Chapter 10

Agile Architecture

This Chapter Covers

- Architecture techniques and concepts
- Change cases
- Component diagrams
- Deployment diagrams
- Free form diagrams
- Network diagrams
- Package diagrams
- Layering your architecture

Architecture Techniques & Concepts

- Put architectural decisions off as long as possible;
- Accept that some architectural decisions are already made;
- Prove it with code;
  - Spike solution in XP
  - Technical prototype or skeleton in RUP
- Set an architectural change strategy;
  - Architect for change
  - Trust that you can meet tomorrow's needs tomorrow
- Consider reuse;
- Roll up your sleeves;
- Be prepared to make trade-offs;
- Consider adopting the Zachman framework;
- Apply architectural patterns gently.

Consider Reuse

- Will you build something from scratch?
- Will you adopt open source software (OSS) solutions?
- Will you start an OSS project to share resources with your peers?
- Will you reuse an existing resource as is?
- Will you modify an existing resource to meet your unique needs?
- Will you take advantage of a common business architecture framework?
- Will you develop common business architecture and then reuse it?
- Will you reuse, or develop for reuse, common technical infrastructure functionality?
  - Data sharing
  - File management
  - Interprocess Communication (IPC)
  - Persistence
  - Printing
  - Security
  - System management
  - Transaction management

Zachman Framework

	Structure (What)	Activities (How)	Locations (Where)	People (Who)	Time (When)	Motivation (Why)
Objectives/ Scope (Planner's View)	Most significant business concepts	Mission	International view of where organization operates	Human resource philosophies and strategies	Annual planning	Enterprise vision
Enterprise Model (Business Owner's View)	Business language used	Strategies and high-level business processes	Offices and relationships between them	Positions and relationships between positions	Business events	Goals, objectives, business policies
Model of Fundamental Concepts (Architect's View)	Specific entities and relationships between them	Business functions and tactics	Roles played in each location and relationships between roles	Actual and potential interactions between people	System events	Detailed business rules
Technology Model (Designer's View)	System representation of entites and relationships	Program functions/ operations	Hardware, network, middleware	User interface design	System triggers	Business rule design
Detailed Representation (Builder's View)	Implementation strategy for entities and relationships	Implementation design of functions/ operations	Protocols, hardware components, deployed software items	Implementation of user interface	Implementation of system triggers	Implementation of business rules
Functioning System	Classes, components, tables, ...	Deployed functions/ operations	Deployed hardware, middleware, and software	Deployed user interface (including documentation)	Deployed systems	Deployed software

Computation Independent Models

Platform Independent Models (PIMs)

Platform Specific Models (PSMs)

Change Cases

- Change cases are used to describe new potential requirements for a system or modifications to existing requirements.
- Consist of:
  - Description
  - Likelihood
  - Impact
- Good questions to ask yourself:
  - How can the business change?
  - What is the long-term vision for our organization?
  - What technology can change?
  - What legislation can change?
  - What is your competition doing?
  - What systems will we need to interact with?
  - Who else might use the system and how?

Change Case Examples

Change case: Need to support both Linux and Microsoft platforms.
Likelihood: Very likely to happen for application and database servers within six months; medium probability that Linux will be adopted by the school for desktop machines.
Impact: Unknown. Currently application servers for other applications run Microsoft-based operating systems and will likely continue to do so. New servers will likely have Linux installed. Desktop impact is hard to quantify as the Linux market is evolving rapidly. Should be re-evaluated six months from now.

Change case: Need to support new database vendor.
Likelihood: Medium. The school is currently renegotiating the contract with our existing vendor.
Impact: Potentially large if SQL is hard-coded into the application.

Change case: The University will open a new campus.
Likelihood: Certain. It has been announced that a new campus will be opened in two years across town.
Impact: Large. Students will be able to register in classes at either campus. Some instructors will teach at both campuses. Some departments, such as Computer Science and Philosophy, are slated to move their entire programs to the new campus. The likelihood is great that most students will want to schedule courses at only one of the two campuses, so we will need to make this easy to support.

