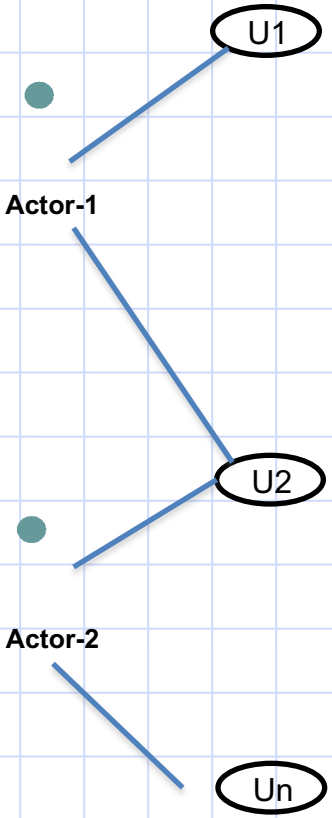
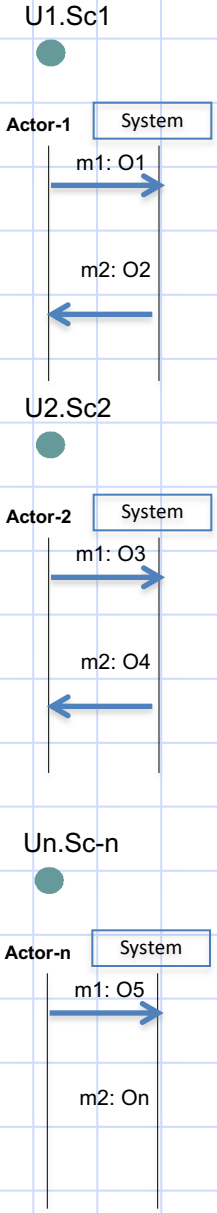
Summary of Guidelines to Develop Object-oriented Design and Architecture

Use-case Model | Behavior Model | Finding Domain Concepts | Domain/Conceptual Models

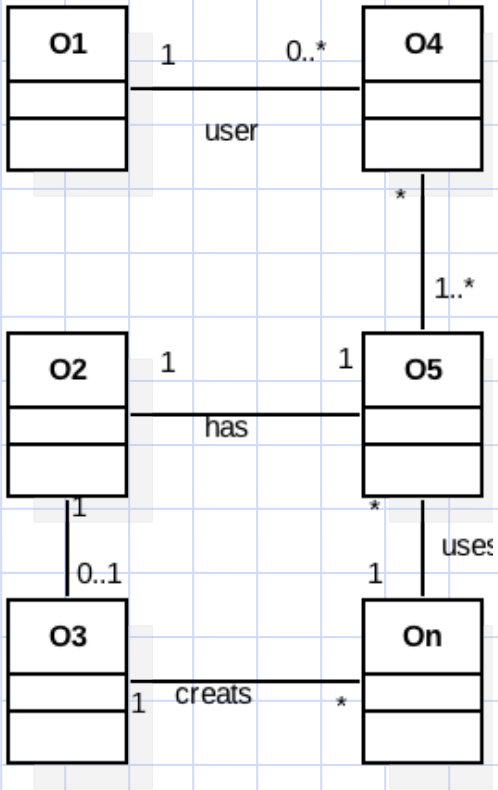




From behavior models. If behavior models are developed carefully with a domain expert, most of the domain concepts must have been explored.

At this stage we should extract all nouns from model. Then from list of extracted nouns we should select the concepts that are good candidates to be used in our conceptual/domain model.

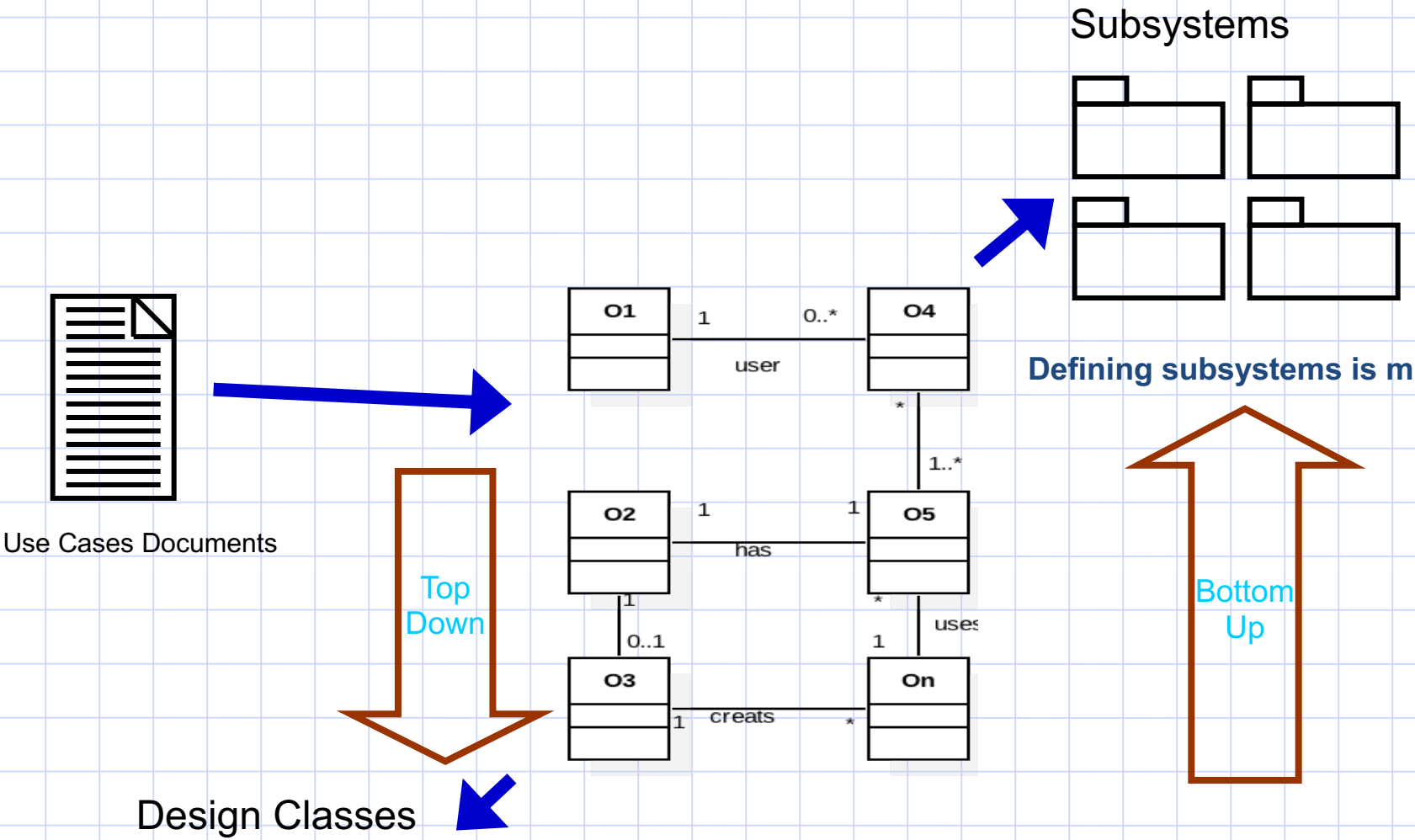
Good candidates are those concepts/nouns that have a structure and are not just simple nouns that can be implemented by int or string..



