SWEN30006
Software Design and Modelling

Architectural Analysis and Logical Architecture

Textbook: Larman Chapter 13, 33 and 38.2

"Error, no keyboard - press F1 to continue."

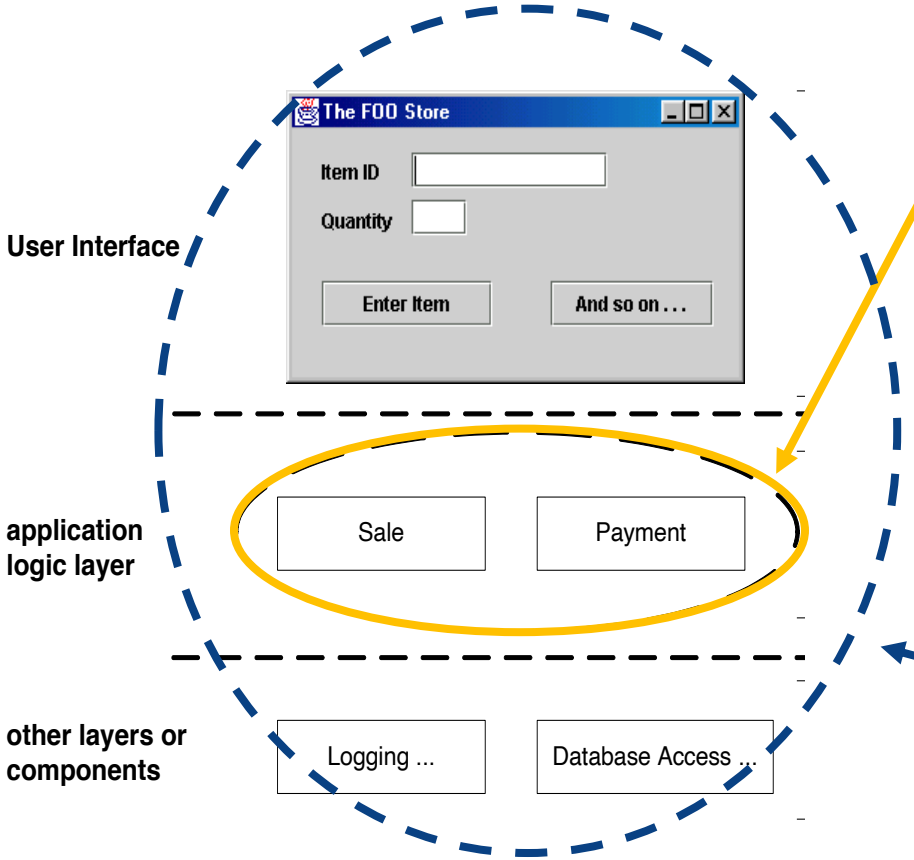
—early PC BIOS message

Learning Objectives

On completion of this topic you should be able to:

- Create architectural factor tables.
- Create technical memos that record architectural decisions.
- Define a logical architecture using layers.
- Illustrate the logical architecture using UML package diagrams.

(Revisited) Software Design



Weeks 3-9: The principles and approaches of the software classes in the application logic layer should be structured and interacting with each other.

- **GRASP**: Fundamental principles of assigning responsibilities, while concerning the maintainability of the system
- **GoF patterns**: a set of patterns to address specific problems, while achieving GRASP principles.

This week: A more holistic view of the system.

Software Architecture

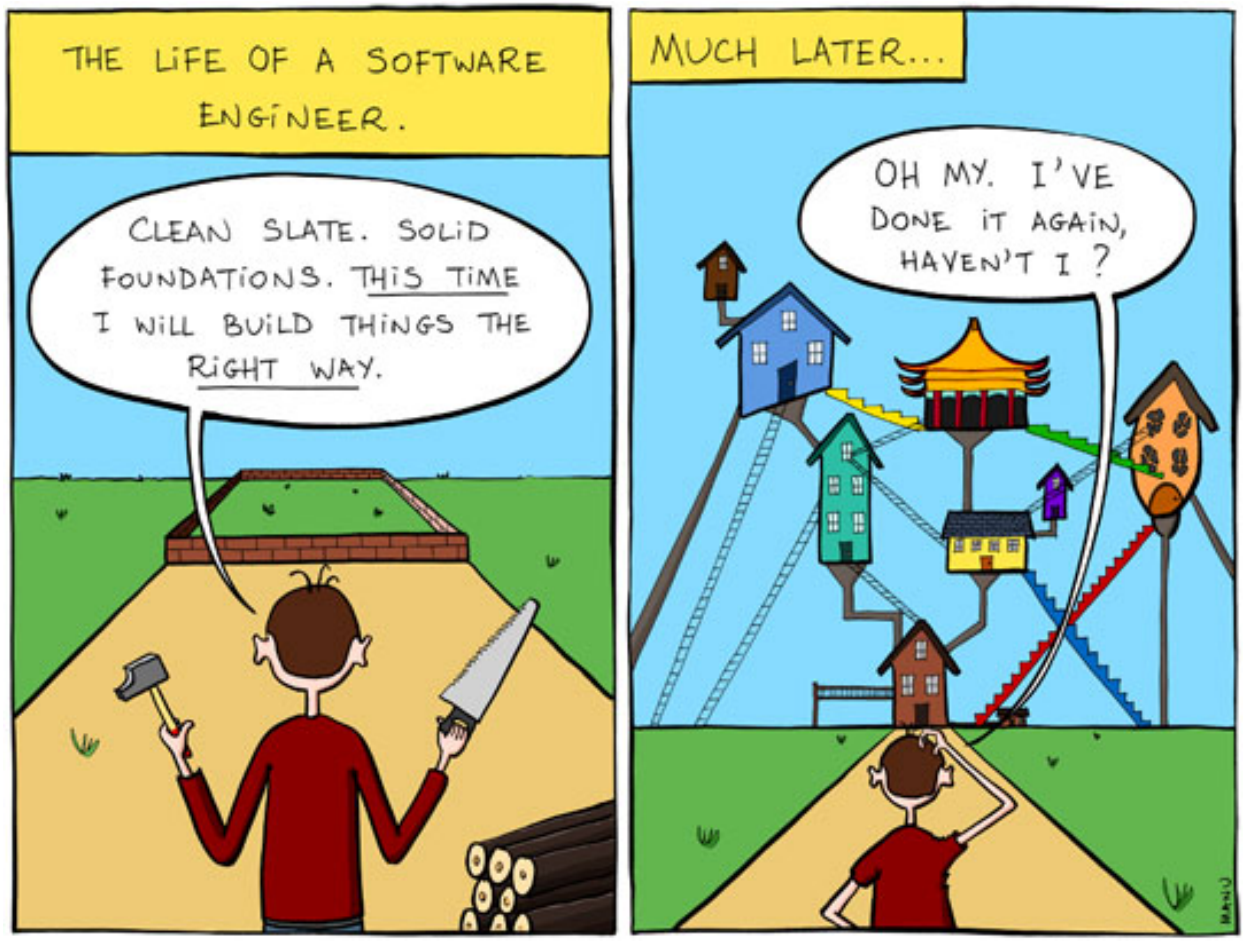
Definition: Software Architecture is the large-scale organization of elements in a software system.