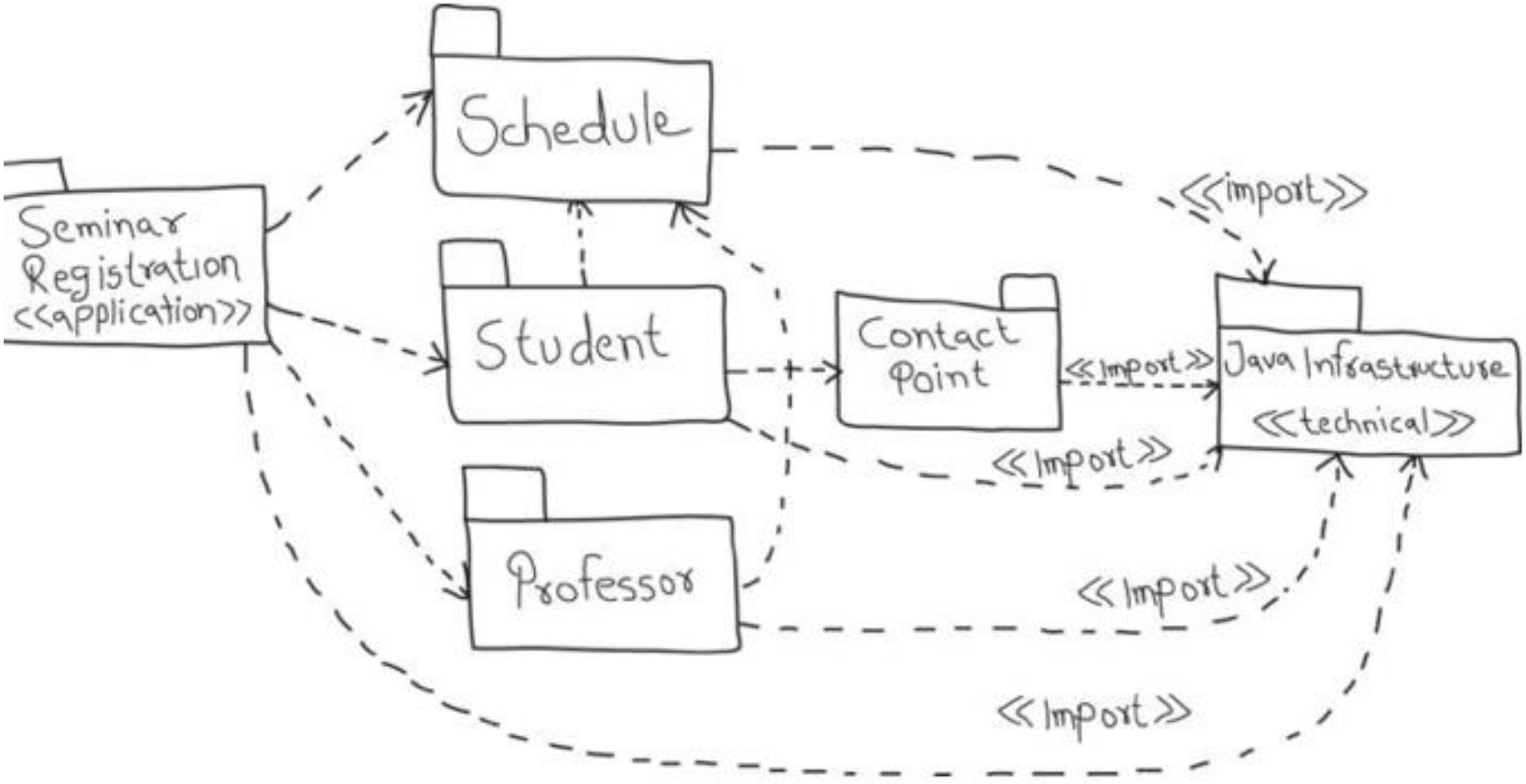
UML Package Diagrams

- Packages are UML constructs that enable you to organize model elements into groups, making your UML diagrams simpler and easier to understand.
- Packages are depicted as file folders and can be used on any of the UML diagrams.
- Most commonly used on use case diagrams and class diagrams because these models have a tendency to grow.
- Packages should be cohesive. A good test is you should be able to give your package a short descriptive name.

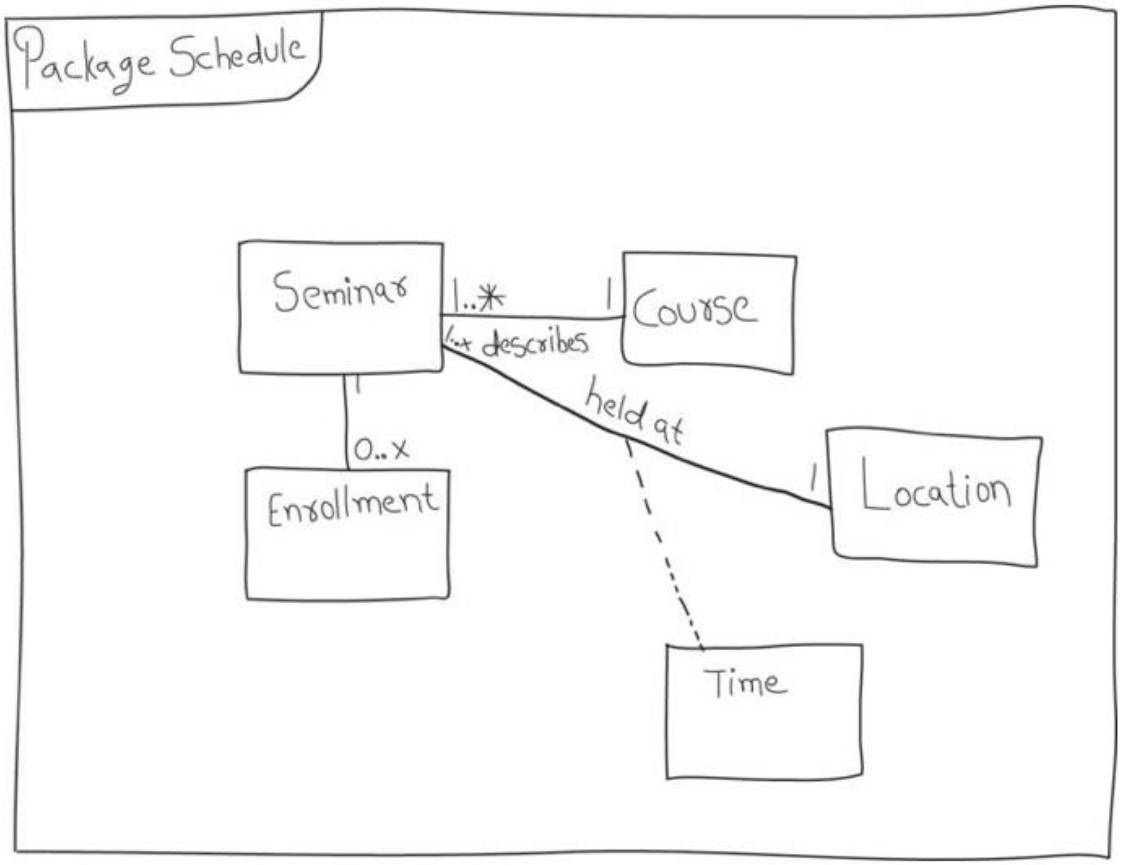
Class Package Diagrams

- First, classes in the same inheritance hierarchy typically belong in the same package.
- Second, classes related to one another via composition often belong in the same package.
- Third, classes that collaborate with each other—information reflected by your sequence diagrams (Chapter 11) and communication diagrams (Chapter 11)— often belong in the same package.