Steps in Analysis Phase

Object-Oriented Design Activities

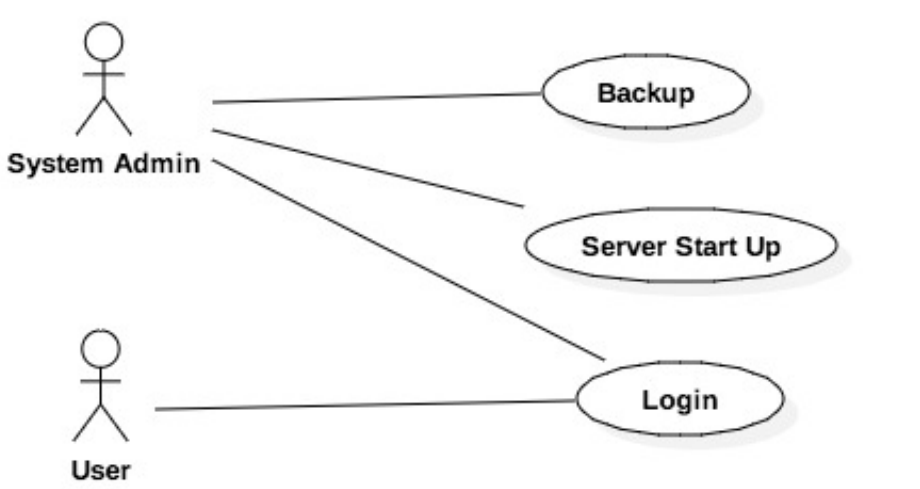
Design and Architecture is a Top-Down and Bottom-Up Process



Stepwise Design Activities

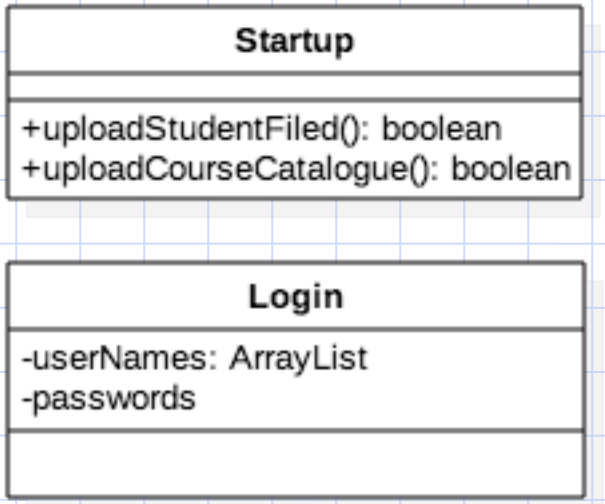
Step 1: Add Design Use Cases

- Add missing processes/use cases that are required for system maintenance, security, reliability, etc:
  - "START UP USE CASE".
  - Login
  - Logout
  - Backup



Step 2: Add New Classes

- Add new classes needed because of new uses cases added in step 1.
- Add new classes for each actor that its information must be saved. For example: Customer, Staff, etc.

