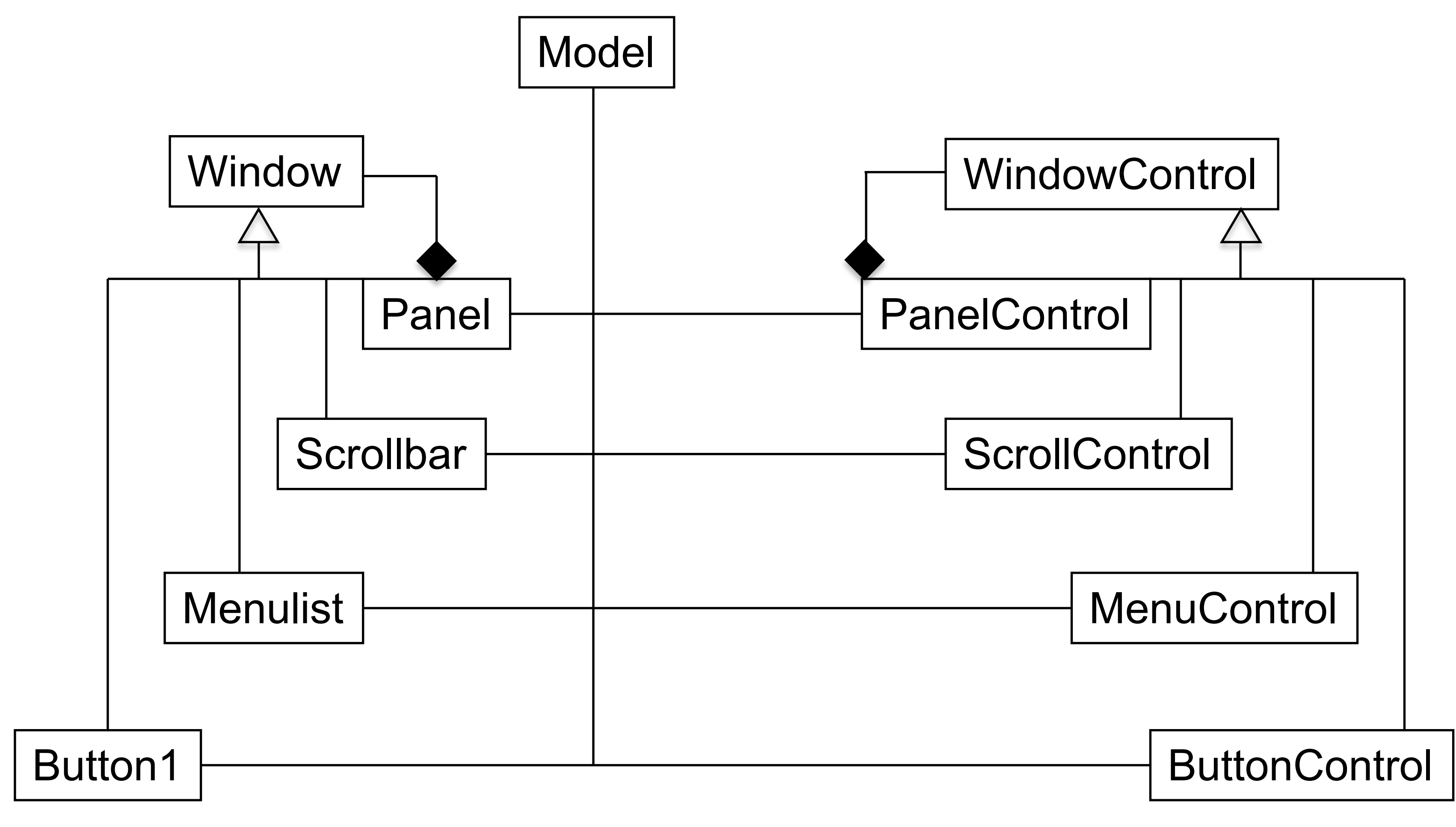
```
public abstract class Model
{
    private readonly ICollection<View> Views = new Collection<View>();
    public void Attach(View view)
    {
        Views.Add(view);
    }
    public void Detach(View view)
    {
        Views.Remove(view);
    }
    public void Notify()
    {
        foreach (View o in Views)
        {
            o.Update();
        }
    }
}

public class ConcreteModel : Model
{
    public object ModelState { get; set; }
}

public abstract class View
{
    public abstract void Update();
    private readonly Controller Controller;
    protected View()
    {
    }
    protected View(Controller controller)
    {
        Controller = controller;
    }
    public void ContextInterface()
    {
        Controller.AlgorithmInterface();
    }
}

public class ConcreteView : View
{
    private object ViewState;
    private ConcreteModel Model { get; set; }
    public ConcreteView(ConcreteModel model)
    {
        Model = model;
    }
    public override void Update()
    {
        ViewState = Model.ModelState;
    }
}

public abstract class Controller
{
    public abstract void AlgorithmInterface();
}

public class ConcreteController : Controller
{
    public override void AlgorithmInterface()
    {
        // code here
    }
}
```

The MVC style: class diagram

- **Model** has the knowledge about the application domain – it is also called *business logic*
- **View** takes care of making application objects visible to system users
- **Controller** manages the interactions between the system and it users



A hierarchy of views and controllers



MVC and design patterns

In [GoF] MVC is not defined as a design pattern

- but as a “set of classes to build a user interface”
- The main structural pattern used is Composite
- The main behavioral patterns used are Observer and Strategy

MVC can also use:

- Factory Method to specify the default controller class for a view;
- Decorator to add scrolling to a view;
- Chain of Responsibility for handling events

`http://c2.com/cgi/wiki?ModelViewControllerAsAnAggregateDesignPattern`

Observer design pattern

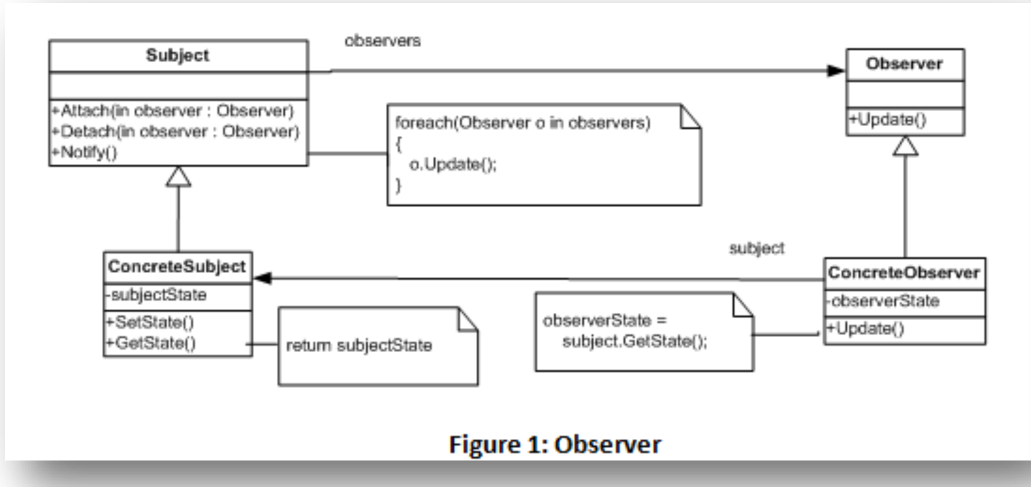
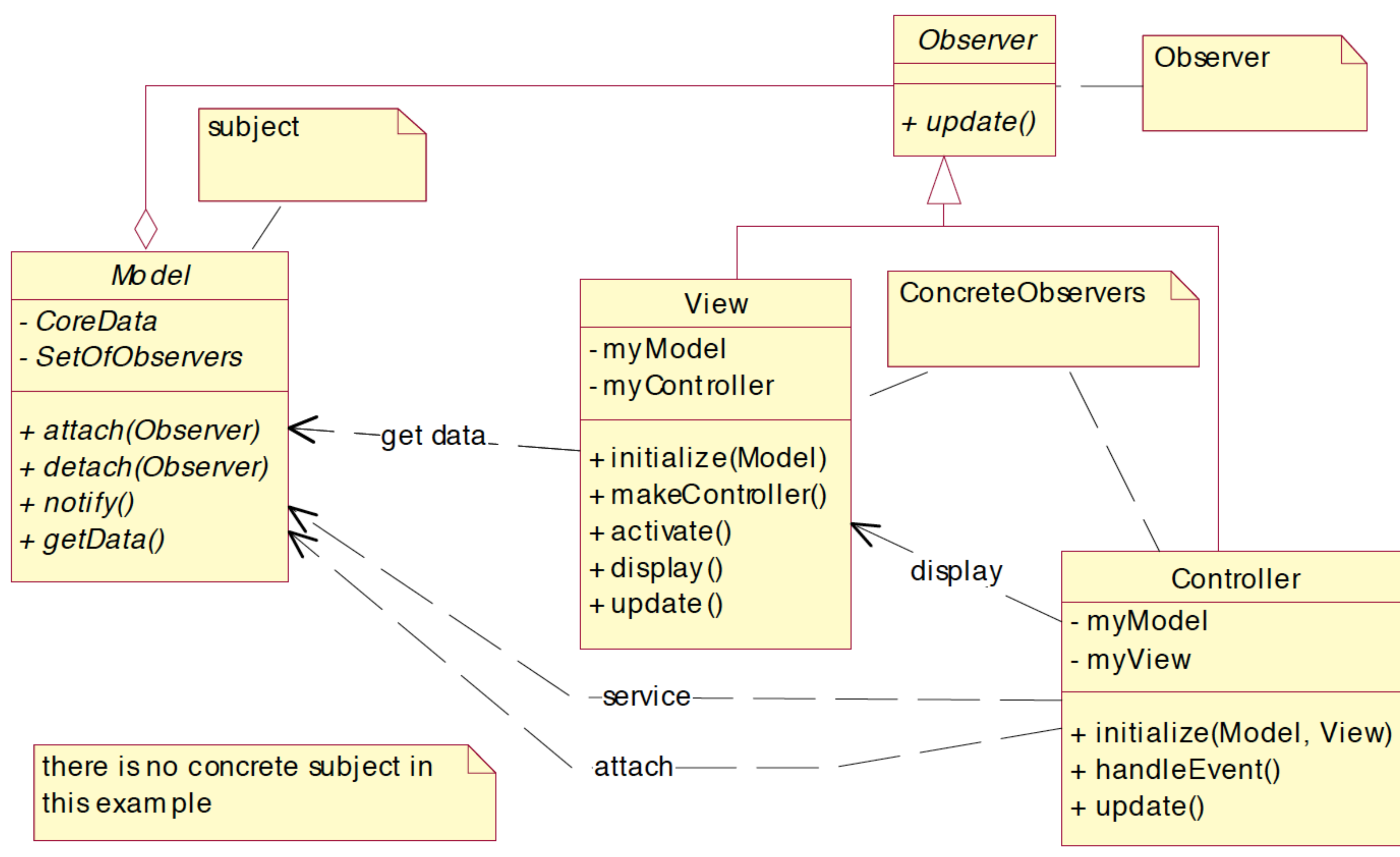


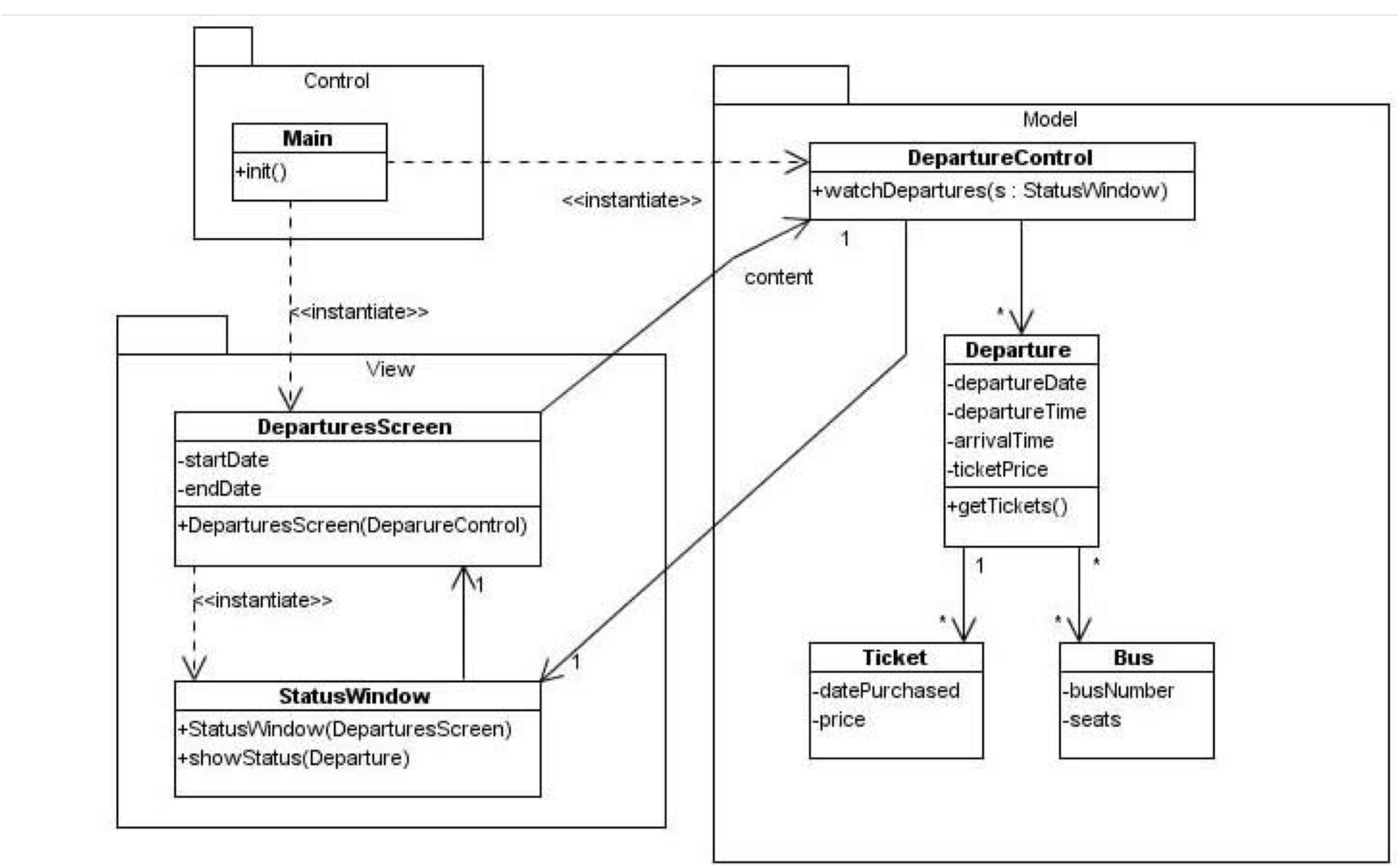
Figure 1: Observer

```
public abstract class Subject
{
    private readonly ICollection<Observer> Observers =
            new Collection<Observer>();

    public void Attach(Observer observer)
    {
        Observers.Add(observer);
    }
    public void Detach(Observer observer)
    {
        Observers.Remove(observer);
    }
    public void Notify()
    {
        foreach (Observer o in Observers)
        {
            o.Update();
        }
    }
}

public class ConcreteSubject : Subject
{
    public object SubjectState { get; set; }
}

public abstract class Observer
{
    public abstract void Update();
}

public class ConcreteObserver : Observer
{
    private object ObserverState;
    private ConcreteSubject Subject { get; set; }

    public ConcreteObserver(ConcreteSubject subject)
    {
        Subject = subject;
    }
    public override void Update()
    {
        ObserverState = Subject.SubjectState;
    }
}
```