- A design that concerns whether the system will align with business logic and be easy to maintain and evolve

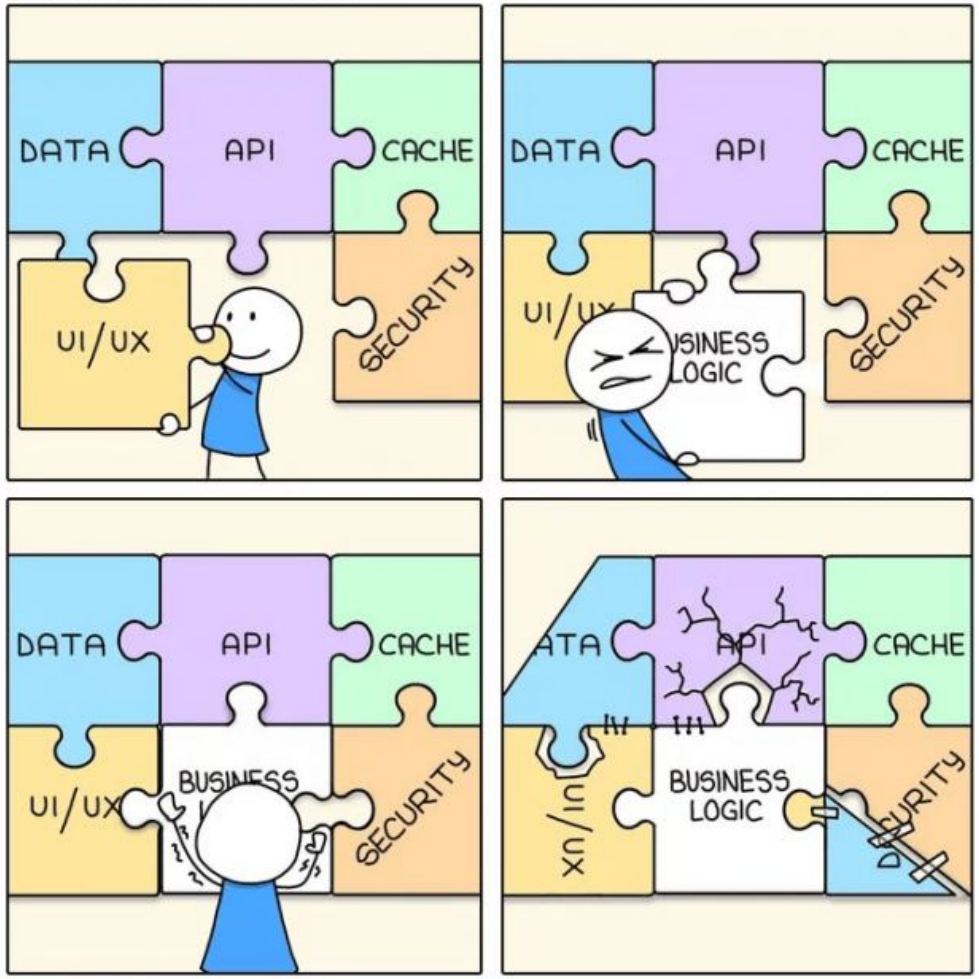
Software Architecture involves a set of significant ***decisions*** which concern:

- *Structural elements*: How should the software classes/components be in the system?
- *Interfaces*: How should each element is composed altogether?
- *Collaboration*: How should these elements work together according to the business logic?
- *Composition*: How should these elements be grouped progressively into larger subsystems?

Why Software Architecture is important?



Design the system based on partial sets of requirements



Design the system without focusing on the business logic

Today Topics

- Architectural Analysis
  - Identifying the important factors
- Logical Architecture
  - Organizing software elements

THE UNIVERSITY OF MELBOURNE

Architectural Analysis

Architectural Analysis

Definition: An activity to identify *factors* that will influence the architecture, understand their variability and priority, and resolve them

- To identify and resolve system's non-functional requirements in the context of its functional requirements

Goal: To reduce risk of missing a critical factor in the *design* of a system, focuses effort on high priority requirements, and aligns the product with business goals

Architectural Analysis includes identifying and analysing:

- *Architecturally significant requirement*: The requirement that can have a large impact on the design (especially when it was not considered at the beginning)
- Variation points
- Potential evolution points

Examples: Architecturally Significant Functional Requirements

Function	Description
Auditing	Provide audit trails of system execution.
Licensing	Provide services for tracking, acquiring, installing, and monitoring license usage.
Localization	Provide facilities for supporting multiple human languages.
Mail	Provide services that allow applications to send and receive mail.
Online help	Provide online help capability.
Printing	Provide facilities for printing.
Reporting	Provide reporting facilities.
Security	Provide services to protect access to certain resources or information.
System management	Provide services that facilitate management of applications in a distributed environment.
Workflow	Provide support for moving documents and other work items, including review and approval cycles.