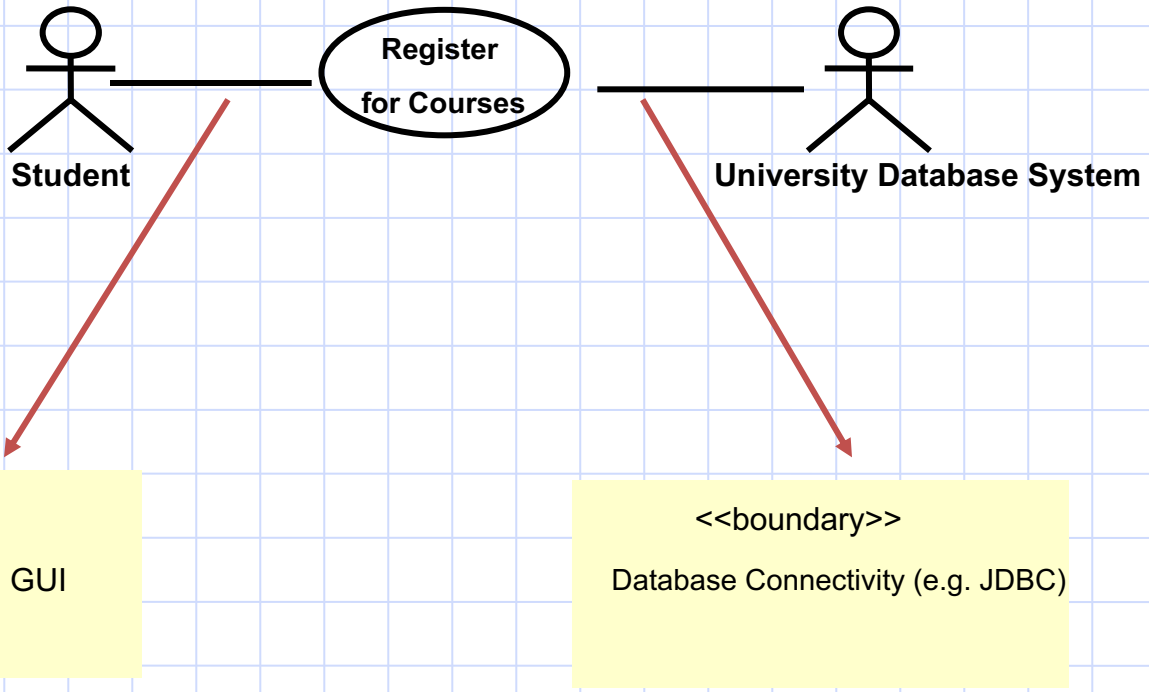


Step 3: Add Classes to Separate Concepts

- Add new classed to help designing a multi-tier architecture that allows separation of concepts:
  - Presentation Layer
  - Domain Layer
  - Controllers
  - Etc.
- This can be achieved by:
  - Adding boundary classes
  - Identifying or adding entity classes
  - Identifying or adding controller classes

Step 3.1: Add Boundary Classes?

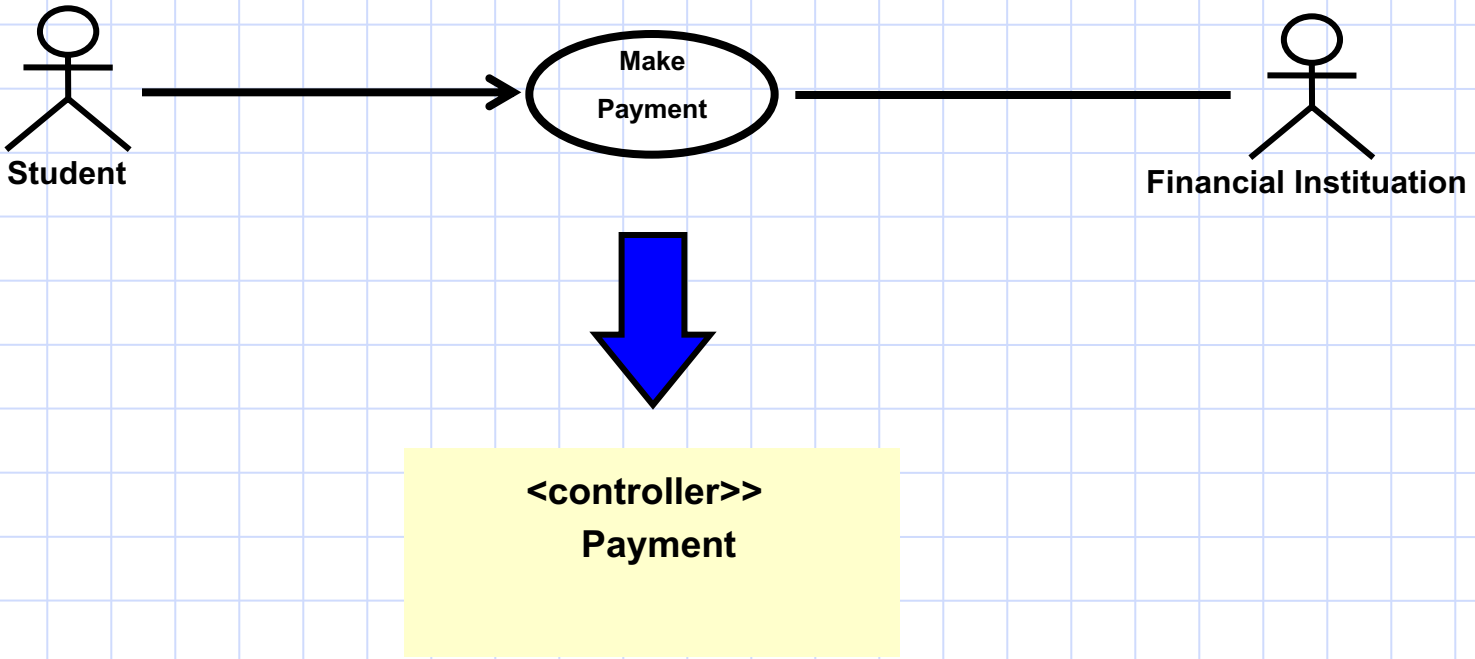
- Decide on Interfaces
  - Which association link between actor and a use case is a good candidate to be presented as boundary class
  - Should every line be considered as an interface



Step 3.2: Decide on Control Classes

Decide on Controllers:

- Start with considering each use case to be considered as a controller class:

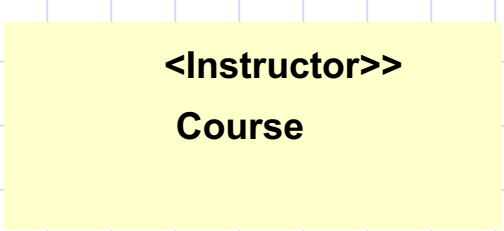


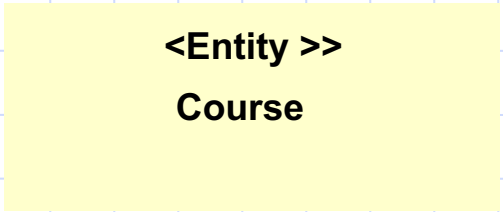
Step 3.2 Continued

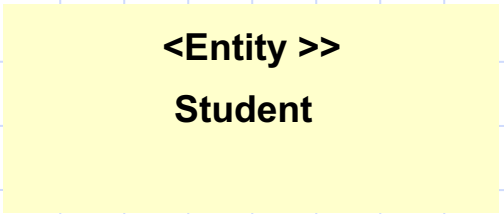
- If control classes seem to be just "pass-through" from the boundary classes to the entity classes, they may be eliminated.
- A use case is as a good candidate to be a Controller Class if:
  - encapsulate significant control flow behavior
  - its behavior is likely to change in future
  - its behavior must be distributed across multiple processes and/or processors
  - its behavior requires some transaction management.

