SOFTWARE DESIGN

Concurrency and Presentation

CONTENT

- Handling concurrency
- Presentation layer

REFERENCES

- Martin Fowler et. al, Patterns of Enterprise Application Architecture, Addison Wesley, 2003 [Fowler]
- Microsoft Application Architecture Guide, 2009 [MAAG]
- SaaS Course Stanford
- Ólafur Andri Ragnarsson, Presentation Layer Design, 2014.

OFFLINE CONCURRENCY PATTERNS

- Multiple threads that manipulate the same data

- A solution -> Transaction managers….as long as all data manipulation is within a transaction.

- What if data manipulation spans transactions?

BUSINESS TRANSACTIONS

- ACID
- Transactional resource (ex. Database)
- Increase throughput -> short transactions
- Transactions mapped on a single request
- Late transactions -> read data first, start transaction for updates
- Transactions spanning several requests -> long transactions
- Lock escalation (row level -> table level)

CONCURRENCY PROBLEMS

- Lost updates
- Inconsistent read => Correctness failure
- Liveness – how much concurrency can the system handle?

EXECUTION CONTEXTS

“A **request** corresponds to a single call from the outside world which the software works on and optionally sends back a response”

“A **session** is a long running interaction between a client and server.”

“A **process** is a, usually heavyweight, execution context that provides a lot of isolation for the internal data it works on.”

“A **thread** is a lighter-weight active agent that's set up so that multiple threads can operate in a single process.”

APPLICATION SERVER CONCURRENCY

process-per-session

- Uses a lot of resources

process-per-request

- Pooled processes
- Sequential requests
- Resources for a request should be released

thread-per-request

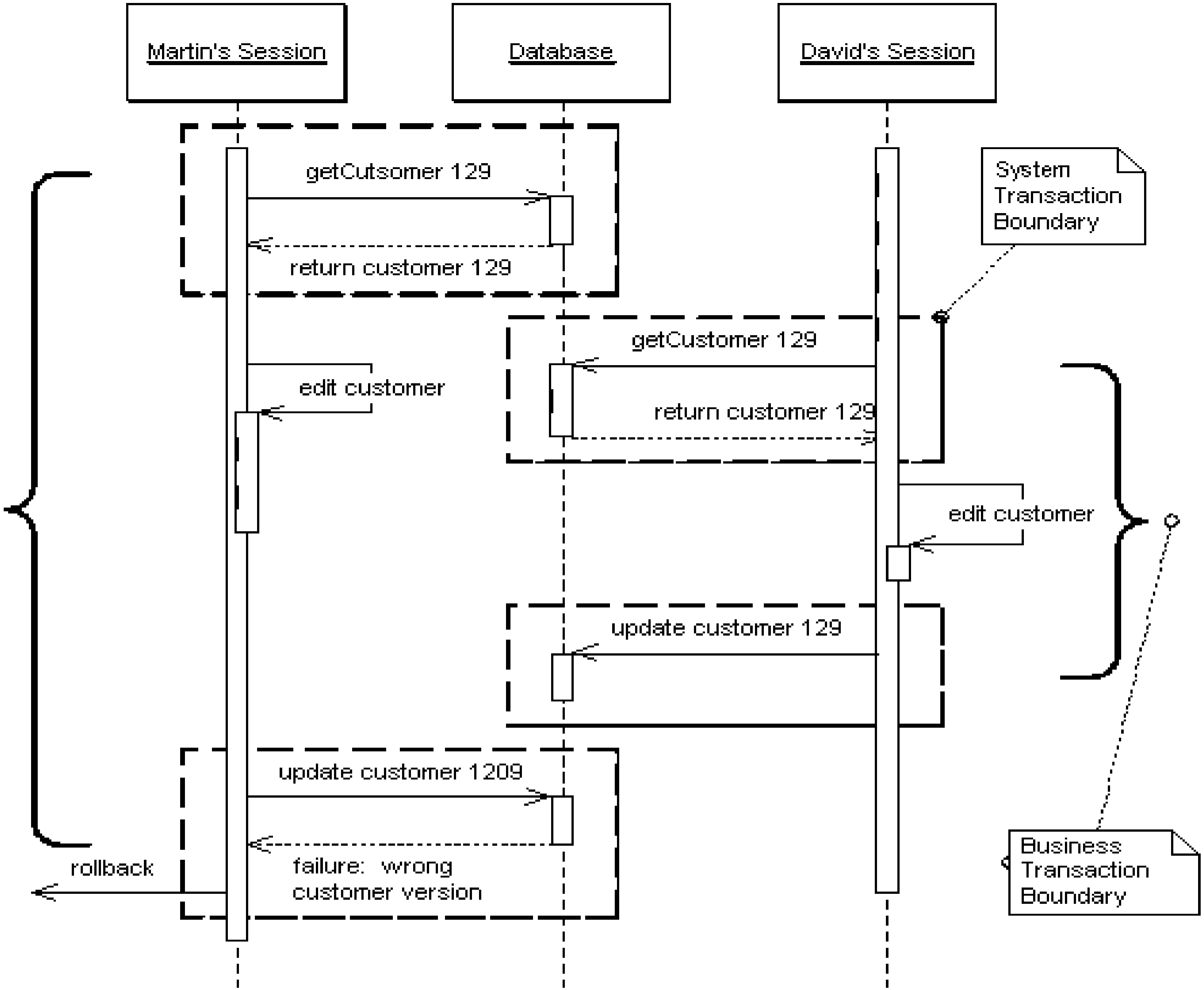
- More efficient
- No isolation

SOLUTIONS

isolation: partition the data so that any piece of data can only be accessed by one active agent.

immutable data: separate the data that cannot be modified.

mutable data than cannot be isolated => Concurrency Control


Martin's Session
Database
David's Session
getCutsomer 129
return customer 129
System
Transaction
Boundary
getCustomer 129
edit customer
return customer 129
edit customer
update customer 129
update customer 1209
failure: wrong
customer version
rollback
Business
Transaction
Boundary

OPTIMISTIC CONCURRENCY CONTROL

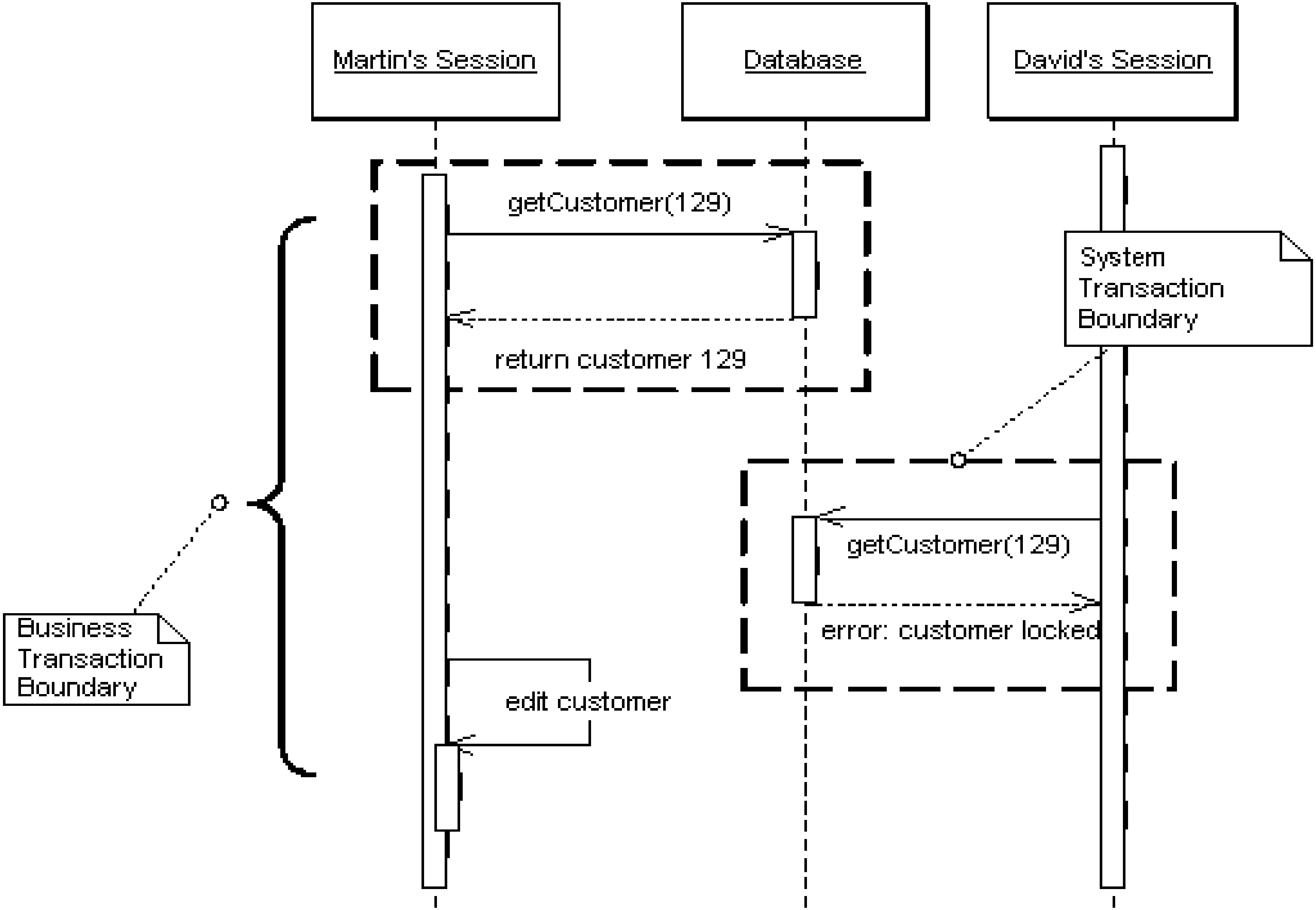
Handles conflicts between concurrent business transactions, by detecting a conflict and rolling back the transaction.

- Conflict detection
- Lock hold during commit
- Supports concurrency
- Suitable for low frequency of conflicts
- Used for not critical consequences

PESSIMISTIC CONCURRENCY CONTROL

Prevents conflicts between concurrent business transactions by allowing only one business transaction to access data at once.

- Conflict prevention
- Lock hold during the entire transaction
- Does not support concurrency
- Used for critical consequences


Martin's Session
Database
David's Session
getCustomer(129)
return customer 129
System Transaction Boundary
getCustomer(129)
error: customer locked
Business Transaction Boundary
edit customer

PREVENTING INCONSISTENT READS

Optimistic control

- Versioning

Pessimistic control

- Read ->shared lock
- Write -> exclusive lock

Temporal reads

- Date+time stamps
- Implies full history storage

DEADLOCKS

- Pick a victim
- Locks with deadlines
- Preventing:
  - Force to acquire all the necessary locks at the beginning
  - Enforce a strategy to grant locks (ex. Alphabetical order of the files)

Combine tactics

LOCKING

To implement it you need to:

- know what type of locks you need,
- build a lock manager,
- define procedures for a business transaction to use locks

Lock types

- Exclusive write lock
- Exclusive read lock
- Read/write lock
  - Read and write locks are mutually exclusive.
  - Concurrent read locks are acceptable

LOCK MANAGER

- Responsibility = to grant or deny any request by a business transaction to acquire or release a lock
- A table that maps locks to owners
- Locks should be private to the lock manager.
- Business transactions should access only the lock manager

Protocol of Business transaction to use the lock manager

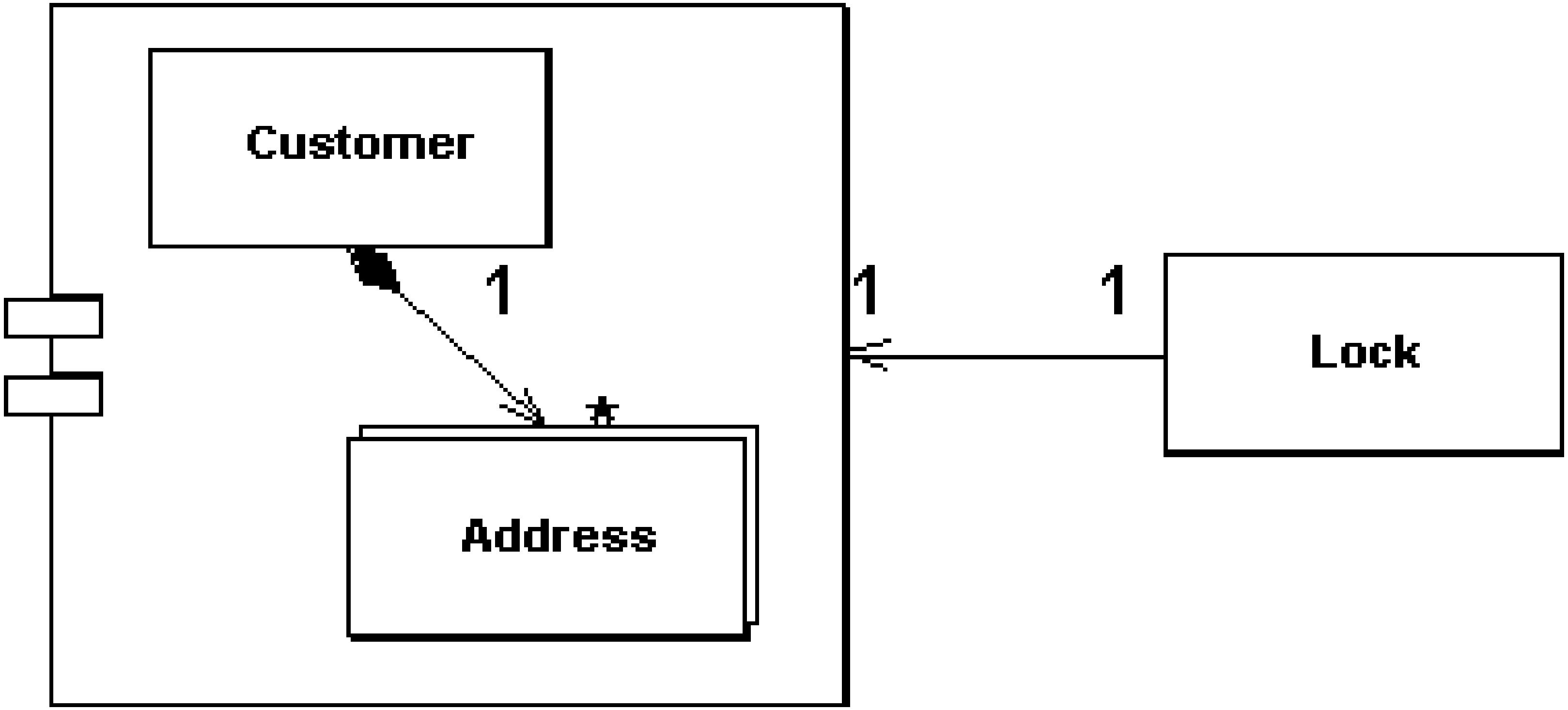
- what to lock (i.e. the ID/primary key)
- when to lock (i.e. lock first, load data next)
- when to release a lock (i.e. after transaction completion),
- how to act when a lock cannot be acquired.

