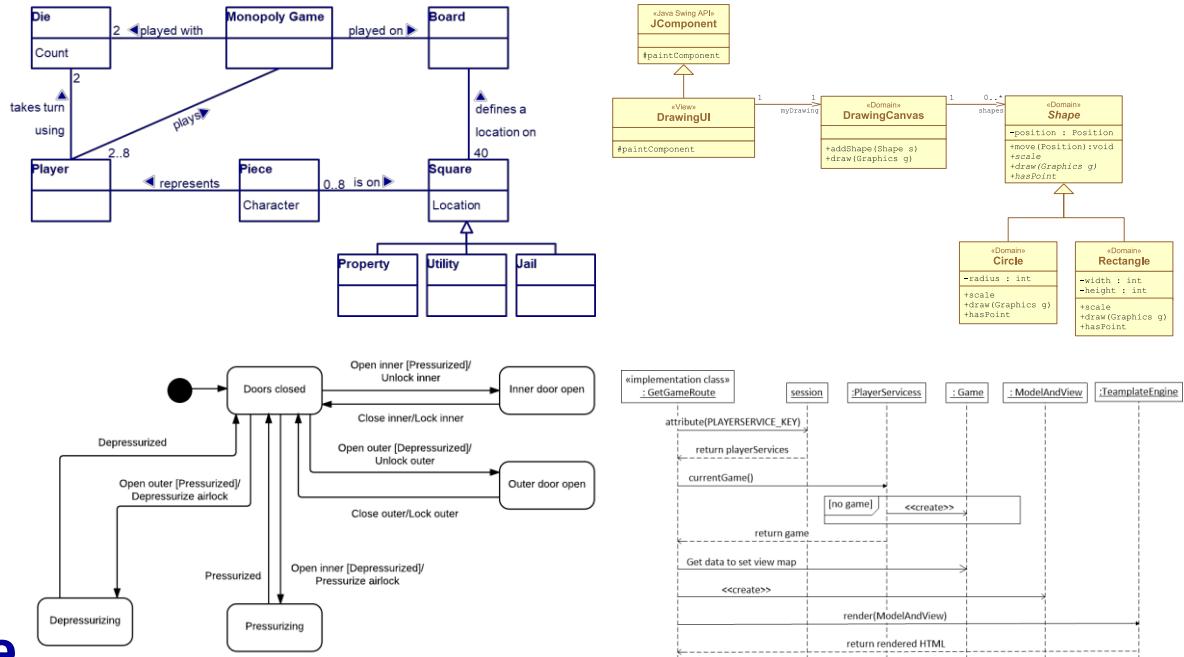


# Design and Code Communication



## SWEN-261 Introduction to Software Engineering

Department of Software Engineering  
Rochester Institute of Technology

```
/*
 * Get the {@linkplain GuessGame game} for the current user.
 * The user is identified by a {@linkplain Session browser session}.
 *
 * @param session
 *   The HTTP {@link Session}, must not be null
 *
 * @return
 *   An existing or new {@link GuessGame}
 *
 * @throws NullPointerException
 *   when the session parameter is null
 */
public GuessGame get(final Session session)
```



# Your communication about a project is not just in the form of presentations and meetings.

- The systems that you will develop are complex and have both static and dynamic design characteristics.
- To describe those characteristics you will use several UML models.
  - *Domain, class, statechart, sequence*
- Those who must use your implementation need a more productive description than studying lines of code.
- Those who must maintain your implementation must be able to quickly understand the code.



# The domain model describes the product owner's understanding of the application's scope.

- Domain model
  - *Describes the context in which the application will operate.*
  - *Helps developers share the product owner's understanding of this context.*
  - *Describes the product owner's world view of the domain entities and relationships between them.*
- The domain model will help developers create a structure for the implementation to the extent that is possible.



# The class model defines the static structure of your implementation.

- It captures many constructs embodied in your implementation
  - *Class attributes and methods with visibilities*
  - *Relationships between classes with multiplicities*
  - *Navigation between classes*
  - *Structure via inheritance/interface*
  - *Architectural tiers*
- The domain model inspires the first-cut for the implementation class structure.
  - *Try to have the software structure match the product owner's domain structure, i.e. domain entities become implementation classes*



# You also must describe the application's dynamic characteristics to fully describe its operation.

- The dynamic behavior is often state-based and succinctly described with a statechart.
  - *Exchanges between a client and web application*
  - *User interface operation*
  - *Communication protocols*
  - *Individual classes with state-based characteristics*
- An application's execution of a feature/operation involves multiple classes across architectural tiers.
  - *The sequence diagram indicates which classes and methods are involved in an execution scenario.*
  - *Formulate a user story solution with one or more sequence diagrams created before starting the implementation.*



# How your code "reads" is critically important for the humans who will read it.

*Any fool can write code that a computer can understand. Good programmers write code that humans can understand.*

*Refactoring: Improving the Design of Existing Code*  
Martin Fowler, et. al (1999)



# Code is read by humans as much as by machines.

- Code must be readable and understandable by all team members.
- Clear code communication includes:
  - *A shared code style*
  - *Use of good, meaningful names*
  - *Component APIs are clearly documented*
  - *Algorithms are clarified using in-line comments*
  - *Indication of incomplete or broken code*



# A shared code style is good etiquette.

- No code style is inherently better than any other one.
- A code style includes:
  - *Spaces vs tabs*
  - *Where to put curly-braces*
  - *Naming conventions*
    - ◆ CamelCase for class names
    - ◆ UPPER\_CASE for constants
    - ◆ lowerCamelCase for attribute and method names
  - *And so on*
- Every team should choose a style and stick to it.
  - *IDEs provide support for defining a code style*
  - *If your team cannot choose one then we recommend using Google Java style (see resources)*



# Make names reflect what they mean and do.

- Dos:
  - *Use names that reflect the purpose*
  - *Use class names from analysis and domain model*
  - *Use method names that are verbs in your analysis*
  - *Use method names that describe what it does not how it does it*
- Don'ts:
  - *Don't abbreviate; spell it out*
    - ◆ pricePerUnit is better than pPU or worse just p
  - *Don't use the same local variable for two purposes; create a new variable with an appropriate name*
  - *Don't use "not" in a name*
    - ◆ isValid is better than isNotValid.



# Document your component's API.

- In Java the `/** ... */` syntax is used to denote a documentation for the thing it precedes.
- For example:

```
/**  
 * A single "guessing game".  
 *  
 * @author <a href='mailto:joecool@rit.edu'>Joe Cool</a>  
 */  
public class GuessGame
```

- At a minimum you should document all public members.
  - *Also good to document all methods including private methods*
  - *Document attributes with complex data structures*



# A method's javadoc must explain how to use the operation.

- Every method must have an opening statement that expresses what it does.
  - *Keep this statement concise*
  - *Additional statements can be added for clarification*
- Document the method signature
  - *Use @return to describe what is returned*
  - *Use @param to describe each parameter*
  - *Use @throws to describe every exception explicitly thrown by the method*
- Link to other classes
  - *Use @link to link to classes*
  - *Use @linkplain in opening statement*



# Example method javadocs.

```
/**  
 * Get the {@linkplain GuessGame game} for the current user.  
 * The user is identified by a {@linkplain Session browser session}  
 *  
 * @param session  
 *   The HTTP {@link Session}, must not be null  
 *  
 * @return  
 *   An existing or new {@link GuessGame}  
 *  
 * @throws NullPointerException  
 *   when the session parameter is null  
 */  
  
public GuessGame get(final Session session)
```

Use `@linkplain` in the opening statement.

Use `@link` in all other clauses.

**get**

```
public GuessGame get(spark.Session session)  
  
Get the game for the current user. The user is identified by a browser session.  
  
Parameters:  
session - The HTTP Session, must not be null  
  
Returns:  
An existing or new GuessGame  
  
Throws:  
NullPointerException - when the session parameter is null
```



# Use in-line comments to communicate algorithms and intention.

- Use in-line comments to describe an algorithm

- **Dos:**

- ◆ Use pseudo-code steps
    - ◆ Explain complex data structures

- **Don'ts:**

- ◆ Don't repeat the code in English

```
count++; // increment the count
```

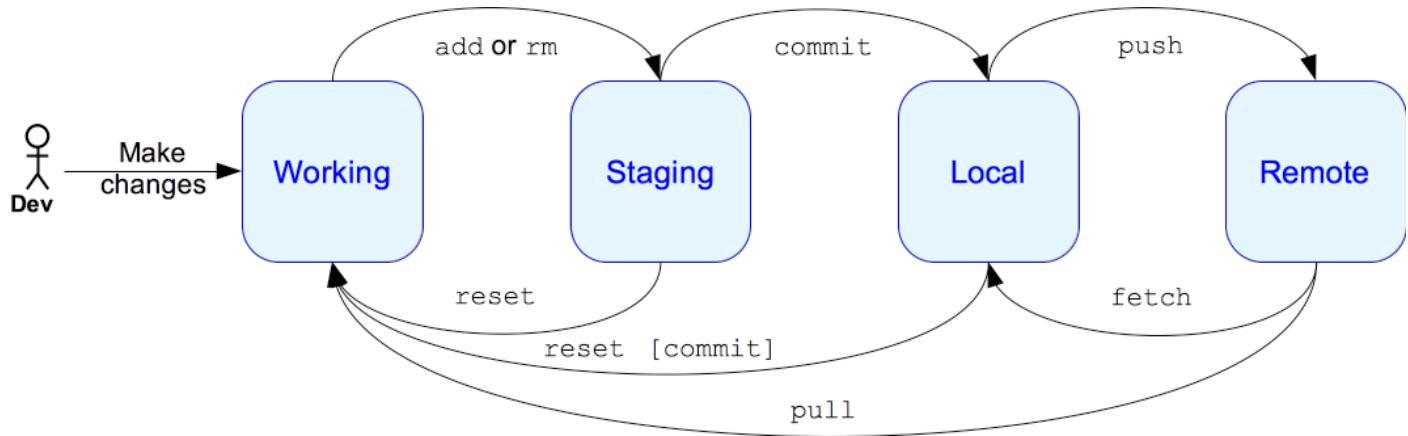
- Use comments to express issues and intentions

- **A *TODO* comment hints at a future feature**

- **A *FIXME* (or *FIXME*) comment points to a known bug that is low priority**



# Review Version Control Concepts



**SWEN-261**  
**Introduction to Software**  
**Engineering**

Department of Software Engineering  
Rochester Institute of Technology



# Managing change is a constant aspect of software development.

- The Product and Sprint backlogs represent the upcoming changes.
- A software release is a snapshot of code at a certain time.
  - *Capturing a certain set of user stories*
  - *Multiple releases may be done so the team needs to keep track of multiple snapshots (aka versions)*
- Version control systems (VCS) are used to manage changes made to software and tag releases.

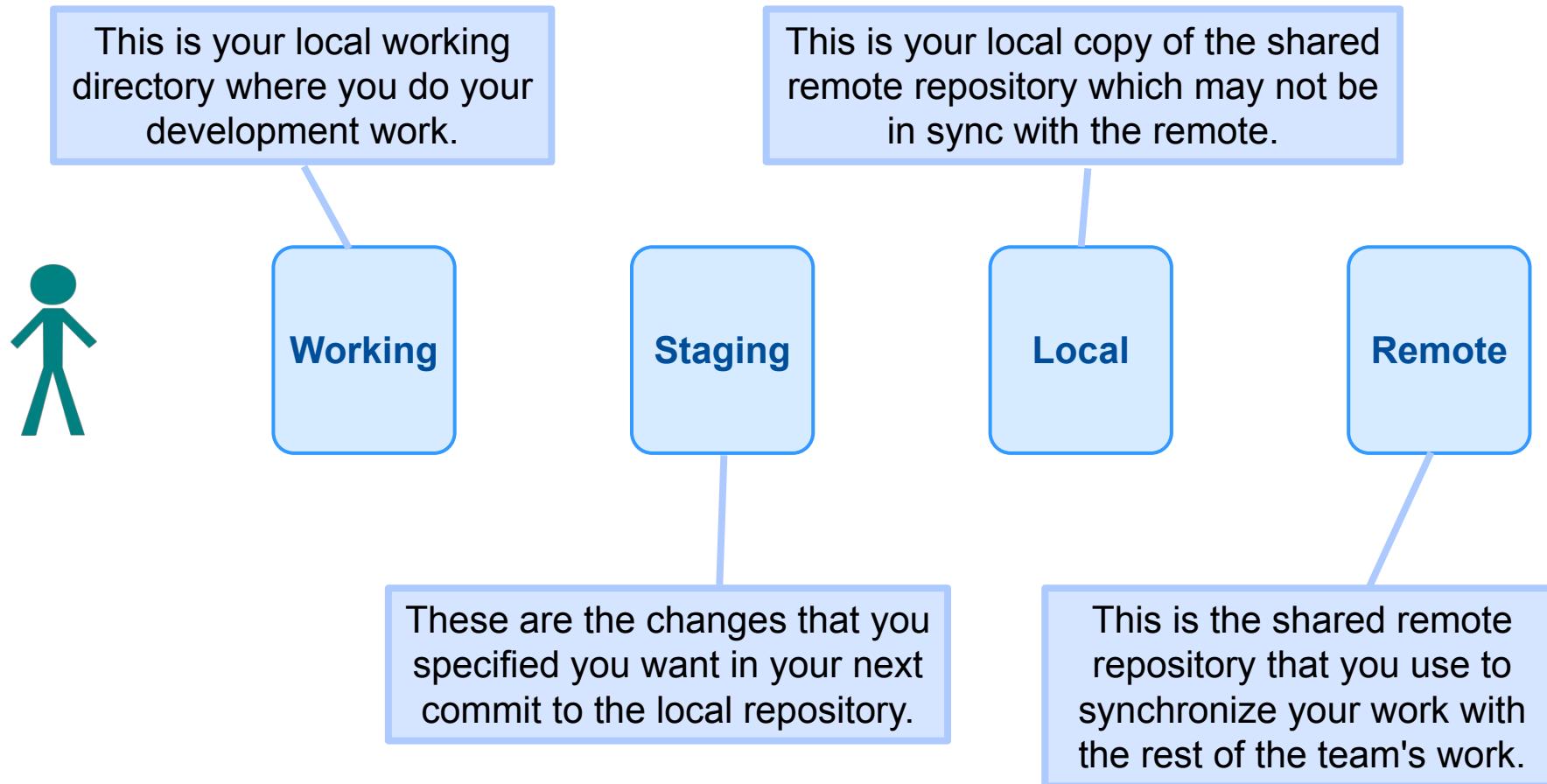


# There are a few fundamental activities of change management that every VCS supports.

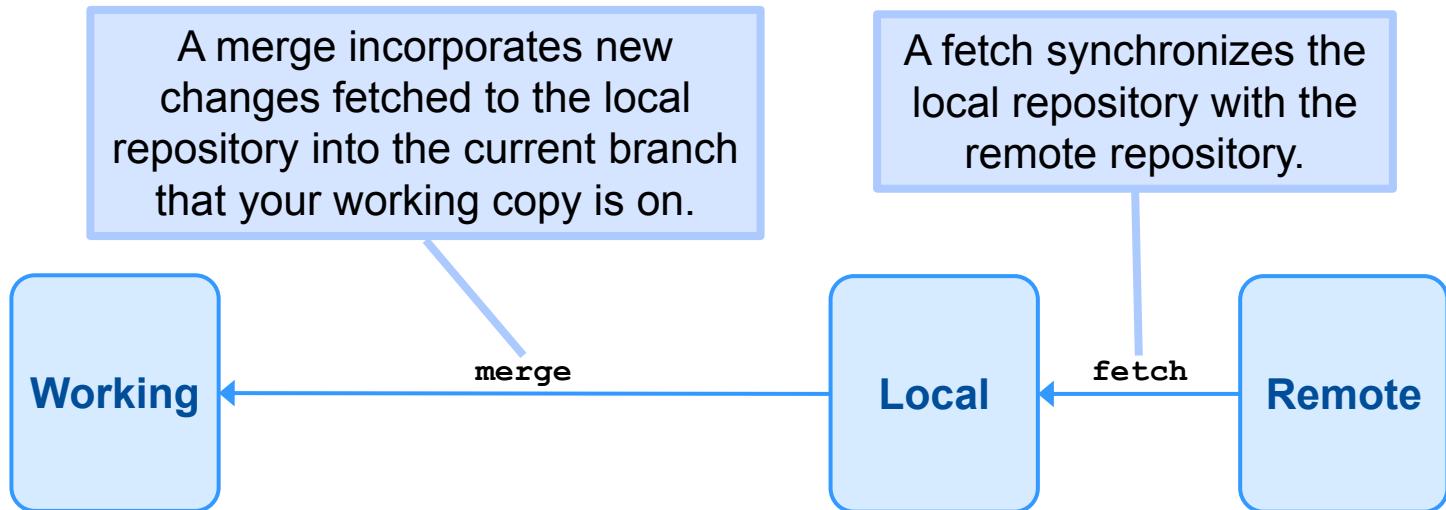
- Directories and files are tracked in a *repository*.
  - *Each developer has their own workspace.*
  - *But share a common remote repository.*
- You can
  - *Make changes in your workspace*
    - ◆ Add files or directories
    - ◆ Remove files or directories
    - ◆ Modify or move files
    - ◆ Binary files are tracked as a single unit
  - *Commit the changes to a repository.*
  - *Sync your workspace with a repository.*
  - *Create branches to track user stories.*
  - *Explore the history and changes to a repository.*



# Git has four distinct areas that your work progresses through.



# Your local repository and working copy do not automatically stay in sync with the remote.



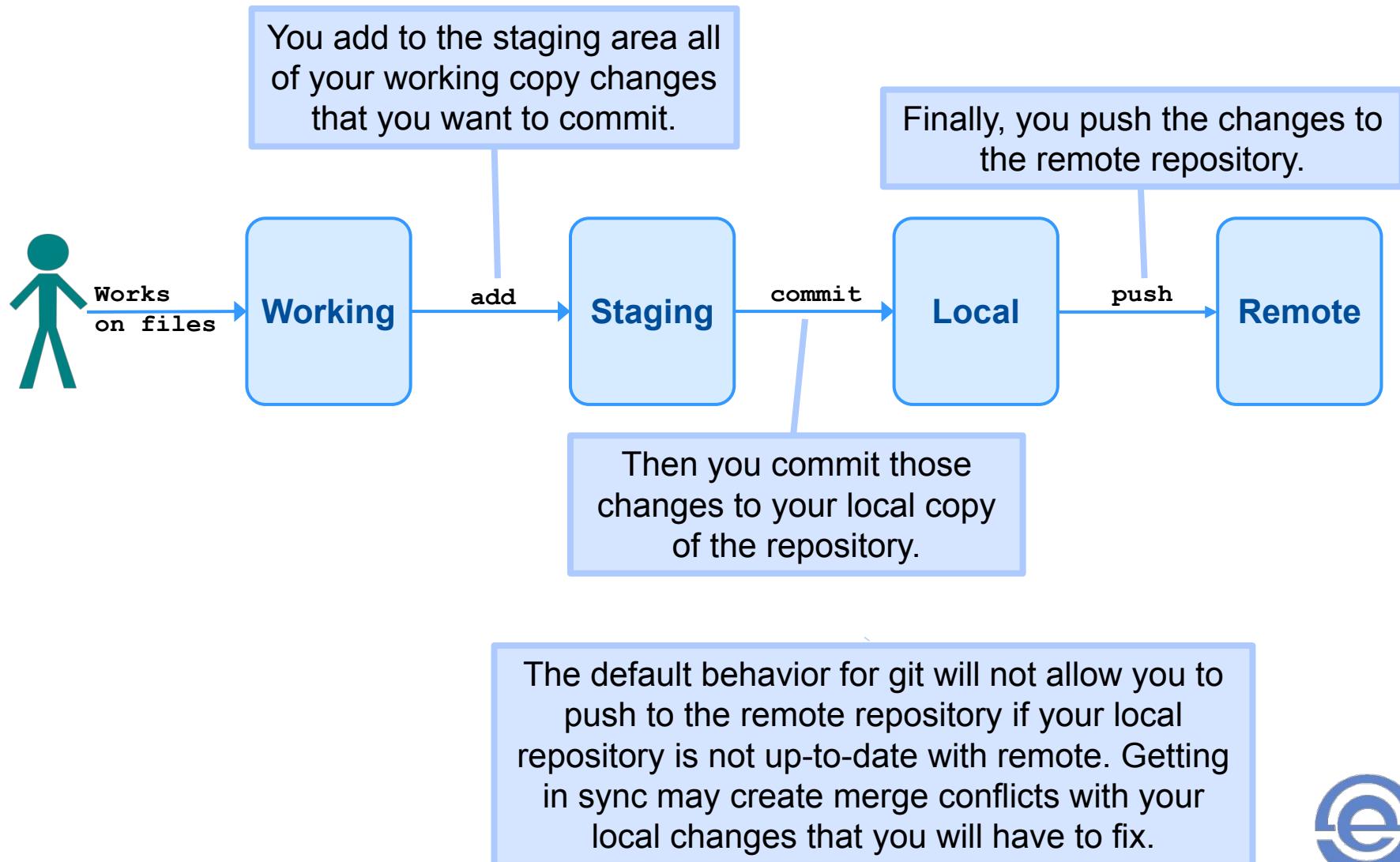
**pull**  $\equiv$  **fetch** + **merge**

A branch is an independent stream of repository changes which isolates work from the rest of the repository.

A merge may detect changes that can not be automatically incorporated. This is called a merge conflict.



# When you make local changes, those changes must pass through all four areas.

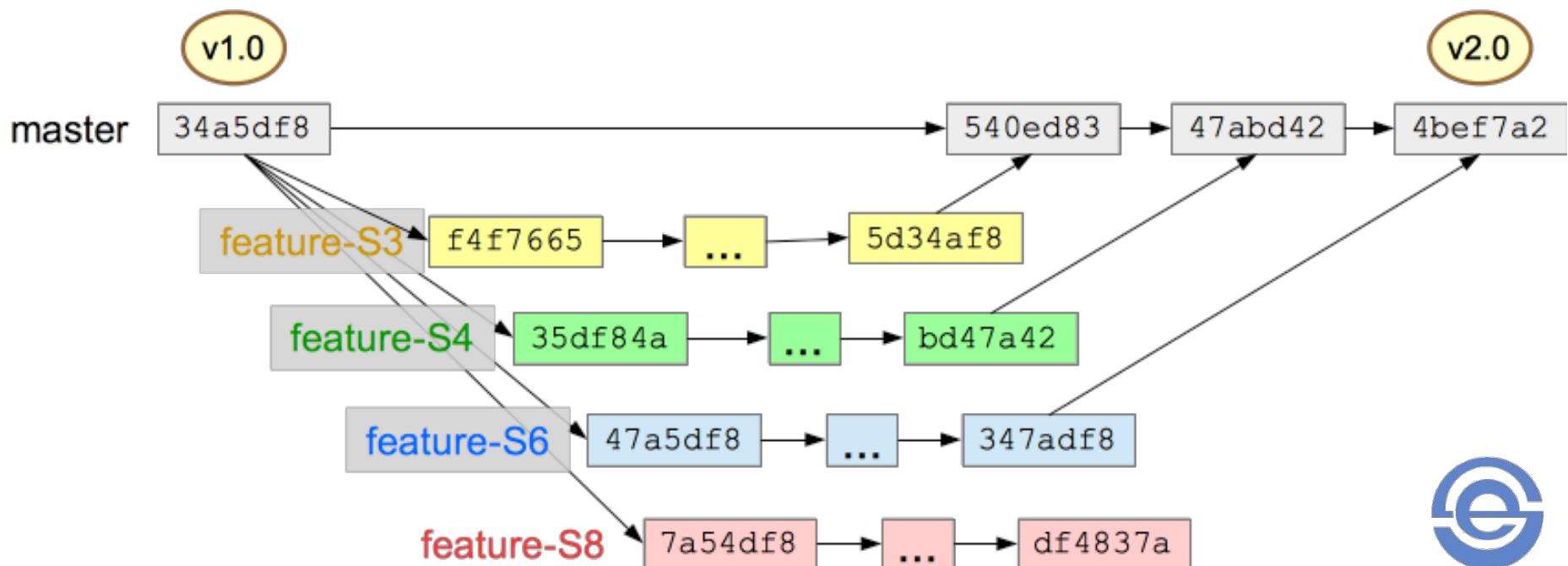
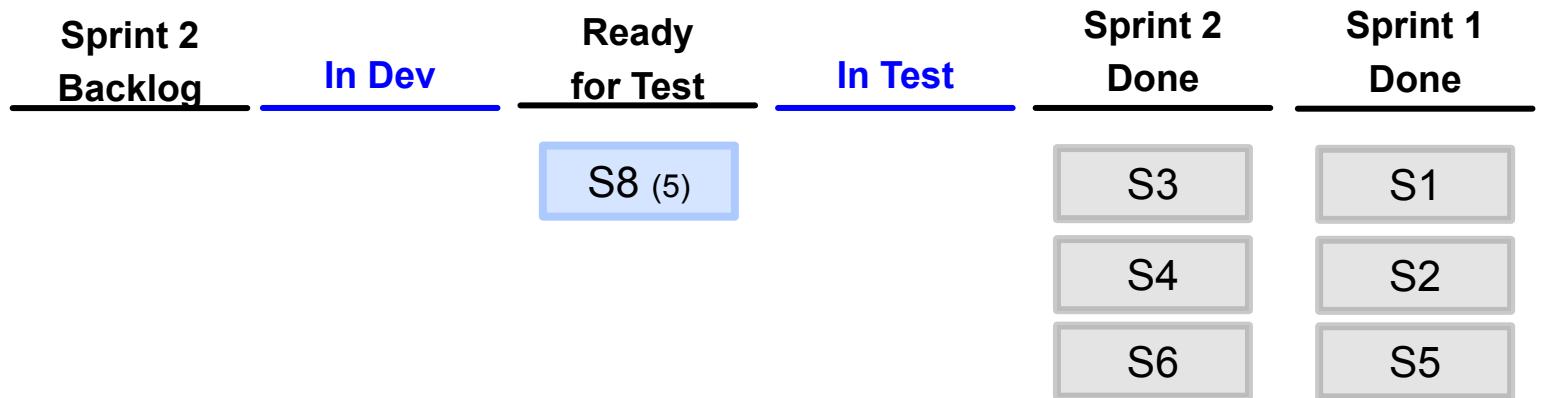


# Version control branching supports the ability to manage software releases.

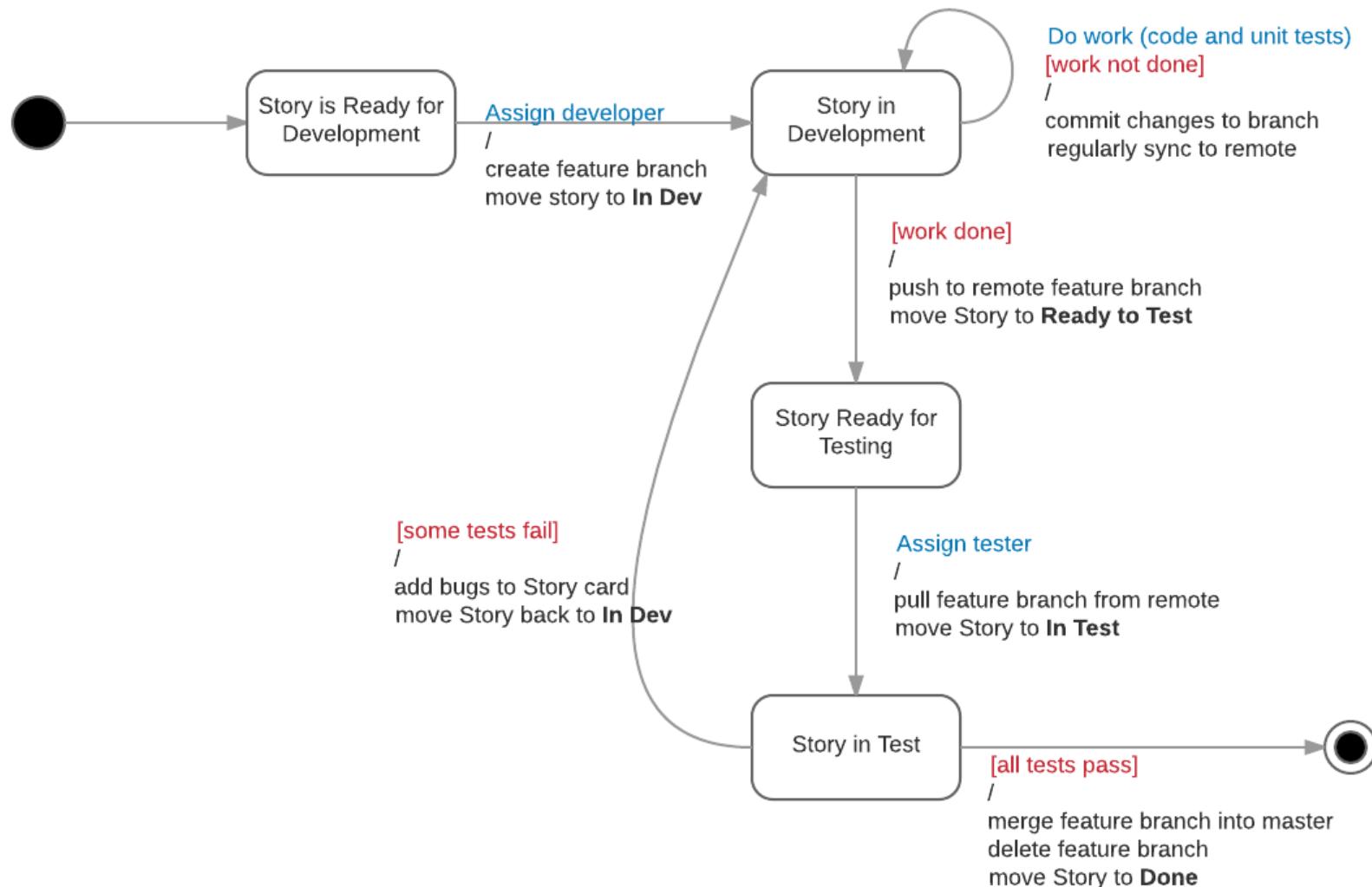
- At the end of a sprint, the team will want to include *done* stories but exclude incomplete stories.
- This cannot be done when all of the stories are developed in the master branch.
- Feature branching is a technique that creates a branch for each story during development.
  - *Changes are isolated in specific branches.*
  - *When the story is done, the feature branch is merged into the master branch.*
  - *The master branch never receives incomplete work.*
  - *Thus master is always the most up-to-date, working increment.*



# An example sprint at the end.

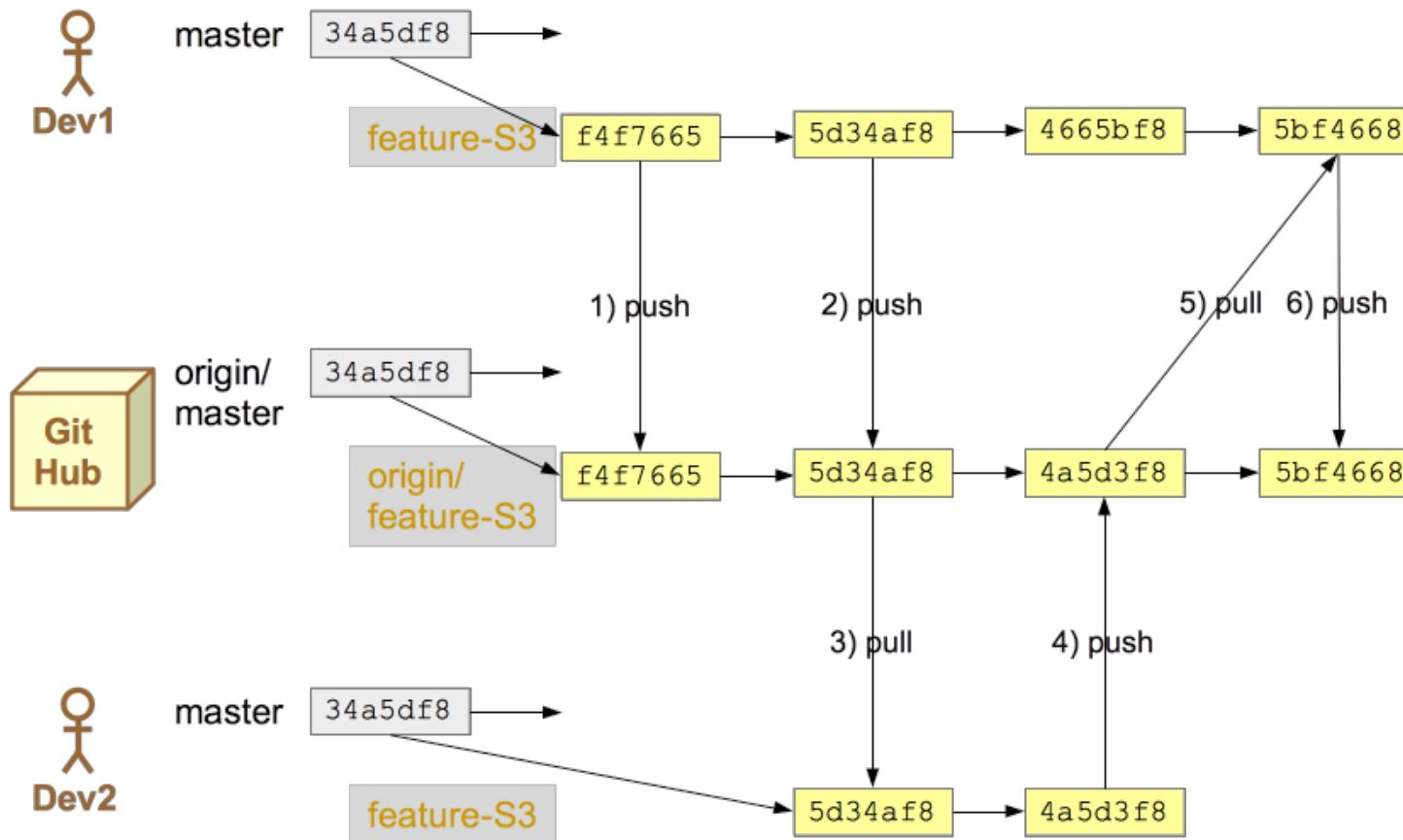


# The life cycle of the feature branch is tied to the development cycle of the user story.



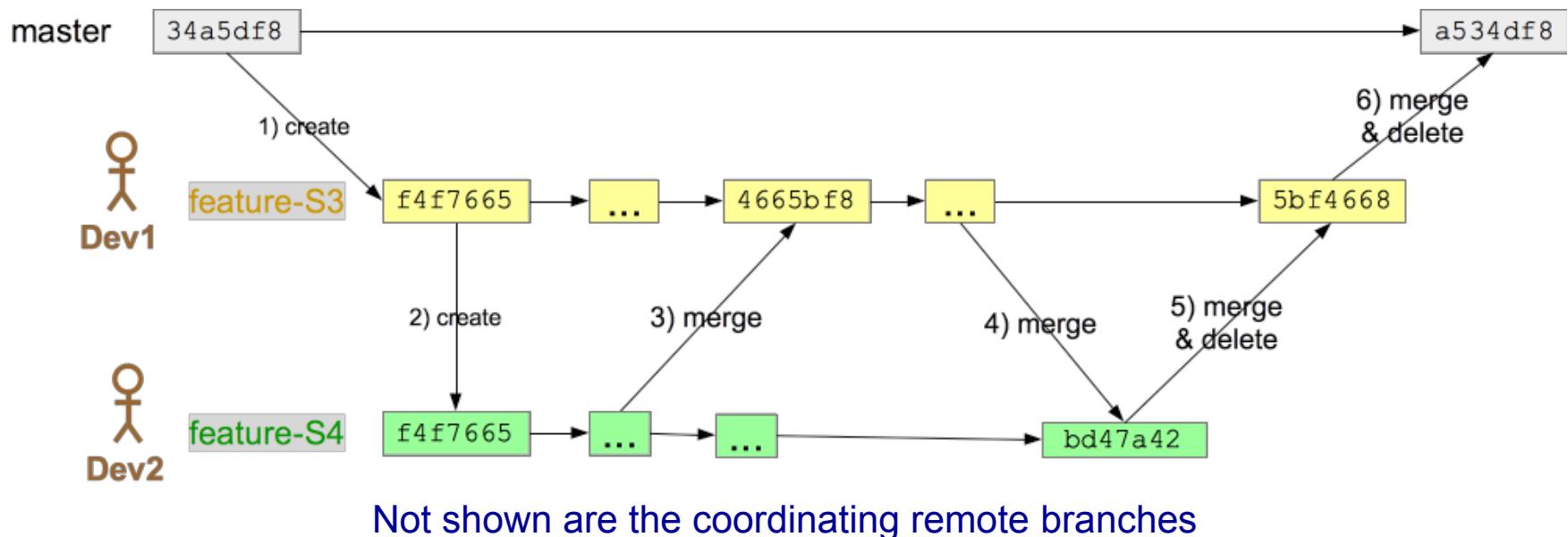
# Two developers collaborate on a story by working on the same feature branch.

- The developers share code on a story by syncing to the same remote feature branch.



# Two interdependent stories can share changes across two branches.

- The first story branch is created from master and the second branch is created from the first.



# Merging happens a lot and usually goes well; other times *not so much*.

- Every time you sync with the remote repository a merge occurs.
- A *merge conflict* occurs when there is at least one file with overlapping changes that can not be automatically resolved.



# Here's an example of a merge conflict.

- Consider Betty and Sam independently fix this bug.

```
/**  
 * Calculate a half-off discount.  
 */  
public float calculateDiscount(final float cost) {  
    return cost * 2;  
}
```

- Betty did this: `return cost / 2;`
- Sam did this: `return cost * 0.5f;`
- When Sam merges in the code from Betty:

→ `git merge dev1`

Auto-merging src/main/java/com/example/model/Promotion.java

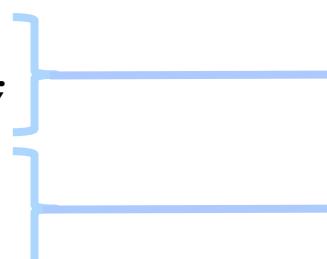
CONFLICT (content): Merge conflict in  
src/main/java/com/example/model/Promotion.java

Automatic merge failed; fix conflicts and then commit the result.



# Resolving a simple text conflict is often easy.

- When a conflict occurs git reports the affected files.

```
public float calculateDiscount(final float cost) {  
    <<<<< HEAD  
        return cost * 0.5f;  
=====  
        return cost / 2;  
>>>>> dev1  
    }  
A diagram illustrating a code conflict. On the left, there is a snippet of Java code. In the middle, two curly braces extend from the 'return' statements. The top brace is associated with a callout box containing the text 'The HEAD in Sam's workspace.' The bottom brace is associated with a callout box containing the text 'This is the code from Betty's branch.'
```

- Determine the best solution, and remove the other solution and the marker text.
- Then follow through with an add, commit, and push.



# To minimize the number of times when conflicts will not resolve easily, follow several guidelines.

1. Keep code lines short; break up long calculations.
2. Keep commits small and focused.
3. Minimize stray edits.
4. If multiple developers are collaborating on a feature, each developer should sync with the remote feature branch regularly.  
*Merge in the remote feature branch and then push to it, if you have changes.*
5. If development of a feature is taking a long time, back merge master to sync completed features for this sprint into the feature branch.



# Using feature branches will be a standard part of your development workflow.

**Definition of Done Checklist** [Delete...](#)

0%

- acceptance criteria are defined
- solution tasks are specified
- feature branch created
- unit tests written
- solution is *code complete*, i.e. passes full suite of unit tests
- design documentation updated
- pull request created
- user story passes all acceptance criteria
- code review performed
- feature branch merged into master
- feature branch deleted



# Acceptance Testing

**SWEN-261**  
**Introduction to Software**  
**Engineering**

Department of Software Engineering  
Rochester Institute of Technology



# Acceptance testing verifies that the software meets the requirements of the stakeholders.

- Each user story identifies a project requirement.
- The Product Owner will be involved with the definition of the user stories to ensure that the overall system will satisfy the business needs.
- A user story must pass its acceptance tests to be considered done.
  - *Performing acceptance testing is a development team responsibility.*
  - *An embedded Product Owner may be involved with acceptance testing.*
- Ideally, user acceptance testing would be done with each user story, but this is rare.



# The user story text is too broad and vague to be used directly for acceptance testing.

- One aspect of backlog refinement is defining the **acceptance criteria** for each user story before the team can move the user story from the Product Backlog to the Sprint Backlog.
- The acceptance criteria provide details of what it means to satisfy a user story's requirement.
- Each user story will typically have multiple acceptance criteria defined.



# Each acceptance criterion defines one aspect of the user story's requirement.

- Acceptance criteria are phrased in the format:
  - *GIVEN some condition WHEN some action occurs THEN the system does something.*
- In many cases, more details in the form of specific test cases will be specified to completely define the acceptance criteria.
- The Acceptance Test Plan is the set of all the acceptance criteria for all the user stories along with any specific test cases that are defined.



# For your term project, you will keep an up-to-date Acceptance Test Plan.

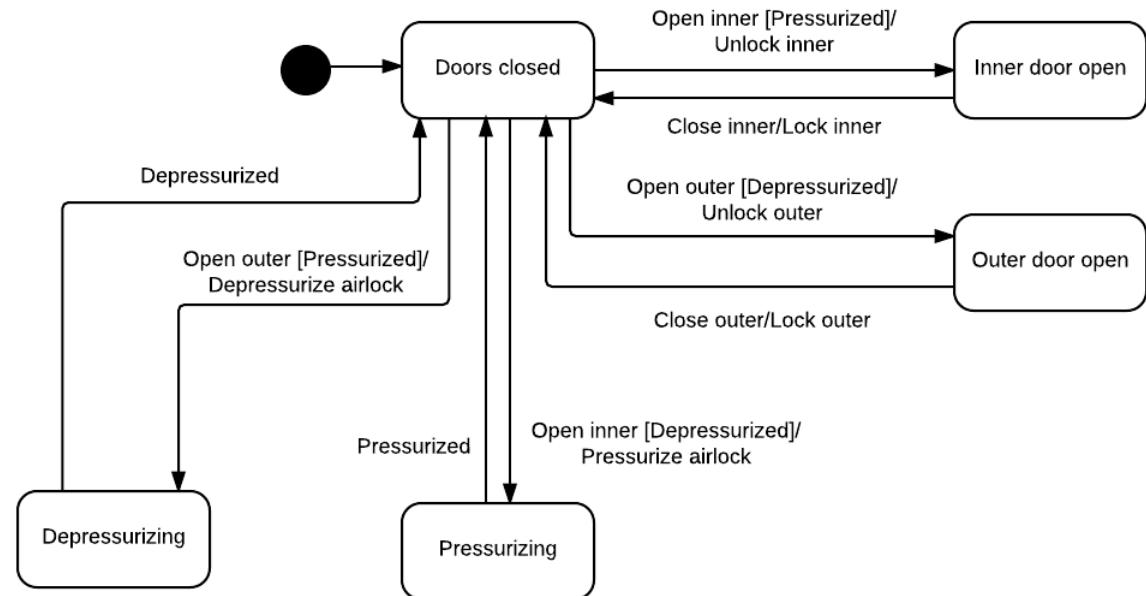
- The team will submit an Acceptance Test Plan with Sprints 1 through 3 and use it to drive the demos of their project.



# State-based Behavior I

**SWEN-261**  
**Introduction to Software**  
**Engineering**

Department of Software Engineering  
Rochester Institute of Technology



# A large part of software behavior is dependent on the state of the system, i.e. state-based.

- A finite-state machine is a notational mechanism for capturing this state-based behavior.
  - *The UML diagram is called a statechart.*
- Explicitly defining this state-based behavior provides a common specification for the team.
- Allowing the state behavior to evolve implicitly creates a situation where every team member may not have the same model of the behavior.



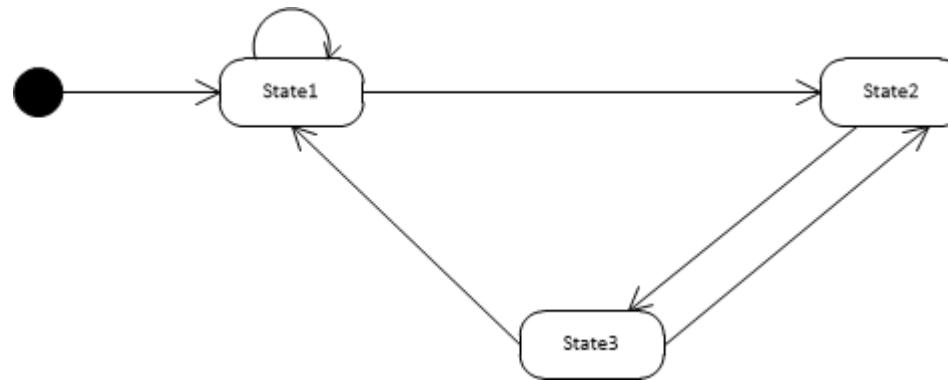
# A statechart can define behavior in multiple areas in your application.

- Defining the operation of an interface, such as, the web application interface.
  - *For an after class exercise, you will create a statechart that describes the sample webapp's web application interface.*
- Specifying the behavior for a single object.
  - *Later in the course, we will come back to statecharts and discuss how to implement the state-based behavior explicitly.*

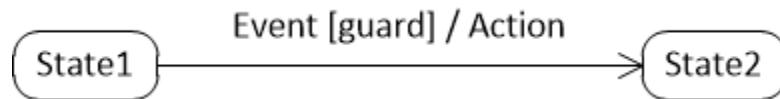


# A statechart identifies the recognizable conditions that a system can be in over intervals of time.

- The system can exist in only one state at a time.
  - *You typically want a deterministic state machine to define behavior of your software systems.*
- The system exists in a state for a period of time.
- A solid ball specifies the starting point.



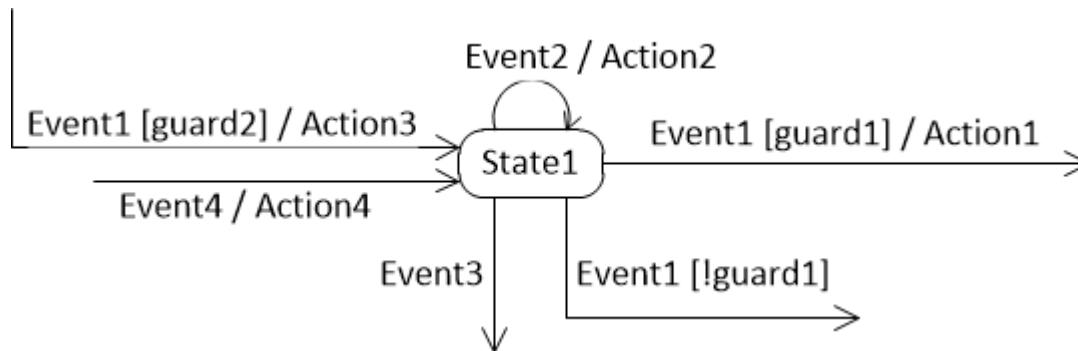
# Transitions provide the mechanism for the system to move from one state to another.