Examples: Architecturally Significant Non-Functional Requirements

- Usability
  - e.g. aesthetics and consistency in the UI.
- Reliability
  - e.g. availability (the amount of system "up time"), accuracy of system calculations, and the system's ability to recover from failure.
- Performance
  - e.g. throughput, response time, recovery time, start-up time, and shutdown time.
- Supportability
  - e.g. testability, adaptability, maintainability, compatibility, configurability, installability, scalability, and localizability.

Effects of requirements on design

Examples of how the requirements can effect the design decision.

- Reliability and fault-tolerance requirements:
  - Ex: POS uses remote services (e.g., tax calculator). Will the remote services failover to local services be allowed?
- Adaptability and configurability requirements:
  - Ex: What if we want to sell POS to many retailers but they have variations in business rules. Can the retailers change the rules? Should POS be configurable?
- Brand name and branding requirements:
  - Ex: Retailers want to put their own brand in the POS user interfaces. Can the branding be changed in POS?

The design can be significantly changed based on the answer of this question.

Thinking Time

Which of the following requirement(s) for a new LMS is architecturally significant?

Participate at

https://pollev.com/unimelbse

Which of the following requirement(s) for new LMS is architecturally significant?

- The system will be deployed on Microsoft Windows or Linux server
- The system must record the activity log (e.g., page view) of each student
- A list of students will be retrieved from the University enrolment system
- The default font size is 12pts

Which of the following requirement(s) for new LMS is architecturally significant?

The system will be deployed on Microsoft Windows or Linux server

The system must record the activity log (e.g., page view) of each student

A list of students will be retrieved from the University enrolment system

The default font size is 12pts

Common Steps in Architectural Analysis

Steps (occurs in early elaboration):

1. Identify/analyse *architectural factors:* requirements with impact on the architecture (especially non-functional requirements)
   - overlaps with requirements analysis
   - some identified/recorded during inception, now investigated in more detail
2. For the architectural factors, analyse alternatives and create solutions: *architectural decisions*
   - *e.g. remove requirement; custom solution; stop project; hire expert*

Priorities

1. Inflexible constraints
   - e.g. safety; security, legal compliance
2. Business goals
   - e.g. demo for clients: tradeshow in 18 months
   - e.g. competitor-driven window-of-opportunity
3. Other goals
   - requirements all relate back to higher-level goals, and eventually to business goals
   - e.g. extendible → new release every 6 months

Architectural Factor Table

A documentation that records the influence of the factors, their priorities, and their variability (immediate need for flexibility and future evolution)

1. Factor
2. Measures and quality scenarios
3. Variability (current flexibility and future evolution)
4. Impact of factor (and its variability) on stakeholders, architecture and other factors
5. Priority for success
6. Difficulty or risk

Example: Architecture Factor Table

Reliability—Recoverability of POS

Factor	Recovery from remote service (e.g., Tax Calculator) failure
Measures and quality scenarios	When remote service fails, re-establish connectivity with it within 1 min. of its detected re-availability, under normal store load in a production environment.
Variability (current flexibility and future evolution)	current flexibility - our SME says local client-side simplified services are acceptable (and desirable) until reconnection is possible. evolution - within 2 years, some retailers may be willing to pay for full local replication of remote services (such as the tax calculator). Probability? High.
Impact of factor (and its variability) on stakeholders, architecture and other factors	High impact on large-scale design. Retailers really dislike it when remote services fail, as it prevents them from using a POS to make sales.
Priority for Success	High
Difficulty or Risk	Medium

Technical Memo

A documentation that records alternative solutions, decisions, influential factors, and motivations for the noteworthy issues and decisions

1. Issue
2. Solution Summary
3. Factors
4. Solution
5. Motivation
6. Unresolved Issues
7. Alternatives Considered

Example: Technical memo

Technical Memo: Issue: Reliability—Recovery from Remote Service Failure

Factors

Robust recovery from remote service failure, e.g., tax calculator, inventory

Solution

To satisfy the quality scenarios of reconnection with the remote services ASAP, use smart Proxy objects for the services, that on each service call test for remote service reactivation, and redirect to them when possible.

Where possible, offer local implementations of remote services. For example, implementing a small cache to store data (e.g., tax rates)

Achieve protected variation with respect to location of services using an Adapter created in a ServicesFactory.

Example: Technical memo (cont)

Motivation

Retailers really don't want to stop making sales! Therefore, if the NextGen POS offers this level of reliability and recovery, it will be a very attractive product, as none of our competitors provide this capability.

The small product cache is motivated by very limited client-side resources.

The real third-party tax calculator is not replicated on the client primarily because of the higher licensing costs, and configuration efforts (as each calculator installation requires almost weekly adjustments).

This design (Adapter and ServiceFactory) also supports the evolution point of future customers willing and able to permanently replicate services such as the tax calculator to each client terminal.

Unresolved Issues

none

Alternatives Considered

A “gold level” quality of service agreement with remote credit authorization services to improve reliability. It was available, but much too expensive.