ANALYSIS

- Access to the lock table must be serialized
- Performance bottleneck
- Consider granularity (Coarse grained lock)
- Possible deadlocks
- Lock timeout for lost sessions

COARSE-GRAINED LOCK

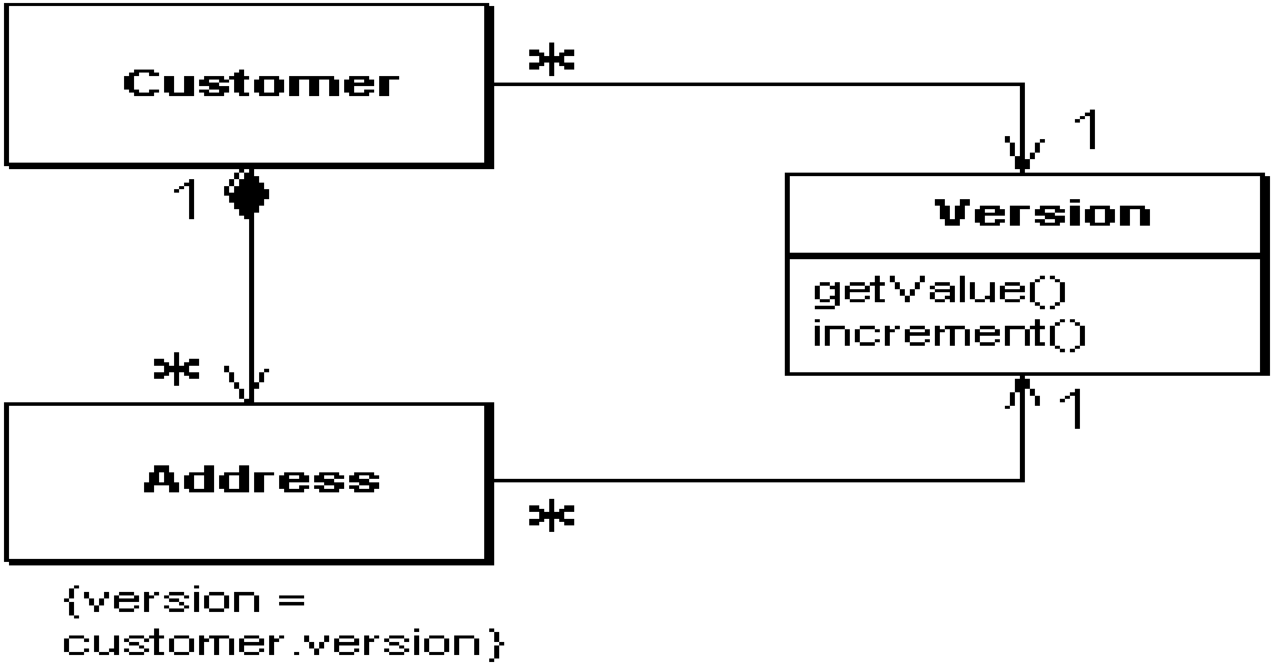
Lock a set of related objects (aggregates)with a single lock



HOW IT WORKS

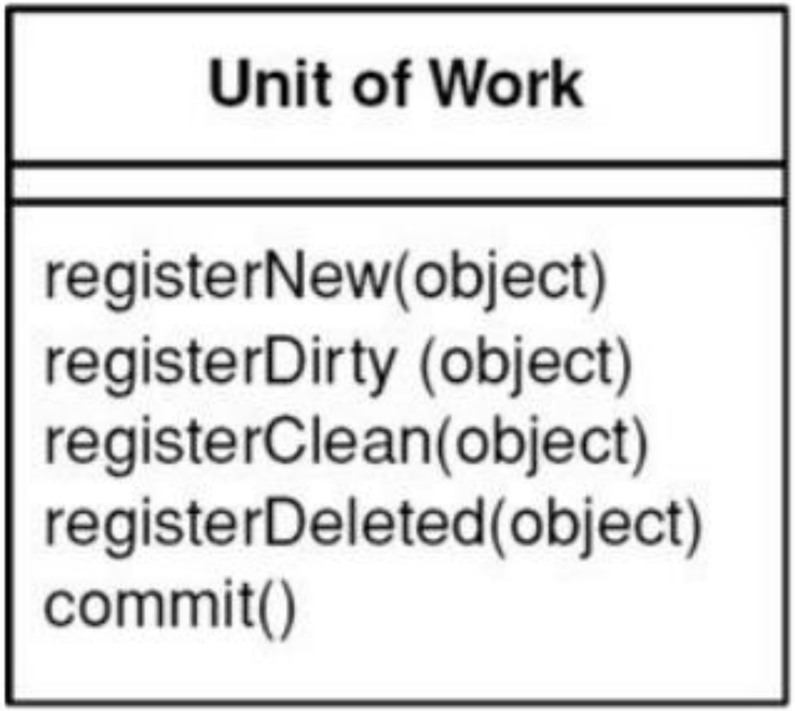
A single point of contention for locking a group of objects

Optimistic Lock – shared version



UNIT OF WORK

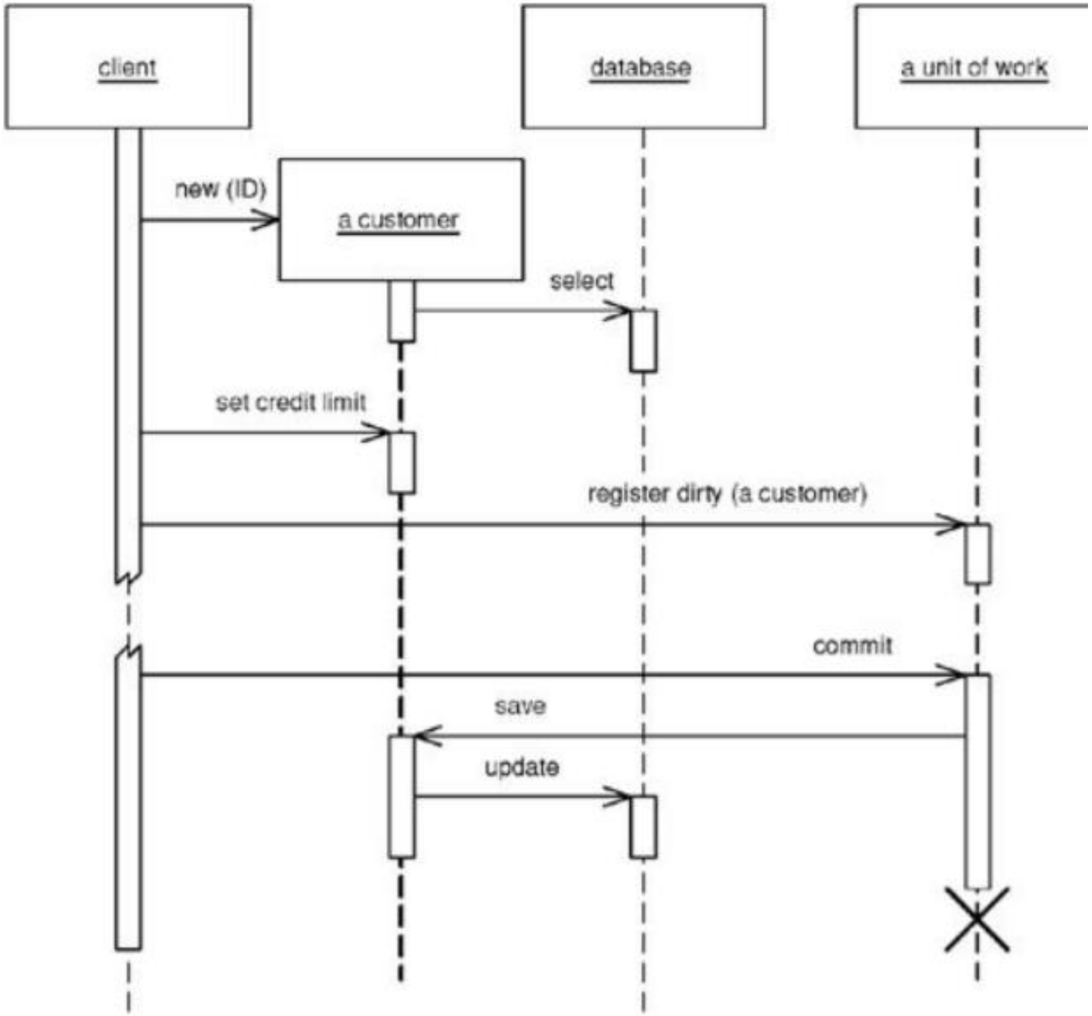
- factors the database mapping controller behavior into its own object.
- maintains a list of objects affected by a business transaction and coordinates the writing out of changes and the resolution of concurrency problems.



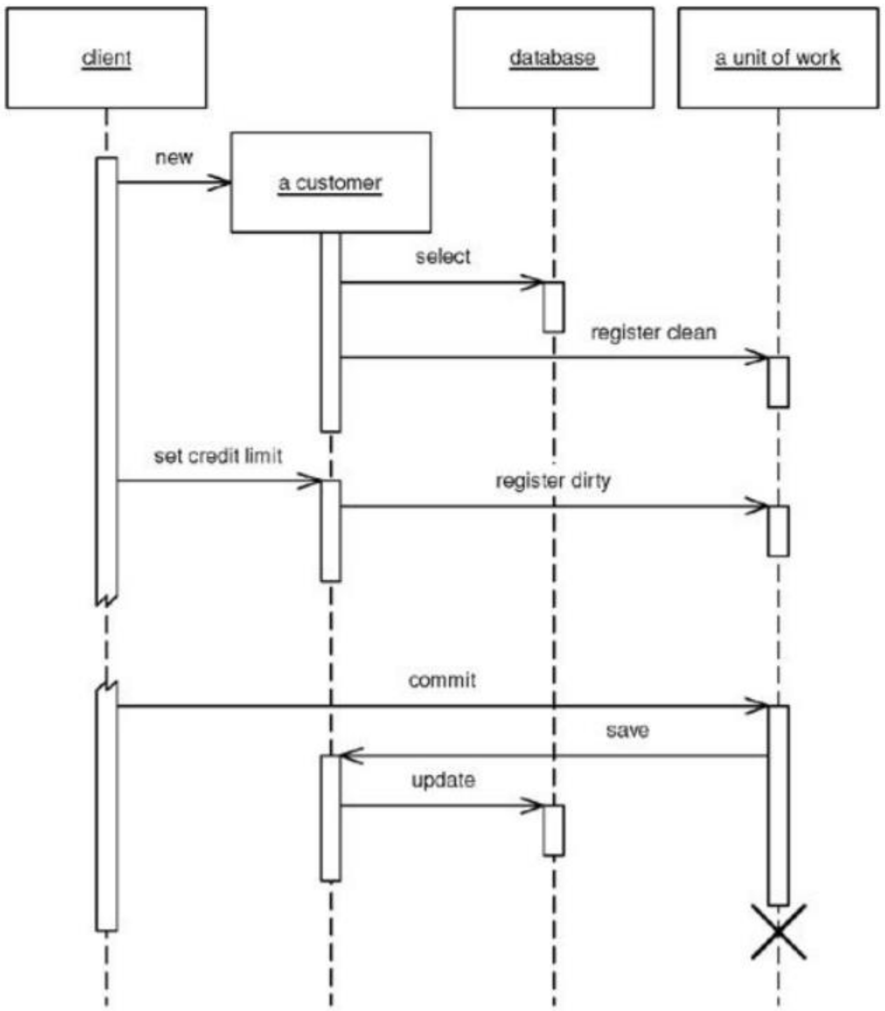
Unit of Work

registerNew(object)
registerDirty (object)
registerClean(object)
registerDeleted(object)
commit()

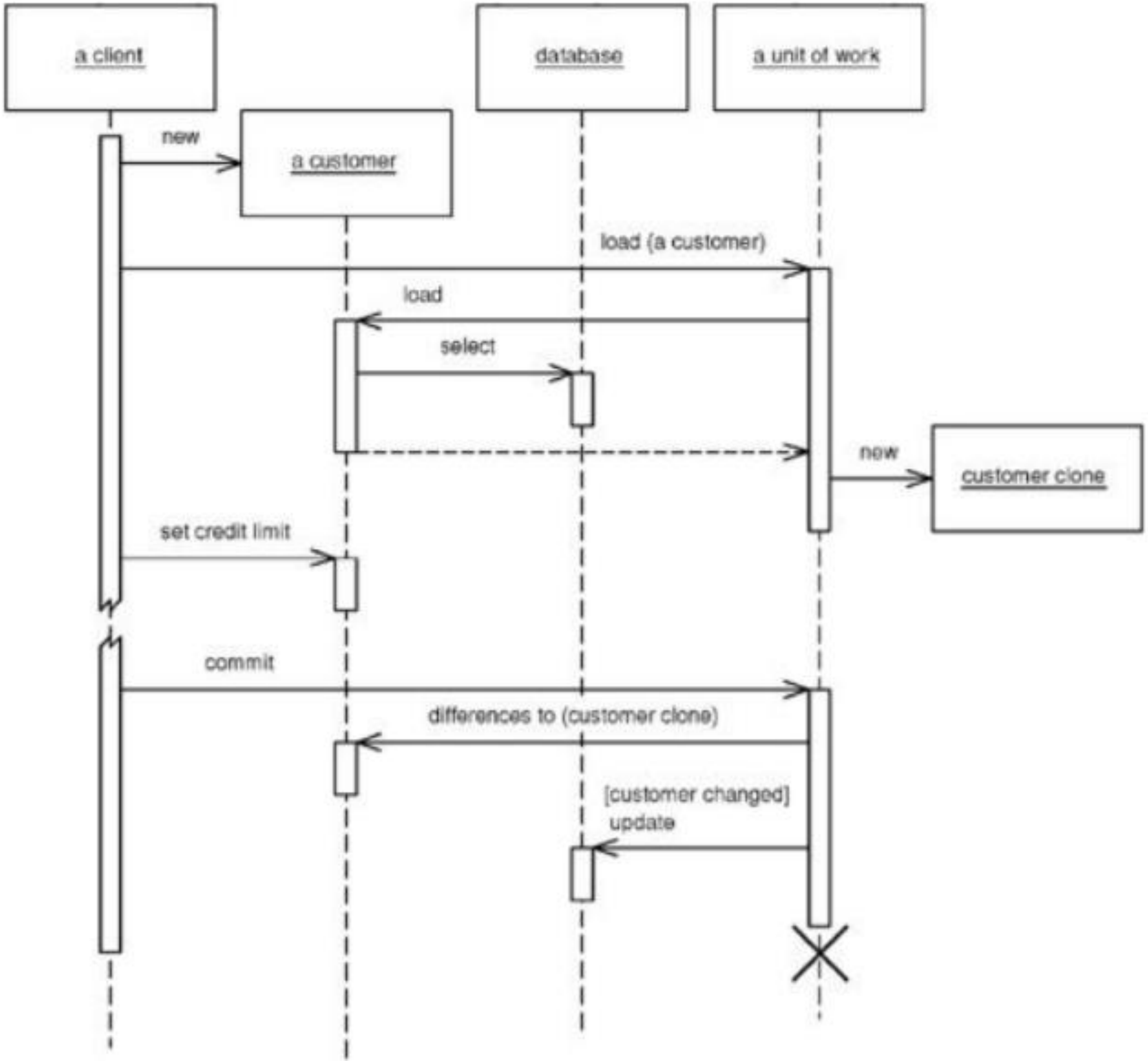
THE CALLER REGISTERS A CHANGED OBJECT



THE RECEIVER OBJECT REGISTERS ITSELF



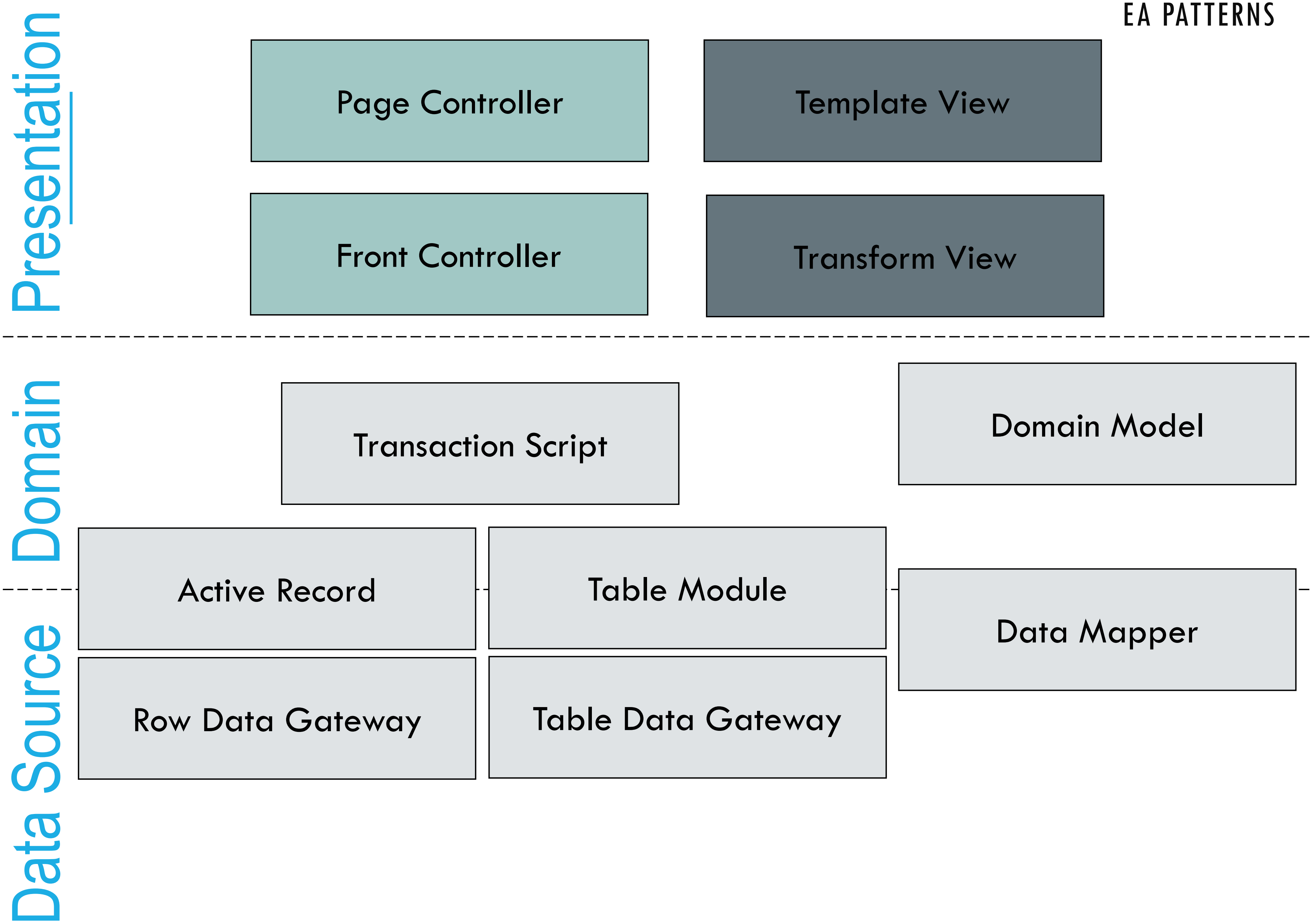
UNIT OF WORK AS THE CONTROLLER FOR DATABASE ACCESS