Step 3.3: Decide on Entity Classes

- Only business objects that need to be persistent and accessed by the system need to be modeled as an entity classes.
- If the entity is relatively short-lived and non-persistent don't consider it as an entity class. Example: a shopping cart compared to a sale-item in a Point of Sale System.
- Examples of entity classes in a University Registration System

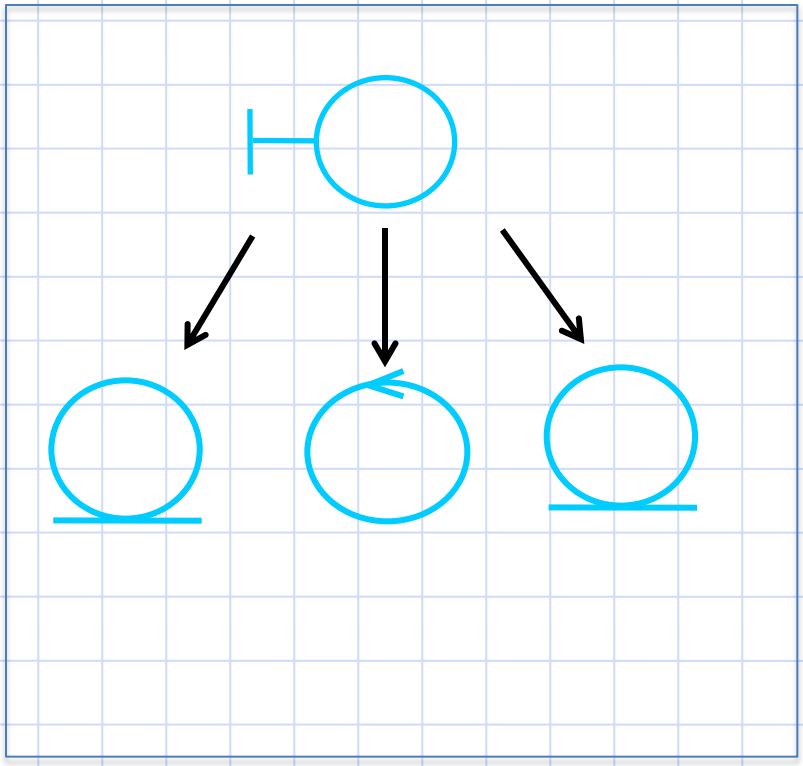


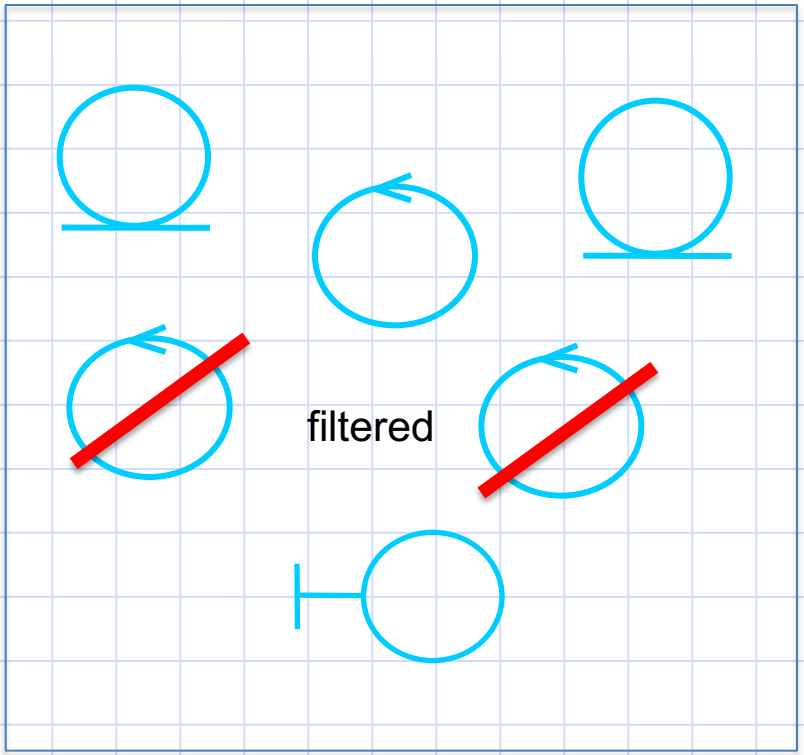




Step 4: Unify or decompose analysis classes:

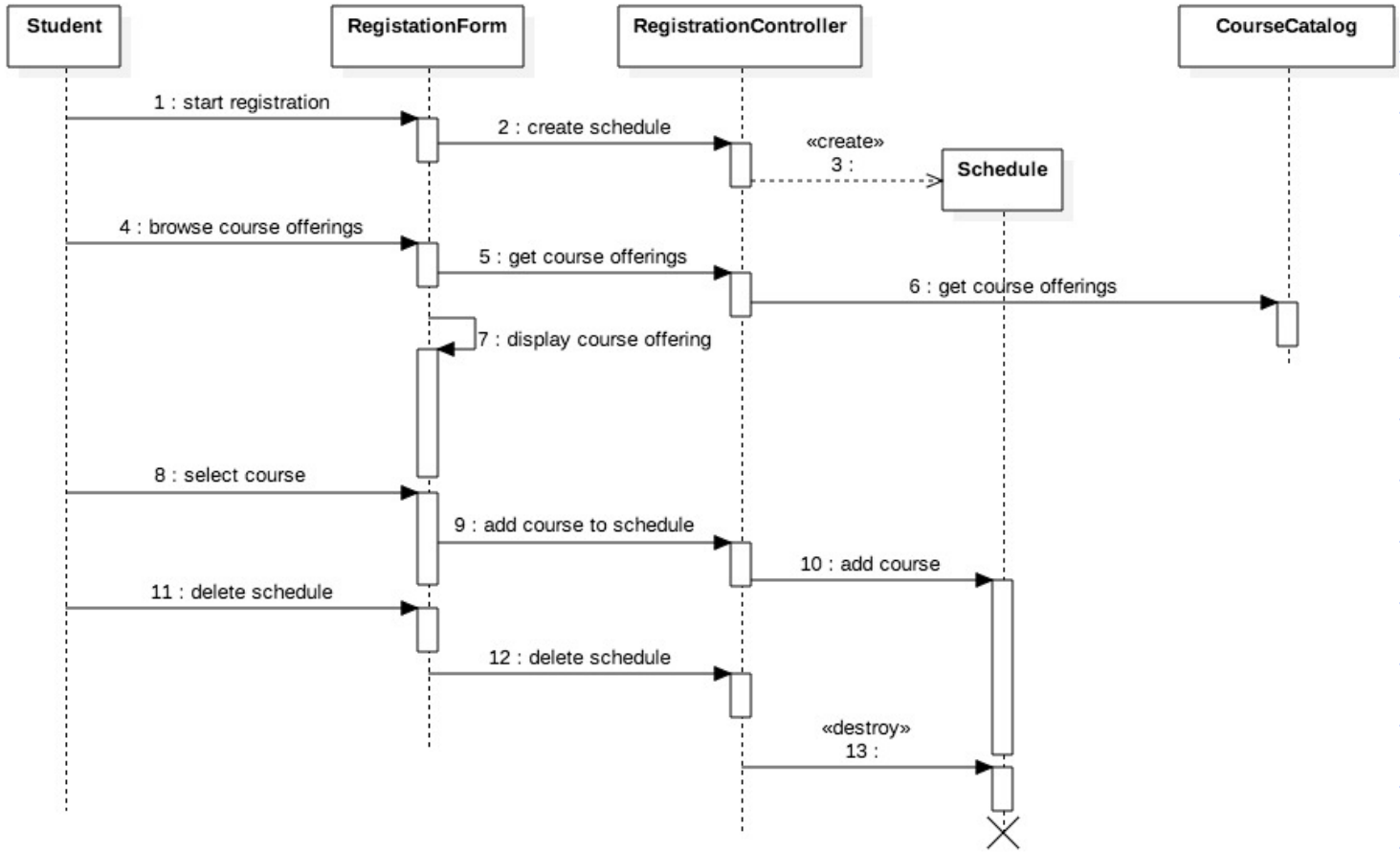
- Some classes in the analysis stage may need to be divided to more classes. Or, some analysis classes need to be filtered to ensure that each analysis class represents a single well-defined concept, with no overlapping responsibilities.





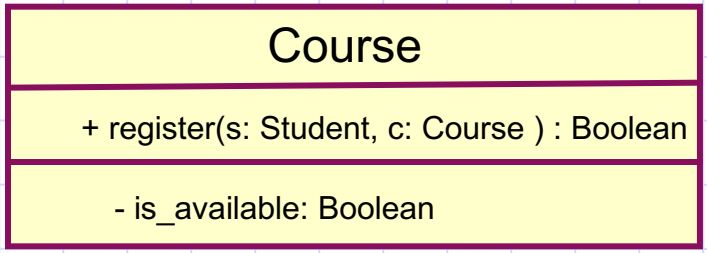
Step 5: Discover Class Behaviors

- Develop detail sequence diagrams as needed to discover major operations need between object. A good approach is to develop a detailed sequence diagram for at least a few major use cases that show how the majority of the domain objects are involved.



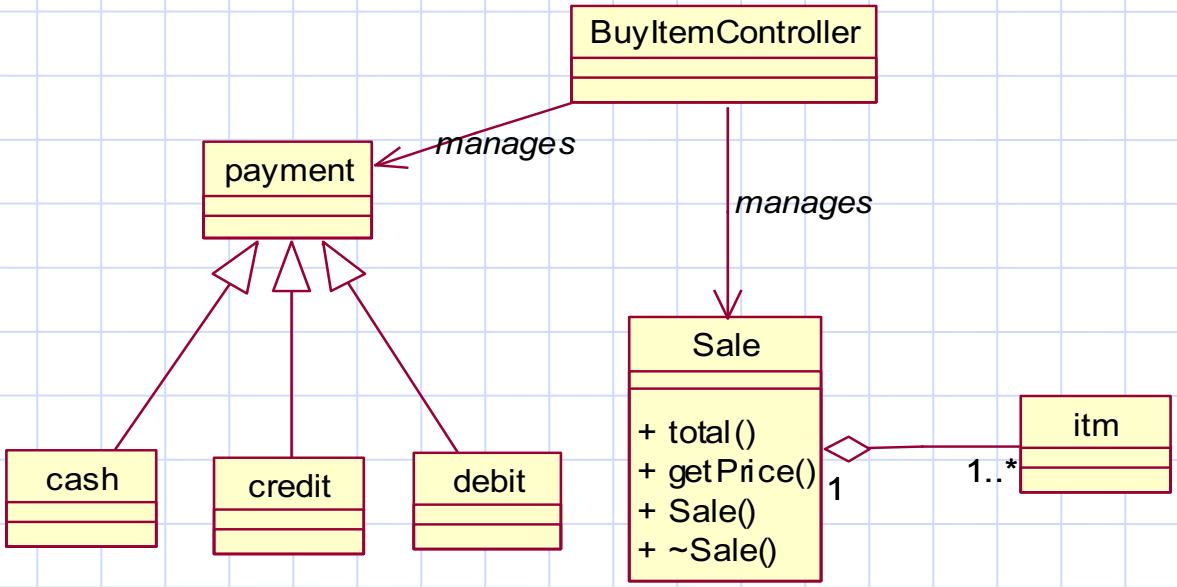
Step 6: Add Attribute and Operations to Classes

- Add operation signatures, and Attributes using proper UML syntax
  - Operation Name (parameter : class...): returnType
    - return values and types, parameters and their types.
    - *Don't worry about access* methods such as get, set etc.