Summary of Architectural Analysis

1. Concerns are especially related to non-functional requirements with awareness of business context, but addressing functional requirements and their variability.
2. Concerns involve system-level, large-scale, broad problems, with resolution involving large-scale or fundamental design decisions.
3. Must address interdependencies and trade-offs (e.g. security/performance, or anything/cost)
4. Involves the generation/evaluation of alternatives.

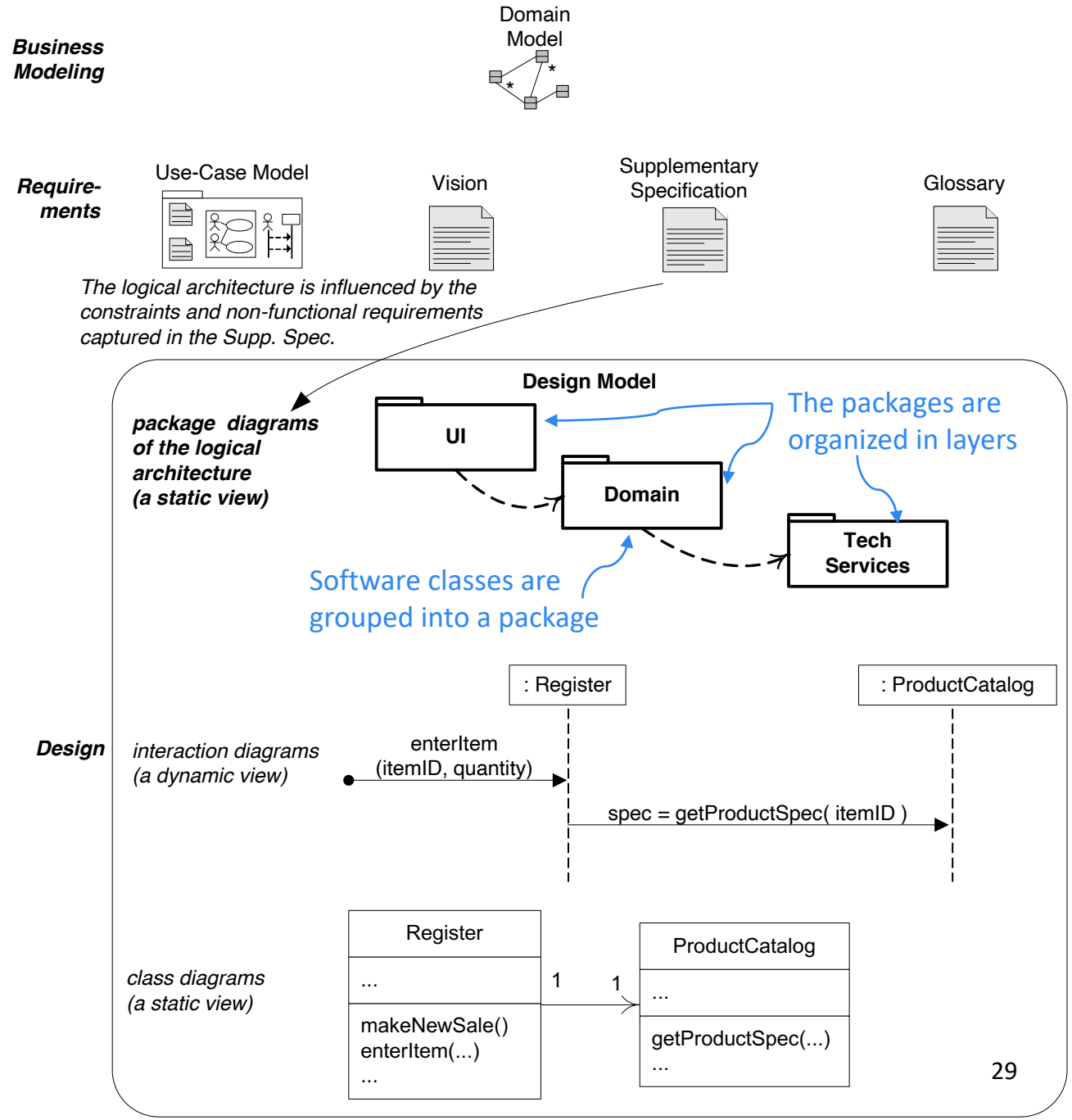
Logical Architecture

Logical Architecture

Logical Architecture (LA) - Definition:
The large-scale organisation of the software classes into *packages*, *subsystems* and *layers*

- Not concerned with networking, physical computers, or operating system processes (such concerns are for the *deployment architecture*)

LA defines the packages in which software classes are defined.



Layered Architecture

Layers - Definition: Coarse-grained grouping of classes, packages, or subsystems that has cohesive responsibility for a major aspect of the system.