DISCUSSION

- Unit of Work can be helpful:
  - Controlling the update order (if the database uses referential integrity and checks it with each SQL call)
  - Minimize deadlocks (if transactions use the same sequence of tables to edit, store the order)
  - Handle batch updates

EA PATTERNS



Presentation

Page Controller

Template View

Front Controller

Transform View

Domain

Transaction Script

Domain Model

Data Source

Active Record

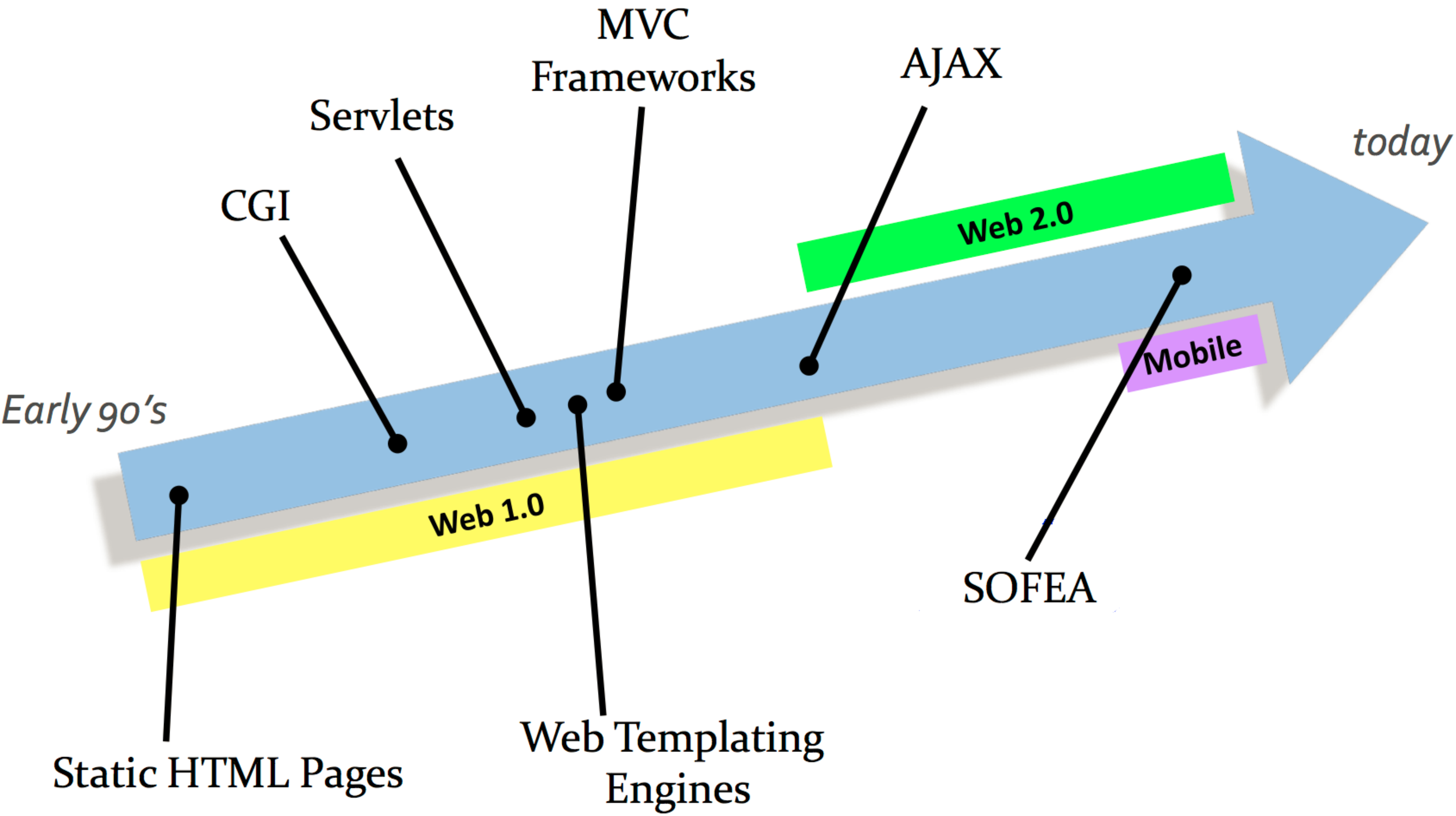
Table Module

Data Mapper

Row Data Gateway

Table Data Gateway

EVOLUTION OF WEB APPLICATION ARCHITECTURE



EARLY TECHNOLOGY

HTML (HyperText Markup Language)

- Standard markup language used to create Web pages

CGI (Common Gateway Interfaces)

- Scripts (usually Perl) using common interface between the Web server and programs that generate Web content

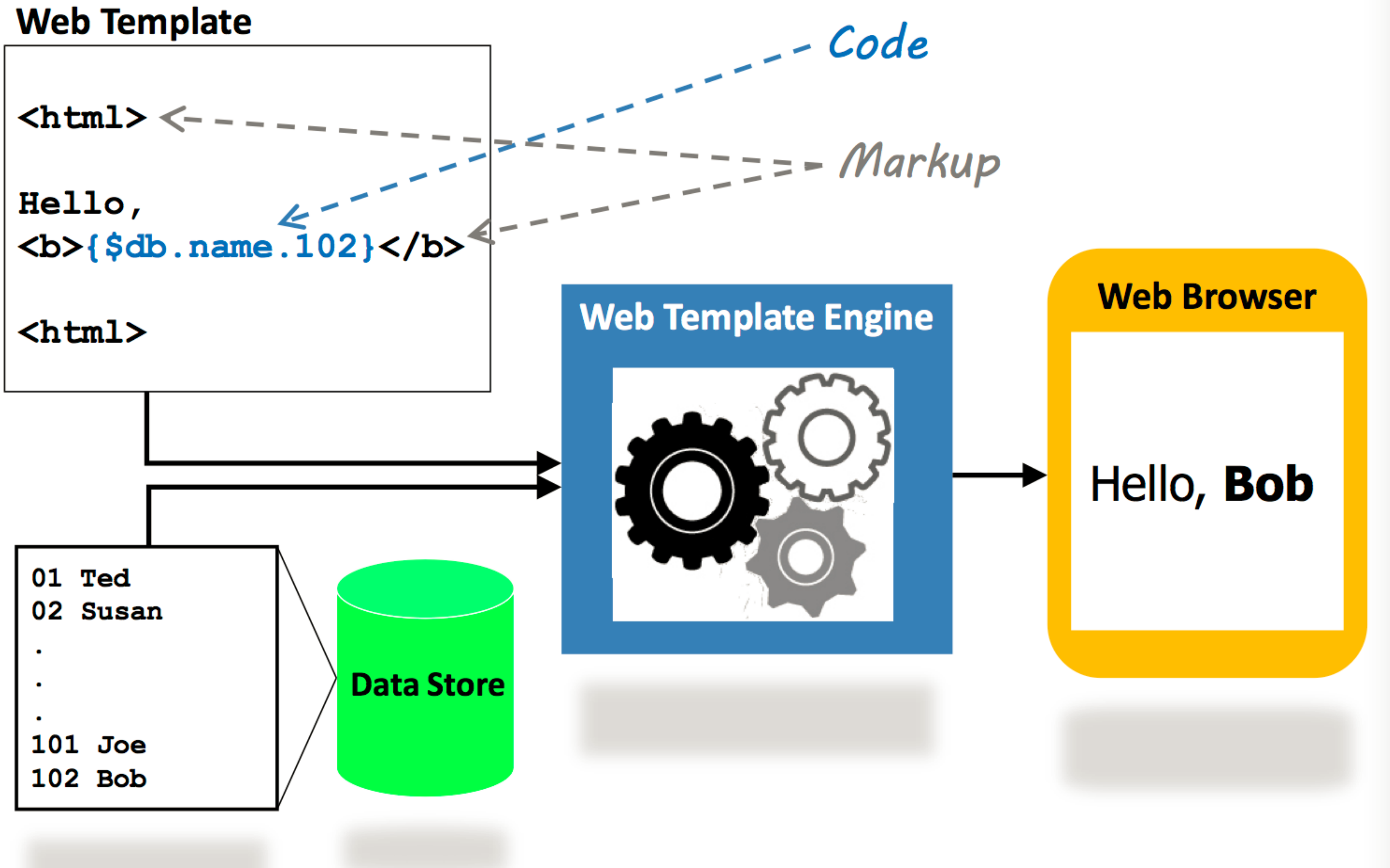
Servlet

- Java programming to extend the capabilities of the web server
- Well defined API through run-time environment
- Typically used for dynamic web content generation

WEB TEMPLATING ENGINE

- Embedded code within static HTML elements
- Mix of static and dynamic HTML
  - “Model 1” Architecture
- Examples
  - Java Server Pages (JSP)
  - PHP
  - Active Server Pages (ASP) .Net

WEB TEMPLATING ENGINE



```
<html>

Hello,
<b>{$db.name.102}</b>

<html>
```

```
01  Ted
02  Susan
.
.
.
101  Joe
102  Bob
```

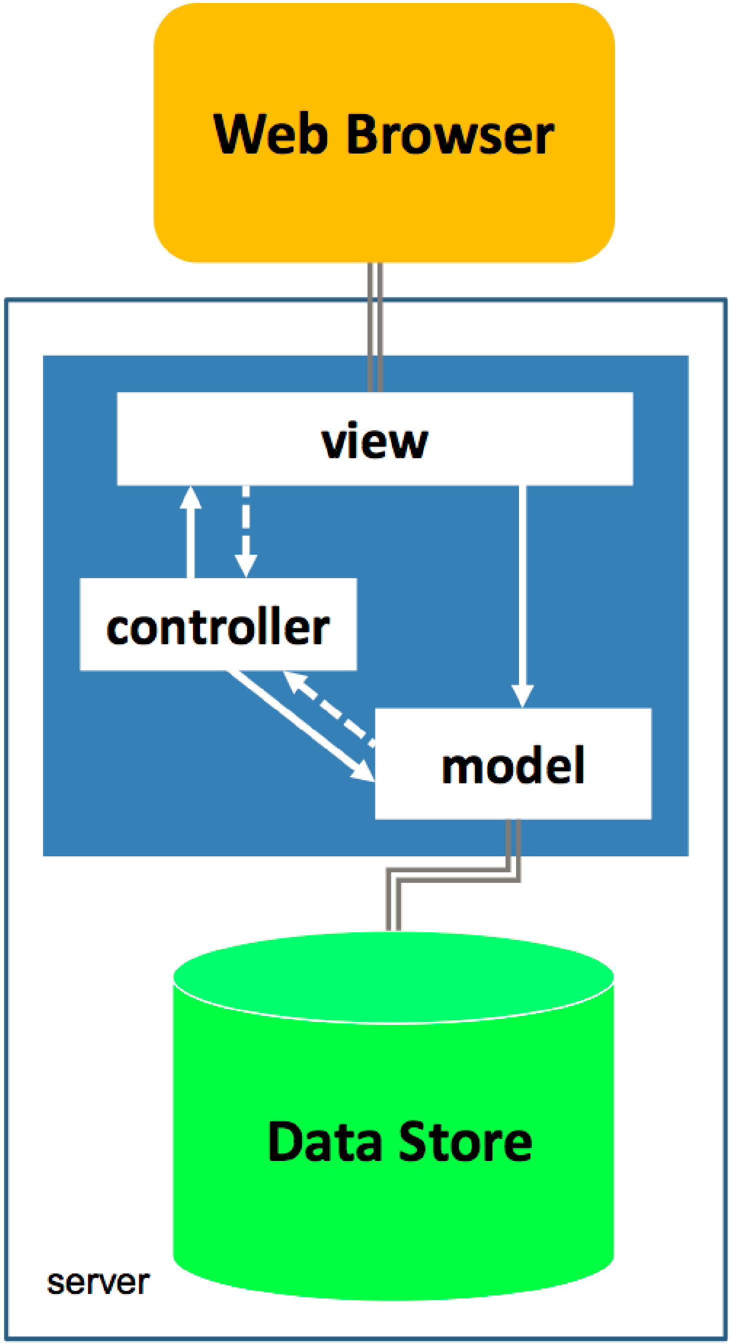
MVC FRAMEWORKS

MVC Pattern

- ServerSide Framework
- “Model 2” Architecture

Examples

- ASP. NET MVC Framework
- Struts, Spring MVC (Java)
- Ruby on Rails (Ruby)
- Django (Python)
- Grails (Groovy)

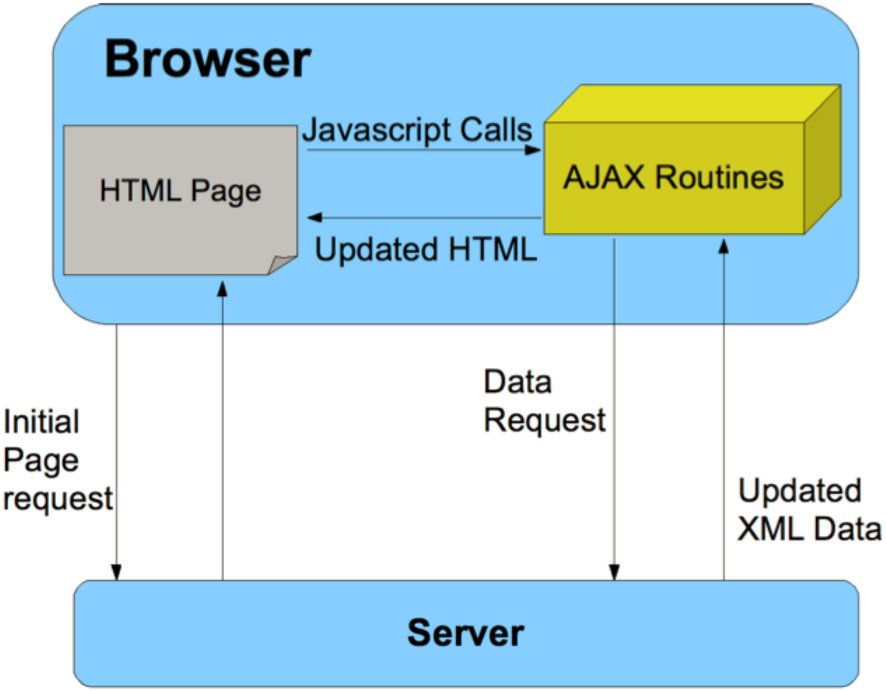


AJAX

- **A**synchronous **J**avaScript **A**nd **X**ML
- Not a programming language
- Dynamic content changes without reloading the entire page
- HTML/CSS + DOM +

XmlHttpRequest Object +

JavaScript + JSON/XML



PROCESS OF WEB APPLICATION

1. **Application Download**

Mobile code (JavaScript, HTML, Applets, Flash) download to the client (web browser)

