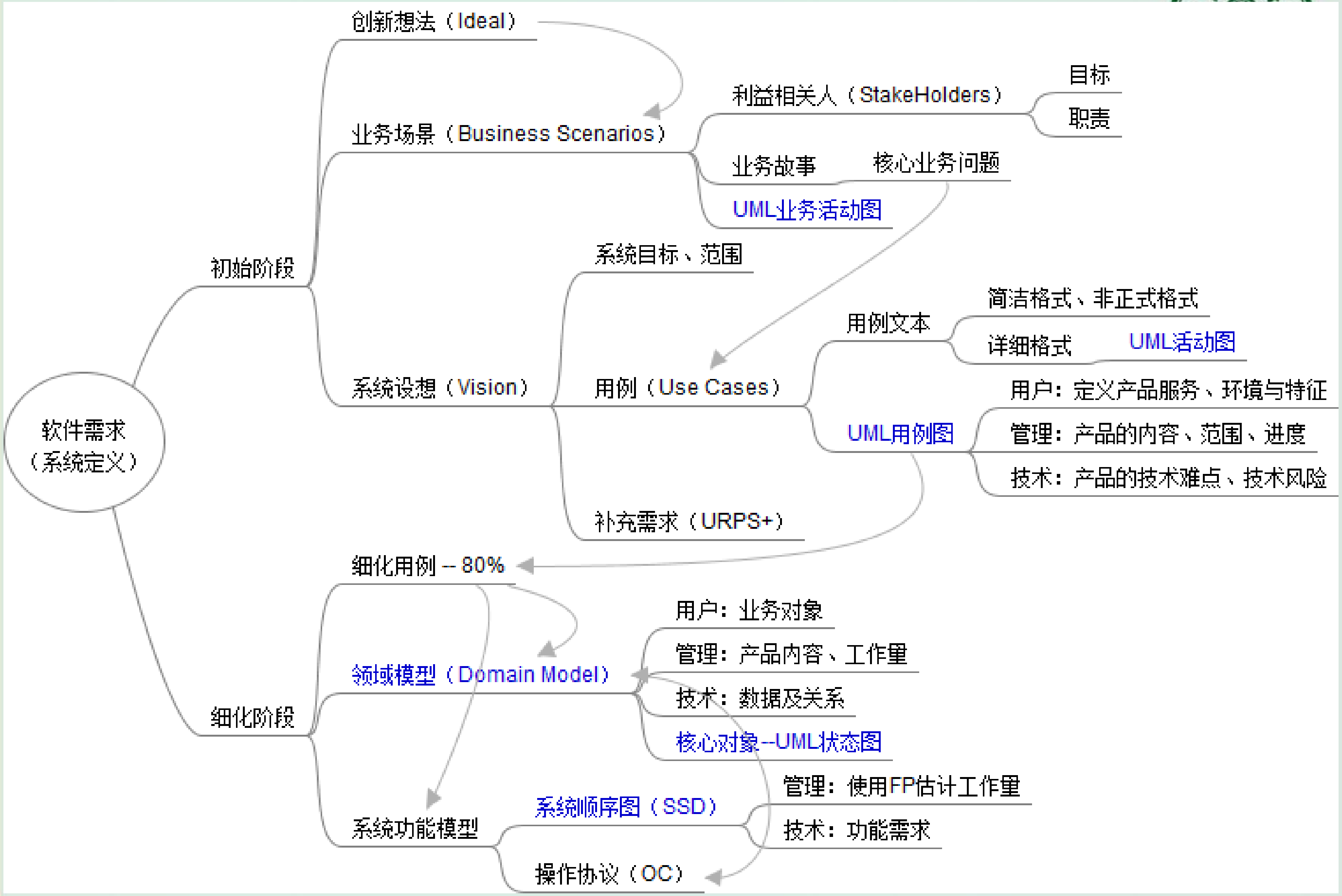
Applying UML and Patterns

An Introduction to
Object-oriented Analysis
and Design
and Iterative Development

Part III Elaboration Iteration I – Basic[1]

Chapters


软件需求（系统定义）
初始阶段
创新想法（Ideal）
业务场景（Business Scenarios）
利益相关人（StakeHolders）
目标
职责
业务故事
核心业务问题
UML业务活动图
系统设想（Vision）
系统目标、范围
用例（Use Cases）
用例文本
简洁格式、非正式格式
详细格式
UML活动图
UML用例图
用户：定义产品服务、环境与特征
管理：产品的内容、范围、进度
技术：产品的技术难点、技术风险
补充需求（URPS+）
细化阶段
细化用例 -- 80%
领域模型（Domain Model）
用户：业务对象
管理：产品内容、工作量
技术：数据及关系
核心对象--UML状态图
系统功能模型
系统顺序图（SSD）
管理：使用FP估计工作量
技术：功能需求
操作协议（OC）

Chap 12
Requirements to Design Iteratively

Iteratively
Do the Right Thing, Do the Thing Right

- Object-oriented requirements analysis
  - focused on to do the right thing;
    - understanding some of the outstanding goals, and related rules and constraints.
- The following design work stress to do the thing right
  - skillfully designing a solution to satisfy the requirements for this iteration.
- In iterative development, a transition from primarily requirements/ analysis to primarily design and implementation in each iteration.
  - Early iterations will spend more time on analysis activities.
  - Later iterations it is common that analysis lessens; there's more focus on just building the solution.