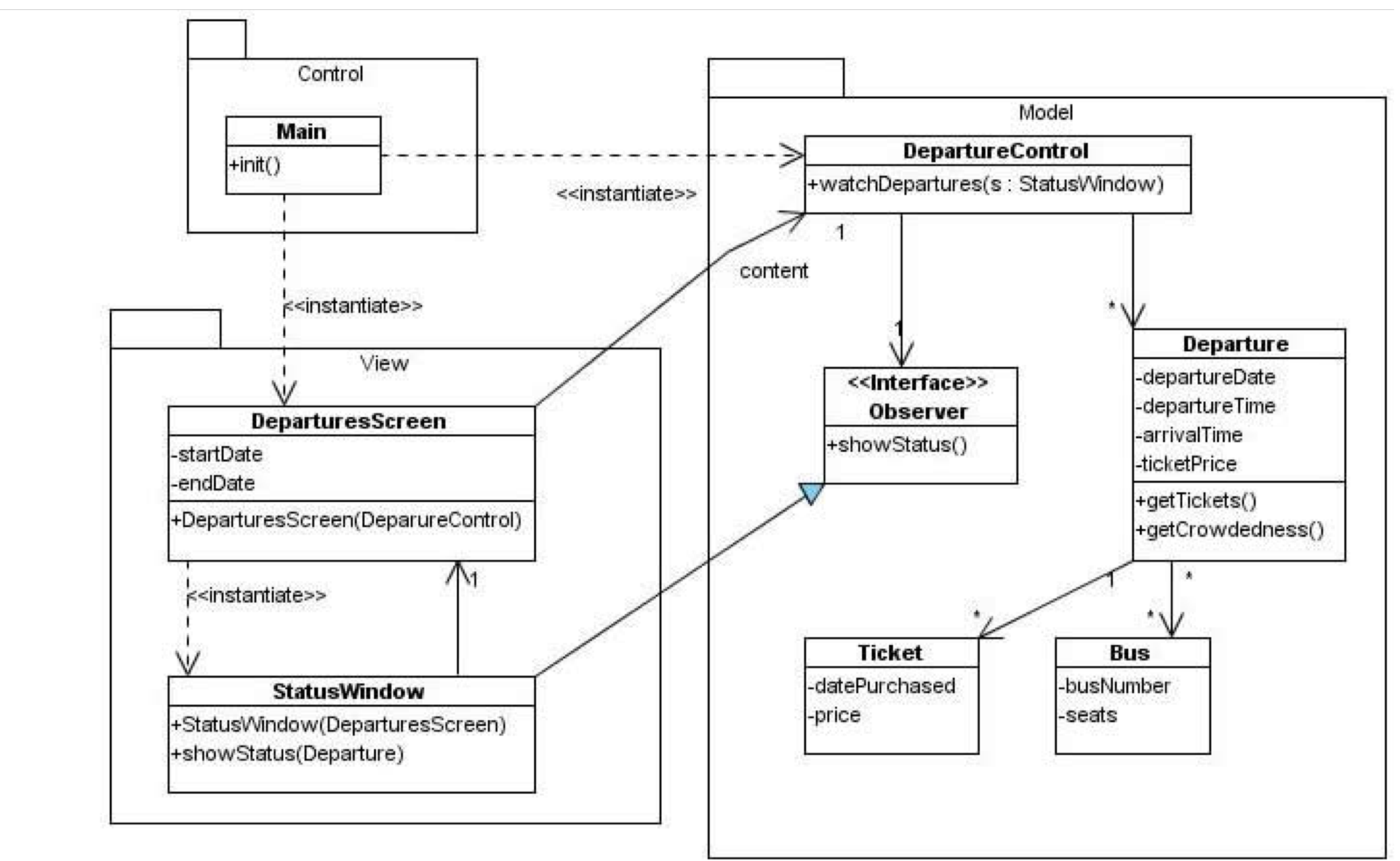
MVC based on Observer dp



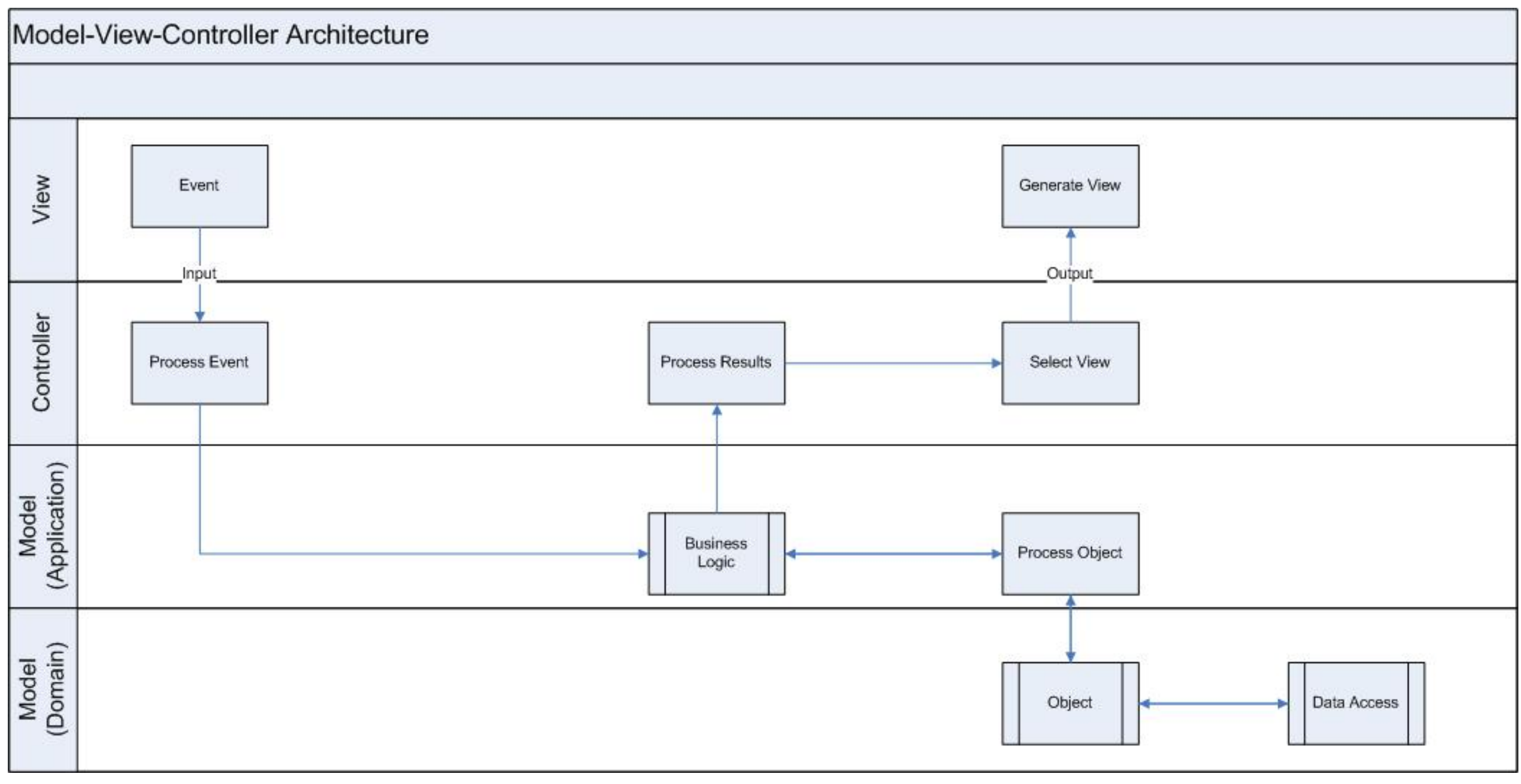
Example 2: a harmful dependency



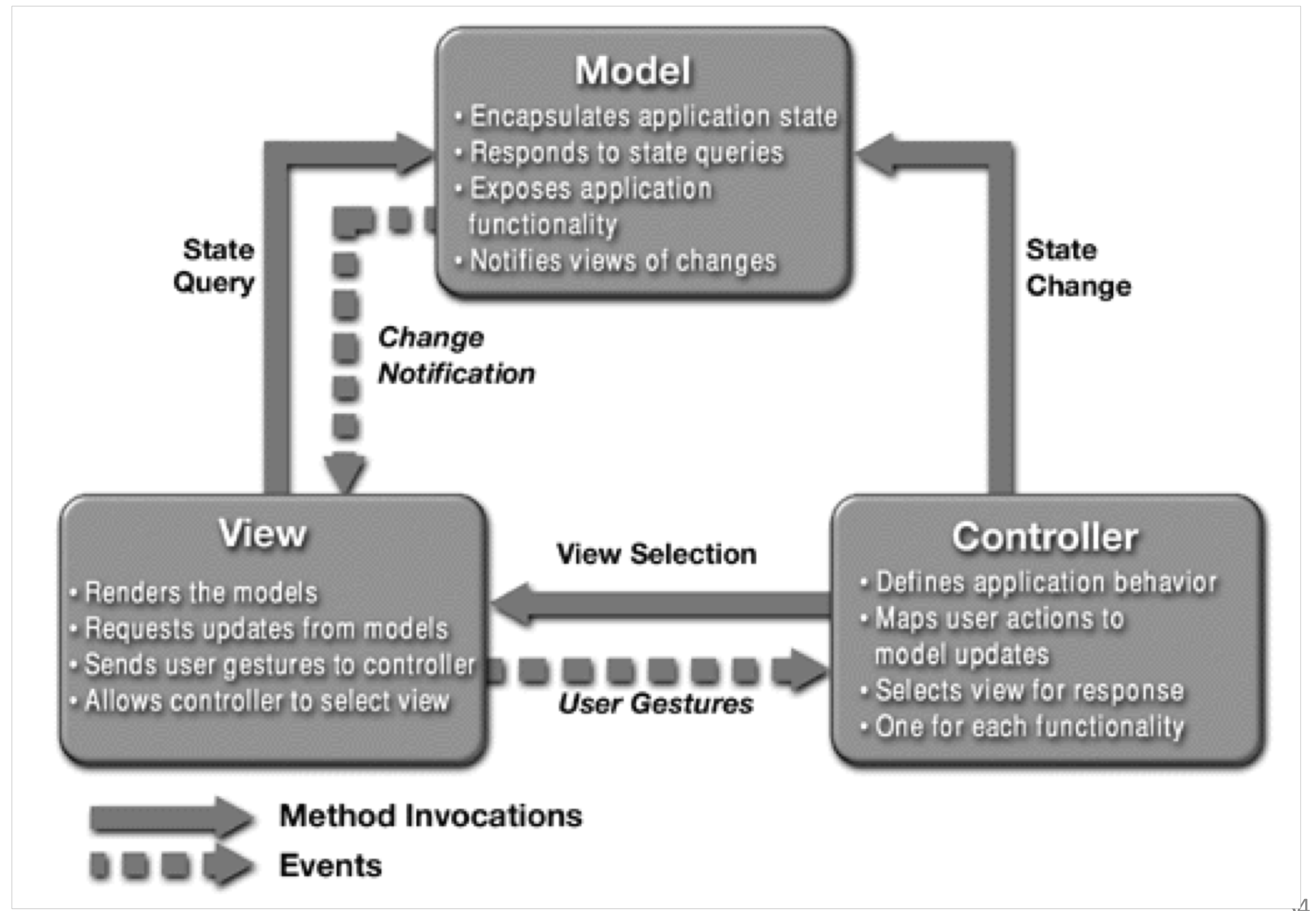
Example 2: deleting a harmful dependency