2. **Presentation Flow**

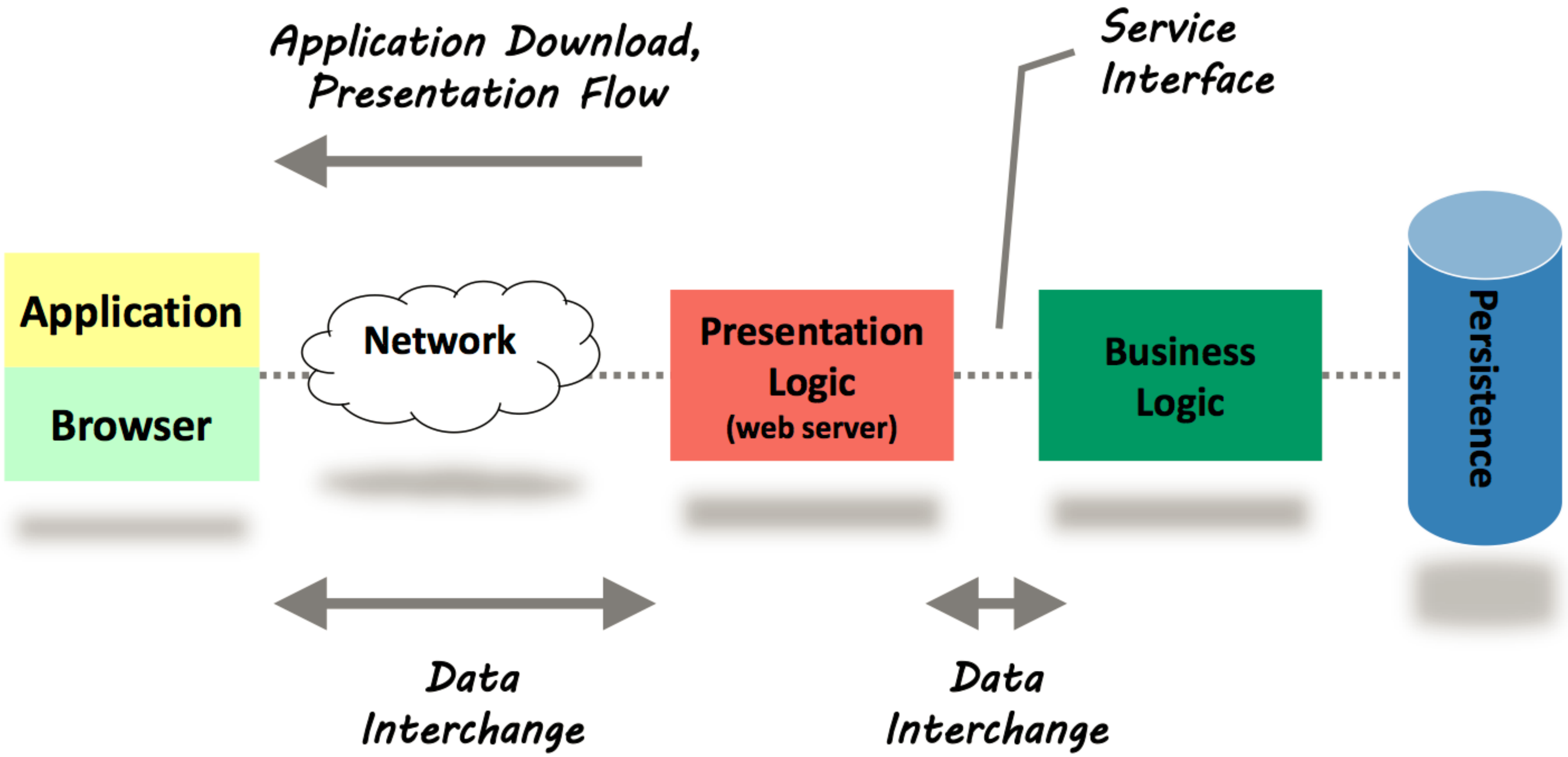
Dynamic visual rendering of the UI (screen changes, new

screens, etc.) in response to user input and data state changes

3. **Data Interchange**

The exchange of data between two software components or tiers (search, updates, retrieval, etc.)

WEB TEMPLATING ENGINE FRAMEWORK



CHARACTERISTICS

Tight coupling between Presentation Flow and Data Interchange (both in the web server)

- Triggering a Presentation Flow (web page update) in a web application initiates a Data Interchange operation
- Every Data Interchange operation results in a Presentation Flow operation

Presentation Flow and Data Interchange are *orthogonal* concerns that should be decoupled

- Separate concerns

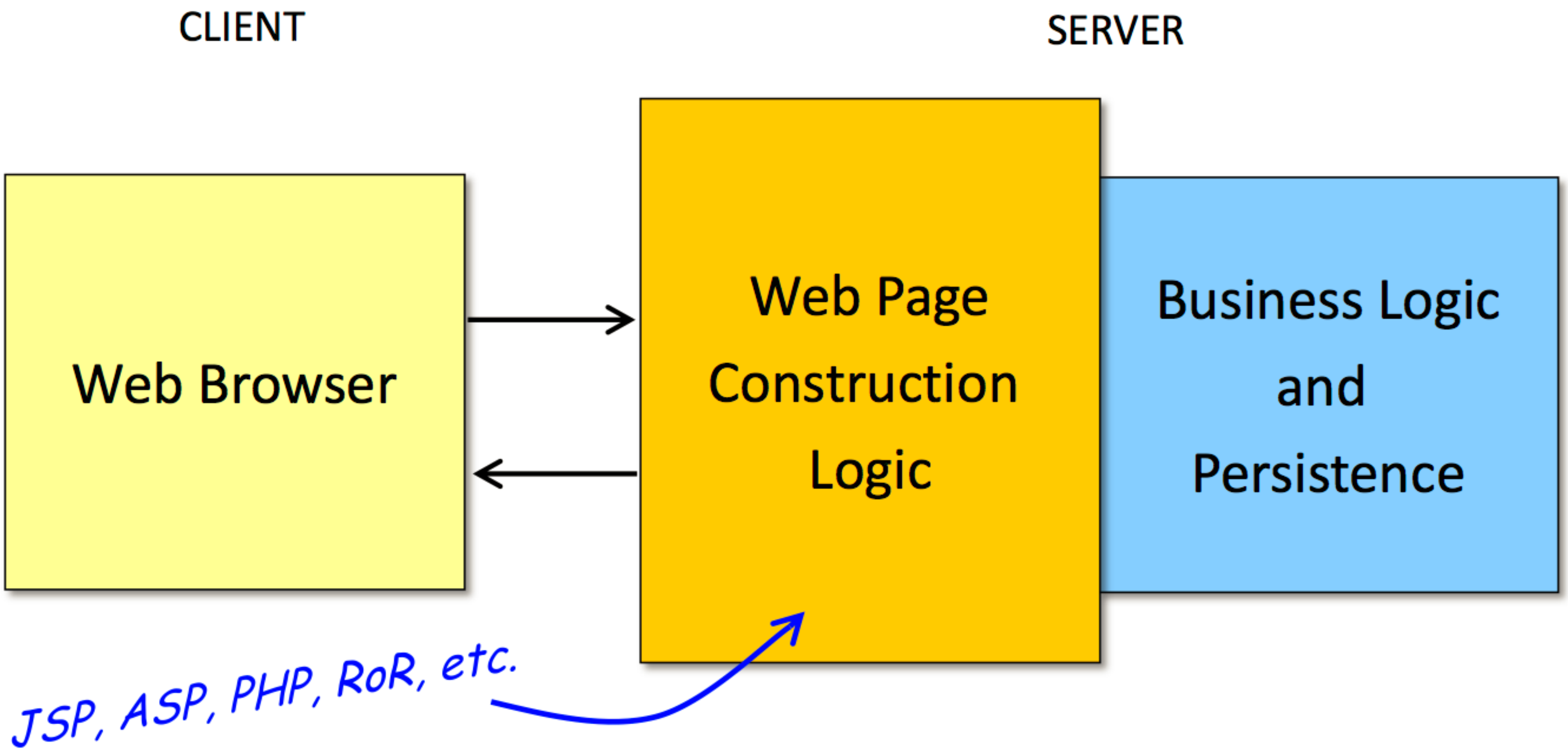
SERVICE ORIENTED FRONT END ARCHITECTURE (SOFEA)

Service **O**riented **F**ront **E**nd **A**rchitecture – Synonymous with "Single Page" Web Applications (SPA)

Life above the Service Tier

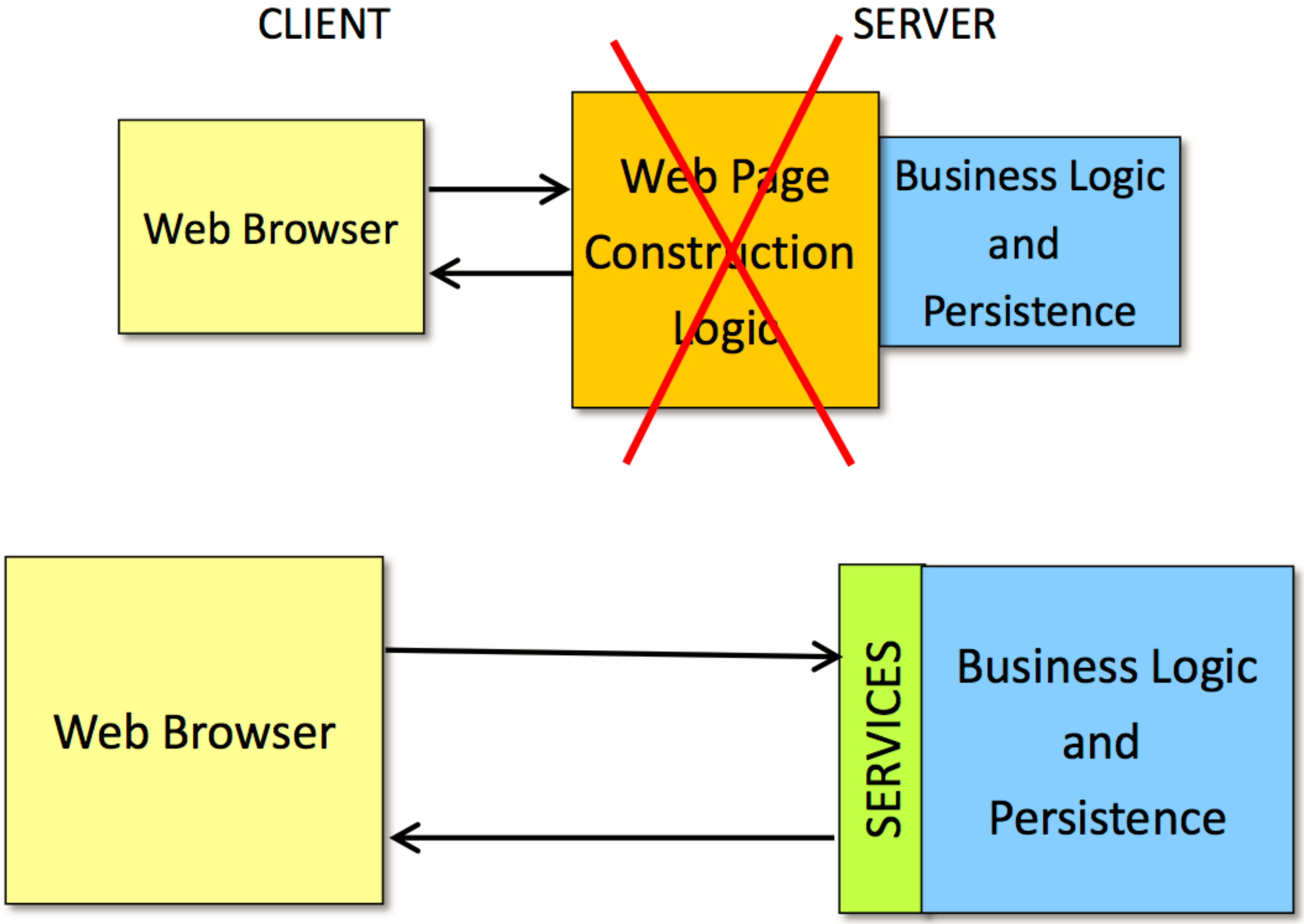
- How to Build Application Front-ends in a Service-Oriented World (*Ganesh Prasad, Rajat Taneja, Vikrant Todankar*)

LEGACY ARCHITECTURE



Typical Enterprise Web Application Architecture

SOFEA



CLIENT

SERVER

Web Browser

~~Web Page Construction Logic~~

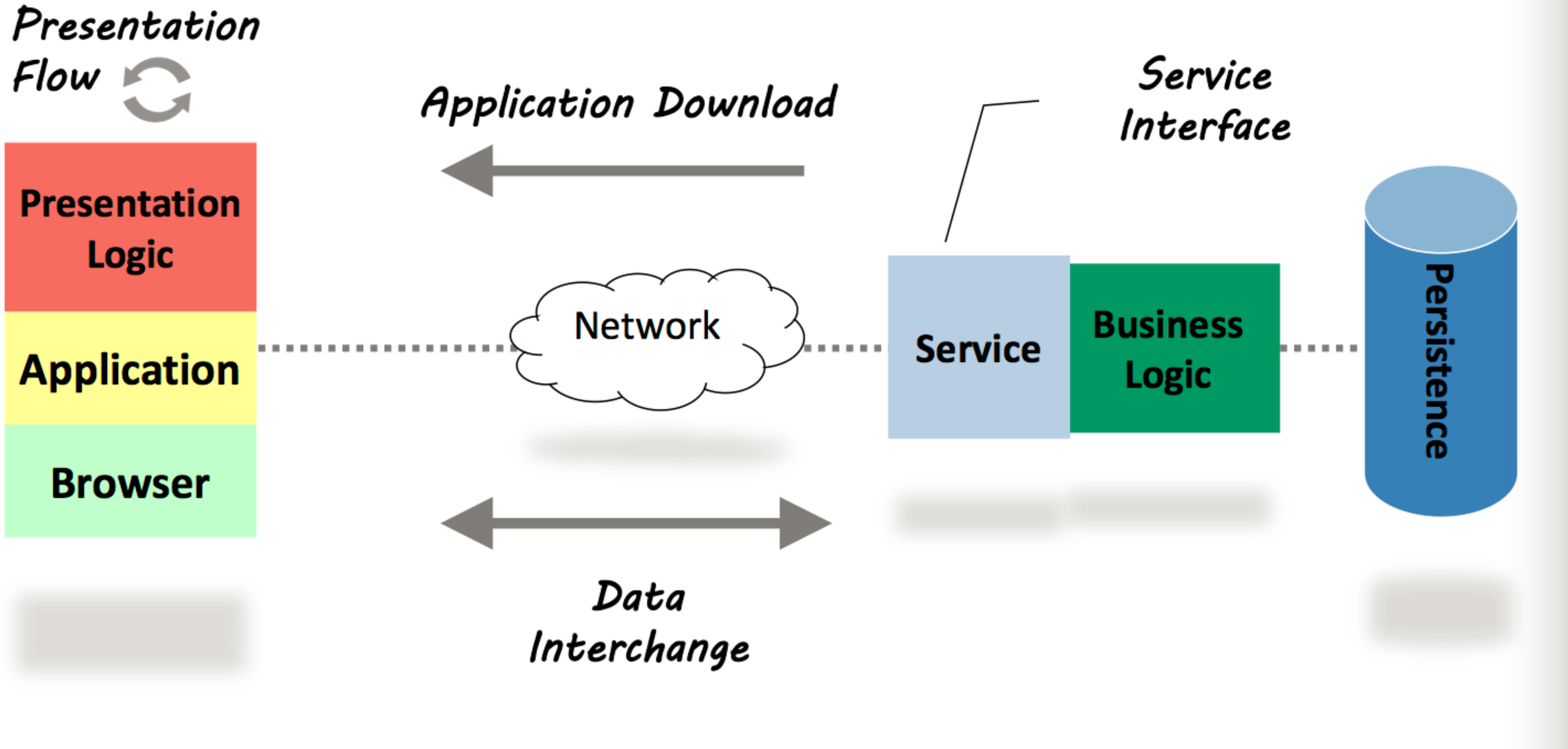
Business Logic and Persistence

Web Browser

SERVICES

Business Logic and Persistence

SOFEA PROCESS ALLOCATION



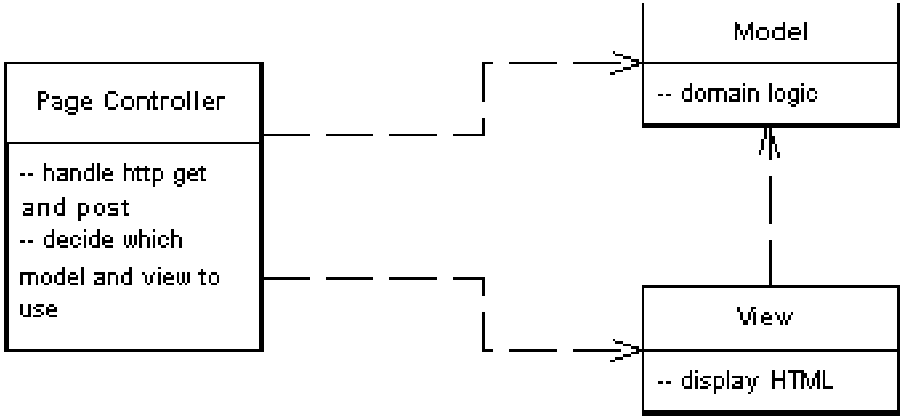
SOFEA PRINCIPLES

- Application Download, Data Interchange, and Presentation Flow must be *decoupled*
  - No part of the client should be evoked, generated or templated from the server-side.
- Presentation Flow is a *client-side* concern only
- All communication with the application server should be using services (REST, SOAP, etc.)
- The MVC design pattern belongs in the client, not the server

Going back to MVC frameworks based approach

CONTROLLERS

Page controller – an object that handles a request for a specific page or action on a Web site.