Provoking Early Change

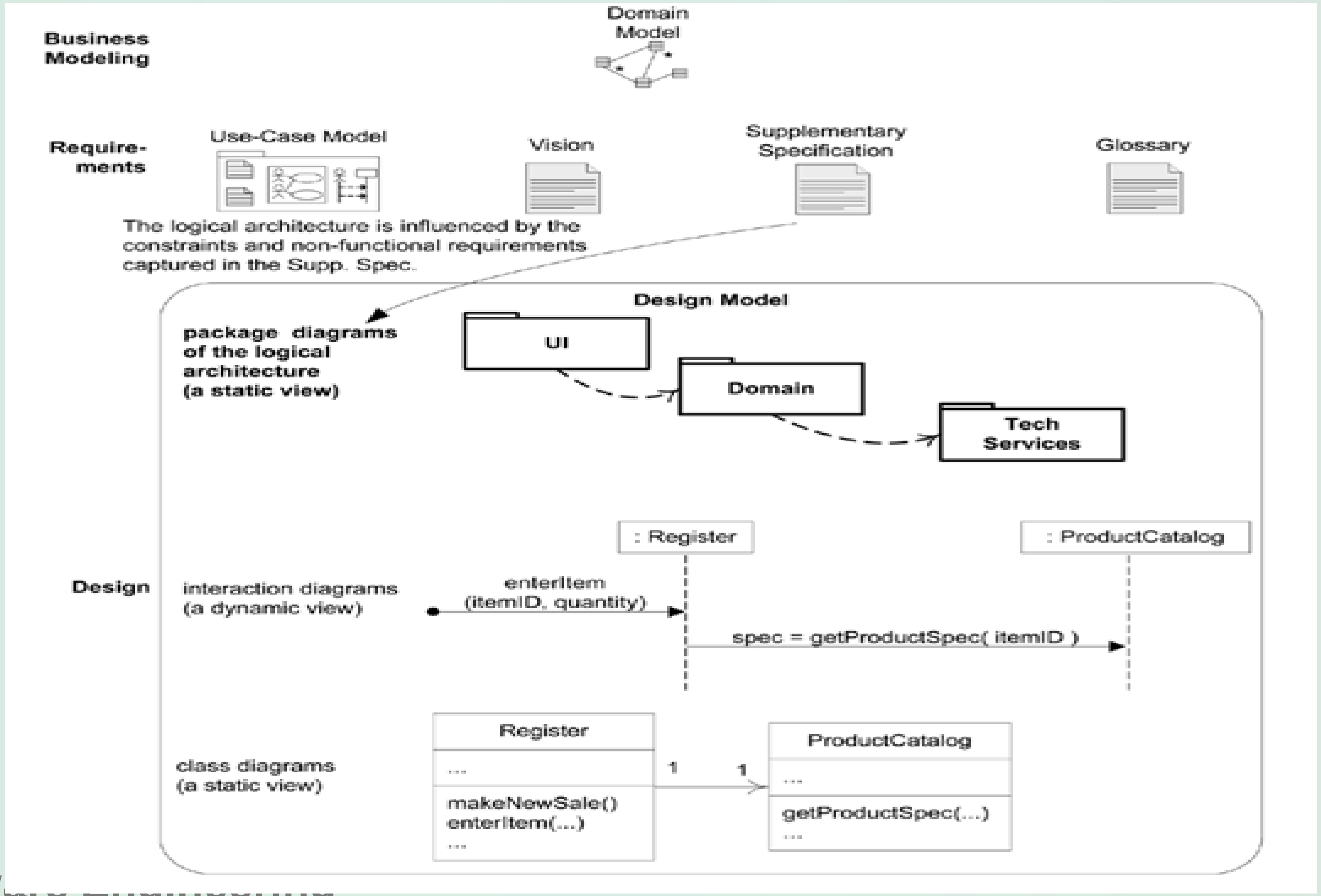
- It is **natural** to change some requirements during the design and implementation work, especially in the early iterations.
  - Iterative and evolutionary methods "embrace change“
  - to have a more stable goal (and estimate and schedule) for the later iterations.
  - Early programming, tests, and demos help provoke the inevitable changes early on.
- The discovery of changing specifications will both clarify the purpose of the design work of this iteration and refine the requirements understanding for future iterations.