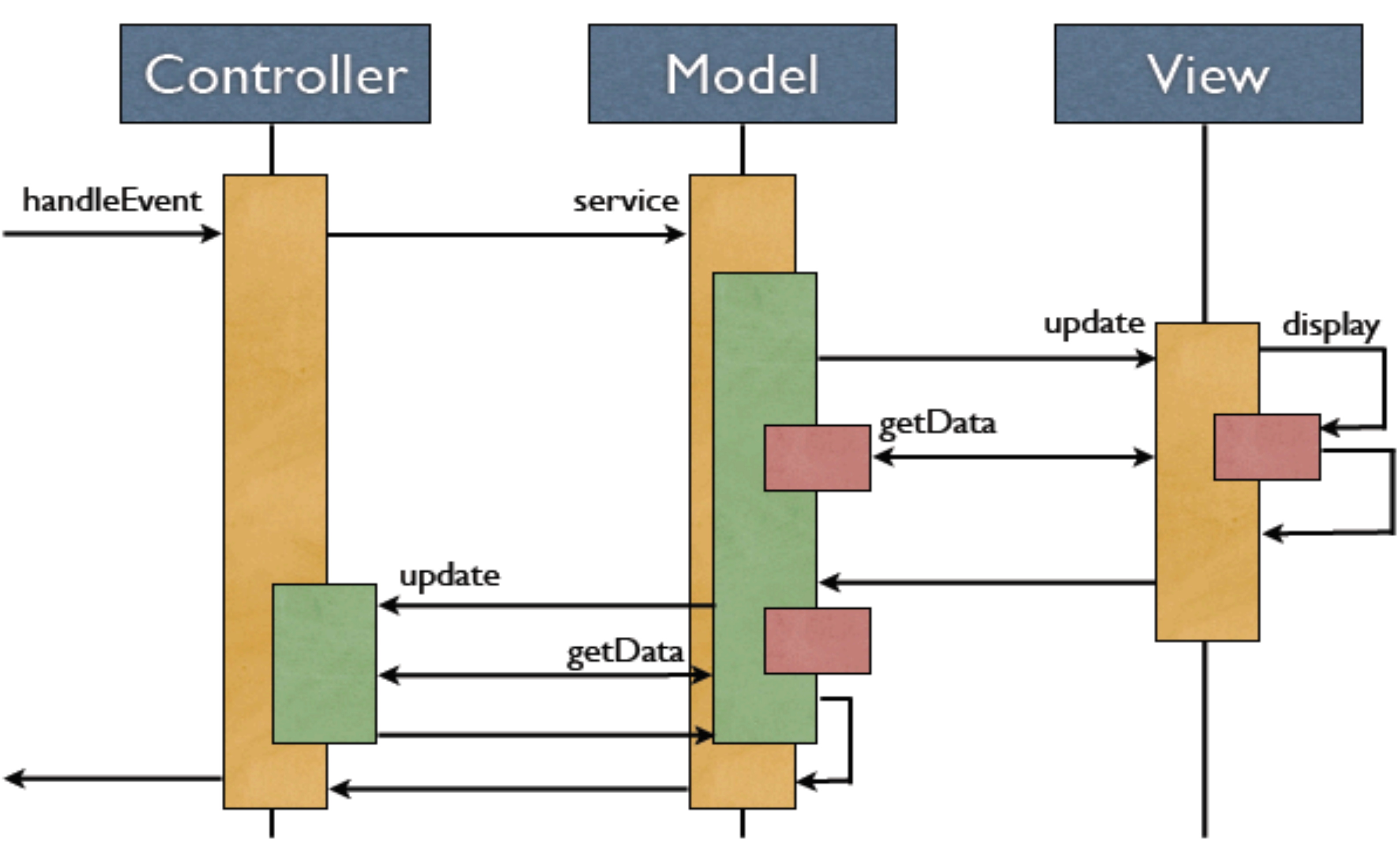
MVC behavior



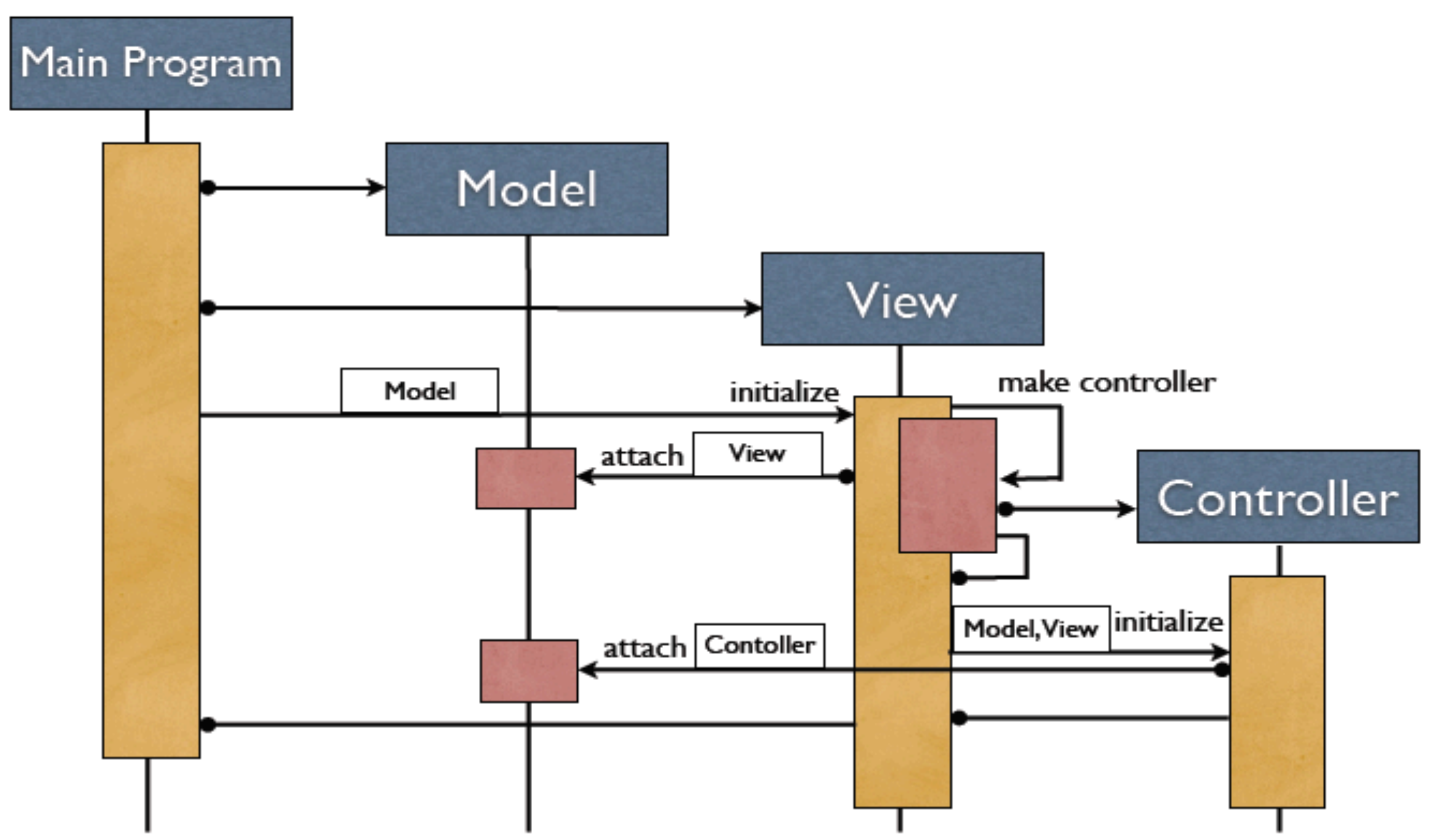
MVC interactions



Basic interaction for MVC



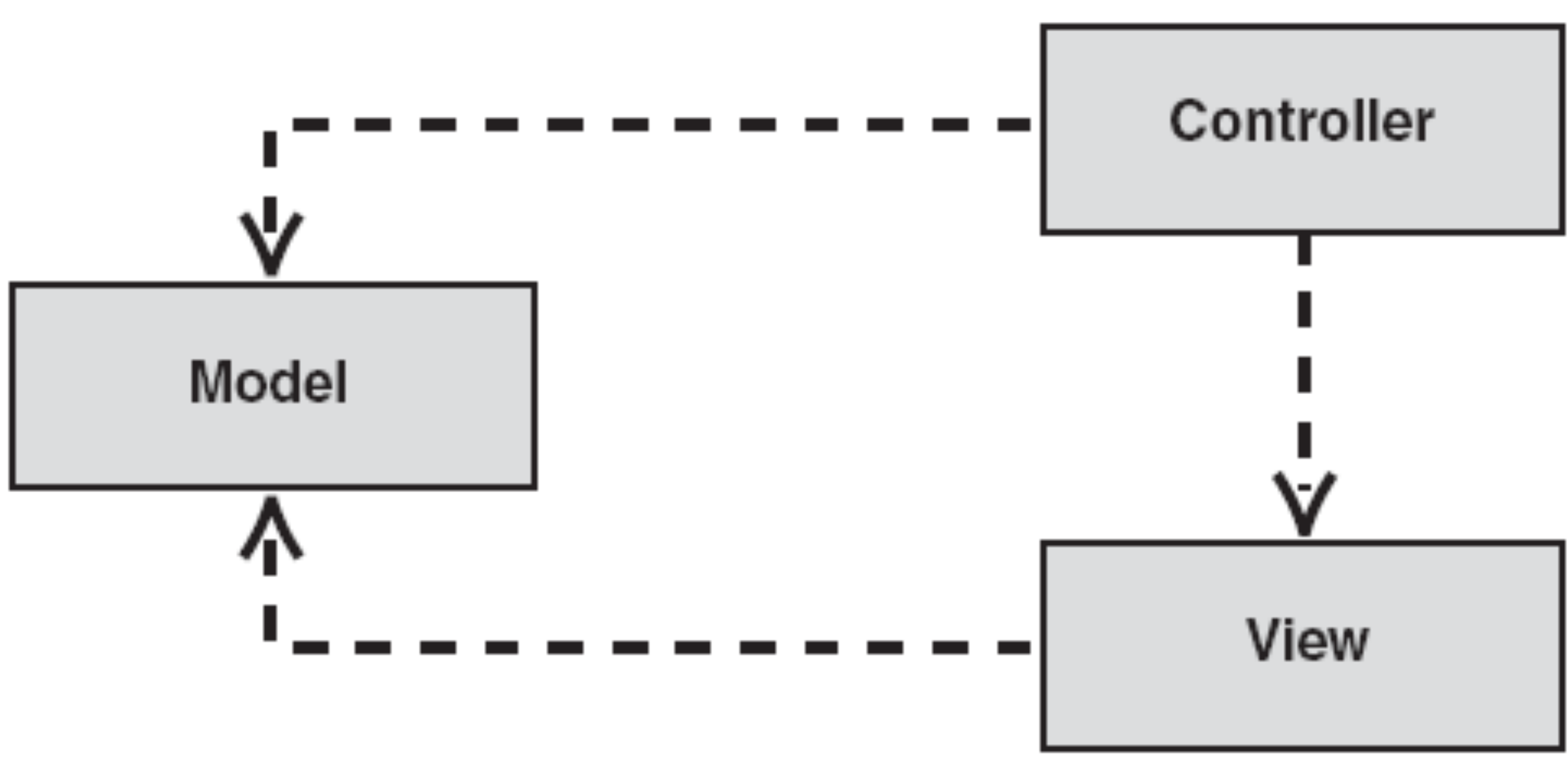
Initialization for MVC



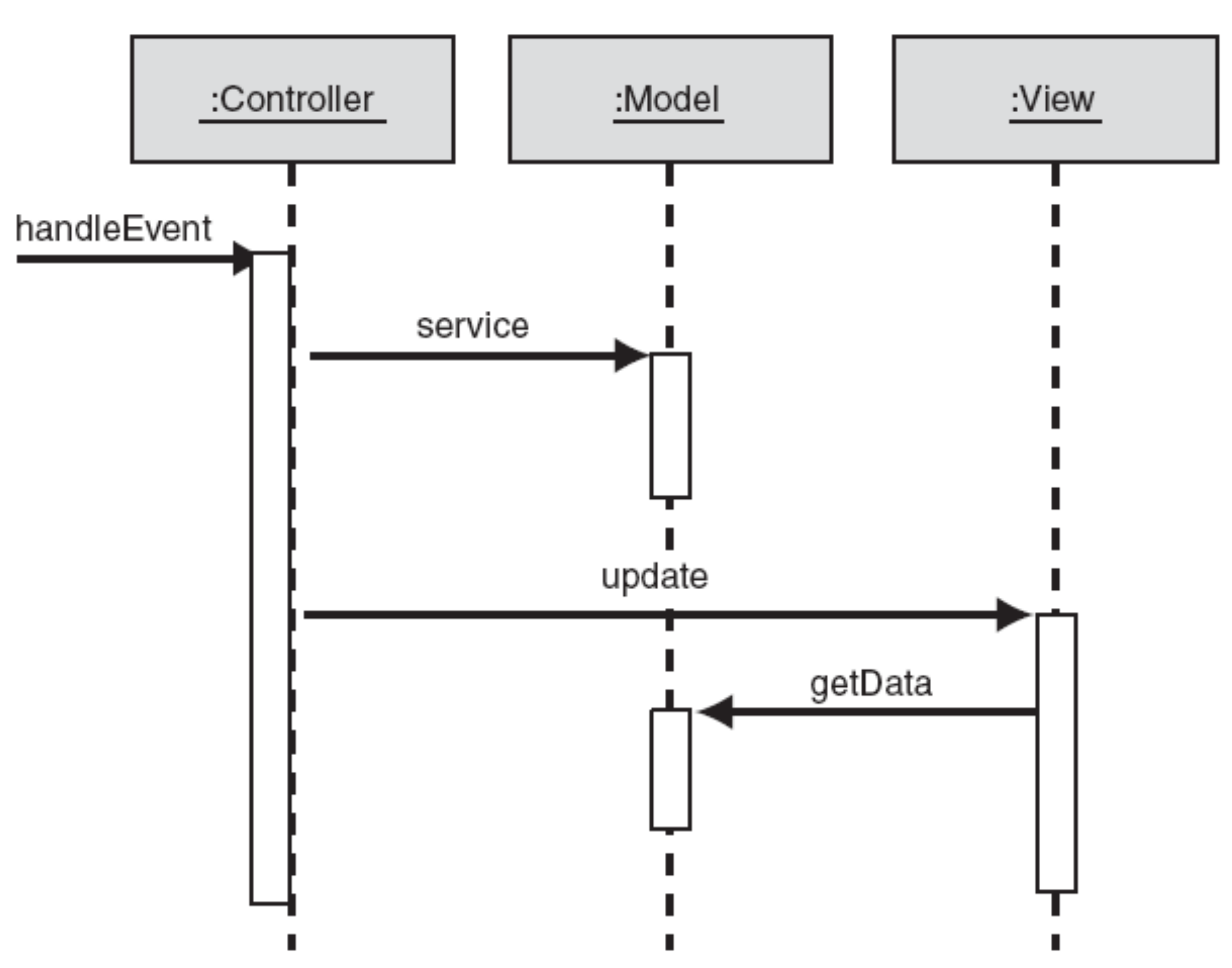
Design of the Presentation Layer

- Passive Model
  - Model does not report state changes
- Active Model
  - Model reports state changes to view(s)
    - Observer pattern (Publish/Subscribe)

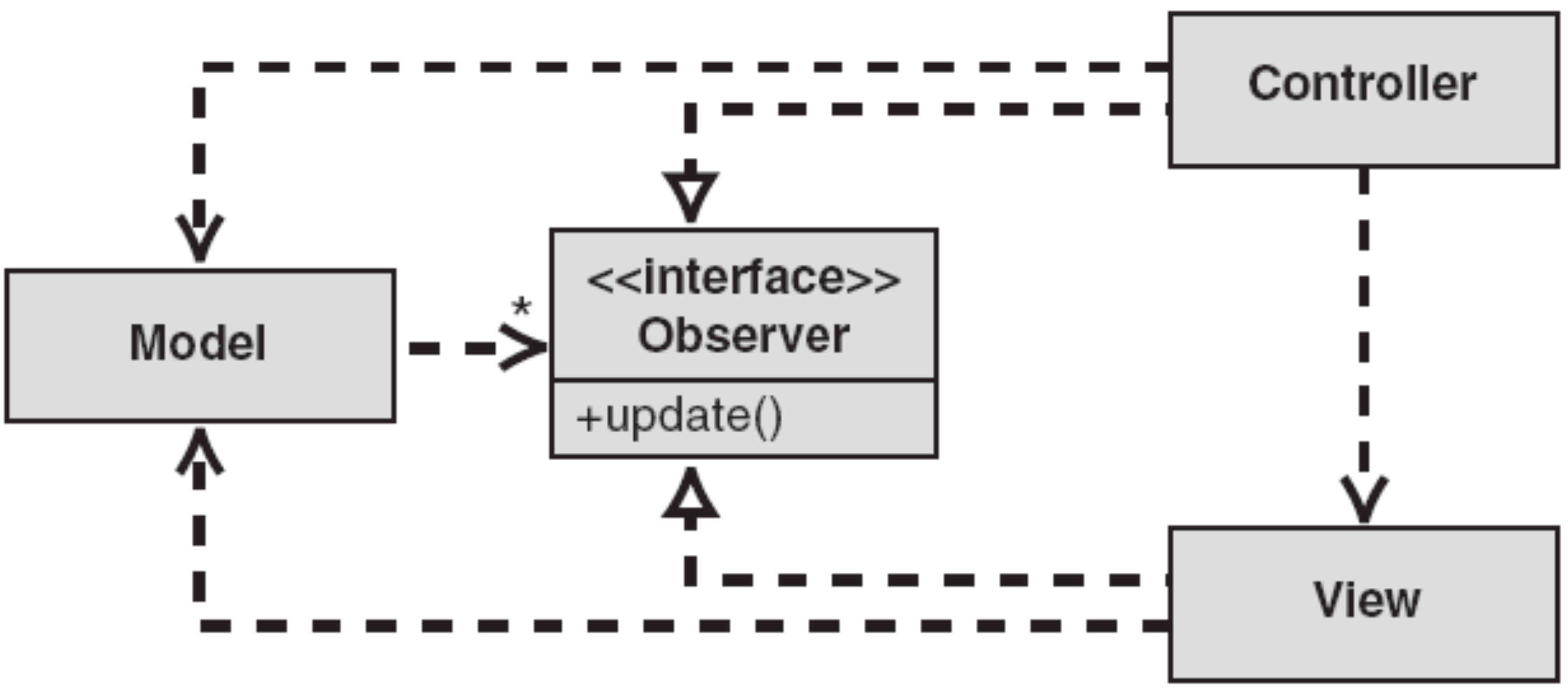
MVC passive



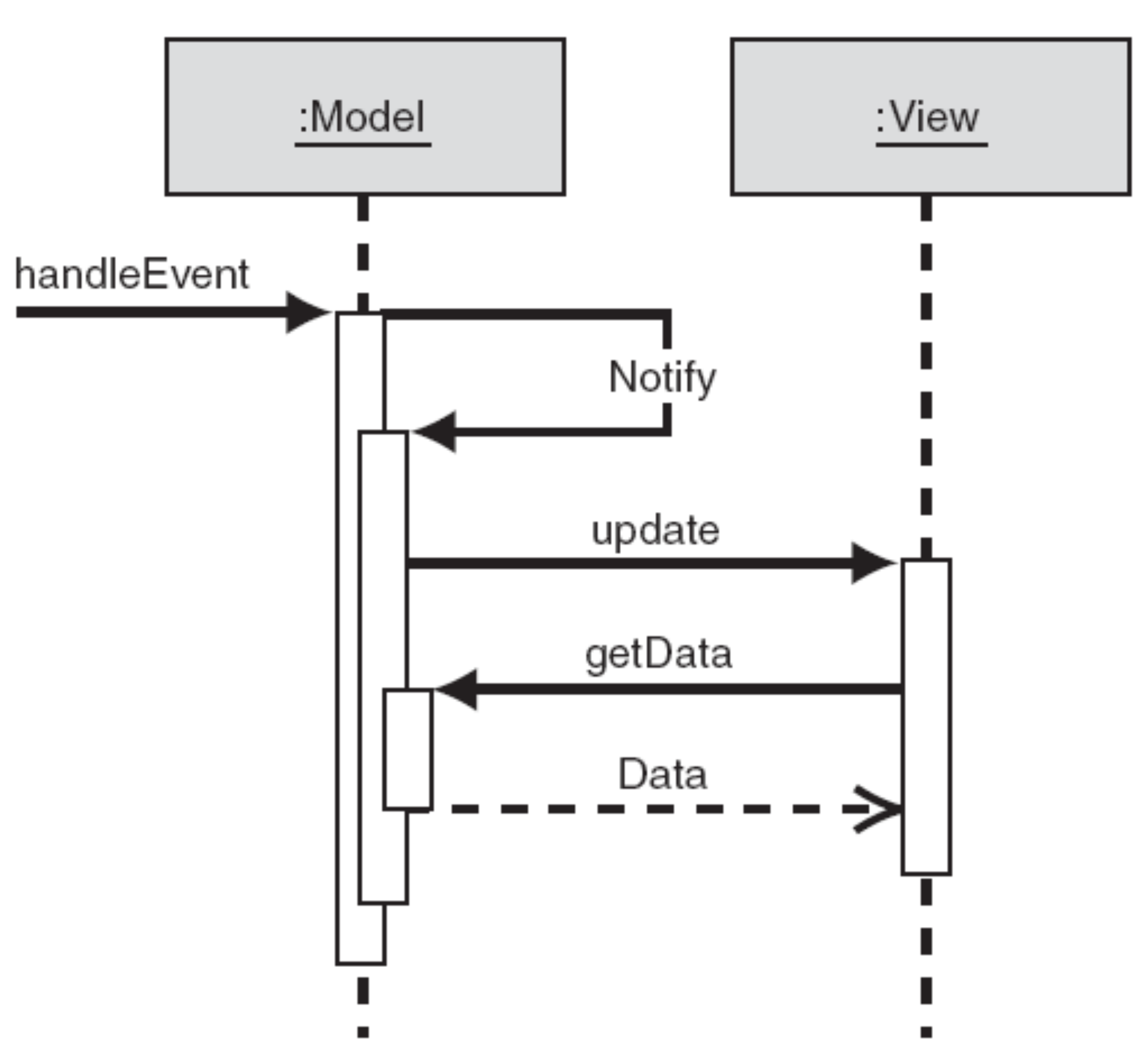
MVC passive



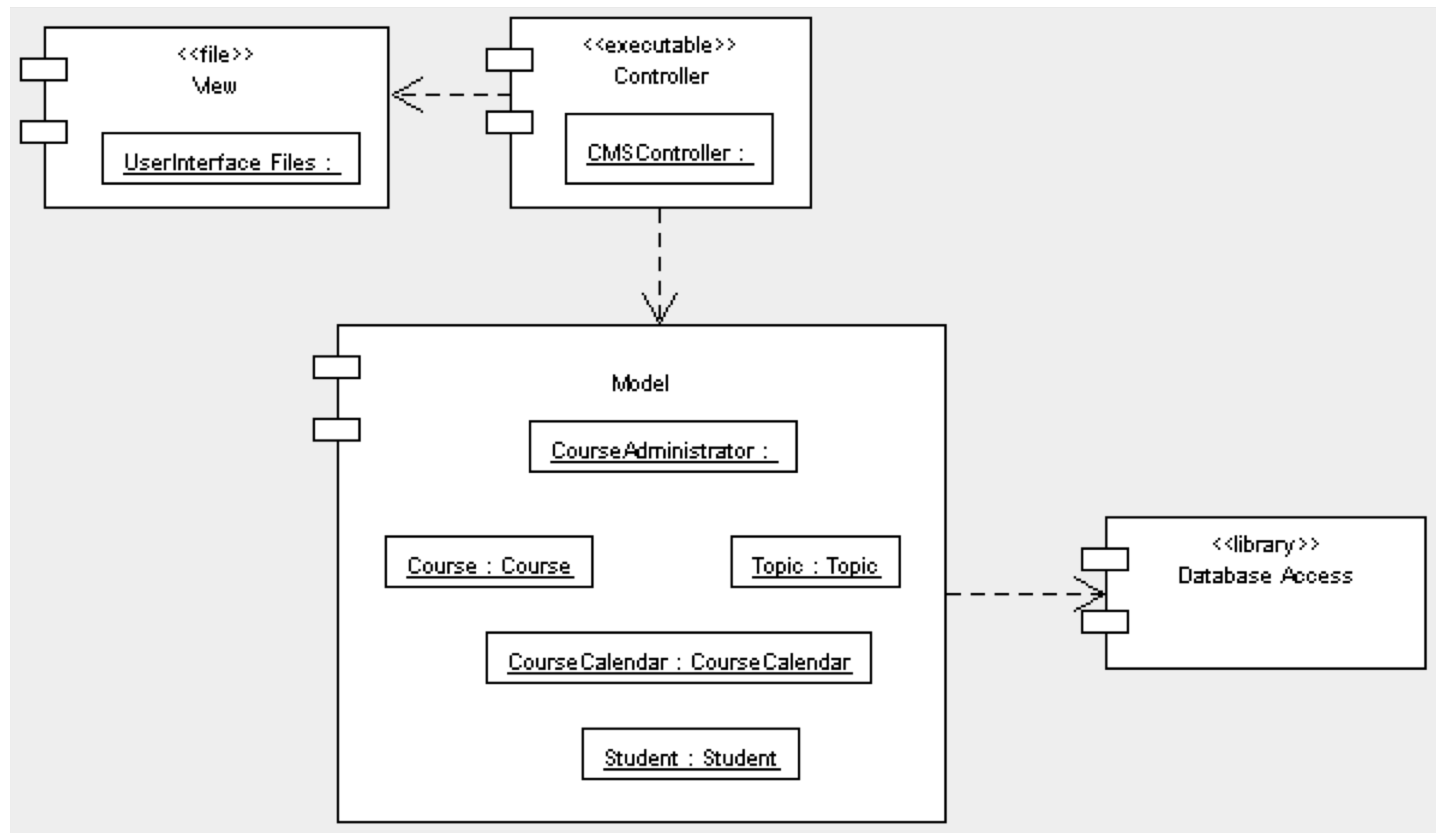
MVC active



MVC active



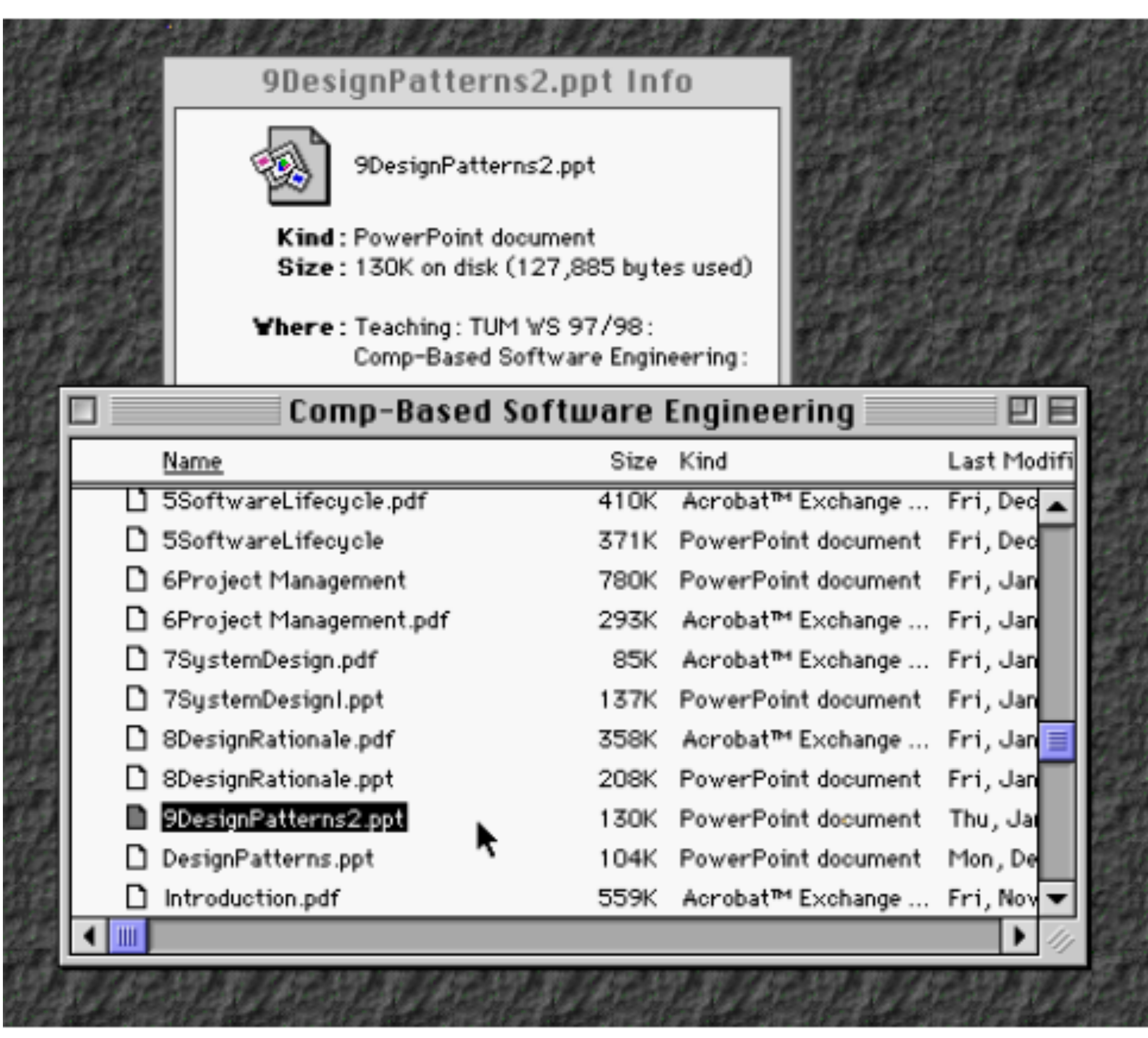
MVC: component diagram example



MVC Architectures

- The Model should not depend on any View or Controller
- Changes to the Model state are communicated to the View subsystems by means of a “subscribe/notify” protocol (eg. Observer pattern)
- MVC is a combination of the repository and three-tiers architectures:
  - The Model implements a centralized data structure;
  - The Controller manages the control flow: it receives inputs from users and forwards messages to the Model
  - The Viewer pictures the model

