Outline

- Introduction
  - The Layering Technique
- Layers in Web Applications
  - The Two-Layer Model
  - The Three-Layer Model
- Web Presentation
- View Patterns
- Summary
Introduction

• **Architecture** is a shared understanding of a system's design by the expert developers.

• It has two major elements
  – the highest-level breakdown of a system into its parts and their interaction
  – decisions that should be made right early on because they are hard to change

• There is no unique way to state a system's architecture
  – there are multiple architectures in a system
Web Applications

- Web applications are commonly categorized as enterprise applications, as they usually involve:
  - persistent data
  - a large amount of data
  - concurrent access to data
  - a lot of (web) user interface screens
  - integrated with other enterprise applications
  - complex business logic
Layering

- **Layering** is one of the most common techniques for breaking apart a complicated system
- **Examples:**

<table>
<thead>
<tr>
<th>Application</th>
<th>FTP</th>
</tr>
</thead>
<tbody>
<tr>
<td>OS</td>
<td>TCP</td>
</tr>
<tr>
<td>Device Driver</td>
<td>IP</td>
</tr>
<tr>
<td>Device</td>
<td>Ethernet</td>
</tr>
</tbody>
</table>
Example: The OSI Model

OSI Model

Layer

Data

Application
Network Process to Application

Presentation
Data Representation and Encryption

Session
Interhost Communication

Transport
End-to-End Connections and Reliability

Network
Path Determination and IP (Logical Addressing)

Data Link
MAC and LLC (Physical addressing)

Physical
Media, Signal, and Binary Transmission

Host Layers

Data

Segments

Packets

Frames

Bits
Layering Principles

- In a layered system, we imagine subsystems arranged as a layer cake, where each layer rests on a lower layer.
- The higher layer uses various services defined by the lower layer, but the lower layer is unaware of the higher layer.
- Each layer usually hides its lower layers from the layers above.
  - e.g., layer 4 uses the services of layer 3, which uses the services of layer 2, but layer 4 is unaware of layer 2.
Layering: Pros

- You can understand a single layer as a coherent whole without knowing much about the other layers
- You can substitute layers with alternative implementations of the same basic services
- You minimize dependencies between layers
- Layers make good places for standardization
- Once you have a layer built, you can use it for many higher-level services
Layering: Cons

- Layers encapsulate some, but not all things well
  - As a result, you sometimes get cascading changes
  - Ex: adding a field that needs to display on the UI, must be in the database, and every layer in between

- Extra layers can harm performance
  - At every layer things typically need to be transformed from one representation to another
  - However, a layer that controls transactions can be optimized and will then make everything faster!
Layers in Web Applications
The Two-Layer Model

• The notion of layers became more apparent in the '90s with the rise of client–server systems
• The client held the user interface and other application code, and the server was usually a relational database
• The problem comes with domain logic: business rules, validations, calculations, and the like
Where to Put Domain Logic?

- **On the client**
  - embed logic into the UI screens
  - this makes it easy to duplicate code
  - simple changes results in hunting down similar code in many screens

- **On the server**
  - put logic in the database as stored procedures
  - stored procedures give limited structuring mechanisms, which leads to awkward code
  - removes the option of changing the database vendor
The Three-Layer Model

- At the same time that client–server was gaining popularity, the object-oriented world was rising.
- The object community had an answer to the problem of domain logic: Move to a three-layer system.
- In this approach you have a presentation layer for your UI, a domain layer for your domain logic, and a data source.
- This way you could move all of that intricate domain logic out of the UI and put it into a layer where you could structure it properly with objects.
Three Principal Layers

- **Presentation**
  - provision of services, display of information, handling interactions

- **Domain Logic**
  - logic that is the real point of the system

- **Data Source**
  - communication with databases, messaging systems, transaction managers, other packages
Dependency Rules

• The domain and data source should never be dependent on the presentation
  – There should be no subroutine call from the domain or data source code into the presentation code
  – This rule makes it easier to substitute different presentations on the same foundation and makes it easier to modify the presentation without serious ramifications deeper down

• The relationship between the domain and the data source depends upon the architectural patterns used for the data source
Separation Test

- It is usually difficult to recognize what is domain logic and what is other forms of logic.
- An informal test is to imagine adding a radically different layer to an application, such as a command-line interface to a Web application.
- If there's any functionality you have to duplicate, that's a sign of where domain logic has leaked into the presentation.
- Similarly, do you have to duplicate logic to replace a relational database with an XML file?
Where to Run Layers?

- **On the server**
  - pros: ease of maintenance, upgrade and fix
  - cons: needs a server roundtrip for every request

- **On the client**
  - pros: responsiveness and disconnected operation
  - cons: challenges regarding disconnected operation

- **Hybrid**
  - data source on the server
  - presentation either on the client (for rich-clients) or on the sever for Web interfaces
  - domain logic on either side or split on both
Web Presentation
Web Interfaces

• One of the biggest changes to enterprise applications in the past decade has been the rise of Web-browser-based user interfaces

• Main advantages:
  – no client software to install
  – a common UI approach
  – easy universal access
  – many environments to easily build Web apps
Model-View-Controller

- Model-View-Controller (MVC) is a software architecture pattern which separates the representation of information from the user's interaction with it.
MVC Components

- The **model** consists of application data, business rules, logic, and functions.
- A **view** can be any output representation of data, such as a chart or a diagram.
- The **controller** mediates input, converting it to commands for the model or view.
View Patterns
View Patterns

• There are three patterns on the view side
  – Transform View
  – Template View
  – Two-Step View
Template View

- **Template View** allows you write the presentation in the structure of the page and embed markers into the page to indicate where dynamic content needs to go.

- Some popular platforms such as PHP, ASP, and JSP are based on this pattern.

- While being powerful and flexible; sadly, it also leads to very messy code that's difficult to maintain.

- If you use server page technology you must be very disciplined to keep programming logic out of the page structure, often by using a helper object.
Transform View

- The Transform View uses a transform style of program
- The usual example is XSLT
- This can be very effective if you're working with domain data that's in XML format or can easily be converted to it
- An input controller picks the appropriate XSLT transform and applies it to XML gleaned from the model
Single-Stage View

- One view component for each screen in the application
Two-Step View

- One first-stage view for each screen, but only one second-stage view for the whole application
Summary

• Layering is an important technique in software architecture
• The hardest part of a layered architecture is deciding what layers to have and what the responsibility of each layer should be
• Architectural patterns such as model-view-controller will help you better design your layered system
References

- Patterns of Enterprise Application Architecture
  - By M. Fowler, D. Rice, M. Foemmel, E. Hieatt, R. Mee, and R. Stafford

- Wikipedia