Didn't All That Analysis and Modeling Take Weeks To Do

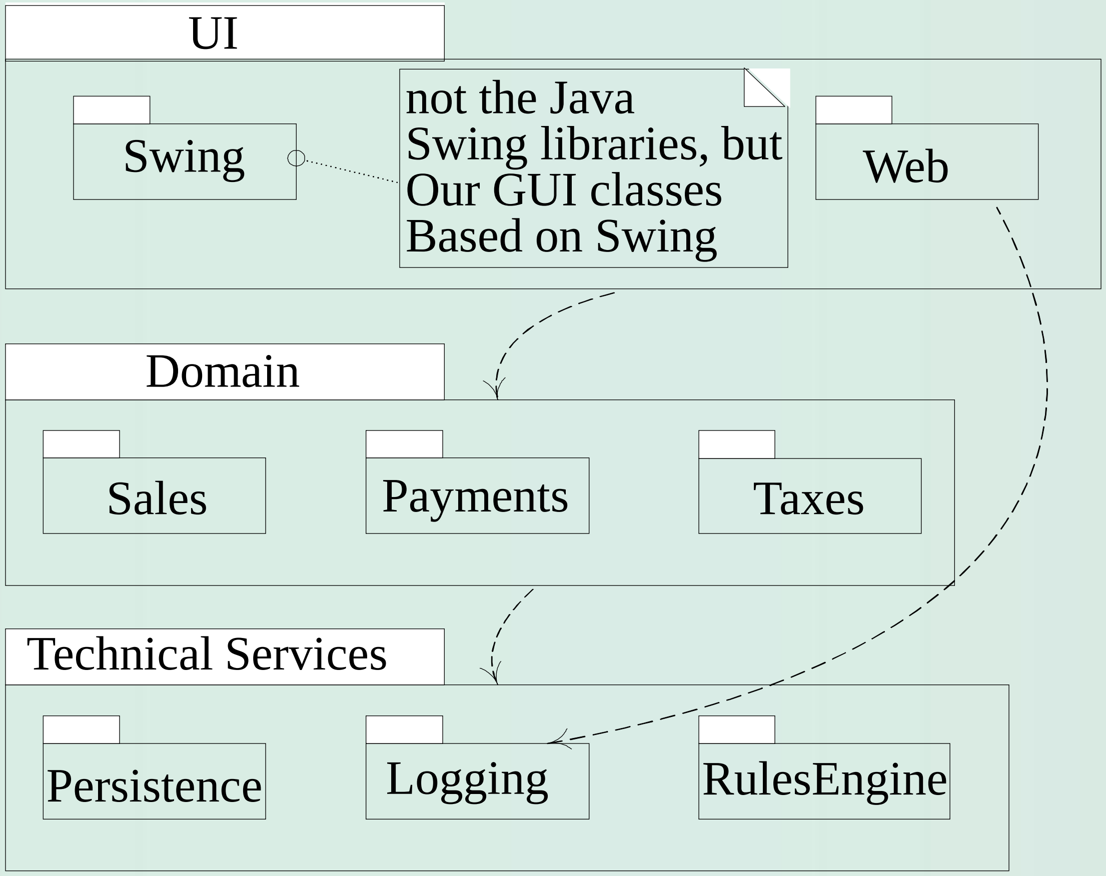
- The duration to do all the actual modeling (use case writing, domain modeling..) that has been explored so far is realistically just a **few hours or days**.
- Many other activities of project planning, such as proof-of-concept programming, finding resources (people, software, …), planning, setting up the environment could consume a few weeks of preparation.

Chap 13
Logical Architecture and UML Package Diagrams

OOD vs. LA


Business Modeling
Domain Model
Require-ments
Use-Case Model
Vision
Supplementary Specification
Glossary
The logical architecture is influenced by the constraints and non-functional requirements captured in the Supp. Spec.
Design Model
package diagrams of the logical architecture (a static view)
UI
Domain
Tech Services
Design
interaction diagrams (a dynamic view)
: Register
: ProductCatalog
enterItem (itemID, quantity)
spec = getProductSpec(itemID)
class diagrams (a static view)
Register
...
makeNewSale()
enterItem(...)
...
1
1
ProductCatalog
...
getProductSpec(...)
...

POS Example: Partial Layered, Logical Architecture



Software Architecture

- An architecture
- the set of <u>significant decisions</u> about the organization of a software system,
- the selection of the <u>structural elements</u> and <u>their interfaces</u> by which the system is composed
- their <u>behavior as specified</u> in the collaborations among those elements,
- the <u>composition of</u> these structural and behavioral elements into progressively larger subsystems,
- the <u>architectural style</u> guides this organization, these elements and their interfaces, their collaborations, and their composition.

Logical Architecture

- Logical architecture
  - the large-scale organization of the software classes into **packages** (or namespaces), **subsystems,** and **layers**.
  - there's no decision about how these elements are deployed across different operating system processes or across physical computers in a network (these latter decisions are part of the deployment architecture).