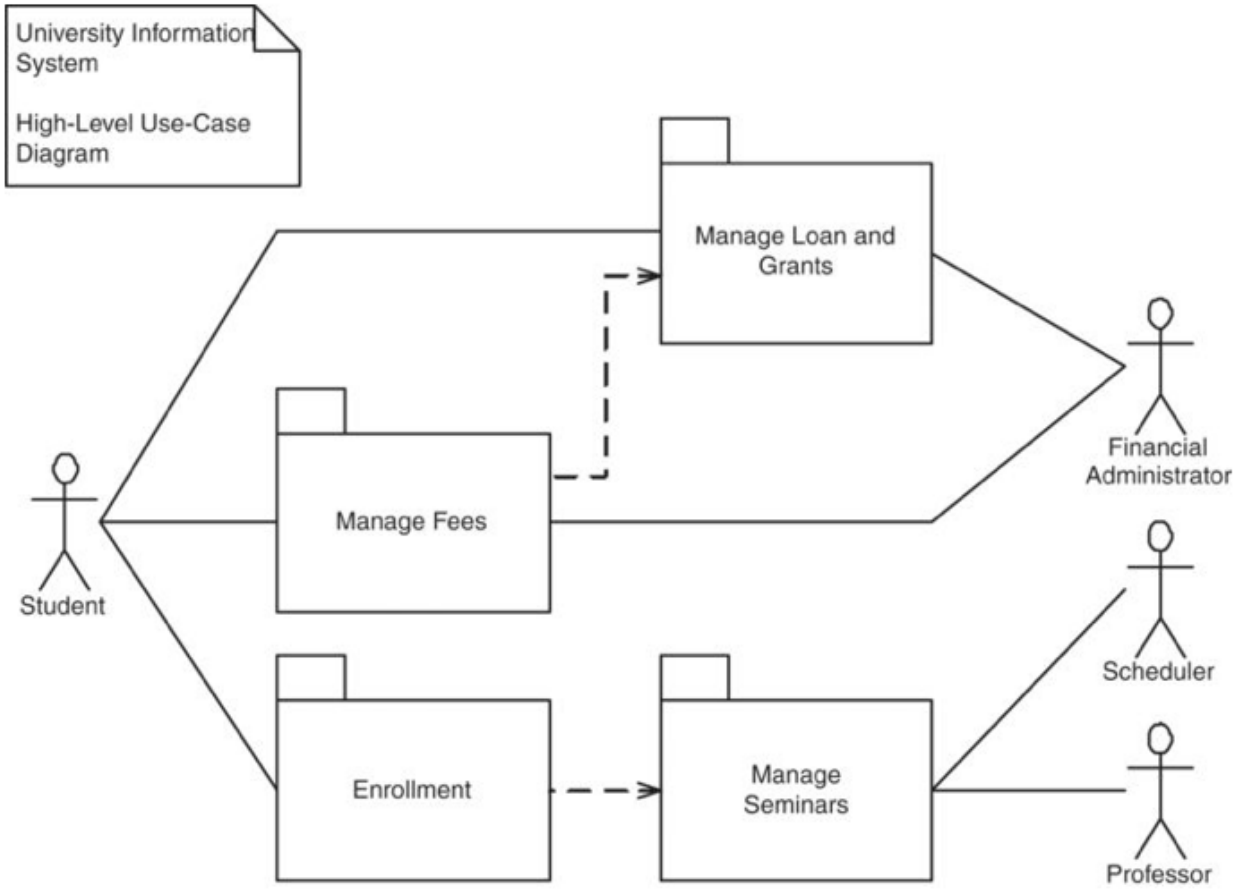
Class Package Example



Frame Example



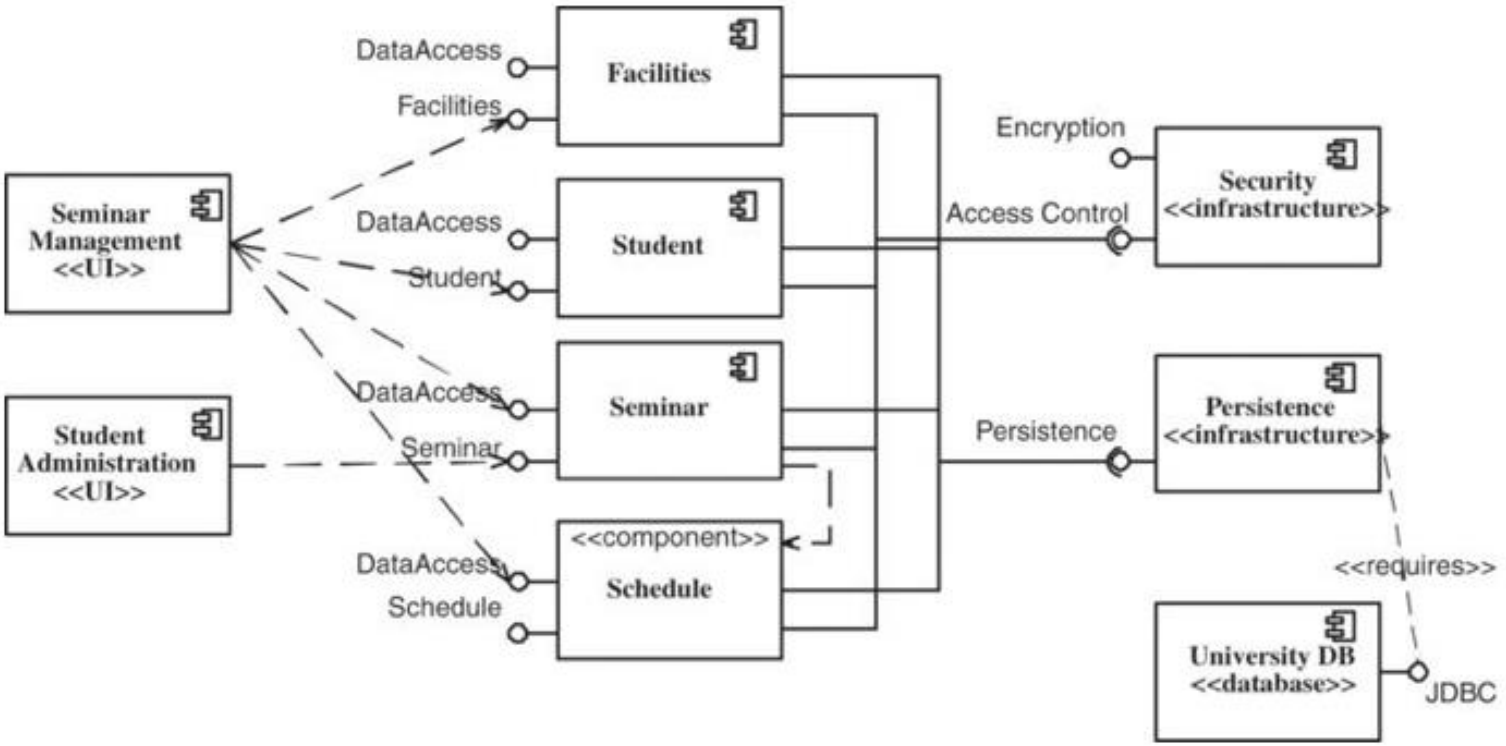
Use Case Package Diagram



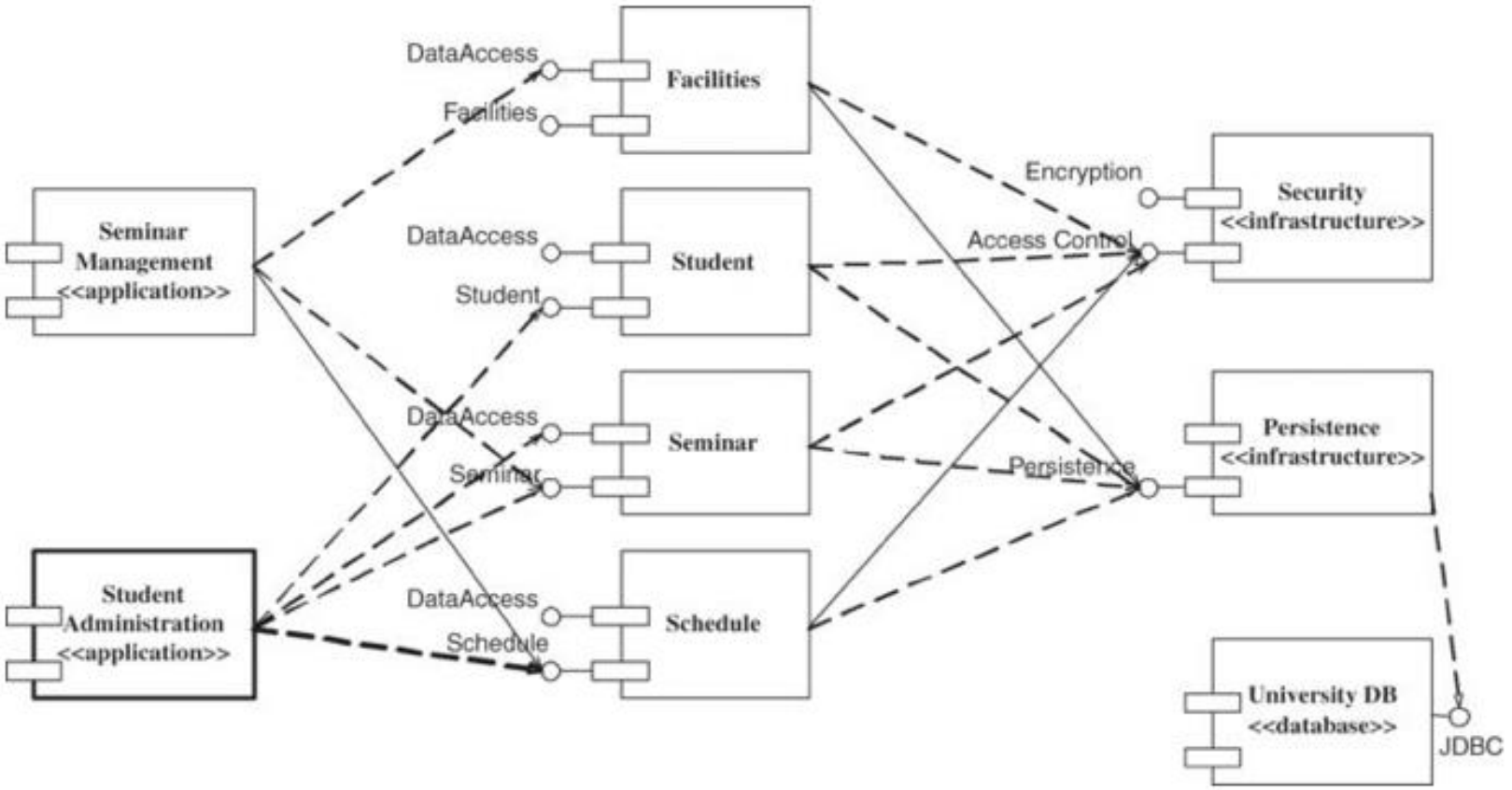
UML Component Diagrams

- Components may both provide and require interfaces.
- Often referred to as “wiring diagrams” because they show how the various software components are “wired together” to build your overall application.
- The lines between components are called connectors, the implication being that some sort of messaging will occur across them.