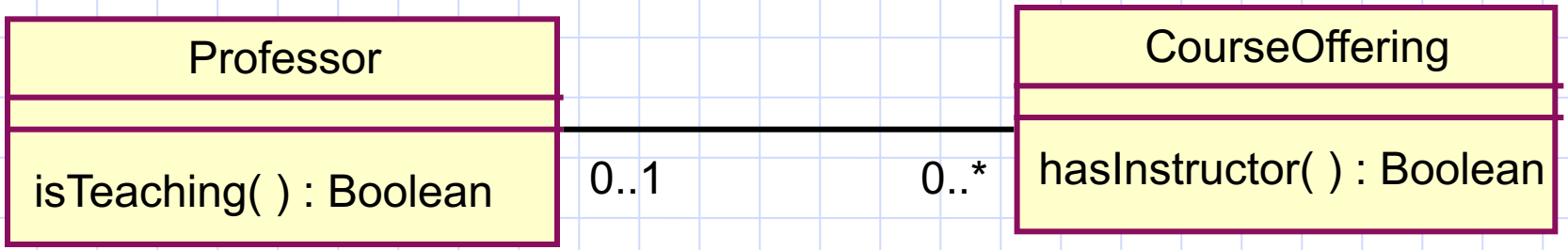
Step 7: Consider Advanced Class Relationships

- Consider generalization-specialization relationship exists between classes.
- Consider whole-part relationships as aggregation or composition.
- Don't forget to indicate multiplicity and optionality when applies:



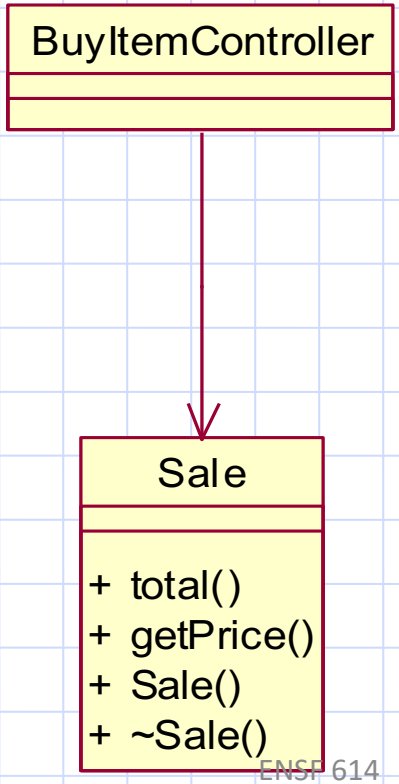
Step 8: Multiplicity Design: Optionality

- If a link is optional, make sure to include an operation to test for the existence of the link
- Example:
  - if a Professor can be on sabbatical, a suitable operation should be included in the Professor class to test for the existence of the CourseOffering link.



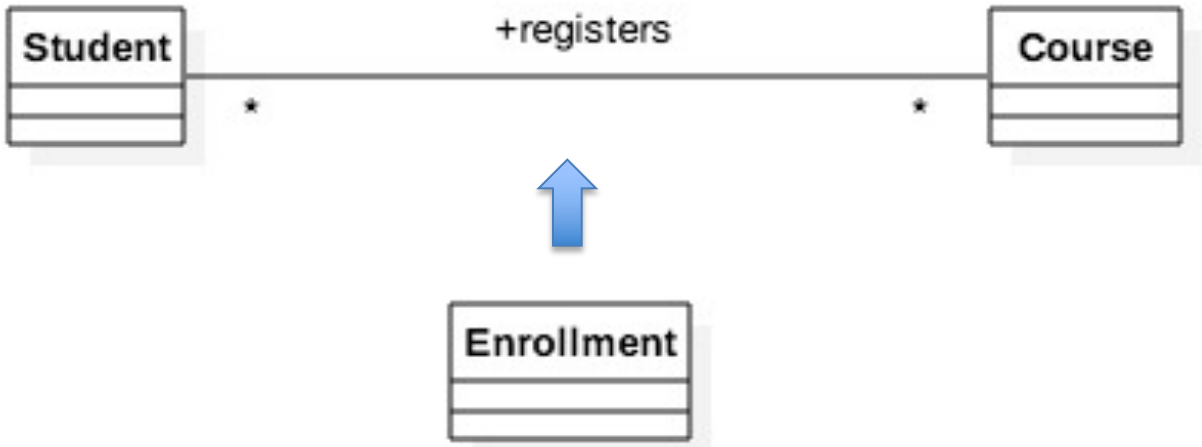
Step 9: Add Navigability and Dependencies

- If needed, you should add navigability to express the dependency of one class over another class. For example, Class BuyItemController needs to know Sale, but Sales doesn't need to know its controller class:



Step 10: Association Class

- A class is "attached" to an association
- Contains properties of a relationship
- One instance per link



In many cases, association classes are used to resolve many-to-many relationships.

Step 11: Consider Design Patterns

- If applicable consider using "Design Patterns". Most common patterns include:
  - Strategy Pattern
  - Observer Pattern
  - Decorator Pattern
  - Singleton Pattern
  - Bridge Pattern

Step 12: Add Tactical Design Decisions

- Add classes to the model to reflect decisions made concerning the key mechanisms of the system (decision regarding common standards, policies, and practices). This is always referred as ***tactical design decisions.*** Such as decisions on:
  - » Look and feel of user interface
  - » Exception handling
  - » Networking
  - » Etc.