- The *event* triggers the transition to be taken.
  - *The event could be calling a method, receipt of a signal, end of a time period, or end of an activity.*
- The *guard* is a Boolean condition that must be true for the transition to be triggered.
- When the transition is triggered, the *action* list is executed.



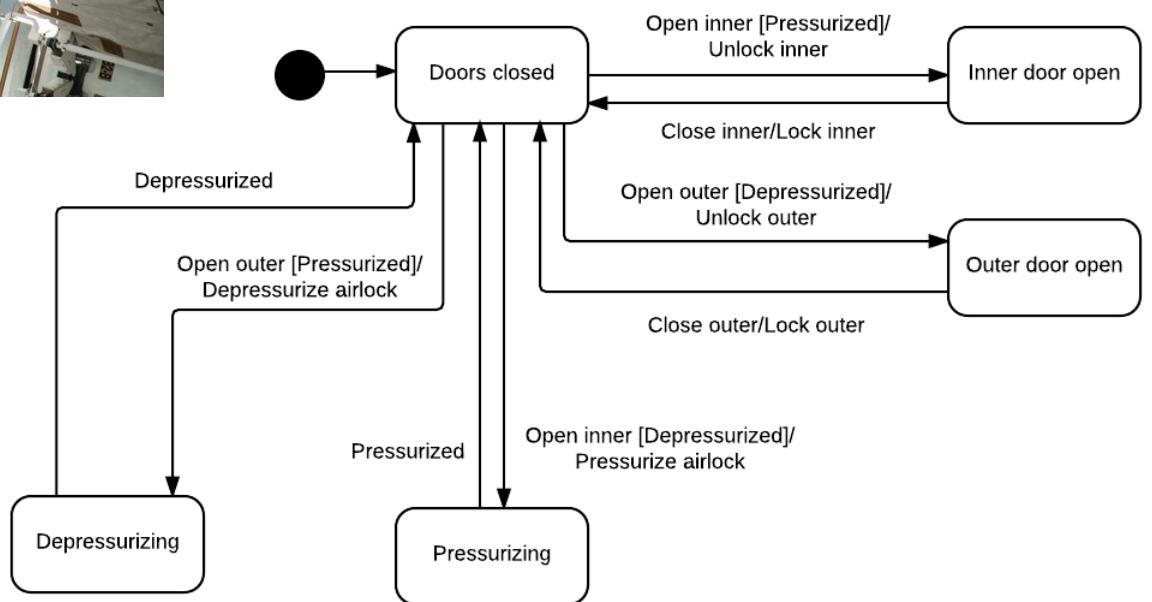
# Transitions provide the mechanism for the system to move from one state to another.



- A transition executes instantaneously relative to the time spent in a state.
- A transition can return the system to the same state that it was in when the transition was taken.
  - *Any actions would be executed before returning to the state just left.*
- There can be any number of transitions entering or leaving a state.



# Here is a statechart for control of an airlock.



# Follow these guidelines when creating your statecharts.

- Pick meaningful state, event, and guard names.
- Always specify a starting point without a guard on the transition to the initial state.
- Guards on multiple transitions from a state with the same trigger event should be mutually exclusive.
- Evaluating the guard should have no side effects, and the guard cannot use side effects of actions on a transition it is guarding.

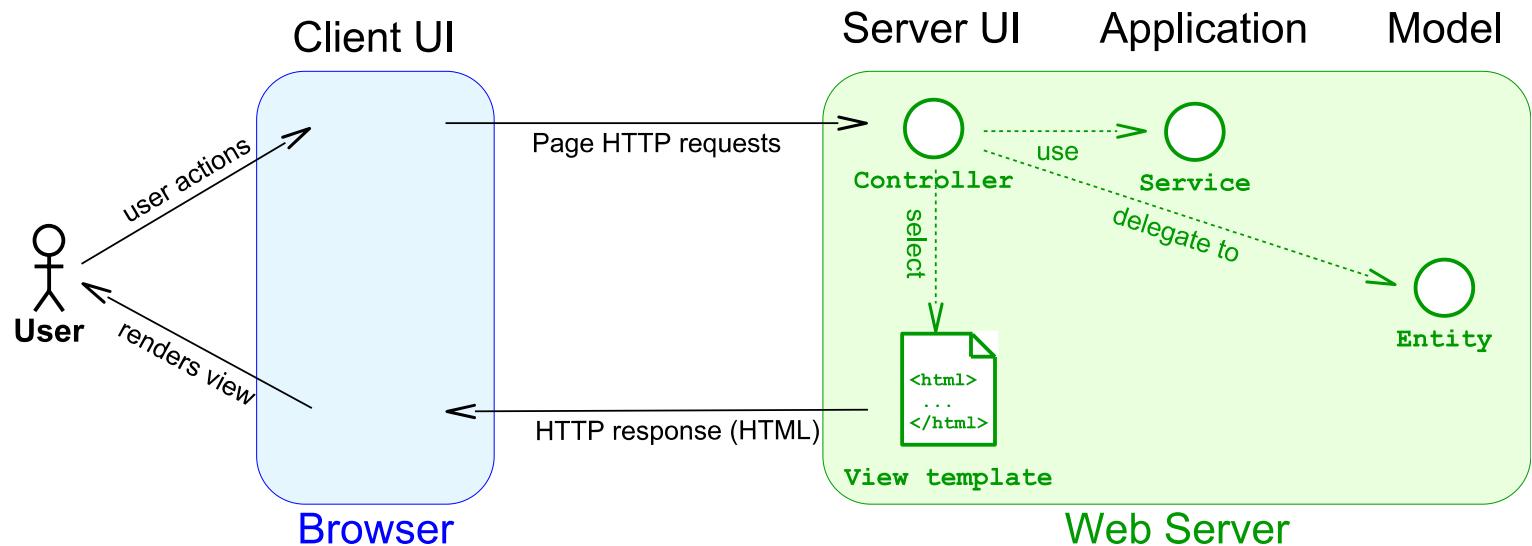


# Use a statechart to get a shared understanding of the state-based behavior of a system.

- Even if you do not explicitly implement the states, you can more clearly capture the system behavior.
- There are frameworks that provide an implementation of state-based behavior, such as StatefulJ FSM (<https://github.com/statefulj>) or squirrel-foundation (<https://github.com/hekailiang/squirrel>)
- The full statechart notation has even richer semantics defined (entry/exit actions, composite states, orthogonal/concurrent states).



# 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:

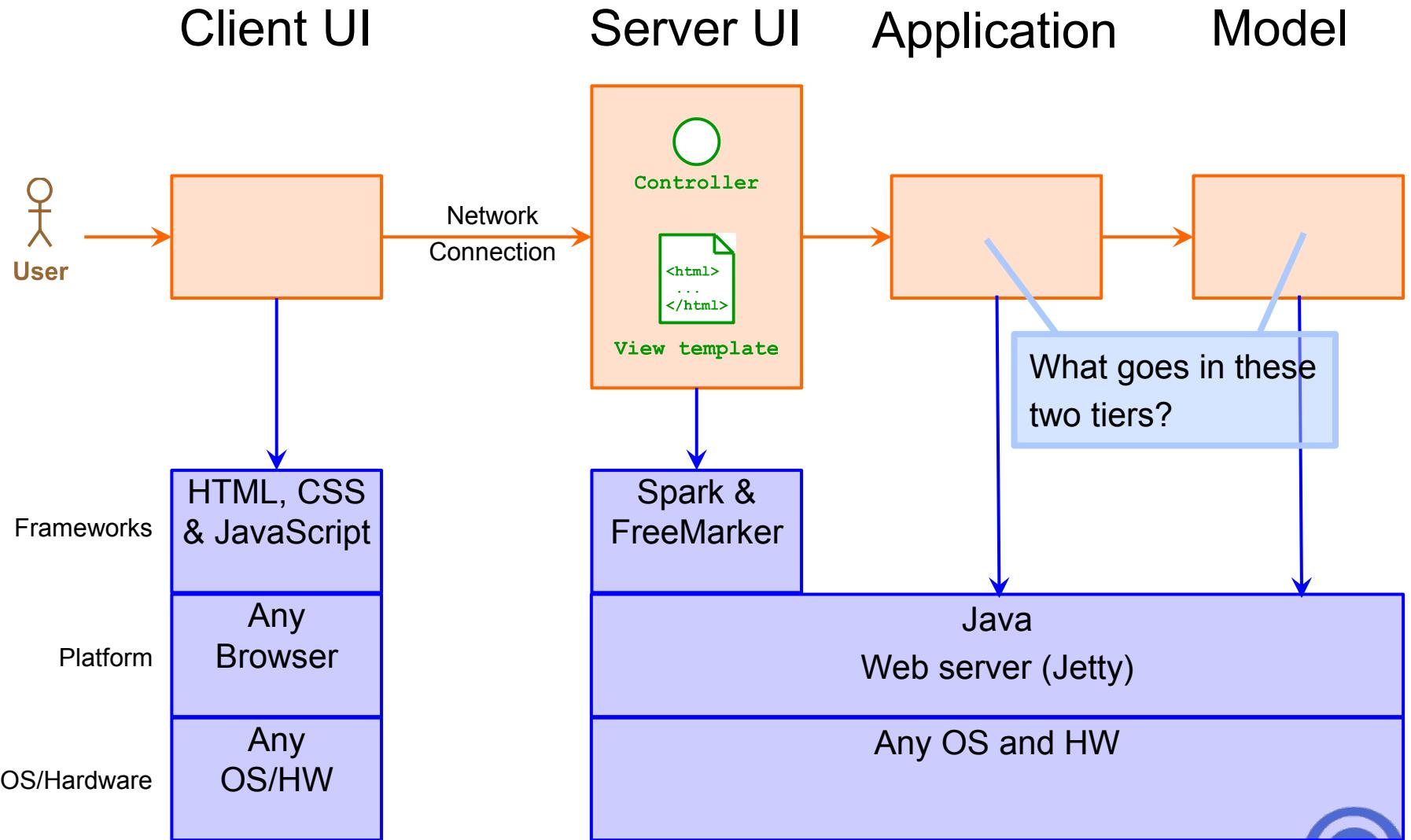
Domain-driven design (DDD) is an approach to developing software for complex needs by deeply connecting the implementation to an evolving model of the core business concepts

- The premise:

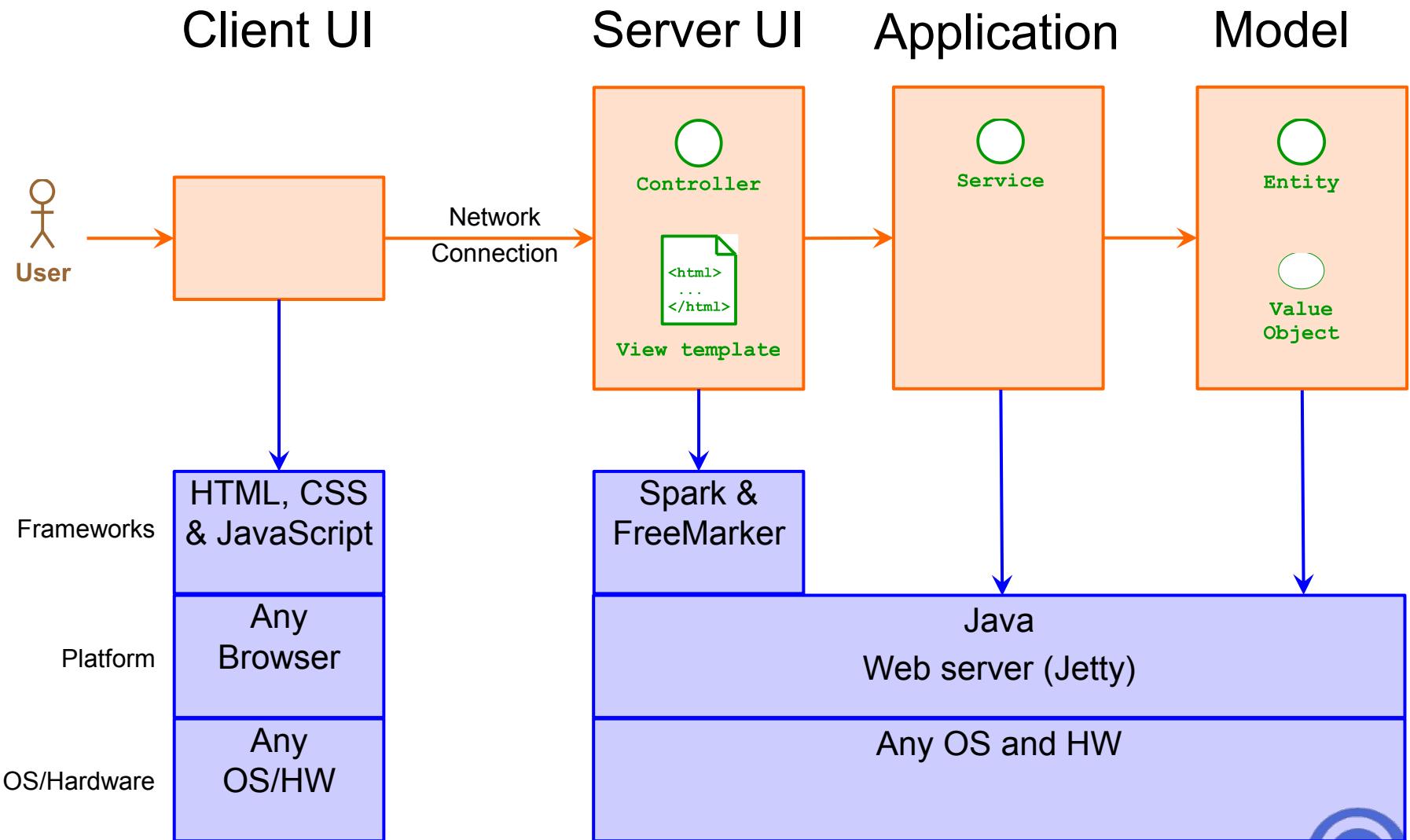
- *Place the project's primary focus on the core domain and domain logic*
- *Base complex designs on a model*
- *Initiate a creative collaboration between technical and domain experts to iteratively cut ever closer to the conceptual heart of the problem*



# Let's review our project architecture.



# DDD provides guidance for the remaining tiers.



# 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 2<sup>nd</sup> Ave.*
  - *You create a new address object of 352 2<sup>nd</sup> 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.

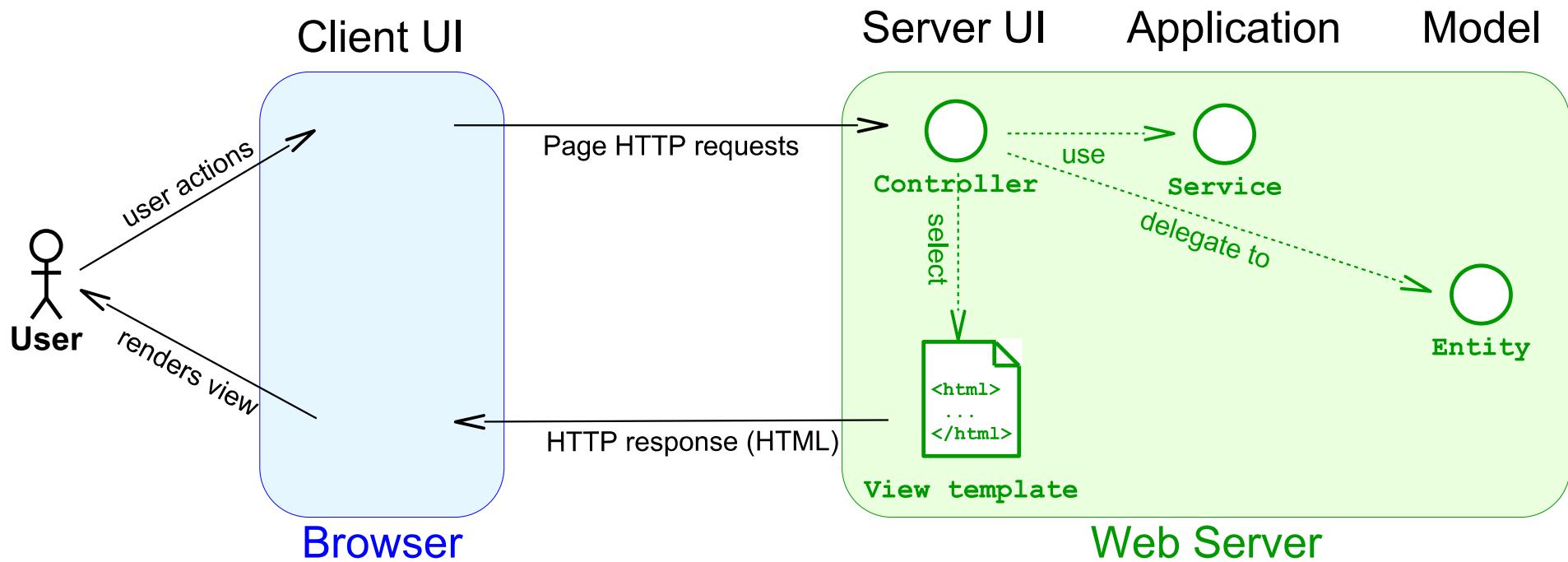


# A semantically correct value object can be used as a key in a map collection.

- 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.

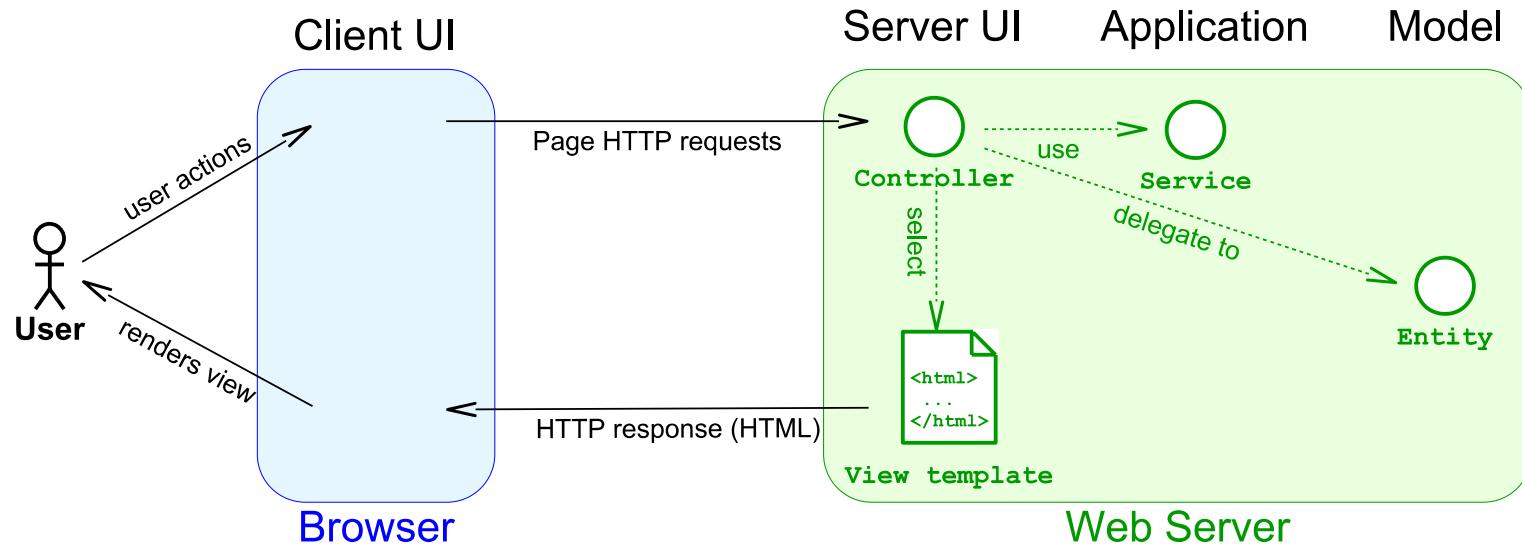


# This is the list of component responsibilities.

UI Tier	Application Tier	Model Tier
<p><b>UI Controller:</b></p> <ul style="list-style-type: none"><li>• Control the views based on the state of the application</li><li>• Query the Model and Application tier as necessary to get information to present to the user</li><li>• Perform simple input validation and data conversion based on input modality, e.g. String to integer</li><li>• Initiate processing of user requests/commands possibly providing data the user input</li><li>• Perform data conversion for display by views</li></ul> <p><b>UI View:</b></p> <ul style="list-style-type: none"><li>• Provide an interface to the user</li><li>• Present information to the user in a variety of ways</li><li>• Provide a mechanism for user to input data and requests</li></ul>	<p><b>Service:</b></p> <ul style="list-style-type: none"><li>• Manage application-wide logic and information</li><li>• Provide client-specific services to the UI tier</li></ul>	<p><b>Entity:</b></p> <ul style="list-style-type: none"><li>• Process user requests/commands</li><li>• Effect changes to the Model based on user requests/commands</li><li>• Validate model rules</li><li>• Maintain the state of the model</li></ul> <p><b>Value Object:</b></p> <ul style="list-style-type: none"><li>• Provide immutable value semantics</li><li>• Provide value-based logic</li></ul>



# Web Architecture and Development



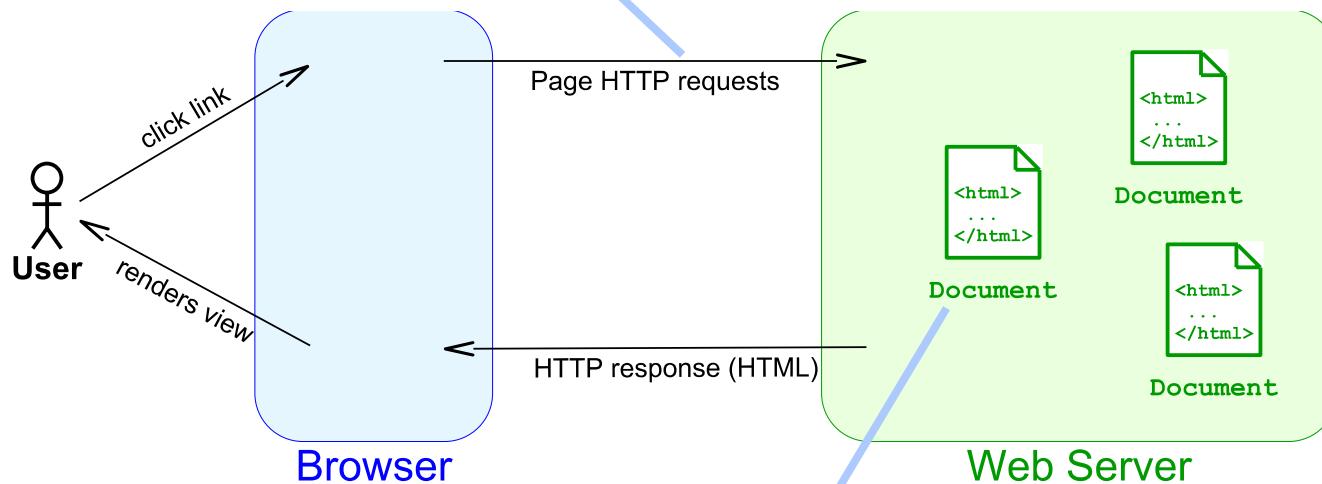
**SWEN-261**  
**Introduction to Software**  
**Engineering**

Department of Software Engineering  
Rochester Institute of Technology



# HTTP is the protocol of the world-wide-web.

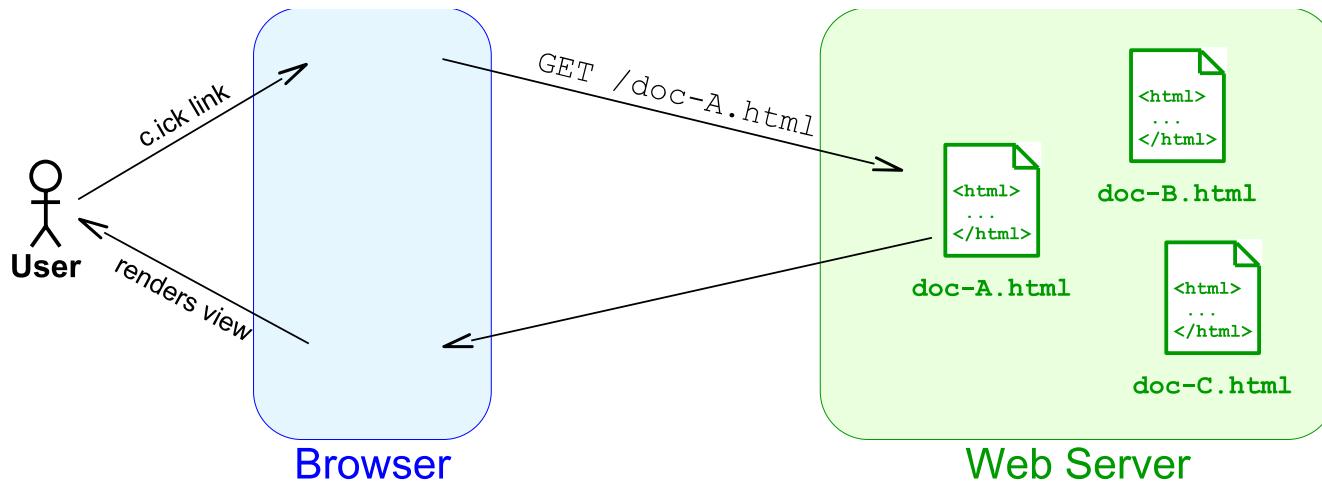
The Hypertext Transfer Protocol (HTTP) was designed to exchange hypertext documents.



Hypertext Markup Language (HTML) is the standard format of these documents.



# HTTP has a standard set of verbs.



- GET retrieves *resources*.
- There are several others: POST, PUT, DELETE and more



# HTML is a standard document format.

- Base structure:

```
<!DOCTYPE html>
<html>
  <head><!-- page metadata --></head>
  <body><!-- page content --></body>
</html>
```

- HTML5 includes many element types including:

- *Content: h1-h5, p, img, a, div/span, many more...*
- *Lists: ul, ol, li*
- *Forms: form, input, button, select/option ...*
- *Tables: table, thead, tbody, tr, th/td ...*

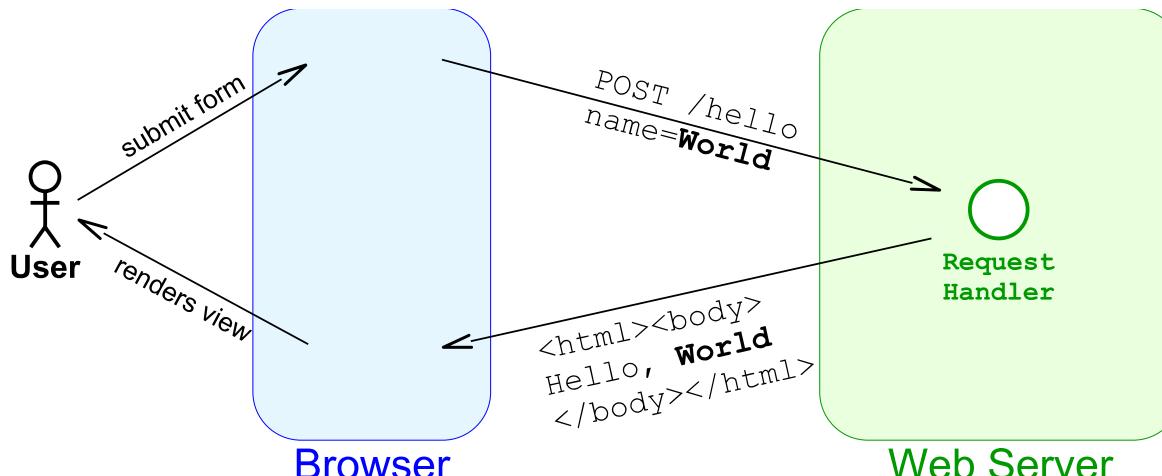


# Web 1.0 applications create "forms" within documents.

- Example simple web form:

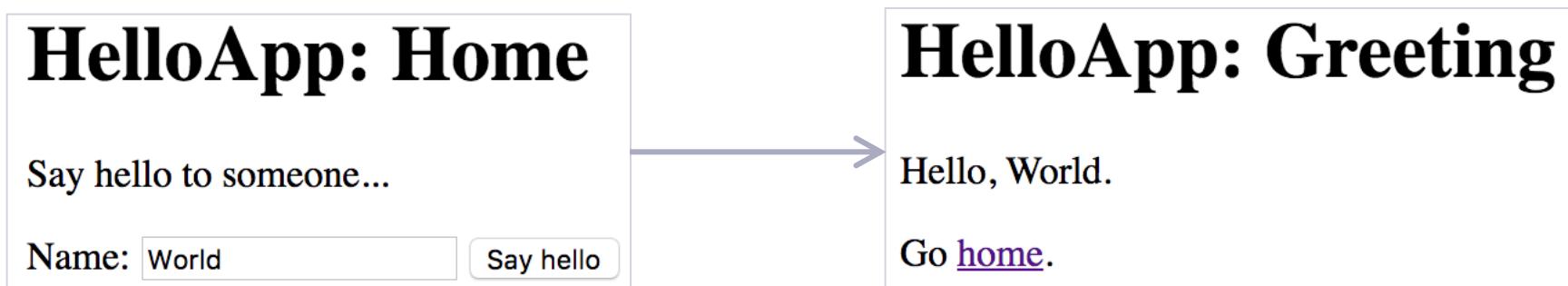
```
<form action='hello' method='POST'>
  <label for='name'>Name:</label>
  <input name='name' placeholder='Enter a name' />
  <button type='submit'>Say hello</button>
</form>
```

- The POST HTTP verb sends form data to the action URL.

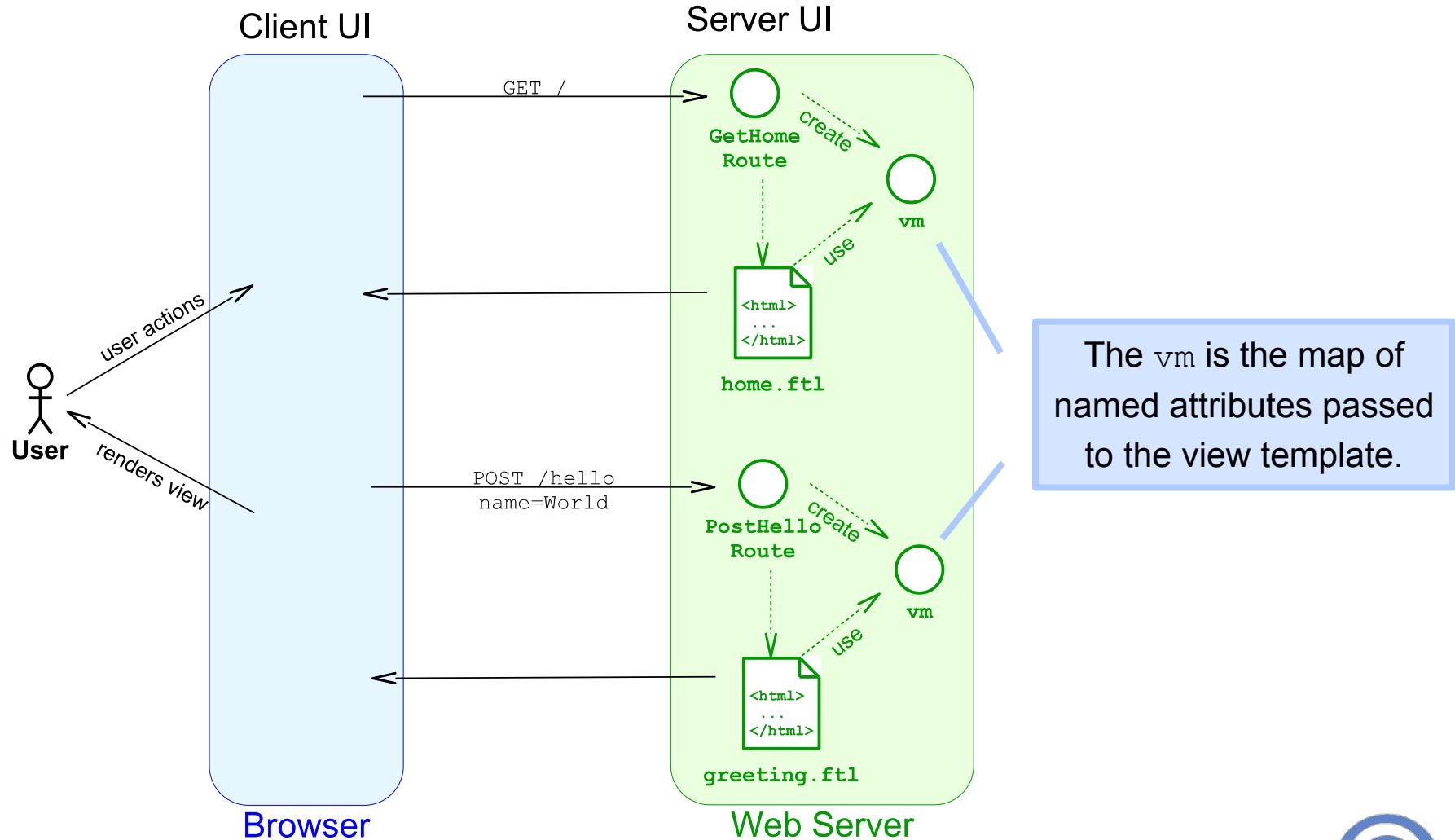


# The web server handles each HTTP request.

- A web server must have two types of components:
  - *HTTP route handler (aka a UI controller)*
  - *HTML view templates (aka a UI view)*
- This is an application of the *Separation of Concerns* principle.
- Consider a trivial Hello World web app:



# This app happens to need a View and a route Controller for each verb-URL pair.



# The structure of a Spark web application.

- Spark is a Java-based, web micro-framework
  - *It handles HTTP requests*
  - *It delegates the HTML generation to a template engine*
- Here's the configuration code for the HelloApp

```
import static spark.Spark.*;
import spark.TemplateEngine;
import spark.template.freemarker.FreeMarkerEngine;

public class HelloApp {
    public static void main(String[] args) {
        final TemplateEngine templateEngine = new FreeMarkerEngine();
        get("/", new GetHomeRoute(templateEngine));
        post("/hello", new PostHelloRoute(templateEngine));
    }
}
```



# Here's an example Spark route controller.

```
import java.util.HashMap;
import java.util.Map;

import spark.*;

public class GetHomeRoute implements Route {
    private final TemplateEngine templateEngine;
    // constructor not shown
    public Object handle(Request request, Response response) {
        final Map<String, Object> vm = new HashMap<>();
        vm.put("pageTitle", "Home");
        return templateEngine.render(new ModelAndView(vm, "home.ftl"));
    }
}
```

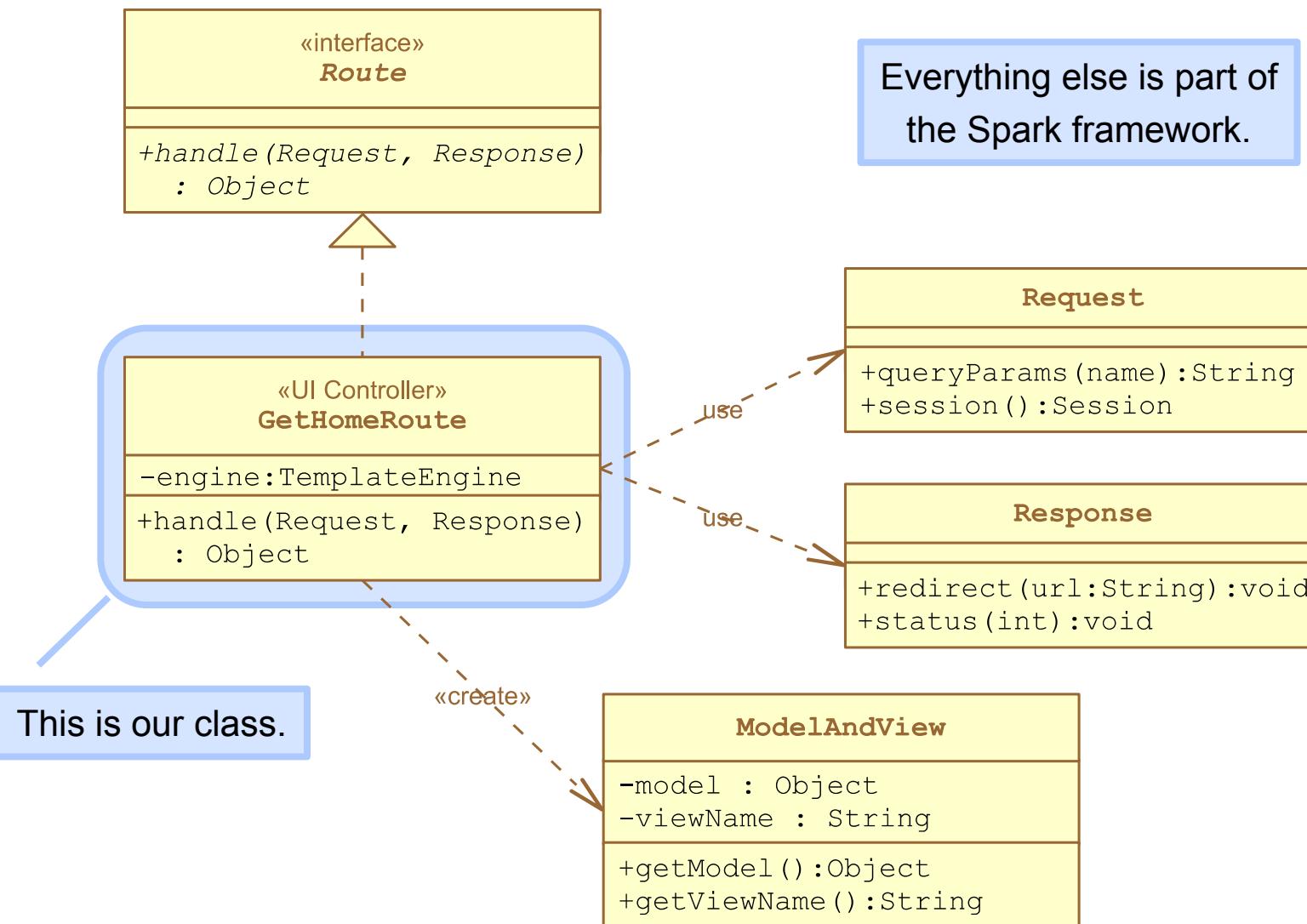
We use the convention to name route controllers as VerbUrlRoute.

The View-Model is a Java map of name/value pairs called *attributes*.

This is the name of the FreeMarker template file.



# Here's the model of the example Spark controller.



# Here's an example FreeMarker view template.

- A FreeMarker resource is a text file with HTML plus special tags for additional *rendering logic*.

```
<!DOCTYPE html>
<head>
</head>
<body>

<h1>HelloApp: ${pageTitle}</h1>

<p>Say hello to someone...</p>

<form action='hello' method='POST'>
  <label for='name'>Name:</label>
  <input name='name' placeholder='Enter a name...' />
  <button type='submit'>Say hello</button>
</form>

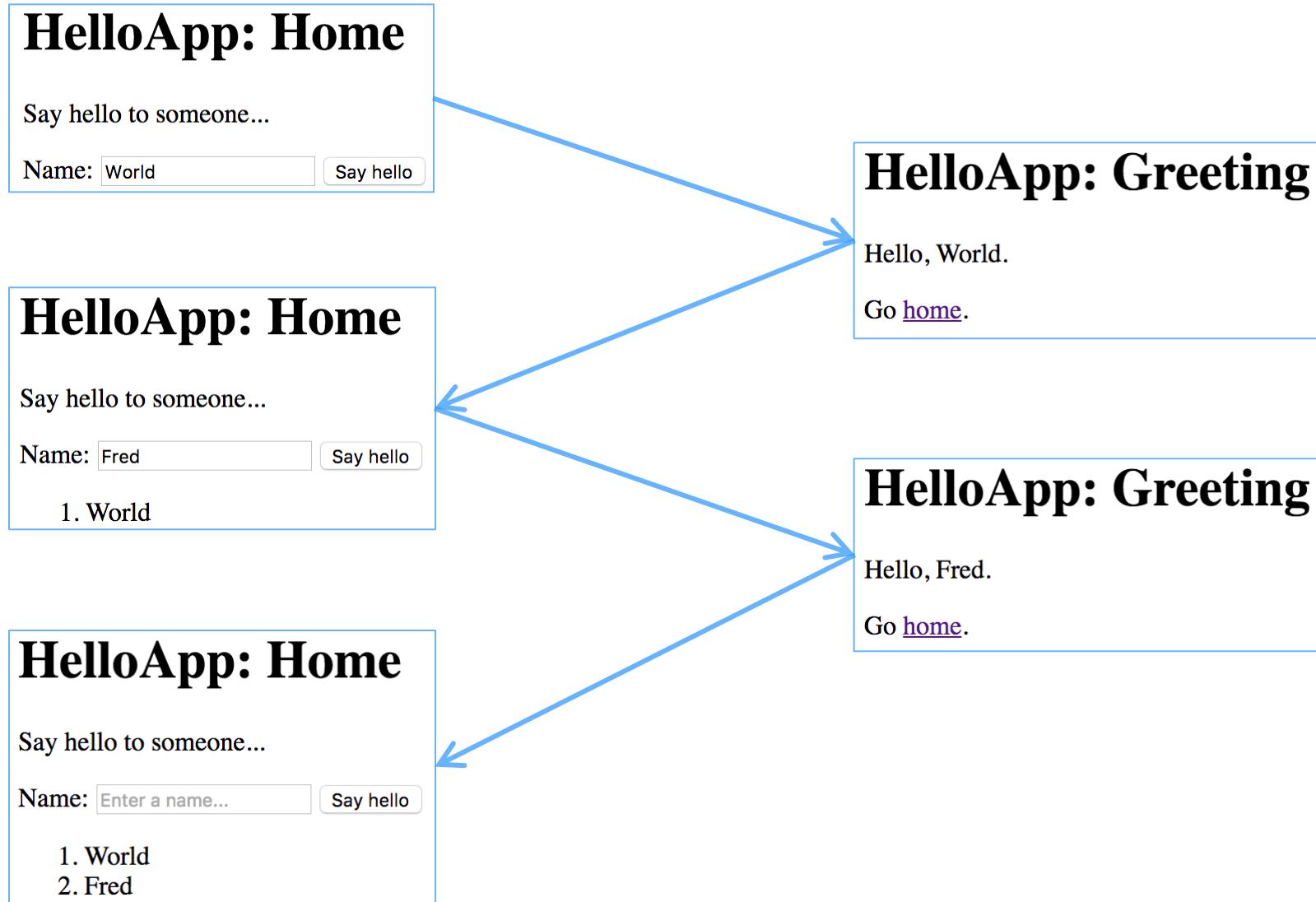
</body>
</html>
```

This is the key for an attribute in the *View-Model* map object.

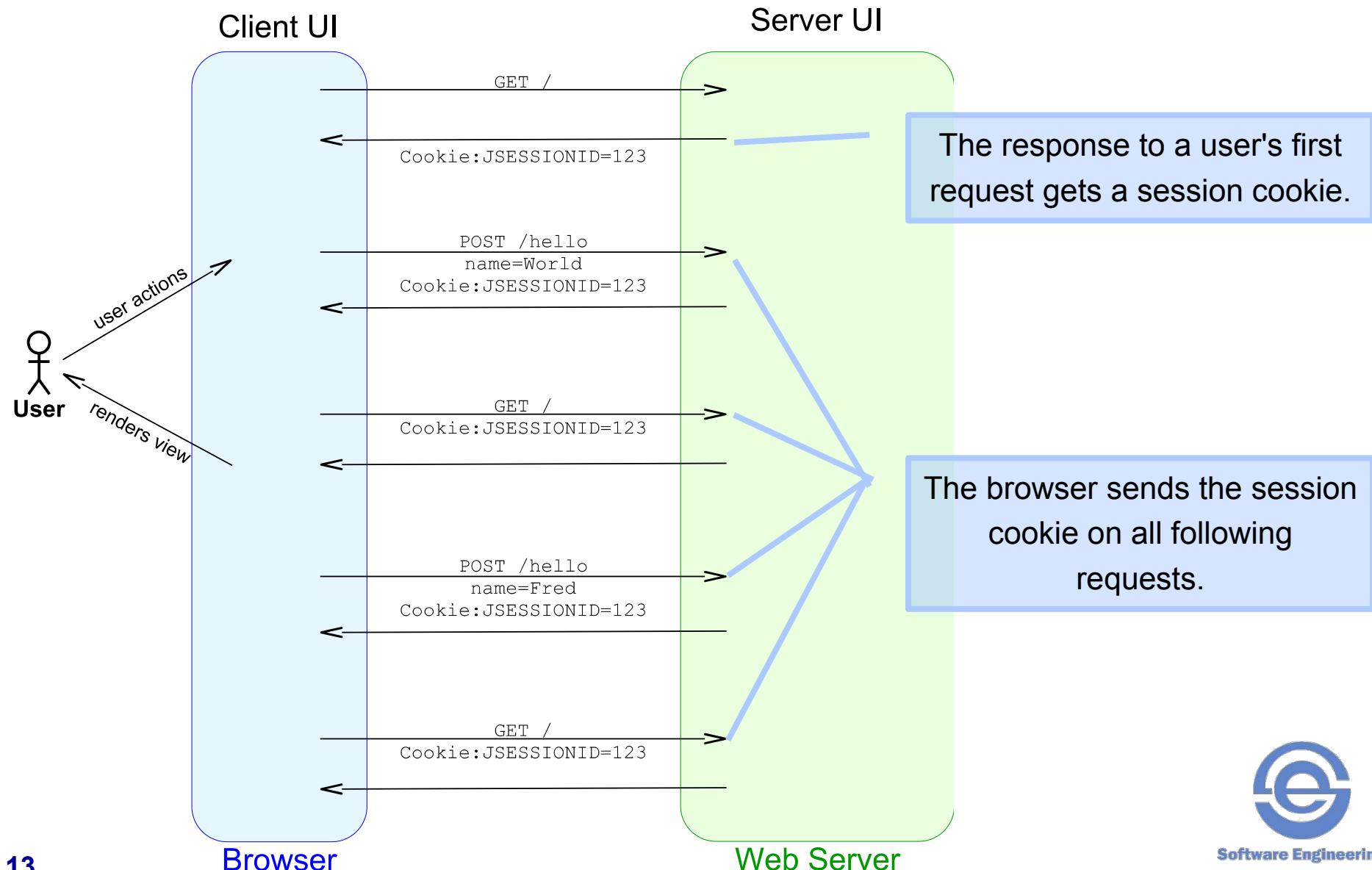
- View logic includes conditionals and loops.