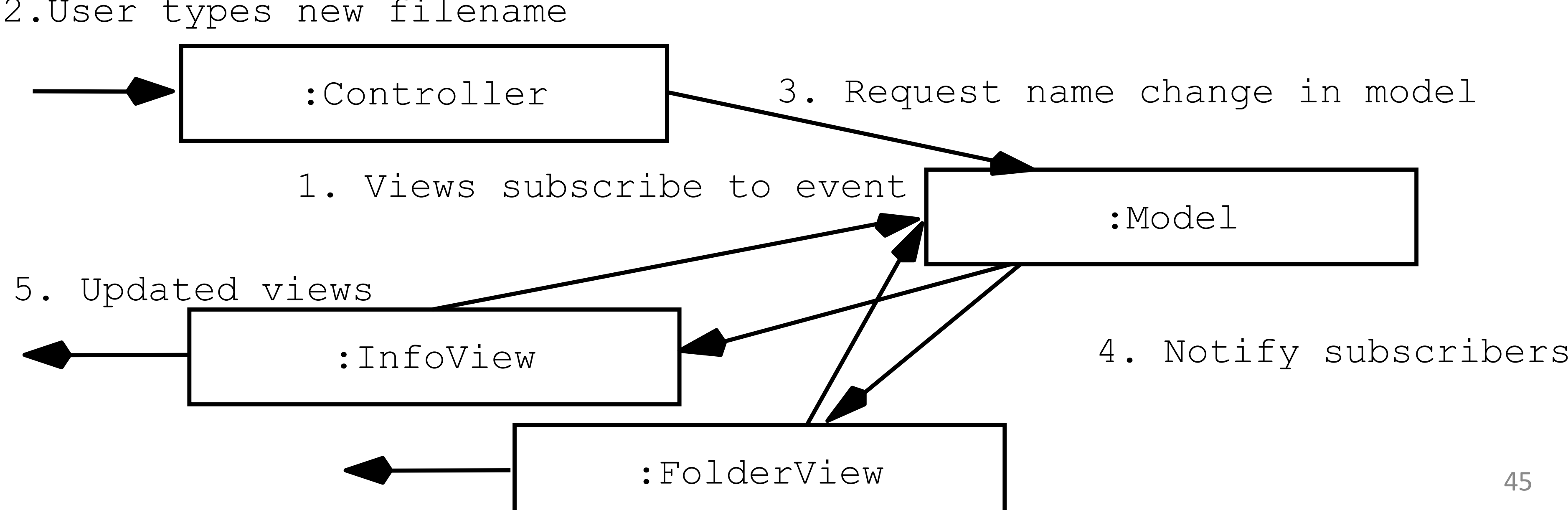
MVC: an example



- Two different views of a file system:
- The bottom window visualizes a folder named Comp-Based Software Engineering
- The up window visualizes file info related to file named 90DesignPatterns2.ppt
- The file name is visualized into three different places

MVC: communication diagram

1. Both InfoView and FolderView subscribe the changes to the model File when created
2. The user enters the new filename
3. The Controller forwards the request to the model
4. The model actually changes the filename and notifies the operation to subscribers
5. InfoView and FolderView are updated so that the user perceives the changes in a consistent way



The benefits of the MVC style

- The main reason behind the separation (Model, View, and Controller) is that the user interface (Views) changes much more frequently than the knowledge about the application domain (Model)
- The model and the views can become very reusable
- This style is easy to reuse, to maintain, and to extend
- The MVC style is especially effective in the case of interactive systems, when multiple synchronous views of the same model have to be provided

MVC benefits

- The model is strictly separated from the UI
- Changes to the data in the underlying model are reflected in all the views automatically
- The UI (Views and controller) can be changed without changing the data model
- The views do not interact
- MVC architectures can be used as extensible frameworks, simplifying maintenance and evolution
- MVC is quite portable

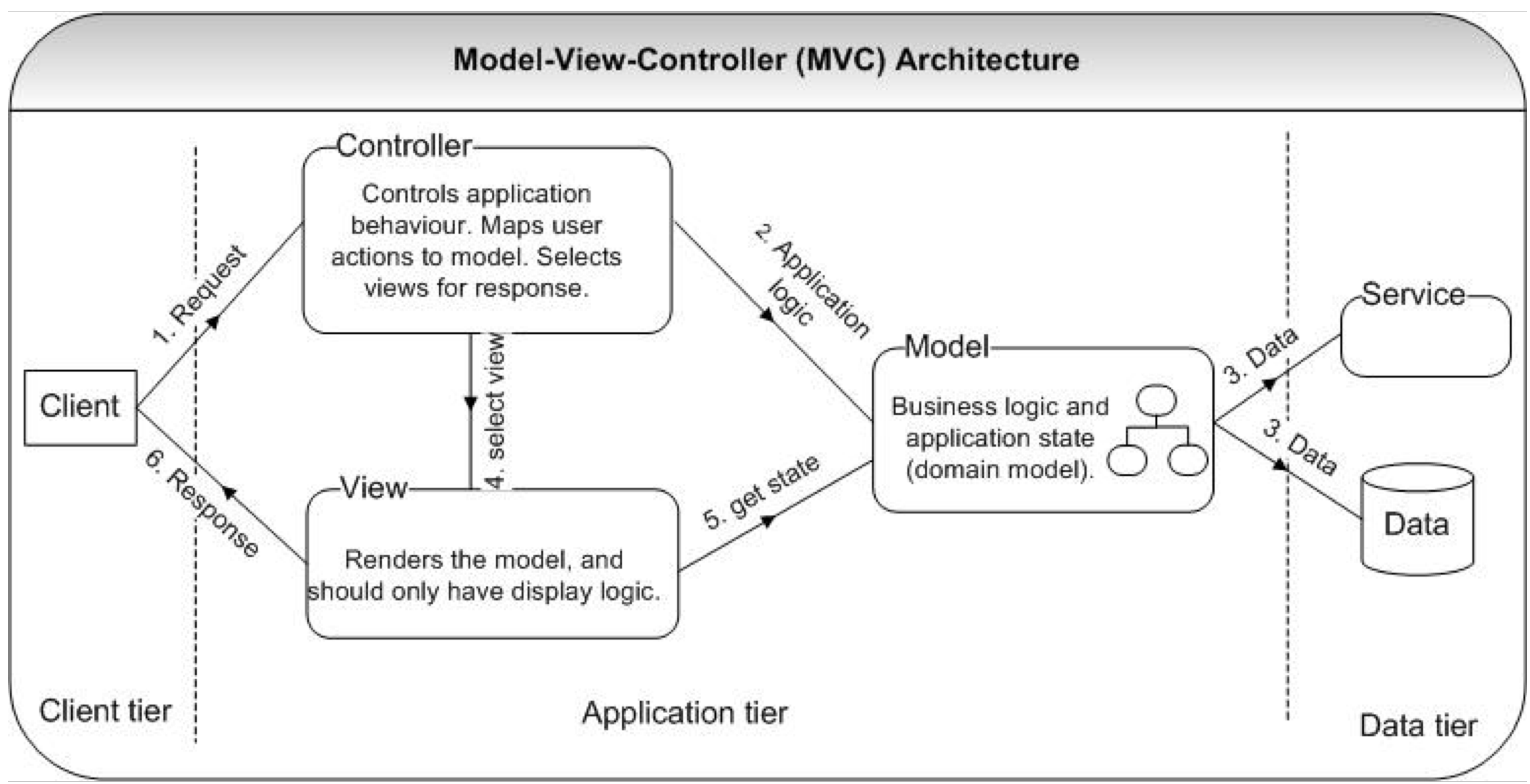
MVC weaknesses

- Complexity is increased
- Changes to the Model must be published to all the Views which subscribed to them
- Each pair Controller/View has strong interdependencies
- Controllers have to know well the Model
- Inefficient data access can result because the separation of Views and Model data

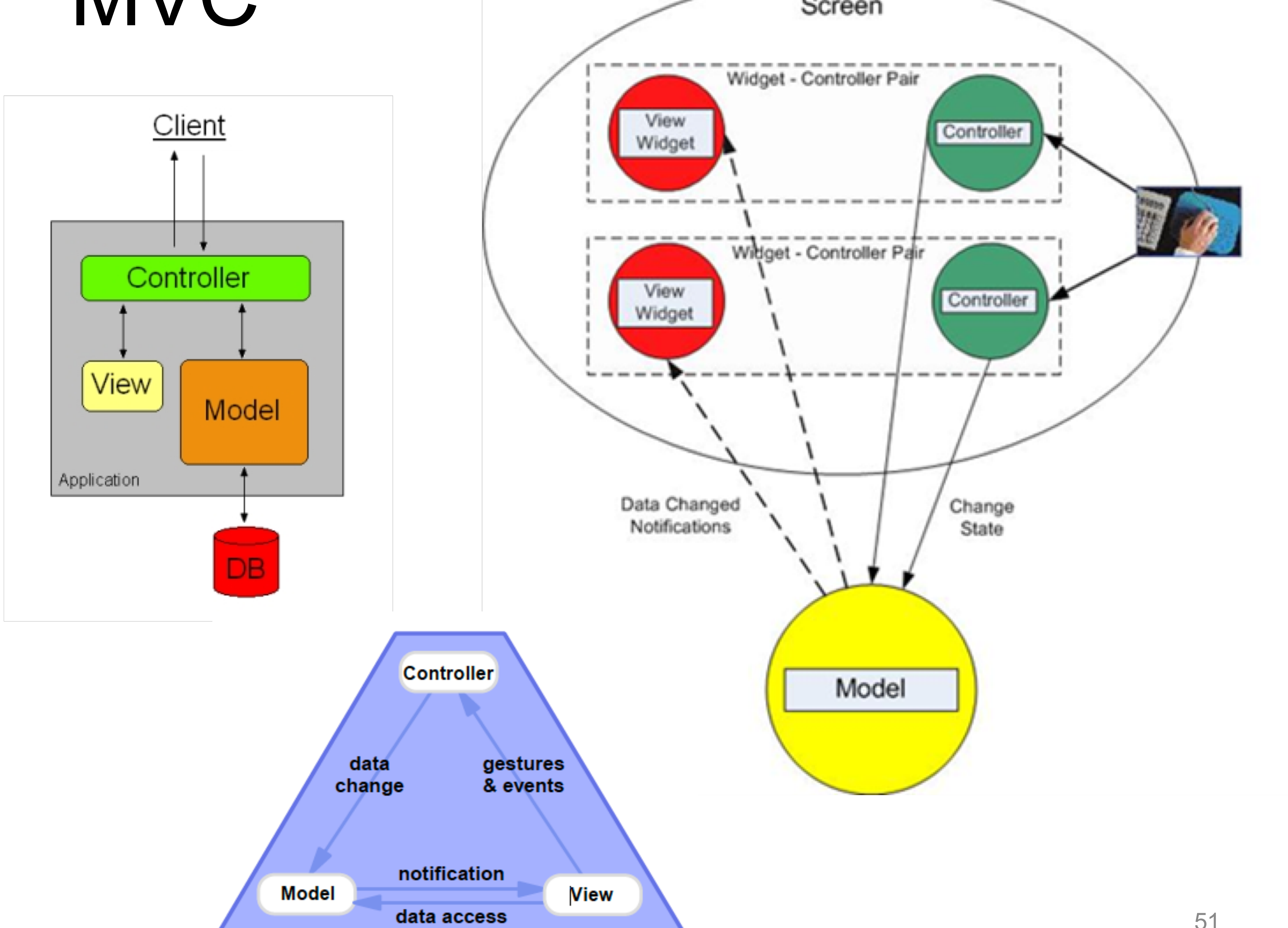
ClientServer 3-tiers vs MVC

- The CS 3-tiers style may look similar to the MVC style; however, they are different
- A fundamental rule in a CS 3-tiers architecture is: the client tier **never** communicates directly with the data tier; all communication must go through the middleware tier: the CS 3-tiers style is linear
- Instead, the MVC style is triangular: the View sends updates to the Controller, the Controller updates the Model, and the Model updates directly the Views

MVC on CS 3-tiers



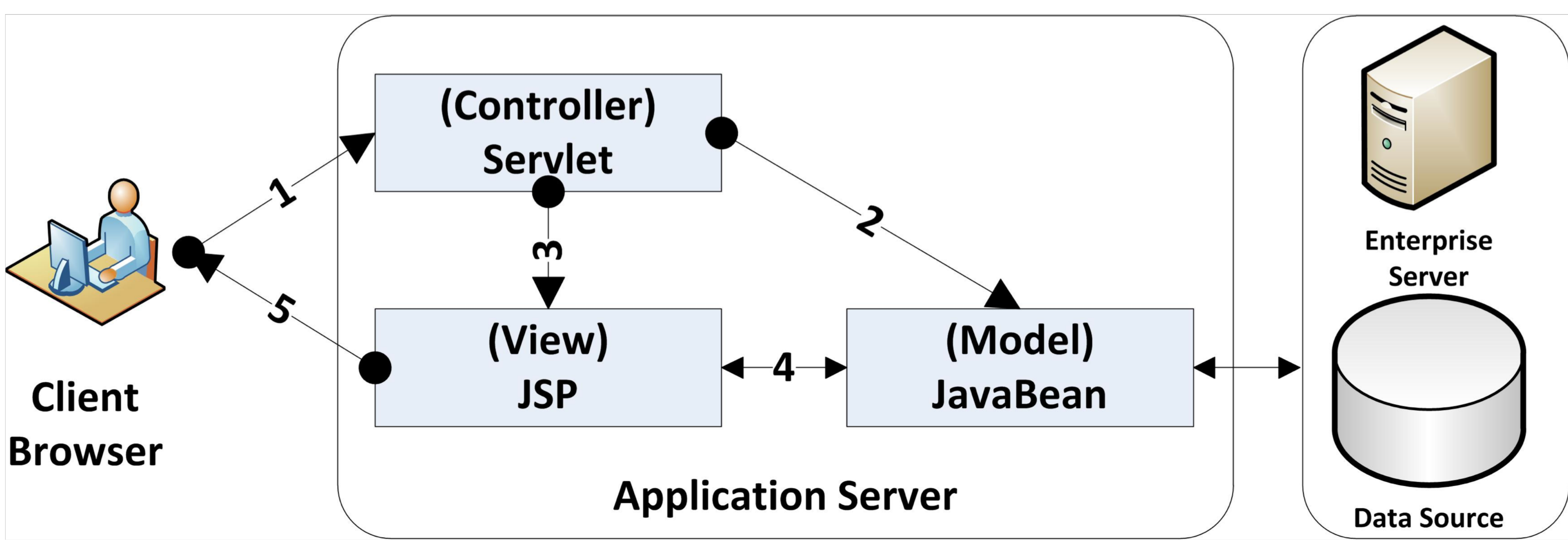
MVC



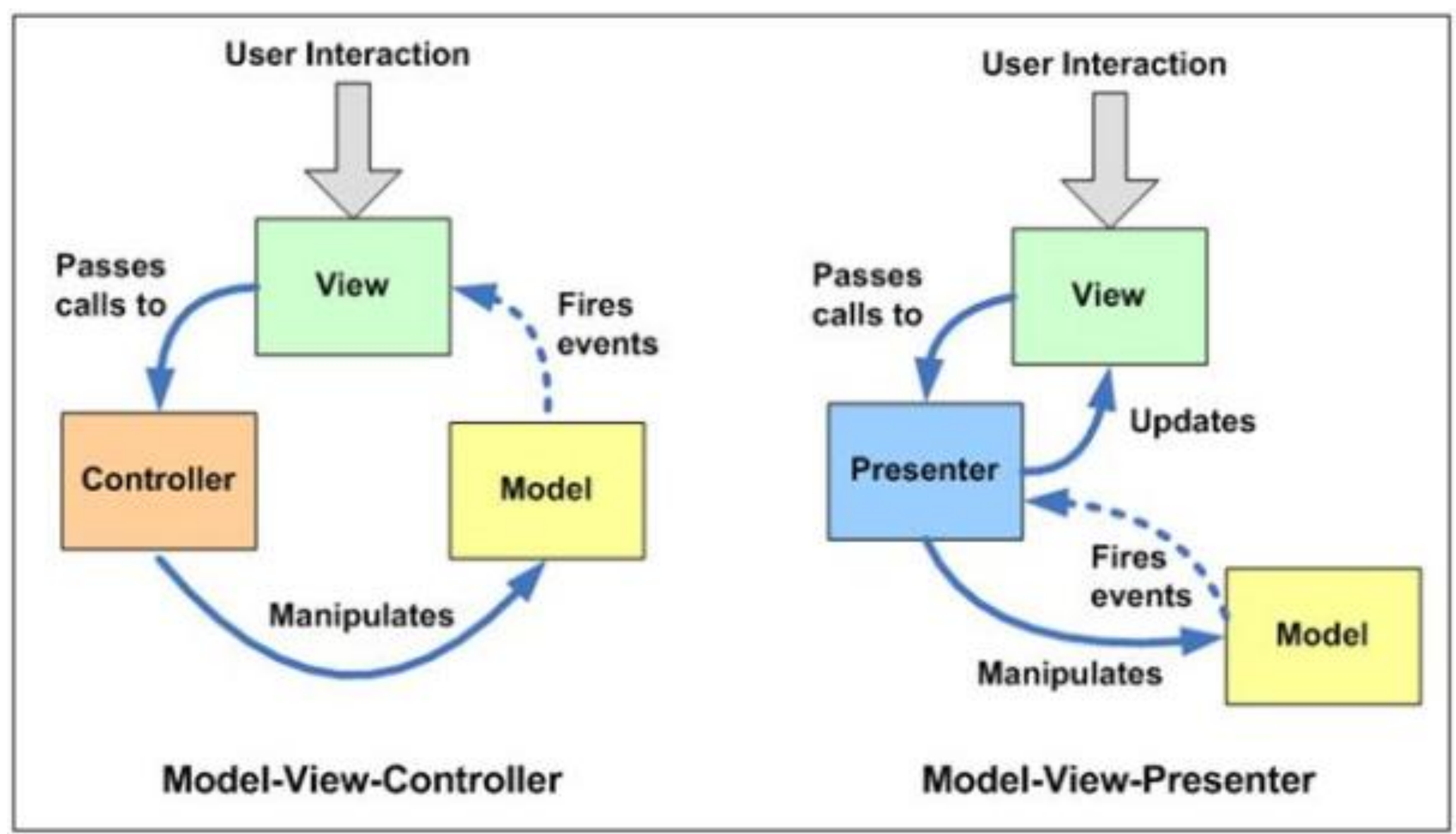
Java interaction frameworks

- Most MVC frameworks follow a push-based architecture also called "action-based". They use actions that do the required processing, and then "push" the data to the view layer to render the results
- An alternative is a pull-based architecture, also called "component-based". These frameworks start with the view layer, which can then "pull" results from multiple controllers as needed. Multiple controllers can be involved with a single view
  - Action-based frameworks: Apache Struts, Spring
  - Component-based frameworks: Apache Click, Apache Tapestry, Apache Wicket, Java Server Faces