Moving to Higher Level Architecture

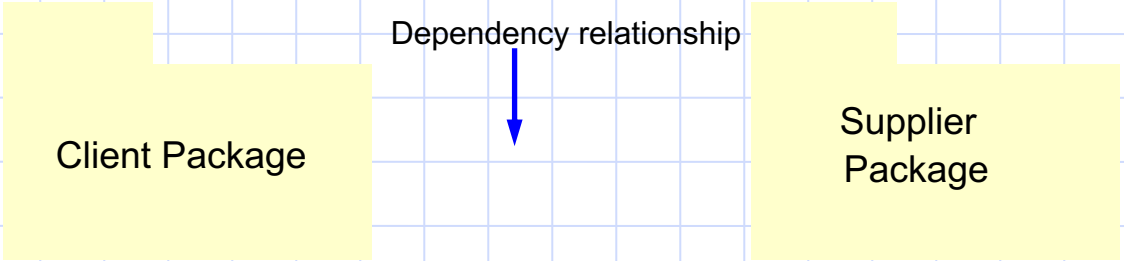
What is Good Architectural Design

- The attributes of good architectures are:
  - Good architectures are constructed in well-defined layers of abstraction
  - There is a clear separation between the interface and implementation of each layer
  - The architecture is simple

What is Package Diagram?

UML Packages:

- Formal Definition:
  - **Package** is a **namespace** used to group together elements that are semantically related and might change together
  - Package Diagram:
  - Is another **UML structure diagram** which shows **packages** and dependencies between the packages.
- Dependency Implications
  - The Client package cannot be reused independently because it depends on the Supplier package



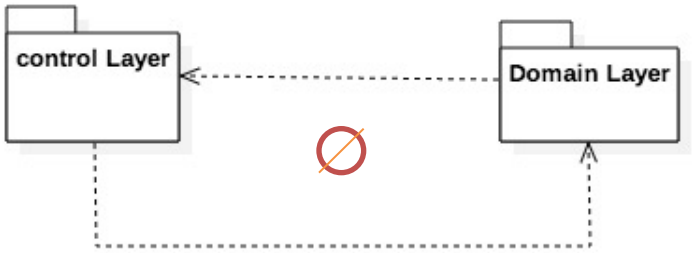
Layering Considerations

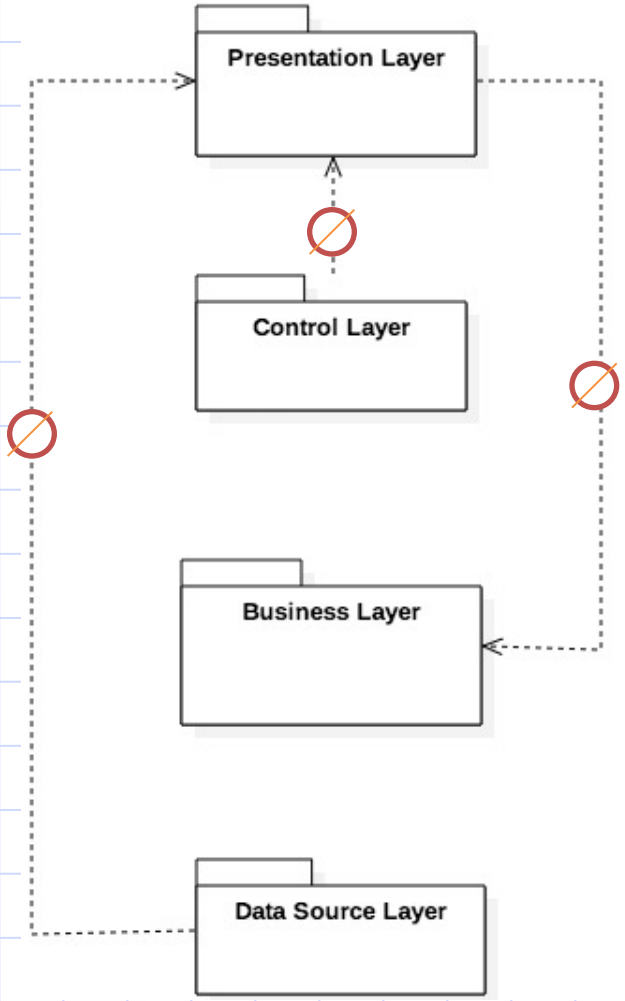
- **A few recommendation**
  - Visibility
    - If possible, the dependencies should be only within current layer and the next lower layer
  - Volatility
    - Upper layers most likely will be more affected by requirements changes
    - Lower layers most likely will be affected by systems environment changes (hardware, OS, other systems such as database, etc.)
  - Generality
    - The elements of the system that are more abstract should reside in lower layers. Example frameworks
  - Number of layers:

    The number of layer depends on how complex the underdevelopment system is or how it is subject to future changes. Systems that are more complex or more subject to future changes must have more separation of layers.
  - Layers must access each other only by dependency on a public class.

More on Layering Considerations

- Packages should not be cross-coupled
- Packages in lower layers should not be dependent upon packages in upper layers
- In general, dependencies should not skip layers