# Most webapps need to connect all of a single user's requests together.



# Web application frameworks provide an HTTP cookie that identifies the user.



# To support keeping track of the names the Hello Post controller must store the name in the session.

- Here's the code

```
public class PostHelloRoute implements Route {  
    private final TemplateEngine templateEngine;  
    // constructor not shown  
    public Object handle(Request request, Response response) {  
        final String name = request.queryParams("name");  
        storeName(name, request.session());  
        final Map<String, Object> vm = new HashMap<>();  
        vm.put("pageTitle", "Greeting");  
        vm.put("name", name);  
        return tempEngine.render(new ModelAndView(vm, "greeting.ftl"));  
    }  
    private void storeName(String name, Session session) {  
        // on next slide  
    }  
}
```

Note the use of a private helper method  
to make the code more readable.



# You can define attributes in the Session object which can hold any object with any name.

- Here's the code to store the list of names:

```
public class PostHelloRoute implements Route {  
    public Object handle(Request request, Response response) {  
        // code on previous slide  
    }  
    private void storeName(String name, Session session) {  
        List<String> names = session.attribute("names");  
        if (names == null) {  
            // Initialize it  
            names = new ArrayList<>();  
            session.attribute("names", names);  
        }  
        names.add(name);  
    }  
}
```

- Limit how many attributes you put in the session.
- Pick meaningful attribute names.



# Now that we've stored the list of names in the session let's see how to use it.

- Here are the changes to the GetHomeRoute:

```
public class GetHomeRoute implements Route {  
    public Object handle(Request request, Response response) {  
        final Session session = request.session();  
        final Map<String, Object> vm = new HashMap<>();  
        vm.put("pageTitle", "Home");  
        vm.put("names", session.attribute("names"));  
        return tempEngine.render(new ModelAndView(vm, "home_v2.ftl"));  
    }  
}
```

- Change to the Home view:

```
<#if names??>  
    <ol>  
        <#list names as n>  
            <li>${n}</li>  
        </#list>  
    </ol>  
</#if>
```



# The Session object is like a Java Map object; it can store any number of named elements.

- The Java Map API:
  - *The put(key, value) method stores an element.*
  - *The get(key) method retrieves an element, or null if no element is found*
- The Spark Session API:
  - *The attribute(name, value) method stores an element.*
  - *The attribute(name) method retrieves an element, or null if no element is found*
- Use the Session object sparingly.

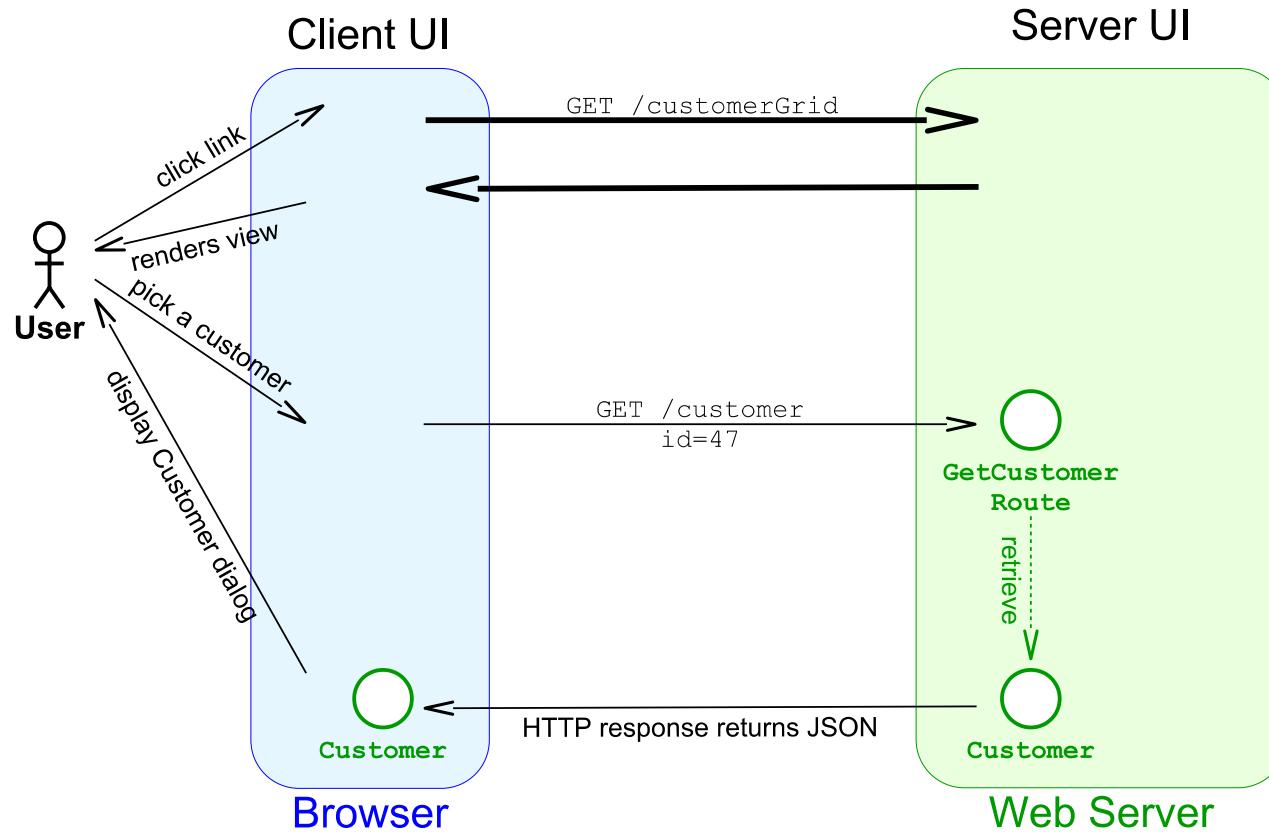


# Web 1.0 is server-oriented, while Web 2.0 is client-oriented.

- Web 1.0 is a vision of web development in which the server generates the view (the HTML).
- Web 2.0 is a vision in which the client is responsible for generating the view.
  - *This is done with manipulation of a Document Object Model (DOM).*
  - *And with Ajax calls to the server to exchange data*
- A hybrid approach is also possible.
  - *The term project uses a hybrid approach*
  - *You will need to understand how to build Ajax route handlers in Spark*



# Ajax is a technique for the browser to call the server without rendering a new page.



# Gson can parse the JSON strings in a client Ajax call to Java objects.

- An Ajax Route to insert a new Customer entity:

```
public class PostCustomerRoute implements Route {  
    private final Gson gson;  
    // constructor not shown  
    public Object handle(Request request, Response response) {  
        final String customerJSON = request.body();  
        final Customer customer =  
            gson.fromJson(customerJSON, Customer.class);  
        // TODO: add database insert code  
        return "Customer saved.";  
    }  
}
```

This object will be injected into the instantiated route object when it is constructed.

This will parse the received JSON request stored in customerJSON and return a Customer object with its attributes that match by name with attributes in the JSON request initialized to the value associated with the matching JSON attribute.



# The response to the Ajax call requires the route to convert a Java object into a JSON string.

- Gson is a Google library for JSON
- An Ajax Route:

```
public class GetCustomerRoute implements Route {  
    private final Gson gson;  
    // constructor not shown  
    public Object handle(Request request, Response response) {  
        // TODO: add database lookup code  
        return gson.toJson(new Customer(47, "Fred"));  
    }  
    // JSON would be: {id:47, name:"Fred"}  
}
```

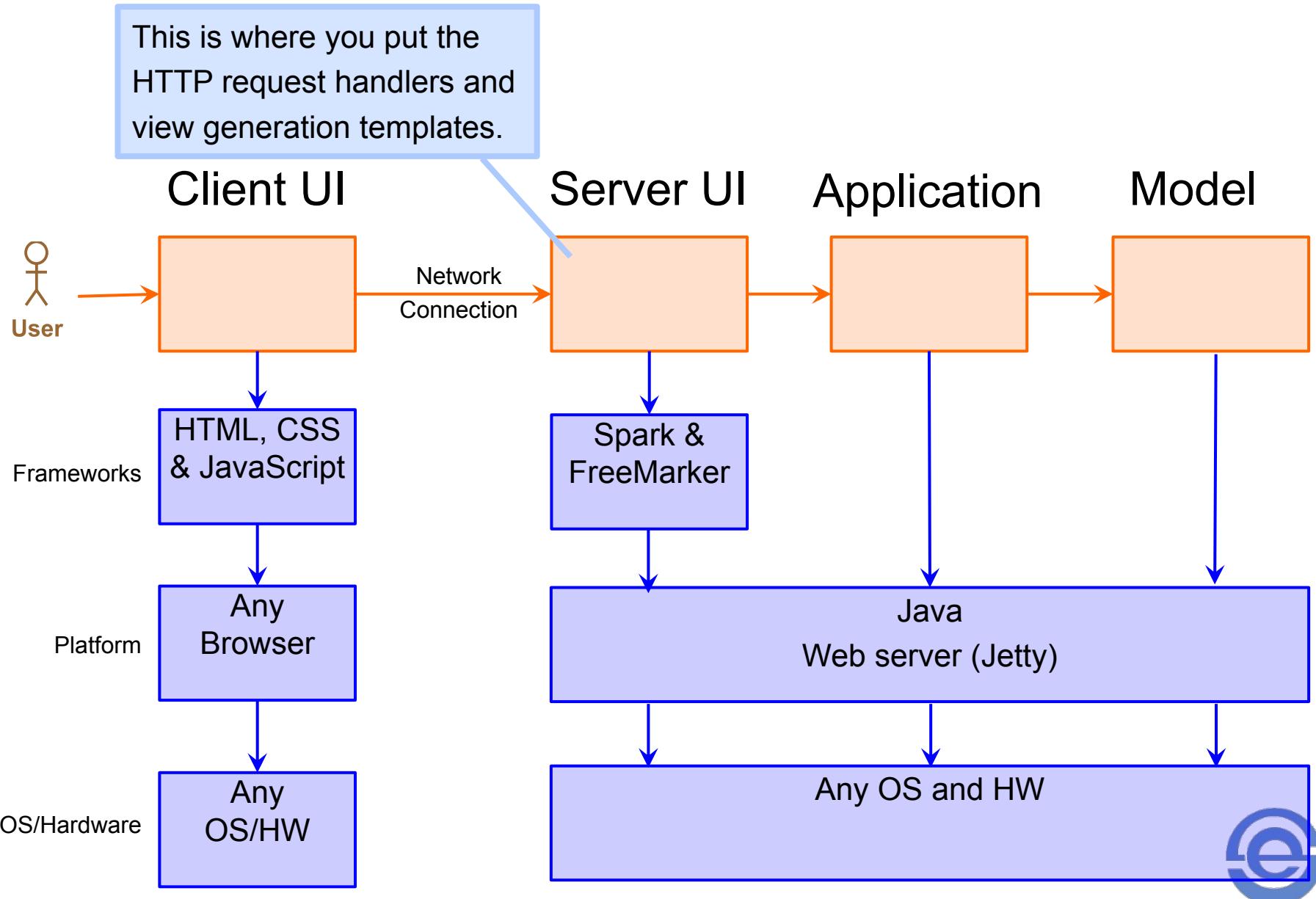
This will generate a JSON response instead of the HTML.

- How this route is configured:

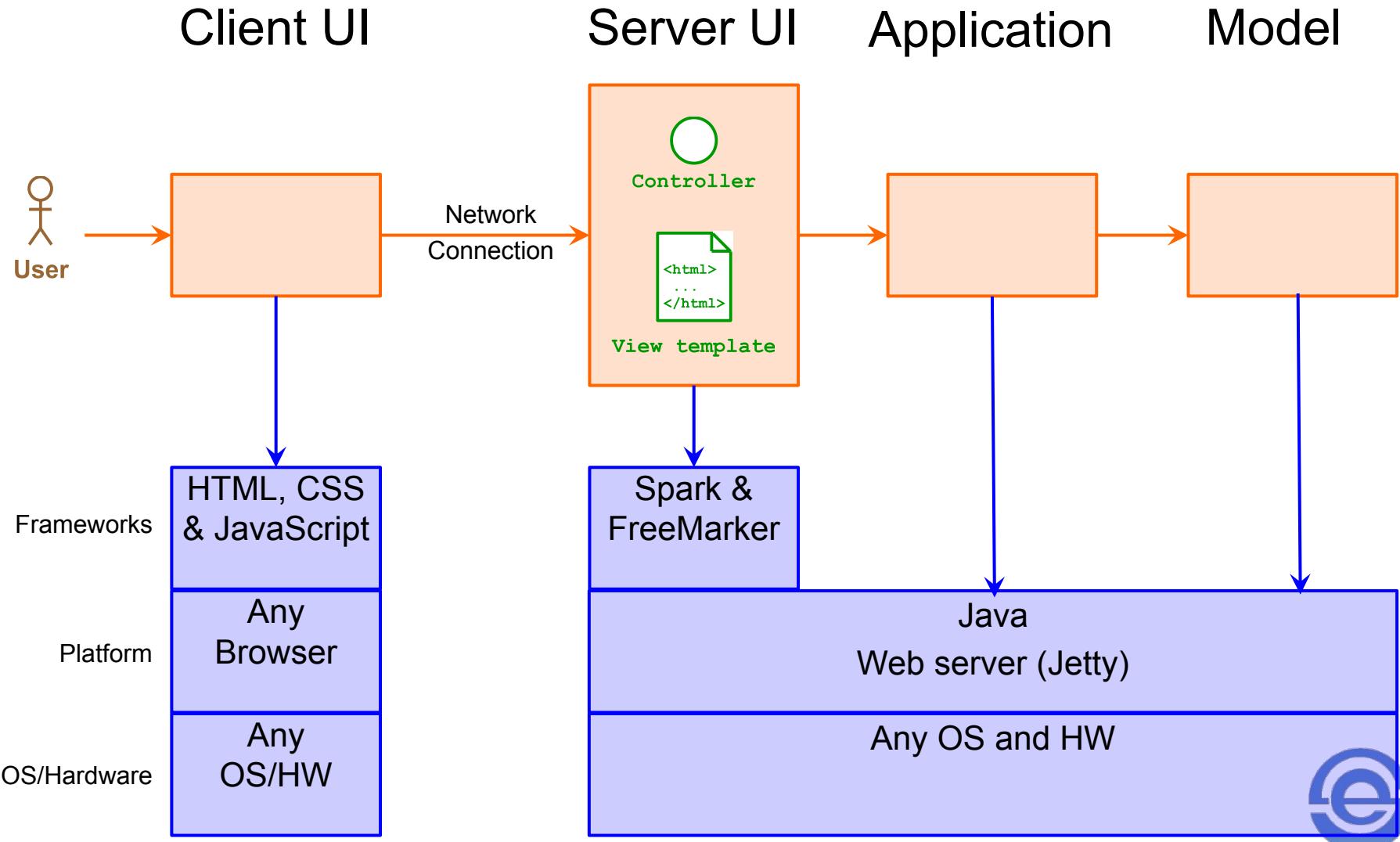
```
public class AjaxSampleApp {  
    public static void main(String[] args) {  
        final Gson gson = new Gson();  
        get("/customer", new GetCustomerRoute(gson));  
    }  
}
```



# Remember the architecture for the term project



# Now you have seen examples of Server UI components: views and controllers



# These are the responsibilities of UI components.

- UI Views

- *Provide an interface to the user*
- *Present information to the user in a variety of ways*
- *Provide a mechanism for user to input data and requests*

- UI Controllers

- *Control the views based on the state of the application*
- *Query the Application and Model tiers as necessary to get information to present to the user*
- *Perform simple input validation and data conversion based on input modality, e.g. String to Object*
- *Initiate processing of user requests/commands possibly providing data the user entered*
- *Perform data conversion for display by views*



# Maven is a build tool for Java applications.

- There have been many build tools over the years:  
UNIX make, Ant, Maven and Gradle.
- Maven provides these build services:
  - *Compile sources files*
  - *Download third-party libraries (such as Spark)*
  - *Assemble all files into an archive (JAR or WAR, etc)*
  - *Run test suite*
  - *Execute programs*
  - *Generate project reports*



# Maven provides a default project structure.

- The source code is in:
  - *src/main/java : holds your Java code*
  - *src/main/resources : holds all non-Java web resources files and FreeMarker templates*
- The test code is in:
  - *src/test/java : holds your Java test code*
- The build area is in the target directory.
- The pom.xml file provides the Project Object Model
  - *A description of your project*
  - *The third-party libraries to be included*
  - *Any plugins, such as testing or analysis tools*



# Maven is run from the command-line or from within your IDE.

- To build and assemble the project:

```
mvn compile
```

- To run a Java program:

```
mvn exec:java
```

- To run the project's test suite:

```
mvn test
```

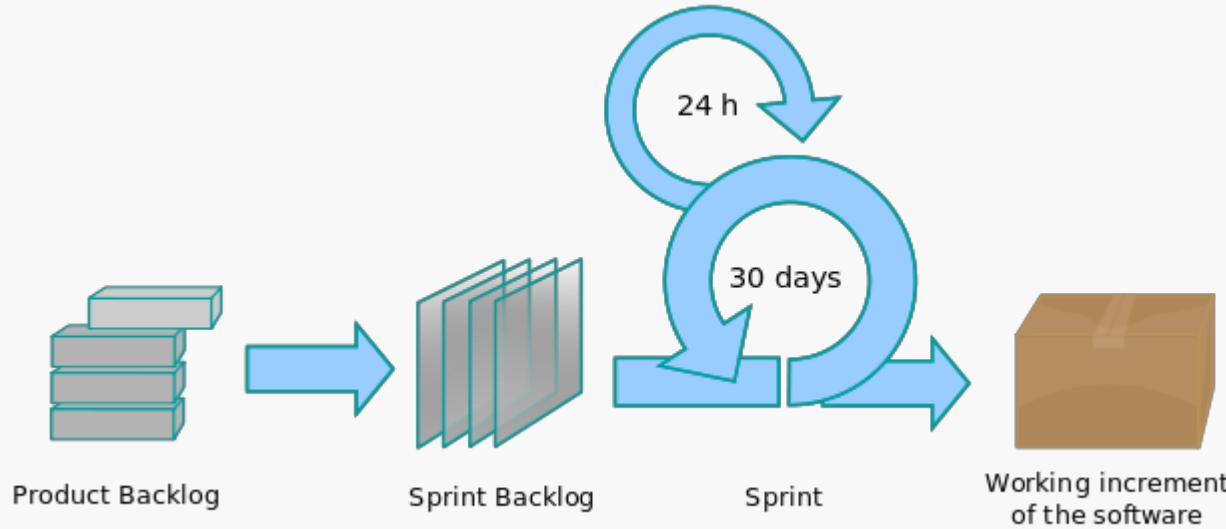


# Browsers have developer tools to help diagnose problems with a webapp.

- View DOM structure
  - *You can edit the DOM or change/add element attributes*
- View the CSS styles assigned to any given element
  - *You can edit CSS rules to see how that affects the visual aspects of an element*
  - *You can add new CSS rules on the fly*
- View the sequence of HTTP requests; including resources and Ajax calls
- View the JavaScript console
  - *There is also a REPL (read-eval-print loop)*



# Sprint Planning



By Lakeworks - Own work, GFDL, <https://commons.wikimedia.org/w/index.php?curid=3526338>

## SWEN-261 Introduction to Software Engineering

Department of Software Engineering  
Rochester Institute of Technology

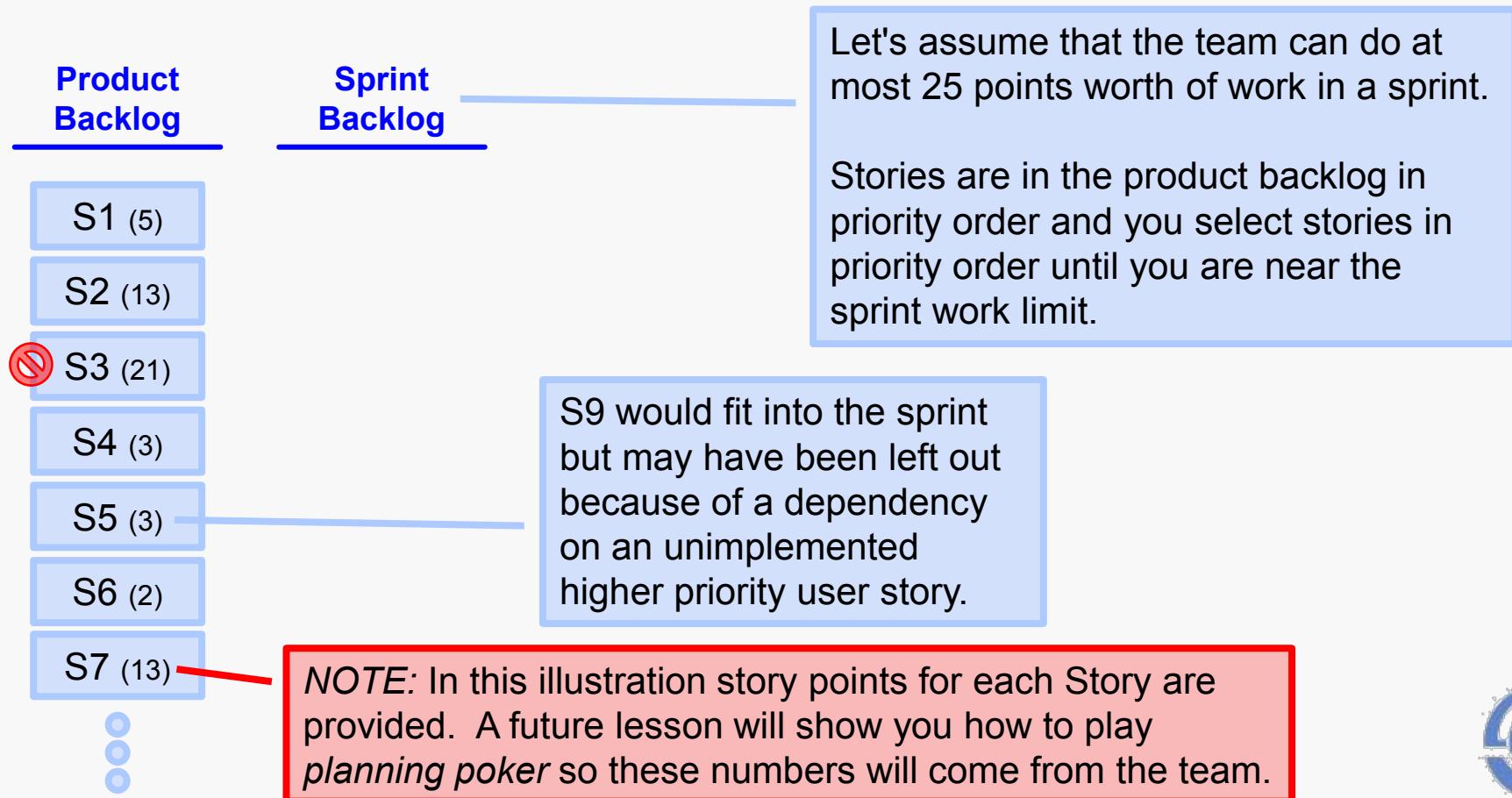
# A *sprint plan* is a plan to build a small, working increment of the product.

- Each sprint is time-boxed; your sprints will be about three weeks long
- Each sprint has a backlog of work to be done during the sprint
  - *This is the team's commitment to the Product Owner*
  - *It is the sole focus on the team's effort during this sprint*
- The sprint planning meeting establishes the next sprint's backlog and launches the sprint
- Let's first consider how a sprint flows...



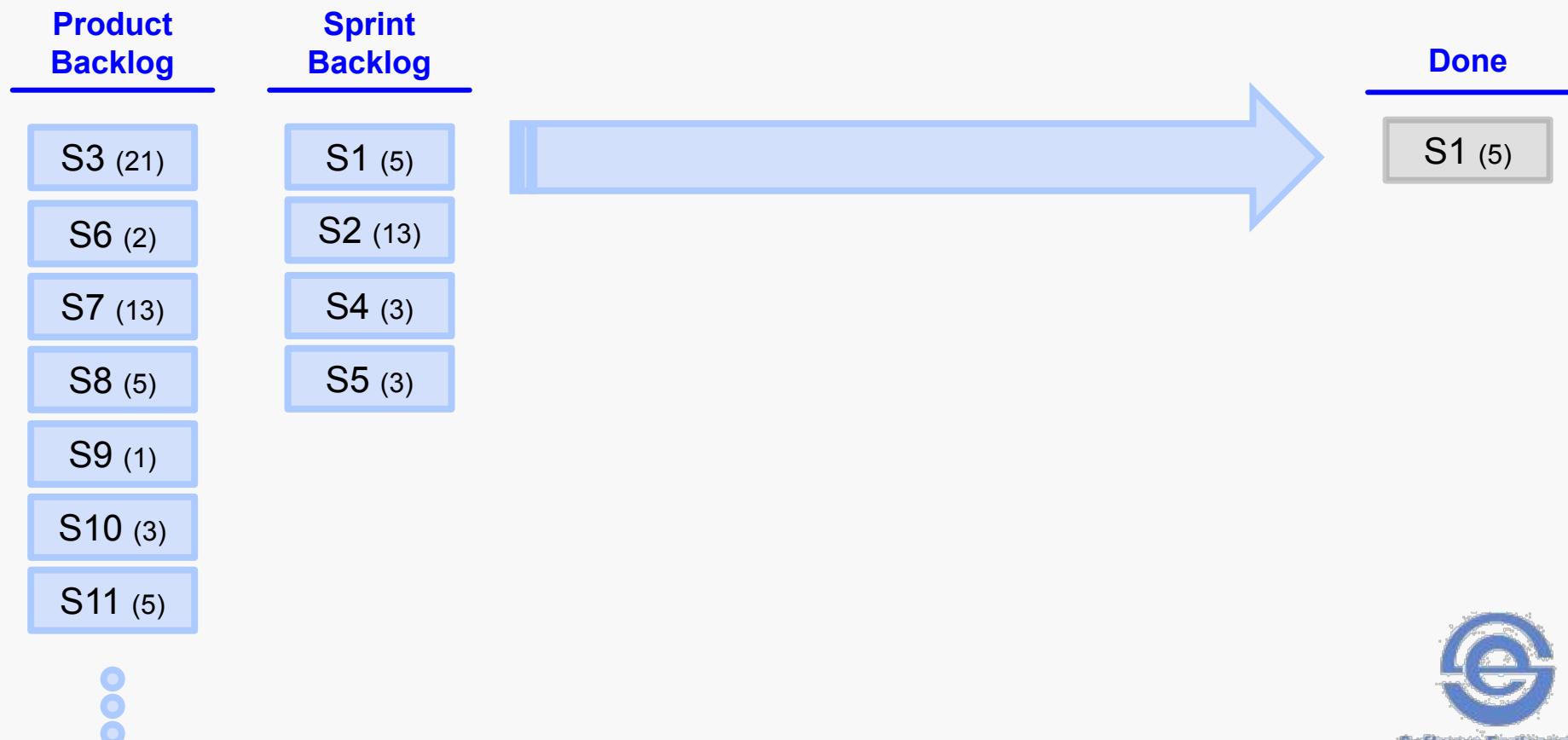
# So we start with a backlog for the new sprint.

- During the Sprint Planning meeting, stories from the Product Backlog are put on the Sprint Backlog.



# How does a story get from Backlog to Done?

- When do you know that you're done with a story?

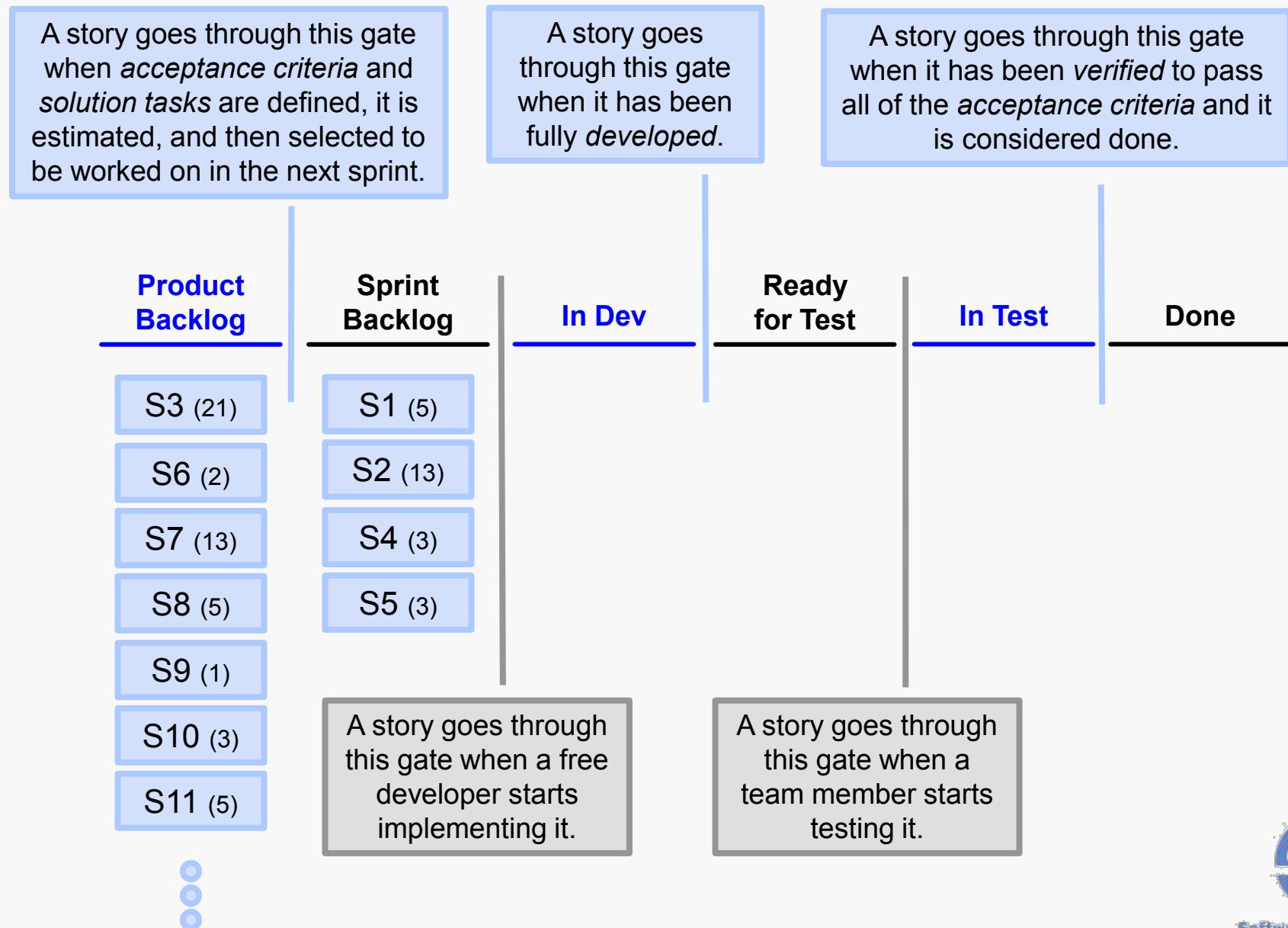


# The team must establish a *Definition of Done*.

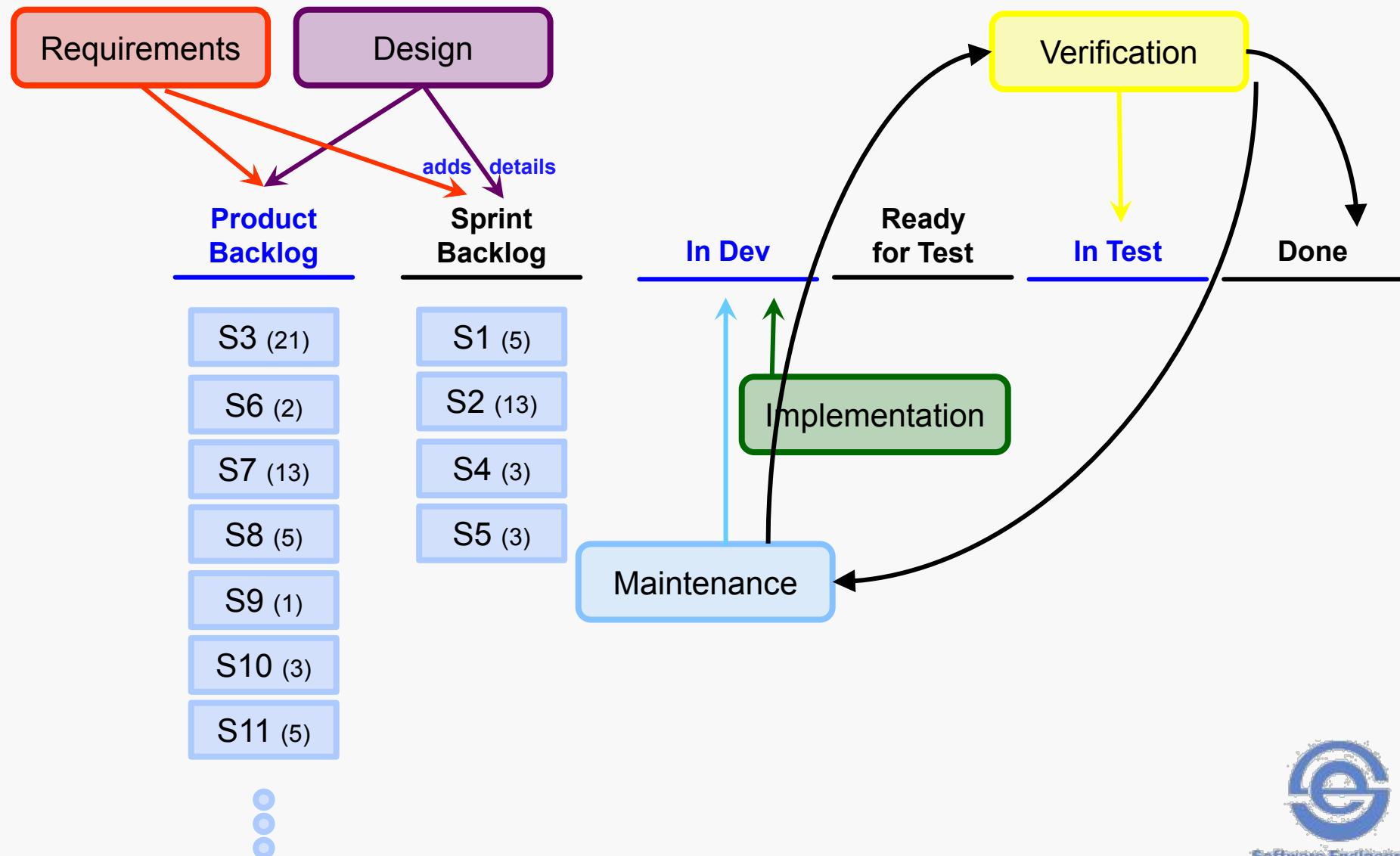
- Core activities:
  - *Acceptance criteria are defined*
  - *Solution is designed*
  - *Solution is developed; aka "code complete"*
  - *Feature has been tested (manually)*
- Other activities we'll add throughout the course:
  - *Feature branches (configuration management)*
  - *Unit testing*
  - *Code coverage analysis (goals)*
  - *Code reviews*
  - *Demo increment to stakeholders*



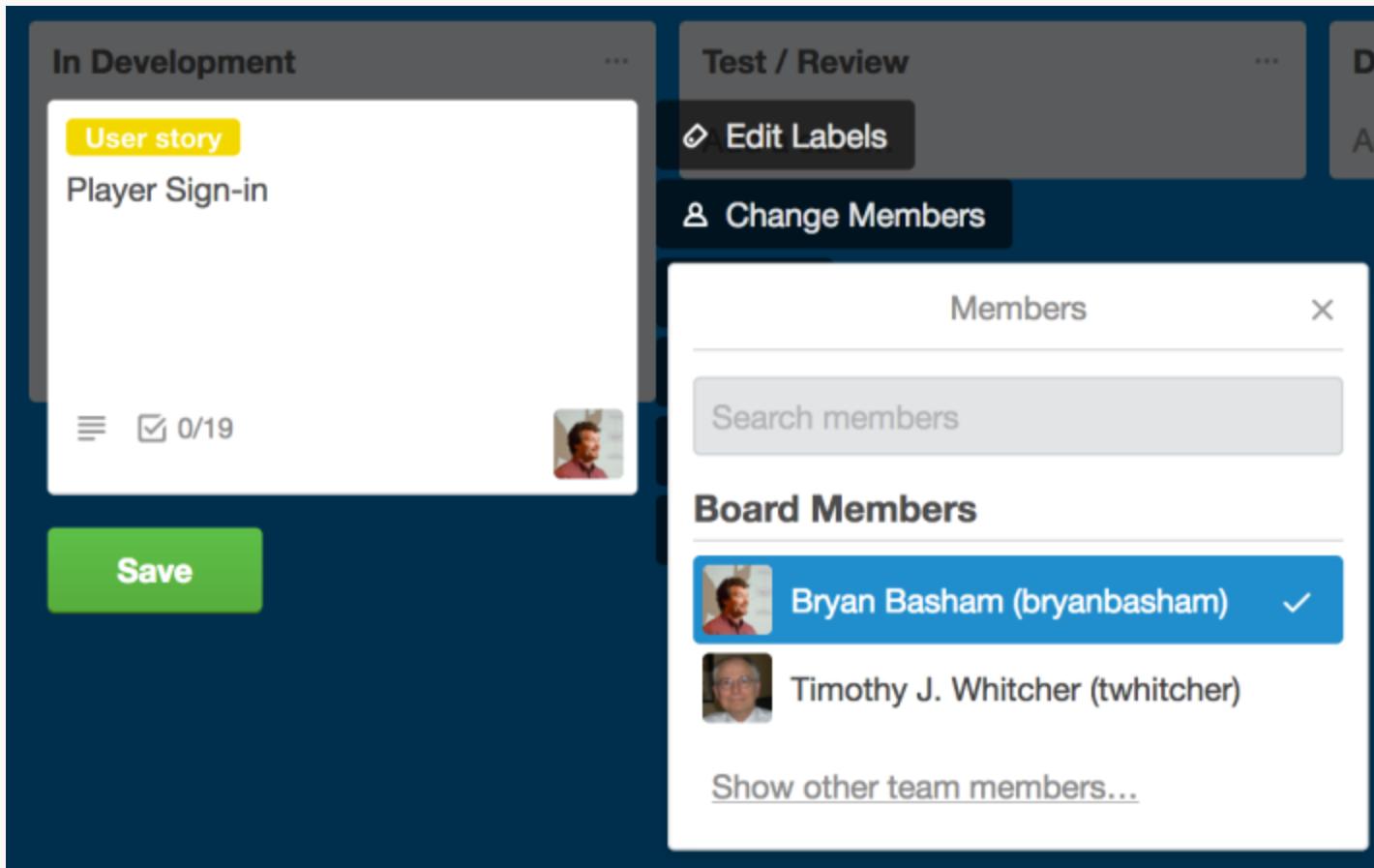
# A story goes through five "gates" to get to *Done*.



# A sprint holds a mini-waterfall of activities



# When members work on a Story they should be assigned to the card.



# As solution tasks are completed, the person doing the work checks off the task as done.

The screenshot shows a software development interface for a user story titled "Player Sign-in".

**User Story Details:**

- Members:** A placeholder icon with a plus sign.
- Labels:** A yellow button labeled "User story" with a plus sign.
- Description:** "As a Player I want to sign-in so that I can play a game of checkers."
- Story Points:** 5
- Feature Branch:** TBD

**Solution Tasks:**

- Status:** 25% complete (indicated by a blue progress bar).
- Tasks:**

  - Create the `GET /sign-in` Route component to render the sign-in page template.
  - Create the `sign-in.ftl` template with a text field and one button. The form action is a `POST` to the `sign-in` URL.
  - Create the `POST /sign-in` Route component to process the sign-in request.
  - Create the `Player` Model tier entity to hold the unique player name.



# Check-off *Definition of Done* items as the work gets completed.

The screenshot shows a user story card for "Player Sign-in" in the "In Development" list. The card includes fields for Members (with a placeholder profile picture), Labels, Description (with an "Edit" link), Story text ("As a Player I want to sign-in so that I can play a game of checkers."), Story Points (5), Feature Branch (TBD), and a Definition of Done Checklist. The checklist has a progress bar at 33% completion, with three items checked: "code-complete", "manually test the story", and "prepare for the feature demo".

Player Sign-in  
in list In Development

Members Labels

User story +

Description Edit

**Story**

*As a Player I want to sign-in so that I can play a game of checkers.*

**Story Points**

5

**Feature Branch**

TBD

**Definition of Done Checklist** Hide completed items Delete...

33%

- code-complete
- manually test the story
- prepare for the feature demo



# The tester checks off *Acceptance Criteria* as they validate the behavior of the system.

Player Sign-in  
in list Sprint Backlog

Labels

User story +

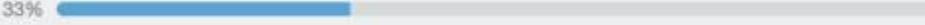
Description Edit

### Story

*As a Player I want to sign-in so that I can play a game of checkers.*

### Story Points

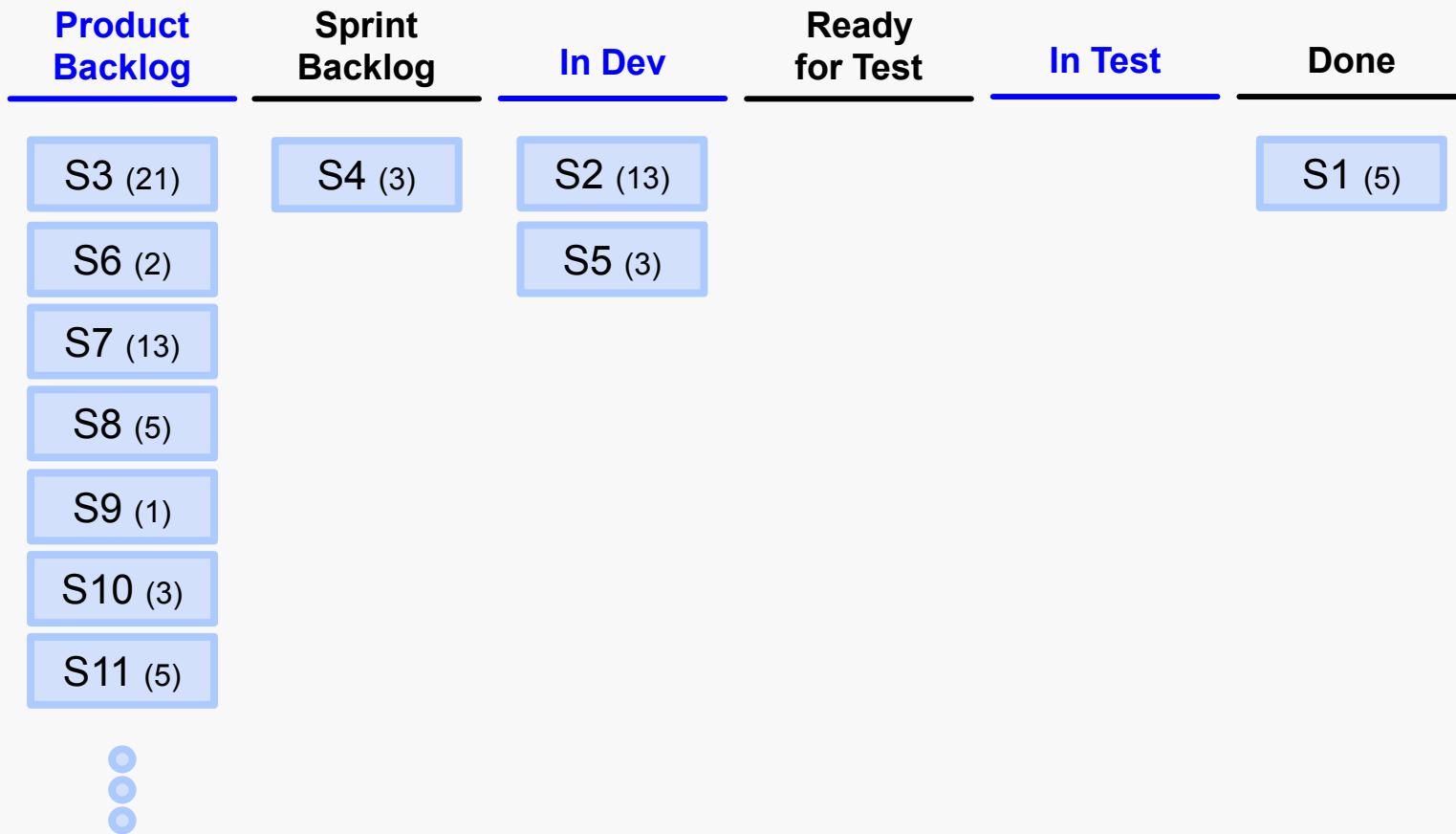
8

**Acceptance Criteria** Hide completed items Delete...  
33% 

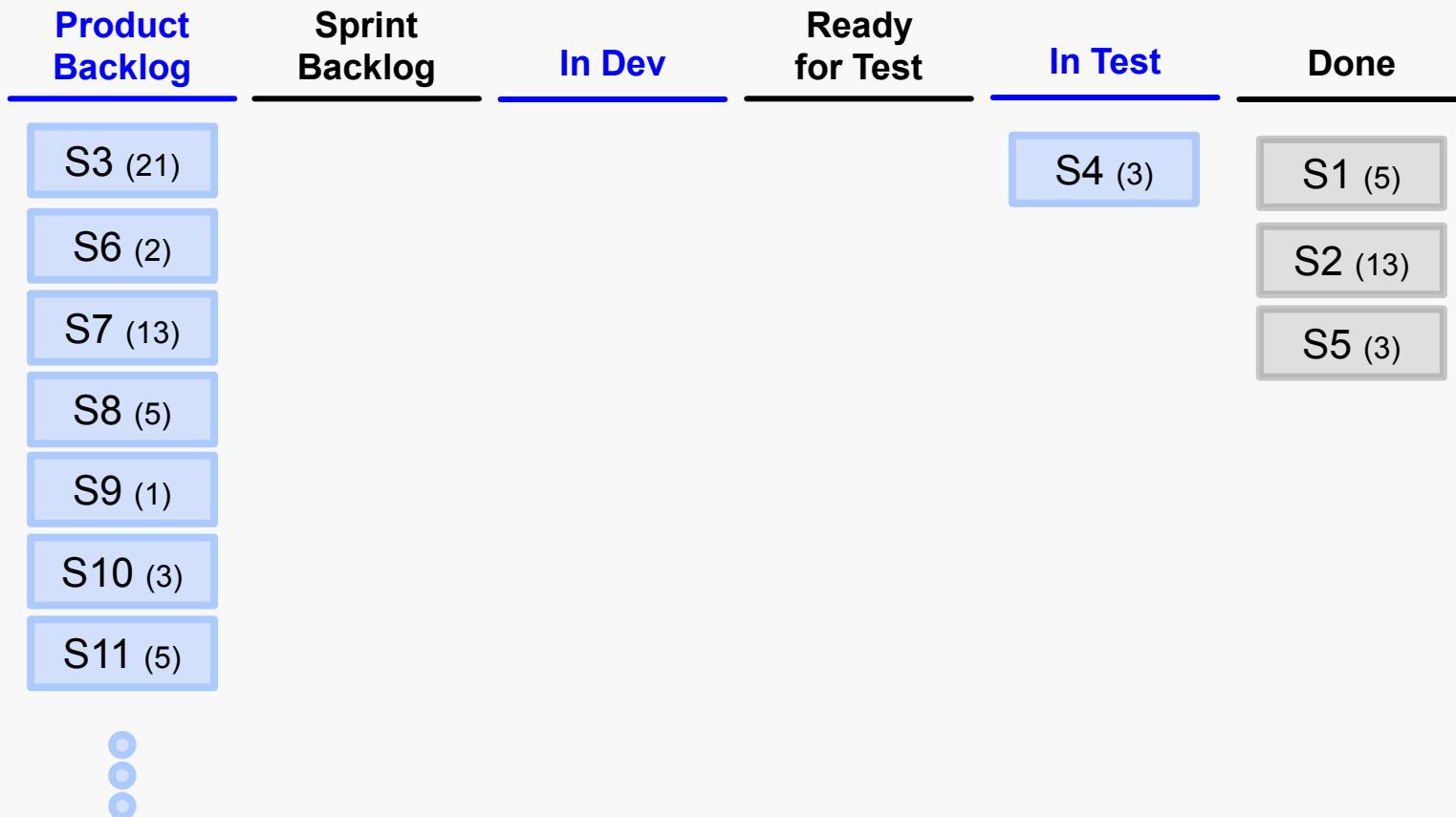
- Given that I have not yet signed-in when I see the Home page then I must see a means to sign-in. (such as a link or button)*
- Given that I am not signed-in when I do click on the sign-in link then I expect to be taken to the Signin-page, with a means to enter a player name.*
- Given that no one else is using my name when I enter my name in the sign-in form and click the Sign-in button then I expect system to reserve my name and navigate back to the Home page.*
- Given that no one else is using my name when I enter a name with a double quote ( " ) then I expect the system to reject this name and return the sign-in form. IOWs, double quote is not a valid character in a Player's name.*



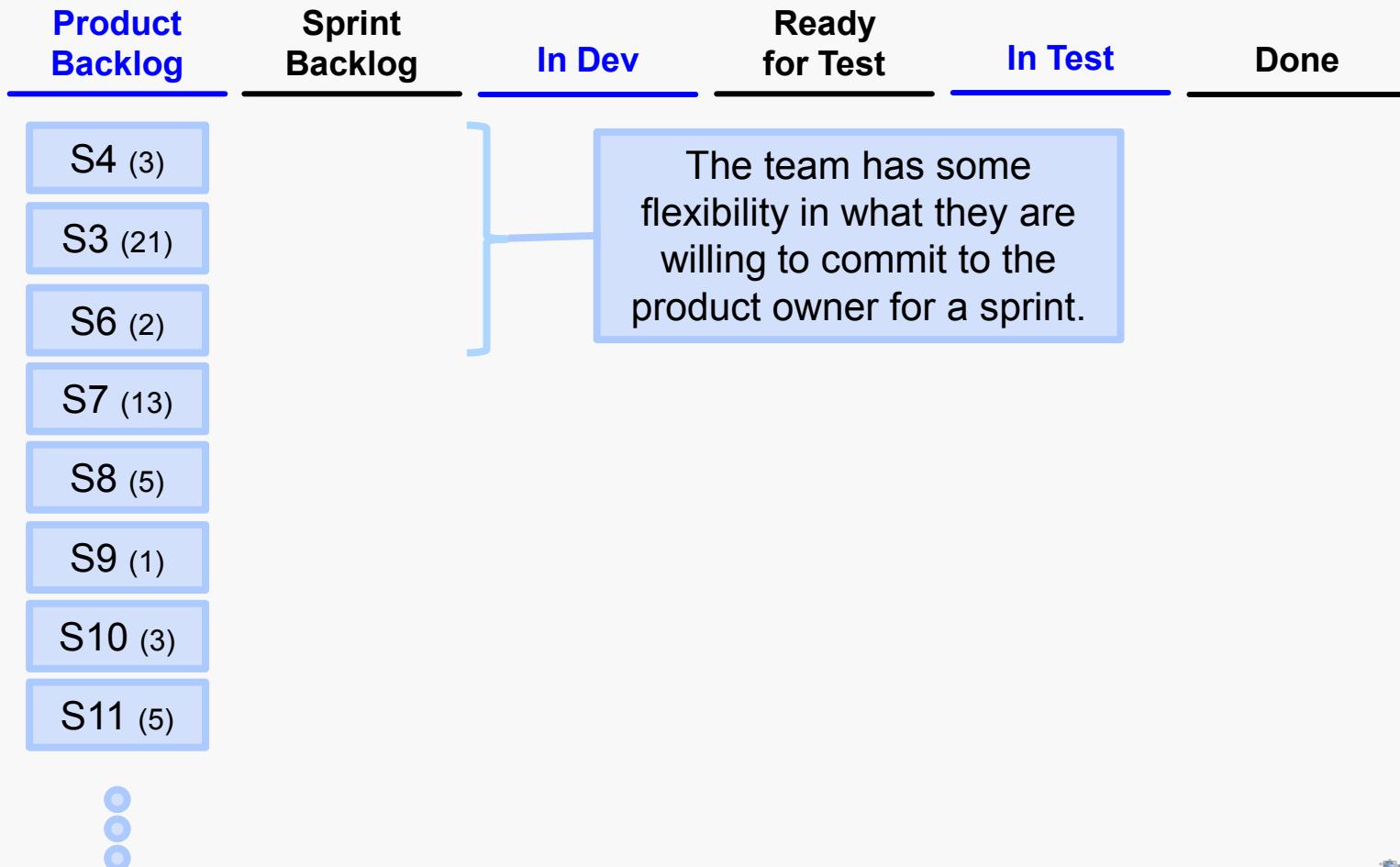
# On any given day, the sprint board tells us the status of all sprint work.