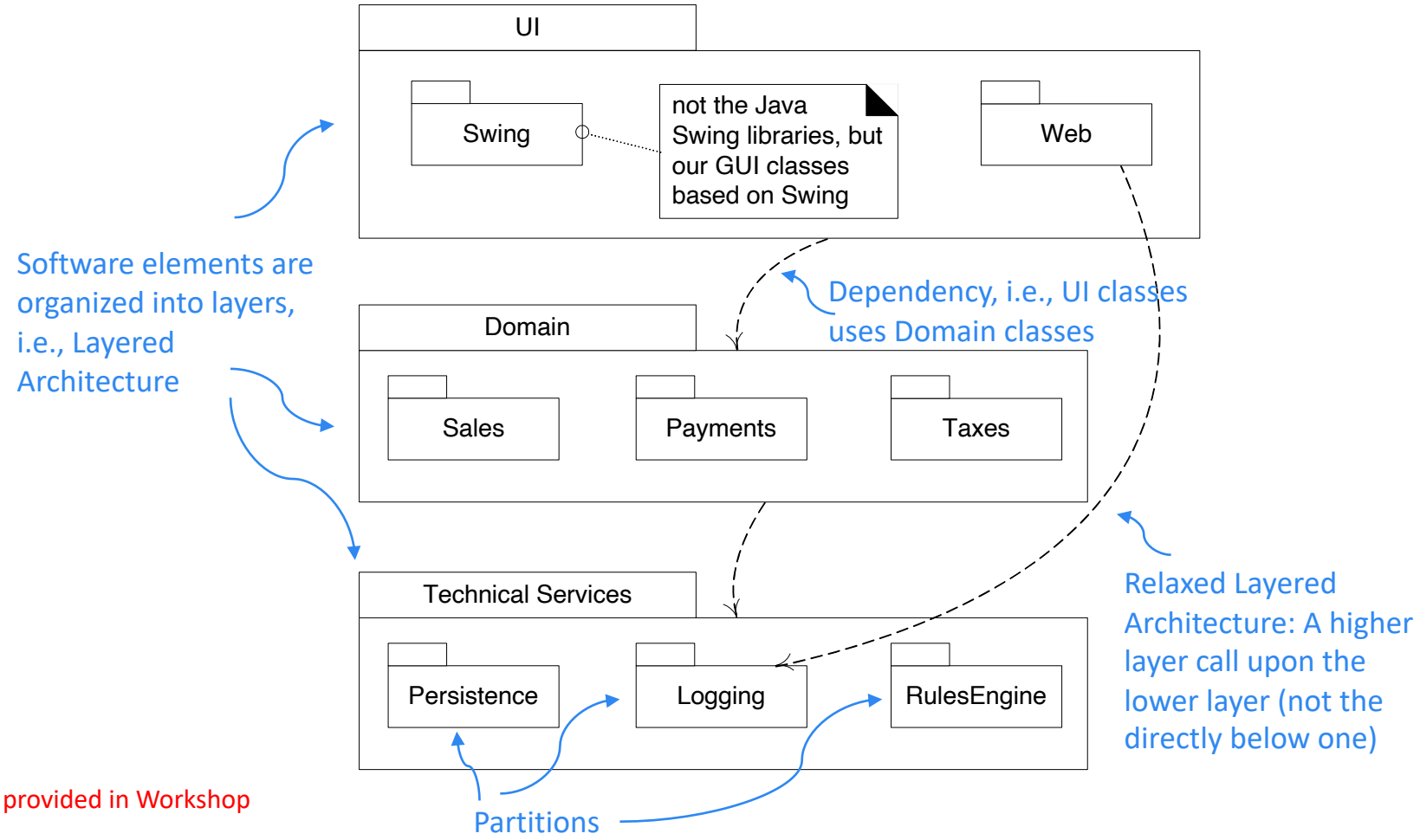
- Example layers:
  - **User Interface**.
  - **Application Logic and Domain Objects—**software objects representing domain concepts (*Sale* class) that fulfill application requirements.
  - **Technical Services—**general purpose objects and subsystems that provide supporting technical services (e.g., interfaces with DB)

Strict layered architecture: A layer only calls upon the services of the layer directly below it (e.g., a network protocol stack), not common in information systems

Relaxed layered architecture: A higher layer calls upon several lower layers. Common in IS.