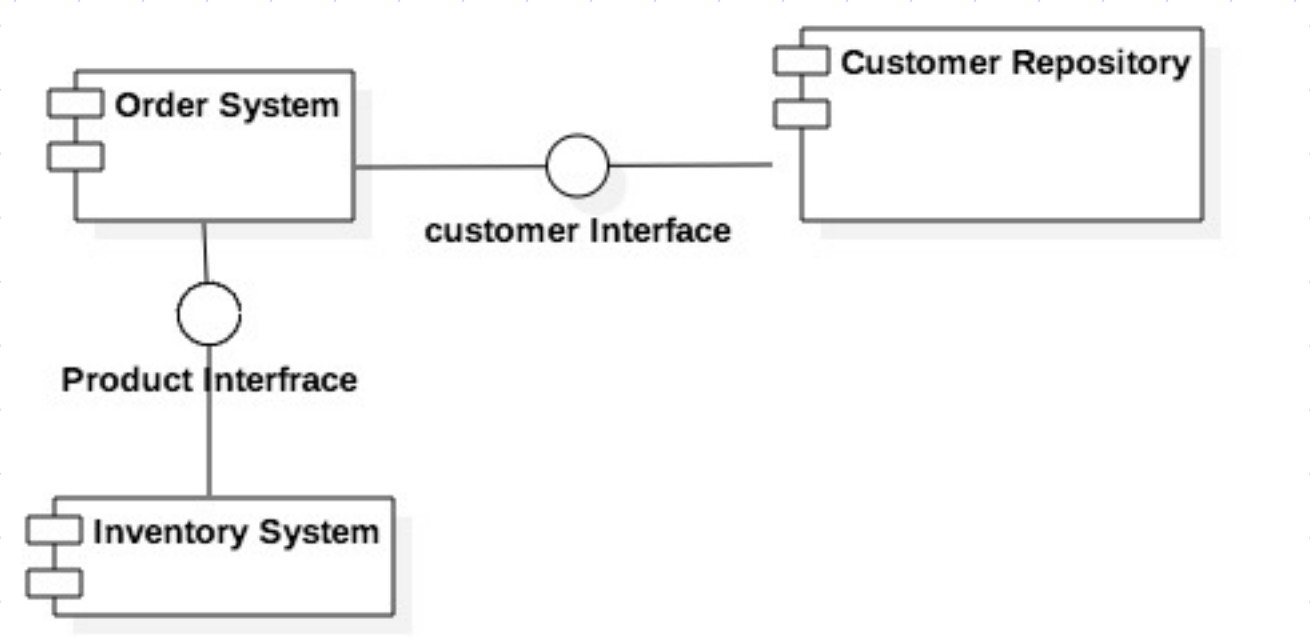


What is a Component Diagram

What is Component Diagram

- A component diagram shows relationship between different components in a system.
- Commonly used for Component-Based Development (CBD). Examples of Component Models: EJB (Enterprise JavaBeans), COM (Component Object Model), CORBA (Common Object Request Broker Architecture.
- But in general can be used to represent:
  - A module of classes that execute a specific task and normally should be easily replaced with different module.
  - A subsystem that perform a specific task.
  - An independent executable

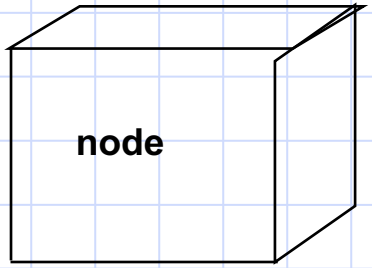
Example


Order System
Customer Repository
customer Interface
Product Interfrace
Inventory System

What is Deployment Diagram?

Notation for Deployment Diagrams

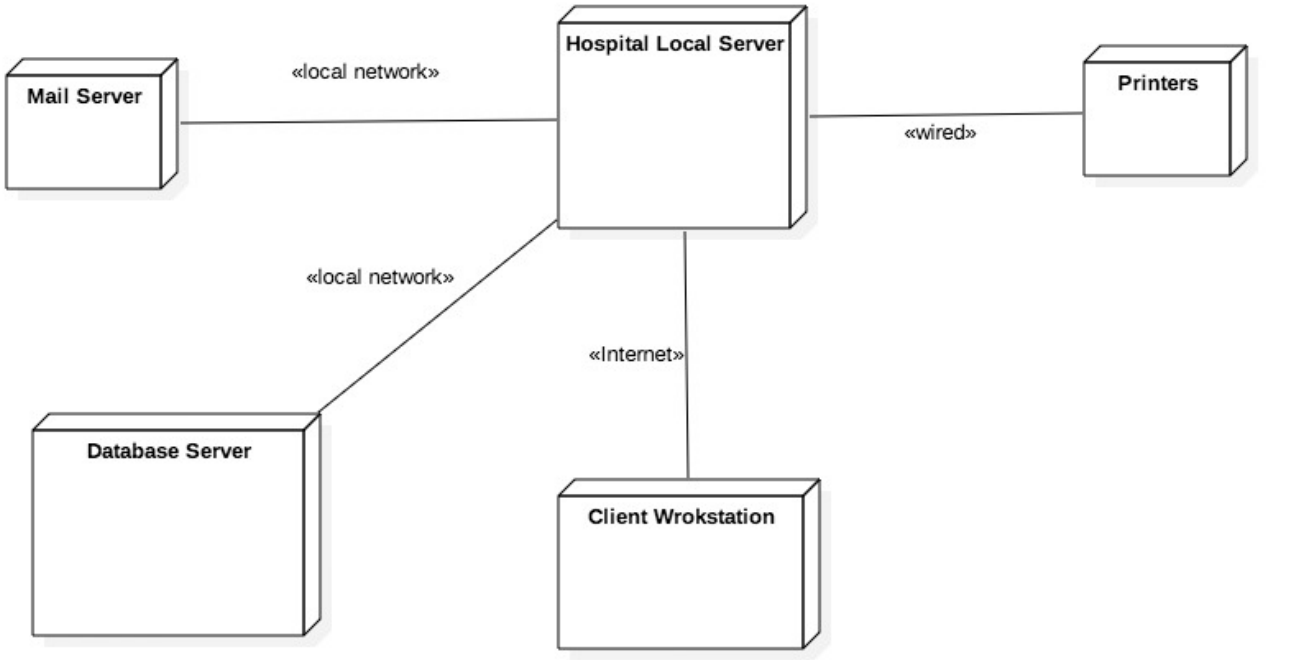
- A **node** is a run-time physical object representing computational resources
- A **connection** indicates communication, usually by means of direct hardware coupling



connection

Example: Hospital System Deployment Diagram

- This diagram shows nodes and the devices within a Hospital System that communicate with each other:



Summary: Architectural Design

- The logical view of architecture addresses the functional requirements of the system
  - Class diagrams which represent the key abstractions of the system
- The process view of architecture addresses the system's availability, reliability, scalability, integrity, performance, system management, and synchronization
  - System is decomposed into a set of independent tasks which are grouped into processes
- The development view of architecture addresses the software organization of the system
  - Component diagrams are created to show the packages and components that make up the system.
- The physical view of architecture maps software to processing nodes
  - Deployment diagrams are created to show the different processors and devices in the system
- Scenarios demonstrate and validate the logical, process, development, and physical views of architecture