# At the end of a sprint, unfinished work gets put back onto the Product Backlog.



# And the process starts all over again at the start of the next sprint.



# But how do you really know how many story points the team can accomplish in a sprint?

- Velocity is measured from the average of the last three sprints.
  - *Only use the completed stories.*
  - *This rolling average will change over time.*
- The average velocity is then used to cap the number of story points for the next sprint.
- Example:
  - *Sprint 7: 45 story points committed; 42 completed.*
  - *Sprint 8: 40 committed; 50 completed*
  - *Sprint 9: 48 committed; 47 completed*
  - *Velocity = (42 + 50 + 47) / 3 ~= 46.3333*
  - *Sprint 10 will be capped at 46 story points.*



# Velocity is specific to one team working on one project.

- This assumes that sprint length and team membership remain consistent.
  - *If either of these two change, then velocity measurement must start over with a new running average.*
- Velocity is only measured for a single project, single team.
  - *Story points are level of effort estimates*
  - *Story points are determined by the team, for the team*
  - *Thus you cannot compare velocity's between teams*
- Management cannot **set** a team's velocity.

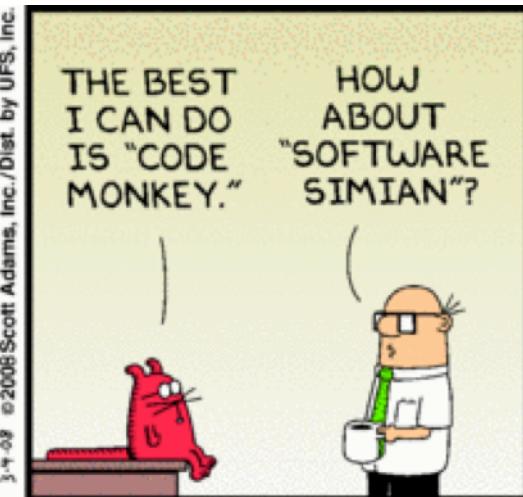


# Velocity is not the same as the team's overall capacity.

- Velocity is only a measure of effort for working on user stories.
  - *It does not include company meetings, email communication, small "outside" tasks*
  - *For class, it does not include most before- and after-class activities*
- What about these issues?
  - *Holidays or vacations*
  - *Members given large, outside tasks*
- Normally these issues can be ignored being smoothed out by the averaging process.
- In extreme cases, the team can make adjustments to the calculated velocity usually by lowering it.



# Appreciation for Software Architecture



## SWEN-261 Introduction to Software Engineering

Department of Software Engineering  
Rochester Institute of Technology



Software Engineering  
Rochester Institute  
of Technology

# Object-Oriented Design I

**SWEN-261**  
**Introduction to Software**  
**Engineering**

Department of Software Engineering  
Rochester Institute of Technology

Single responsibility  
High cohesion

Information expert  
Low coupling

Law of Demeter

Dependency inversion



# Up to this point, you have studied object-oriented design mostly at the class level.

- This set of skills needs to be expanded to design larger scale systems.
- You need to consider the interactions between classes and the effect of classes on other classes.
- The software engineering community has put forward sets of design principles to follow.
  - **SOLID** (*Bob Martin, Principles of OOD*)
  - **GRASP** (*Craig Larman, Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development.*)
- We will look at some of these principles, along with the Law of Demeter, in two lessons.



# SOLID and GRASP provide two sets of object-oriented design principles.

## SOLID

- Single responsibility
- Open/closed
- Liskov substitution
- Interface segregation
- Dependency inversion

## GRASP

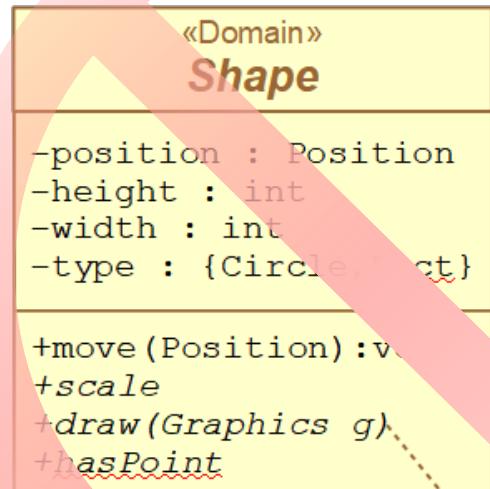
- Controller
- Creator
- Indirection
- Information expert
- High cohesion
- Low coupling
- Polymorphism
- Protected variations
- Pure fabrication

Law of Demeter

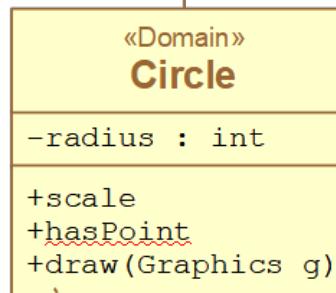
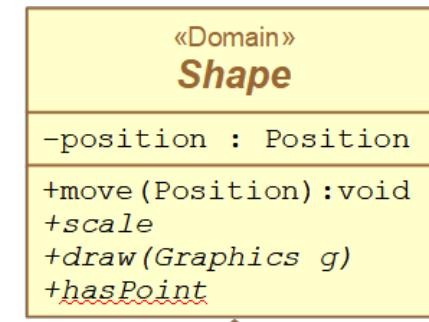


# The *Single responsibility principle* is perhaps the most important object-oriented design principle.

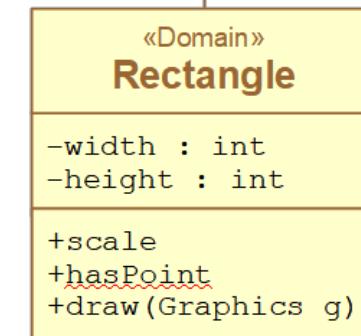
A class should have only a *single responsibility*.



Draw the shape based on its type. A circle has no width or height.



Draw a circle using the graphics object.



# The *Single responsibility principle* will lead to smaller classes each with less responsibility.

*A class should have only a single responsibility.*

- A class should have a single, tightly focused responsibility.
- This leads to smaller and simpler classes, but more of them.
  - *Easier to understand the scope of a change in a class.*
  - *Easier to manage concurrent modifications.*
  - *Separate concerns go into separate classes.*
- Helps with unit testing.



# ***High Cohesion* aims for focused, understandable, and manageable classes.**

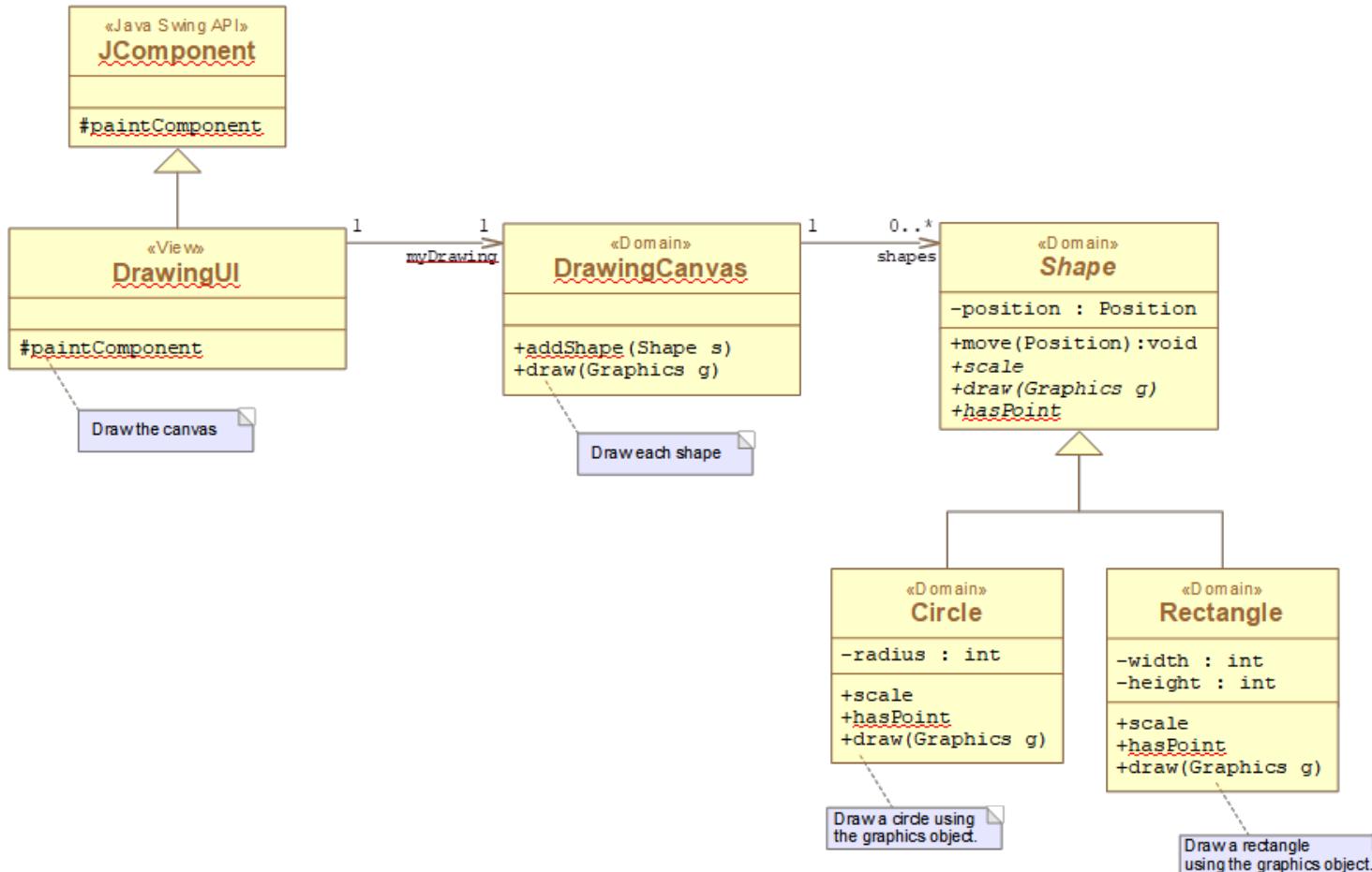
*Assign responsibility so the cohesion of classes remains high.*

- High cohesion leads to smaller classes with more narrowly defined responsibilities.
- This design goal should have a higher priority than most other goals.



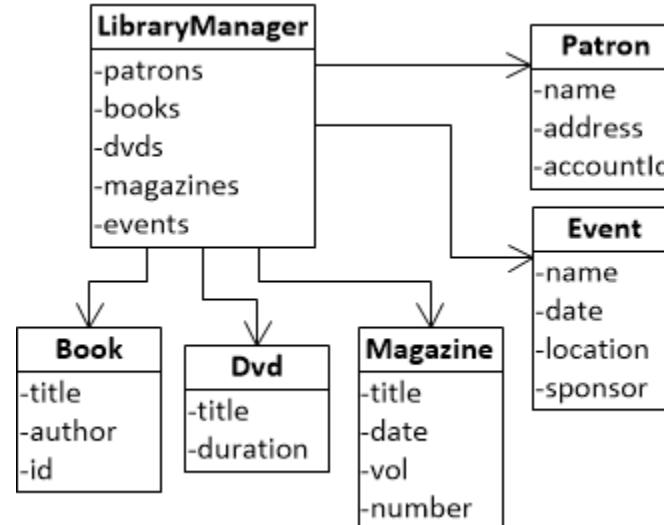
# High Cohesion vs. Low Coupling

- High cohesion will require coupling to more classes to get work accomplished.
  - *Do not be afraid of requiring a few more relationships in the pursuit of improved cohesion.*

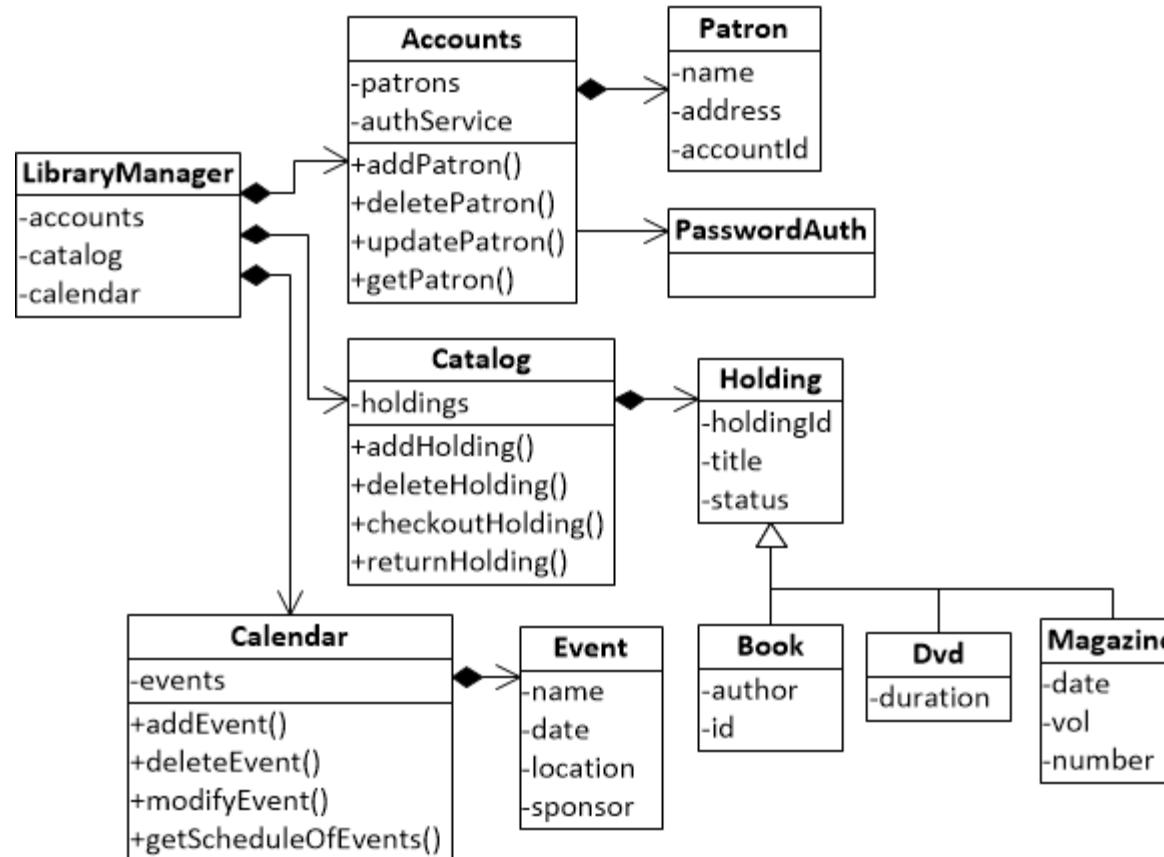


# Consider that you are implementing a library management system.

- You could place most of the functionality into a `LibraryManager` class.
- This class would have too many responsibilities.
  - *Maintaining the library catalog*
  - *Maintaining patron accounts*
  - *Scheduling library events*



# Separate the concerns into more classes each with a single highly focused responsibility.



# Make a class as simple as possible, but not simpler.

*Everything should be made as simple as possible, but not simpler. - Einstein*

- Aim to implement the behaviors that directly work with the class' attributes.
- Consider what clients will want to do with the attribute data—put those behaviors in the class.
- If you are a client doing processing with the attribute data, consider putting your operation in the class.



# Some classes are more complex than you think they are.

- Consider that you are building an airline flight reservation system.
- A **Flight** entity will definitely be in the domain model and be a class in your implementation.
- You could consider this to be only a data holder class with no other behavior.
- This would lead to something like →

Flight
-departure : unsigned int
-arrival : unsigned int
-destination : String
-origin : String
+getDeparture() : unsigned int
+getArrival() : unsigned int
+getDestination() : String
+getOrigin() : String



# Student project code often does not do right by the client of their classes.

Flight
-departure : unsigned int
-arrival : unsigned int
-destination : String
-origin : String
+getDeparture() : unsigned int
+getArrival() : unsigned int
+getDestination() : String
+getOrigin() : String

Note: Time is stored in 24 hour notation.

- A client of Flight had this code:

- `flt.getDestination().equals("JFK")`
- `flt1.getOrigin().equals("ROC") && flt1.getDestination().equals("JFK")`
- `flt.getArrival() < flt.getDeparture()`
- `flt1.getArrival() + 60 < flt2.getDeparture()`

- Why does the client of Flight have to do this "heavy-lifting"?



# **Information Expert** looks to have behavior follow data.

*Assign responsibility to the class that has the information needed to fulfill the responsibility.*

- The first place to consider placing code that uses/processes attribute data is in the class that holds the attributes.
- Instead of the client of Flight implementing this:
  - `flt.getDestination().equals("JFK")`
  - `flt1.getOrigin().equals("ROC") && flt1.getDestination().equals("JFK")`
  - `flt.getArrival() < flt.getDeparture()`
  - `flt1.getArrival() + 60 < flt2.getDeparture()`
- Consider Flight as the Information expert

- `boolean Flight.destinationIs(String airportCode)`
- `boolean Flight.itineraryIs(String originCode, String destinationCode)`
- `boolean Flight.arrivesNextDay()`
- `boolean Flight.canConnectWith(Flight nextFlight)`



# **Low Coupling** attempts to minimize the impact of changes in the system.

*Assign responsibility so that  
(unnecessary) coupling remains low.*

- Note the unnecessary word. Coupling is needed in your system.
- Resist lowering coupling simply to reduce the number of relationships.
  - *A design with more relationships is often better than the design with fewer relationships.*
  - *You need to balance all the design principles.*
  - *Beginning designers often place low coupling at the top of their design principles list. It should be lower down!*



# The Law of Demeter addresses unintended coupling within a software system.

- Limit the range of classes that a class talks to
  - *Each unit only talks to its friends; don't talk to strangers.*
  - *Each unit only talks to its immediate friends; don't talk to friends of friends*
  - *Chained access exposes each intermediate interface*
- From a previous checkers project
  - `board.getPieceAt(i, j).getType()` → *eliminate violation of Law of Demeter and reduce coupling for this class, i.e. may not need to know about Piece*
  - `board.getPieceTypeAt(i, j)` or *predicate*
- If you need to talk to something "far away"
  - *Get support from your friend, i.e. new method*
  - *Get a new friend, i.e. new direct relationship*



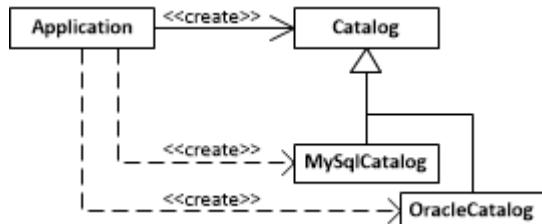
# The *Dependency inversion* principle provides looser coupling between dependent entities.

*High-level modules should not depend on low-level modules. Both should depend on abstractions.*

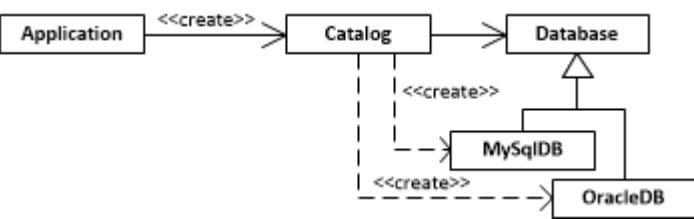
- A common manifestation of this is *dependency injection*.
  - *The low-level module does not have responsibility for instantiating a specific dependent class.*
  - *The high-level module injects the dependent element.*
  - *The low-level module is only dependent on the (high-level) abstraction not the (low-level) implementation.*
  - *The injection can be during construction, using a setter, or as a parameter in an operation method.*
  - *Critical for doing unit testing since we can inject test/mock objects.*



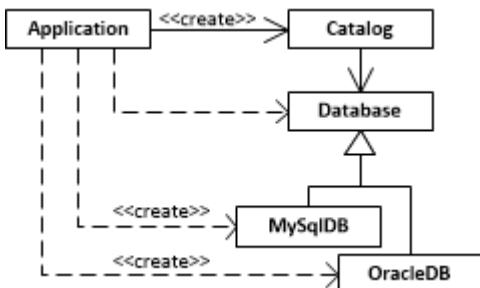
# Here is how an application's design might evolve to incorporate dependency injection.



- Higher level has responsibility for instantiation.
- Design does not adequately separate the concerns of catalog operations vs database operations.
- It will be difficult to unit test just catalog operations.



- Lower level has responsibility for instantiation.
- It will be difficult to unit test the Catalog class.



- Higher level has responsibility for instantiation of the specific database implementation.
- It injects this Database dependency into the Catalog when it is instantiated.
- Catalog only deals with higher level Database abstraction.
- Use test version of Database to test Catalog.



# Why is architecture important?

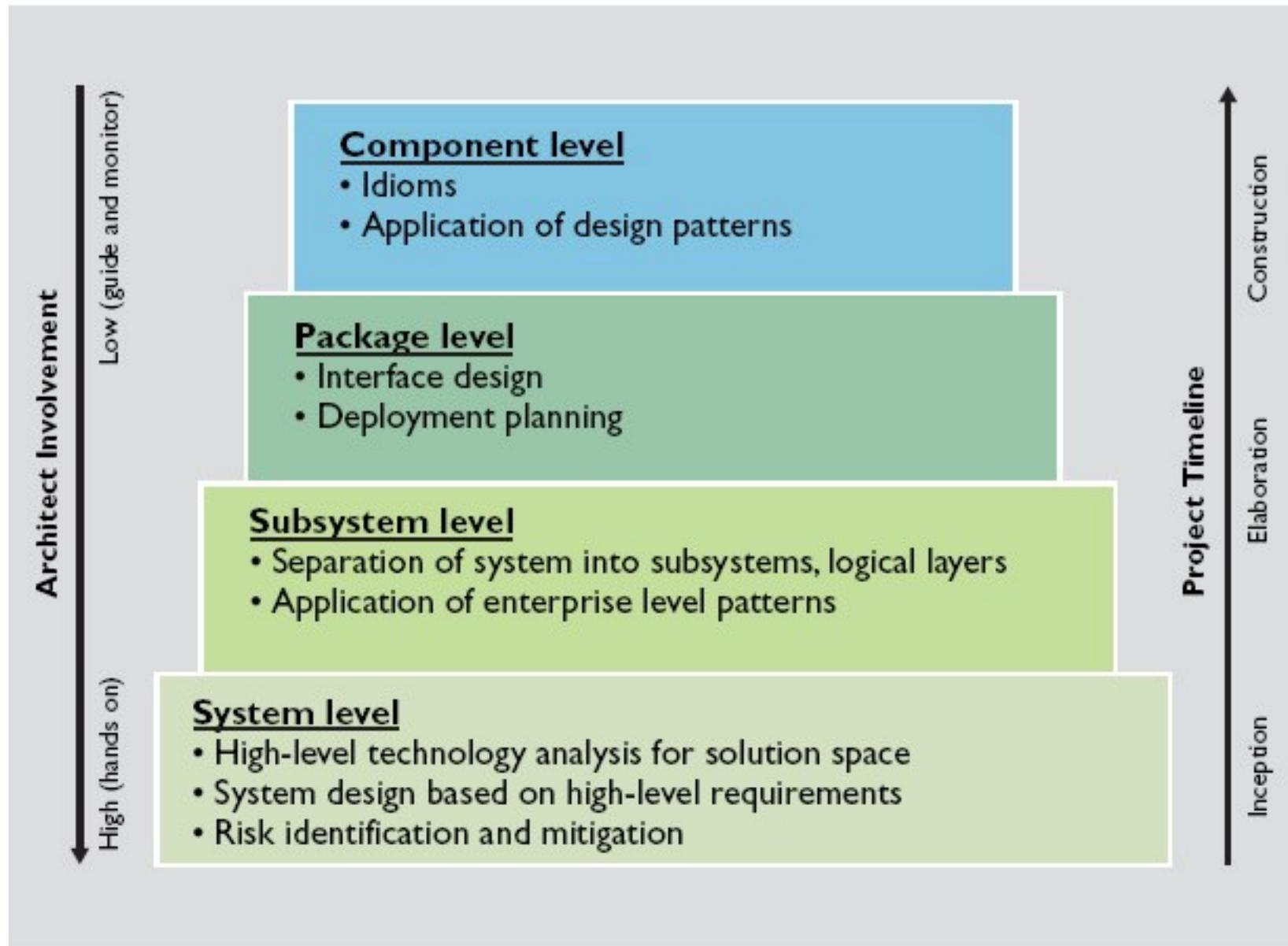
- Martin Fowler in *Patterns of Enterprise Application Architecture*:

*“The highest-level breakdown of a system into its parts; the decisions that are hard to change; there are multiple architectures in a system; what is architecturally significant can change over a system's lifetime; and, in the end, architecture boils down to whatever the important stuff is.”*

- **Solution communication and consensus** among stakeholders
- **Earliest** and most **fundamental design analysis** and decisions
  - *Directs and constrains remaining software development, deployment, and maintenance*
  - *Dictates structure of development organization*
  - *Enables early evolutionary prototyping*
  - *Enables more accurate cost and schedule estimation*



# Architecture is the highest-level design of a system



# Requirements affect the system architecture

- Non-functional requirements (NFRs) and constraints lead to a logical and physical architecture.
- Operational NFRs:
  - *Scalability*
  - *Availability* ...
- Developmental NFRs:
  - *Testability*
  - *Portability* ...
- Constraints:
  - *Pre-chosen system components (eg, database)*
  - *Pre-chosen frameworks* ...



# Architecture is guided by principles and patterns

- Manage risk
- Build vs buy vs open source
- Separation of concerns
- Architectural patterns are selected to satisfy NFRs
  - *Failover* and *Load Balancing*
  - *Model-View-Controller*
  - *Tiers and Layers*
    - ◆ For example: UI, Application and Model
  - *Java EE Patterns*
- No one architecture is right or wrong, just more or less useful for a given application. (attribution unknown)
  - *Does it satisfy the NFRs?*
  - *Ad-hoc architecture is not very useful.*



# Visualize the architecture in the Inception phase

WebCheckers Inc hired a contract architect who produced the Vision document:

## Architecture

First, the application must be on the web.

The WebCheckers webapp will use a Java-based web server. The team will use the Spark web micro framework and the FreeMarker template engine to handle HTTP requests and generate HTML responses. The team has access to the standard Java v9 libraries and language features. The team may also include additional, third-party frameworks. The architect has also identified the Product Owner.

## Process

The team will use the OpenUP method for planning activities across the life span of the project. The team will use the Scrum process for day-to-day operations.

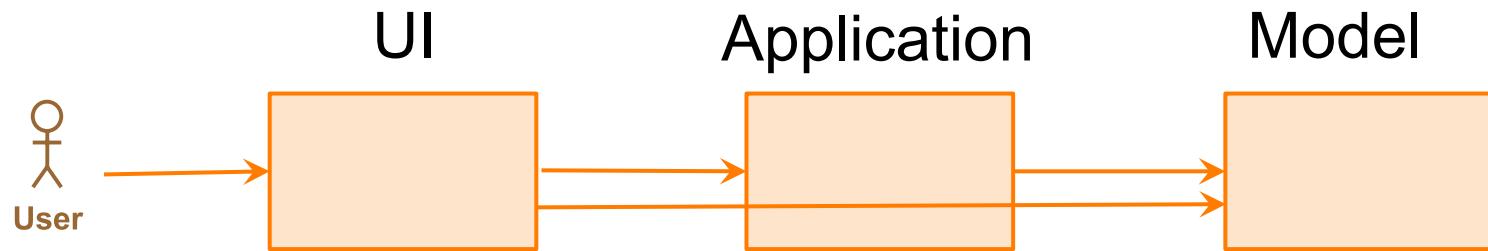


# Build out the architecture in the Elaboration phase

- The architecturally-significant user stories are prioritized during the Elaboration.
- The development team is frequently guided or lead by an architect during this phase.
- The working increment at the end of Elaboration forms the starting point of the system architecture.
  - *There will be architectural additions over the lifetime of the system.*
  - *Avoid changing established architectural norms.*



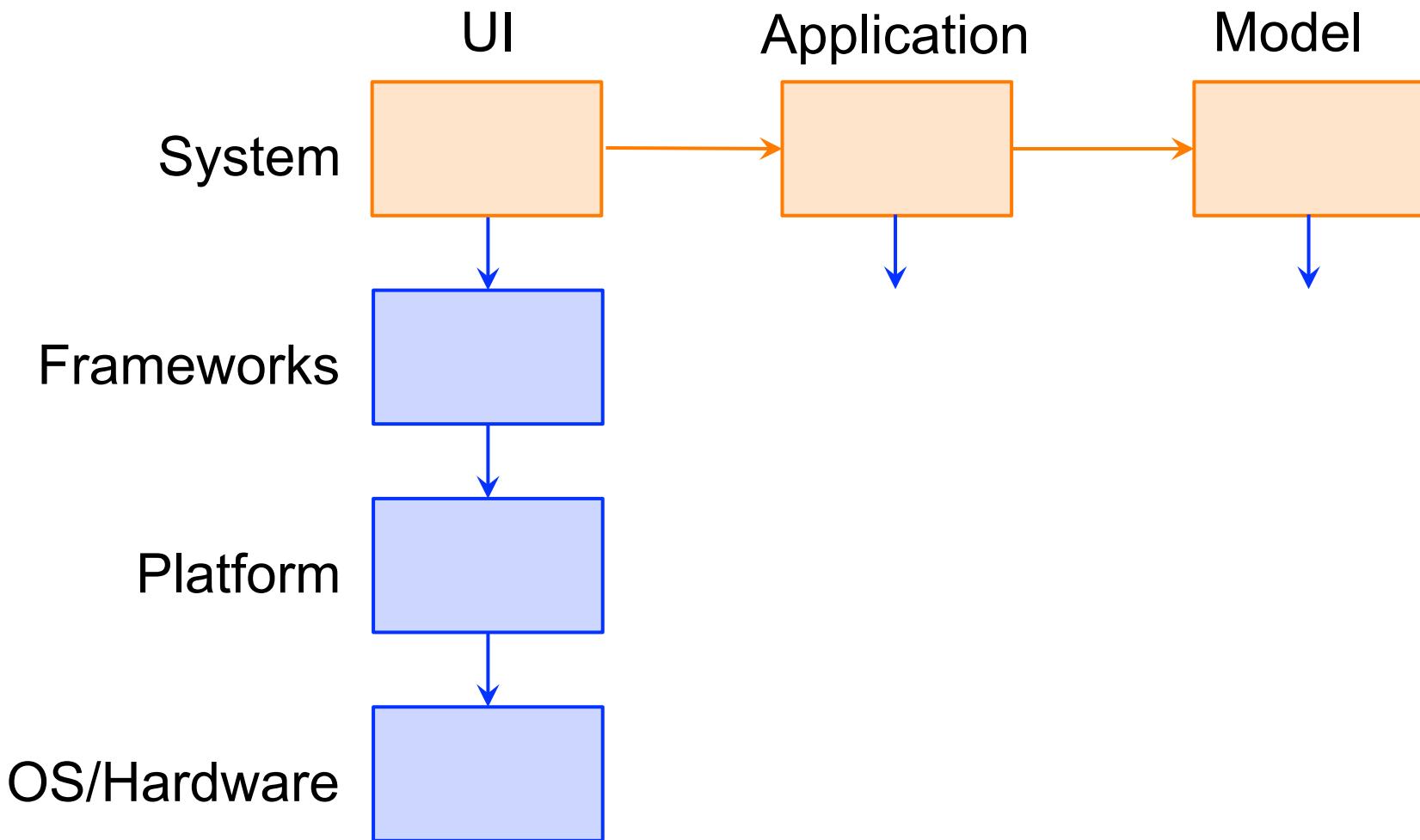
# Architecture is modeled using tiers



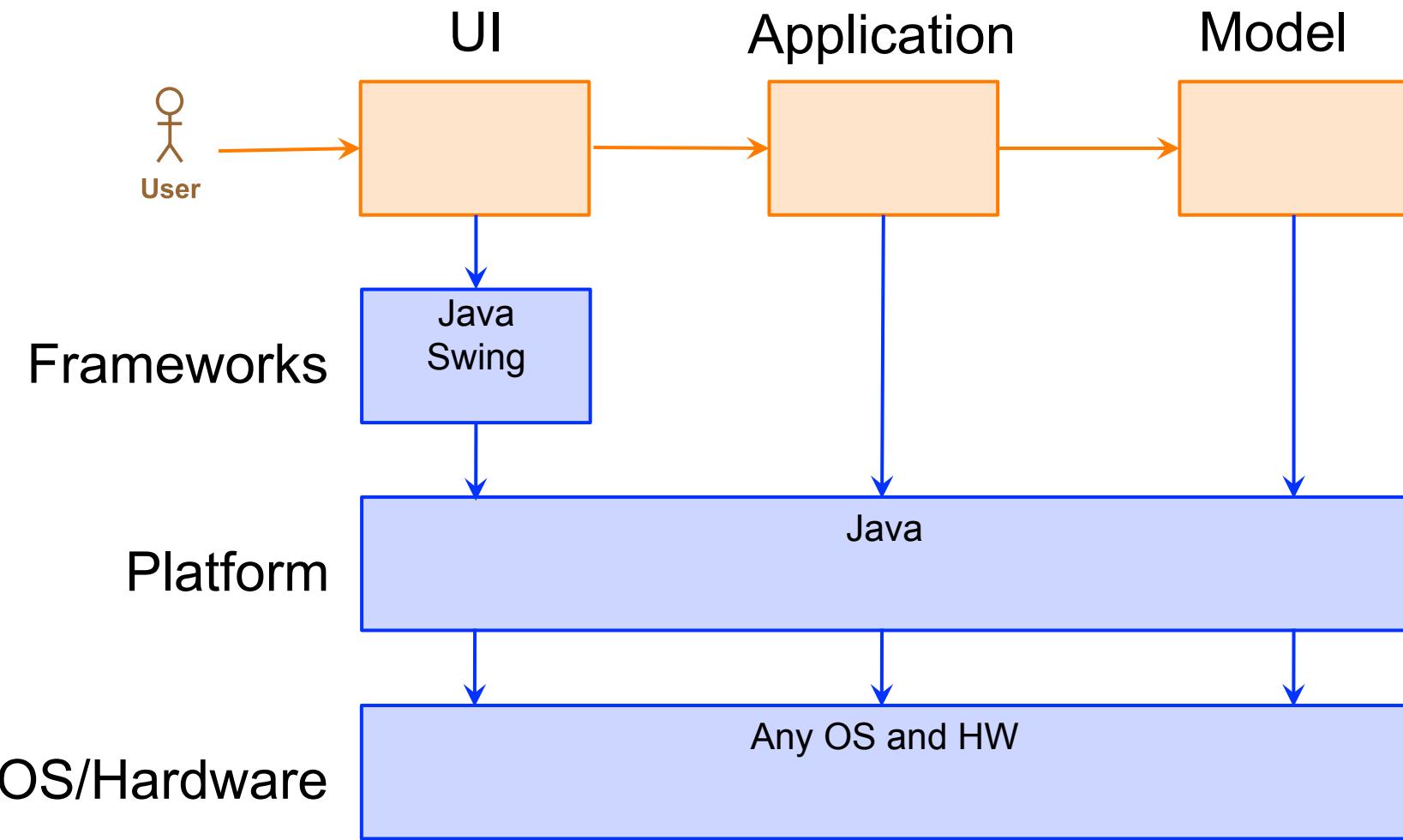
- The User interacts with the User Interface (UI) tier.
- The UI tier interacts with the Application and Model tiers.
- The Application tier holds logic that controls the flow of the application.
- The Model tier holds the core domain (aka "business") logic.



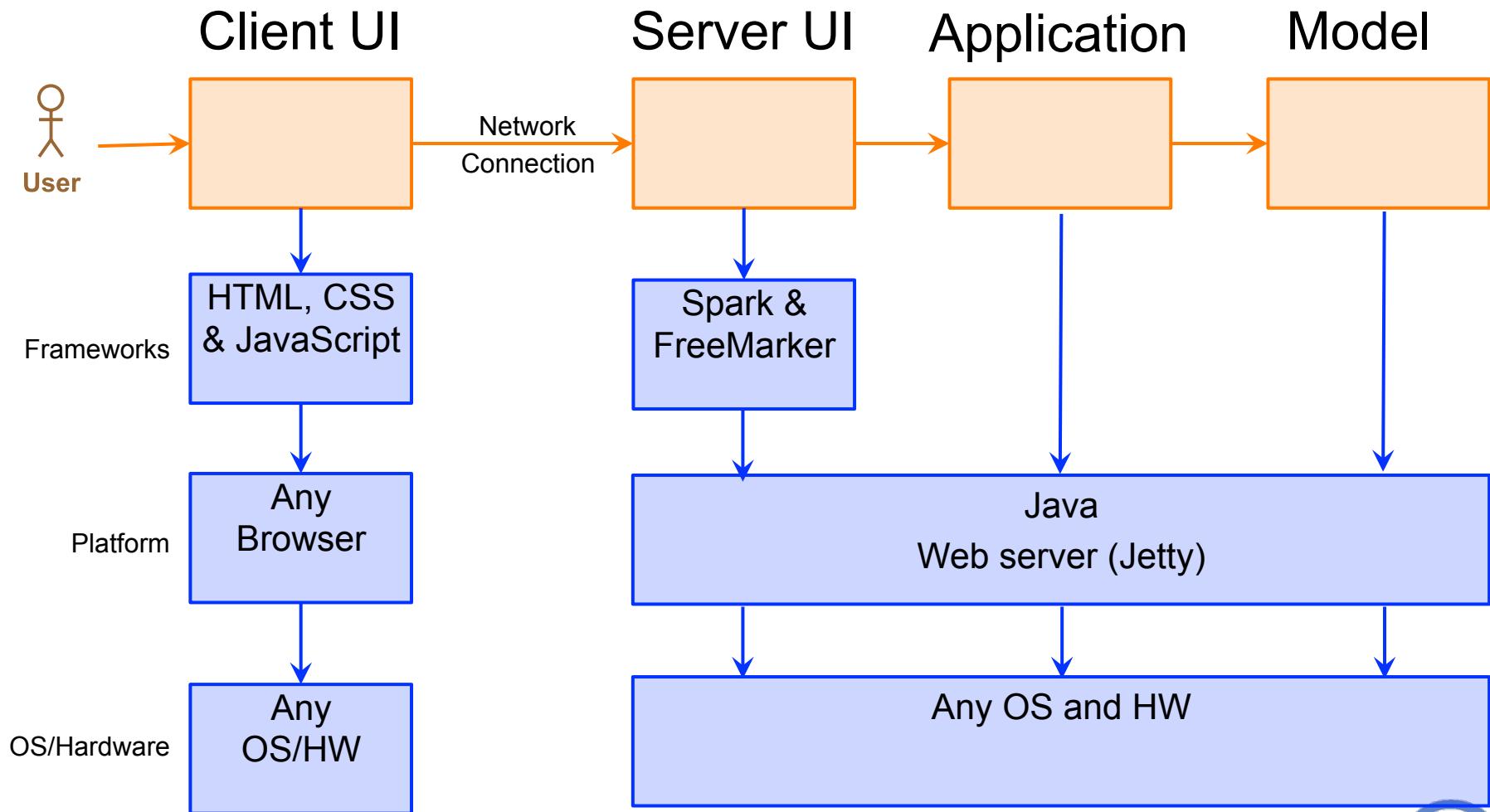
# Architecture must also consider layers



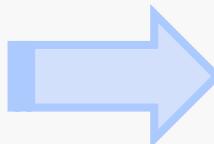
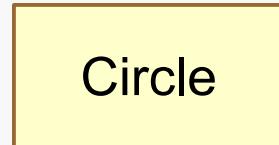
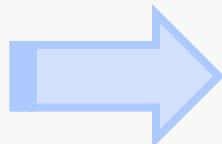
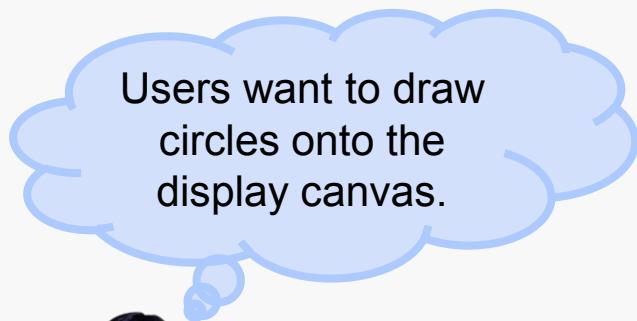
# Let's start with a simple, desktop architecture



# This is the architecture you will use in the web-based term project.



# Reviewing OO Concepts



```
public class Circle {  
    // more code here  
}
```

**SWEN-261**  
**Introduction to Software**  
**Engineering**

**Department of Software Engineering**  
**Rochester Institute of Technology**

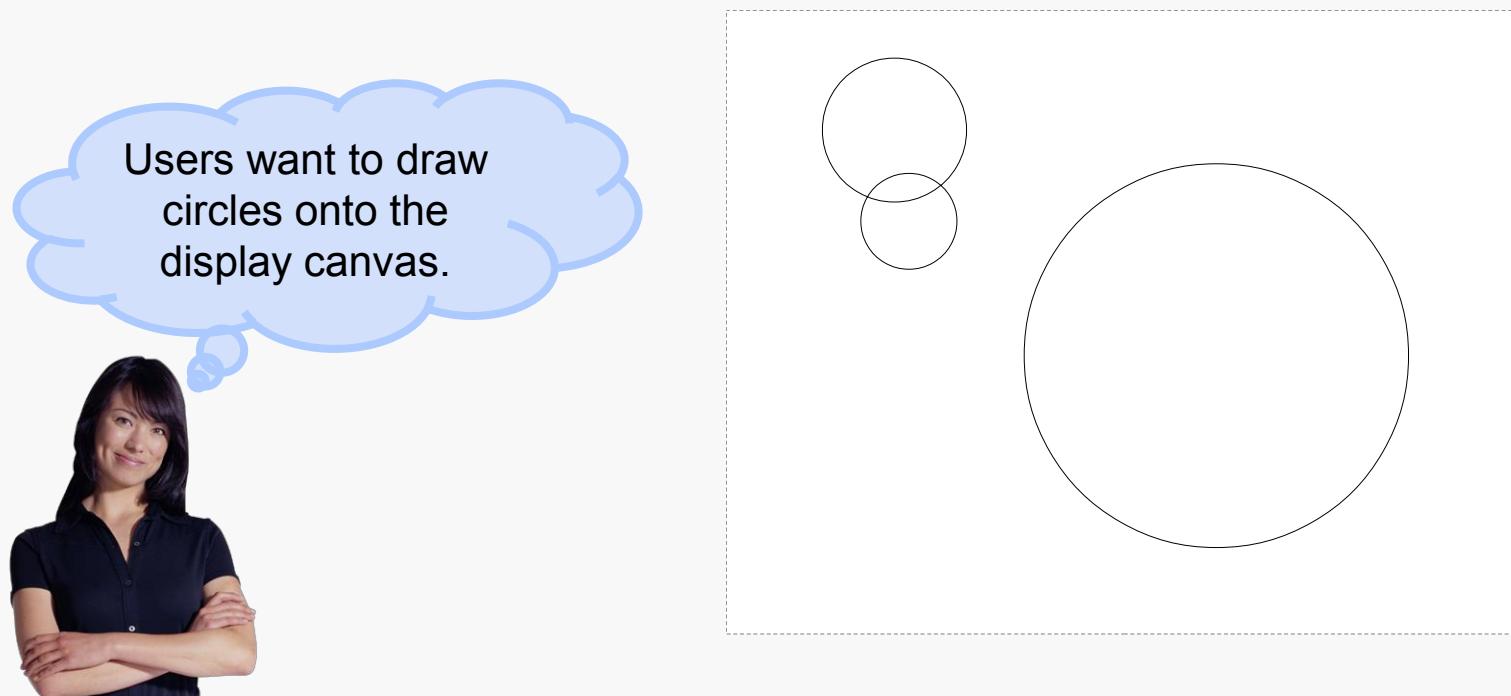
# OO Programming is about visualizing a class, modeling the class and then coding the class.