Page Controller

-- handle http get and post
-- decide which model and view to use

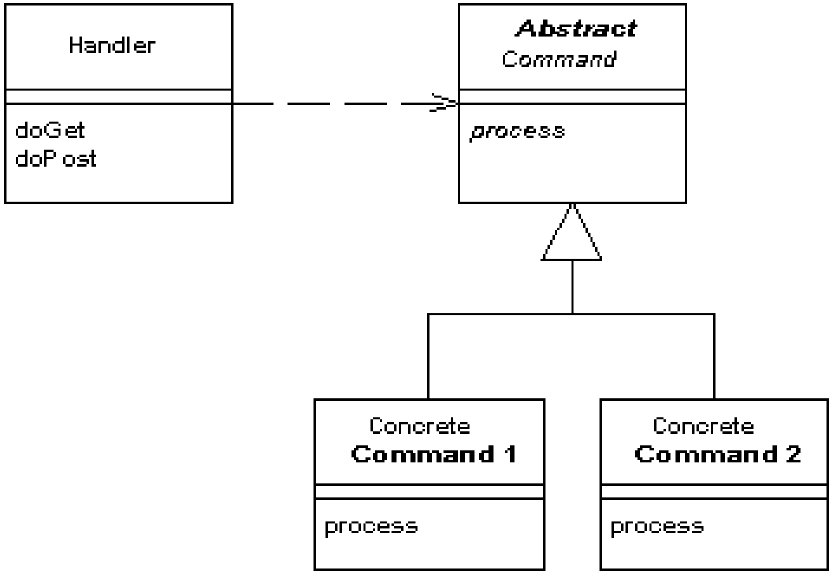
Model

-- domain logic

View

-- display HTML

Front controller – an object that handles all requests for a web site



Handler

doGet
doP ost

Abstract
Command

process

Concrete
Command 1

process

Concrete
Command 2

process

PAGE CONTROLLER

As Script

- Servlet or CGI program
- Web Applications that need logic and data

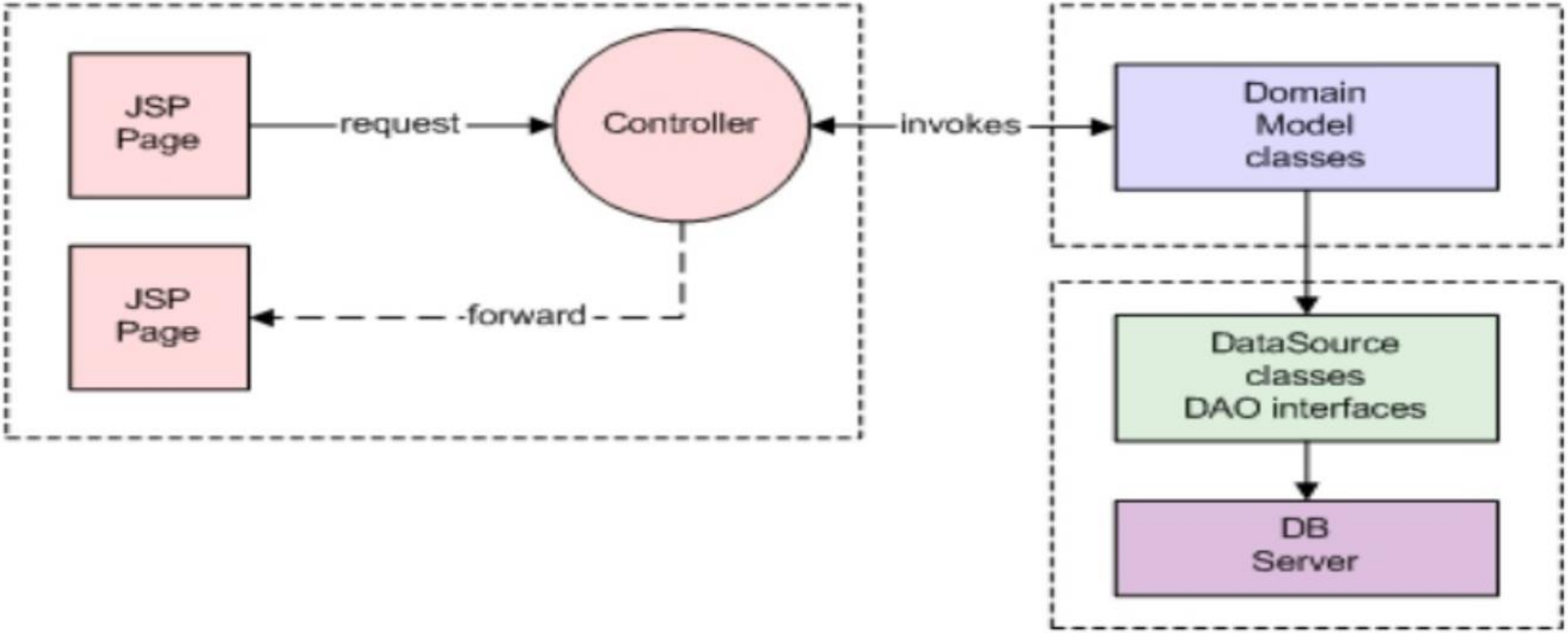
As Server Page

- ASP, PHP, JSP
- Use helpers to get data from the model
- Logic is simple to none
- Combines Page Controller + Template View

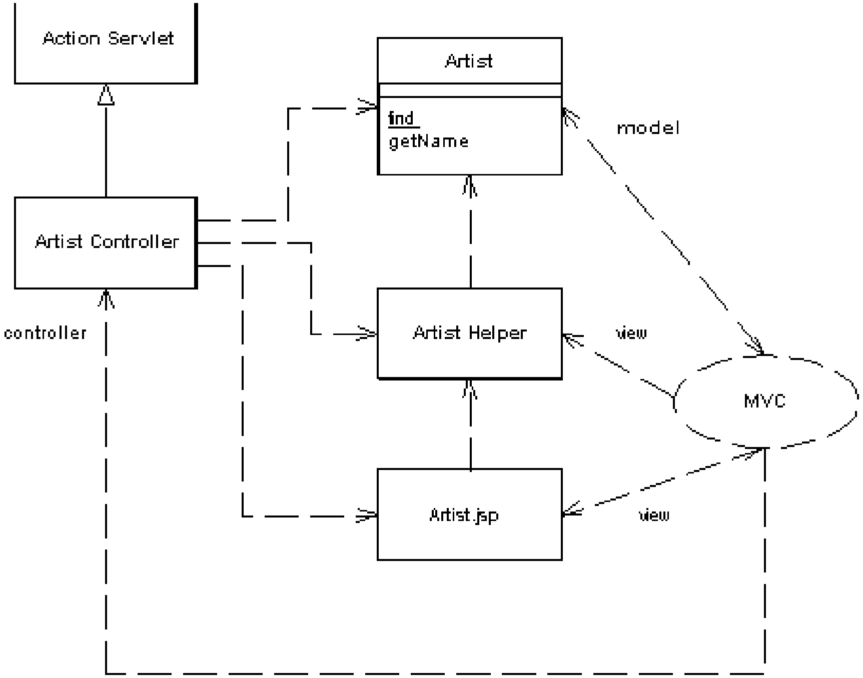
Basic responsibilities

- **Decode the URL** and extract all data for the action.
- **Create and invoke any model objects** to process the data. All relevant data from the HTML request should be passed to the model so that the model objects don’t need any connection to the HTML request.
- **Determine which view** should display the result page and forward the model information to it.

PAGE CONTROLLER



SERVLET CONTROLLER AND A JSP VIEW (JAVA)

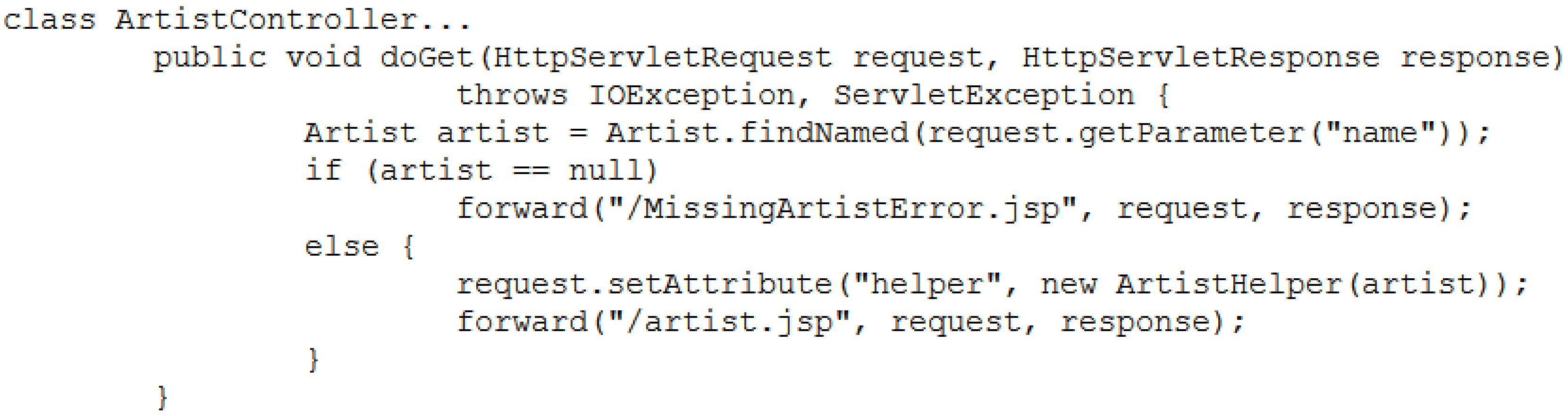


http://www.thingy.com/recordingApp/artist?name=danielaMercury.

In web.xml map /artist to a call to ArtistController

```
<servlet>
<servlet-name>artist</servlet-name>
<servlet-
class>actionController.ArtistController
</servlet-class>
</servlet>

<servlet-mapping>
<servlet-name>artist</servlet-name>
<url-pattern>/artist</url-pattern>
</servlet-mapping>
```

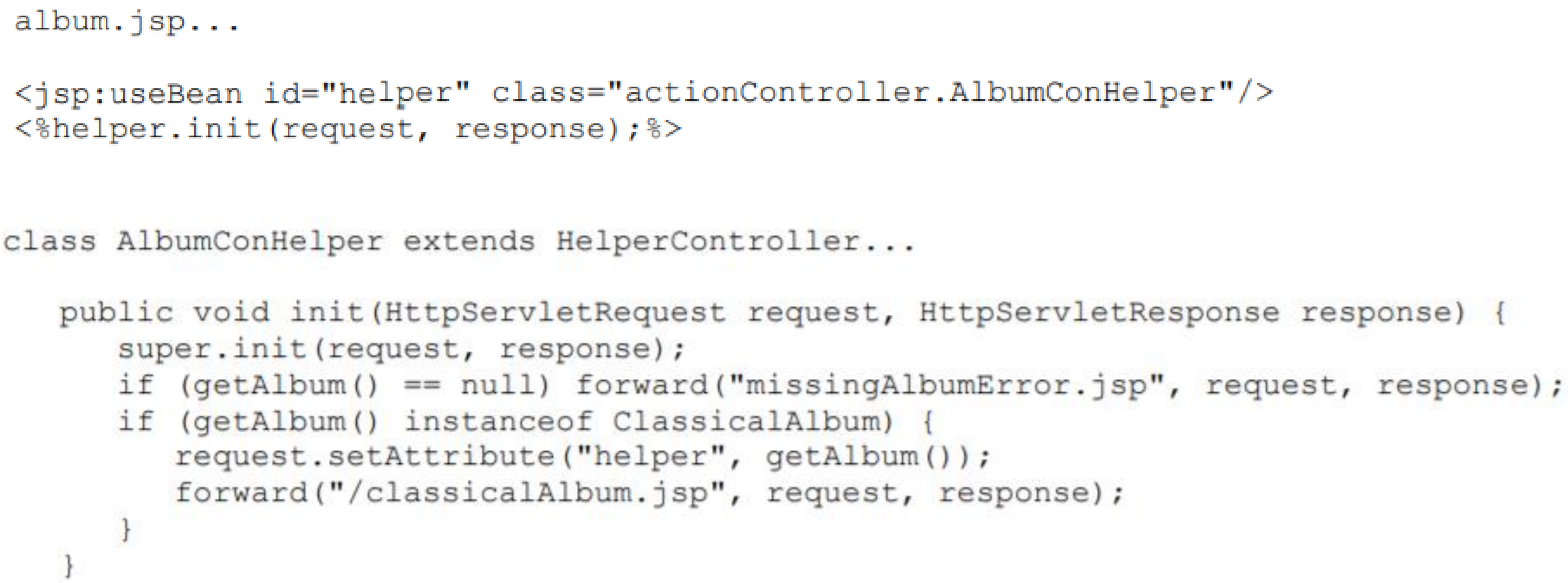


```
class ArtistController...
    public void doGet(HttpServletRequest request, HttpServletResponse response)
                throws IOException, ServletException {
            Artist artist = Artist.findNamed(request.getParameter("name"));
            if (artist == null)
                forward("/MissingArtistError.jsp", request, response);
            else {
                request.setAttribute("helper", new ArtistHelper(artist));
                forward("/artist.jsp", request, response);
            }
    }
```

JSP AS REQUEST HANDLER

Delegates control to the helper

The handler JSP is the default view



```
album.jsp...

<jsp:useBean id="helper" class="actionController.AlbumConHelper"/>
<%helper.init(request, response);%>
```

```
class AlbumConHelper extends HelperController...

   public void init(HttpServletRequest request, HttpServletResponse response) {
      super.init(request, response);
      if (getAlbum() == null) forward("missingAlbumError.jsp", request, response);
      if (getAlbum() instanceof ClassicalAlbum) {
         request.setAttribute("helper", getAlbum());
         forward("/classicalAlbum.jsp", request, response);
      }
   }
```

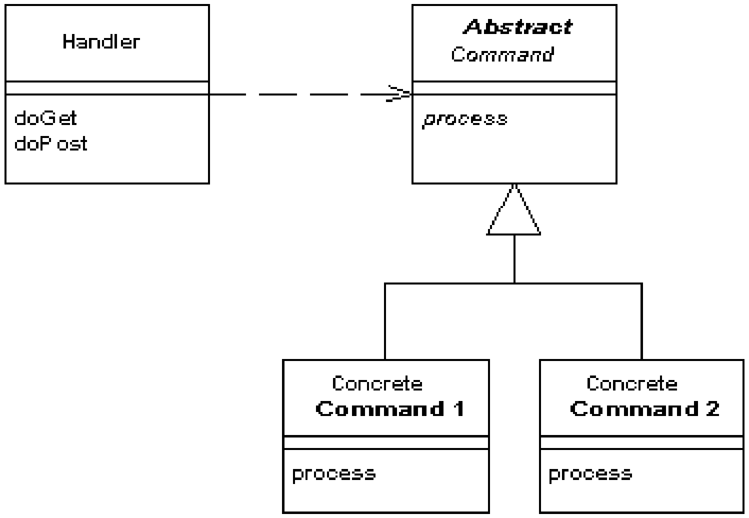
FRONT CONTROLLER

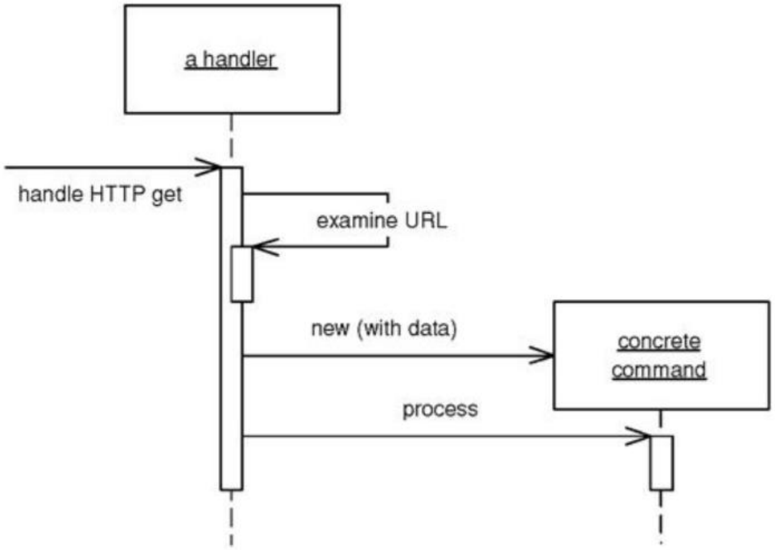
If many similar things are done when handling a request (i.e. security, internationalization, etc.)