UML 2 Component Diagram Example



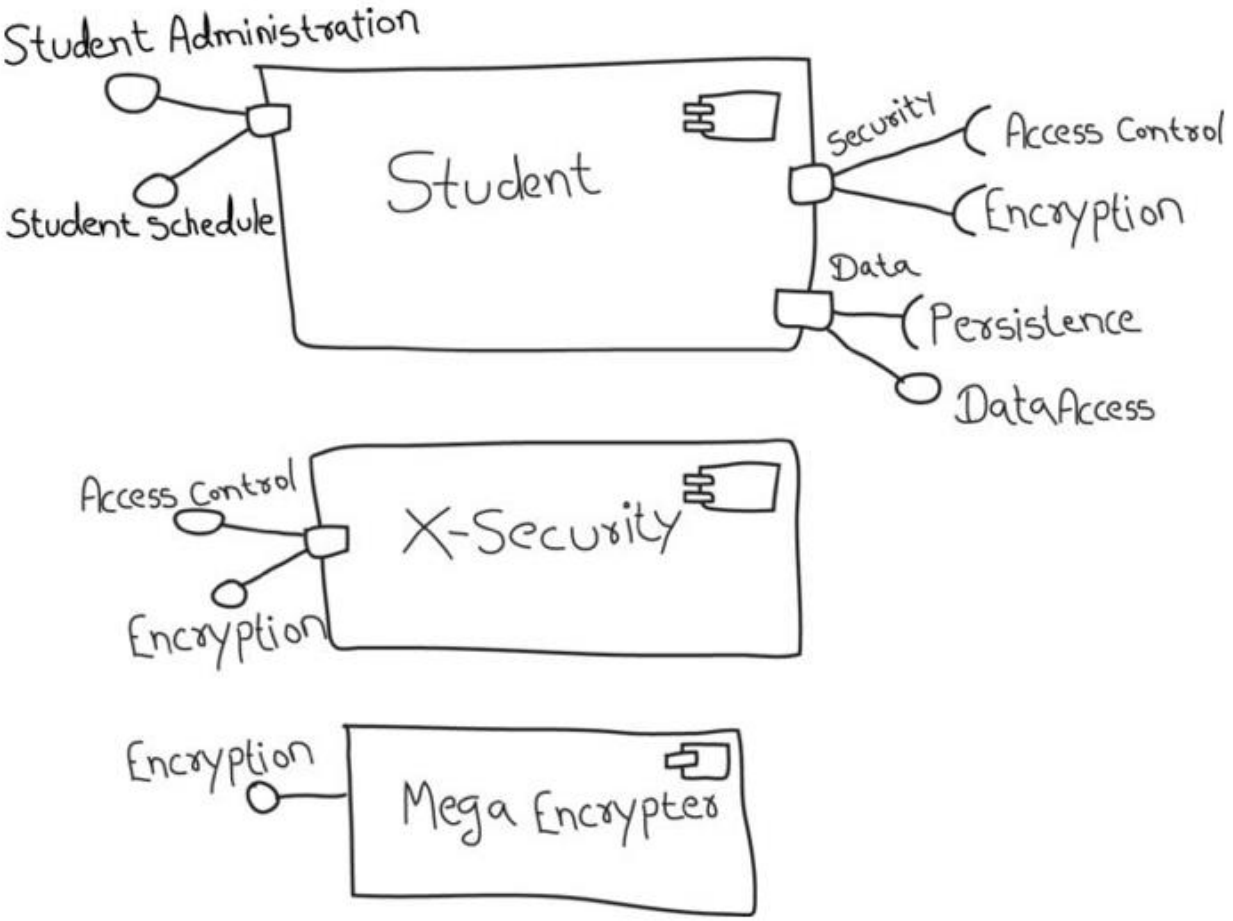
UML 1 Component Diagram Example



Interfaces And Ports

- An interface is the definition of a collection of one or more methods, and zero or more attributes, ideally one that defines a cohesive set of behaviors.
- A provided interface is modeled using the lollipop notation and
- A required interface is modeled using the socket notation.
- A port is a feature of a classifier that specifies a distinct interaction point between the classifier and its environment.