- Programming is and will always be a mental activity.
- UML modeling gives shape to your mental model.
  - *To make your mental model more concrete*
  - *To validate your mental model with stakeholders*
  - *To share with other developers*
- The UML model acts as a guide during development.



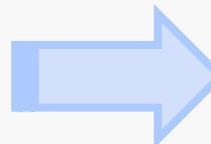
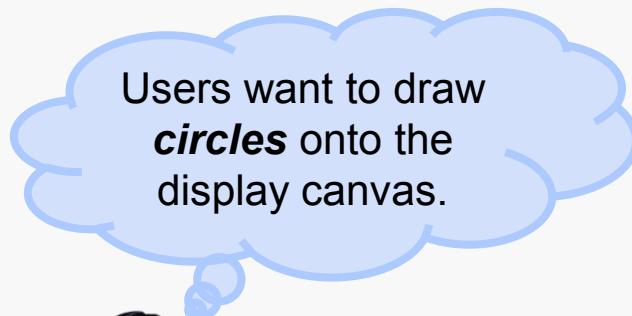
# We'll use a drawing application as our example application domain.

- Imagine a drawing application in which the user can place shapes on a canvas. Let's start with a circle.



# All OO programming starts with classes and objects.

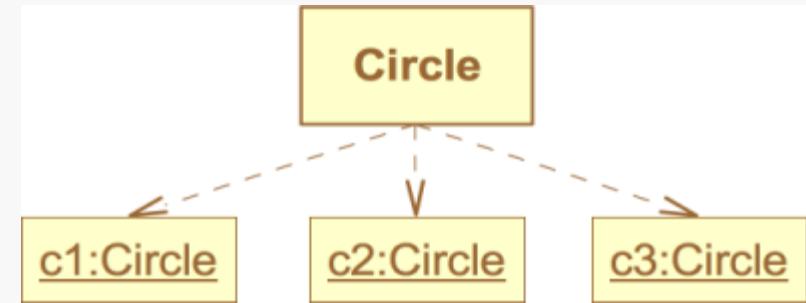
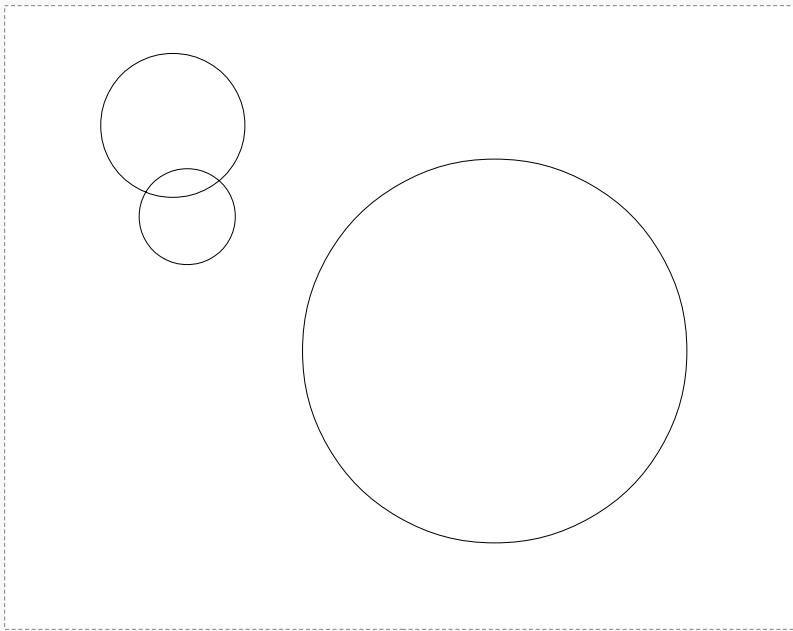
- A *class* is a template for run-time *objects*.
- Use UML class notation to model your mental model of a circle.
- Java classes implement these models.



```
public class Circle {  
    // more code here  
}
```



# One class may have many unique objects.

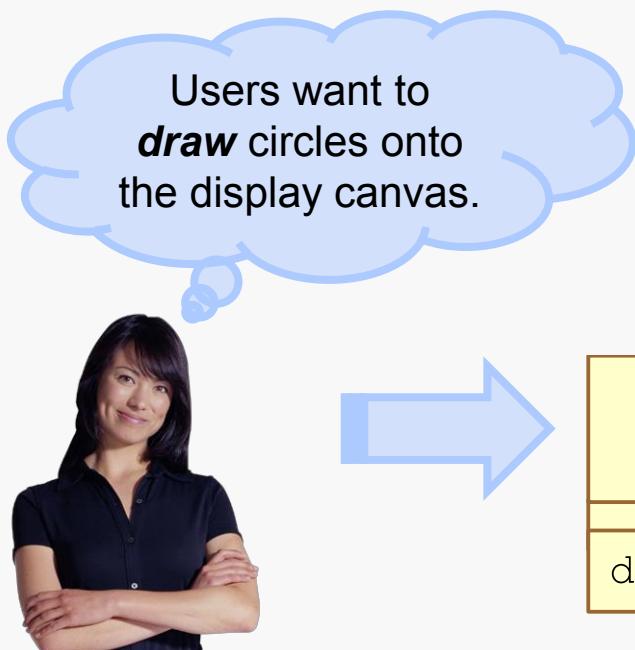


```
public void make_multiple_objects() {  
    Circle c1 = new Circle();  
    Circle c2 = new Circle();  
    Circle c3 = new Circle();  
    if (c1 != c2) {  
        // Two distinct objects have different identities.  
    }  
}
```



# Objects perform behaviors defined by their class.

- Look to the verbs to identify behaviors.



```
public class Circle {  
    void draw() {  
        // TBD  
    }  
}
```

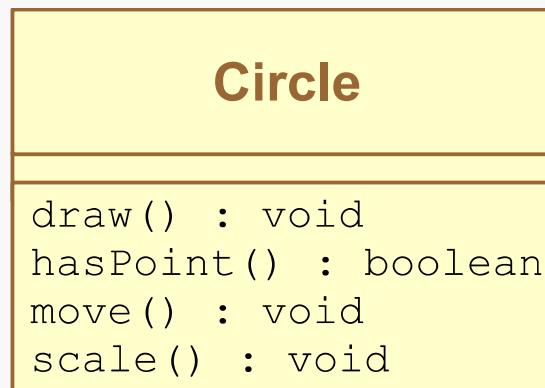
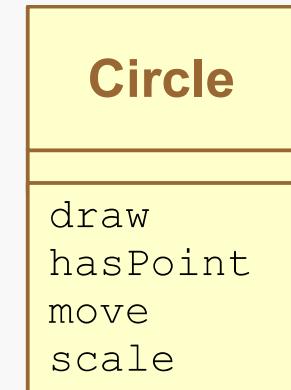
# OO design is all about assigning responsibilities to classes

- In a drawing app the user will need to:
  - *Select a shape by clicking on it.*
  - *Move a shape by dragging it to a new position.*
  - *Scale the shape by dragging a corner.*
- Of course the set of behaviors is totally dependent upon the domain of the specific application. For example a CAD app also provides:
  - *Show measurements (perimeter and area) of a shape*
  - *Align shapes to a grid*
  - *Calculate shape unions and intersections and exclusions*
- We'll talk about design later but for now let's focus on OO concepts and UML.



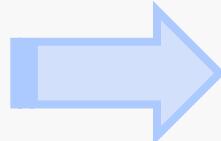
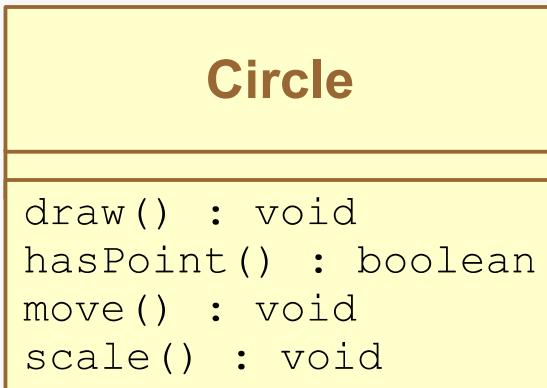
# So the design for our drawing app will be...

- As an artist I also need to:
  - *Select a shape to interact with.*
  - *Move a shape by dragging it to a new position.*
  - *Scale the shape by dragging a corner.*
- In some cases we know return values of methods, shown here:



# This design starting point can even form a sketch of a Java class.

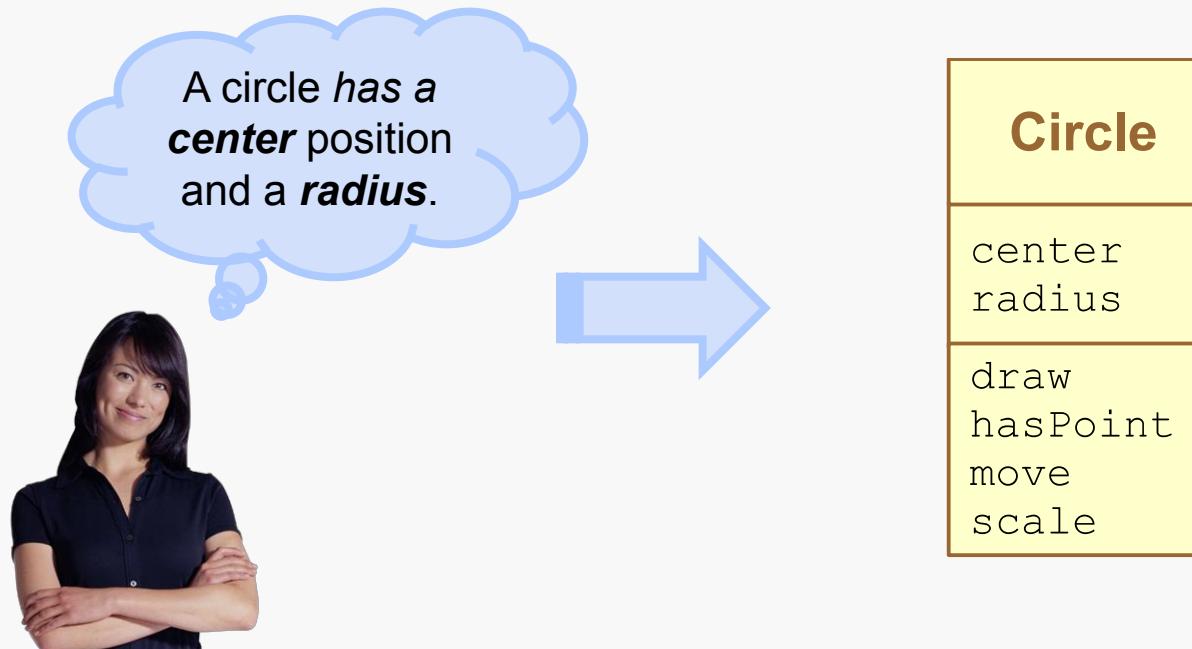
- We still don't have some details, such as parameters to these methods.
- But we can sketch out a skeleton class:



```
public class Circle {  
    void draw() { /* TBD */ }  
    boolean hasPoint() { /* TBD */ }  
    void move() { /* TBD */ }  
    void scale() { /* TBD */ }  
}
```

# In order to do the work of behaviors, objects will use attributes defined by the class.

- Include the known attributes of an object into the class definition.

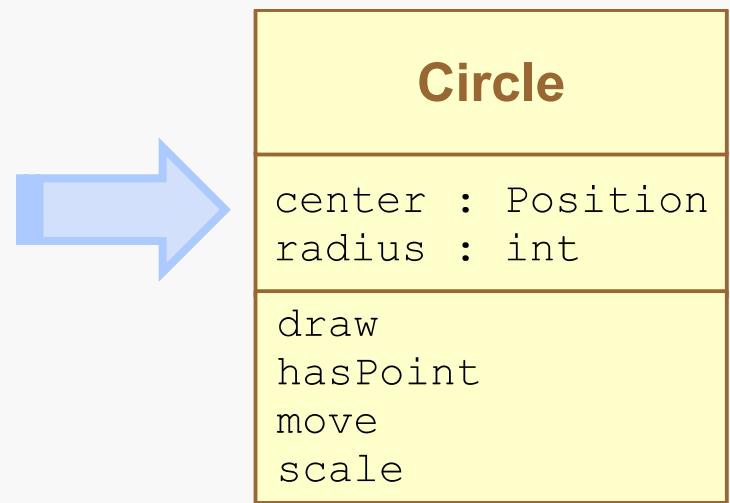


# Attributes have data types.

- Identify the data types for each attribute.
  - Might be "primitives" like int and String*
  - Or it might be other domain types, like Position*



Well, I did say the center is a **position**. The radius must be a number.

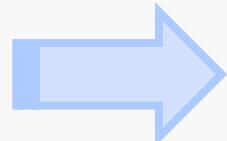
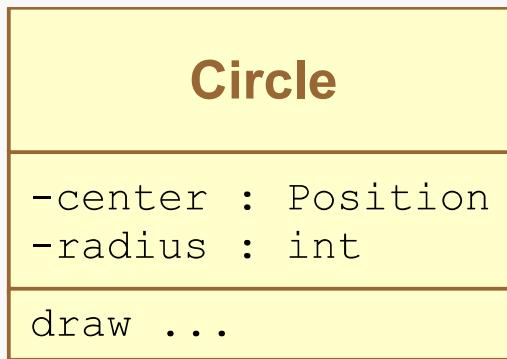


```
public class Circle {  
    Position center;  
    int radius;  
    // more code here  
}
```



# But make sure that you hide your attributes within the class.

- The way to hide a class's attributes is to make them private.



```
public class Circle {  
    private Position center;  
    private int radius;  
    // more code here  
}
```

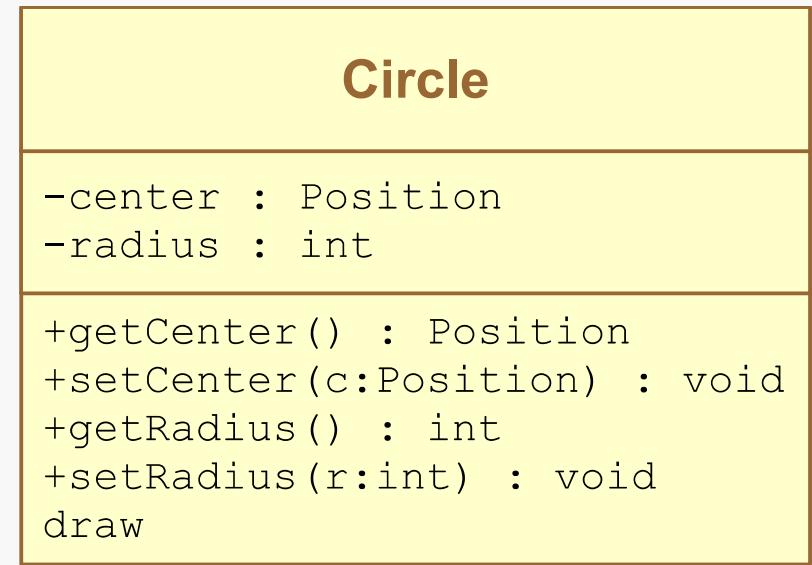
- And then provide methods to inspect or mutate the object.
  - Only expose data if necessary; provide methods to do what the client needs rather than expose data*
  - Only provide mutator methods as required, and use the domain language*



# Getters and Setters are not benign.

- *Don't do this:*

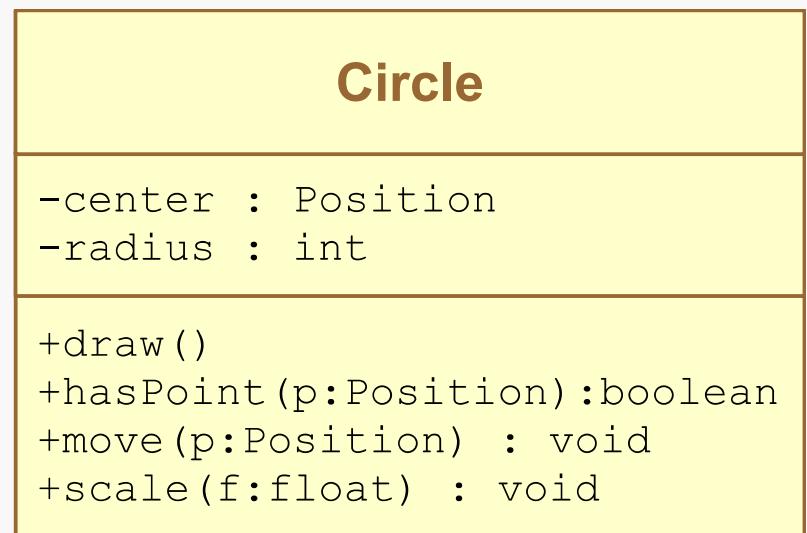
```
public class Circle {  
    private Position center;  
    private int radius;  
  
    public Position getCenter() {  
        return center;  
    }  
    public void setCenter(Position c) {  
        this.center = c;  
    }  
  
    public int getRadius() {  
        return radius;  
    }  
    public void setRadius(int r) {  
        this.radius = r;  
    }  
}
```



# Design methods that are semantically interesting.

- Don't set the center, rather the circle *moves*.

```
public class Circle {  
  
    private Position center;  
    private int radius;  
  
    public void draw() { /* TBD */ }  
  
    public void move(Position p) {  
        this.center = p;  
    }  
  
    public void scale(float factor) {  
        this.radius = (int) (radius * factor);  
    }  
  
    public boolean hasPoint(Position p) {  
        return p.distanceTo(center) <= radius;  
    }  
}
```



# You should also hide your class's data structures such as lists, sets, maps and other collections.

- Don't create getters/setters to the collection:

```
public class DisplayCanvas {  
    private Set<Circle> circles = new HashSet<>();  
    public Set<Circle> getircles() {  
        return circles;  
    }  
    public void setircles(Set<Circle> circles) {  
        this.circles = circles;  
    }  
}
```

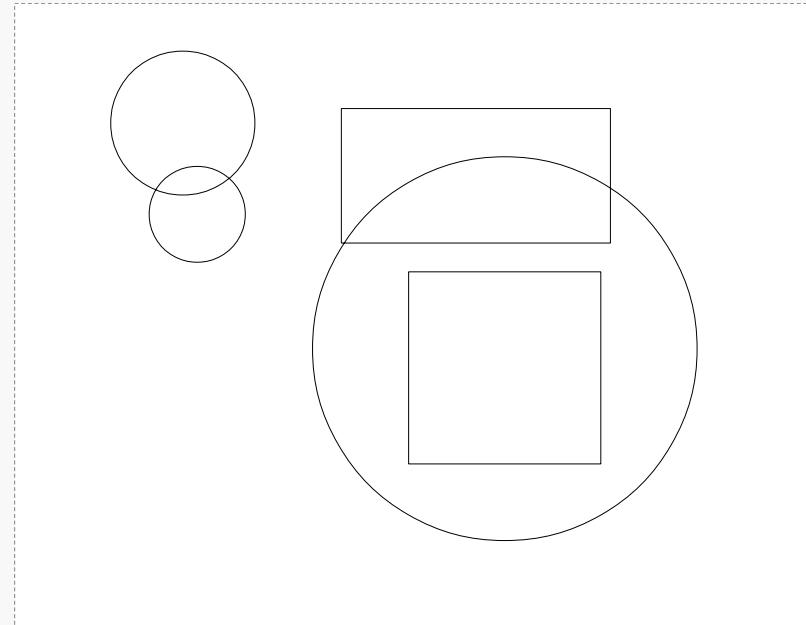
- Protect your data structures:

```
public class DisplayCanvas {  
    private Set<Circle> circles = new HashSet<>();  
    public Iterable<Circle> getircles() {  
        return circles;  
    }  
    public void addCircle(Circle circle) {  
        this.circles.add(circle);  
    }  
}
```

# OK, let's go back to our developer. She now needs to design a Rectangle class.



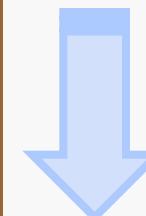
Users want to draw rectangles onto the display canvas. And select, move and scale them.



## Rectangle

```
-topLeftCorner : Position  
-width : int  
-height : int  
  
+move (p:Position) : void  
+scale(f:float) : void  
+draw()  
+hasPoint(p:Position) : boolean
```

Code on the next page.



# The Rectangle implementation looks like this.

```
public class Rectangle {  
    private Position topLeftCorner;  
    private int width;  
    private int height;  
  
    public Rectangle(  
        final Position topLeftCorner,  
        final int width,  
        final int height) {  
        this.topLeftCorner = topLeftCorner;  
        this.width = width;  
        this.height = height;  
    }  
  
    public void move(Position toPosition) {  
        this.topLeftCorner = toPosition;  
    }  
  
    public void scale(float factor) {  
        width = (int) factor * width;  
        height = (int) factor * height;  
    }  
  
    public void draw() {  
        /* TBD */  
    }  
  
    public boolean hasPoint(Position p) {  
        /* TBD */  
    }  
}
```

Do you notice any duplication with Circle?



# There's a principle in software development: *Don't repeat yourself.*

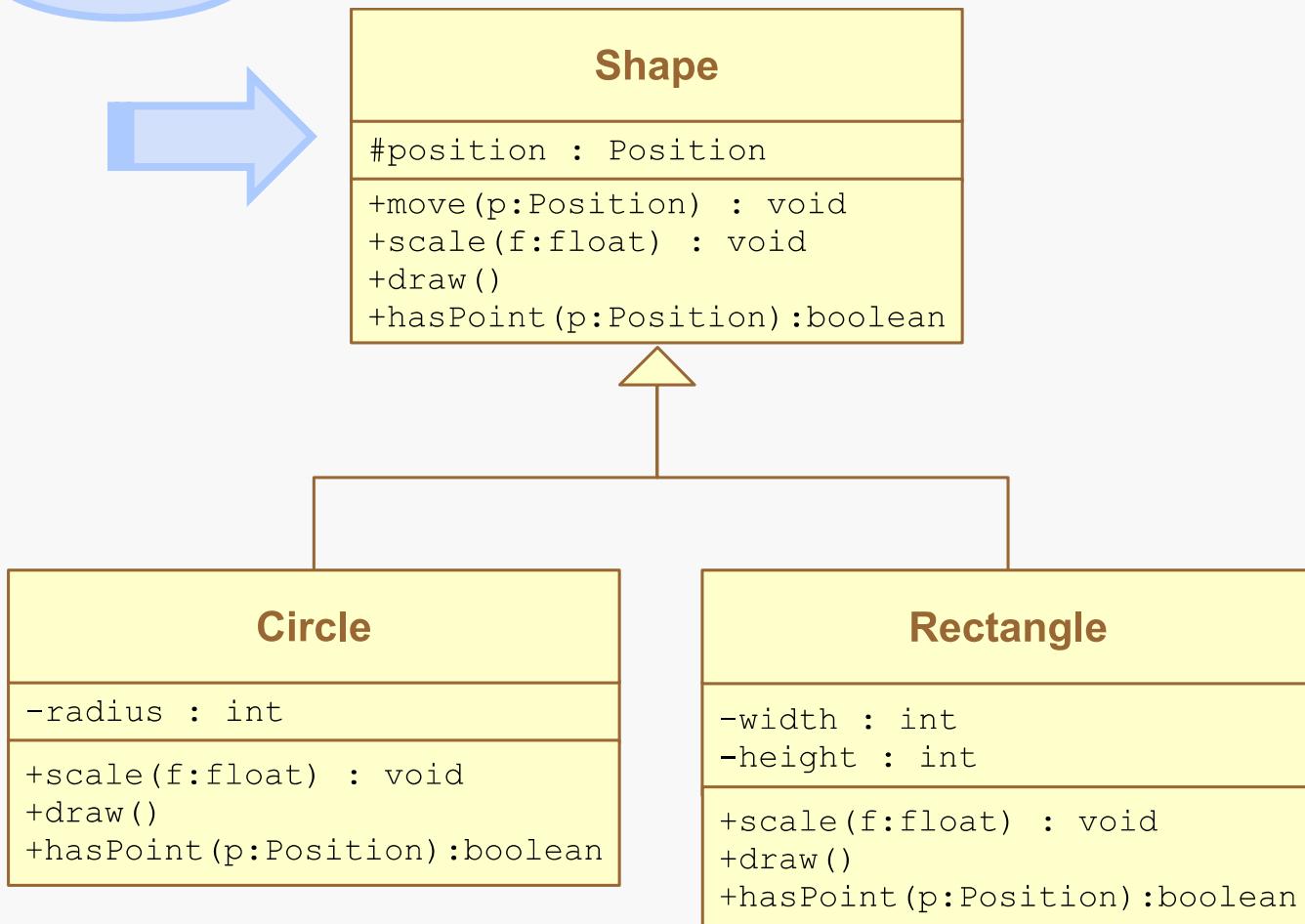
- Both Circle and Rectangle have a position.
- They have identical move methods and other methods with identical signatures.
- What should you do to not repeat yourself?



# Pull shared attributes and behaviors into a super class.



The drawing app now deals with two kinds of shapes:  
circles and rectangles.



# Here's the code for the Shape super class.

```
public class Shape {  
  
    protected Position position;  
  
    public Shape(final Position position) {  
        this.position = position;  
    }  
  
    public void move(Position position) {  
        this.position = position;  
    }  
  
    public void draw() {  
        /* TBD */  
    }  
  
    // more code not shown  
}
```

# Here's the code for the Circle subclass.

```
public class Circle extends Shape {  
    private int radius;
```

Use the `extends` keyword to allow the `Circle` class to inherit the attributes and methods of the super class: `Shape`.

```
public Circle(final Position center, final int radius) {  
    super(center);  
    this.radius = radius;  
}
```

Use the `super` keyword to invoke the `Shape` constructor.

```
public void draw() { /* TBD */ }
```

```
public void scale(float factor) {  
    this.radius = (int) (radius * factor);  
}
```

```
public boolean hasPoint(Position p) {  
    return p.distanceTo(position) <= radius;  
}
```

You can use protected members of the `Shape` class.



# Should the super class be abstract?

- Specifically for the drawing app, can you add a "shape" (ie, a generic shape) to the canvas?
  - *If so, then the current implementation is fine.*
  - *If not, then restrict the ability to instantiate the Shape class.*



# Use **italics** on labels for abstract "things".

## *Shape*

```
#position : Position  
+move(Position):void  
+scale  
+draw  
+hasPoint
```

```
public abstract class Shape {  
    protected Position position;  
  
    protected Shape(final Position position) {  
        this.position = position;  
    }  
  
    public void move(Position position) {  
        this.position = position;  
    }  
    public abstract void draw();  
    // more code not shown  
}
```

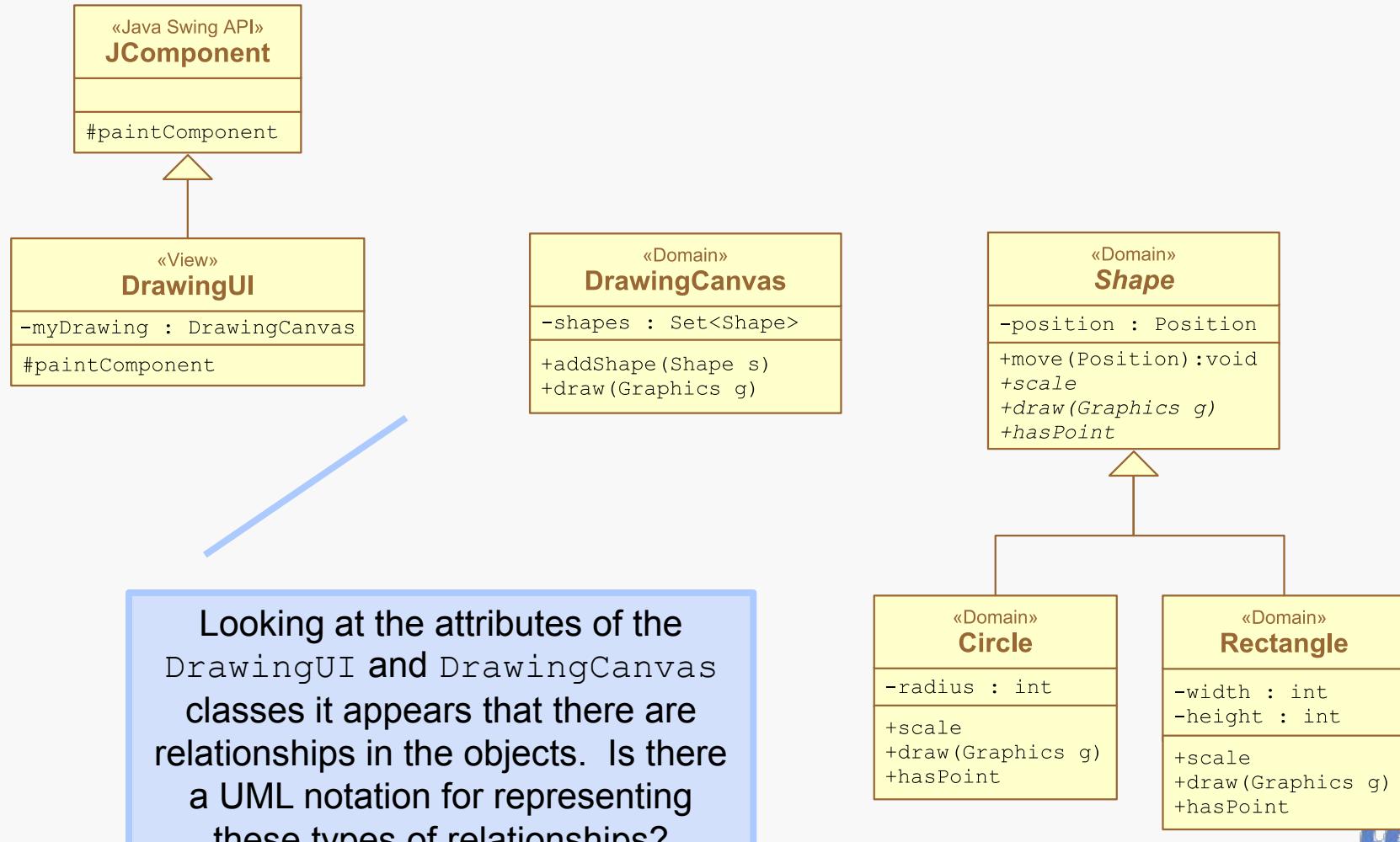
Make the class abstract.

Make all constructors protected.

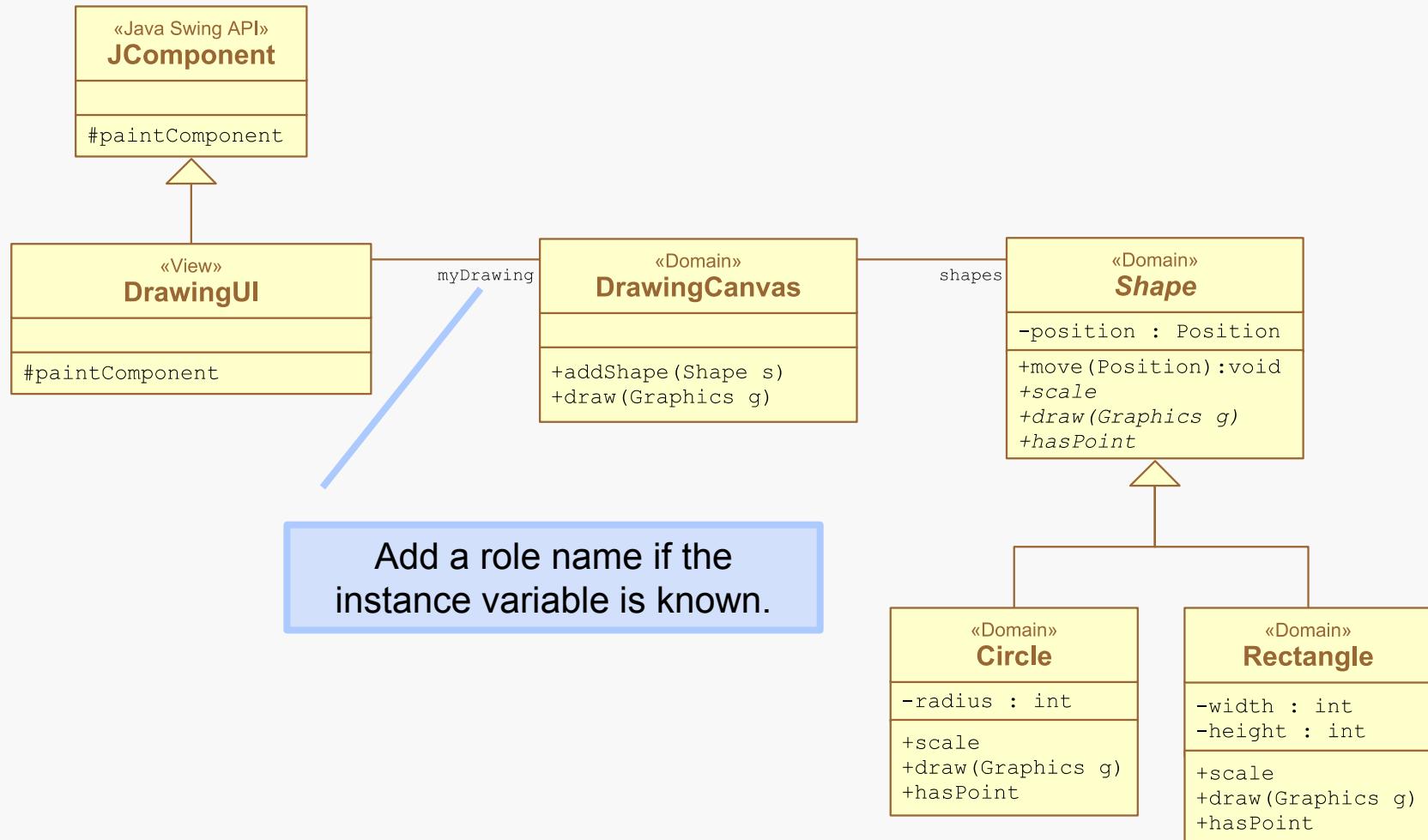
Make some methods abstract.



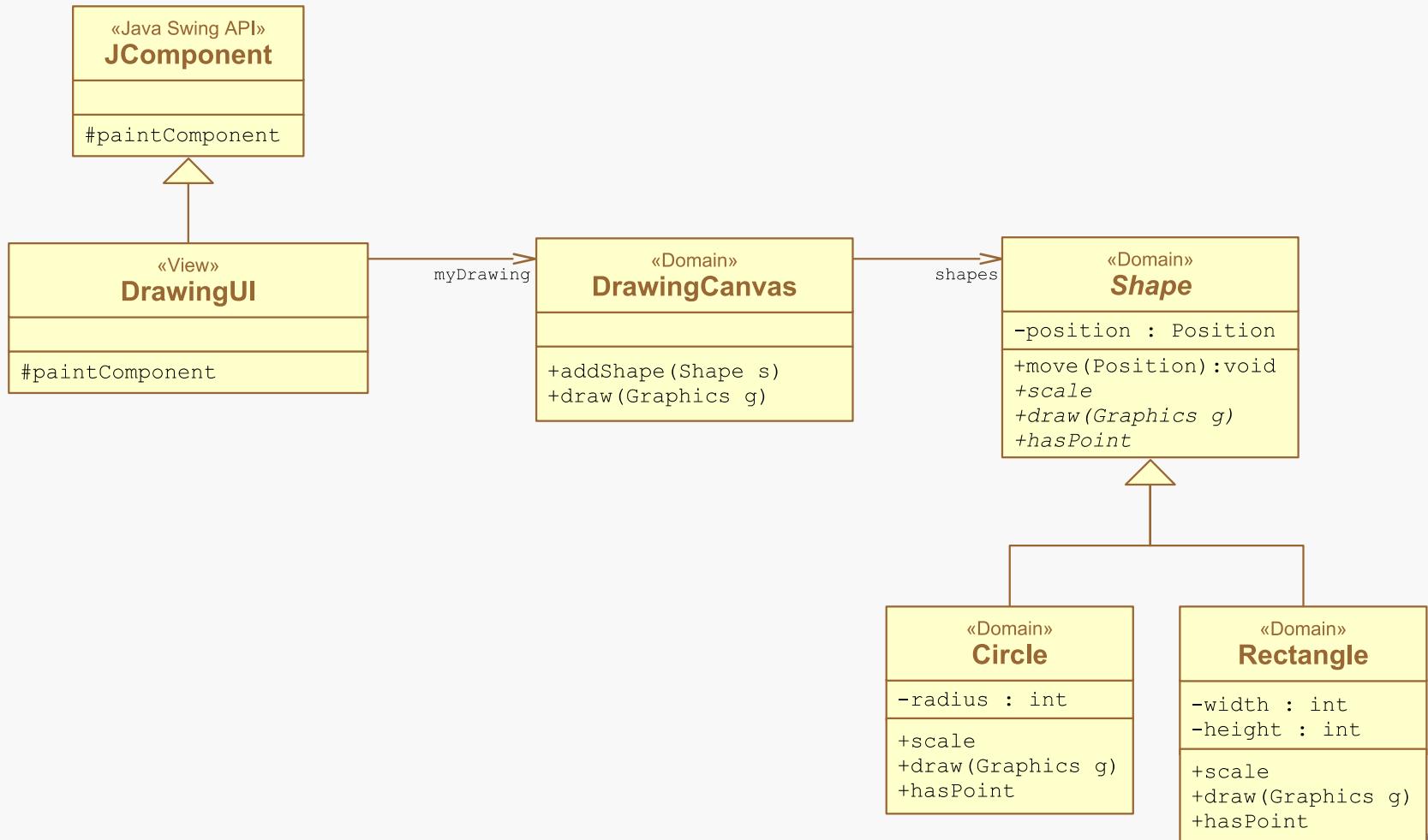
# Our developer has been busy and has created the following Java/Swing application architecture.



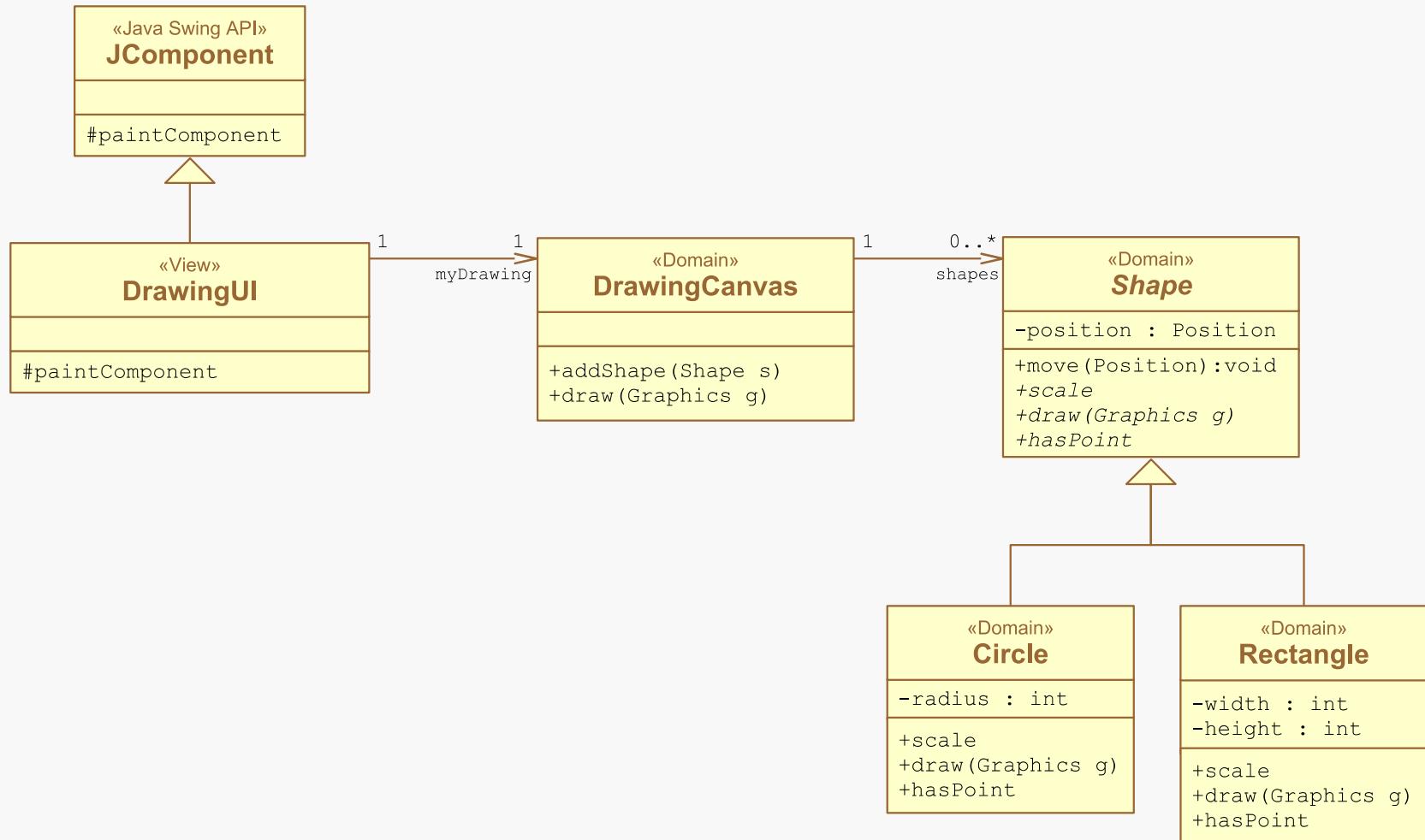
# UML uses a line to connect classes that have associations.



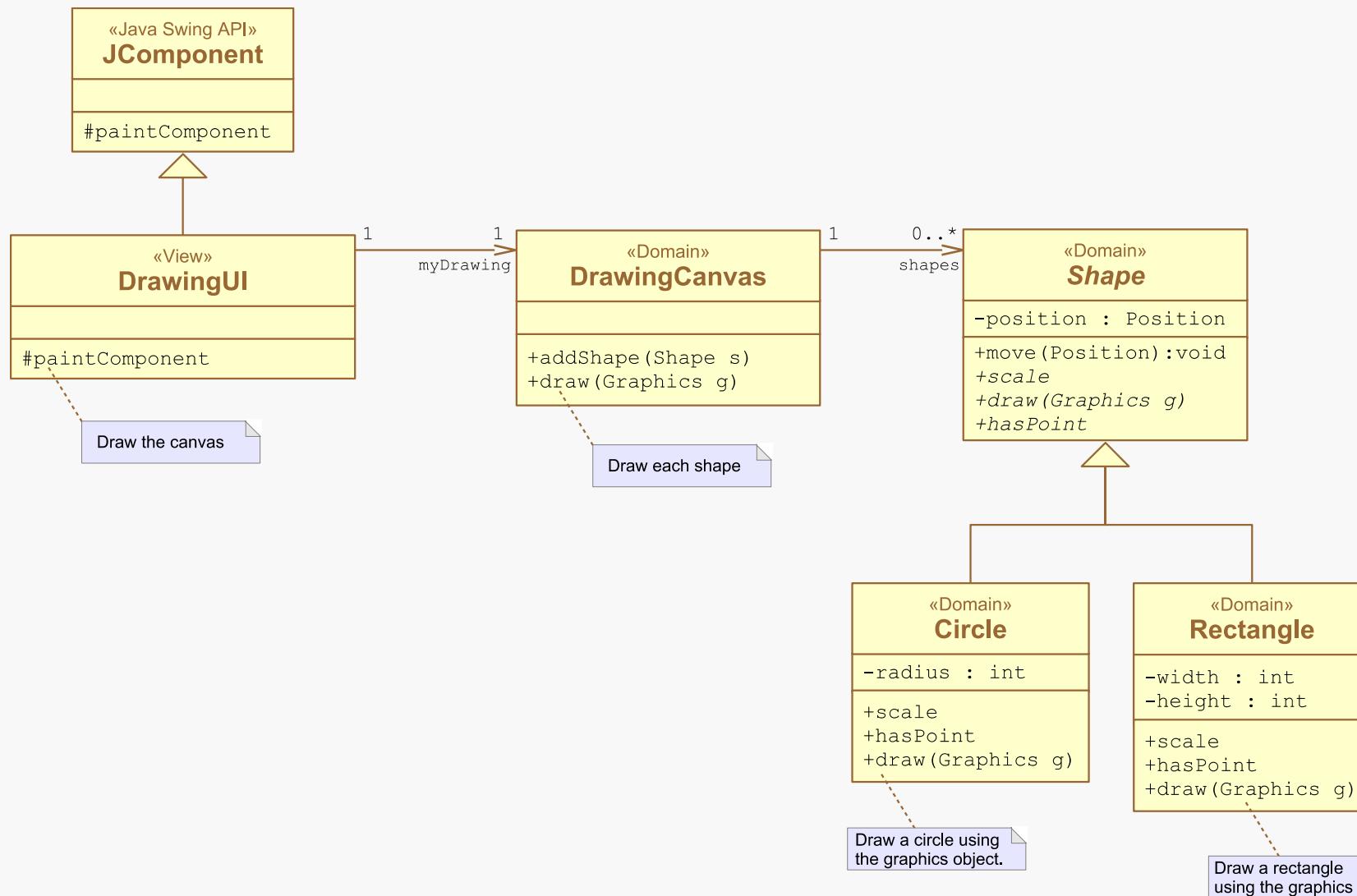
# We can specify directionality of the associations.