Layer Architecture

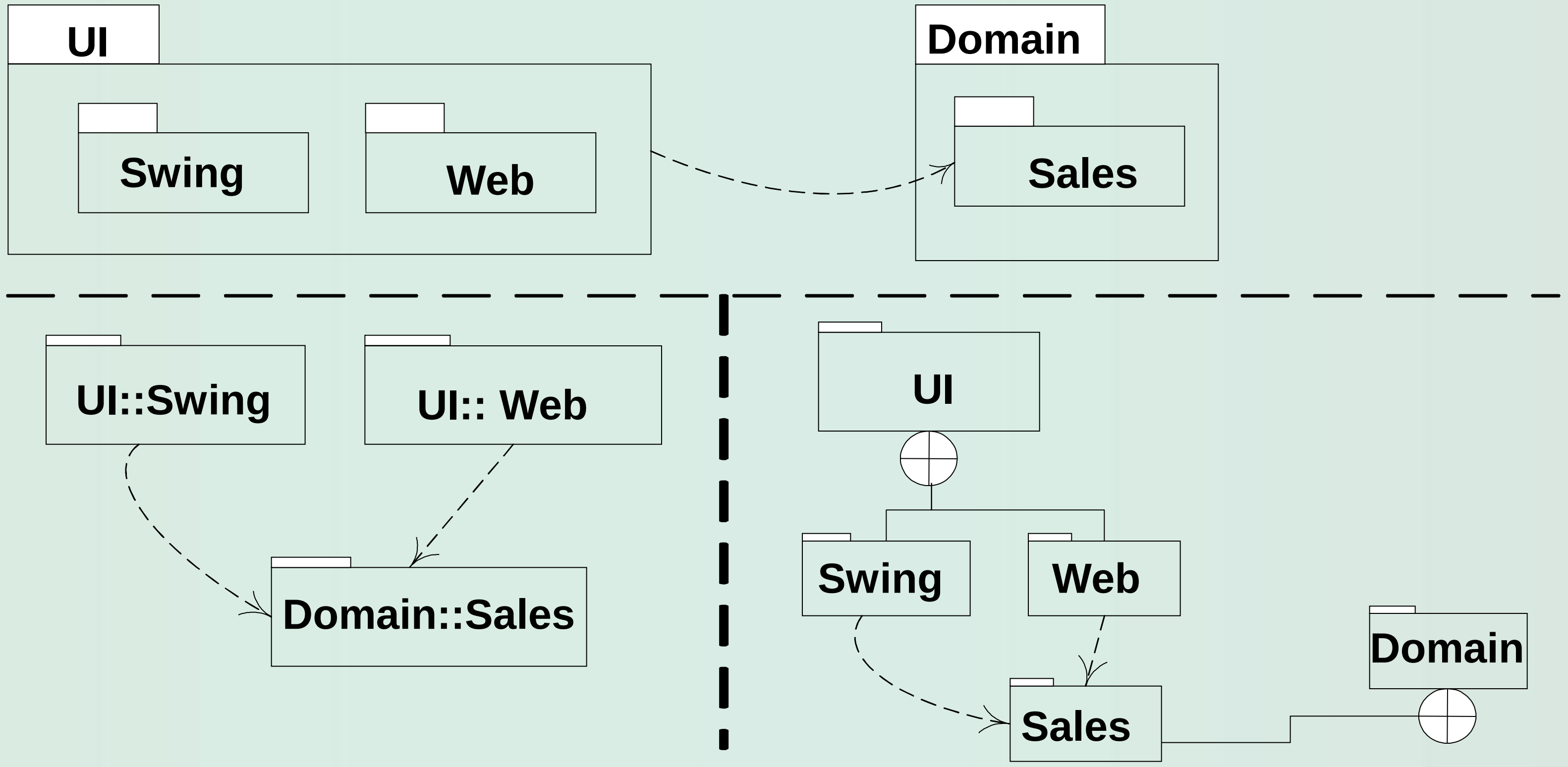
- Layers in an OO system include:
  - User Interface.
  - Application Logic and Domain Objects
    - representing domain concepts (e.g. software class Sale) that fulfill application requirements, such as calculating a sale total.
  - Technical Services
    - provide supporting technical services, such as interfacing with a database or error logging. These services are usually application-independent and reusable across several systems.
- In network protocol stacks, a layer only calls upon the services of the layer directly below it(strict layered architecture).
- Information system usually has a relaxed layered architecture, in which a higher layer calls upon several lower layers.

Applying UML: Package Diagrams 1

- UML package diagrams are used to illustrate the logical architecture of a system, the layers, subsystems, packages.
  - A layer can be modeled as a UML package; e.g., the UI layer modeled as a package named UI.
- A UML package diagram provides a way to group elements.
  - can group anything: classes, other packages, use cases...
  - Nesting packages is very common, java::util::Date.
  - A UML package is a more general concept than simply a Java package or .NET namespace.
  - To show dependency (a coupling) between packages so that developers can see the large-scale coupling in the system.
  - a dashed arrowed line with the arrow pointing towards the depended-on package.

Applying UML: Package Diagrams 2

- to show package nesting, using embedded packages, UML fully-qualified names, and the circle-cross symbol



Guideline: Design with Layers 1

- Organize the large-scale logical structure of a system into discrete layers of distinct, related responsibilities, with a clean, cohesive separation of concerns such that the "lower" layers are low-level and general services, and the higher layers are more application specific.
- Layers helps address problems
  - Source code changes are rippling throughout the system, many parts of the systems are highly coupled.
  - Application logic is intertwined with the user interface, so it cannot be reused with a different interface or distributed to another processing node.
  - technical services or business logic is intertwined with more application-specific logic, so it cannot be reused, distributed to another node, or easily replaced with a different implementation.
  - There is high coupling across different areas of concern. It is thus difficult to divide the work along clear boundaries for different developers.

Guideline: Design with Layers 2

- **Benefits of Using Layers**
  - A separation of high from low-level services, and of application-specific from general services. This reduces coupling and dependencies, improves cohesion, increases reuse potential, and increases clarity.
  - Related complexity is encapsulated and decomposable.
  - Some layers can be replaced with new implementations. This is generally not possible for lower-level Technical Service or Foundation layers (e.g., java.util), but may be possible for UI, Application, and Domain layers.
  - Lower layers contain reusable functions.
  - Some layers (primarily the Domain and Technical Services) can be distributed.
  - Development by teams is aided because of the logical segmentation.

Layers in IS Logical Architecture

GUI windows
reports
speech interface
HTML, XML, XSLT, JSP, Javascript, ...