Component Port Example



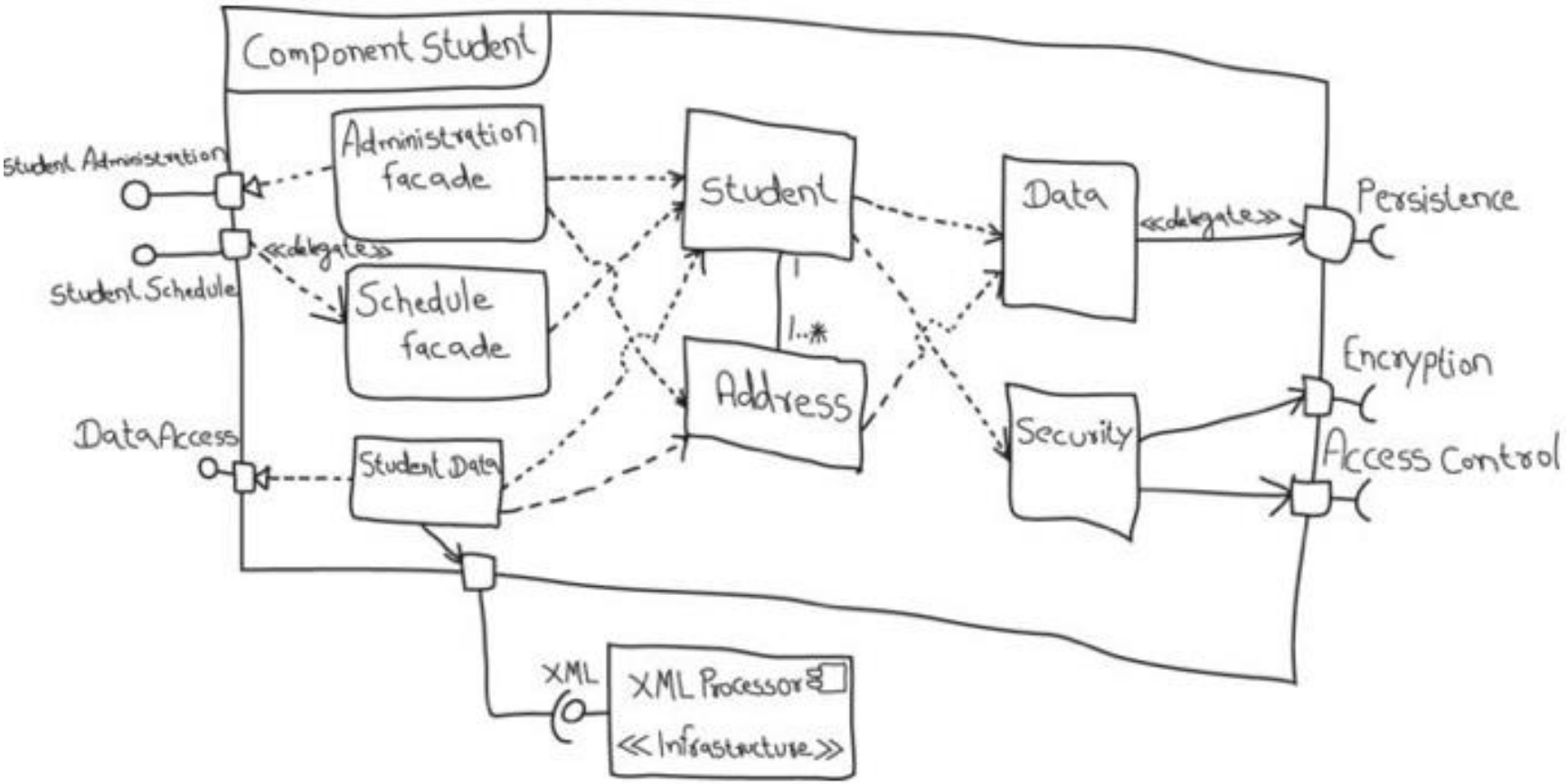
Tips For Designing Components

- Keep components cohesive.
- Assign user-interface classes to application components.
- Assign technical classes to infrastructure components.
- Define class contracts: any method that directly responds to methods sent from other objects.
- Assign hierarchies to the same components.
- Identify domain components: a set of classes that collaborate among themselves to support a cohesive set of contracts.

Tips For Designing Components 2

- Identify the “collaboration type” of business classes:
  - A server class is one that receives messages, but does not send them.
  - A client class is one that sends messages, but does not receive them.
  - A client/ server class is one that both sends and receives messages.
- Server classes belong in their own component
- Merge a component into its only client
- Pure client classes do not belong in domain components
- Highly coupled classes belong in the same component
- Minimize the size of the message flow between components
- Define component contracts

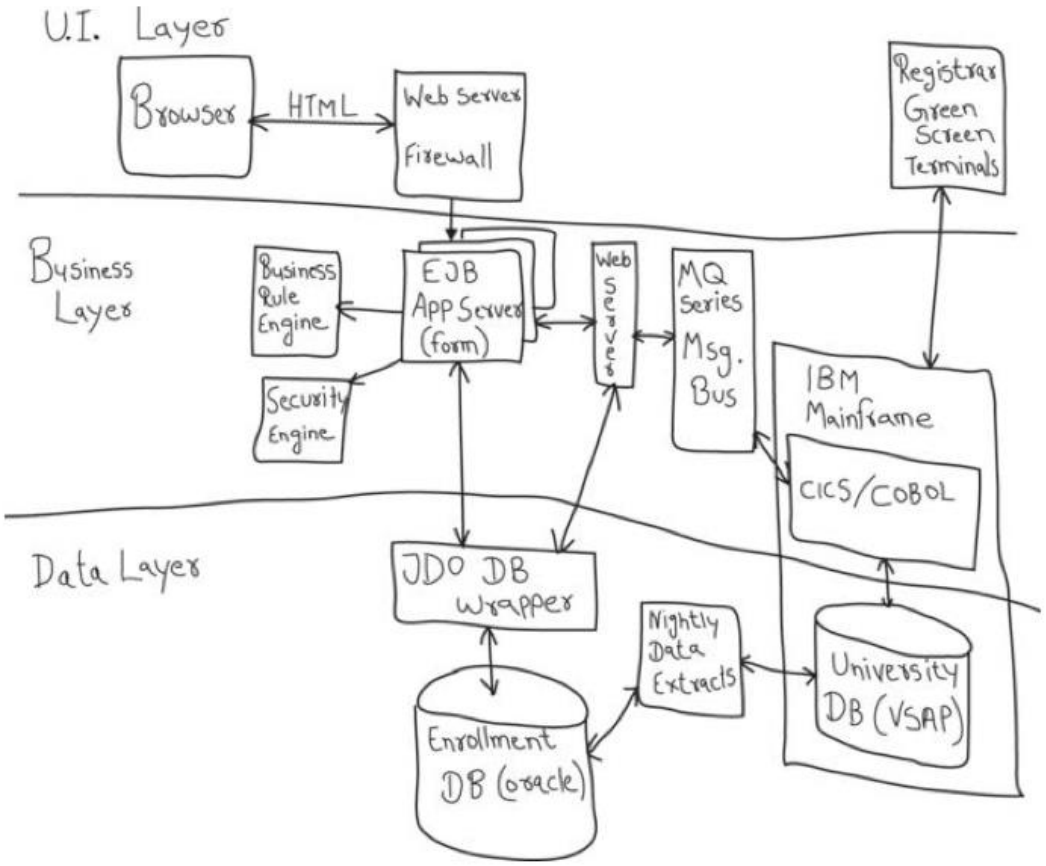
Inside A Component Example



Freeform Diagrams

- One of the most useful, and most common, types of model is a free-form diagram.
- Yet they rarely seem to be recognized as an “official” diagram type.
- A mishmash of information like this several UML diagrams to capture; yet this single sketch seems to communicate the architectural landscape for your system nicely.