# We can also specify multiplicity of the associations.



# Now let's consider how this is coded...



# The UI code that delegates drawing to the canvas object passes in the Graphics context.

```
public class DrawingUI extends JPanel {  
    private final DrawingCanvas myDrawing;  
  
    // more code here  
  
    @Override  
    protected void paintComponent(Graphics g) {  
        super.paintComponent(g);  
        // Draw the canvas  
        myDrawing.draw(g);  
    }  
}
```

# The DrawingCanvas class draws a set of shapes.

```
public class DrawingCanvas {  
    private Set<Shape> shapes = new HashSet<>();  
  
    public void addShape(final Shape s) {  
        shapes.add(s);  
    }  
  
    public void draw(Graphics g) {  
        // Draw each shape  
        for (Shape s : shapes) {  
            s.draw(g);  
        }  
    }  
}
```



# And now each shape's specific draw methods.

- The Circle draw method:

```
public void draw(Graphics g) {  
    final int diameter = 2 * radius;  
    final Position pos = getPosition();  
    g.drawOval(pos.getX() - radius, pos().getY() - radius,  
               diameter, diameter);  
}
```

- The Rectangle draw method:

```
public void draw(Graphics g) {  
    final Position pos = getPosition();  
    g.drawRect(pos.getX(), pos.getY(), width, height);  
}
```



# Did you notice the polymorphism in the drawing code?

- Take a look again at the DrawingCanvas code:

```
public void draw(Graphics g) {  
    // Draw each shape  
    for (Shape s : shapes) {  
        s.draw(g);  
    }  
}
```

How does the compiler know which  
shape draw method is invoked?  
(Circle or Rectangle?)



# The lecture reviewed OO concepts and used defensive programming practices.

## OO Concepts Reviewed

- Object identity
- Encapsulation
- Information hiding
- Inheritance
- Abstraction
- Associations
- Polymorphism

## Defensive programming

- Private/protected attributes and methods
- Final attributes and parameters
- Minimized use of getters and setters
- Hide internal data structures

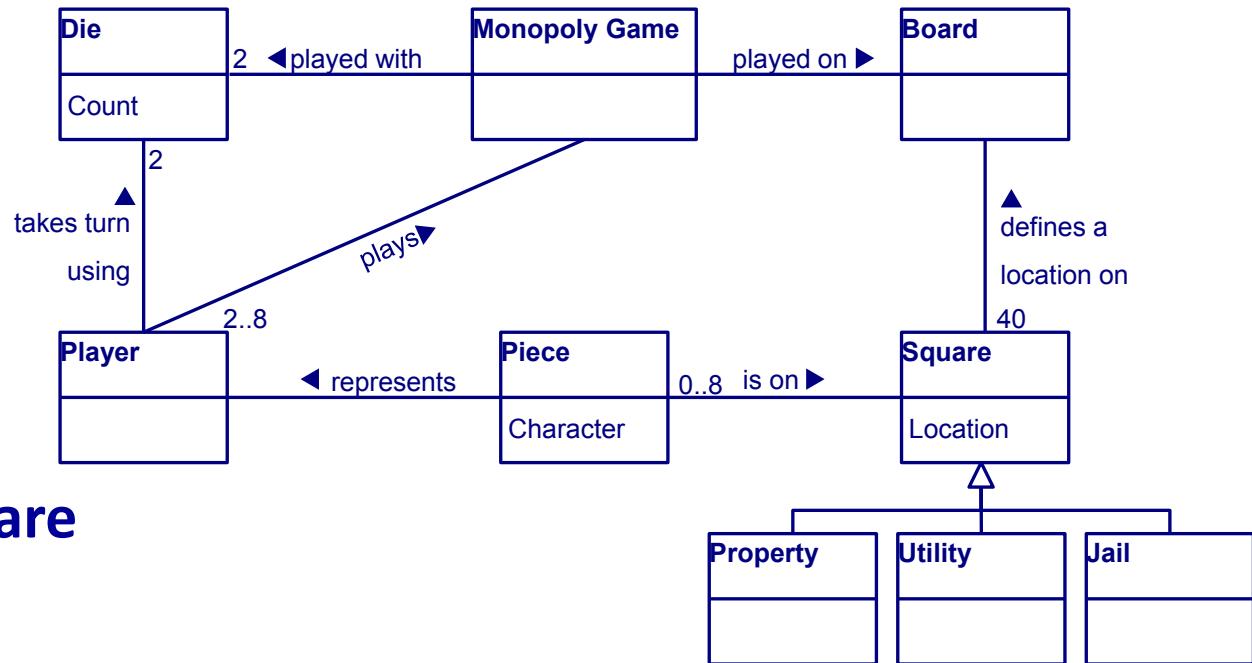


# Domain Analysis

SWEN-261

Introduction to Software  
Engineering

Department of Software Engineering  
Rochester Institute of Technology

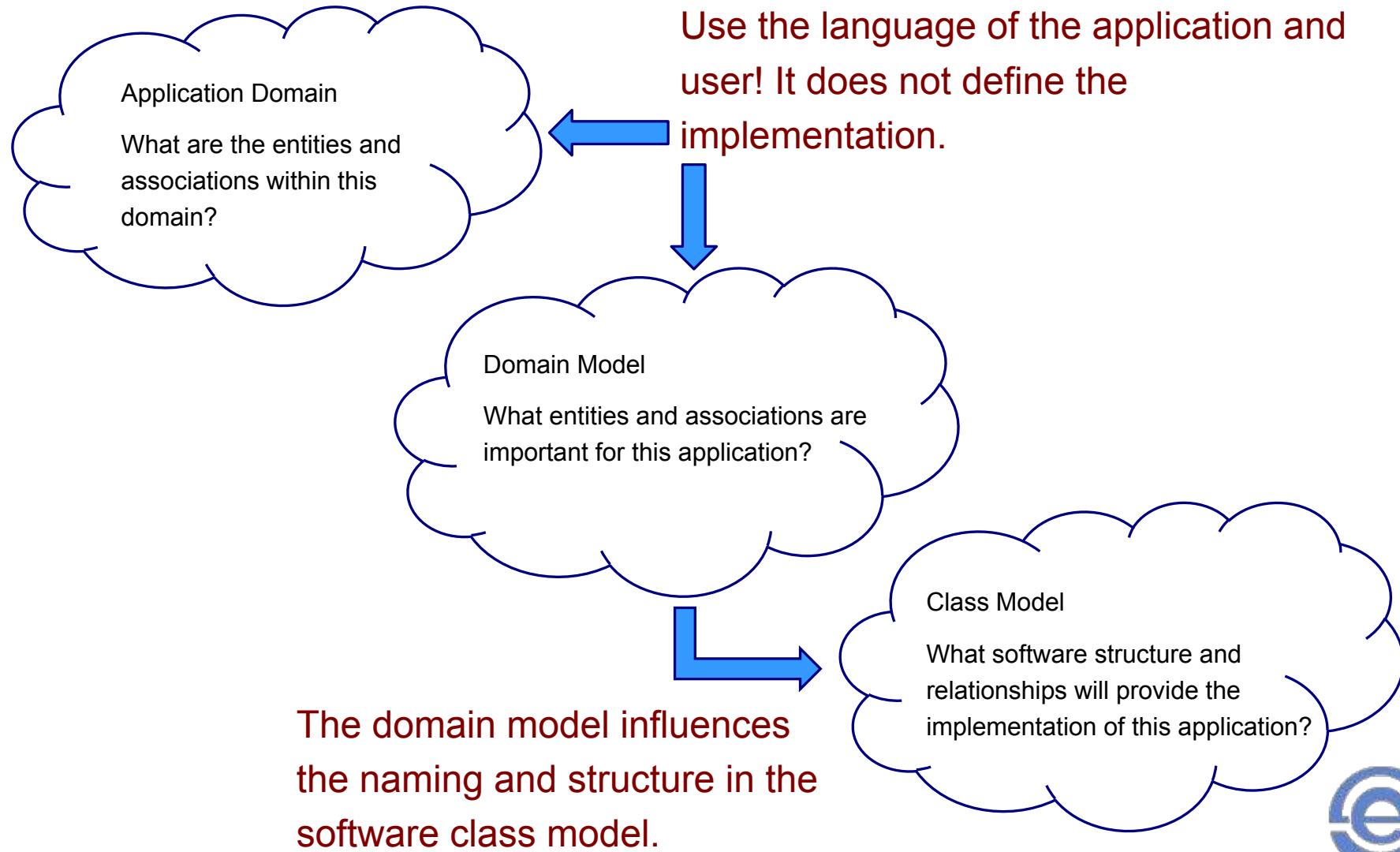


# The domain for a software system defines the context in which the software operates.

- This is also referred to as the *application domain*.
  - *Retail sales*
  - *Banking*
  - *Customer contact management*
  - *Checkers playing*
- The domain model describes the ubiquitous world in which the system's experts and users exist and work on a daily basis.
  - *Domain entities*
  - *Domain language*
  - *Associations/relationships between entities*



# Domain analysis provides an understanding of the application problem space.



# The domain model identifies important aspects of the application not the implementation.

- Only use vocabulary from the problem statement
  - *For example, a unique identifier needed to store data with no meaning to the user would not be in a domain model.*
- Establishes a common understanding of the problem for customer/user and software team



# Domain model definition starts with an analysis of the nouns in the domain.

- The steps in the noun analysis include
  - *Identify the nouns in the problem statement and language of the domain experts and users.*
  - *Identify any words that might be specializations of other nouns.*
  - *Identify any nouns that might be attributes or properties of other nouns.*
  - *Identify any other associations between nouns.*



# The domain model is typically drawn using a simplified class diagram notation.

- Show the following information
  - *Domain entities*
  - *Attributes in domain entities*
  - *Associations between domain entities*
- Use user vocabulary
  - *Attributes do not indicate data type*
- Associations come from the problem statement
  - *Place label on the association line*
  - *Usually completes a phrase between two domain entities: DE<sub>1</sub>, association DE<sub>2</sub>, (LineItem records-sale-of Product)*
  - *Indicate multiplicity, if known*
  - *Use inheritance, if appropriate*

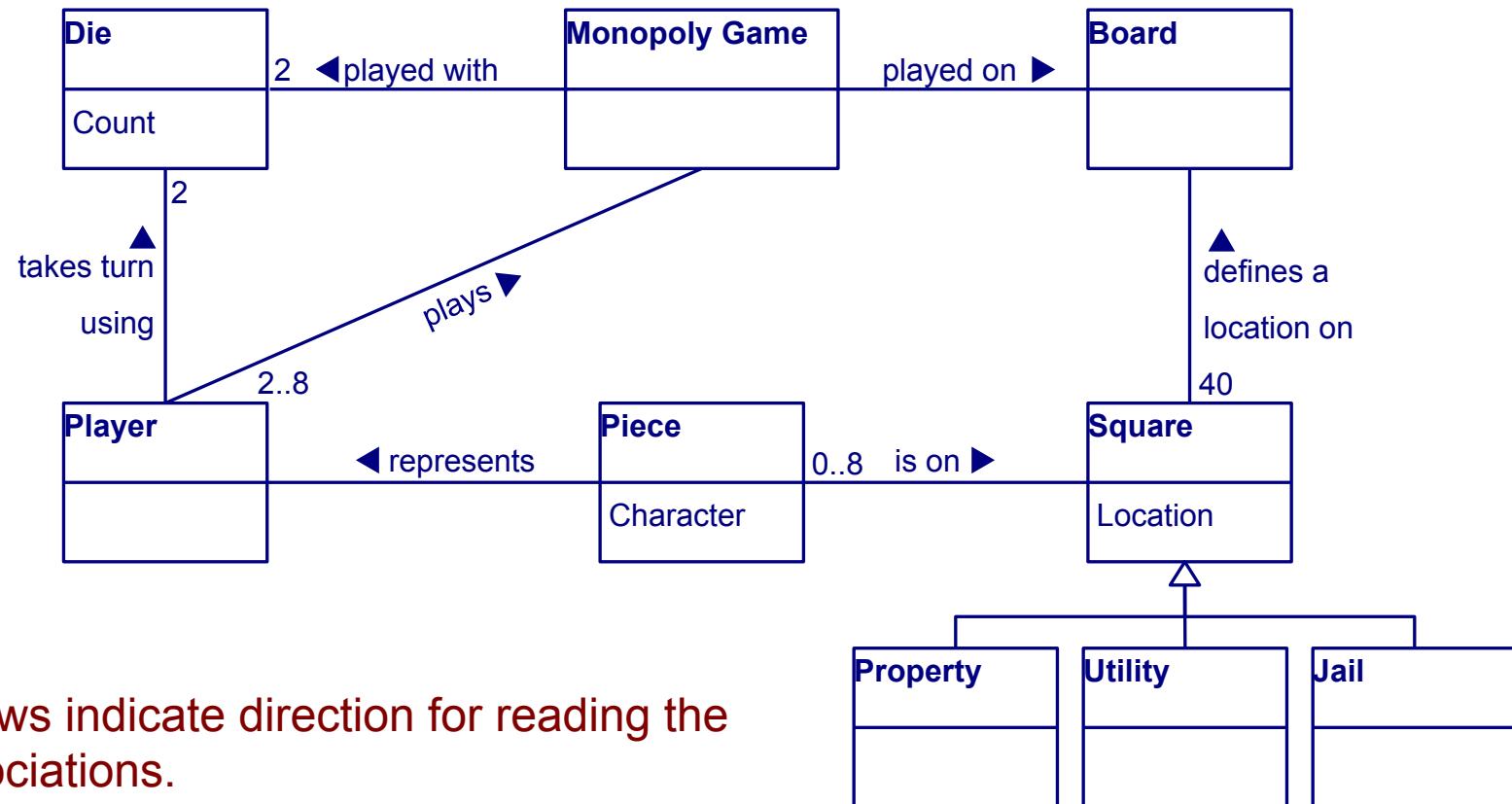


# An association should describe the relationship between two domain entities.

- All associations should have an arrow to indicate the direction to read the association.
- Use the active voice for the verb when possible.
- An association of "has" or "contains" does not describe much about the relationship.
  - ***Reverse the direction and rephrase the association***



# This partial domain model for a game of Monopoly demonstrates these ideas.



# Domain analysis continues through the project.

- The domain model continues to evolve as you learn more about the project.
  - *Working on the project gives you a different understanding of the domain.*
  - *New features change your understanding of the domain.*
  - *When user stories are refined during backlog refinement more details may come out about the domain.*
- Keep your domain model up-to-date so that there is always a common understanding between the development team and Product Owners.



# Team Formation

**SWEN-261**  
**Introduction to Software**  
**Engineering**

Department of Software Engineering  
Rochester Institute of Technology



Software Engineering  
Rochester Institute  
of Technology

# A group of people assigned to work together do not instantly turn into an effective team.

- Tuckman defined several stages that teams go through
  - *Forming – initial team formation; team members behave in formal and reserved manner*
  - *Storming – team members position themselves against one another, often with confrontation*
  - *Norming – confrontation may continue, but the team tackles project issues*
  - *Performing – an effective and productive team is working together; trust between members is high*
  - *Adjourning – final teamwork prior to team disbanding*



# For your team to be effective, you should strive to support these characteristics.

- The bedrock of an effective team is **trust** in individual members, and individual members trusting the team.
- The team **manages conflicts** that occur and does not avoid or bury them.
- Team members have a **full commitment** to the team.
- Team members **feel accountable** to the team and the team holds individual members accountable.
- With the previous characteristics present, the team can focus on **delivering results** to the customer.



# If conflict arises in your team, try some of these conflict resolution strategies.

- Conflict management styles span an Assertiveness to Cooperativeness range
  - *Assertiveness is the degree to which you try to meet your own needs.*
  - *Cooperativeness is the degree to which you try to help others or the team meet their needs.*
- Collaborative – often finds a new win-win solution
- Competitive – you better be right
- Accommodating – you lose to achieve a higher goal
- Avoiding – a solution to gain time or for low stakes
- Compromising – mutually acceptable solution



# Your project grade is determined by the team's results adjusted by your individual contributions.

- Your team submits work for each of five sprints with a grade assigned for the team's results.
- Your individual contributions can modify that team grade either positively or negatively.
  - *There is an audit trail of accountability in all of the team's activities.*
    - ◆ Planning (Trello board)
    - ◆ Coding and documentation (GitHub repository)
    - ◆ Engagement with team (Team Slack)
  - *You must have an equal presence in team activities.*
  - *Your instructor will look in those areas. If you have little presence, you made no contributions.*
  - *Peer evaluations will also be strongly considered.*



# Agile teams are self-directed.

- Self-directed teams manage their own activities which requires team cohesion.
- Teams tend to be egalitarian in assigning task responsibilities.
  - *There tend not to be fixed roles.*
  - *The team will use the varied and diverse skill set of its members to its best advantage.*
  - *The team is responsible for making sure that all tasks get covered.*
  - *Every team member is eager to pick up new tasks to help further the team's goals.*
  - *There is no individual ownership of artifacts particularly code.*



# Defining Project Requirements



## Register for a League

In list [Product Backlog](#)

[Description](#) [Edit](#)

### Story

As a Player I want to register for a league because I want to play soccer.

**SWEN-261**

**Introduction to Software  
Engineering**

**Department of Software Engineering  
Rochester Institute of Technology**



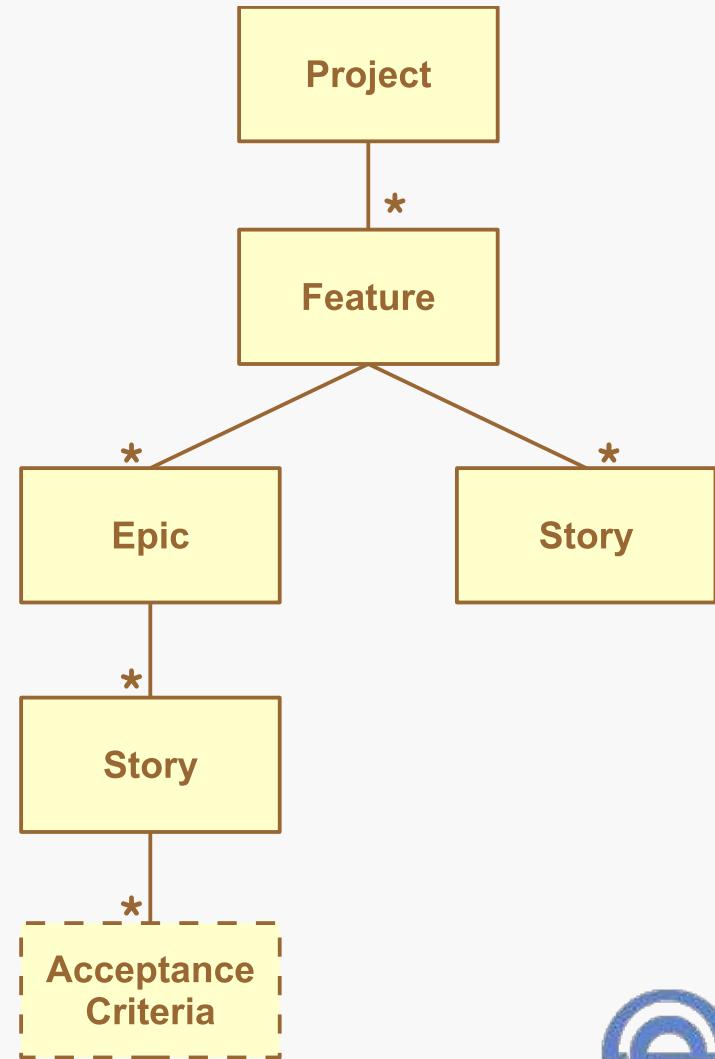
# There are functional and non-functional requirements.

- Functional requirements define the behavior of the application that achieve user goals.
- Non-functional requirements define:
  - *Systemic qualities such as response time, reliability and availability*
  - *Systemic constraints such as security or accessibility*
  - *Evolutionary qualities such as testability and extensibility*



# Functional requirements are in a hierarchy.

- Most applications have many large scale *features*.
- These are decomposed into *epics* and *user stories*.
- Epics are further decomposed into additional stories.
- As analysis progresses, stories will be further defined with *acceptance criteria*.



# Features are large scale application behaviors.

- Soccer league management application

*The League Director manages a soccer league three seasons a year. Players need to register for a league. The Director selects a group of captains to form teams and the Director moderates the drafting of teams by captains. The Director must also schedule the games of the league across the whole season up to 15 weeks. The Director must also be able to record game results and keep track of team rankings.*



# Defining requirements involves a continual discussion with the customer.

- Eliciting requirements through:
  - *Observation*
  - *Interviewing users and subject matter/domain experts*
  - *Brainstorming*
- There are many approaches for specifying requirements
  - *Agile methods write user stories*
- This documentation forms a contract between the Product Owner and the Development Team.



# User stories express goals of users.

- There are many ways to write a user story.
- In class, you will use this notation:  
**"As a <ROLE>, I want <GOAL> so that <BENEFIT>."**  
*Example:*  
*As a Player, I want to register for a league so that I can play soccer.*
- The **ROLE** identifies the type of user to achieve the goal.
- The **GOAL** is what the system will do for the user.
- The **BENEFIT** is the value that the system provides the user.



# More examples of user stories...

- From the Soccer League application:
  - *As a captain I want to enter the scores of my games so my team's rank can be determined.*
  - *As a player I want to see my teams rank so I can gloat to my competitors.*
  - *As the director I want to create a tournament schedule so teams know when and who to play.*
- From a Social Media application:
  - *As a member I want to check-in that I'm having dinner with friends so my mom sees me with my friends.*
  - *As an ad-bot I want to access user usage statistics so that I can create personalized ads.*



# You will want to avoid the characteristics of poorly written user stories.

- Stories with no clearly defined user:  
*"I want to identify soccer league captains."*  
Or worse just: *"identify soccer league captains"*
- Stories with no clear benefit:
  - *What benefit does selecting captains provide?*
- Stories should not:
  - *Over constrain the solution*
  - *Dictate user interface details*



# In Trello you create a new card for each User Story.

- Your team will use [Trello](#) to record your project's requirements.
- Every User Story will be captured in a Trello card, like this:

The image shows a single Trello card with the following details:

- Title:** Manually Enter Games
- Location:** in list Product Backlog
- Actions:** Description [Edit](#)
- Type:** Story
- Description:** *As a League Coordinator I want to manually enter games so I can create the league schedule.*



# Story text is usually high-level, therefore you define acceptance criteria to provide refinement.

- Acceptance criteria (AC) are detailed statements about how the system should behave.
  - ***Use this format:***  
*GIVEN some condition WHEN some action occurs  
THEN system does something.*
  - ***Example:***  
*Given that I have not yet signed in when I see the Home page then I must see a means to sign-in.*
- The AC should drive and constrain the design and development.
- The AC are completed by the story's tester.



# In Trello you add the Acceptance Criteria using a checklist.

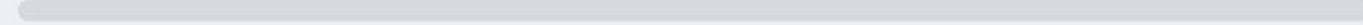
 **Player Sign-in**  
in list Sprint Backlog

Description [Edit](#)

## Story

*As a Player I want to sign-in so that I can play a game of checkers.*

 **Acceptance Criteria** [Delete...](#)

0% 

- Given that I have not yet signed in when I see the Home page then I must see a means to sign-in. (such as a link or button)*
- Given that I am not signed-in when I do click on the sign-in link then I expect to be taken to the Signin page, with a means to enter a player name.*
- Given that no one else is using my name when I enter my name in the sign-in form and click the **Sign-in** button then I expect system to reserve my name and navigate back to the Home page.*



# Sometimes a story is too large and it must be broken into smaller stories that compose an Epic.

- A story must be achievable within a single sprint.
  - *For your project try to create stories that one student can code in a single week.*
- Break up a larger story into smaller stories.
  - *Each sub-story must provide value but it can be a small step towards the epic goal.*
  - *Identify dependencies between sub-stories to produce a priority ordering of these stories.*
- An epic may be spread out over multiple sprints.
- The acceptance criteria of an epic is the successful completion of its sub-stories.



# In Trello you can represent an Epic with a Checklist of links to Story cards.

The screenshot shows a Trello board with the following structure:

- EPIC: Schedule Games** (Card Type: Epic)
  - in list Product Backlog
  - Description Edit
- Story** (Card Type: Story)
  - As a League Director I want to schedule the league games across the complete season so that I can run an efficient league.
  - Stories** (Checklist item)
    - Manually Enter Games
    - Validate Game Conflicts
    - Adjust the Schedule
    - <https://trello.com/c/F8ukujjS/20-schedule-games-automation>

A blue callout box points to the URL in the checklist item, containing the text: "Paste the Story card URL into the checklist item".



# Spikes help explore technology challenges.

- A Spike is:  
*A story or task aimed at answering a question or gathering information, rather than at producing shippable product.*  
(from ScrumAlliance.org)
- Spikes are used for:
  - *Learning new technologies or new techniques*
  - *Typically just proof-of-concept coding; usually throw-away*
- Spikes are usually time-boxed to fit into a single sprint though there are occasionally epic spikes.



# The Minimum Viable Product is all of the stories required to be in the first product release.

- MVP is defined by the Product Owner.
- It clearly identifies the set of stories that must be done before the product can be released.
- These stories must be prioritized to be worked on first.



# Effective Team Communications



**SWEN-261**  
**Introduction to Software**  
**Engineering**

Department of Software Engineering  
Rochester Institute of Technology



Software Engineering  
Rochester Institute  
of Technology

# There are multiple stakeholders who have an interest in a software development project.

- ‘an individual, group, or organization, who may affect, be affected by, or perceive itself to be affected by a decision, activity, or outcome of a project’ (Project Management Institute, 2013)
- Some of the project stakeholders include:
  - *Product owner (customer/sponsor)*
  - *Product users*
  - *Project team members*
  - *Manager*
- Each stakeholder has a different set of interests and communication needs.



# The Product Owner is mainly concerned that the project satisfies their needs.

- Is the team building the right project?
  - *Vision statement for features*
  - *Enumeration of epics and user stories*
  - *Priority of user stories*
- Is the team building the project right?
  - *Acceptance tests for each story*
- Term project
  - *Instructor is the Product Owner*
  - *Trello board for detailing epics and user stories with priority and acceptance criteria*
  - *If it is not in the Trello board, it is not in the project*



# The Product Users are concerned with the overall user experience.

- What is the user experience?
  - *User interface design*
  - *Workflow for using the product*
  - *Interface with other elements of the business processes*
- Project teams, unfortunately, often do not directly deal with the user or only do so through a proxy.
- Term project
  - *Instructor and course assistants as users*
  - *Other teams during cross-team testing*



# Project team members must communicate about the technical aspects of the project.

- Requirements
  - *User story creation with Product Owner (Trello board)*
- Design
  - *Architectural design (UI – Application – Model)*
  - *Structural design (UML class structure diagrams)*
  - *Behavioral design (UML sequence and statechart diagrams, web application interface)*
  - *Design narratives (Design discussion)*
  - *Whiteboard designs*
- Code
  - *Coding style guidelines (Google Java style)*
  - *Code reviews (GitHub pull requests)*
  - *Version control support (GitHub)*

Elements in ()s are related to term project



# Project team members must communicate about the process of doing the project.

- Planning
  - *Sprint planning and backlog refinement (Trello board)*
- Operational information
  - *Stand-up meetings (in-person, Skype/Hangout, Slack)*
  - *Information exchange (Slack)*
  - *Information archive (Trello card)*
- Team coordination
  - *Meeting scheduling (Doodle, When2Meet)*
  - *Information exchange (Slack)*
- Telling others about the project
  - *(Demos and presentations)*

Elements in ()s are related to term project

# Your team must determine what communication can be virtual and what is better face-to-face.

- Improved communication is the #1 item that teams say would help them be more effective
- Most would agree that a face-to-face meeting is the best for communication.
- The nature of modern software development often does not permit this.
- An effective software development team balances
  - *Virtual vs face-to-face meeting*
  - *Asynchronous vs synchronous communication*
- **BUT...**most work will be done asynchronously including coding



# A *Standup Meeting* is a nearly daily form of team communication. Here's how it works...

- A standup is a time-boxed meeting that relates progress and commitments across the team.
  - *This is the Development Team's meeting to run*
  - *The Product Owner may be present*
- Each team member answers three questions:
  - *What did I do yesterday?*
  - *What will I do today?*
  - *Are there any impediments to achieve my task?*



# What are some common gotchas running a Standup Meeting?

- Examples:

- *A non-team-member is directing the meeting*
- *Team lets the meeting run too long*
- *People sit and get too comfortable*
- *People digress on technical issues*
- *People try to solve the impediment*
- *People try to debate the merits of alternate solutions*

- Solutions?

- *Team must have discipline to cut-off rambling*
- *And to take debates and solutions out of the meeting*
  - ◆ Identify someone to help the person
  - ◆ And move on



# The Manager is interested in progress and individual accountability.

- Project progress and projections
  - *Card flow through lists (Trello board)*
  - *Product backlog (Trello board)*
- Individual accountability
  - *The Manager (your Instructor) needs to know who has done what work on the project.*
  - *Team members need visibility in artifacts*
    - ◆ Trello card log
    - ◆ Slack channel presence with meaningful contributions
    - ◆ Code level contributions
      - Commits performed
      - Issues opened/closed
      - Pull requests issued/reviewed
    - ◆ Peer evaluations

Elements in ()s are related to term project

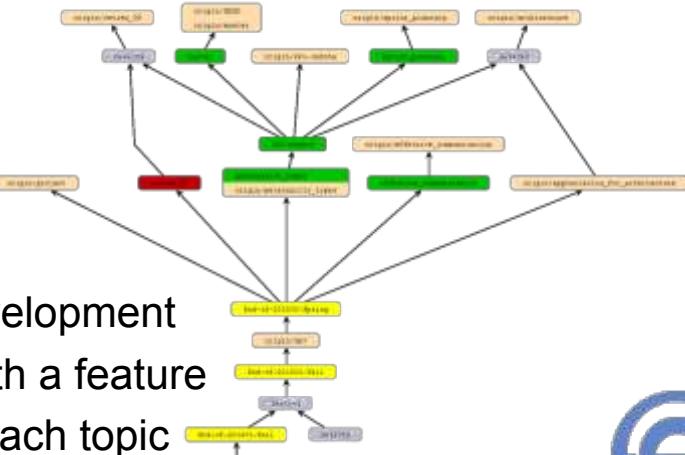


# These same tools were used while developing the course and every term while it is taught.

Planning with Trello board

Communication  
and coordination  
using Slack channels

Content development  
using Git with a feature  
branch for each topic



# Personality Types

**SWEN-261**  
**Introduction to Software**  
**Engineering**

Department of Software Engineering  
Rochester Institute of Technology

<b>ISTJ</b> 'Doing what should be done' -Inspector-	<b>ISFJ</b> 'A High Sense of Duty' -Protector-	<b>INFJ</b> 'An inspiration to others' -Counsellor-	<b>INTJ</b> 'Everything can be improved' -Mastermind-
<b>ISTP</b> 'Try anything once' -Crafter-	<b>ISFP</b> 'Sees much, shares little' -Composer-	<b>INFP</b> 'Noble service to society' -Healer-	<b>INTP</b> 'Love to solve problems' -Architect-
<b>ESTP</b> 'The ultimate realist' -Promoter-	<b>ESFP</b> 'You only live once' -Performer-	<b>ENFP</b> 'Give life an extra squeeze' -Champion-	<b>ENTP</b> 'One exciting challenge after another' -Inventor-
<b>ESTJ</b> 'Love to administer' -Supervisor-	<b>ESFJ</b> 'Love to be the host for everyone' -Provider-	<b>ENFJ</b> 'Smooth talking persuader' -Teacher-	<b>ENTJ</b> 'Life's natural Leaders' -Field Marshall-

<https://toughnickel.com/business/Top-Effective-Leadership-Skills-3-Leadership-Styles>  
© Mohan Kumar 2012



# Each person brings an individual perspective and personality to the project team.

- These differences can maximize the potential of the team, but also can be a source of tension.
- Every team member has to accept that there are different ways to approach work.
  - *None are right or wrong, or good or bad.*
  - *Some may be better suited for certain tasks than others.*
- Study of personality theory can provide insight for how to make a team more effective and reduce tension.

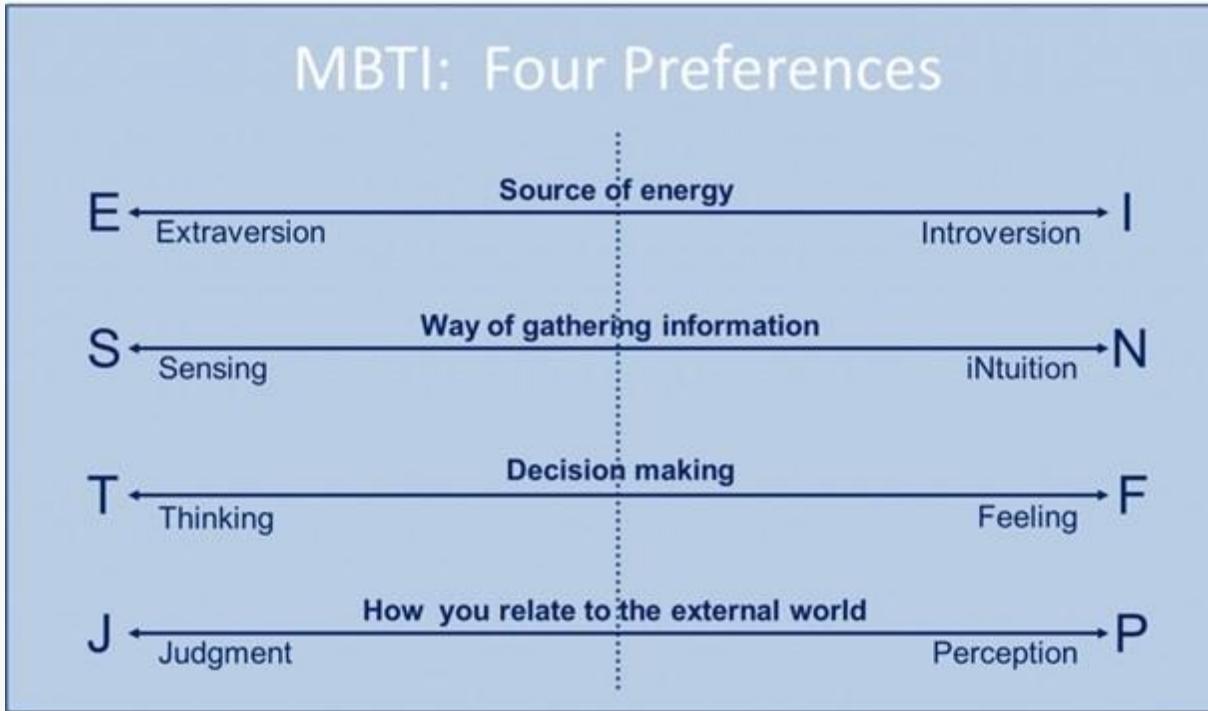


# Personalities, like fingerprints, are unique, but also like fingerprints, have recognizable patterns.

- Most work in personality typing has its roots in the work of psychologist Carl Jung.
- These personality types describe typical or habitual patterns and not fixed and unbreakable behaviors.
  - *Which do I do first?*
  - *Which is most comfortable or reliable for me?*
- Jung's theory was brought to an operational level by Katherine Briggs and Isabel Briggs Myers.



# The Myers-Briggs Type Indicator® defines a personality type on four dimensions.



# The most pervasive personality dimension is where you get your source of energy.

- Internal (Introversion) vs. External (Extraversion)
- Extraverted minds look outside for motivation to act, change, or interact.
- Introverted minds look for internal thoughts and ideas.
- Coming up with new ideas
  - *Extravert – lively brainstorming session*
  - *Introvert – individually working through ideas and reflecting over time*



# How you prefer to gather information is a second personality dimension.

- Direct observation (Sensing) vs. association of ideas, sixth sense (Intuition)
- Sensing prefers direct experimentation, action plans, practical application
- Intuition prefers goals, models, possibilities
- Deciding on a technology to use
  - *Sensing – what have others used and have they been successful*
  - *Intuition – this new technology seems good*



# The third personality dimension is how you decide on the meaning and importance of information.

- Analytic, cause-and-effect (Thinking) vs. value-oriented, idealistic (Feeling)
- Thinking prefers logical arguments, questioning the veracity of the information
- Feeling prefers values, examining impacts on individuals
- Deciding to outsource some project work
  - *Thinking – which will be less expensive and faster*
  - *Feeling – which will keep the team more motivated*



# The final personality dimension is based on how you relate to the external world.

- Judgment vs. Perception
- A Judging preference focuses on deadlines, task completion, making decisions
- A Perceiving preference focuses on adaptability, open to new opportunities
- Discussion during sprint retrospective
  - *J*udgment – *Did we clear the sprint backlog? How can we make better task estimates?*
  - *P*erception – *What new ideas did we uncover for the product?*

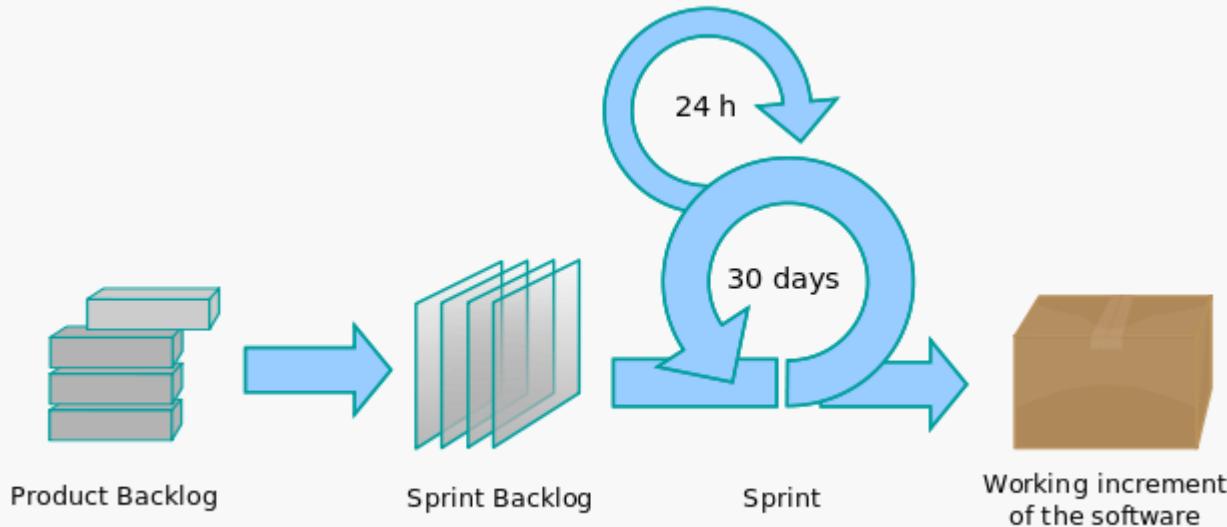


# An appreciation for different personalities is needed.

- Whether one personality theory or another is correct is not the main point.
- A personality model provides insight into the different preferences we each have for approaching our work and interacting with others.
- Your team is composed of different people which will make it a stronger team.



# Appreciation for: Software Development Process



By Lakeworks - Own work, GFDL, <https://commons.wikimedia.org/w/index.php?curid=3526338>

## SWEN-261 Introduction to Software Engineering

Department of Software Engineering  
Rochester Institute of Technology

# What do you want in a Software Development Process?

A few questions to consider:

- How does a software development process provide value to the customer?
- How does a software development process support a team to be predictable and dependable?
- How does a software development process improve software quality?



# These are the principles software engineers follow.

- Customer-focused
  - *A customer representative must be on "the team"*
  - *The customer validates each increment*
  - *The customer defines and prioritizes the requirements*
- Use an iterative process
  - *Build a working increment frequently*
  - *Demo the increment to the customer and get feedback*
- Manage risk
  - *Involve the customer to validate and elaborate requirements*
  - *Work on architecturally-significant features early*
- Use an empirical process
  - *Process is transparent to all stakeholders*
  - *The team inspects their own process each iteration*
  - *The team adjusts their process from lessons learned in the previous iteration*



# These are the core activities software engineers practice.

- Communicate
  - *Gather requirements to understand the customers needs and goals.*
  - *Demonstrate each project increment for the customer.*
- Plan
  - *Plan each iteration (sprint) of work so the stakeholders know what is expected.*
  - *Conduct retrospectives to improve the process.*
- Model
  - *Analyze the domain of the application for a deeper understanding of the requirements.*
  - *Design the system to meet the requirements.*
- Construct
  - *Implement a working increment within each iteration.*
  - *Test the system to validate that it meets the requirements.*
  - *Maintain the system by fixing bugs and creating new features.*
- Deploy
  - *Deploy software to the production environment.*
  - *Train users so they will succeed with the system.*

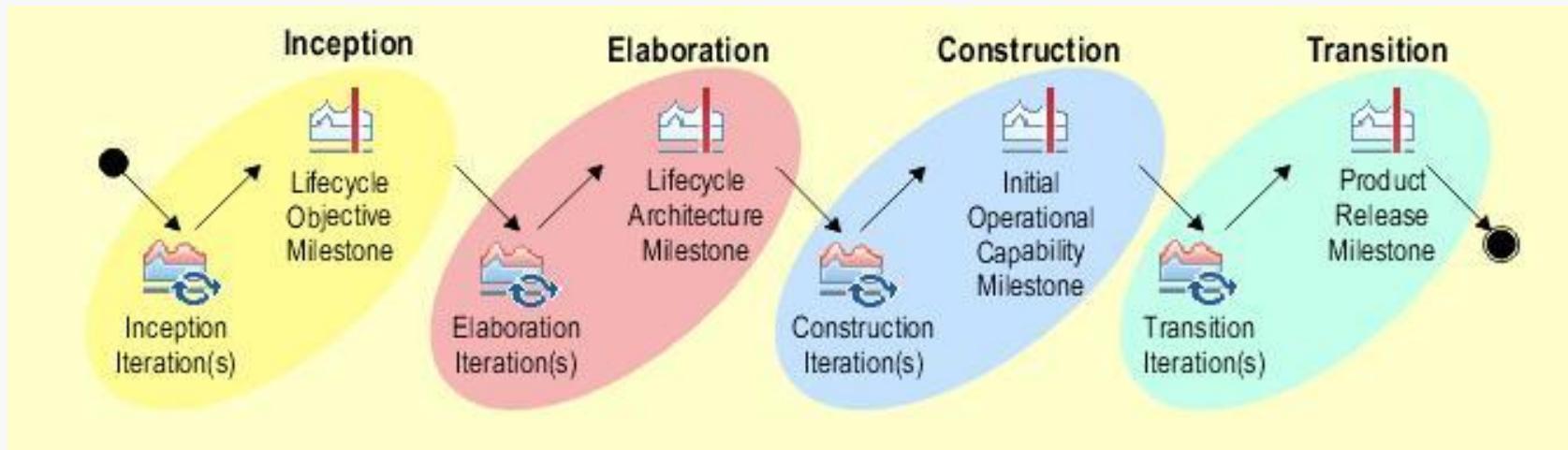


# A defined process codifies the way in which a team will adhere to those principles and practices.

- Traditional waterfall
- Spiral methodology
- Feature-driven development
- Rapid Application Development
- Extreme programming
- Rational Unified Process
- OpenUP ↪ *your project will use this*



# OpenUP describes project phases at the strategic level.

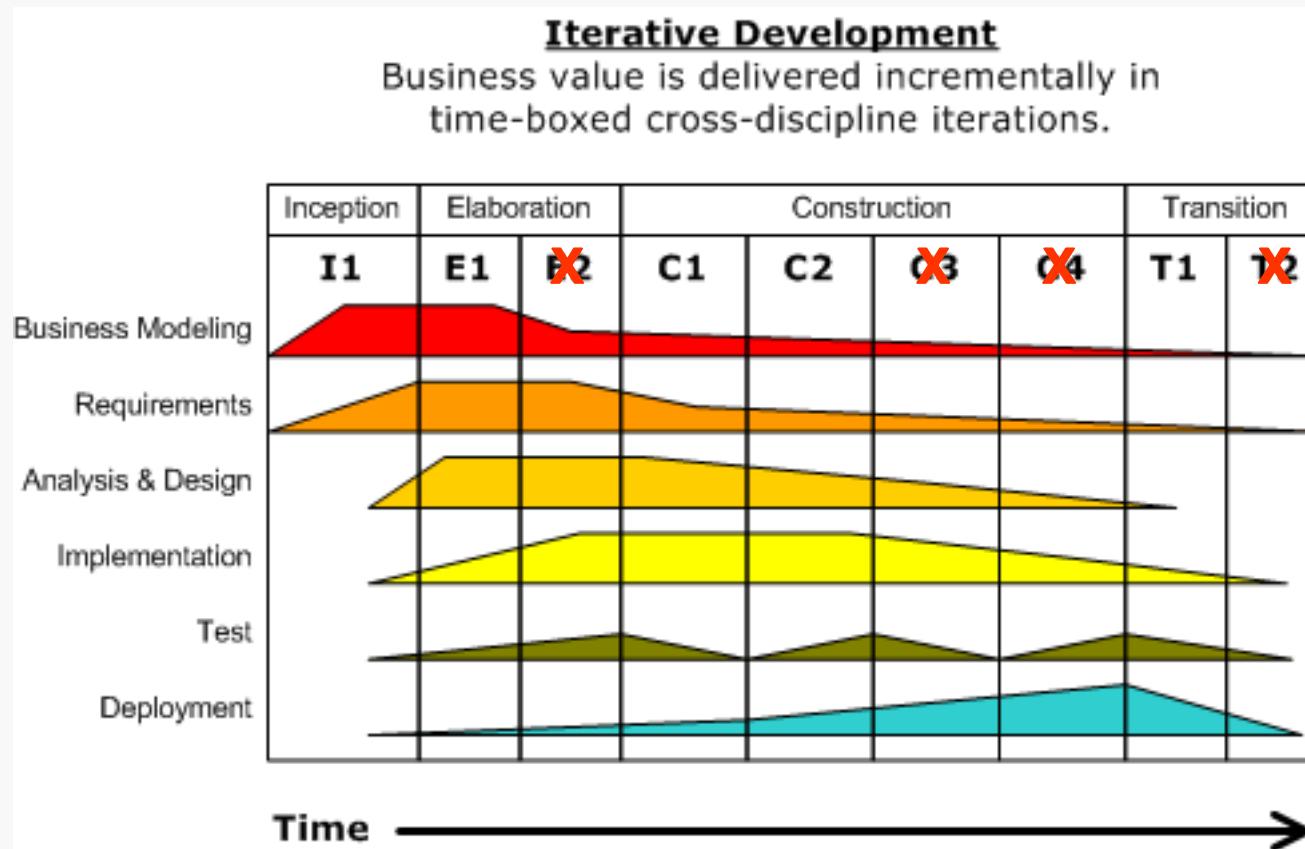


- Inception
  - *Manage requirements risks*
- Elaboration
  - *Manage architectural risks*
- Construction
  - *Build it out*
- Transition
  - *Deploy it*
  - *Training users*

By The original uploader was GFLewis at English Wikipedia - Transferred from en.wikipedia to Commons by IngerAlHaosului using CommonsHelper., EPL,  
<https://commons.wikimedia.org/w/index.php?curid=9030478>



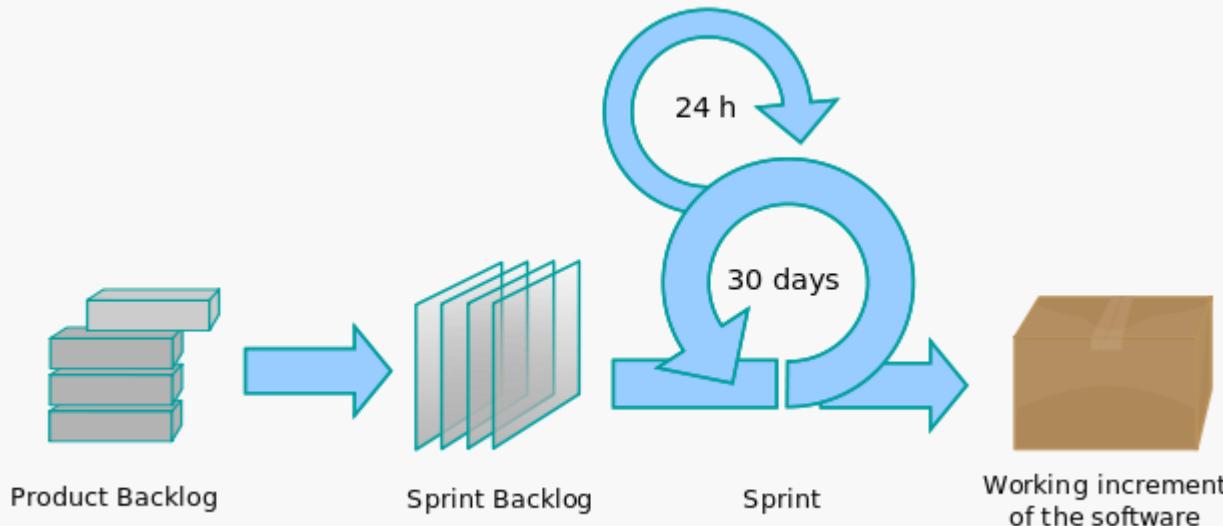
# A team's effort in different activity areas varies by phase.