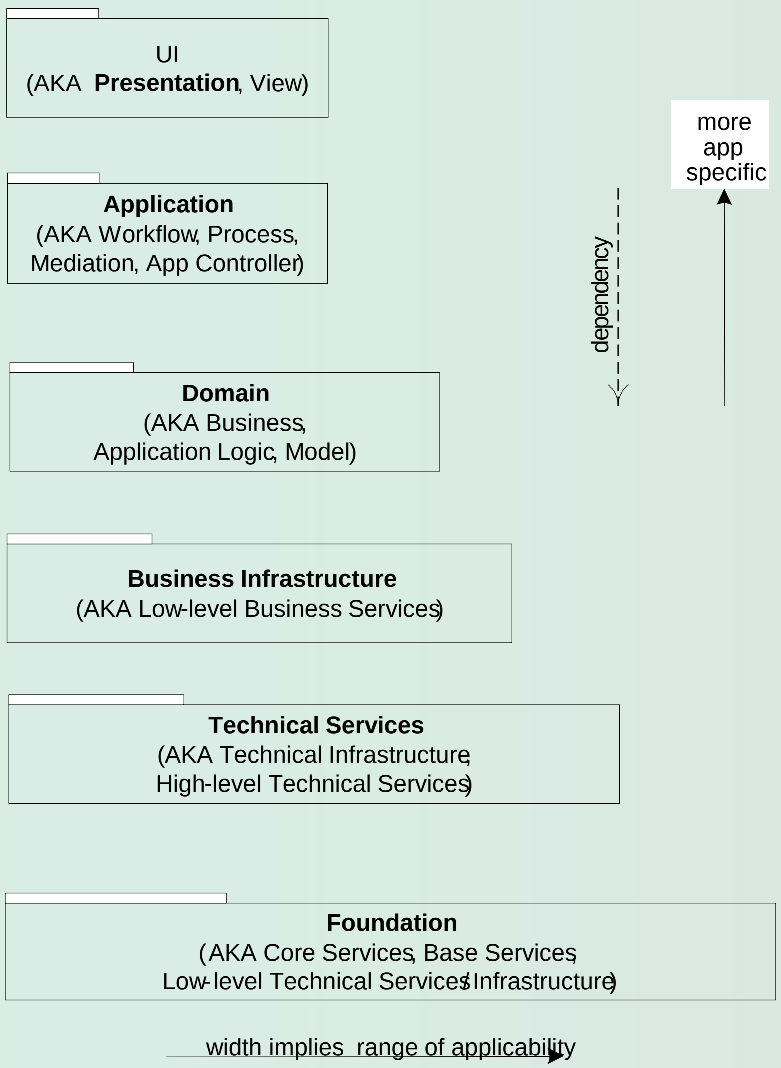
handles presentation layer requests
workflow
session state
window/page transitions
consolidation/transformation of disparate data for presentation

handles application layer requests
implementation of domain rules
domain services (*POS*, *Inventory*)
- services may be used by just one application, but there is also the possibility of multi-application services

very general low-level business services used in many business domains
CurrencyConverter

(relatively) high-level technical services and frameworks
Persistence, *Security*

low-level technical services, utilities, and frameworks
data structures, threads, math, file, DB, and network I/O


UI
(AKA **Presentation**, View)
Application
(AKA Workflow, Process, Mediation, App Controller)
Domain
(AKA Business, Application Logic, Model)
Business Infrastructure
(AKA Low-level Business Services)
Technical Services
(AKA Technical Infrastructure, High-level Technical Services)
Foundation
(AKA Core Services, Base Services, Low-level Technical Services/Infrastructure)
dependency
more app specific
width implies range of applicability

Guideline: Cohesive Responsibilities; Maintain a Separation of Concerns

- The responsibilities of the objects in a layer should be strongly related to each other and should not be mixed with other layers.
  - UI objects should focus on UI work, such as creating windows and widgets, capturing mouse and keyboard events.
  - Objects in the application logic or "domain" layer should focus on application logic, such as calculating a sales total or taxes, or moving a piece on a game board.
  - UI objects should not do application logic. e.g., a Java Swing JFrame object should not contain logic to calculate taxes or move a game piece.
  - Application logic classes should not trap UI mouse or keyboard events. .

Domain Layer/Application Logic Layer /Domain Objects

- To create software objects with names and information similar to the real-world domain, and assign application logic responsibilities to them.
  - The real world of POS: sales and payments.
  - Software solution: Sale and Payment class, and give application logic responsibilities.
  - Domain object: represents a thing in the problem domain space, and has related application or business logic, e.g., a Sale object being able to calculate its total.
- Domain layer of the architecture: contains domain objects to handle application logic work.