Struts2

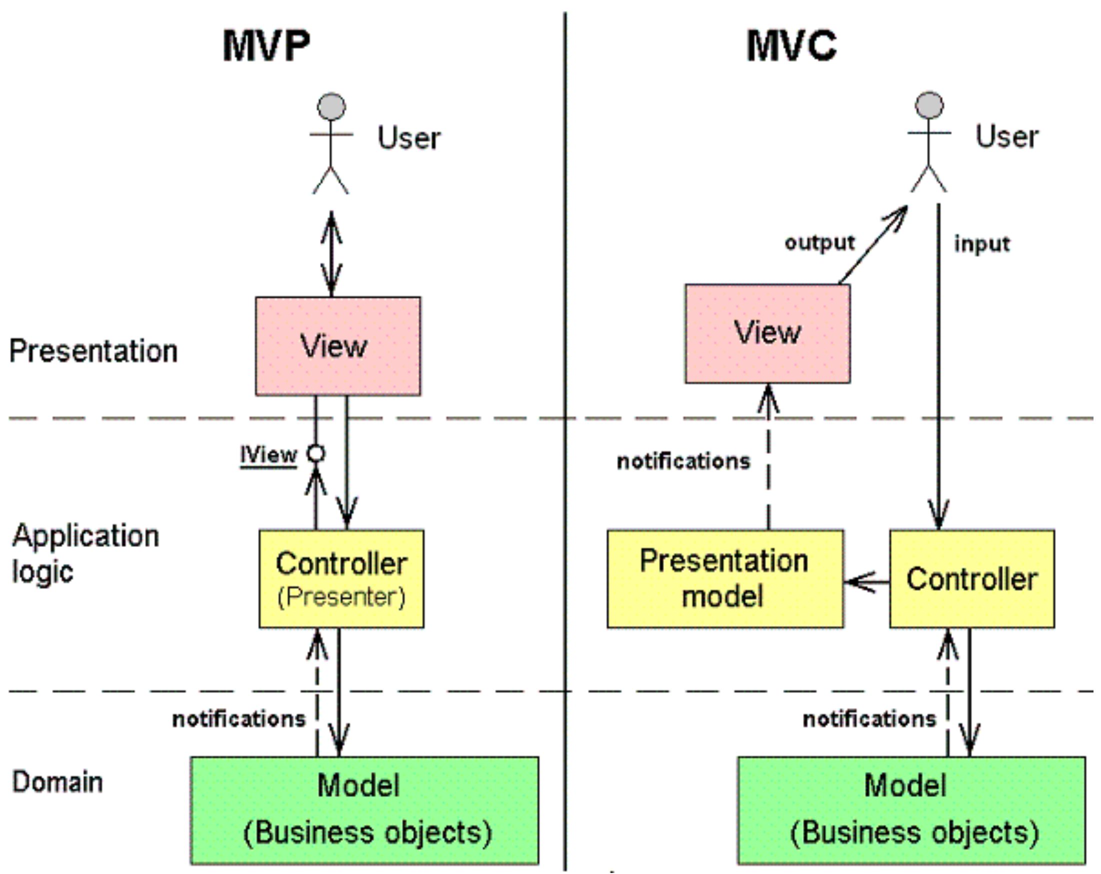


Alternatives to MVC: MVP

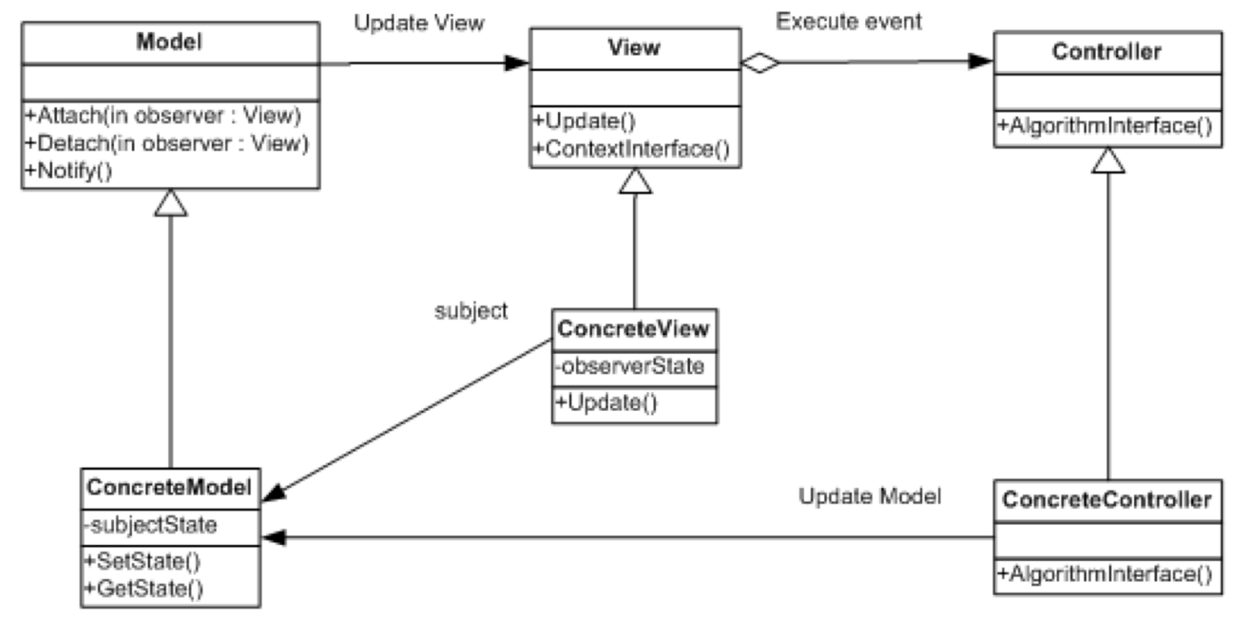


The Presenter is responsible for binding the Model to the View.
Easier to unit test because interaction with the View is through an interface
Usually View to Presenter map one to one. Complex views may have multi presenters

MVC vs MVP



www.mvcsharp.org/Overview/Default.aspx



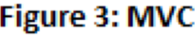


Figure 3: MVC

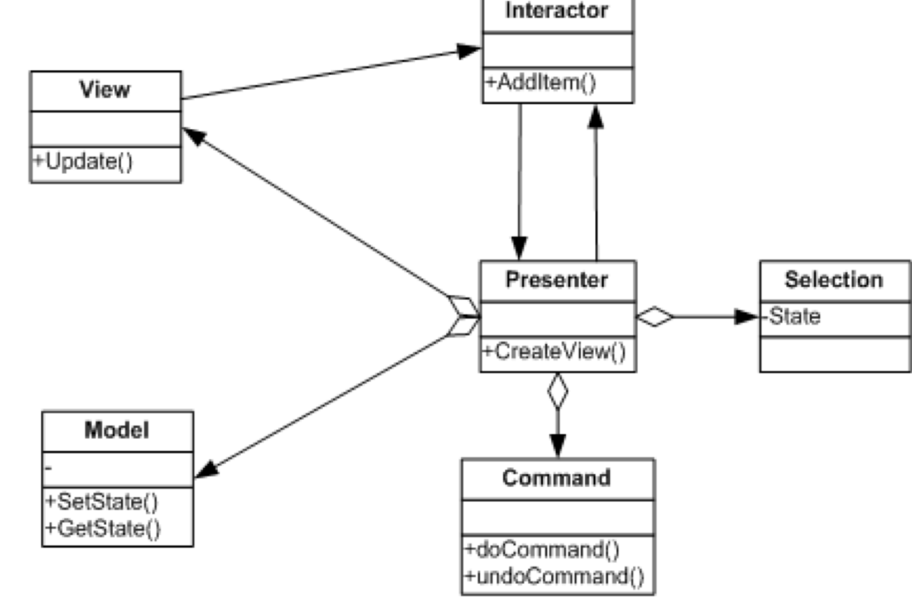
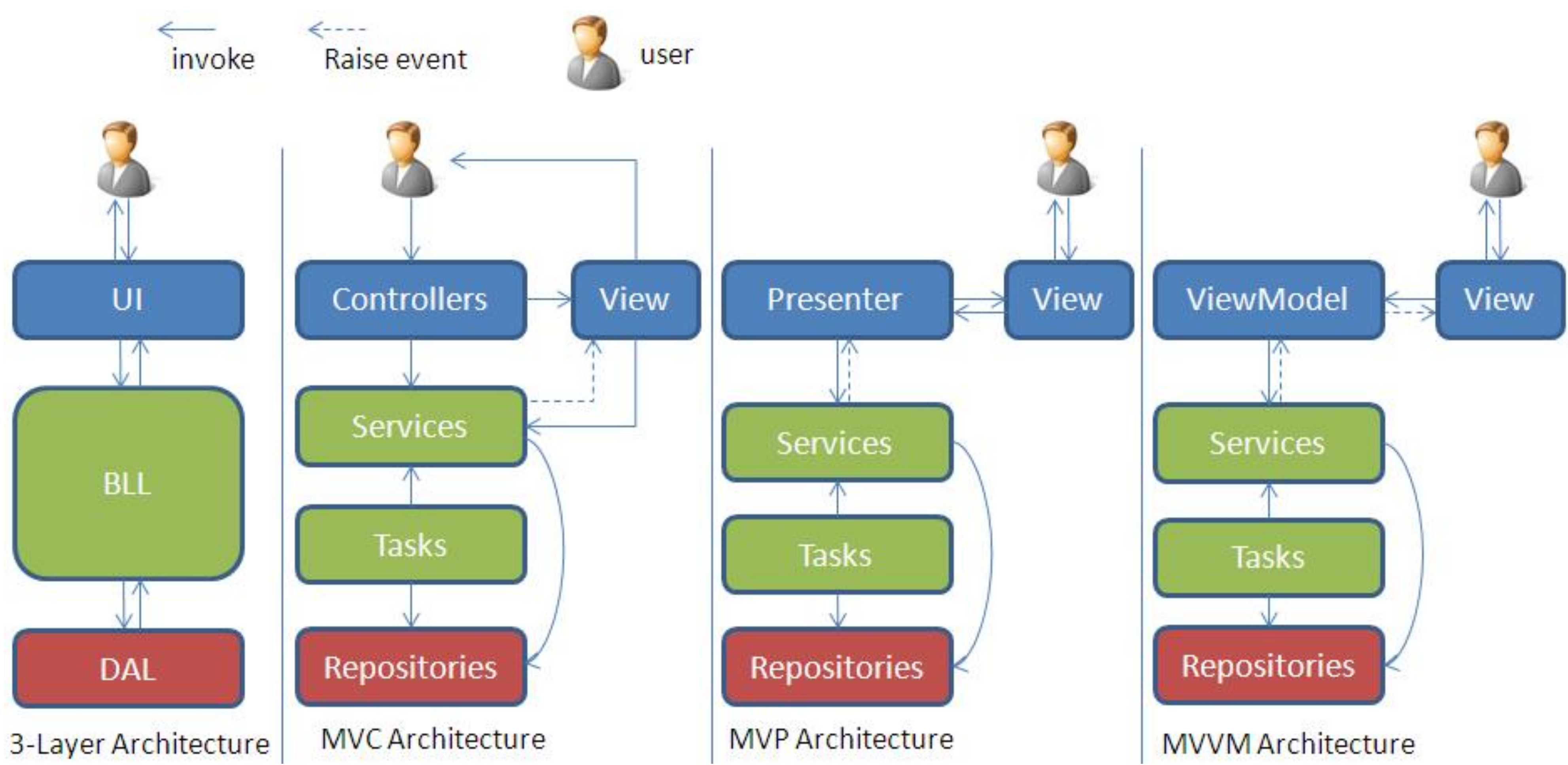


Figure 6: MVP

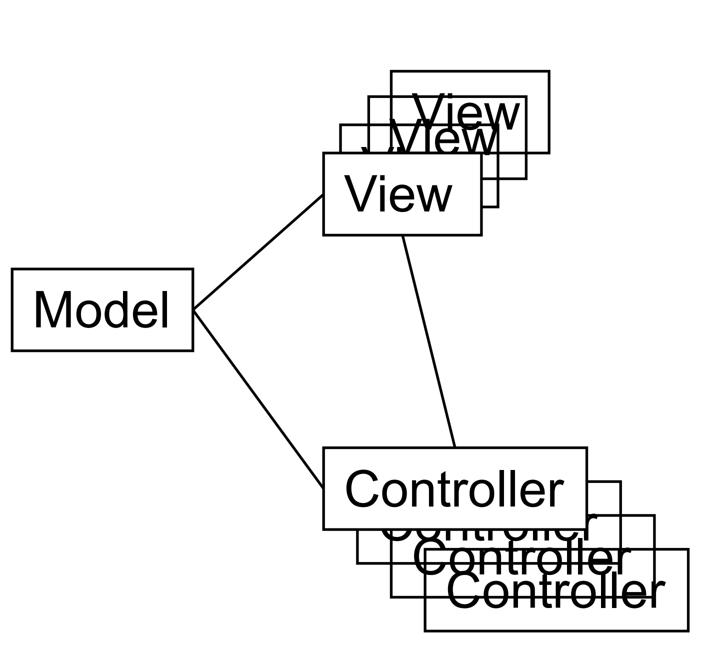
http://www.codeproject.com/Articles/42830/Model-View-Controller-Model-View-Presenter-and-Mod

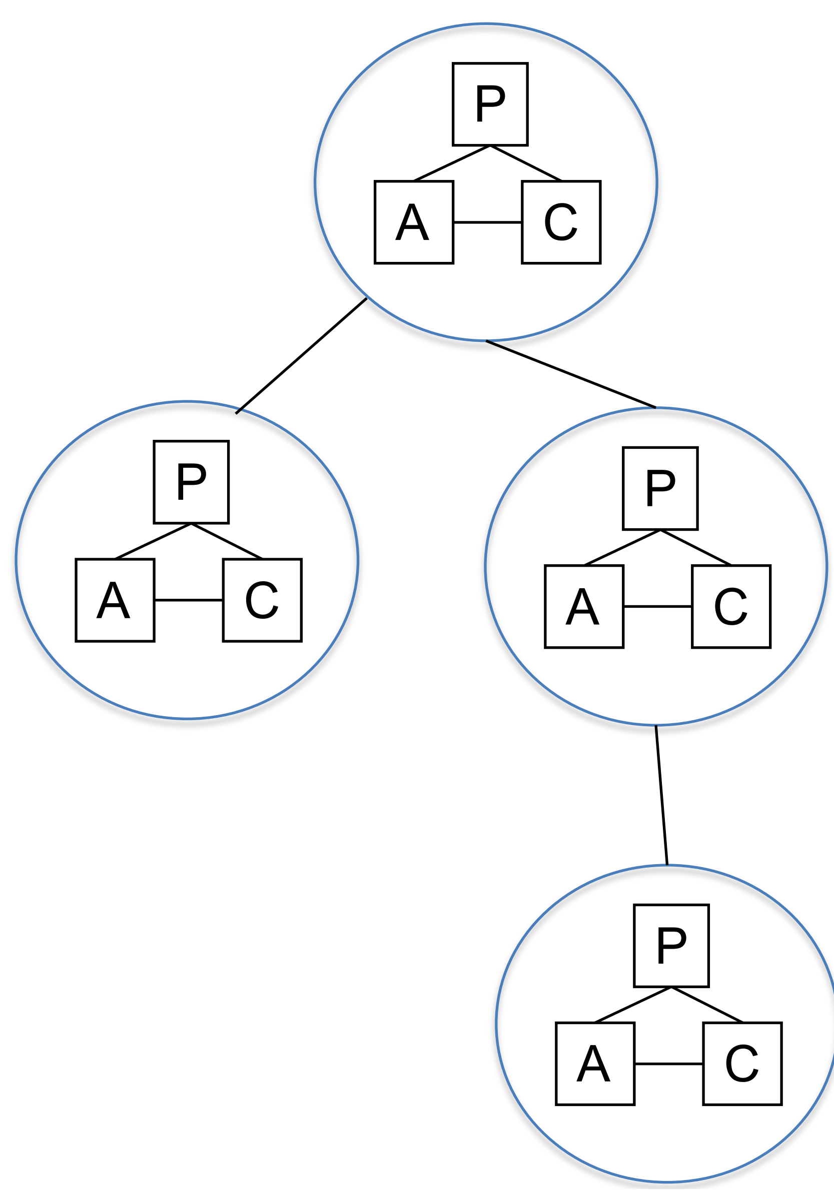
MVP: two flavors

- Passive view: the interaction with the Model is handled only by the Presenter; the View is not aware of changes to the model
- Supervising controller: The interaction with the Model is handled NOT only by the Presenter. The View uses *data binding* to update itself when the Model is updated.

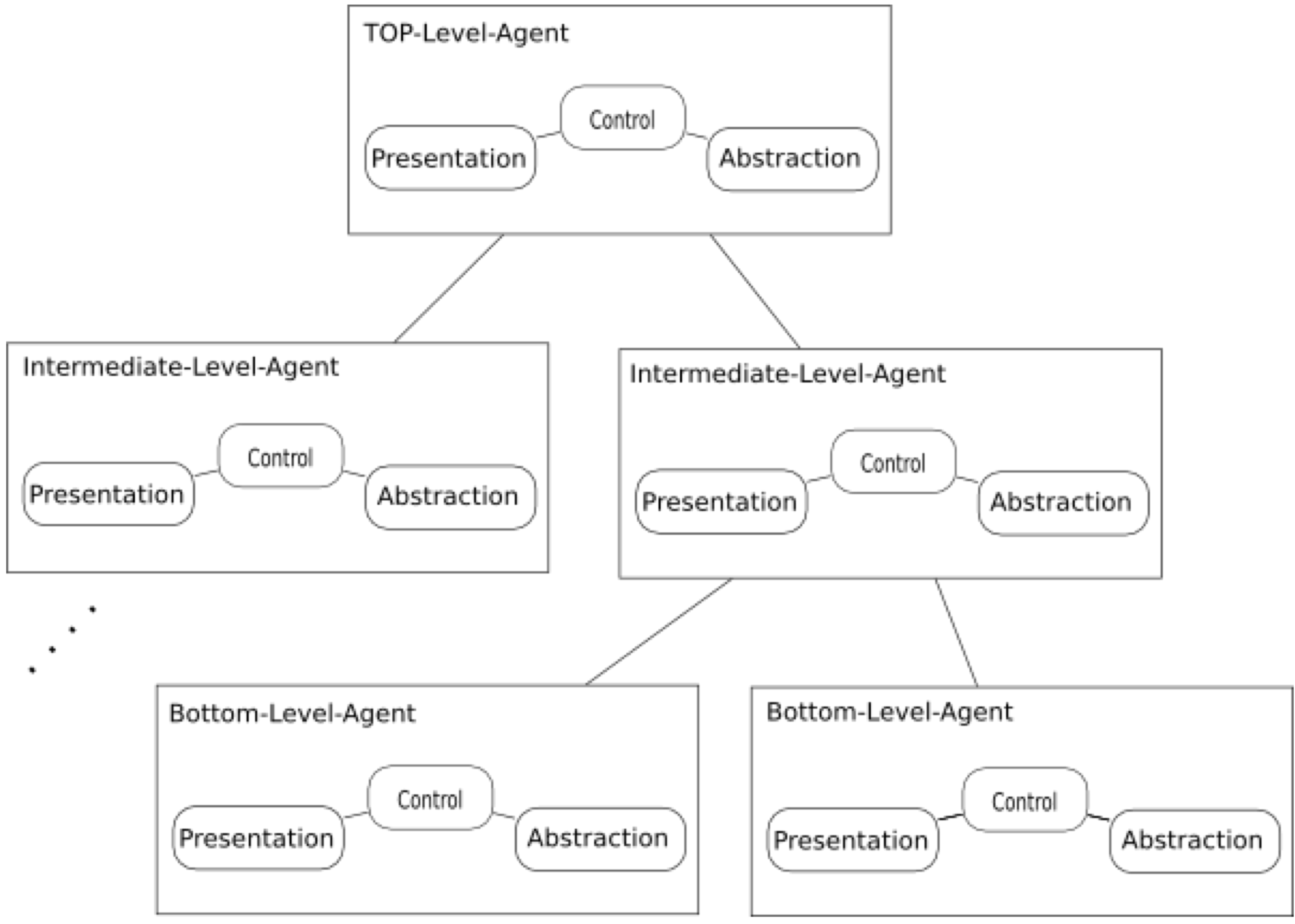

invoke
Raise event
user
UI
BLL
DAL
3-Layer Architecture
Controllers
View
Services
Tasks
Repositories
MVC Architecture
Presenter
View
Services
Tasks
Repositories
MVP Architecture
ViewModel
View
Services
Tasks
Repositories
MVVM Architecture

Alternative to MVC: PAC
(Presentation Abstraction Control)





PAC



PAC: responsibilities

- The Presentation is the visual representation of a particular abstraction within the application, it is responsible for defining how the user interacts with the system
- The Abstraction is the business domain functionality within the application
- The Control is a component which maintains consistency between the abstractions within the system and their presentation to the user in addition to communicating with other Controls within the system
- Note: some descriptions use the term “agent” to describe each Presentation-Abstraction-Control triad