By Dutchguilder - Own work, Public Domain, <https://commons.wikimedia.org/w/index.php?curid=37249677>



# You will use a Scrum process to define tactical activities.



By Lakeworks - Own work, GFDL, <https://commons.wikimedia.org/w/index.php?curid=3526338>

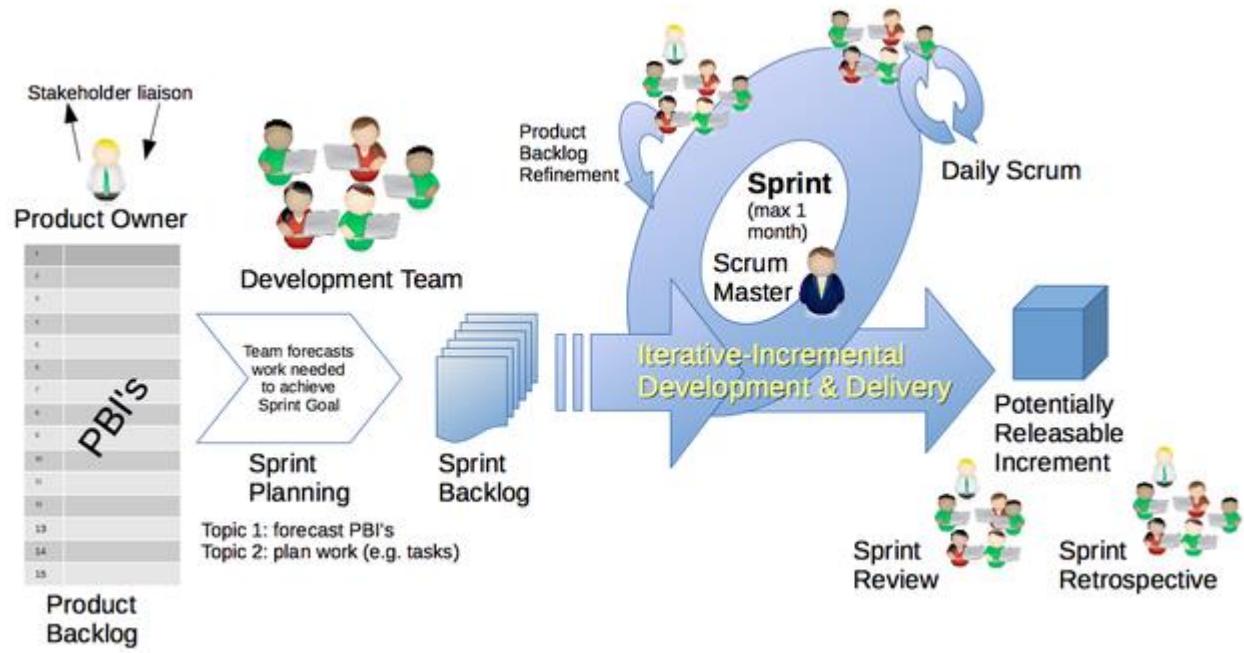
- A *Sprint* is a single iteration of work that results in a working system that delivers identified value to the customer.
- Each Sprint has its own *Sprint Backlog* of features pulled from the *Product Backlog*.
- Every day the team is working the *Sprint Backlog* to direct their development.



# Now you can take the quiz on Appreciation for software development process.



# Backlog Refinement and Estimation



**SWEN-261**

**Introduction to Software  
Engineering**

**Department of Software Engineering  
Rochester Institute of Technology**



**Software Engineering  
Rochester Institute  
of Technology**

# Before a story can be placed in the sprint backlog, it must be refined and given an effort estimate.

- The user story statement alone provides no detail about what is required or how to implement it.
- Backlog Refinement (also Backlog Grooming) adds those details.
  - *Acceptance criteria provide details of requirements*
  - *Solution tasks outline how to implement the story to satisfy the acceptance criteria*
- Based on the solution tasks, the developers can estimate story points using Planning Poker.
- With enough stories having story points, sprint planning can fill the sprint backlog up to the team's velocity.



# Acceptance criteria come from the Product Owner or user representatives.

- Defining acceptance criteria can be done in brainstorming sessions with the product owner
  - *The Product Owner leads the discussion and drives the exploration of the acceptance criteria.*
  - *The developers ask questions to further elaborate the acceptance criteria.*
  - *This can be done all together or in smaller groups discussing subsets of user stories.*



## **Let's review:**

# **What makes good acceptance criteria?**

- Like stories, focus on the *what* not the *how*.
- Use a Given/When/Then format:
  - ***GIVEN some precondition WHEN I do some action THEN I expect some result***
  - *Given* that I'm not signed-in *when* I visit the Home page *then* I expect to clearly see how to Sign-in.
  - *Given* that some other player has already signed-in with my name *when* I attempt to sign-in with my name *then* the system should reject the request with an error message and re-render the Sign-in form.



# With acceptance criteria defined, a developer then fleshes out a skeleton design.

- Evolve the analysis models:
  - *Explore new domain concepts*
  - *Alter existing domain model*
  - *Does the story alter the web interface? Then update the Statechart with your proposed states.*
- The design is very high-level:
  - *Create or modify Views and Controllers in the UI tier*
  - *Create or modify Services in the Application tier*
  - *Create or modify Entity or Value Objects in the Model tier*
  - *Create or modify any other helper component*
  - *Refactoring existing code to improve the overall design*



# These descriptions become tasks in the story's Trello card.

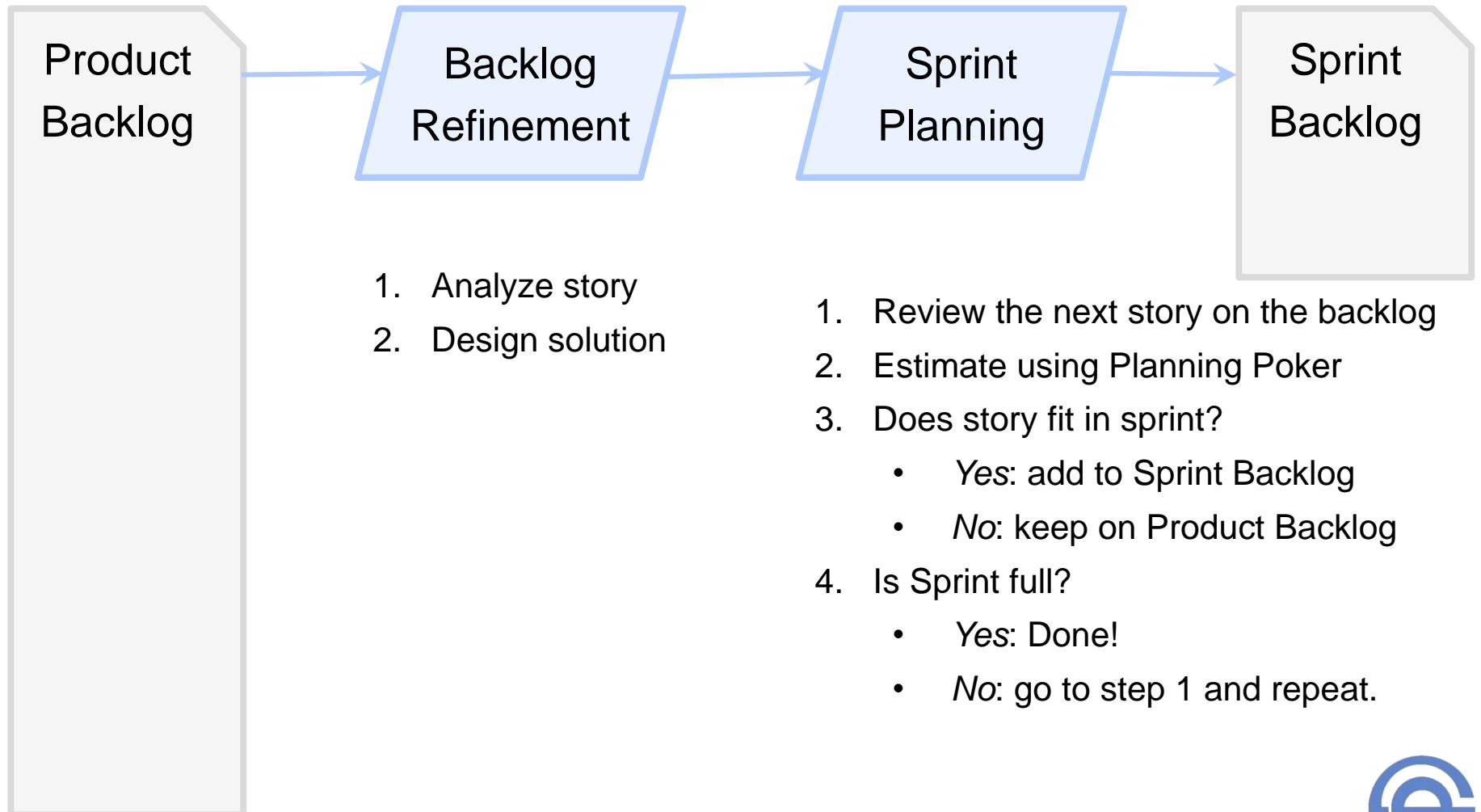
**Solution Tasks** [Delete...](#)

0%

- Create `GET /signin` Route component to render the sign-in page template
- Create the `signin.ftl` template with a text field and one button. The form action is a `POST` to the `signin` URL.
- Create the `POST /signin` Route component to process the sign-in request. Delegate Player signin to the `PlayerLobby` and if successful have the Route stores the `Player` in the HTTP session.
- Create the `Player` Model tier entity to hold the unique player name
- Create the `PlayerLobby` Application tier component to handle sign-in actions.
- Update the `GetHomeRoute` component to display the current (signed-in) Player and provide the list of all players to the View
- Update the `home.ftl` template with the sign-in link and display the current Player's name.
- Update the `home.ftl` template with a list of players, except the current Player. If the current user is not signed-in then display a message with the number of players but don't show a list of names.



# During Sprint X you refine stories in preparation for the Sprint Planning meeting for Sprint X+1.



# Planning poker is a technique devised by Mike Cohn.

- It is a form of *expert estimation* in which every team member is an expert.
- The *points* assigned are abstract; they do not relate to hours of effort.
  - *A sprint's capacity is not in hours but "level of effort"*
- The point system provides relative levels of effort.
  - *Small effort: 0, ½, 1, 2 or 3*
  - *Medium effort: 5, 8 or 13*
  - *Large effort: 20, 40 or 100*
  - *Unknown: ?*



# OK, but how do you estimate a story, really?

- Create an estimate for each *Solution Task* in your story design.
  - *Consider the type of component to build (or modify)*
  - *Consider the complexity of the feature*
  - *Consider how well you know the technology*
- Add up each task estimate and *round up* to the nearest Poker (Fibonacci) number.
- Expert developers do this calculation implicitly based their large experience base.



# Here is an example matrix of component estimation.

Architectural Tier	Component Type	Small / Low	Medium	Large / High
UI	UI View (View Template)	1	3	5
UI	UI Controller (Spark Route)	1	2	3
Application	Service	2	3	5
Model	Entity	1	2	3
Model	Value Object	1	2	3

- Each team member can independently estimate a user story by:
  - *For each class that will get touched/created when implementing the user story, identify its component type.*
  - *For each class, find its estimate in the chart based on your estimated level of development effort needed.*
  - *Add up all the class estimates to get your estimate for the user story.*



# Here's how Planning poker works.

1. The Product Owner reads the top story on the Product Backlog.
2. The team reviews the acceptance criteria and the suggested solution design.
3. To vote, each player picks the point card for his or her estimate.
4. Players reveal their cards all at once.
5. If there is consensus on one number, you're done.
6. Otherwise:
  1. *Have the outliers (high/low) explain their position*
  2. *Team discusses*
  3. *Vote again until consensus is reached*



# What should the team do if no consensus is found?

- There are usually two issues that prevent consensus.
- Product uncertainty:
  - *The requirements (acceptance criteria) are too vague*
  - *Send the story back for further (analysis) refinement*
- Technical uncertainty:
  - *Identify the uncertainty in the solution design*
  - *Create a spike story for this sprint to establish certainty*
  - *Send the story back for further (design) refinement*
- In either situation, the story should stay on the Product Backlog until the uncertainty is resolved.



# Object-Oriented Design II

**SWEN-261**  
**Introduction to Software**  
**Engineering**

Department of Software Engineering  
Rochester Institute of Technology

Controller  
Pure fabrication  
Open/close  
Polymorphism  
Liskov substitution



# The lesson continues building your object-oriented design skills with several other design principles.

- These are also principles from SOLID and GRASP

## SOLID

- Single Responsibility
- Open/closed
- Liskov substitution
- Interface segregation
- Dependency inversion

## GRASP

- Controller
- Creator
- Indirection
- Information expert
- High cohesion
- Low coupling
- Polymorphism
- Protected variations
- Pure fabrication



# **Controller** specifies a separation of concerns between the UI tier and other system tiers.

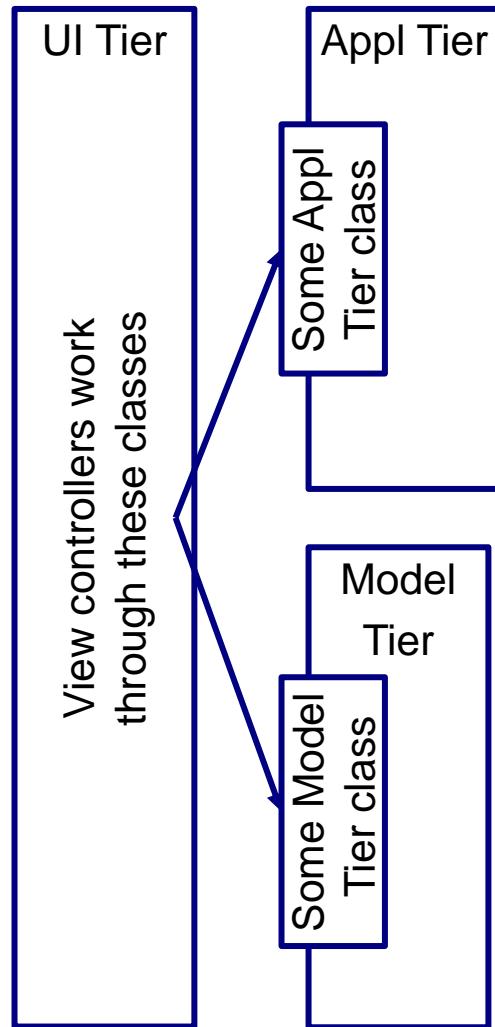
*Assign responsibility to receive and coordinate a system operation to a class outside of the UI tier.*

- "Controller" is an overused term in software design.
  - *In GRASP, this is not the view "controller" which is firmly in the UI tier.*
- In simple systems, it may be a single object that coordinates all system operations.
- In more complex systems, it is often multiple objects from different classes each of which handles a small set of closely related operations.

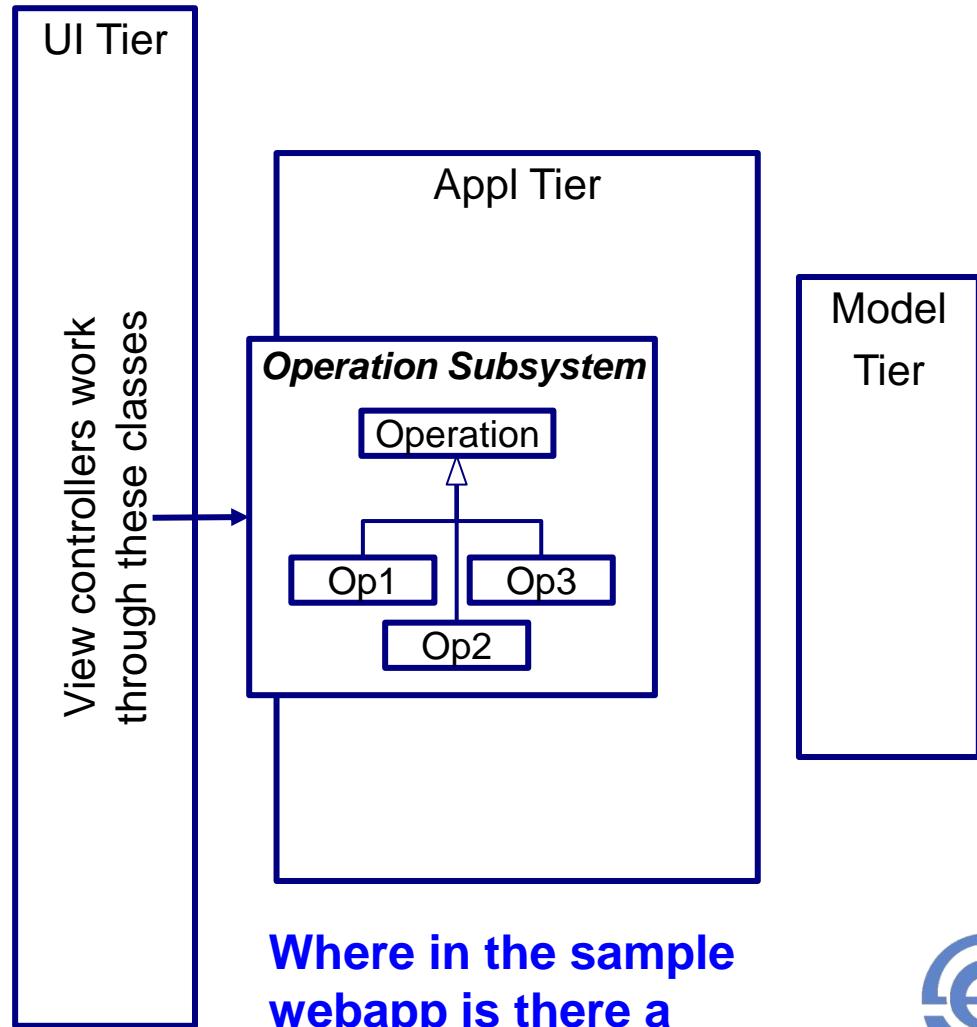


# Here is how GRASP controllers fit into the software architecture.

Simple System



More Complex System



Where in the sample  
webapp is there a  
GRASP-style controller?



# **Pure Fabrication** is sometimes needed to balance other design principles.

*Assign a cohesive set of responsibilities to a non-domain entity in order to support high cohesion and low coupling.*

- Your design should be primarily driven by the problem domain.
- To maintain a cohesive design you may need to create classes that are not domain entities.
- In the previous slide, the Operation Subsystem was a pure fabrication.

What were pure fabrications in the sample webapp? How could you have implemented it without those fabrications?



# The *Open/closed* principle deals with extending and protecting functionality.

*Software entities should be open for extension, but closed for modification.*

- Software functionality should be extendable without modifying the base functionality.
  - *Mostly provided by features of implementation language: inheritance, interface*
- Your design should consider appropriate use of
  - *Inheritance from abstract classes*
  - *Implementation of interfaces*
- Dependency injection provides a mechanism for extending functionality without modification.



# **Polymorphism creates a hierarchy when related behavior varies by class.**

*Assign responsibility for related behavior that varies by class by using polymorphic behavior.*

- Polymorphism is a primary object-oriented concept and should be used whenever possible
- Bad code smells that indicate a potential class hierarchy and use of polymorphism
  - ***Conditional that selects behavior based on a "type" attribute***
  - ***Use of instanceof or similar language constructs to select operations to perform***



# The *Liskov substitution* principle constrains the pre- and post-conditions of operations.

*Objects in a program should be replaceable with instances of their subtypes without altering the correctness of that program*

- Pre-conditions specify what must be true before a method call.
- Post-conditions specify what will be true after a method call.
- Design by Contract is a programming technique that requires formal definition of the pre- and post-conditions and has language support for it.

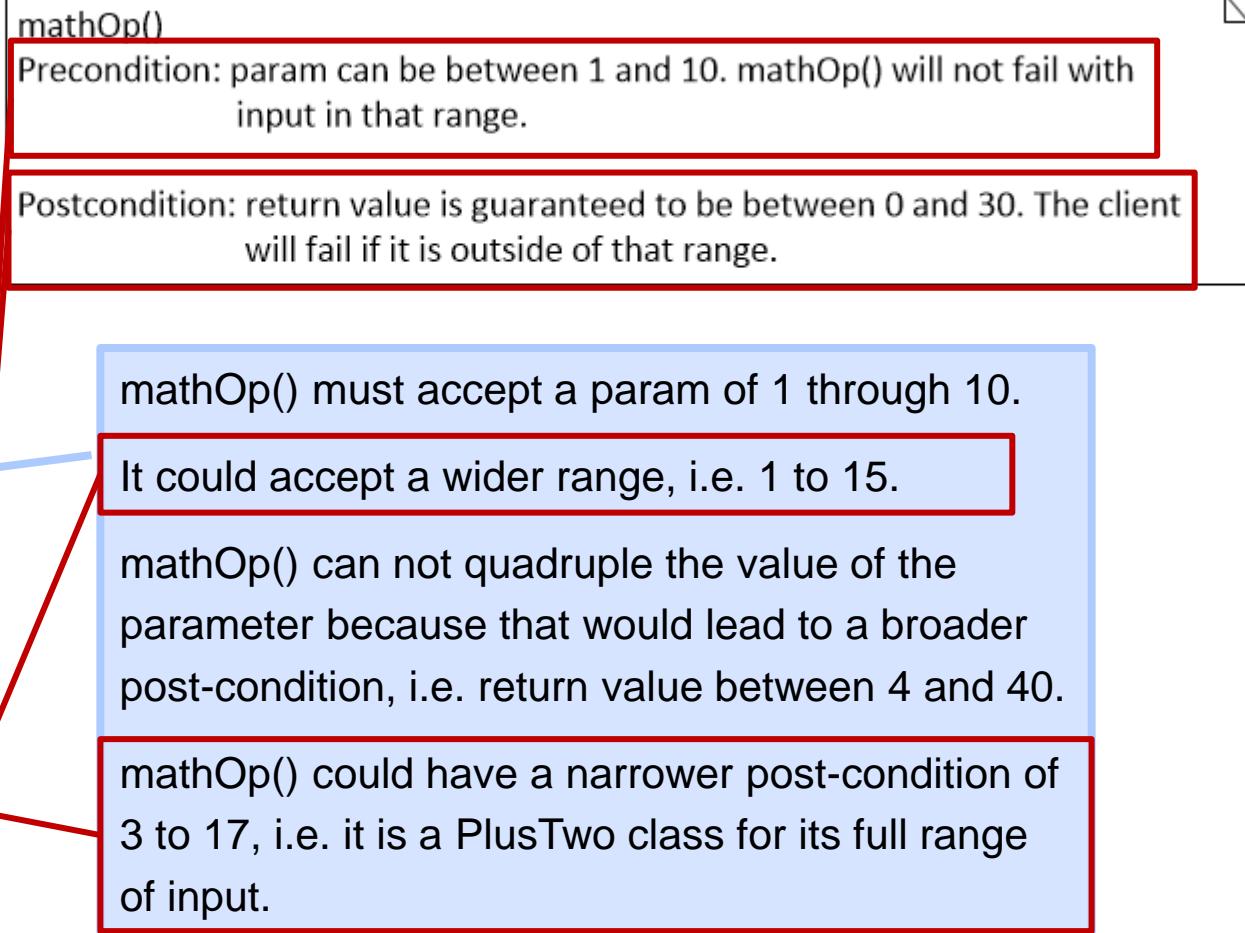
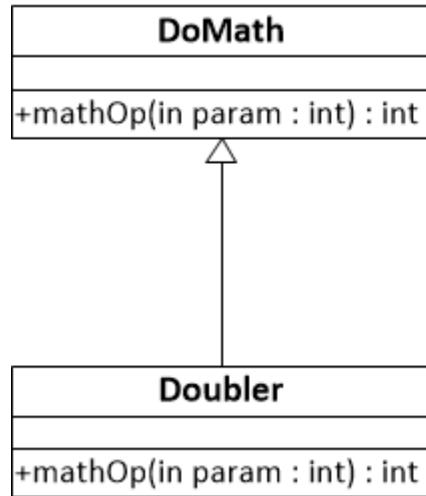


# **Any subclass of a class should be able to substitute for the superclass without error.**

- A subclass must not violate any of the pre- or post-conditions guaranteed by the superclass.
- Superclass clients count on the pre- and post-conditions being true even when polymorphism has the client interacting with a subclass.
- To maintain a pre-condition, a subclass must not narrow the pre-condition, i.e. be a subset.
- To maintain a post-condition, a subclass must not broaden the post-condition, i.e. be a superset.



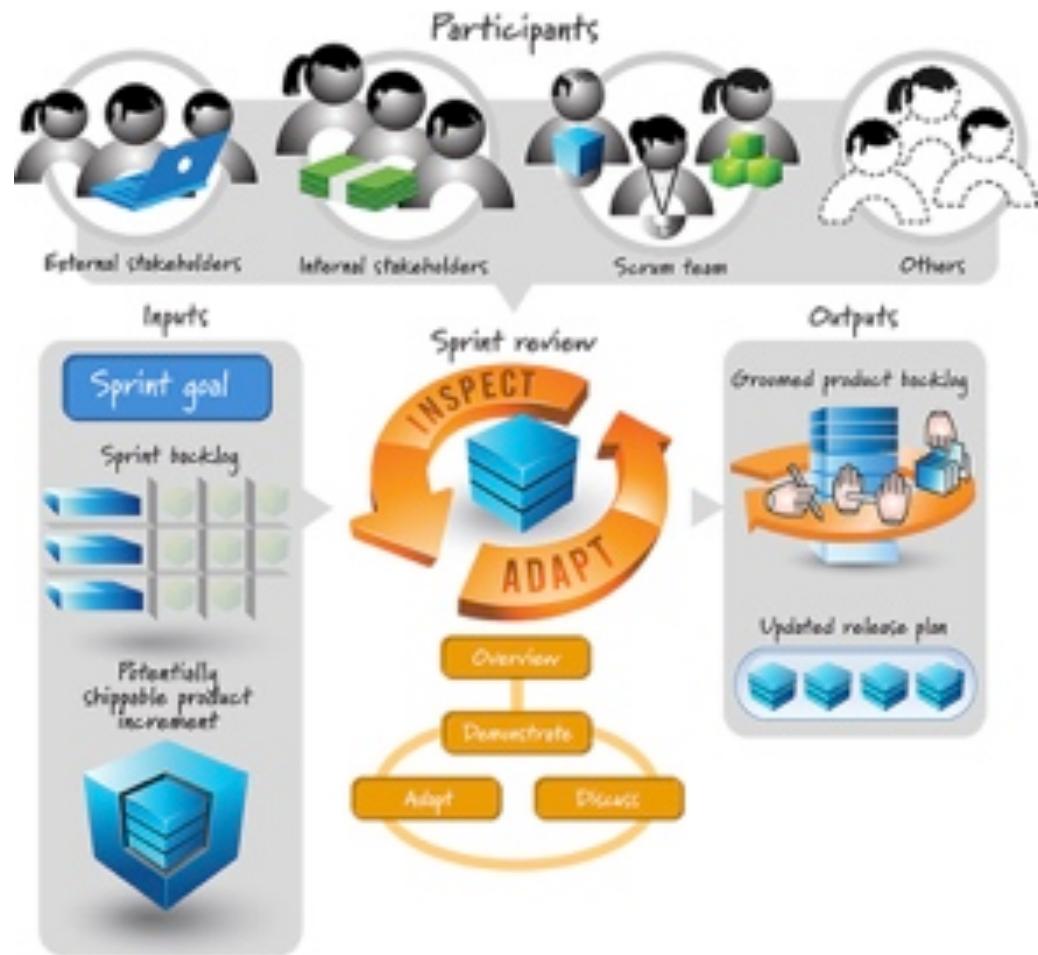
# Here is what Liskov substitution allows.



# Sprint Demo

**SWEN-261**  
**Introduction to Software**  
**Engineering**

Department of Software Engineering  
Rochester Institute of Technology



Copyright © 2012, Kenneth S. Rubin and Innovation, LLC. All Rights Reserved.



# A Sprint Review is a meeting with stakeholders to close out the current sprint.

- Review the stories completed in this sprint
  - *The team reviews the Sprint Board with stakeholders*
  - *The team provides a live demonstration of the stories completed in the current sprint*
- Discuss the direction of the project
  - *Stakeholders speak up about how well the current increment meets their needs*
  - *Provides new direction for the product by re-prioritizing the Product Backlog*
- For this lesson, we focus solely on the demo
  - **NOTE:** The pre-class video suggests that the Product Owner perform the demo. This is not a universal practice. There are teams that have testers perform the demo; as well.



# A successful sprint demo starts with planning.

- The agenda is a review of the sprint
  - *Show the Sprint Board*
  - *Show what is done and what isn't*
  - *Don't reflect on why things aren't done unless asked*
- Follow this with a sequence of story demos
  - *Only demonstrate done stories*
  - *Attempt to weave a story that connects multiple stories, if possible*
  - *Demo all acceptance criteria, if possible and time permits*
  - *Focus on business value and not technical details*
- Planning should take about an hour for 10 minutes of presentation time



# Once you have your plan, prepare for your demo.

- Prepare a statement about the Sprint Board
  - *Review each done story card in Trello*
  - *Briefly discuss the acceptance criteria*
  - *Mention the incomplete stories; open it up for questions*
- Prepare the demo
  - *Create a simple script of the story of each demo*
  - *Practice the demo and tweak the script as needed*
- Preparation should take about two to three hours for every 10 minutes
  - *Practice each demo independently*
  - *Then do at least one complete run-through*
  - *Practice should be done standing up and projecting your voice*

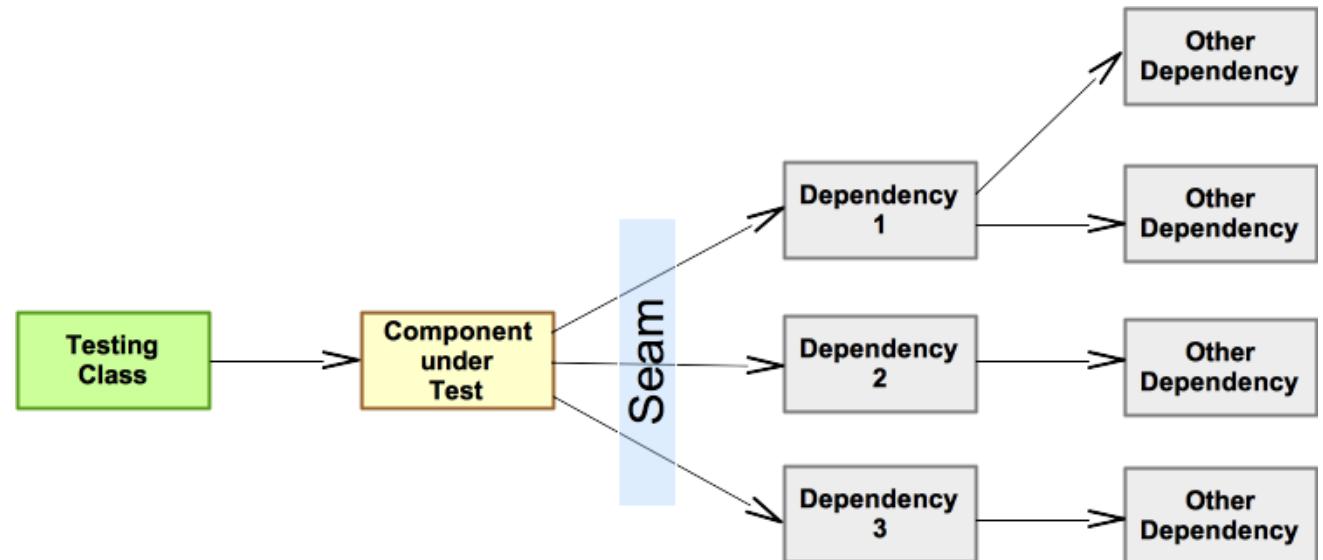


# Finally, rock your sprint demo.

- Giving the sprint demo is 50% theater and 50% story telling.
  - *The demo plan already has a story built in*
- Here are a few theater tips to make your presentation shine:
  - *You may be seated but position yourself facing the audience*
  - *Make eye contact as you speak*
  - *Speak up*
    - ◆ Extroverts usually have a strong tone so only increase your volume a little bit
    - ◆ Introverts tend to have a softer tone so you will need to increase your volume accordingly



# Unit Testing

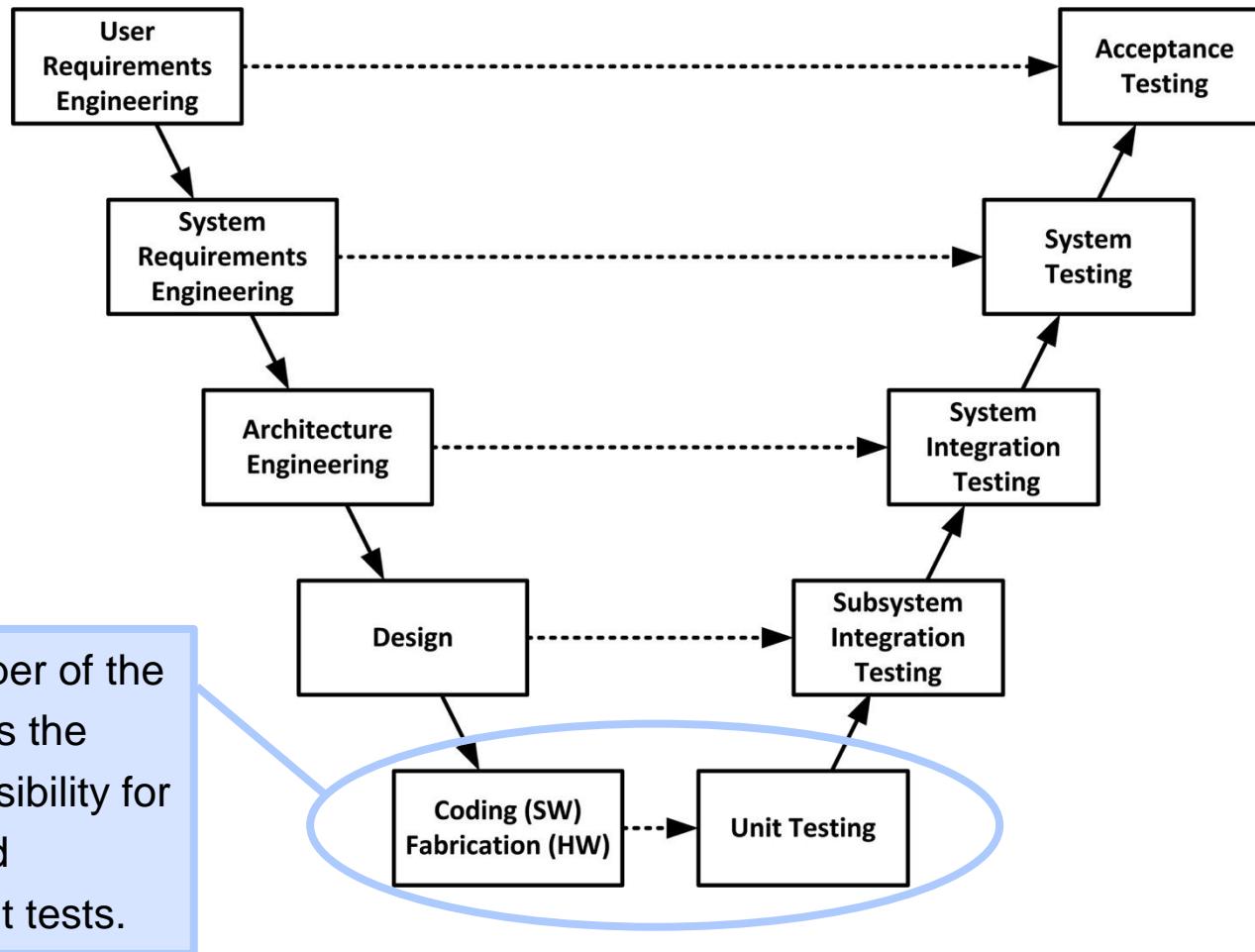


**SWEN-261**  
**Introduction to Software**  
**Engineering**

Department of Software Engineering  
Rochester Institute of Technology



# There are many levels of software testing.



[https://insights.sei.cmu.edu/sei\\_blog/2013/11/using-v-models-for-testing.html](https://insights.sei.cmu.edu/sei_blog/2013/11/using-v-models-for-testing.html)



# The scientific method that you studied in your physics courses is not just for science!

- Software testing and debugging is experimentation using the scientific method
  - ***Make a hypothesis***
    - ◆ Your software executes per the requirements
    - ◆ A certain place in your code is responsible for a bug
  - ***Create an experiment to test the hypothesis***
    - ◆ Write unit tests
    - ◆ Determine the scenario that triggers the bug and set breakpoints where you think the bug arises
  - ***Run the experiment***
    - ◆ Run the unit tests
    - ◆ Run the system to the breakpoint
  - ***Analyze the results to decide if the hypothesis is true***
    - ◆ If unit tests pass, the hypothesis is true
    - ◆ If the problem is seen, you have found the bug location



# These are the essential characteristics that should be your goals when you write unit tests.

- Automatic
- Thorough
- Repeatable
- Independent
  - *Test only one thing at a time*
  - *Tests should not depend on each other*
- Professional
  - *Readable*
  - *Use standard OO principles*
- Fast
  - *Using Maven, unit tests are run on each build*



# This is the checklist of items that you should ensure get tested by a class' unit tests.

- Business logic
  - *Tests for each path through a method*
  - *Happy path as well as failure paths*
- Constructors and accessors
- Defensive programming checks
  - *Validation of method arguments*
    - ◆ NullPointerException
    - ◆ IllegalArgumentException
  - *Validation of component state*
    - ◆ IllegalStateException
- The equals and hashCode methods, as needed
- Exception handling



# Here's the typical formula for a test.

- Create a test class for each component under test (CuT) that has privileged access to the component.
  - *In Java, the test code typically has package access.*
- Create a test method that covers each happy and failure scenario.
  - *Setup test scenario*
  - *Invoke the test*
  - *Analyze the test results*
- This simple formula can be adjusted as needed.
- Note: see the [after-class exercise](#) for more detailed instructions for unit testing with JUnit, Mockito, and Maven along with examples from the **GuessGame** unit tests.



# Unit testing frameworks like JUnit have many built-in assertions for analysis in your tests.

- Test truth-hood
  - *assertTrue (condition[, message])*
  - *assertFalse (condition[, message])*
- Test values or objects for equality
  - *assertEquals (expected, actual[, message])*
  - *assertNotEquals (expected, actual[, message])*
- Test objects for identity (`obj1 == obj2`)
  - *assertSame (expected, actual[, message])*
  - *assertNotSame (expected, actual[, message])*
- Test null-hood
  - *assertNull (object[, message])*
  - *assertNotNull (object[, message])*
- Automatic failure: *fail (message)*



# One tip for coding for testability is to make strings and constants available to the unit test.

- Idioms for testable code:

- *Make message strings (and other values) constants*
- *Make these members package-private (or public)*

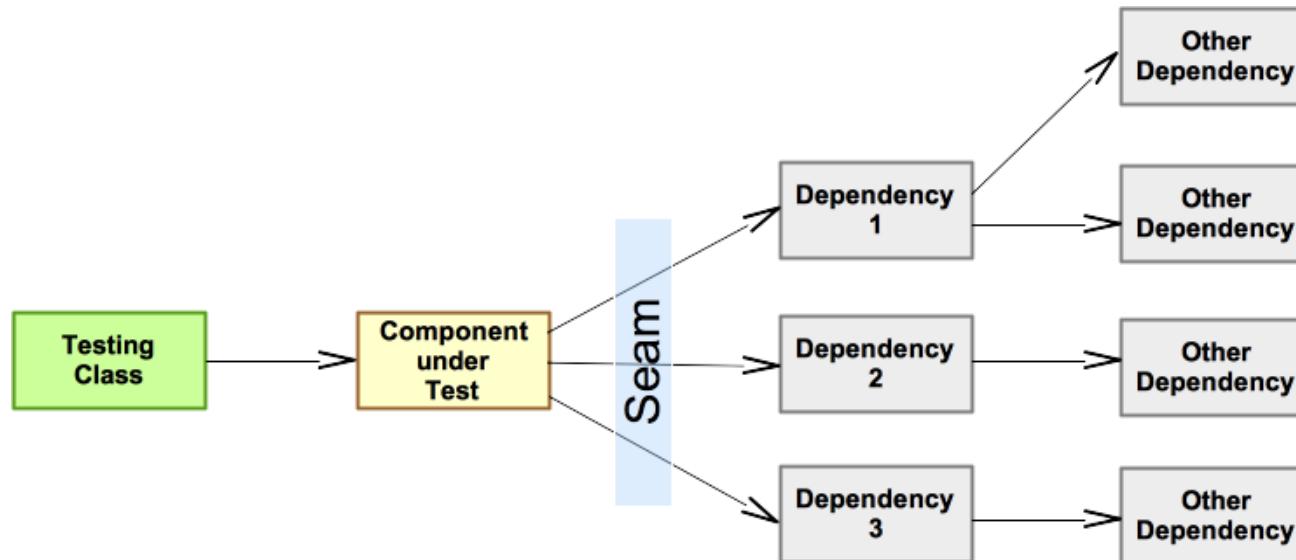
- Example:

```
public class GameCenter {  
  
    final static String NO_GAMES_MESSAGE = "No games have been played so far.";  
    final static String ONE_GAME_MESSAGE = "One game has been played so far.";  
    final static String GAMES_PLAYED_FORMAT = "There have been %d games played."  
  
    public synchronized String getGameStatsMessage() {  
        if (totalGames > 1) {  
            return String.format(GAMES_PLAYED_FORMAT, totalGames);  
        } else if (totalGames == 1) {  
            return ONE_GAME_MESSAGE;  
        } else {  
            return NO_GAMES_MESSAGE;  
        }  
    }  
}
```



# To write an "independent" unit test of a class, you will need to control the dependencies.

- Most components will have dependent classes
  - *Stored as attributes*
  - *Passed in as arguments*
- This coupling forms a "seam" between the component under test and its dependencies:

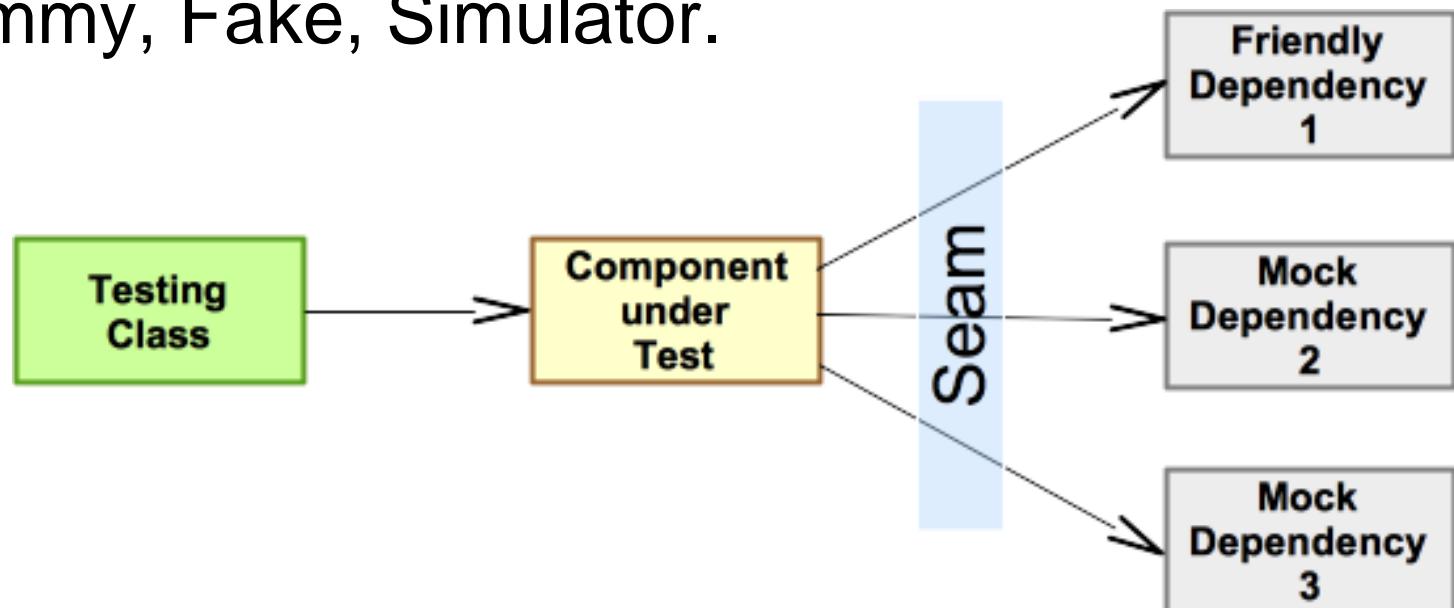


- Use dependency injection rather than direct instantiation.