Domain Layer and Domain Model

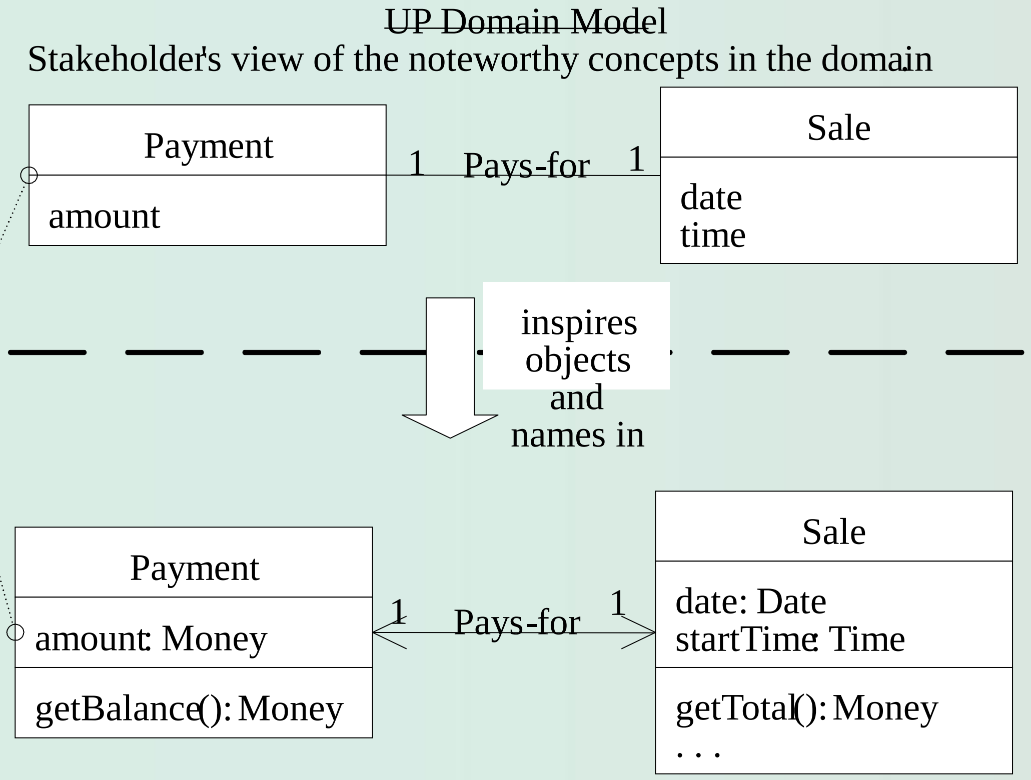
- The domain layer is part of the software
- The domain model is part of the conceptual-perspective analysis.
- Create a domain layer from the domain model, to achieve a lower representational gap, between the real-world domain, and software design.
- e.g., a Sale in the UP, Domain Model helps to creating a software Sale class in the domain layer of the Design Model.

Domain Layer and Domain Model

A Payment in the Domain Model is a concept, but a Payment in the Design Model is a software class. They are not the same thing, but the former *inspired* the naming and definition of the latter.

This reduces the representational gap.

This is one of the big ideas in object technology



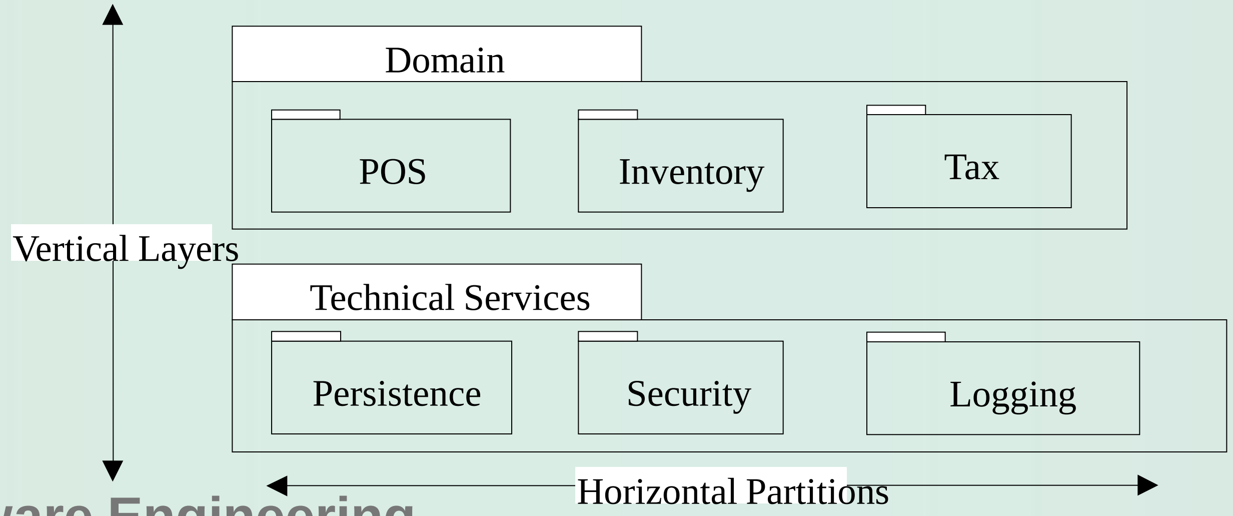
Domain layer of the architecture in the UP Design Model

The object-oriented developer has taken inspiration from the real world domain in creating software classes.

Therefore, the representational gap between how stakeholders conceive the domain, and its representation in software, has been lowered.