PAC: collaborations

- the Presentation-Abstraction-Control pattern approaches the organization of an application as a hierarchy of subsystems rather than layers of responsibility (e.g. Presentation Layer, Domain Layer, Resource Access Layer, etc.)
- Each system within the application may depend on zero or more subsystems to accomplish its function
- By organizing systems into a hierarchy of subsystems, each of which are composed of the same PAC components, any level of granularity of the system can be inspected while maintaining the same architectural model

PAC: benefits

- Separation of concerns: each triad (PAC agent) is responsible for a part of the application
- Adding new agents is easy
- Agents are easy to distribute
- Changes within an agent do not affect other agents

References

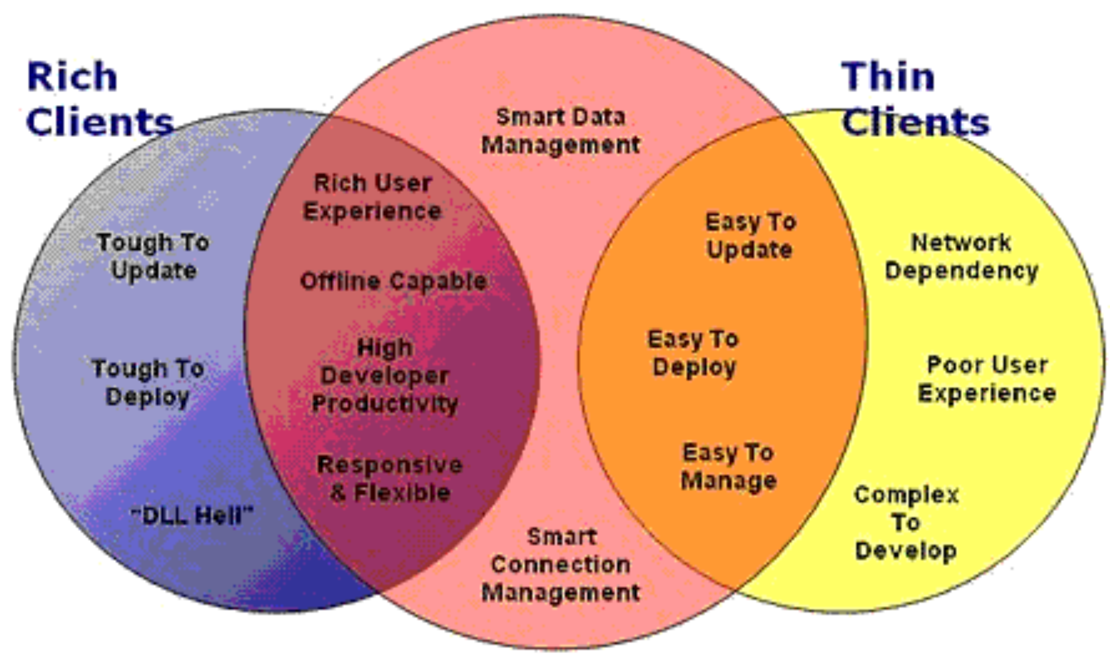
- www.martinfowler.com/eaaDev/ModelViewPresenter.html
- www.codeproject.com/Articles/42830/Model-View-Controller-Model-View-Presenter-and-Mod
- www.infragistics.com/community/blogs/todd_snyder/archive/2007/10/17/mvc-or-mvp-pattern-whats-the-difference.aspx
- www.javacodegeeks.com/2012/02/gwt-mvp-made-simple.html
- msdn.microsoft.com/en-us/library/ff647543.aspx

Styles for the Web

Client-side options

Types of clients

- Rich client
- Thin client
- Smart client
- Special devices



Comparing client-side options

	Rich Client	Thin Client	Smart Client
Advantage	▪ Performance ▪ Offline availability ▪ High integration ▪ Rich UI	▪ Deployment ▪ OS independent ▪ Reach	▪ Performance ▪ Offline availability ▪ High integration ▪ Rich UI ▪ Deployment
Disadvantage	▪ Deployment ▪ OS dependent	▪ Online availability ▪ Simple UI ▪ Limited integration ▪ Performance	▪ OS dependent
Service Interface	▪ Client based ▪ Proxy ▪ Late vs Early binding	▪ Server based ▪ Proxy or direct ▪ Late vs Early binding	▪ Client based ▪ Proxy ▪ Late vs Early binding
Client Technology	▪ Progress 4GL GUI ▪ Microsoft .NET GUI ▪ Java™ GUI	▪ Progress WebSpeed® ▪ Microsoft ASP.NET ▪ Java™ JSP	▪ Progress WebClient™ ▪ Microsoft .NET ▪ Java™ WebStart

Web 2.0

- Primitive UI => Rich UI
  - enable “desktop-like” interactive Web apps
  - enable browser as universal app platform on cell phones
- “Mass customize” to consumer => Social computing
  - tagging (Digg), collaborative filtering (Amazon reviews), etc. => primary value from *users & their social networks*
  - write-heavy workloads (Web 1.0 was mainly read-only)
  - lots of short writes with hard-to-capture locality (hard to share)
- Libraries => Service-oriented architecture
  - Integrate power of other sites with your own (e.g. mashups that exploit Google Maps; Google Checkout shopping cart/payment)
  - Pay-as-you-go democratization of “services are king”
  - Focus on your core innovation
- Buy & rack => Pay-as-you-go Cloud Computing

Rich Internet Apps (RIAs)

- Closing gap between desktop & Web
  - Highly responsive UI's that don't require server roundtrip per-action
  - More flexible drawing/rendering facilities (e.g. sprite-based animation)
  - Implies sophisticated client-side programmability
  - Local storage, so can function when disconnected
- early example: Google Docs + Google Gears
  - include offline support, local storage, support for video, support for arbitrary drawing, ...
- many technologies—Google Gears, Flash, Silverlight...
  - client interpreter must be embedded in browser (plugin, extension, etc.)
  - typically has access to low-level browser state => new security issues
  - N choices for framework * M browsers = N*M security headaches
- proposed HTML5 may obsolete some of these

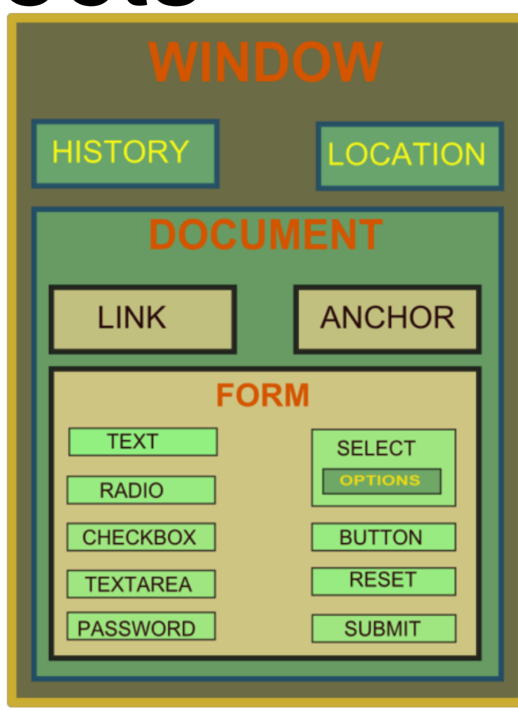
Rich UI with AJAX (Asynchronous Javascript and XML)

- Web 1.0 GUI: click → page reload
- Web 2.0: click → page can update in place
  - also timer-based interactions, drag-and-drop, animations, etc.

How is this done?

1. Document Object Model (1998, W3C) represents document as a hierarchy of elements
2. JavaScript (1995; now ECMAScript) makes DOM available programmatically, allowing modification of page elements *after* page loaded
3. `XMLHttpRequest` (2000) allows async HTTP object requests decoupled from page reload
4. CSS define look and feel

DOM & JavaScript: Document = tree of objects



- hierarchical object model representing HTML or XML doc
- Exposed to JavaScript interpreter
  - Inspect DOM element value/attribs
  - Change value/attribs → redisplay or fetch new content from server
- Every element can be given a unique ID
- JavaScript code can walk the DOM tree or select specific nodes via provided methods

```
<input type="text" name="phone_number" id="phone_number"/>
<script type="text/javascript">
 var phone = document.getElementById('phone_number');
 phone.value='555-1212';
 phone.disabled=true;
 document.images[0].src="http://.../some_other_image.jpg";
</script>
```

JavaScript

- A browser-embedded scripting language
  - OOP: classes, objects, first-class functions, closures
  - dynamic: dynamic types, code generation at runtime
  - JS code can be embedded inline into document...

```
<script type="text/javascript">  <!-- # protect older browsers
  calculate = function() {   ...  }  // -->
</script>
```

  - ...or referenced remotely: `<script src="http://evil.com/Pwn.js"/>`
- Current page DOM available via *window, document* objects
  - *Handlers* (callbacks) for UI & timer events can be attached to JS code, either inline or by function name: `onClick, onMouseOver,...`
  - Changing attributes/values of DOM elements has side-effects, e.g.:

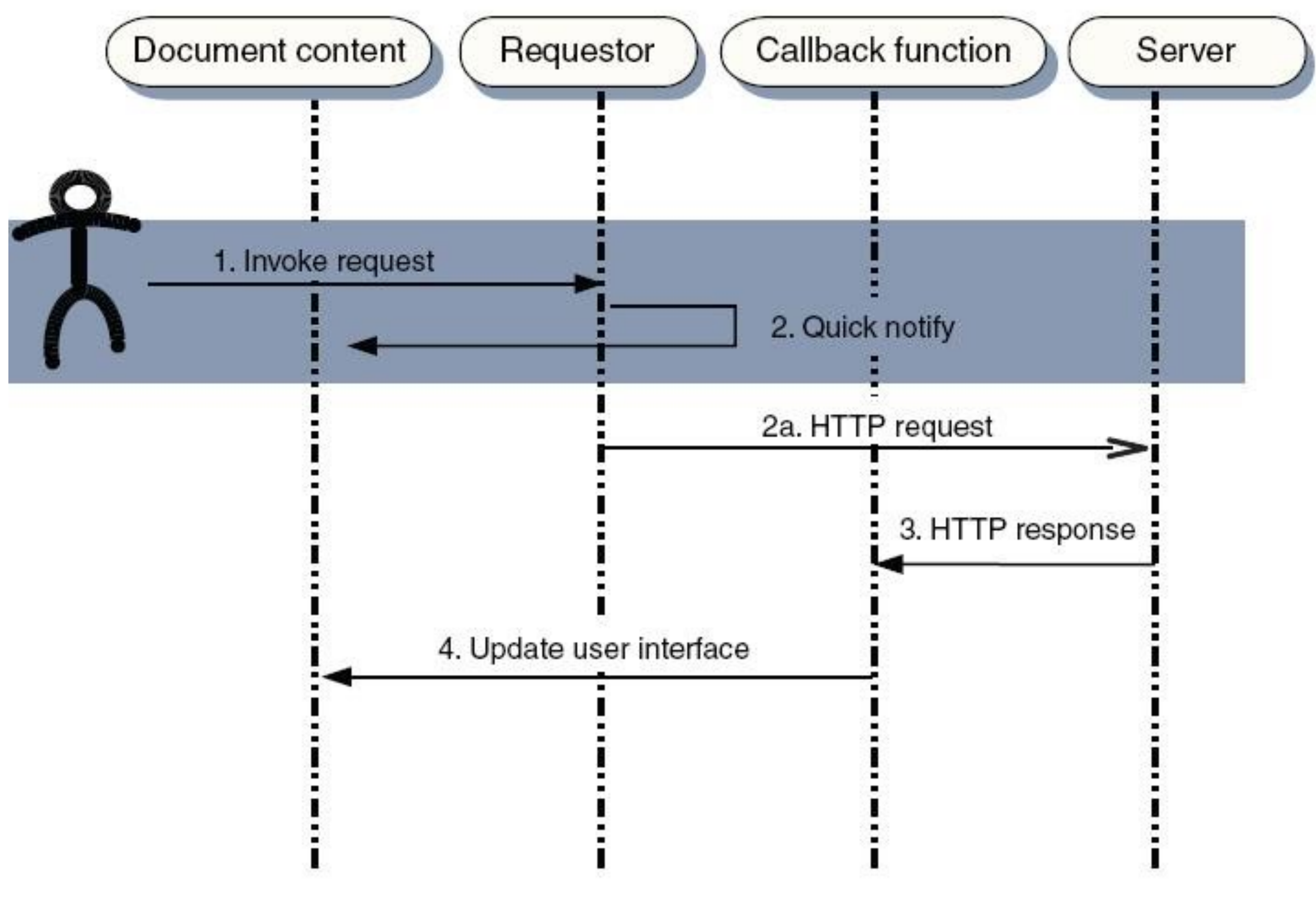
```
<a href="#" onClick="this.innerHTML='Presto!'">Click me</a>
```

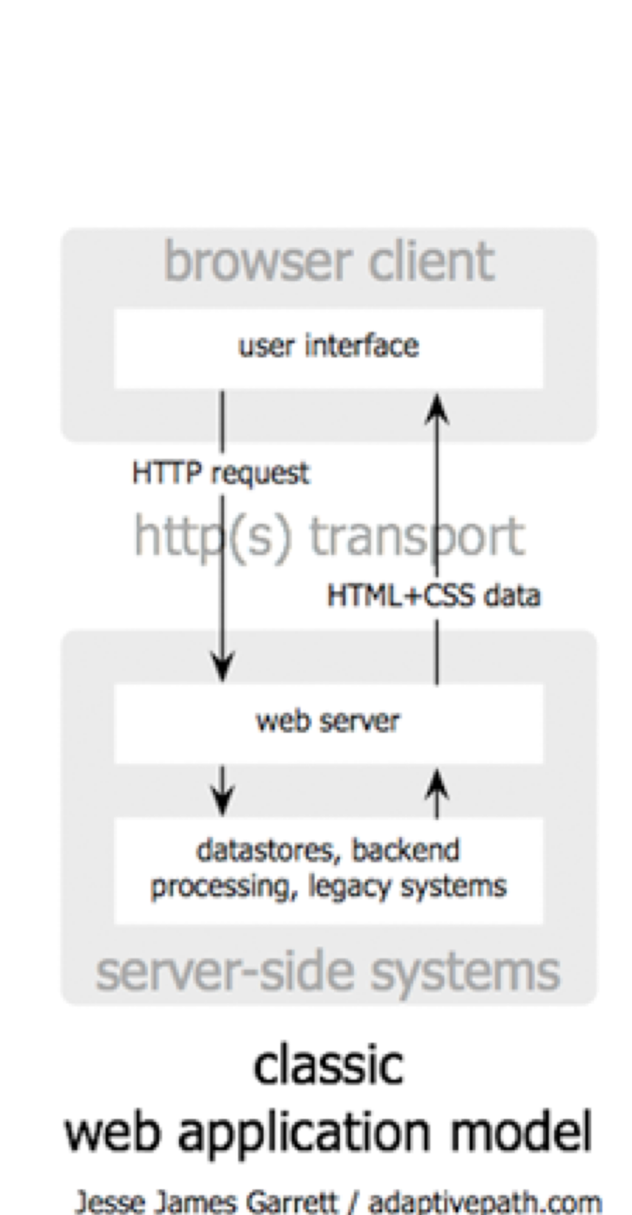
AJAX = Asynchronous Javascript and XML

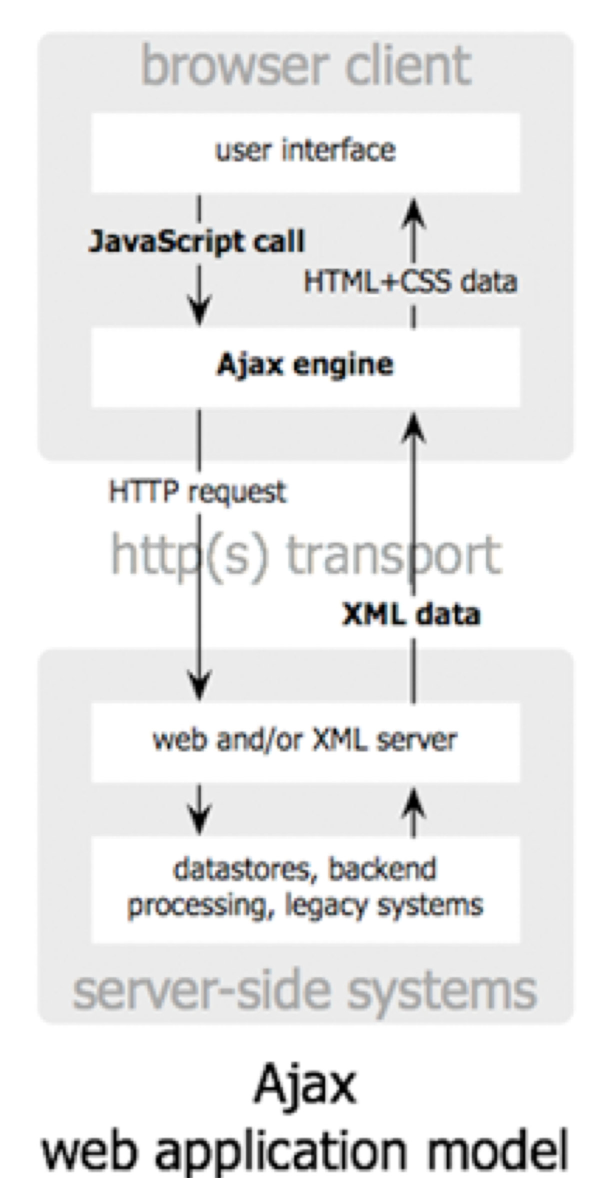
- Recipe:
  - attach JS handlers to events on DOM objects
  - in handler, inspect/modify DOM elements and optionally do *asynchronous* HTTP request to server
  - register callback to receive server response
  - response callback modified DOM using server-provided info
- JavaScript as a target language
  - Google Web Toolkit (GWT): compile Java => emit JS
  - Rails: runtime code generation ties app abstractions to JS

Async comm in a web page

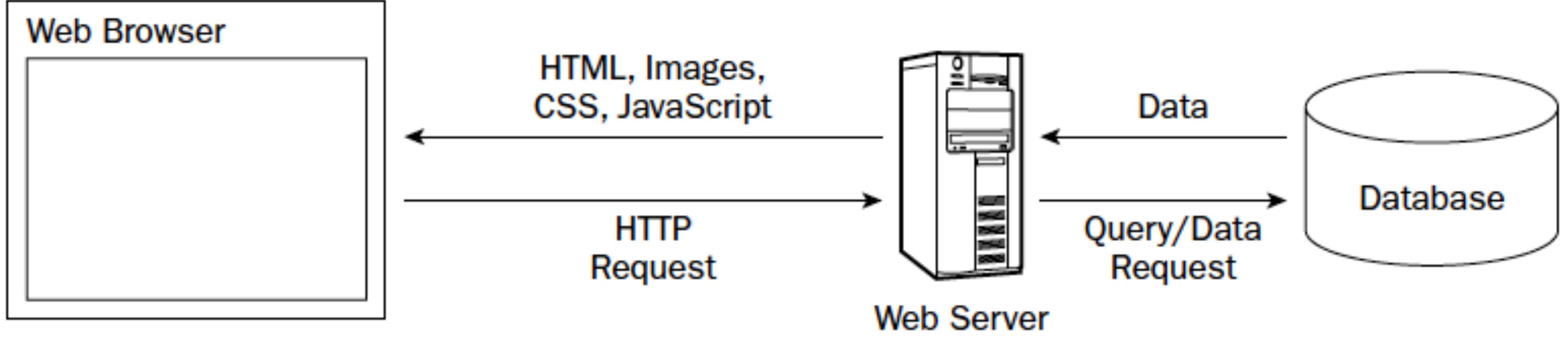
User action invokes a request from an XMLHttpRequest object, which initiates a call to the server asynchronously. The method returns very quickly, blocking the user interface for only a short period of time, represented by the height of the shaded area. The response is parsed by a callback function, which then updates the user interface accordingly.