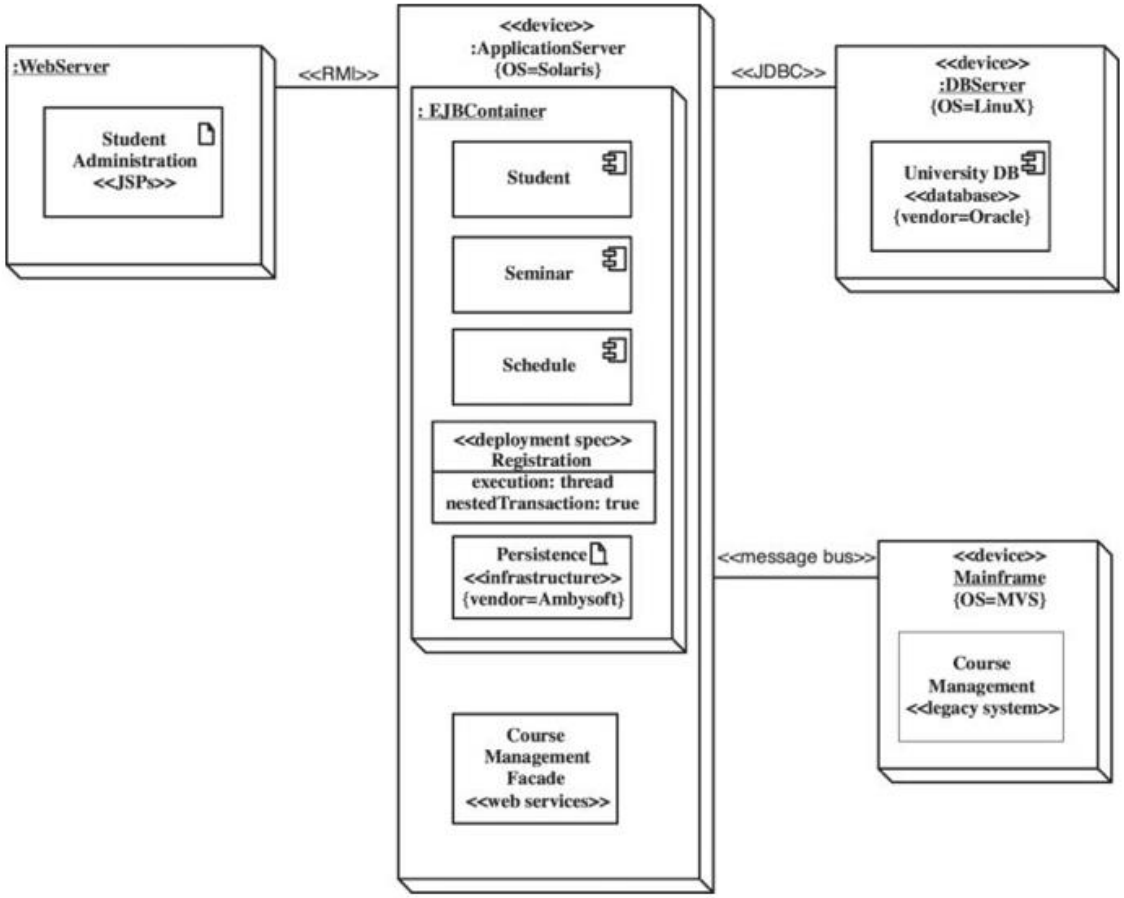
Free Form Diagram Example



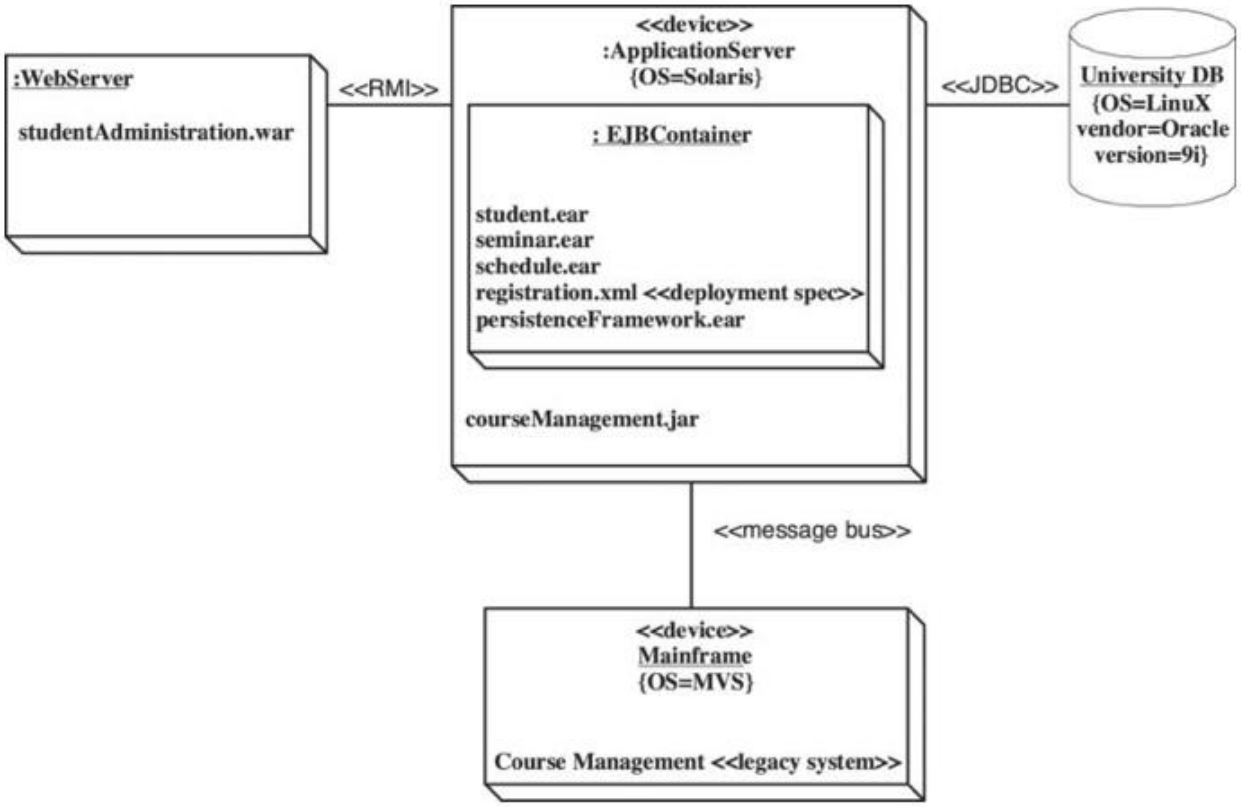
UML Deployment Diagrams

- A UML 2 deployment diagram depicts a static view of the run-time configuration of processing nodes and the components that run on those nodes.
- The three-dimensional boxes represent nodes, either software or hardware.
- Physical nodes should be labeled with the stereotype device, to indicate that it is a physical device such as a computer or switch.
- Nodes can contain other nodes or software artifacts.
- Deployment specifications are basically configuration files, such as an Enterprise Java Beans (EJB) deployment descriptor, which define how a node should operate.
  - They are depicted as two-sectioned rectangles with the stereotype deployment spec; the top box indicates the name and the bottom box lists the deployment properties (if any) for the node.

