Domain-Driven Design

Browser

Client UI

Server UI

Web Server

Controller

Service

Model

Entity

User

Domain-Driven Design

SWEN-261
Introduction to Software Engineering

Department of Software Engineering
Rochester Institute of Technology
Domain driven design centers the architecture on the problem domain.

- **Quote from the DDD Community:**
  Domain-driven design (DDD) is an approach to developing software for complex needs by deeply connecting the implementation to an evolving model of the core business concepts.

- **The premise:**
  - *Place the project’s primary focus on the core domain and domain logic*
  - *Base complex designs on a model*
  - *Initiate a creative collaboration between technical and domain experts to iteratively cut ever closer to the conceptual heart of the problem*
Let's review our project architecture.

Client UI

User

Network Connection

Frameworks

HTML, CSS & JavaScript

Platform

Any Browser

OS/Hardware

Server UI

Controller

View template

Application

Spark & FreeMarker

Model

Java

Web server (Jetty)

Any OS and HW

What goes in these two tiers?
DDD provides guidance for the remaining tiers.

Client UI
- User
- HTML, CSS & JavaScript
- Any Browser
- Any OS/HW

Server UI
- Network Connection
- Controller
- View template
- Spark & FreeMarker

Application
- Service
- Java
- Web server (Jetty)
- Any OS and HW

Model
- Entity
- Value Object
Services provide application logic.

- The Application tier is responsible for managing the user's interaction with the application.
- It is **not** responsible for domain logic which is in the Model layer.
- Application tier elements provide services to each client connection.
  - *Manage application-wide logic and information*
  - *Provide client-specific services for the UI tier*
Entities provide domain logic.

- The Model tier is responsible for managing domain entities and domain logic.

- Entity responsibilities are:
  - *Process user requests/commands*
  - *Effect changes based on user requests/commands*
  - *Validate Model-tier rules*
  - *Maintain the state of the Model*

- Entities often represent information about the world, and are inspired by domain model entities
  - *Customers, products and orders in e-commerce*
  - *Shapes in a drawing app*
Value objects provide values for an entity's "complex" attributes.

- A value object class encapsulates the data that represents an entity's attribute.
  - *Measurements, dates, credit card numbers, money, colors, (x,y) coordinates are some examples.*
  - *Two value objects are equal based on equality of the data in the object not object identity."

- Value objects must be immutable.
  - *An address of 15 N. Main St cannot be changed into 352 2nd Ave.*
  - *You create a new address object of 352 2nd Ave.*

- A value object class is **not** just a data holder class.
  - *Value Object = "value semantics" + immutability + GRASP Information Expert, ref. Flight class in OOD I*
Model objects are frequently used in collections.

- Many of the algorithms used in Model and Application components require using Entities and Value Objects in hash-based collections.
- Normal Java equality semantics are not adequate when dealing with Entities and VOs
  - An Entity must have a distinct id such that two objects with the same id must be considered equal.
  - Two Value Objects with the same data must be equal.
  - These semantic requirements imply specialized equals and hashCode methods.
- The after-class exercise provides instructions on how to create these methods.
A semantically correct value object can be used as a key in a map collection.

- Rather than extracting attributes from the value object to create a key, use the value object directly.

- This will work correctly because
  - The value object is immutable ➔ other code with a reference to the object can not change the object’s value while it is in the map as a key
  - The equals and hashCode methods ensure that two objects with the same value will be considered equal and generate the same hash code.
Let's review the architecture again.
This is the list of component responsibilities.

<table>
<thead>
<tr>
<th>UI Tier</th>
<th>Application Tier</th>
<th>Model Tier</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>UI Controller:</strong></td>
<td><strong>Service:</strong></td>
<td><strong>Entity:</strong></td>
</tr>
<tr>
<td>• Control the views based on the state of the application</td>
<td>• Manage application-wide logic and information</td>
<td>• Process user requests/commands</td>
</tr>
<tr>
<td>• Query the Model and Application tier as necessary to get information to present to the user</td>
<td>• Provide client-specific services to the UI tier</td>
<td>• Effect changes to the Model based on user requests/commands</td>
</tr>
<tr>
<td>• Perform simple input validation and data conversion based on input modality, e.g. String to integer</td>
<td>• Validate model rules</td>
<td>• Maintain the state of the model</td>
</tr>
<tr>
<td>• Initiate processing of user requests/commands possibly providing data the user input</td>
<td>• Maintain the state of the model</td>
<td></td>
</tr>
<tr>
<td>• Perform data conversion for display by views</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>UI View:</strong></td>
<td></td>
<td><strong>Value Object:</strong></td>
</tr>
<tr>
<td>• Provide an interface to the user</td>
<td></td>
<td>• Provide immutable value semantics</td>
</tr>
<tr>
<td>• Present information to the user in a variety of ways</td>
<td></td>
<td>• Provide value-based logic</td>
</tr>
<tr>
<td>• Provide a mechanism for user to input data and requests</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>