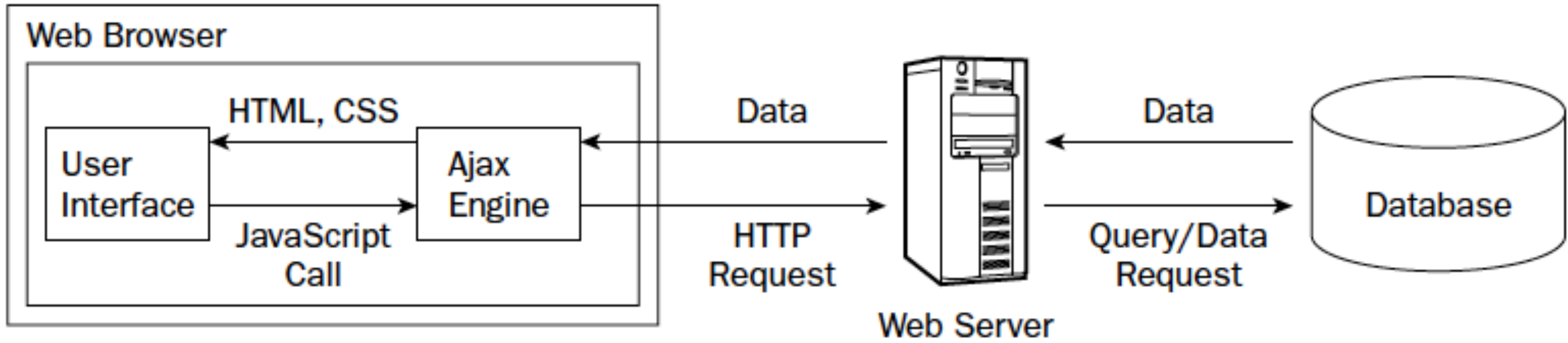

browser client
user interface
HTTP request
http(s) transport
HTML+CSS data
web server
datastores, backend processing, legacy systems
server-side systems
classic
web application model
Jesse James Garrett / adaptivepath.com

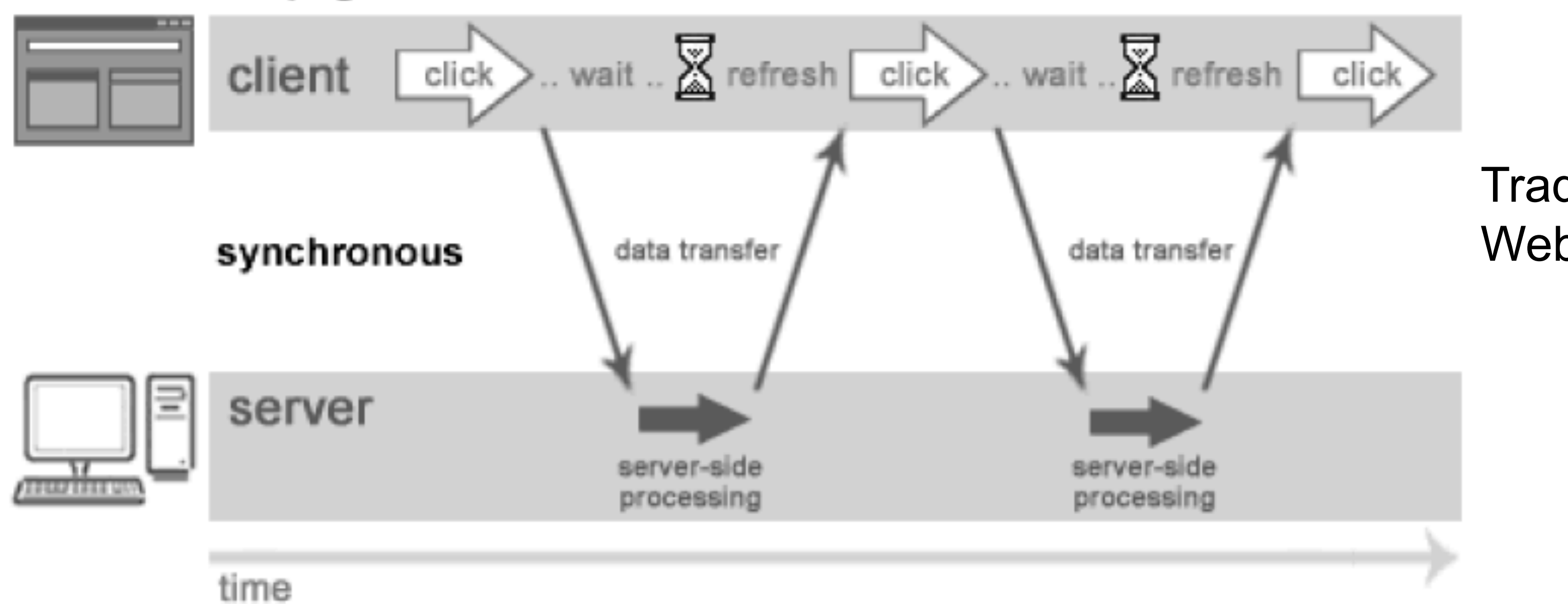

browser client
user interface
JavaScript call
HTML+CSS data
Ajax engine
HTTP request
http(s) transport
XML data
web and/or XML server
datastores, backend processing, legacy systems
server-side systems
Ajax
web application model

Traditional Web Application Model

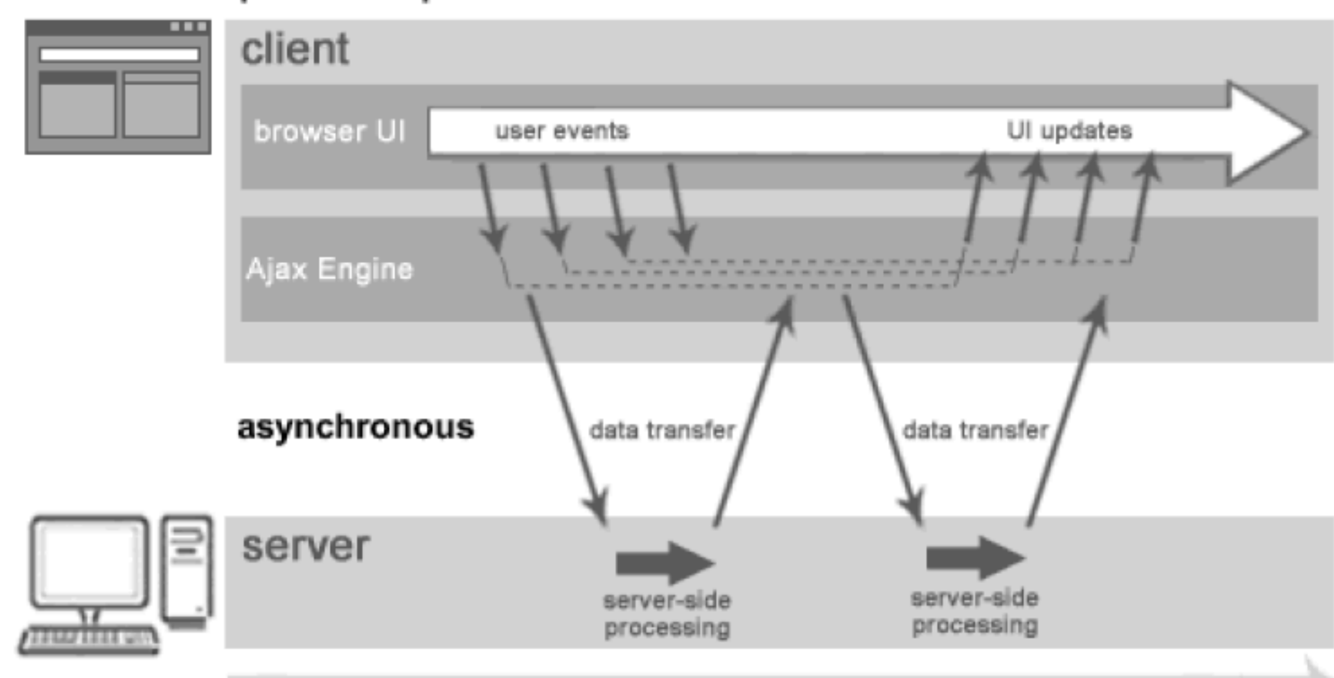


Ajax Web Application Model



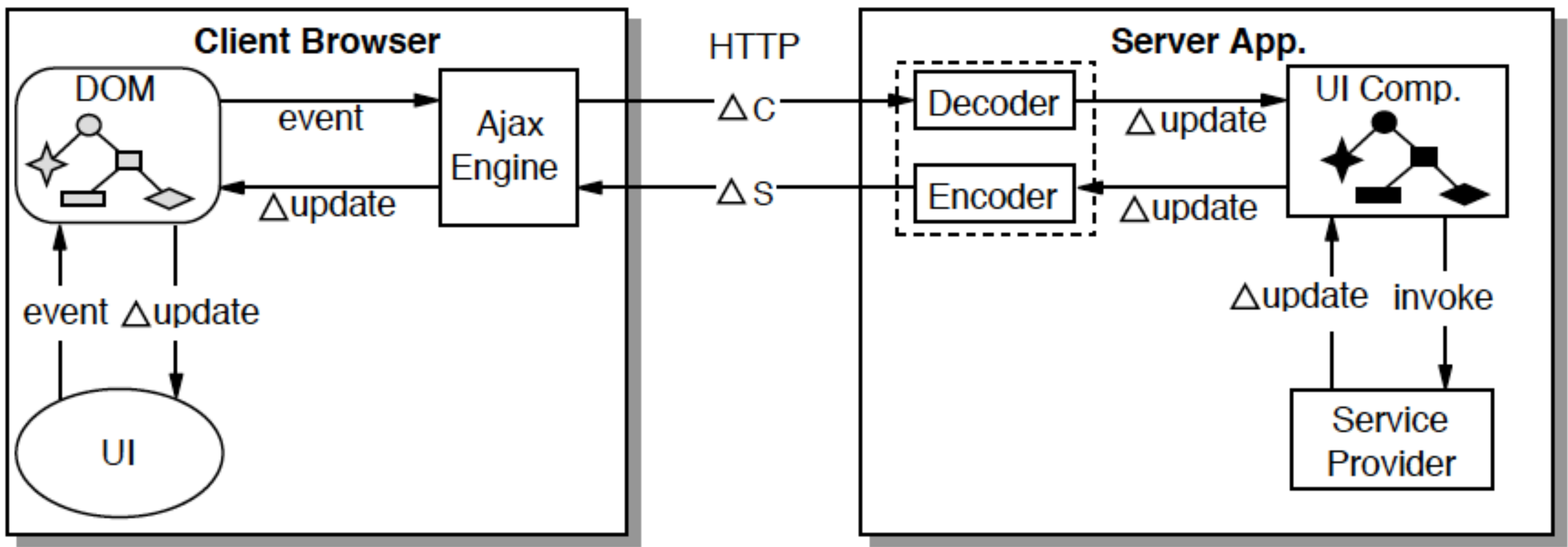


Traditional
Web interaction



AJAX
Web interaction

An architecture based on AJAX



http://arxiv.org/pdf/cs/0608111.pdf

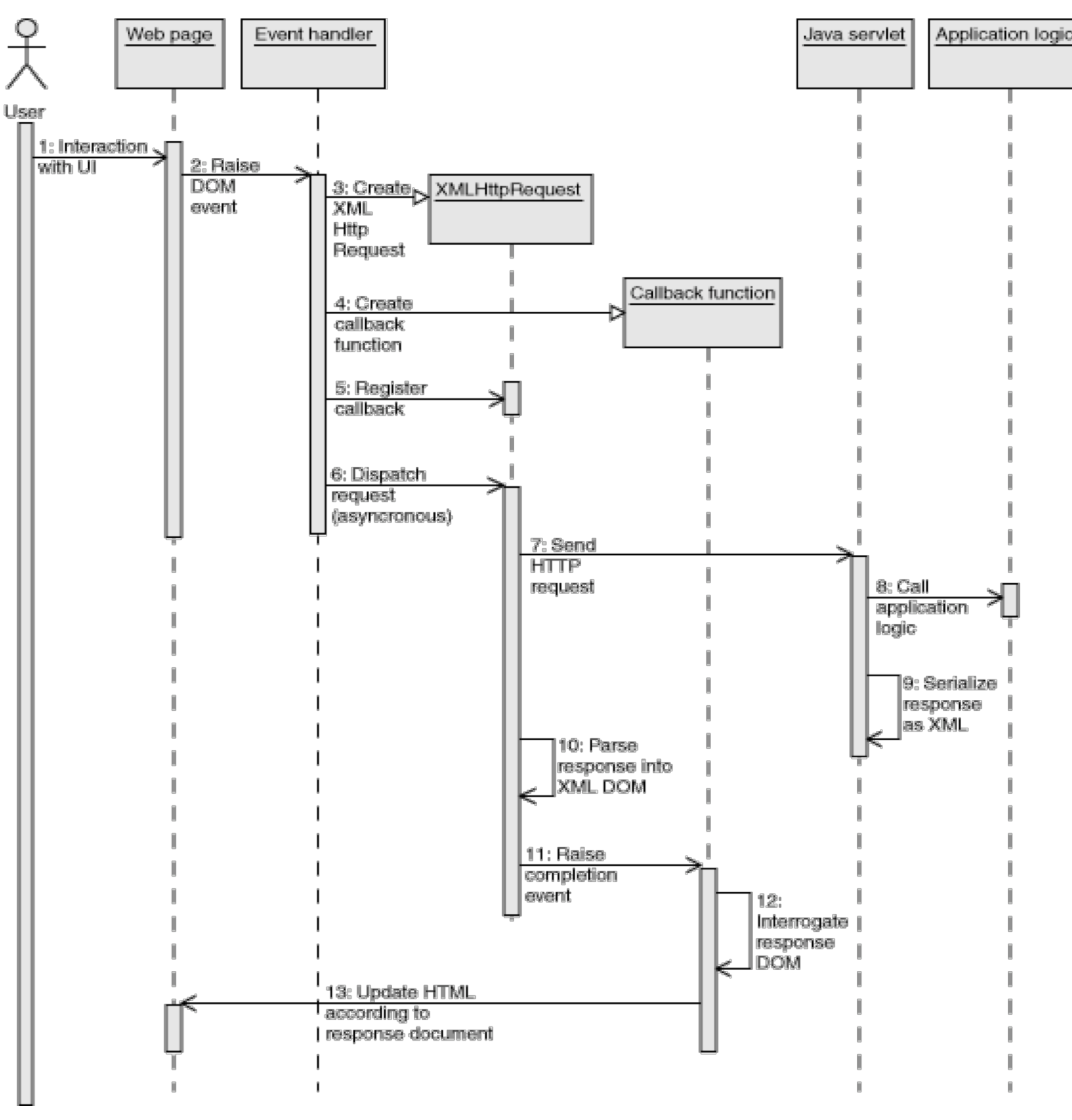
AJAX

- AJAX uses JavaScript (or VBScript) on the client side
- Client-side Javascript libraries: Dojo or Prototype or Jquery
- The server side might be in any language, e.g., PHP, ASP.NET, or Java
- Any XML document or text may be returned to the caller: simple text, JavaScript Object Notation (JSON) or XML are common
- Frameworks (RoR, JSF, etc.,) support Ajax.
- Normally, requests may only go back to the originating server.
- However, mashups (combining diverse site output) may still be constructed by having the originating server make calls to other servers
- Client side mashups may also be built with a dynamic <script> tag

- See www.openajax.org

Some useful concepts

- A servlet is Java code that runs on the server when an HTTP request arrives.
- A Java Server Page (JSP) is XHTML+ Java and is compiled to a servlet
- JavaScript is not Java and runs in the browser
- XHTML is one of many XML languages



This sequence diagram shows a typical AJAX interaction

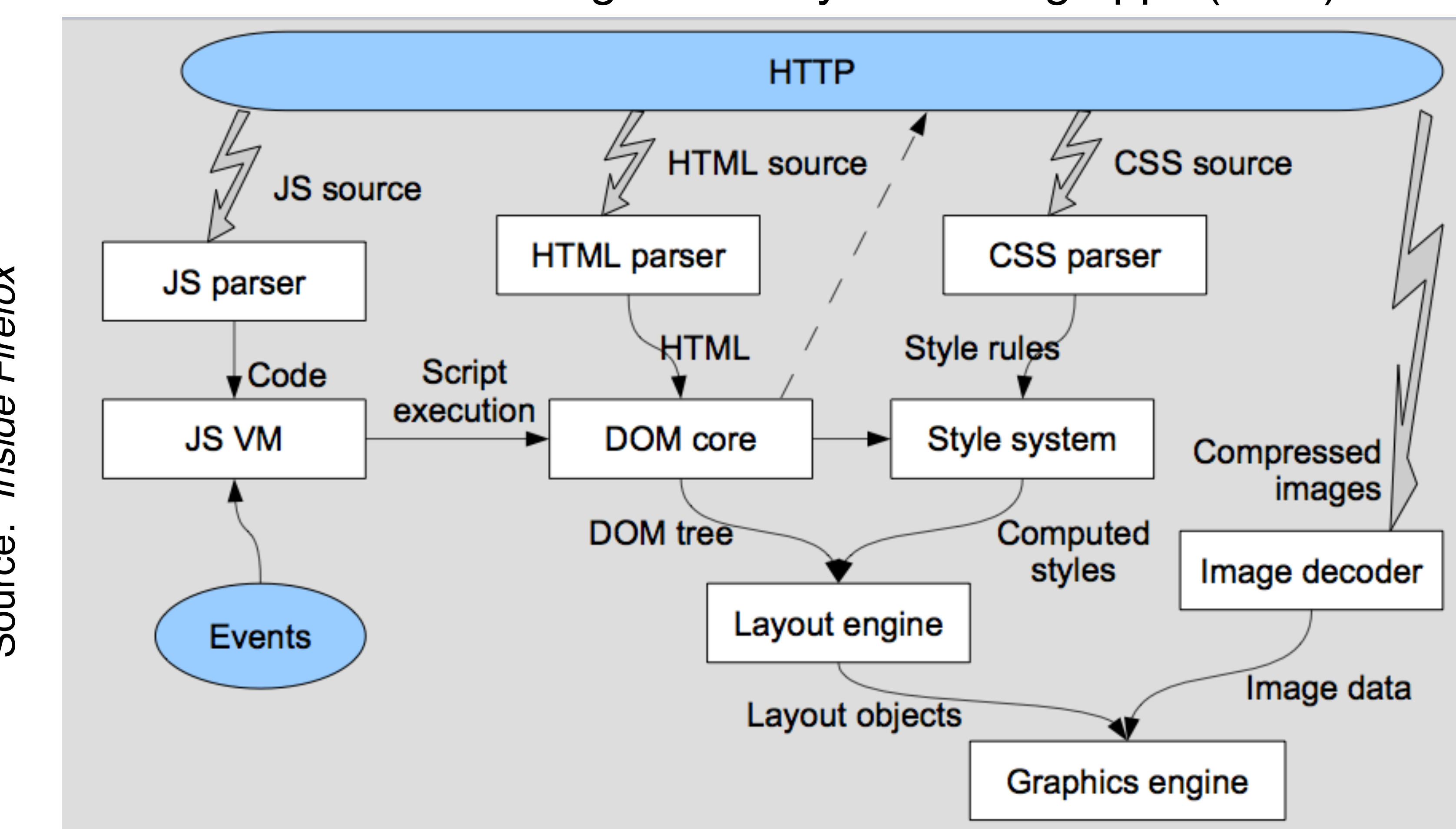
A solid arrowhead represents a synchronous call