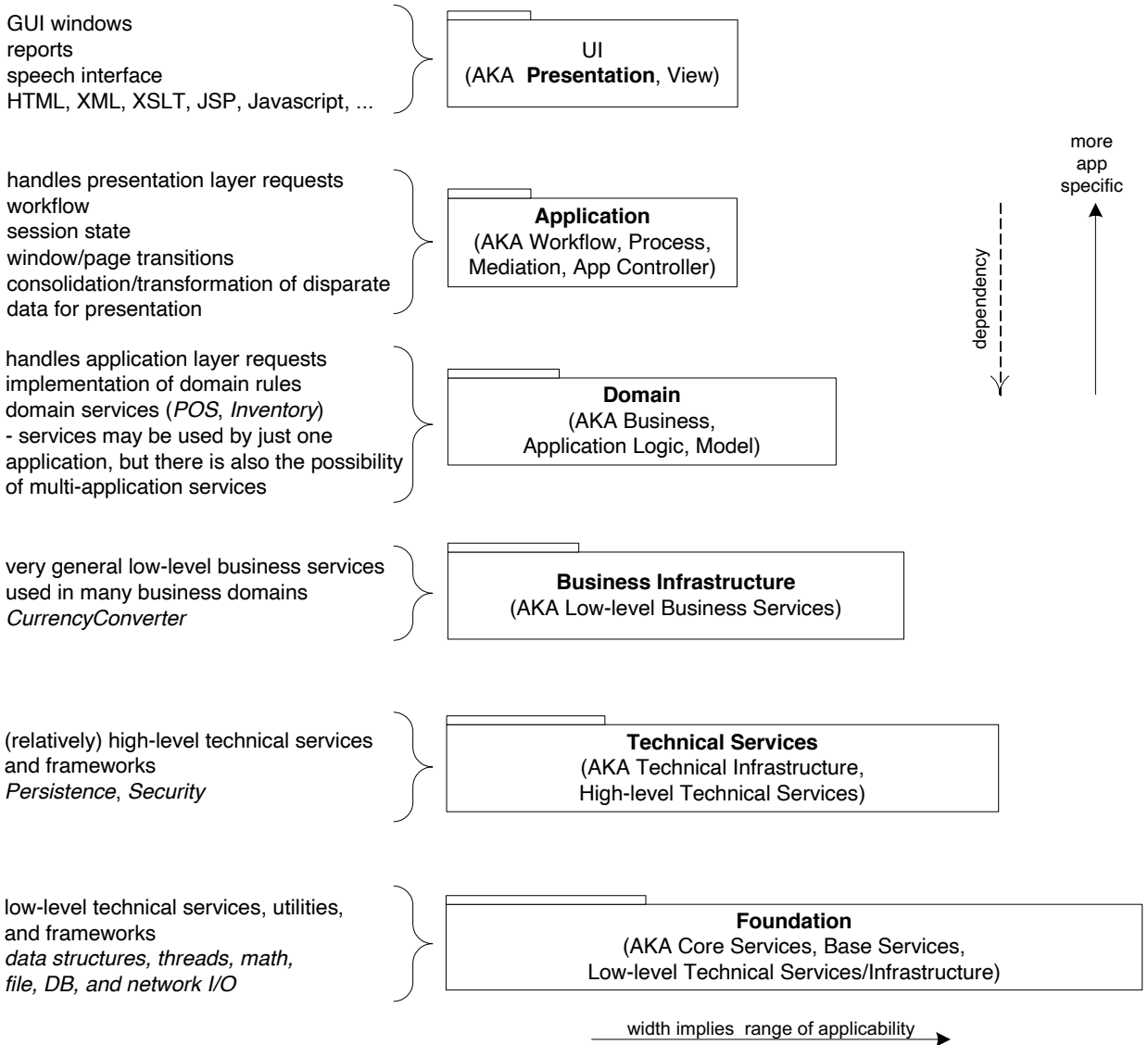
Layers and Partitions

UML Package diagram is used to illustrate the Logical Architecture



**UML Packages Notation is provided in Workshop supplementary materials

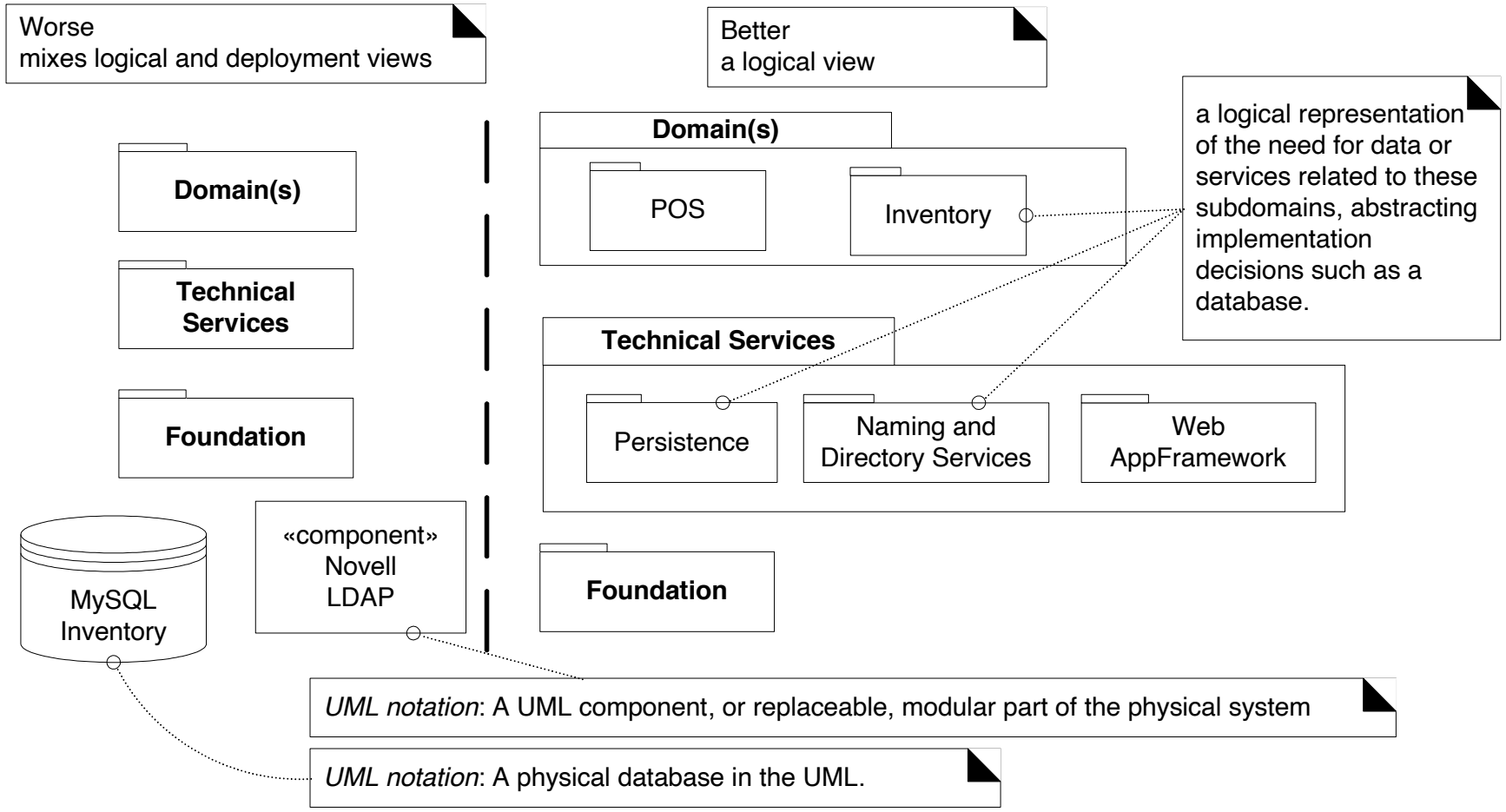
Common Layers: IS Logical Architecture



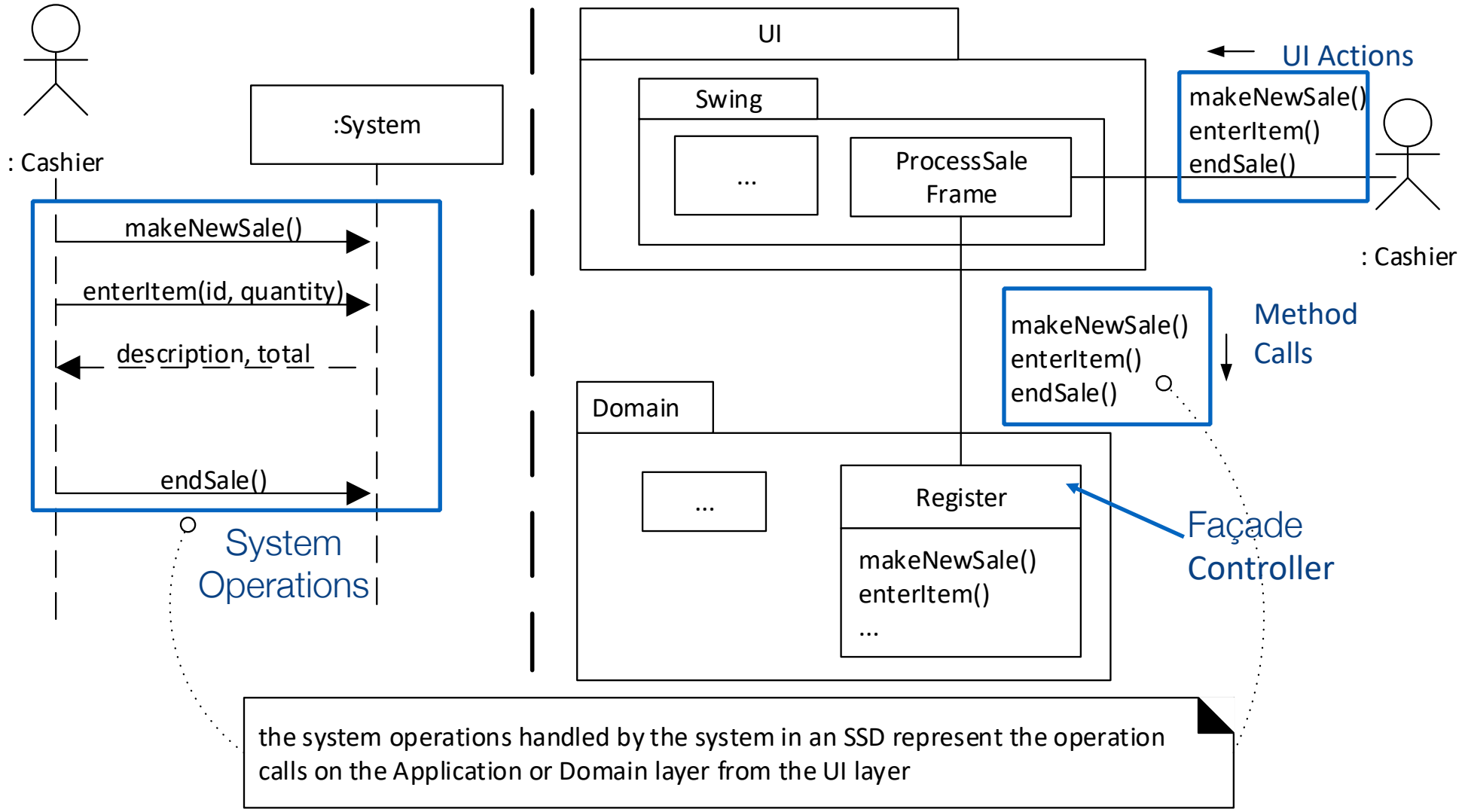
Guidelines:

- Organise large-scale logical structure of system into distinct cohesive layers from high application specific to low general services