Deployment Diagram Example



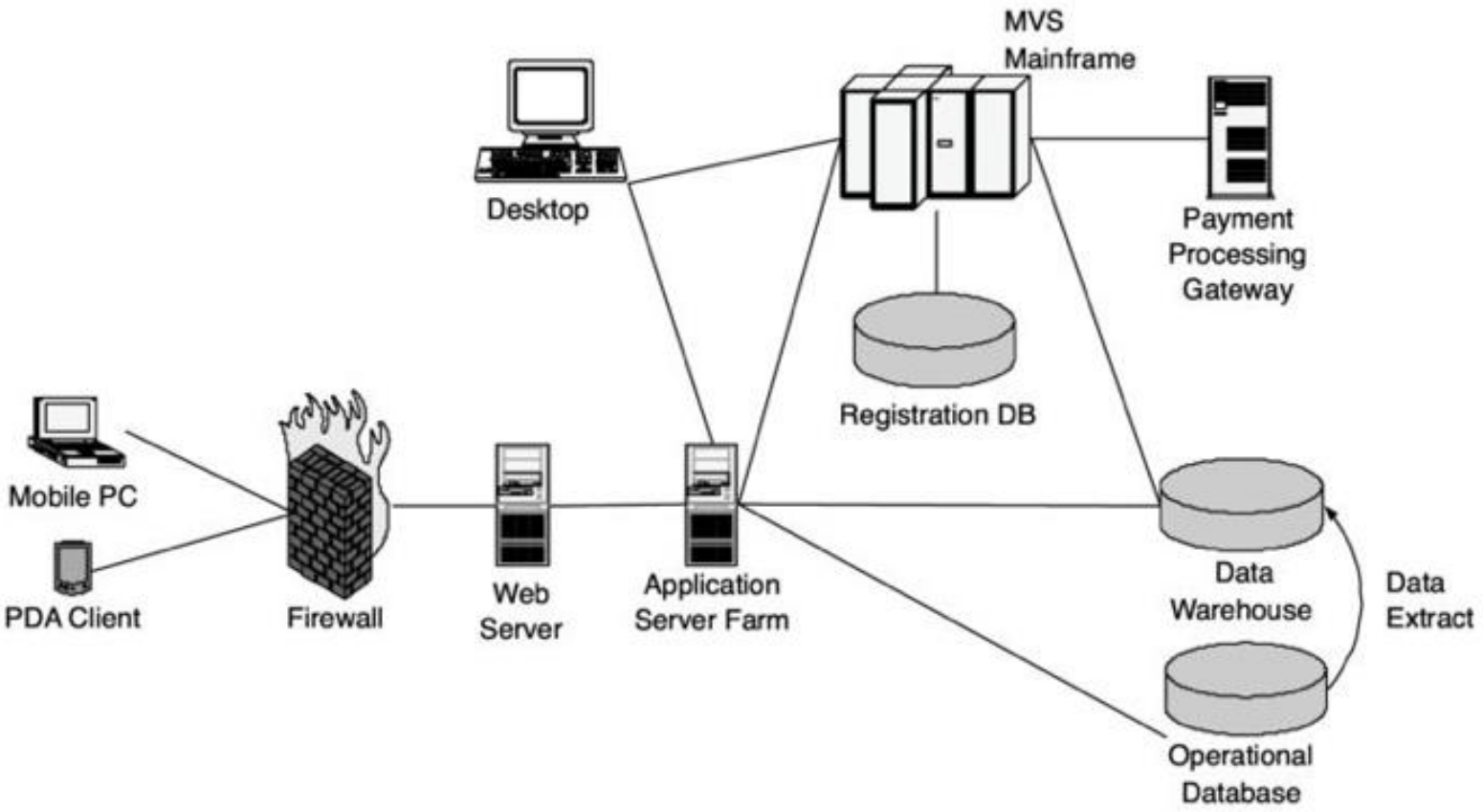
Concise Deployment Diagram Example



Network Diagrams

- Network diagrams are commonly used to depict hardware nodes as well as the connections between them.
- Network diagrams are arguably a high-level form of UML deployment diagram with extensive use of visual stereotypes.

Network Diagram Example



Layering Your Architecture

- Layering is the concept of organizing your software design into layers/ collections of functionality that fulfill a common purpose, such as implementing your user interface or the business logic of your system.
- You should be able to make modifications to any given layer without affecting any other layers.

Common Layers

- Interface classes. These classes wrap access to the logic of your system. There are two categories of interface class:
  - UI classes that provide people access to your system. JSPs and graphical user interface (GUI) screens implemented via the Swing class library are commonly used to implement UI classes within Java
  - system interface (SI) classes that provide access to external systems to your system.. Web services and CORBA wrapper classes are good options for implementing SI classes.
- Domain classes. These classes implement the concepts pertinent to your business domain such as Student or Seminar, focusing on the data aspects of the business objects, plus behaviors specific to individual objects. EJB entity classes are a common approach to implementing domain classes within Java.
- Process classes. Also called controller classes, these classes implement business logic that involves collaborating with several domain classes or even other process classes.
- Persistence classes. These classes encapsulate the capability to store, retrieve, and delete objects permanently without revealing details of the underlying storage technology.
- System classes. These classes provide operating-system-specific functionality for your applications, isolating your software from the operating system (OS) by wrapping OS-specific features, increasing the portability of your application.

Layering Diagram Example