One controller handles all requests

Usually handles in 2 phases:

- Request handling - a web handler (rather a class than a server page)
- Command handling - a hierarchy of commands (classes)





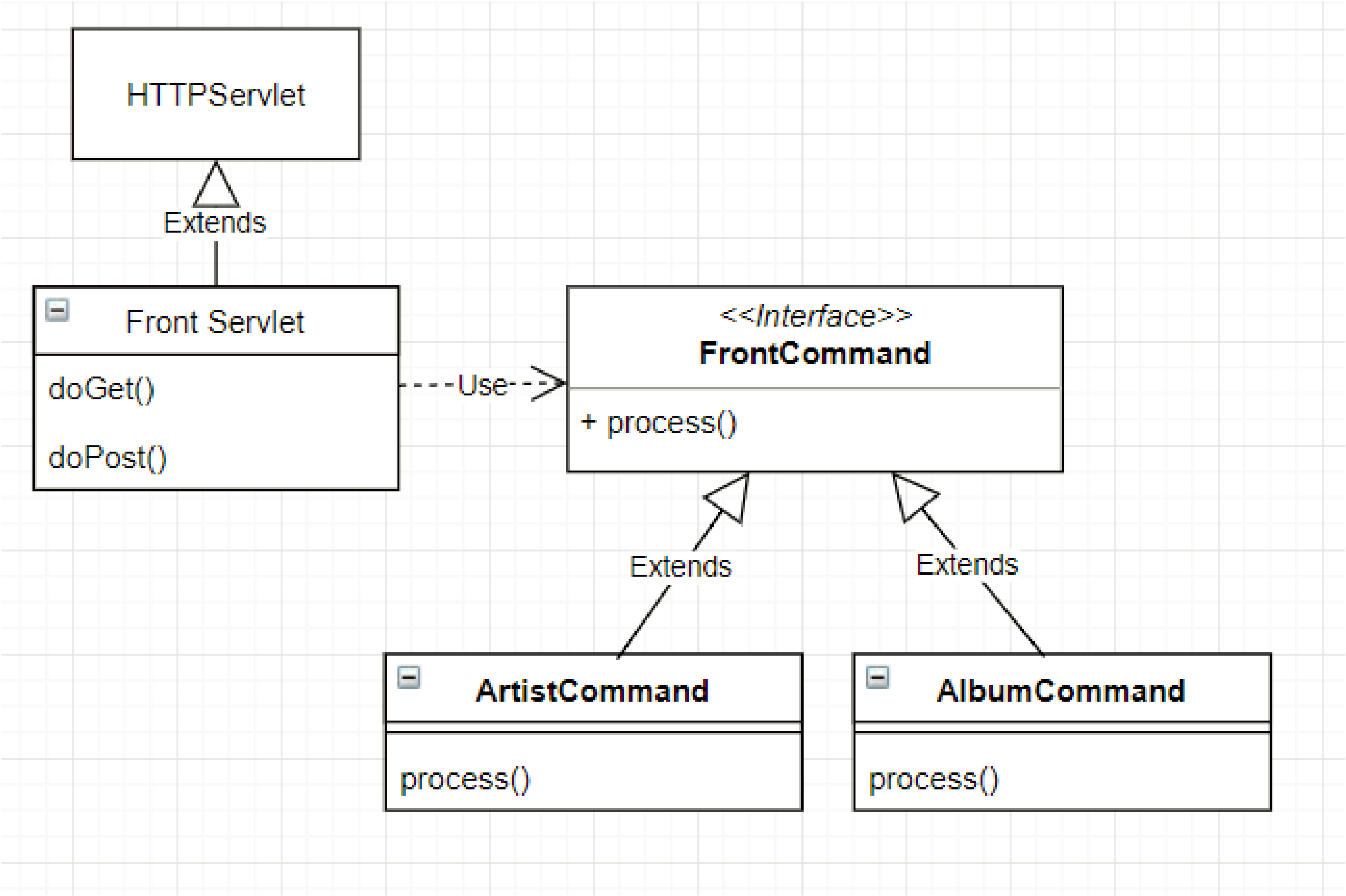
FRONT CONTROLLER

Handler decides what command:

- **statically**
  - parses the URL and uses conditional logic;
  - advantage of explicit logic,
  - compile time error checking on dispatch,
  - flexibility in URL look-up
- **dynamically**
  - takes a standard piece of the URL and uses dynamic instantiation to create a command class;
  - allows to add new commands without changing the Web handler;
  - can put the name of the command class into the URL or can use a properties file that binds URLs to command class names.

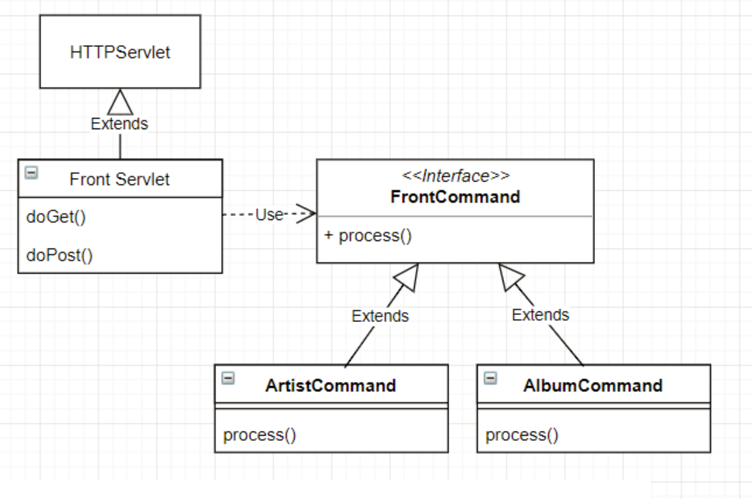
EXAMPLE

http://localhost:8080/isa/music?name=astor&command=Artist



EXAMPLE

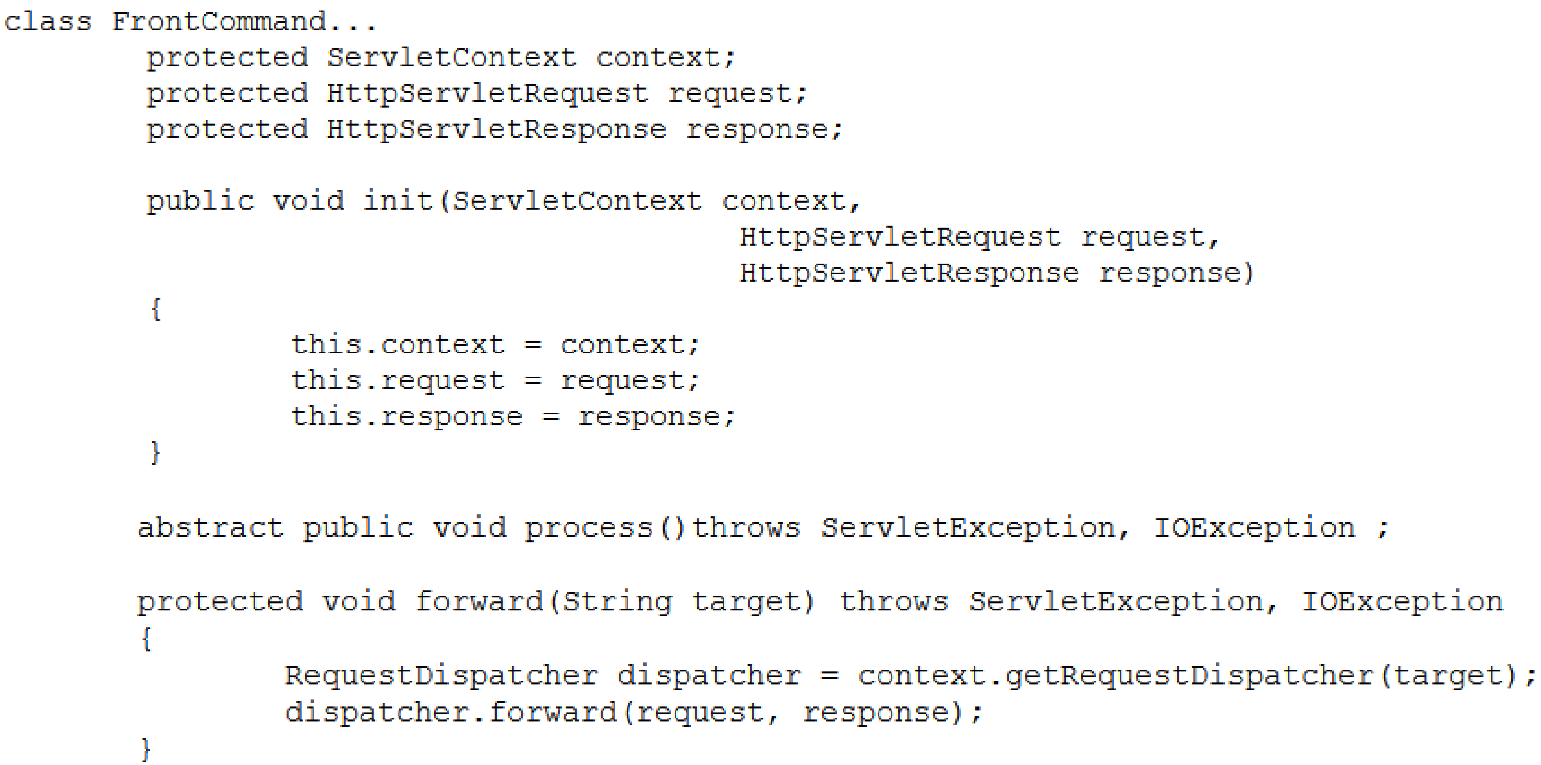
http://localhost:8080/isa/music?name=astor&command=Artist



```
class FrontServlet...
        public void doGet(HttpServletRequest request, HttpServletResponse response)
                        throws IOException, ServletException {
                FrontCommand command = getCommand(request);
                command.init(getServletContext(), request, response);
                command.process();
        }

        private FrontCommand getCommand(HttpServletRequest request) {
                try {
                        return (FrontCommand) getCommandClass(request).newInstance();
                } catch (Exception e) {
                        throw new ApplicationException(e);
                }
        }

        private Class getCommandClass(HttpServletRequest request) {
                Class result;
                final String commandClassName =
                                "frontController." + (String) request.getParameter("command") + "Command";
                try {
                        result = Class.forName(commandClassName);
                } catch (ClassNotFoundException e) {
                        result = UnknownCommand.class;
                }
                return result;
        }
```

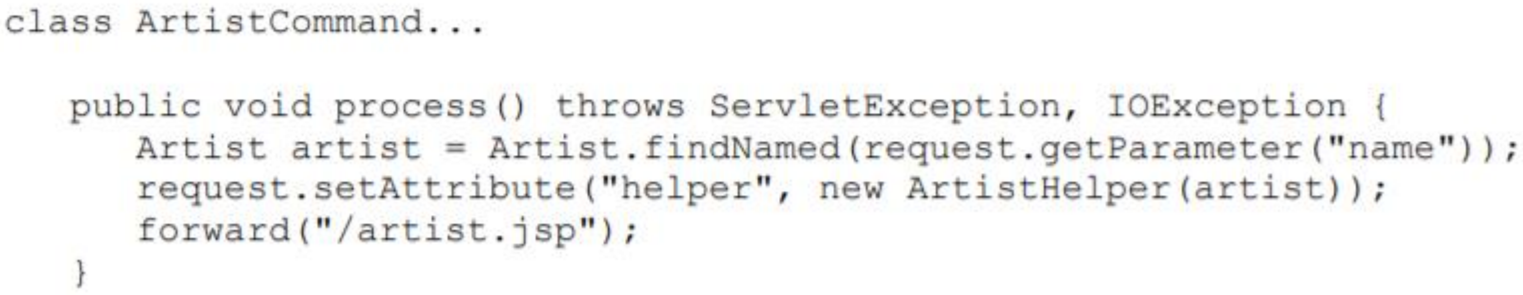


```
class FrontCommand...
        protected ServletContext context;
        protected HttpServletRequest request;
        protected HttpServletResponse response;

        public void init(ServletContext context,
                                         HttpServletRequest request,
                                         HttpServletResponse response)
        {
                this.context = context;
                this.request = request;
                this.response = response;
        }

        abstract public void process()throws ServletException, IOException ;

        protected void forward(String target) throws ServletException, IOException
        {
                RequestDispatcher dispatcher = context.getRequestDispatcher(target);
                dispatcher.forward(request, response);
        }
```



```
class ArtistCommand...

   public void process() throws ServletException, IOException {
      Artist artist = Artist.findNamed(request.getParameter("name"));
      request.setAttribute("helper", new ArtistHelper(artist));
      forward("/artist.jsp");
   }
```

DISCUSSION

- Only one Front Controller has to be configured into the Web server
- You can add new commands without changing anything.
- Because new command objects are created with each request, it is thread safe (provided model objects are not shared!).
- Both the handler and the commands are part of the controller. As a result the commands can (and should) choose which view to use for the response. The only responsibility of the handler is in choosing which command to execute.
- Re-factor code better in command hierarchy

DISCUSSION

Page Controller:

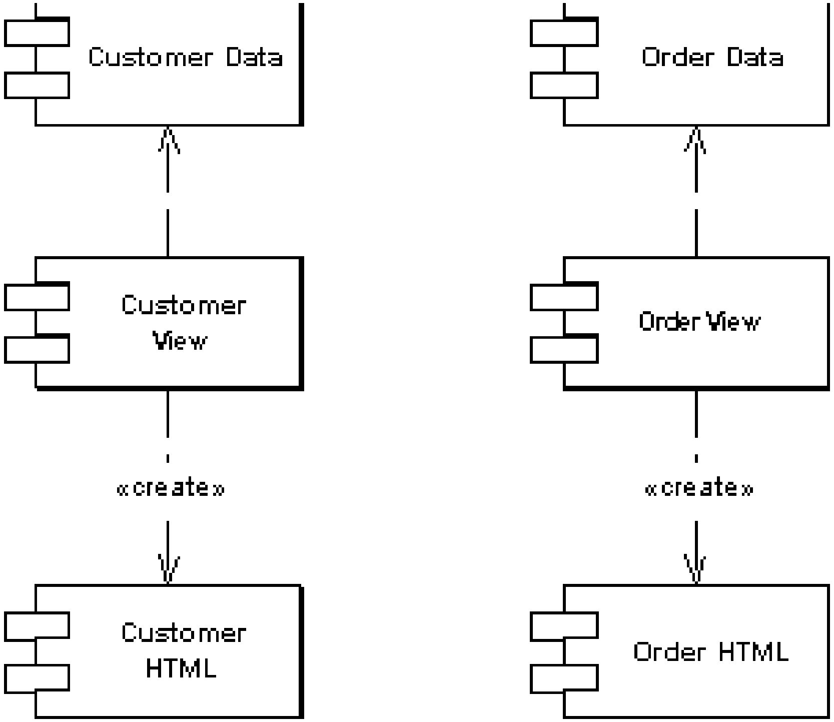
- simple controller logic
- a natural structuring mechanism where particular actions are handled by particular server pages or script classes.