# Isolate the CuT by injecting *friendly* and mock objects.

- Fully tested internal components (usually Model objects like Entities and Value Objects)
- Mock objects that stand in for internal or external components
- There are other variations on test instruments: Spy, Dummy, Fake, Simulator.



# Mock object frameworks like Mockito have APIs for setting up test scenarios and inspecting results.

- Setting up test scenarios
  - *Specify the response, e.g. return value or exception thrown, when a method is called on a mock object*
  - `when(mock.method(args)).thenReturn(value)`
- Verify that interactions occurred during a unit test
  - *Check if methods were called the expected number of times in the manner expected*
  - `verify(mock, times(1)).method(args)`
- Answer objects that allow you to capture and later test intermediate results of unit test



# Make unit testing part of your project's *Definition of Done* and your quality will rise!

- Unit testing is a subtle art.
- Keep your test code isolated from production code.
- Use the testing formula: arrange, invoke, analyze.
- Use an Assertion library like JUnit.
- Use a Mock object framework like Mockito.
- See [after-class exercise](#) for more detailed examples of doing unit tests with JUnit, Mockito, and maven.



# Make unit testing part of your project's *Definition of Done* and your quality will rise!

Definition of Done Checklist [Delete...](#)

0%

- acceptance criteria are defined
- solution tasks are specified
- feature branch created
- unit tests written
- solution is *code complete*, i.e. passes full suite of unit tests
- design documentation updated
- pull request created
- user story passes all acceptance criteria
- code review performed
- feature branch merged into master
- feature branch deleted

See [after-class exercise](#) for more detailed examples of doing unit tests with JUnit, Mockito, and maven.



# Design Documentation

**SWEN-261**  
**Introduction to Software**  
**Engineering**

**Department of Software Engineering**  
**Rochester Institute of Technology**

## Guessing Game Home

Welcome to the Guessing Game Project!

### Team

- Bryan Basham
- Jim Vallino

### Design Documentation

Click above for details of the Guessing Game design documentation.

### Setup Guide

Click above for details about how to setup your development environment to work on this project.



**Software Engineering**  
Rochester Institute  
of Technology

# Design documentation can be a valuable communication tool.

A design document is a way for you to communicate to others what your design decisions are and why your decisions are good decisions.

From [How to Write an Effective Design Document](#) by Scott Hackett

- Design documentation should be short and easy to read.
- It should communicate key architecture and design decisions.
- It should generally move from high-level to low-level.
- It should provide justification for design decisions.



# We recommend a simple design document structure.

- Executive Summary
  - *Purpose*
  - *Glossary and Acronyms*
- Requirements
  - *Definition of MVP*
  - *MVP Features*
  - *Roadmap of Enhancements*
- Application Domain
  - *Overview of Major Domain Areas*
  - *Domain Area Detail*
- Application Architecture
  - *Summary*
  - *Overview of User Interface*
  - *Tier Designs (UI, Application, Model)*
    - ◆ Summary
    - ◆ Static Model(s)
    - ◆ Dynamic Model(s)



# These general tips for effective writing apply to your design documentation too.

- Create a narrative to engage the reader.
- Writing a spec is like writing code for a brain to execute.
- Write as simply as possible.
  - *Use the active voice.*
  - *Use short, declarative statements.*
- Review and reread several times.
- Balance text with diagrams.
  - *Don't have long stretches of text.*



# You should follow these tips to maximize the effectiveness and professionalism of your models.

- Define a purpose for each model/diagram and use a level of abstraction appropriate for the purpose.
- Use standard modeling techniques (ie, UML).
- Use non-standard models when they are clearer than the alternatives.
- Use a professional modeling tool.
- Create a layout that is easy to comprehend.
- Use color, fonts and styles that enhance understanding (high-light important elements)
- *BUT...* do not use such stylistic frills for solely aesthetic purposes.



# The software design is just one aspect of a project that can be documented.

- Others include:
  - *Setup guide*
  - *UI and UX design and style guide*
  - *Acceptance test suite*
  - *Online and in-system help docs*
  - *Training docs and video tutorials*
- Project documents must *live*:
  - *Use collaborative, versionable documentation tools.*



# Keeping your design documentation up-to-date will now be part of your standard workflow.

**Definition of Done Checklist** [Delete...](#)

0%

- acceptance criteria are defined
- solution tasks are specified
- feature branch created
- unit tests written
- solution is *code complete*, i.e. passes full suite of unit tests
- design documentation updated**
- pull request created
- user story passes all acceptance criteria
- code review performed
- feature branch merged into master
- feature branch deleted



# Code Coverage

**SWEN-261**  
**Introduction to Software**  
**Engineering**

Department of Software Engineering  
Rochester Institute of Technology

Element	Missed Instructions	Cov.	Missed Branches	Cov.
● <a href="#">makeGuess(int)</a>		93%		83%
● <a href="#">GuessGame(int)</a>		100%		100%
● <a href="#">isFinished()</a>		100%		100%
● <a href="#">toString()</a>		100%		n/a
● <a href="#">isValidGuess(int)</a>		100%		100%
● <a href="#">static {...}</a>		100%		n/a
● <a href="#">isGameBeginning()</a>		100%		100%
● <a href="#">hasMoreGuesses()</a>		100%		100%
● <a href="#">GuessGame()</a>		100%		n/a
● <a href="#">guessesLeft()</a>		100%		n/a
Total	3 of 148	97%	2 of 28	92%
Model Tier				



# Code coverage analysis is measuring how well your unit tests exercise the production code.

- Code coverage works like this:
  1. *Compile the project into bytecode*
  2. *Instrument the bytecode with "touch points"*
  3. *Run the unit tests, which gathers coverage data*
  4. *Generate a coverage report from the gathered data*
- There are a few Java coverage tools.
  - *Your project will use JaCoCo*
  - *It integrates well with Maven*
- Having this information is a double-edge sword.
  - *It's mostly a positive thing; telling the team where to spend additional testing effort.*
  - *But don't be a slave to the metrics; we'll talk more about this later.*



# JaCoCo's coverage report is a simple HTML web site that lets you drill down for more information.

- The report is stored in /target/site/jacoco.

Element	Missed Instructions	Cov.	Missed Branches	Cov.
<a href="#">com.example</a>		0%		n/a
<a href="#">com.example.ui</a>		83%		85%
<a href="#">com.example.model</a>		90%		100%
<a href="#">com.example.appl</a>		94%		100%
Total	159 of 780	79%	1 of 43	97%

Element	Missed Instructions	Cov.	Missed Branches	Cov.
<a href="#">GuessGame</a>		97%		92%
<a href="#">GuessGame.GuessResult</a>		100%		n/a
Total	3 of 197	98%	2 of 28	92%

# It's at the class-level where you can start a meaningful analysis.

Element	Missed Instructions	Cov.	Missed Branches	Cov.
makeGuess(int)		93%		83%
GuessGame(int)		100%		100%
isFinished()		100%		100%
toString()		100%		n/a
isValidGuess(int)		100%		100%
static {...}		100%		n/a
isGameBeginning()		100%		100%
hasMoreGuesses()		100%		100%
GuessGame()		100%		n/a
guessesLeft()		100%		n/a
Total	3 of 148	97%	2 of 28	92%

Model Tier

```
113.     public synchronized GuessResult makeGuess(final int myGuess) {  
114.         final GuessResult thisResult;  
115.         // validate arguments  
116.         if (myGuess < 0 || myGuess >= UPPER_BOUND) {  
117.             thisResult = GuessResult.INVALID;  
118.         } else {  
119.             // assert that the game isn't over  
120.             if (howManyGuessesLeft == 0) {  
121.                 throw new IllegalStateException("No more guesses allowed.");  
122.             }  
123.         }  
124.     }  
125. }
```

## Color Legend

Green → covered

Yellow → partially covered

Red → not covered



# The GuessGame code had 97% coverage. So what do you do?

- On the one hand:
  - *That's REALLY good already.*
  - *The only missing test is a defensive check so maybe say "that's good enough."*
- On the other hand:
  - *This is a core Model tier class.*
  - *We want these to be "friendly" test dependencies.*
  - *So maybe the team agrees to make this 100% covered.*
- What tests need to be added?



# There needs to be a test to check making an invalid guess.

- Here's a test.

@Test

```
public void make_an_invalid_guess() {  
    final GuessGame CuT = new GuessGame();  
    assertEquals(CuT.makeGuess(TOO_SMALL), GuessResult.INVALID);  
    assertFalse(CuT.isFinished(), "Game is not finished");  
}
```

- Here's the updated analysis:

```
113. public synchronized GuessResult makeGuess(final int myGuess) {  
114.     final GuessResult thisResult;  
115.     // validate arguments  
116.     if (myGuess < 0 || myGuess >= UPPER_BOUND) {  
117.         thisResult = GuessResult.INVALID;  
118.     } else
```

This line is tested but only through this part of the branch.

We need to test the second part of the branch.



# If we test that second branch, we should be there.

- Here's a test of a guess that is too big.

@Test

```
public void make_an_invalid_guess_too_big() {  
    final GuessGame CuT = new GuessGame();  
    assertEquals(CuT.makeGuess(TOO_BIG), GuessResult.INVALID);  
    assertFalse(CuT.isFinished(), "Game is not finished");  
}
```

- Here's the updated analysis:

```
113.     public synchronized GuessResult makeGuess(final int myGuess) {  
114.         final GuessResult thisResult;  
115.         // validate arguments  
116.         if (myGuess < 0 || myGuess >= UPPER_BOUND) {  
117.             thisResult = GuessResult.INVALID;  
118.         } else {
```

- Now the Model tier is fully tested!

**com.example.model**

Element	Missed Instructions	Cov.	Missed Branches	Cov.
GuessGame	<div style="width: 100%;"> </div>	100%	<div style="width: 100%;"> </div>	100%
GuessGame.GuessResult	<div style="width: 100%;"> </div>	100%		n/a
Total	0 of 197	100%	0 of 28	100%

Model Tier



# Deciding what level of coverage depends upon several factors...

- Some components (Model tier) are used across multiple other architectural tiers.
  - *We recommend 95% or better for Model tier.*
- Others, like the UI Controllers, are only used by the web server.
  - *We recommend 80% or better in all other tiers.*
- Other factors:
  - *Team and company culture*
  - *Application domain*
    - ◆ Regulatory requirements may specify testing requirements.
    - ◆ Those defensive checks may be safety checks. You can not know if the system is safe if you do not test the checks.



# The coverage data is cumulative across all tests which may make results look better than they are.

- You want to gather coverage data from unit tests of a class not use of the class by tests of other classes.
  - *The ultimate is to test one class at a time which is not reasonable.*
  - *A reasonable compromise is measure code coverage for testing one tier at a time.*
- The JUnit framework and build tools allow that.
  - `@Tag ("name")` **each test file to place it into a tier category. Use** Model-tier, Application-tier, UI-tier
  - *Reset the coverage data after each tier is tested and generate the report in a separate location.*



# Your project's pom.xml file has several test execution ids defined.

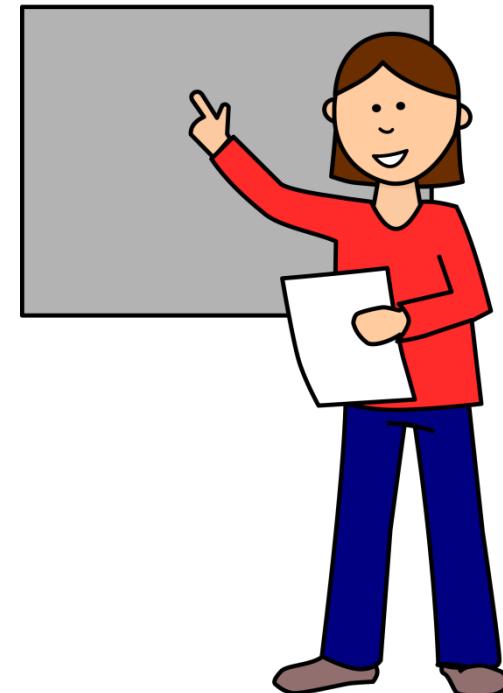
- Clean the target directory, and run all three tier-based tests
  - mvn exec:exec@tests-and-coverage
  - ***The reports are in*** /target/site/jacoco/**tier**/index.html **where tier is** model, ui, **or** appl.
- To run tests on a single tier
  - mvn clean test-compile surefire:test@**tier** jacoco:report@**tier** **where tier is** model, ui, **or** appl.
  - ***The report is in the directory listed above.***



# How to Make a Presentation

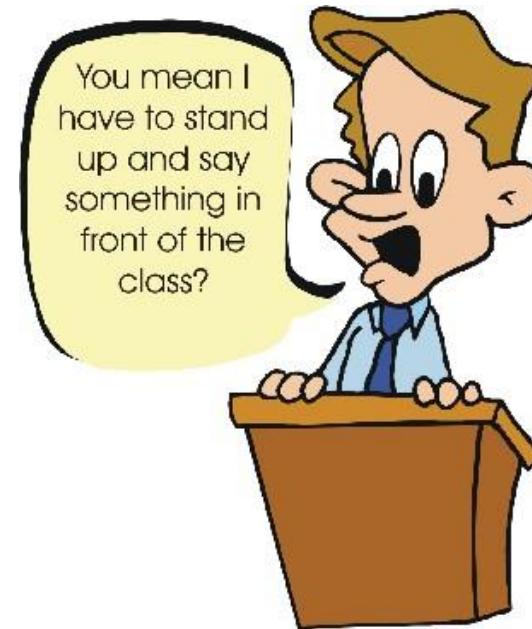
**SWEN-261**  
**Introduction to Software**  
**Engineering**

Department of Software Engineering  
Rochester Institute of Technology



Software Engineering  
Rochester Institute  
of Technology

# Making a presentation can be a frightening experience for some people.



- How to deliver an effective presentation is an important skill for your professional career.
- It would be good for you to be comfortable communicating your technical ideas to others.



# But first, let's reflect on that instructor you had whose lectures were just so bad.

- What made them that bad?
  - *Droned on in a monotone*
  - *Spoke too softly for you to hear*
  - *Only looked at the screen or board*
  - *Presented dense slides filled with text, charts, and images that you could not read*
  - *Simply read the text to you*
  - *Material had no apparent organization or flow*
  - *Tried to present too much so rushed or skipped stuff*
  - *Did not seem to know the material very well*
  - *Told lousy jokes*
  - *Never allowed for any questions*
- Make sure your presentation avoids these faults



# A presentation is not something that just happens.

- To give a good presentation, especially within the allowed time, takes preparation and organization.
- You need to start with top-level "requirements".
  - ***Who is the audience?***
    - ◆ What is their expertise level in the area?
  - ***What is the purpose of the presentation?***
    - ◆ What do you want to get across to the audience?
    - ◆ What does the audience want to hear or learn?
  - ***How much time do you have?***



# Based on the requirements that you have, create an outline for the presentation.

- Your outline can be in terms of slides that you will have in the presentation.
  - *Roughly identify the purpose for each slide in terms of how it will contribute to the overall purpose*
  - *At this top level, identify the flow of information from one slide to the next*
- You will know more details than you will have time to cover.
  - *What are the most important points to hit?*
  - *Where should you spend time on details?*
  - *Two minutes per slide is a good ballpark; unless the slides have minimal information, one minute per slide will be rushing the slide*



# Like most things, you will get better with practice.

- Doing more presentations should improve your general presentation skills and reduce anxiety.
- Practicing a specific talk will help you nail it.
  - *Go over it in your head*
  - *Run through it by yourself*
  - *Have the team practice the entire talk*
- Learn from your practice
  - *Carefully note areas that seemed rough*
  - *Use different wording if you stumble on something*
  - *The team should constructively critique each other's section of the presentation*



# You must provide graphics to support your design discussion.

- Your documentation will provide many more details than you can give in your presentation.
- You will use a variety of graphics
  - *Break the system into multiple class diagrams that are readable by the audience, i.e. less detail*
  - *Statecharts for web application interface, and class behavior*
  - *Sequences diagrams to show flow of a feature*
- Be clear on the purpose for the slide
  - *Have your diagram target that level of information.*



# Finally, you should be proud of the work that you have done on the project.

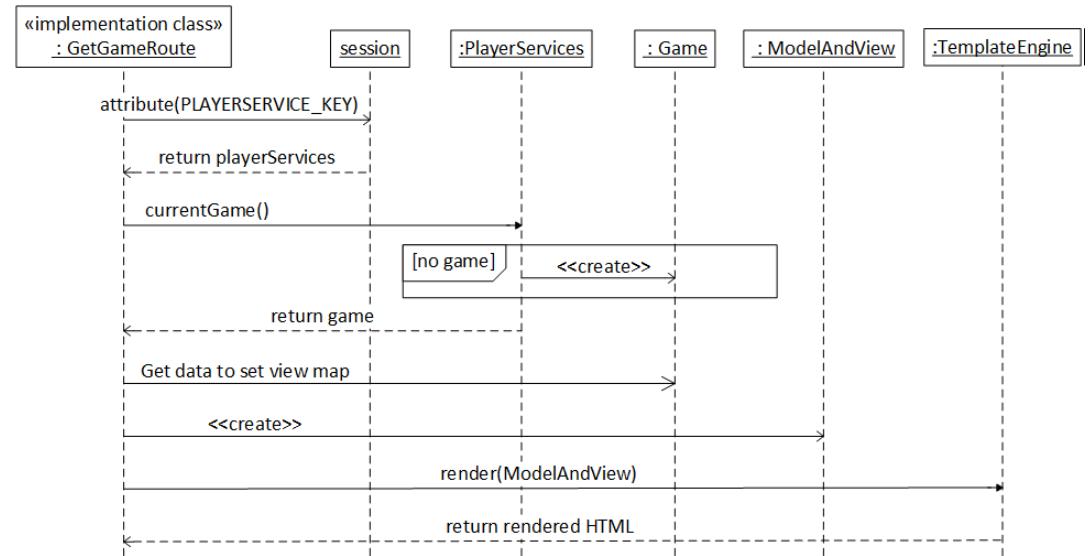
- Let that pride show through to the audience.
- This is your work. Shout out about it.
  - *Tell the audience about what you did.*
  - *It will not be perfect.*
  - *There may be places with critical comments given.*
  - *If you made a truly diligent effort on the project work as a team you have done what was asked of you.*



# Sequence Diagrams

**SWEN-261**  
**Introduction to Software**  
**Engineering**

Department of Software Engineering  
Rochester Institute of Technology



# The sequence diagram is a basic tool for modeling dynamic interactions between software entities.

- Sequence diagrams can be used at various levels of abstraction.
  - *Business workflow*
  - *User story feature flow*
  - *Object-level interactions* ← We will look at this level.
- At any abstraction level, the diagram captures the high-level information not every detail
- The notation is simple to grasp
  - *Time progresses top to bottom*
  - *Operations generally flow left to right*
  - *Show method calls with parameters*
  - *Show return values when important*
  - *Can show creation and deletion of objects*



# These are the basic notations for sequence diagrams that you can use.

Object name indicates everything

Exact object not known or does not add information

Fully qualified

Object1

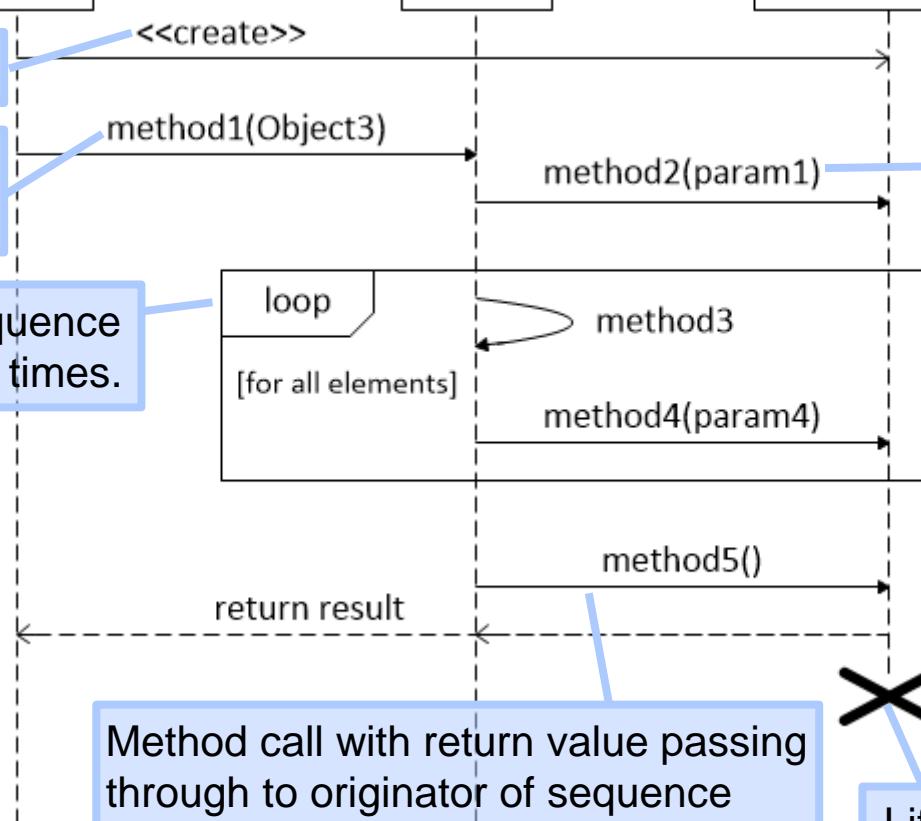
: Class2

Object3 : Class3

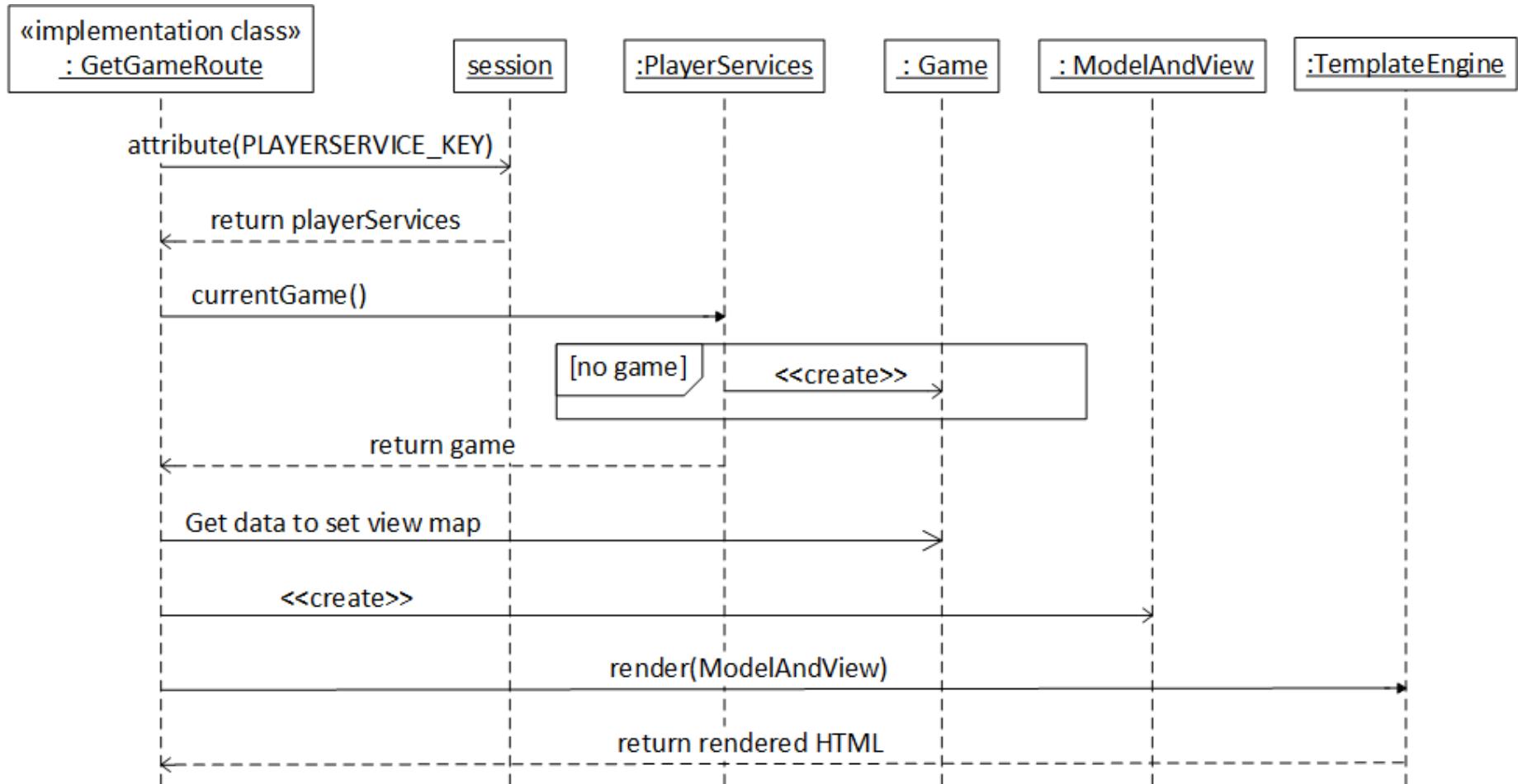
Object creation

What design principle does this show?

Execute sequence a number of times.



# This is a sequence diagram for getting a game page in the Guessing Game sample webapp.



# Code Metrics

class	Cyclic	Dcy	Dcy*	Dpt	Dpt*
rx.operators.OperationOnExceptionResumeNextViaObservableTest.TestObse...	0	4	7	1	1
rx.operators.OperationTimeout.Timeout	216	14	1610	1	398
rx.operators.OperationSingle	216	10	1610	1	398
rx.operators.OperationSkip.Skip.ItemObserver	216	3	1610	1	398
rx.operators.OperationTakeUntil.OtherObservable	216	8	1610	1	398
rx.operators.OperationTimeout.TimeoutSelector	216	13	1610	1	398
rx.schedulers.TestScheduler.CompareActionsByTime	0	1	7	1	33
rx.schedulers.TestScheduler.InnerTestScheduler	1	5	13	1	32
rx.android.observables.AndroidObservable	0	4	1631	1	1
rx.observers.SynchronizedObserverTest.CompletionThread	0	2	3	1	1
rx.operators.OperationConditionals	0	12	1625	1	2
rx.operators.OperationSkipWhile.SkipWhile.SkipWhileObserver	216	4	1610	1	398
rx.operators.OperationSample.SampleWithObservable.ResultManager.Sampler	216	3	1610	1	398
rx.apache.http.consumers.ResponseConsumerBasic	0	8	1615	1	3

## SWEN-261 Introduction to Software Engineering

Department of Software Engineering  
Rochester Institute of Technology



# A metric is not just a number.

- A metric is a quantitative function that calculates some characteristic and produces a numeric measurement which will be used to make a decision.
- For software product development, metrics fall into three broad categories
  - *Process – measurements of the software process that apply across projects*
  - *Project – measurements of one project team's activities*
  - *Product – measurements of the resulting software product*



# Software product metrics fall into multiple categories that look at different characteristics.

- Complexity
  - *Lines of Code is the most familiar*
  - *Cyclomatic Complexity*
- Coupling and Dependency
  - *Robert Martin Package Metrics*
- Counting/averaging lots of things that can be counted/measured
  - *Average lines per method*
  - *Average parameters per method*
  - *Average number of methods per class*
- Some metrics will apply at multiple levels, such as project, package, class, or method



# Even though you can count something, it does not necessarily count for anything.

- A metric is only as good as the decisions that it will be used to make.
- Measuring something without it having a solid connection to possibly improving what you are doing is a waste of time and resources.
- Target values for measurements should be set based on a record of past measurements and resulting performance.
  - *Measurement not in some range → some project quality was poorer*
- Initially, measurements need to be made to find the correlations.



# A metric target is not absolute.

- A measurement falling outside of a target range is not an absolute indictment.
- Measurements that do not fall in the target range indicate a place for additional scrutiny.
  - *For product metrics, they indicate possible "code smells".*
  - *Places to consider for refactoring, redesign, or reimplementations*

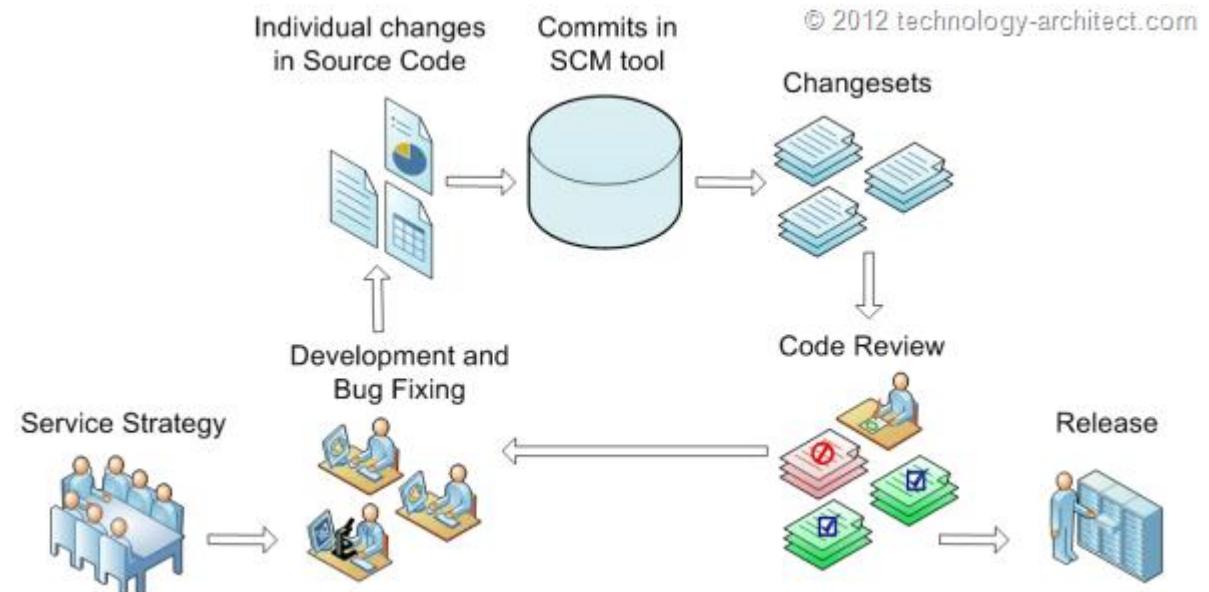


# These are some of the more popular metrics for object-oriented software systems.

- Cyclomatic complexity
  - *Count of execution paths through a method*
- Chidamber and Kemerer
  - *Coupling between object classes*
  - *Lack of cohesion in methods*
- Martin Package Metrics
  - *Fan-out coupling – classes need something outside package*
  - *Fan-in coupling – classes outside package use something inside package*
  - *Instability – ratio of fan-out to fan-out + fan-in*
  - *Abstractness – ratio of abstract classes and interfaces to total number in package*



# Code Review



**SWEN-261**  
**Introduction to Software**  
**Engineering**

Department of Software Engineering  
Rochester Institute of Technology



# A code review can improve the quality of the product and the quality of the team.

- Increase product quality
  - *Identify and fix design or coding violations*
  - *Identify and fix code communication issues*
  - *Analyze test coverage, identify new test scenarios*
- Increase overall team skill
  - *Discuss code communication*
  - *Share coding and testing techniques*
  - *Discuss design principles & patterns, as appropriate*



# There are several situations that warrant a code review.

- For new members of the team
  - *Along with reading the Design documentation*
  - *Code review (walk-through) with a senior developer*
- For Spikes
  - *To impart lessons from the Spike to the rest of the team*
- For User Stories
  - *To improve the quality of the feature code*
  - *To share best practices with the rest of the team*
  - *Even trivial stories should have reviews*



# There are several code review techniques.

- Individual
  - *A senior developer sits with a junior developer*
  - *The review can be focused on a specific problem or for general understanding a subsystem*
- Synchronous
  - *A team meets to review some code*
  - *Usually the most formal process*
  - *Disadvantage of needing to sync schedules*
- Asynchronous
  - *A developer uses an online tool to create a review*
  - *Shows the diffs between two branches*
  - *Reviewers make comments in the tool*
- Hybrid approaches



# A team will often have a checklist of things to look for during the code review.

- Coding practices
  - *Code communication*
  - *Defensive programming practices*
- Design practices
  - *Adherence to architectural tiers*
  - *Adherence to core OO principles*
  - *Adherence to OO design principles*
- Testing practices
  - *Are test suites comprehensive (enough)*
  - *Test code follows good code and design practices*
- Design documentation
  - *Is the documentation being kept up-to-date*

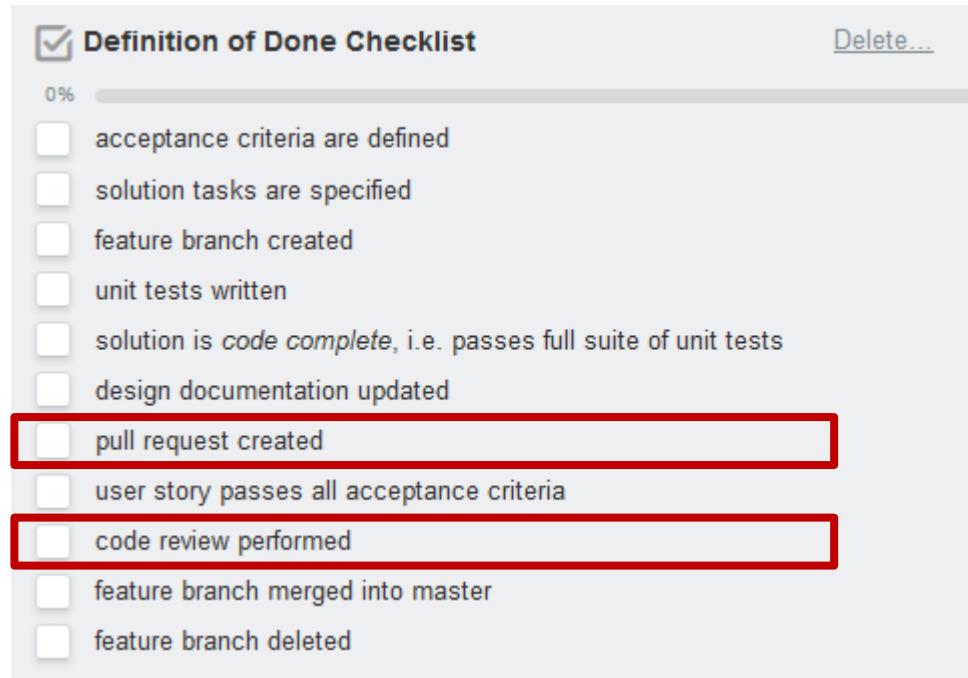


# The activity will guide the team through doing an asynchronous review.

- You will create a git *pull request* for a selected feature branch.
- Team members will review the code using GitHub's PR review user interface.
  - *We'll provide a checklist and document to record your suggested changes*
  - *Team submits the document to a Dropbox*
- After the changes are approved, the feature branch is merged into master



# Issuing pull requests and performing code reviews will now be a part of your development workflow.



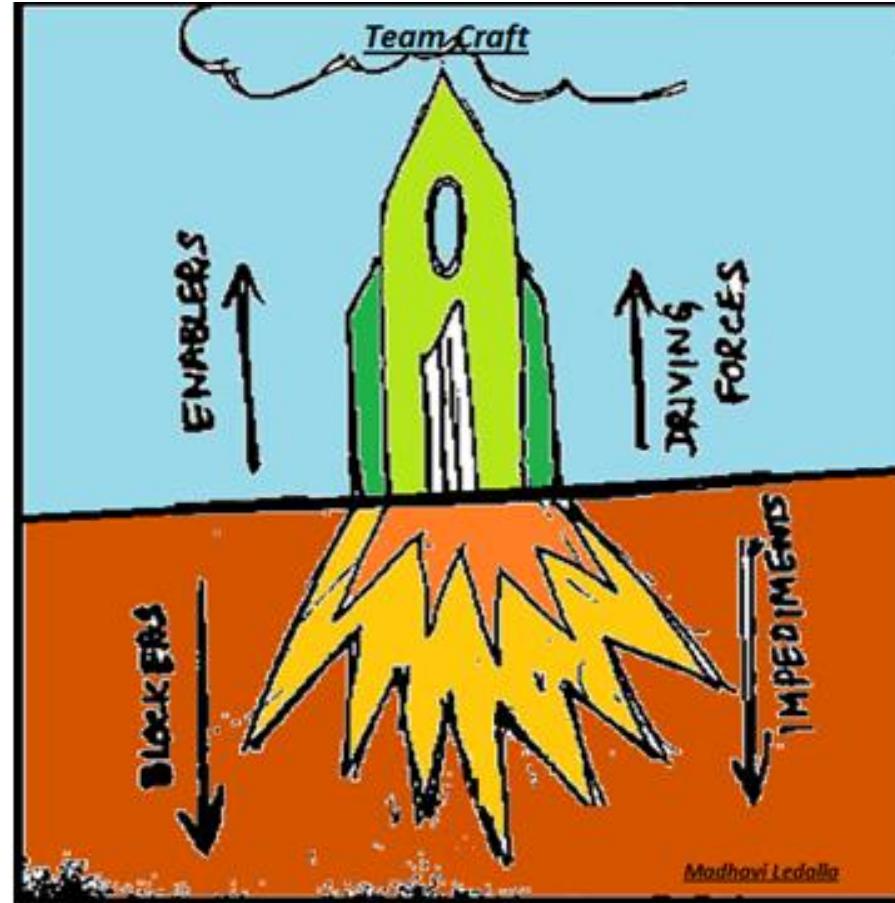
- The *Pull Request* is made when the story moves to *Ready for Test*, i.e. after the user story is code complete, and the design documentation is updated.
- Review should be done by a minimum of two team members other than the developer of the story.
- Acceptance testing can be performed in parallel.



# Sprint Retrospective

**SWEN-261**  
**Introduction to Software**  
**Engineering**

Department of Software Engineering  
Rochester Institute of Technology



From [7 Ways to Make Retrospectives Fun and Engaging](#) by Vibhu Srinivasan



# Team retrospectives are all about improving the team and your process.

- The team reflects (introspects) on three main questions:
  - *What went well?*
  - *What didn't go well?*
  - *What can we do to improve?*
- Do a retrospective on each sprint.
  - *This allows a team to make frequent course corrections (aka improvements)*
  - *Iterative and incremental is a principle to be applied to your process (as well as the product)*
- There are dozens of specific retrospective techniques; we'll teach you one.



# The starfish technique uses five categories of issues.

- Keep doing
  - *These issues highlight an activity that worked well*
  - *No change necessary*
- More of
  - *These issues request more of an activity*
- Start doing
  - *These issues request the start of a new activity*
- Less of
  - *These issues request less of an activity*
- Stop doing
  - *These issues request stopping an activity that isn't serving the team, the product or the stakeholders*



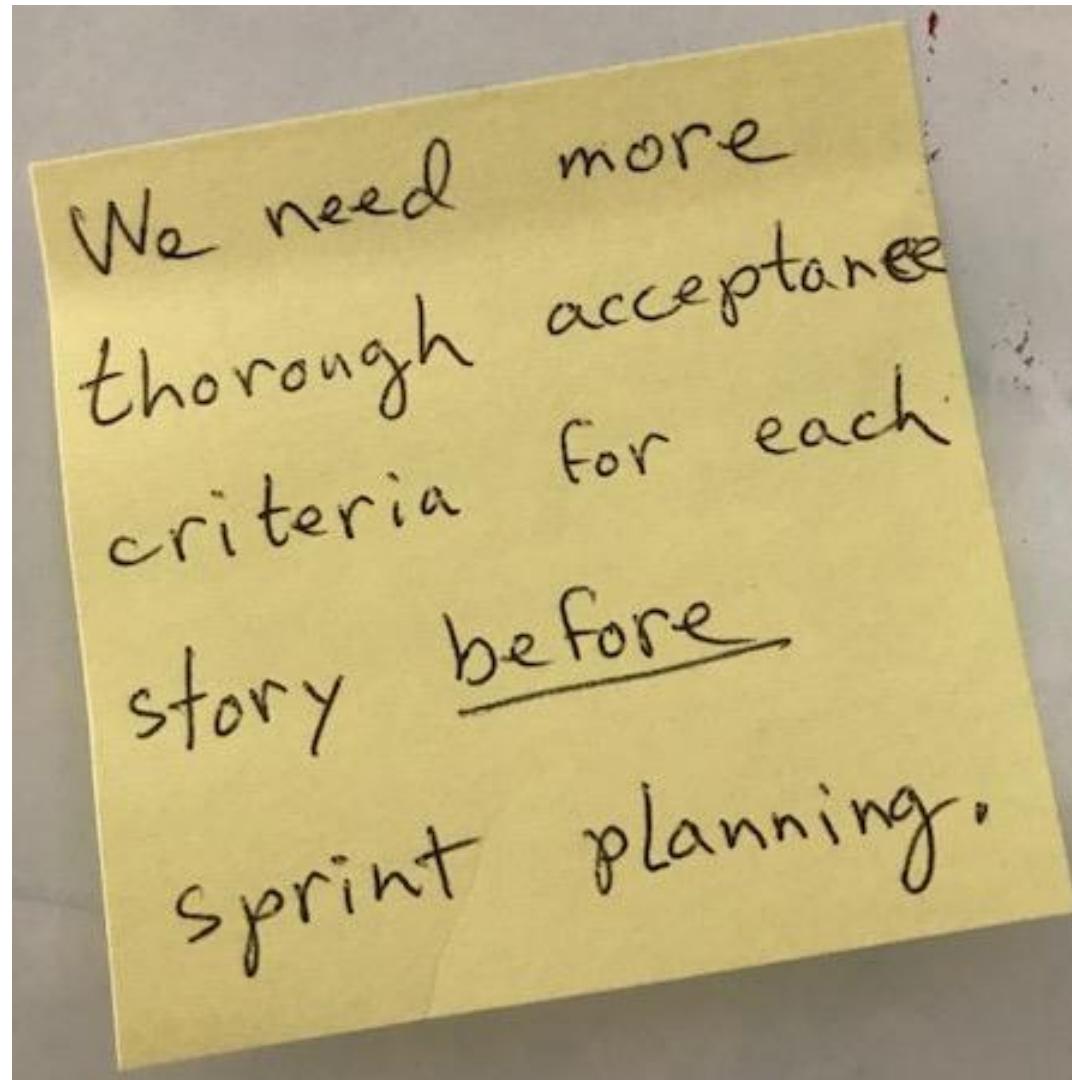
# The process of a retrospective follows these steps.

1. Every member creates issue cards
2. Members place each card on the starfish chart
3. A facilitator reads aloud each issue and groups common issues together
4. Every member votes for five issues on the chart
5. The facilitator picks top three issues
6. The team brainstorms on solutions to each of these top issues
7. The team creates action items for the next sprint to satisfy the top issues



# Keep your issues concise yet complete.

- Here's an example:



# Issues run the gamut from process, design, teamwork and communication.

- Adding, removing or improving processes:
  - *backlog refinement*
  - *sprint planning*
  - *calculating team capacity (velocity)*
  - *daily standups*
  - *sprint review/demo*
  - *sprint retrospective (the meta process)*
- Adding, removing or improving reviews:
  - *code reviews*
  - *reviewer engagement*
  - *pre-development design reviews*
  - *pre-planning design reviews*



**Issues run the gamut from process, design, teamwork and communication.**

- Adding, removing or improving testing:
  - *unit testing*
  - *unit testing code coverage*
  - *integration testing (esp for web apps)*
  - *acceptance (manual) testing*
- Adding or improving developer or team skills:
  - *better use of Spikes*
  - *addition of tech talks*
  - *formal member training*
- Adding or improving team communication:
  - *more or better use of communication tools*
  - *more or better face-to-face meetings*
  - *more or better discussions with the Product Owner*
  - *better virtual (distributed) teaming practices*

# Professional Responsibility



**SWEN-261**  
**Introduction to Software**  
**Engineering**

Department of Software Engineering  
Rochester Institute of Technology



Software Engineering  
Rochester Institute  
of Technology

# Agile was created out of the need for professionalism in our industry.

*"These are disciplines, not process steps.*

*They are promises you make; they are not tasks to follow."*

-- "Uncle" Bob Martin

## ■ Process

- ***Individuals and interactions*** over processes and tools
- ***Working software*** over comprehensive documentation
- ***Customer collaboration*** over contract negotiations
- ***Responding to change*** over following a plan

## ■ Design

- ***Adhere to architecture and design principles***

## ■ Teamwork & Communication

- ***Meet commitments to team and Product Owner***



# Our industry grows when we all share in the ever growing knowledge.

- Always strive to learn new techniques, tools and methods.
- Become a team mentor.
  - *Teach by example in your code and designs*
  - *Teach by documentation*
  - *Teach team members and others face-to-face*



# Like any critical industry, you must consider ethical judgments.

- Do right by your client or employer.
- Do right by your team.
- Do right by society with the broad range of what that means for you.



# Ultimately, it all rests on individual actions.

- Make your actions count and do them so that you can be proud of the actions you have done.
  - *Care about your craft*
  - *Think about your work*
  - *Invest regularly in your knowledge portfolio*
  - *It's both what you say and the way you say it*
  - *Sign your work*
- Professionalism and discipline is a personal endeavor.



# Personal discipline also includes psychological and, for some, even spiritual concerns.

- Develop personal maturity
  - *Be respectful*
  - *Be kind*
  - *Be aware of both conscious and unconscious biases*
- Develop psychological discipline
  - *Be impeccable with your word*
  - *Don't take anything personally*
  - *Don't make assumptions*
  - *Be skeptical, but learn to listen*
- Always do your best!

(*Don Miguel Ruiz, The Fifth Agreement*)