A stick arrowhead represents an asynchronous signal

A browser is an application container

- “Web apps” include animation, sound, 3D graphics, disconnection, responsive GUI...
  - Browser =~ new OS: manage mutually-untrusting apps (sites)

Source: *Inside Firefox*

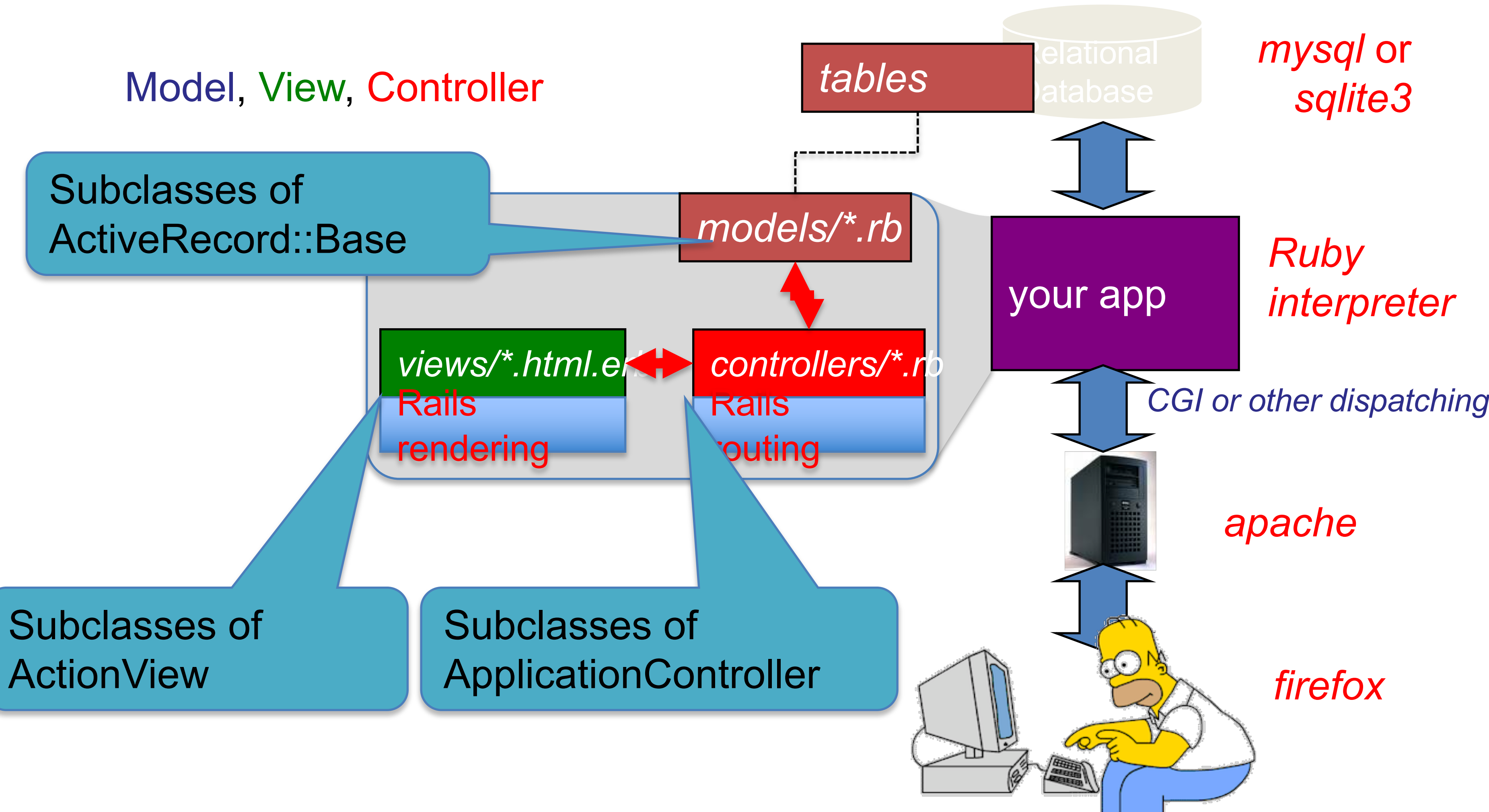


Browsers are fatter and fatter

- RIA frameworks/plugins: Flash, HTML 5, Silverlight
- Performance, security, energy issues
- Cookie and JavaScript management
- CSS layout, side-effects of DOM manipulation
- Documents can change anytime after loading
- Device variation: new issues due to power limits
- Bring “desktop-quality” browsing to mobile tel
- Enabled by 4G networks and better output devices
- Parsing, rendering, scripting are all potential bottlenecks
- *See* `www.google.com/googlebooks/chrome` for a description in comic-book form

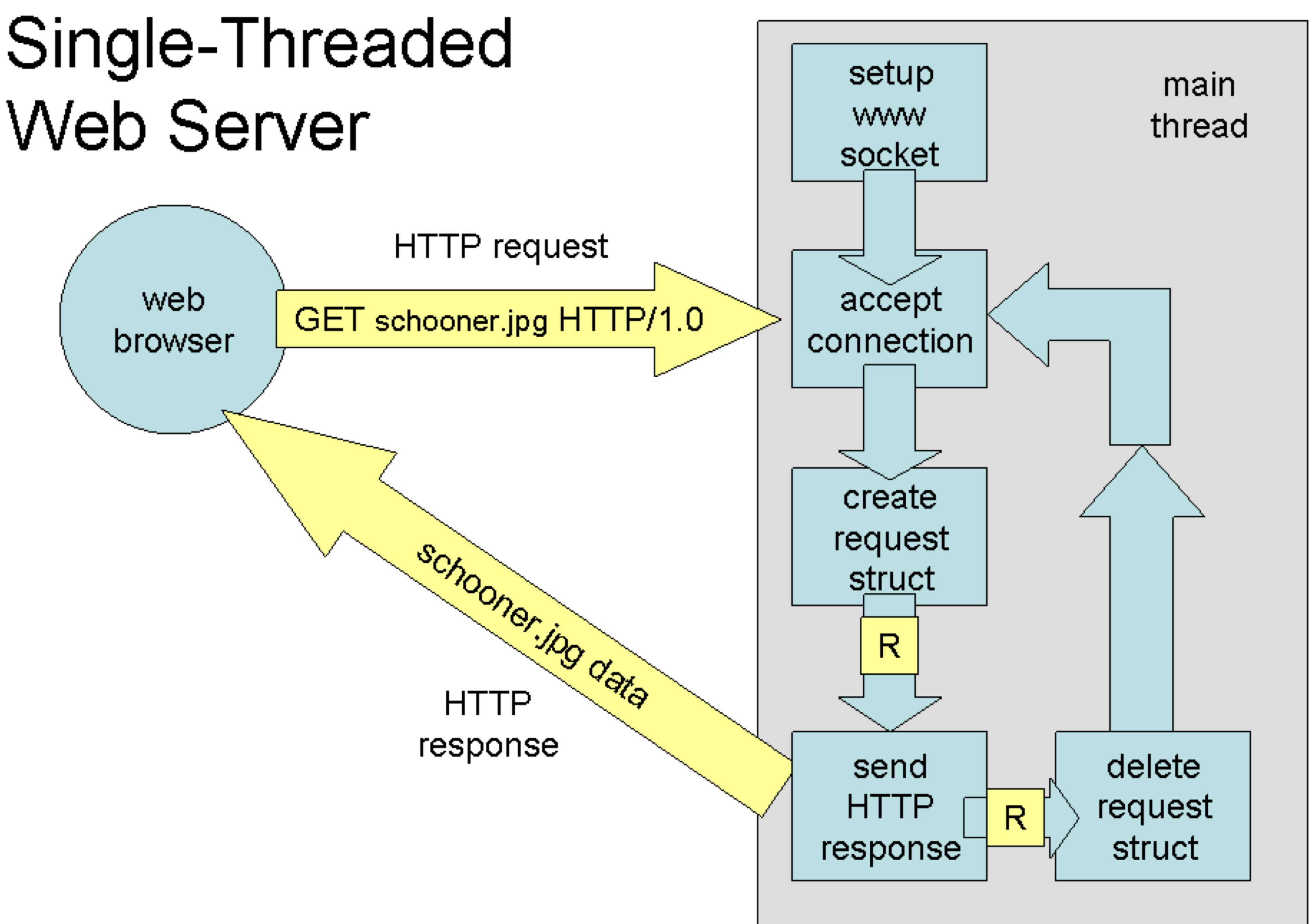
Example: Rails, a Ruby-based MVC Framework

- Implemented almost entirely in Ruby
- Distributed as a Ruby “gem” (collection of related libraries & tools)
- Connectors for most popular databases

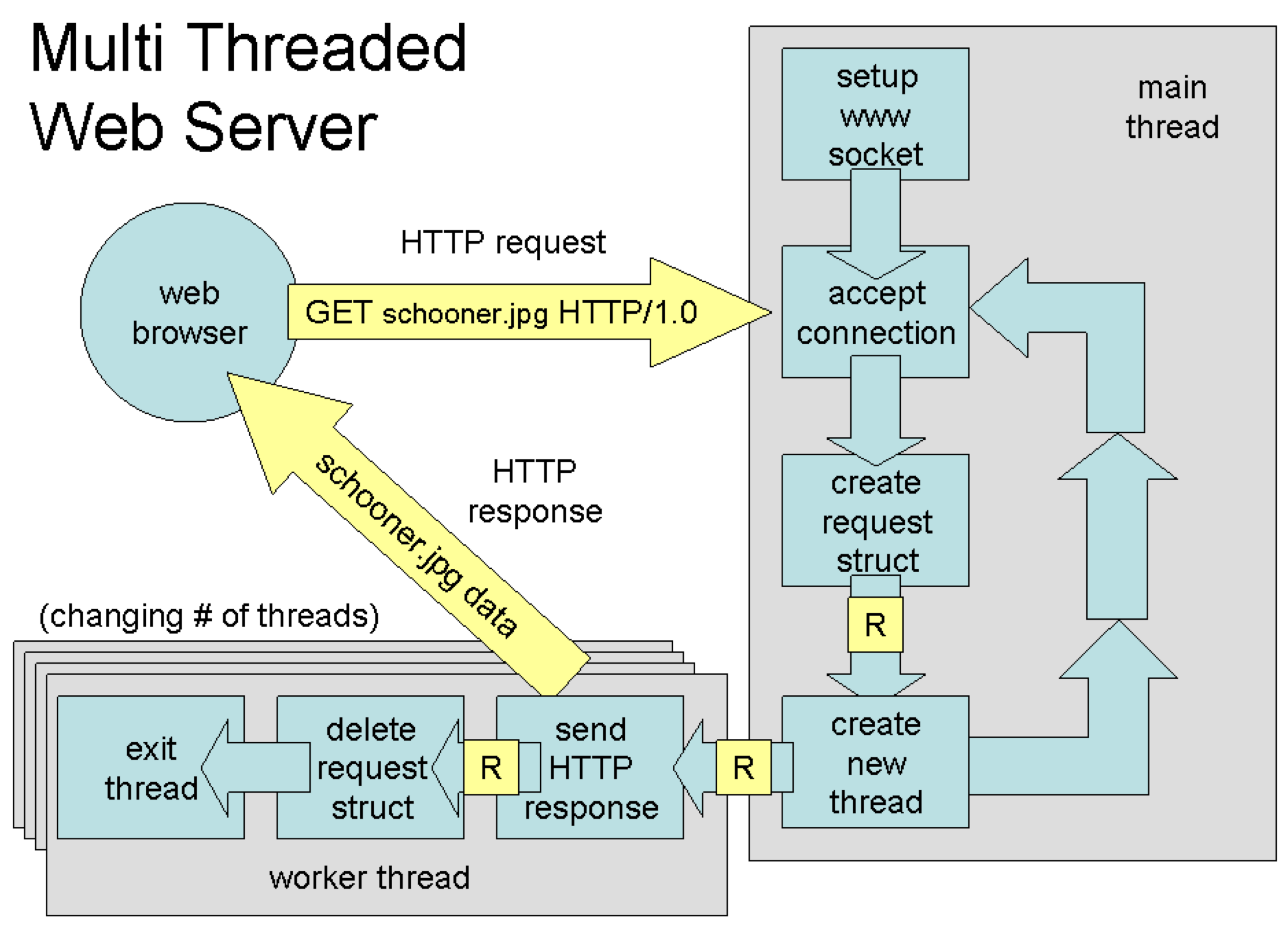


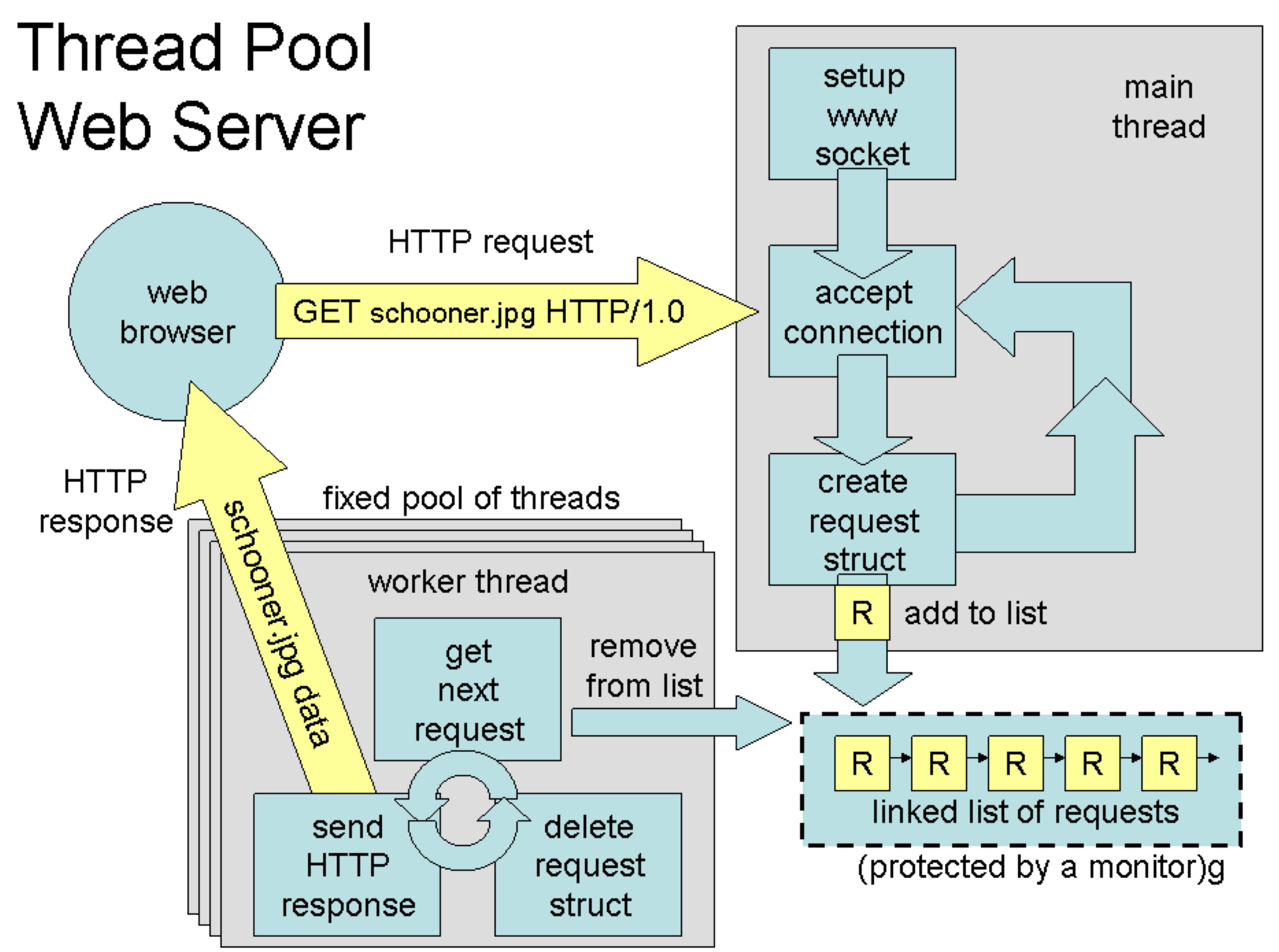
Pool of threads:
an architectural pattern for WebServers

- A single threaded web server is not likely to scale up to many simultaneous clients
- In a multi-threaded web server, the main loop listens for incoming client connections. When it discovers one, it reads the HTTP request and creates a request structure. It then creates a new thread and passes it the request structure
- However, most thread packages are limited to a fixed maximum number of threads. Also, thread creation and deletion can be expensive operations unnecessarily repeated under high load. Finally, a given machine may achieve optimum performance for a certain number of threads, independent of the number of actual clients
- If we control the number of threads without regard to the number of clients, the server can be tuned to maximum performance. So, most real-world servers use a thread-pool approach



www3.nd.edu/~dthain/courses/cse30341/spring2009/project4/project4.html


Multi Threaded
Web Server
main
thread
setup
www
socket
web
browser
HTTP request
GET schooner.jpg HTTP/1.0
accept
connection
HTTP
response
schooner.jpg data
create
request
struct
R
(changing # of threads)
exit
thread
delete
request
struct
R
send
HTTP
response
R
create
new
thread
worker thread


Thread Pool
Web Server
setup
www
socket
main
thread
HTTP request
web
browser
GET schooner.jpg HTTP/1.0
accept
connection
create
request
struct
HTTP
response
fixed pool of threads
schooner.jpg data
worker thread
R
add to list
get
next
request
remove
from list
R
R
R
R
R
linked list of requests
(protected by a monitor)g
send
HTTP
response
delete
request
struct

Self test

- Which are the motivations at the basis of MVC?
- Which are the motivations at the basis of MVP?
- Which are the motivations at the basis of AJAX?
- Compare the CS style with the MVC style

References

- Gamma et al., Design patterns
- Bushmann et al., POSA 1, Wiley
- aspiringcraftsman.com/2007/08/25/interactive-application-architecture/
- aosabook.org/en/index.html
- apcentral.collegeboard.com/apc/members/courses/teachers_corner/185168.html

Useful sites

- martinfowler.com/eaaDev/uiArchs.html
- todoMVC.com
- www.infoq.com/articles/rest-introduction
- stackoverflow.com/questions/2056/what-are-mvp-and-mvc-and-what-is-the-difference
- www.codeproject.com/Articles/228214/Understanding-Basics-of-UI-Design-Pattern-MVC-MVP
- www.codeproject.com/Articles/34562/COMET-or-Reverse-AJAX-based-Grid-Control-for-ASP-N
- www.programmableweb.com

- Java and MVC java.sun.com/blueprints/patterns/MVC-detailed.html
- .NET and MVC www.devx.com/dotnet/Article/10186/0/page/1
- Interface Hall of Fame homepage.mac.com/bradster/iarchitect/shame.htm
- UI Patterns & Techniques time-tripper.com/uipatterns/Introduction

Questions?