Front Controller:

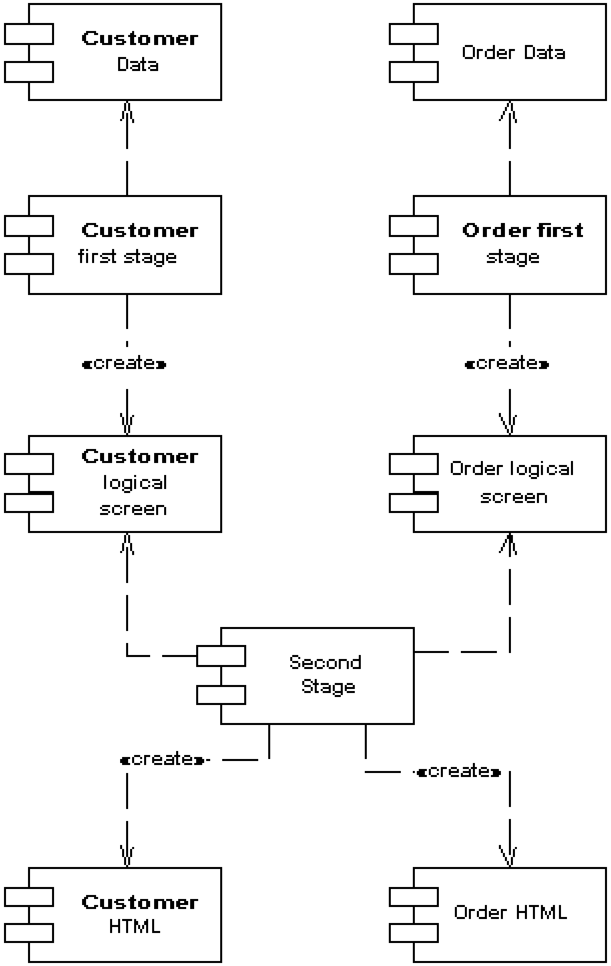
- greater complexity;
- handles duplicated features (i.e. security, internationalization, providing particular views for certain kinds of users) in one place.
- single point of entry for centralized logic

VIEW

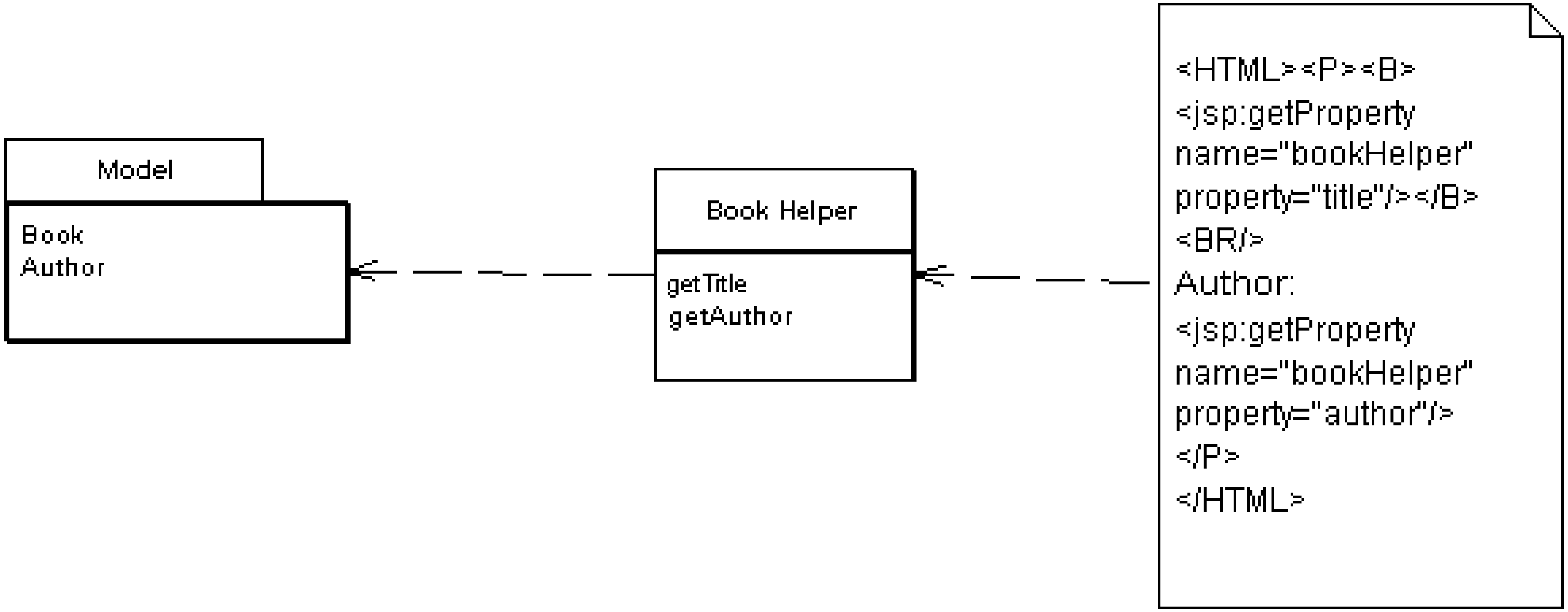
Single step stage



Two step stage



TEMPLATE VIEW



```
<HTML><P><B>
<jsp:getProperty
name="bookHelper"
property="title"/></B>
<BR/>
Author:
<jsp:getProperty
name="bookHelper"
property="author"/>
</P>
</HTML>
```

- Embed markers into a static HTML page when it's written
- When the page is used to service requests, the markers are replaced by the results of some computation
- **server pages**:
  - ASP, JSP, or PHP.
  - allow to embed arbitrary programming logic, referred to as **scriptlets**, into the page.

CONDITIONAL DISPLAY

```
<IF condition = "$pricedrop > 0.1"> ...show some stuff </IF>
```

Templates become programming languages

⇒ Move the condition to the helper to generate the content

⇒ What if the content should be displayed but in different ways?

- Helper generates the markup
- OR use focused tags:

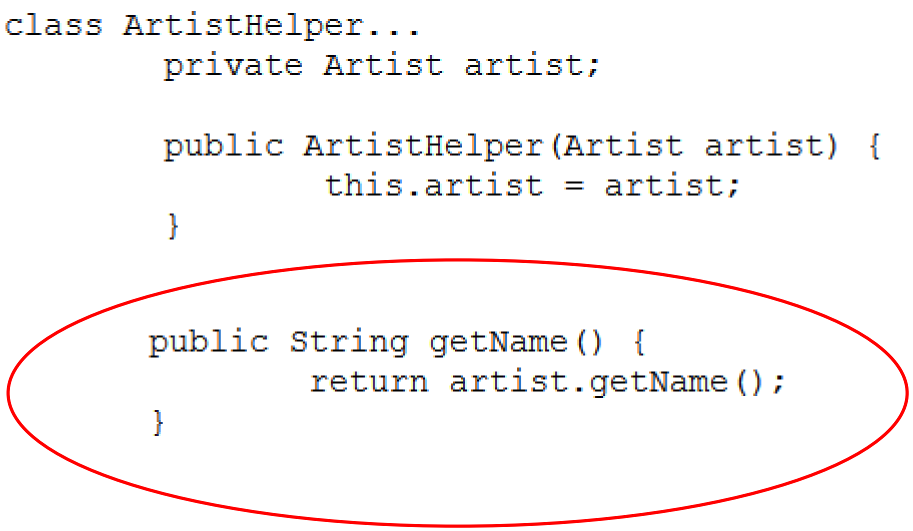
```
<IF expression = "isHighSelling()"><B></IF><property name =
"price"/><IF expression = "isHighSelling()"></B></IF>
```

replaced by

```
<highlight condition = "isHighSelling" style =
"bold"><property name = "price"/></highlight>
```

JSP TEMPLATE VIEW (SEE PAGE CONTROLLER)

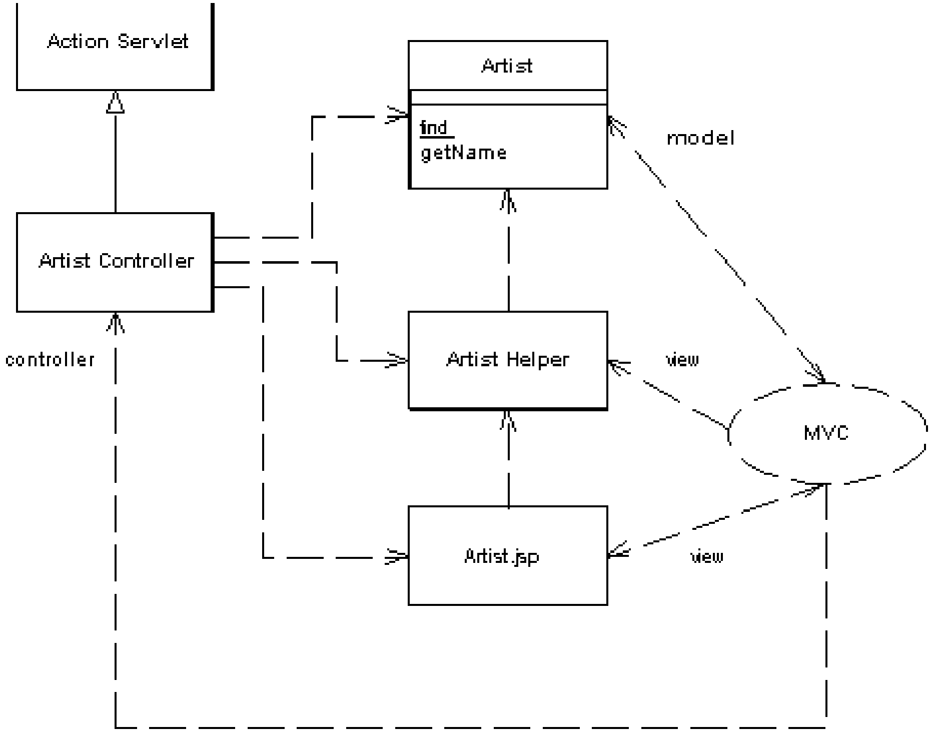
```
<jsp:useBean id="helper" type="actionController.ArtistHelper" scope="request"/>
```



```
class ArtistHelper...
        private Artist artist;

        public ArtistHelper(Artist artist) {
                this.artist = artist;
        }

        public String getName() {
                return artist.getName();
        }
```



To access the information from the helper

```
<B> <%=helper.getName()%></B>
```

or

```
<B><jsp:getProperty name="helper" property="name"/></B>
```

SHOW A LIST OF ALBUMS FOR AN ARTIST

```
<UL>
<%
        for (Iterator it = helper.getAlbums().iterator(); it.hasNext();) {
                Album album = (Album) it.next();%>
        <LI><%=album.getTitle()%></LI>

<%      }   %>
</UL>
```

🚫

```
class ArtistHelper...
        public String getAlbumList() {
                StringBuffer result = new StringBuffer();
                result.append("<UL>");
                for (Iterator it = getAlbums().iterator(); it.hasNext();) {
                        Album album = (Album) it.next();
                        result.append("<LI>");
                        result.append(album.getTitle());
                        result.append("</LI>");
                }
                result.append("</UL>");
                return result.toString();
        }

        public List getAlbums() {
                return artist.getAlbums();
        }
```

🙂

```
<UL><tag:forEach host = "helper" collection = "albums" id = "each">
 <LI><jsp:getProperty name="each" property="title"/></LI>
</tag:forEach></UL>
```

DISCUSSION

Benefits:

- Compose the structure of the page based on the template
- Separate design from code (helper)

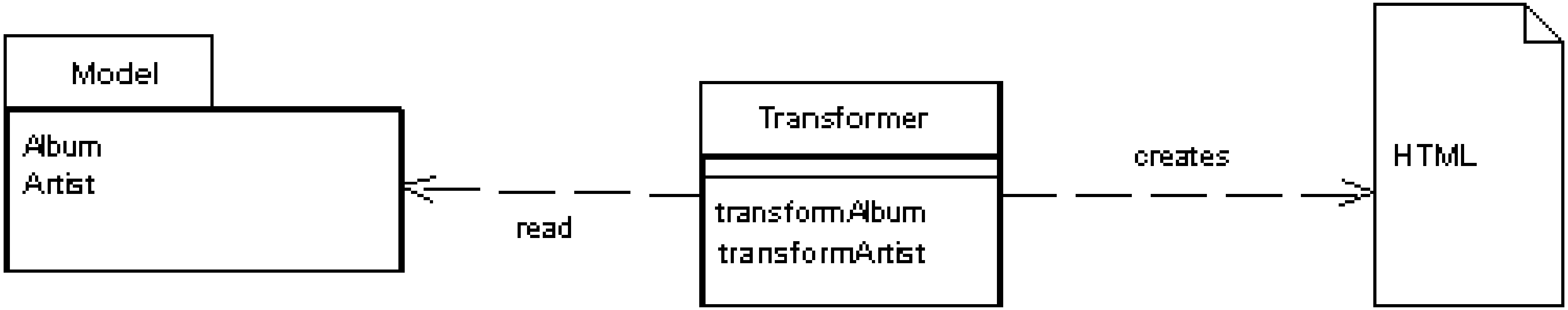
Liabilities

- Common implementations make it too easy to put complicated logic onto the page => hard to maintain
- Harder to test than Transform View

TRANSFORM VIEW

Input: Model

Output: HTML



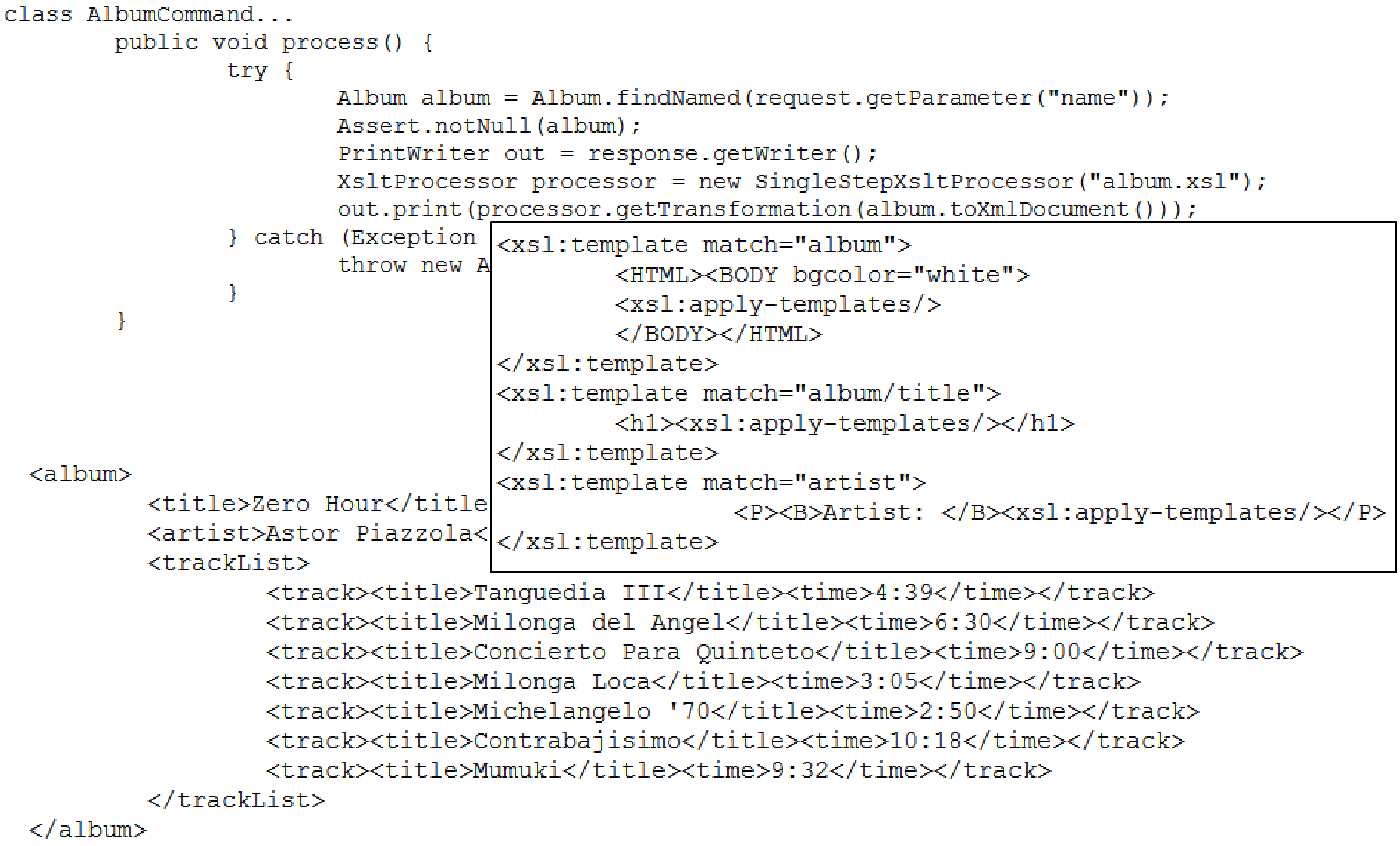
can be written in any language, yet the dominant choice is XSLT (EXtensible Stylesheet Language Transformation).