Tiers, Layers, and Partitions

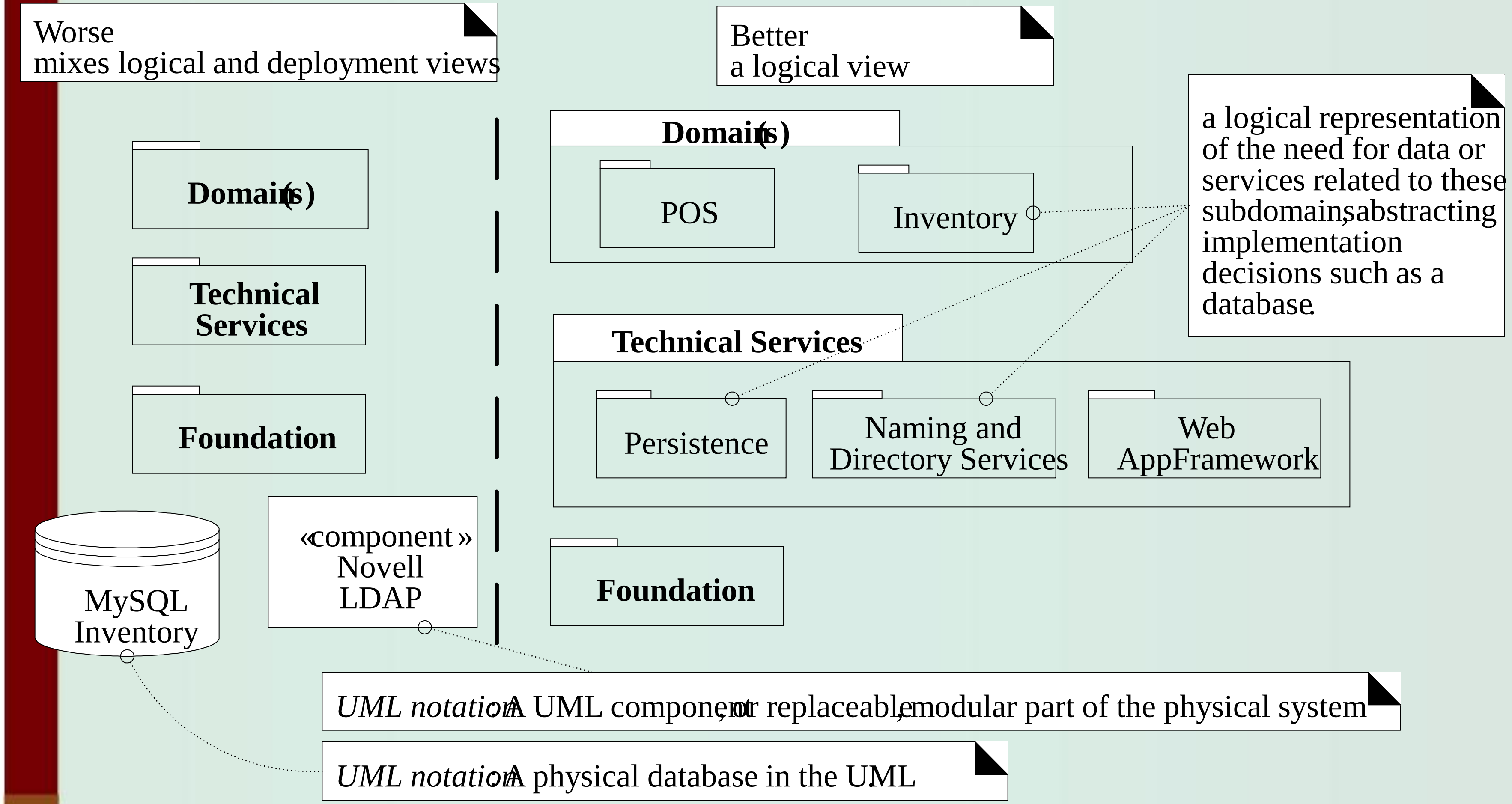
- Tier in architecture means a physical processing node
  - client tier, client computer
- Layers of architecture represents the vertical slices
- Partitions represent a horizontal division of relatively parallel subsystems of a layer.
  - e.g., the Technical Services layer may be divided into partitions such as Security and Reporting.



Guideline:
Don't Show External Resources as the Bottom Layer

- Most systems rely on external resources or services
  - e.g., MySQL in ventory database and Novell LDAP naming and directory service.
  - These are <u>physical implementation components</u>, not a layer in the logical architecture.
  - Showing these external resources in a layer "below" the Foundation layer mixes up the logical view and the deployment views of the architecture.
- The logical architecture and its layers,
  - access to a particular set of persistent data can be viewed as a sub-domain of the Domain Layer, the Inventory sub-domain.
  - the general services that provide access to databases may be viewed as a Technical Service partition, the Persistence service.

Guideline:
Don't Show External Resources as the Bottom Layer


Worse
mixes logical and deployment views
Domain(s)
Technical Services
Foundation
MySQL Inventory
«component» Novell LDAP
Better
a logical view
Domain(s)
POS
Inventory
Technical Services
Persistence
Naming and Directory Services
Web AppFramework
Foundation
a logical representation of the need for data or services related to these subdomains, abstracting implementation decisions such as a database.
UML notation: A UML component, or replaceable, modular part of the physical system
UML notation: A physical database in the UML

Guideline: Model-View Separation Principle 1

- Model-View Separation principle
  - **model** is the domain layer of objects. **View** is UI objects (windows mouse click on a button, Web pages, applets).
  - Do not connect or couple model objects directly to view objects
    - a Sale object should not directly send a message to a GUI window object ProcessSaleFrame, asking it to display something, change color, close.
    - don't let a Sale object have a reference to a Swing JFrame window object.
    - the windows are related to a particular application, while the non-windowing objects may be reused in new applications or attached to a new interface.
  - Do not put application logic (tax calculation) in UI object methods.
    - UI objects should only initialize UI elements, receive UI events, and delegate requests for application logic on to non-UI objects.

Guideline: Model-View Separation Principle 2

- Model-View-Controller (MVC).
  - data objects (models), GUI widgets (views), and mouse and keyboard event handlers (controllers).
  - "MVC" has been applied on a large-scale architectural level. The Model is the Domain Layer, the View is the UI Layer, and the Controllers are the workflow objects in the Application layer.
  - A related pattern of this principle is the **Observer pattern**.
  - e.g. a JFrame window should not have a method that does a tax calculation. A Web JSP page should not contain logic to calculate the tax. These UI elements should delegate to non-UI elements for such responsibilities.