  - Maintain separation of concerns, e.g. No application logic in UI objects
- Collaboration and coupling from higher layers to lower layers. Lower to higher layer coupling is avoided.
- Don't Show External Resources as the Bottom Layer

Examples: Logical Architecture



System Operations: SSDs and Layers



Benefits of Layered Architecture

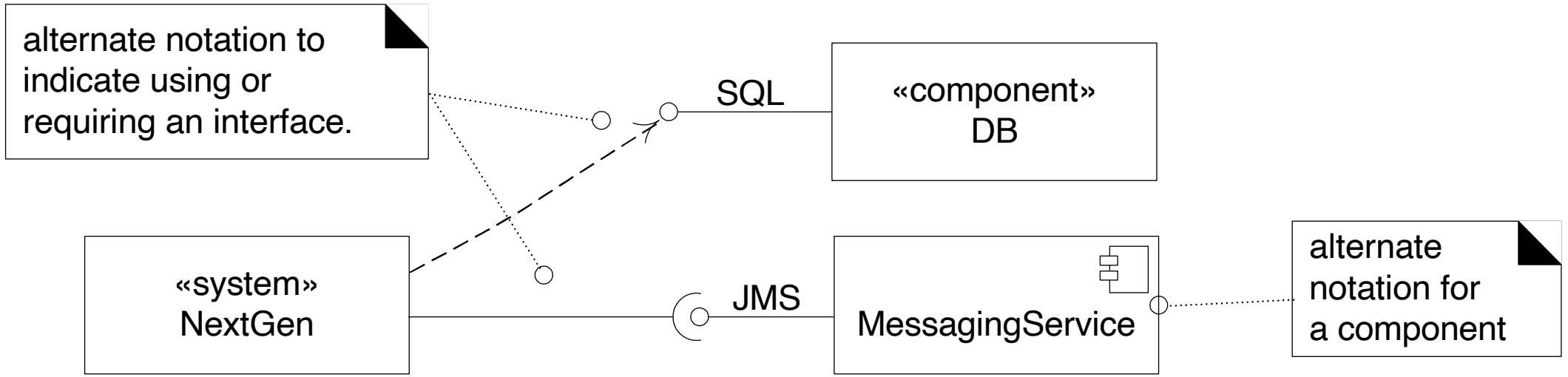
Using layers helps address the following problems:

- Changes rippling through system due to coupling
- Intertwining of application logic and UI, reducing reuse and restricting distribution options
- Intertwining of general technical services or business logic with application specific logic, reducing reuse, restricting distribution, and complicating replacement
- High coupling across areas of concern, impacting division of development work

Implementation View Architecture

- ***Component*** *is* a modular part of a system that encapsulates its contents and replaceable within its environment
- *Component Diagram*
  - Concerns on how to implement the software system in high level
  - Provides the initial architecture landscape for the system
  - Defines its behaviour in terms of provided and required interfaces
- *How is this different from a class?*
  - It can be a class! External resources (e.g., DB) and services are also considered as a component

UML Components

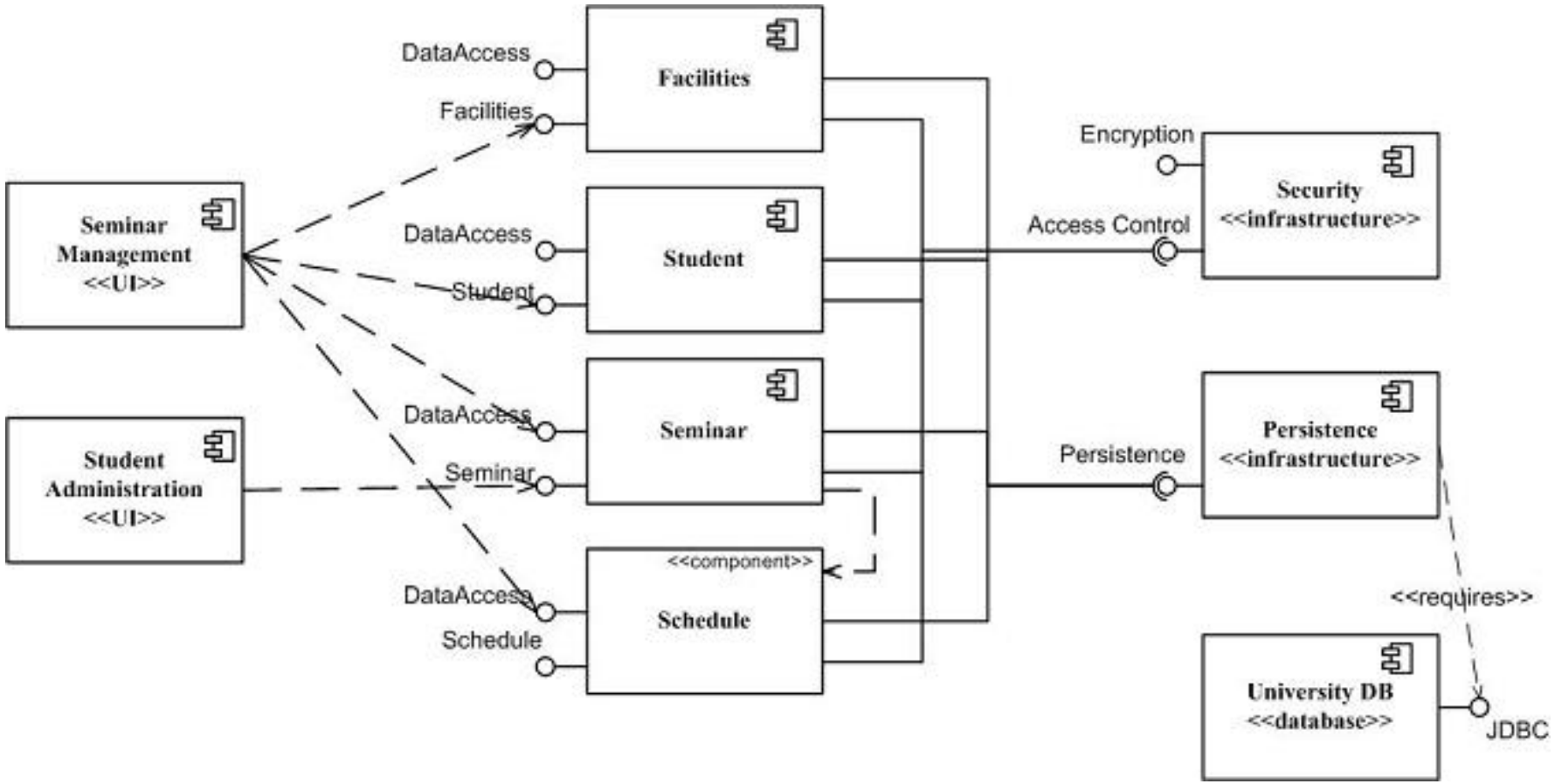


Interface Standards:

- SQL (Structured Query Language)
- JMS (Java Message Service)

**UML Component Notation is provided in Workshop supplementary materials

UML Component Example



http://agilemodeling.com/artifacts/componentDiagram.htm

Summary and Remarks

- **Software Architecture** concerns on a high level organization of software elements in the system and how the element collaborate to address the business logic
- **Architectural Analysis** is an activity of analyzing requirements (especially the non-functional requirements) to identify the factors that can have an impact on the design
- **Logical Architecture (LA)** is the large-scale organisation of the software classes
  - UML Package diagram illustrate the logical architecture
- The logical architecture can organize software classes in layers (i.e., ***Layered Architecture***)
- UML Component diagram provide a high-level implementation view of the architecture which consider the external services (e.g., databases)

NOTE: No live lecture on Week 11. The lecture video will be available by Thursday Oct 22.

Lecture Identification

Lecturer: Patanamon Thongtanunam

Semester 2, 2020

These slides include materials from:

Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development, Third Edition, by Craig Larman, Pearson Education Inc., 2005.