Input: XML

XML data can be returned as:

- natural return type
- output type which can be transformed to XML automatically
- Data Transfer Object, that can serialize as XML

TRANSLATED IN CODE



```
class AlbumCommand...
        public void process() {
                try {
                        Album album = Album.findNamed(request.getParameter("name"));
                        Assert.notNull(album);
                        PrintWriter out = response.getWriter();
                        XsltProcessor processor = new SingleStepXsltProcessor("album.xsl");
                        out.print(processor.getTransformation(album.toXmlDocument()));
                } catch (Exception
                        throw new A
                }
        }
```

```
<xsl:template match="album">
        <HTML><BODY bgcolor="white">
        <xsl:apply-templates/>
        </BODY></HTML>
</xsl:template>
<xsl:template match="album/title">
        <h1><xsl:apply-templates/></h1>
</xsl:template>
<xsl:template match="artist">
                <P><B>Artist: </B><xsl:apply-templates/></P>
</xsl:template>
```

```
<album>
        <title>Zero Hour</title
        <artist>Astor Piazzola<
        <trackList>
                <track><title>Tanguedia III</title><time>4:39</time></track>
                <track><title>Milonga del Angel</title><time>6:30</time></track>
                <track><title>Concierto Para Quinteto</title><time>9:00</time></track>
                <track><title>Milonga Loca</title><time>3:05</time></track>
                <track><title>Michelangelo '70</title><time>2:50</time></track>
                <track><title>Contrabajisimo</title><time>10:18</time></track>
                <track><title>Mumuki</title><time>9:32</time></track>
        </trackList>
</album>
```

ADVANTAGES

- Portability: use the same XSLT with XMLs from J2EE or .NET

- Avoid too much logic in view, hence focus on the HTML rendering

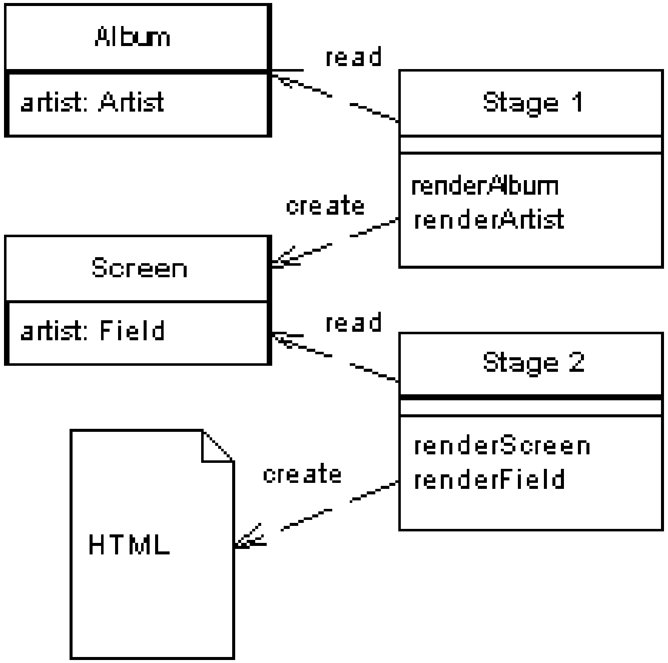
- Easier to test: run the Transform View and capture the output for testing.

- Easier to change the appearance of a Web site: change the common transforms.

TWO STEP VIEW

Multi-page application:

- transforms the model data into a logical presentation without any specific formatting
- converts that logical presentation with the actual formatting needed.



HOW TO DO IT

two-step XSLT:

- domain-oriented XML => presentation-oriented XML,
- presentation-oriented XML => HTML.

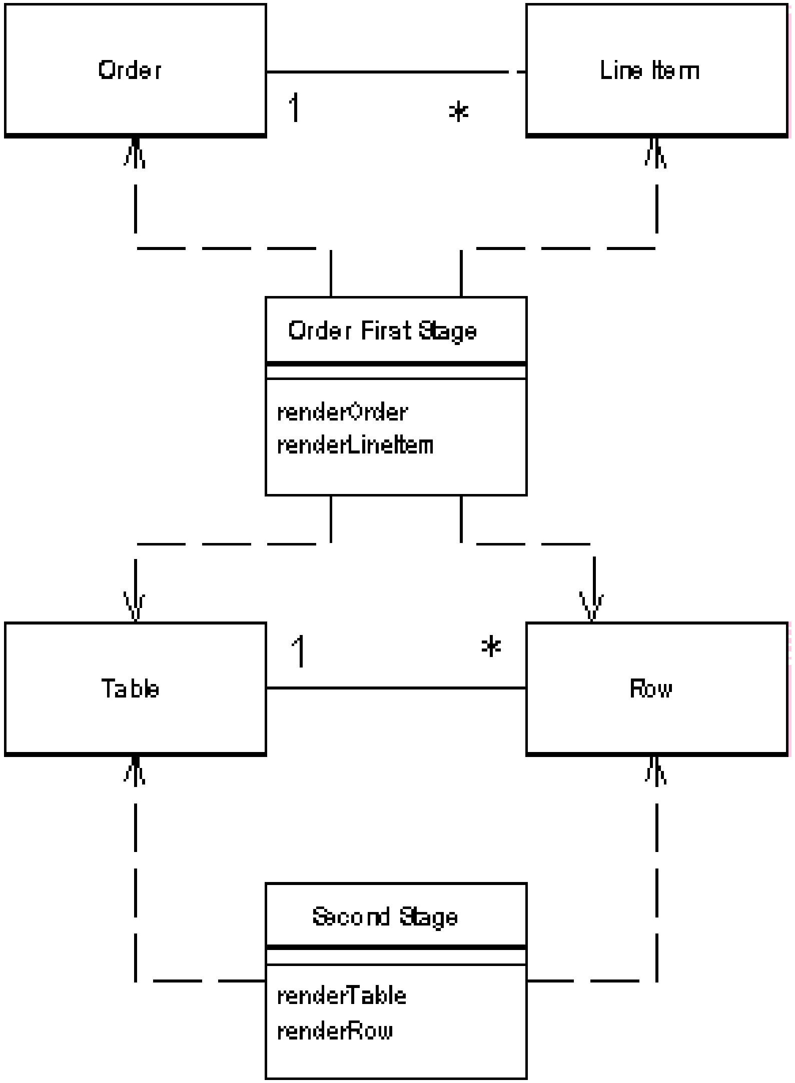
presentation-oriented structure as a set of classes (table/row class):

- domain information instantiates T/R classes.
- renders the T/R classes into HTML
  - each presentation-oriented class generates HTML for itself or
  - having a separate HTML renderer class

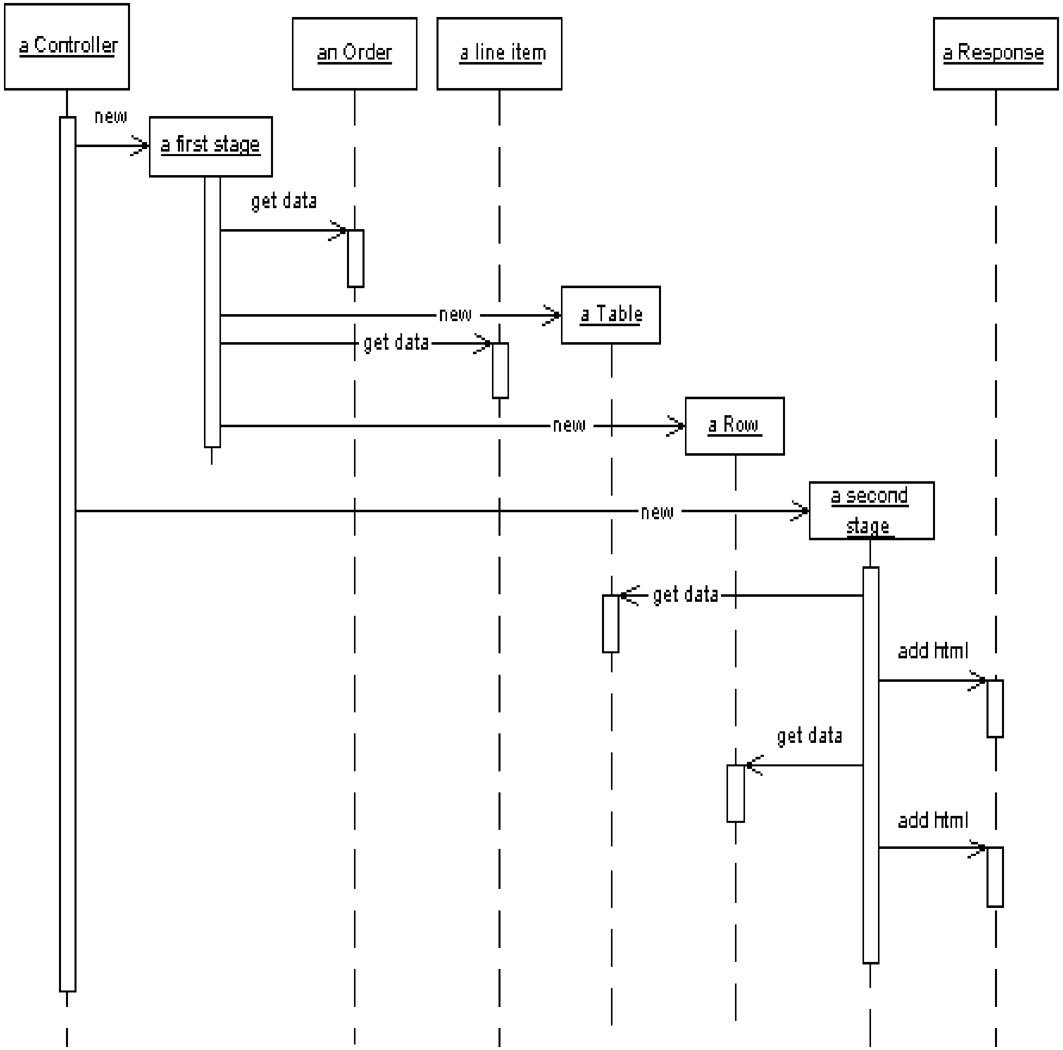
Template View based approach

- The template system converts the logical tags into HTML.

EXAMPLE

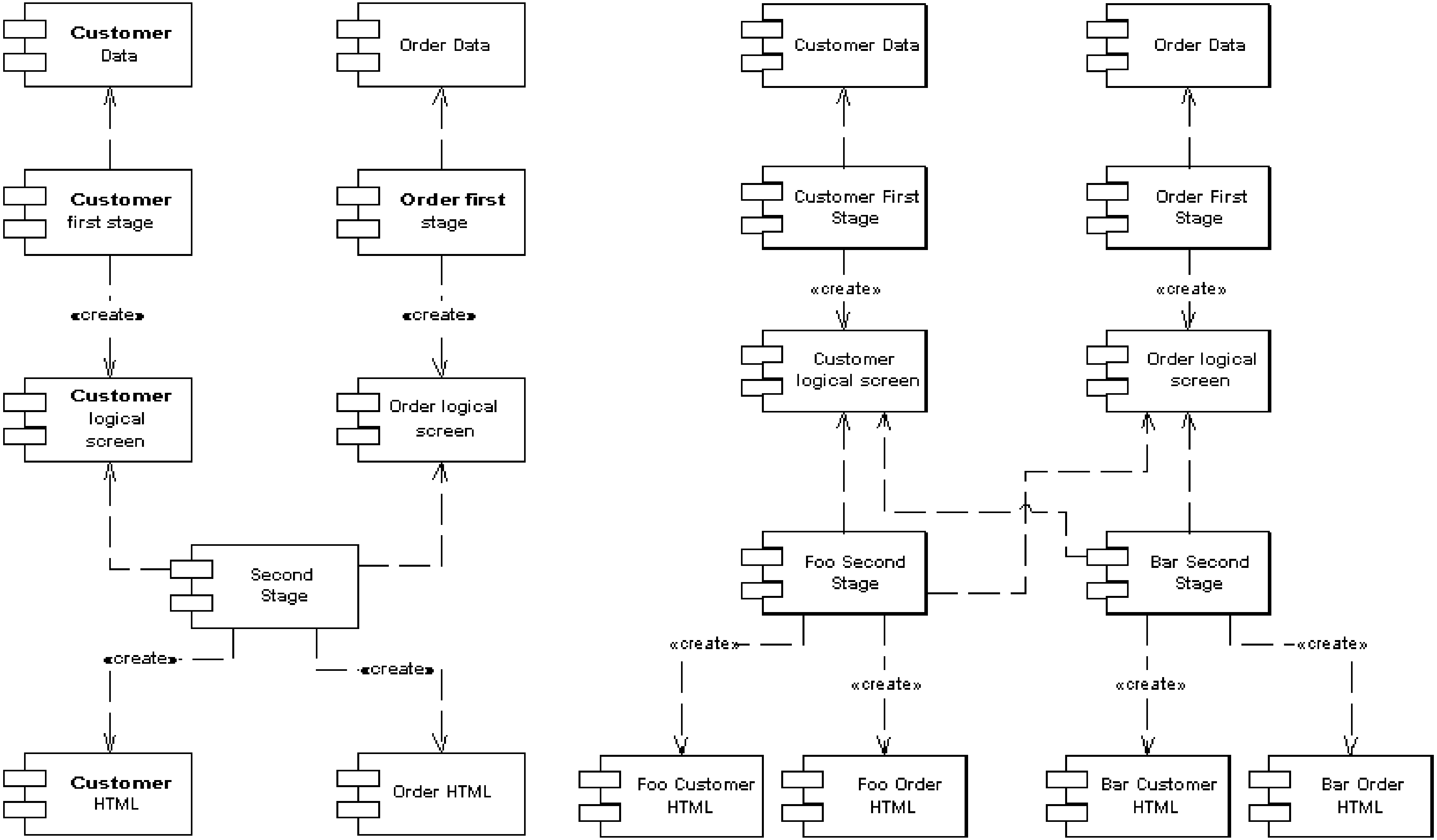


Order
1
*
Line Item
Order First Stage
renderOrder
renderLineItem
Table
1
*
Row
Second Stage
renderTable
renderRow



a Controller
an Order
a line item
a Response
new
a first stage
get data
new
a Table
get data
new
a Row
new
a second stage
get data
add html
get data
add html

ONE VS. TWO APPEARANCES



DISCUSSION

Advantages:

- Two step view solves the difficulty with Transform view w.r.t. **multiple transforms** module & global changes.
- If the website has **multiple appearances/themes,** the complexity is higher. With two step view, the issue is resolved and the advantage is compounded with multiple pages/themes.

Liabilities:

- **Hard to find enough commonality** between the screens to get a simple enough presentation-oriented structure
- **Not for designers/non-programmers.** Programmers have to write code for different rendering.
- **Harder** programming model **to learn**
- Complexity increases if **multiple devices** have to be supported.

NEXT TIME

- Design Patterns
- What about the Midterm exam?
  - Me: post some example questions
  - You: solve at home -> post solutions in Moodle
  - We: discuss the solutions/questions
  - When?
    - Next week (no Easter break)
    - After Easter break