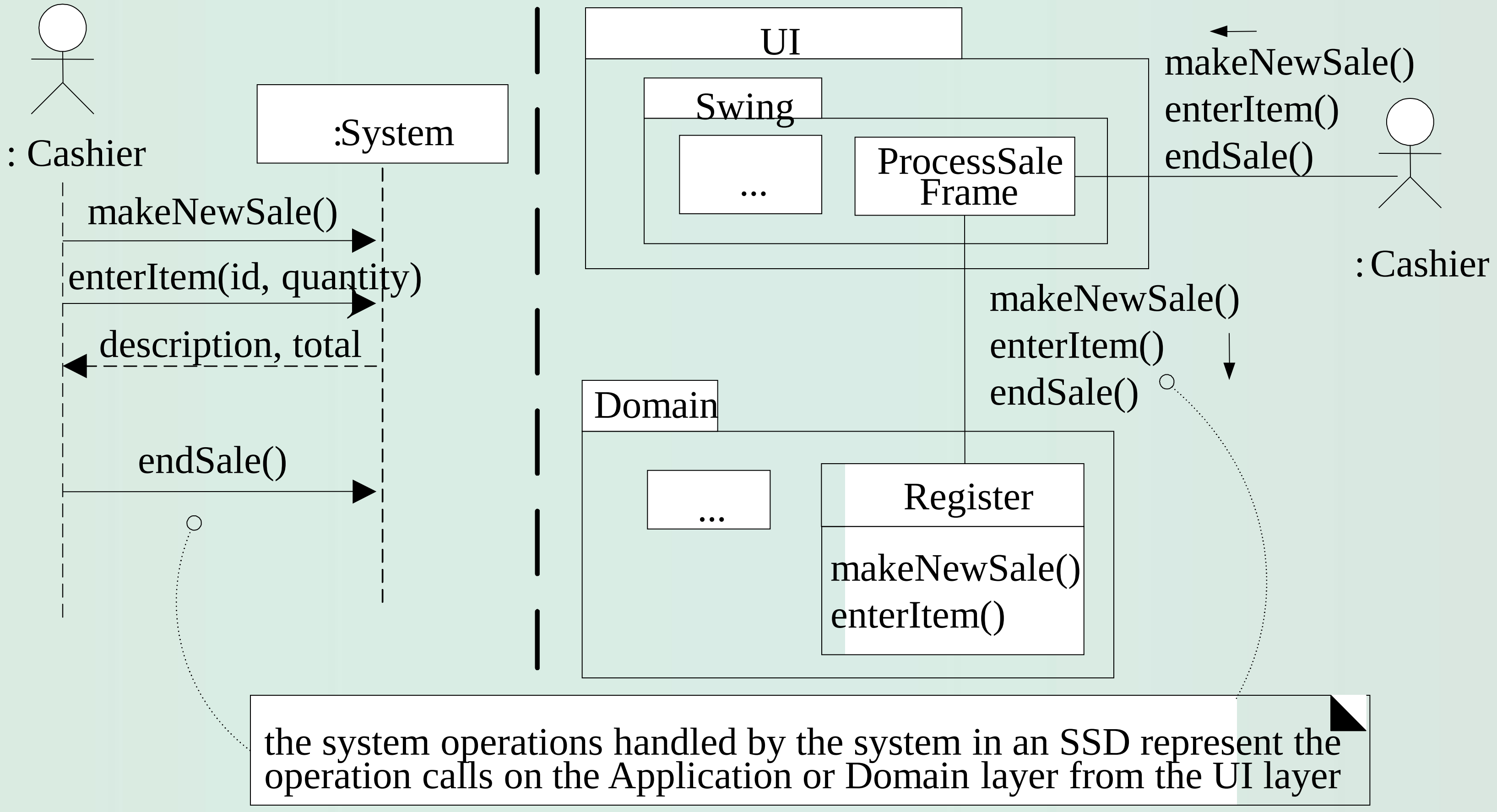
Guideline: Model-View Separation Principle 3

- The motivation for Model-View Separation
  - Support cohesive model definitions that focus on the domain processes, rather than on user interfaces.
  - Allow separate development of the model and user interface layers.
  - Minimize the impact of requirements changes in the interface upon the domain layer.
  - Allow new views to be easily connected to an existing domain layer, without affecting the domain layer.
  - Allow multiple simultaneous views on the same model object, such as both a tabular and business chart view of sales information.
  - Allow execution of the model layer independent of the user interface layer, such as a message-processing or batch-mode system.
  - Allow easy porting of the model layer to another user interface framework.

SSDs, System Operations, and Layers 1

- The SSDs illustrate system operations, hide specific UI objects.
  - The UI layer, object capture these system operation requests, with a rich client GUI or Web page.
  - e.g., makeNewSale and enterItem.
- In a well-designed layered architecture that supports high cohesion and a separation of concerns
  - the UI layer objects forward or delegate the request from the UI layer onto the domain layer for handling.
  - The messages (e.g., enterItem) sent from the UI layer to the domain layer on the SSDs.
  - e.g., Java Swing/GUI window class called ProcessSaleFrame in the UI layer picks up the mouse and keyboard events requesting to enter an item, and then the ProcessSaleFrame object will send an enterItem message on to a software object in the domain layer, such as Register, to perform the application logic.

SSDs, System Operations, and Layers 2



the system operations handled by the system in an SSD represent the operation calls on the Application or Domain layer from the UI layer