

Logical Architecture

Dr. M. Taimoor Khan

Taimoor.khan@nu.edu.pk

Logical Architecture & Layers

- Logical architecture: the large-scale organization of software classes into ***packages***, ***subsystems***, and ***layers***.
 - “**Logical**” because no decisions about how these elements are deployed across different operating system processes or across physical computers in a network
- Layer: a layer is a very **coarse-grained** grouping of classes, packages, or subsystems that have ***cohesive responsibility*** for a major aspect of the system.
 - Layers are organized such that "higher" layers (such as the UI layer) call upon services of "lower" layers, but not normally vice versa.

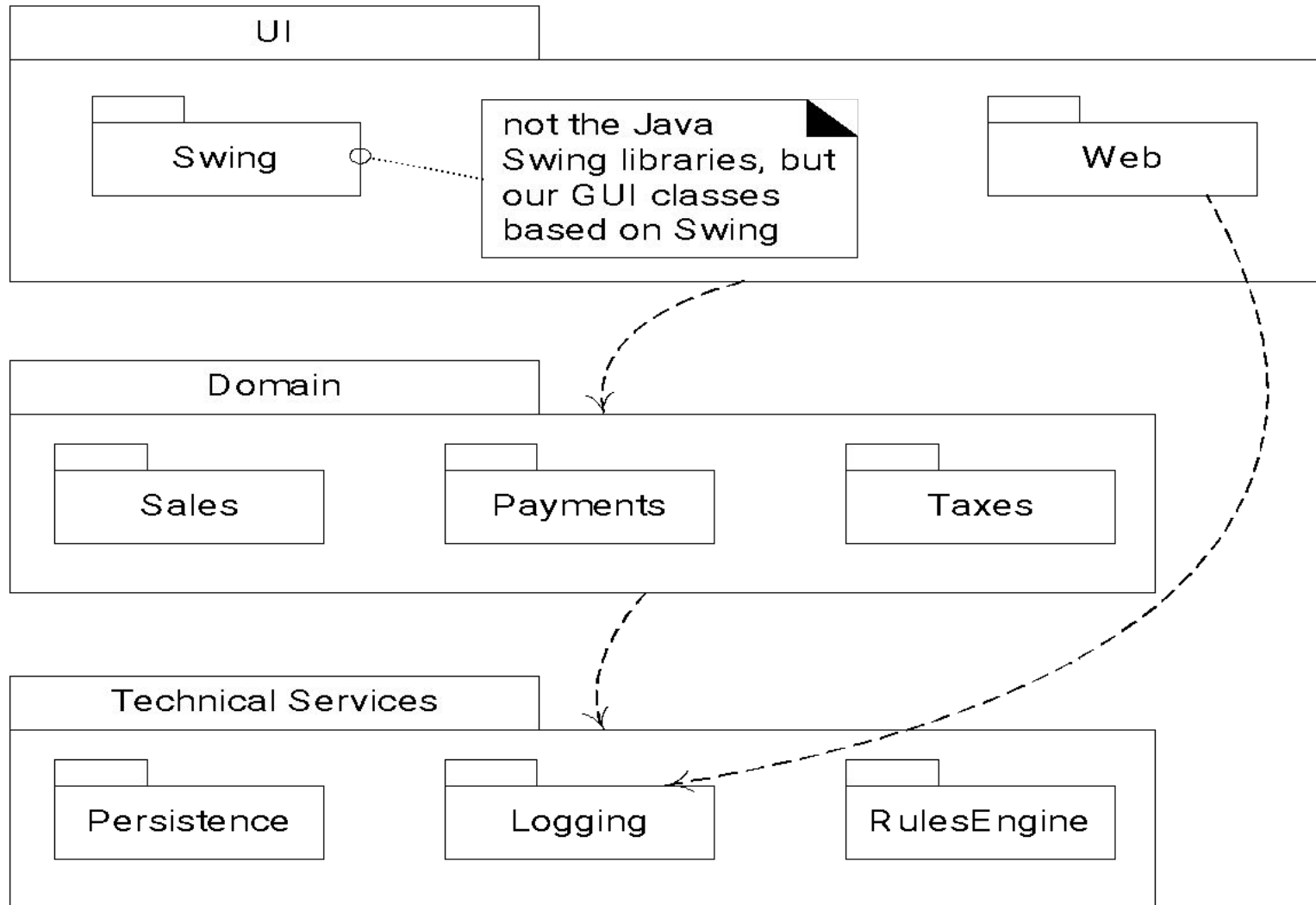
Layers in an OO system include:

- *User Interface:* All activities related to interaction with users such as interfacing, handling user events and triggering the lower level operations.
- *Application Logic and Domain Objects:* software objects representing domain concepts (for example, a software class Sale) that fulfill application requirements, such as calculating a sale total.
- *Technical Services:* general purpose objects and subsystems that provide supporting technical services, such as ***interfacing with a database or error logging***. These services are usually application-independent and reusable across several systems.

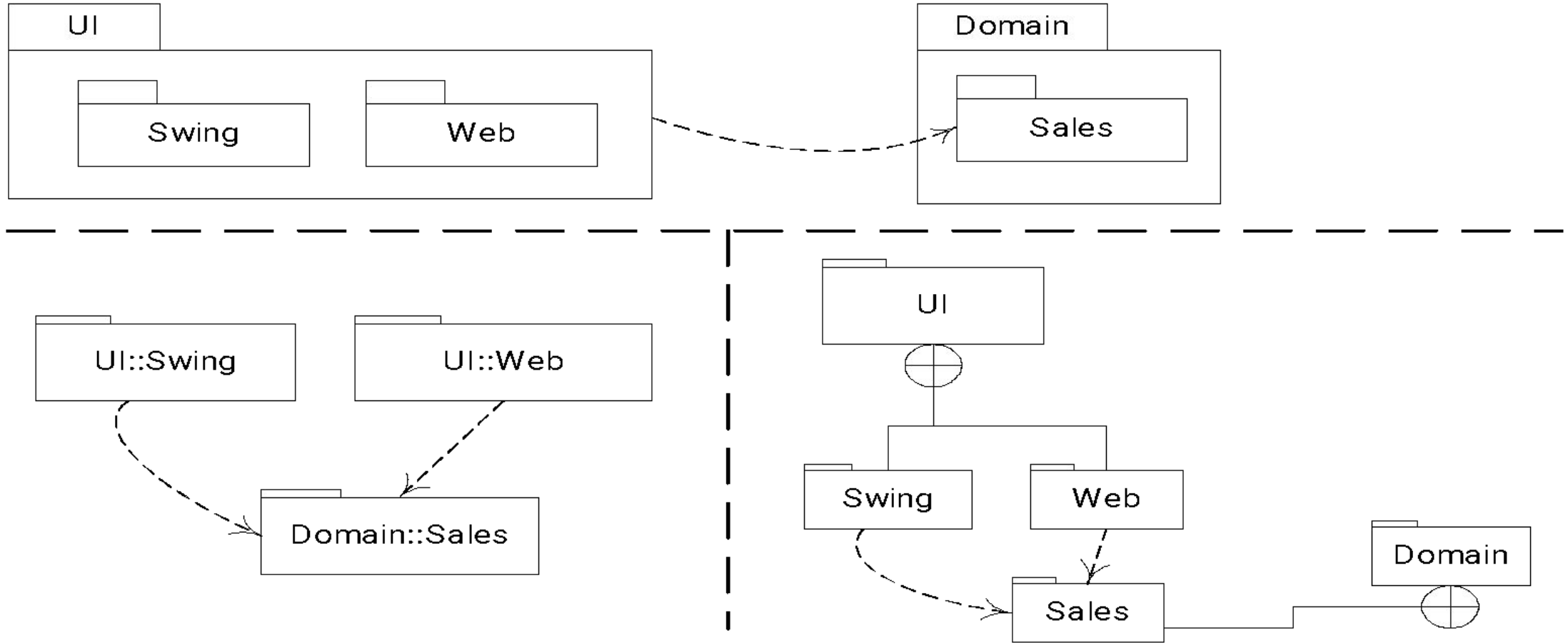
Layered Architectures

- Relationships between layers:
 - **Strict** layered architecture: a layer only calls upon services of the layer directly below it e.g. TCP/IP Stack
 - **Relaxed** layered architecture: a higher layer calls upon several lower layers.

Layers shown with UML package diagram.



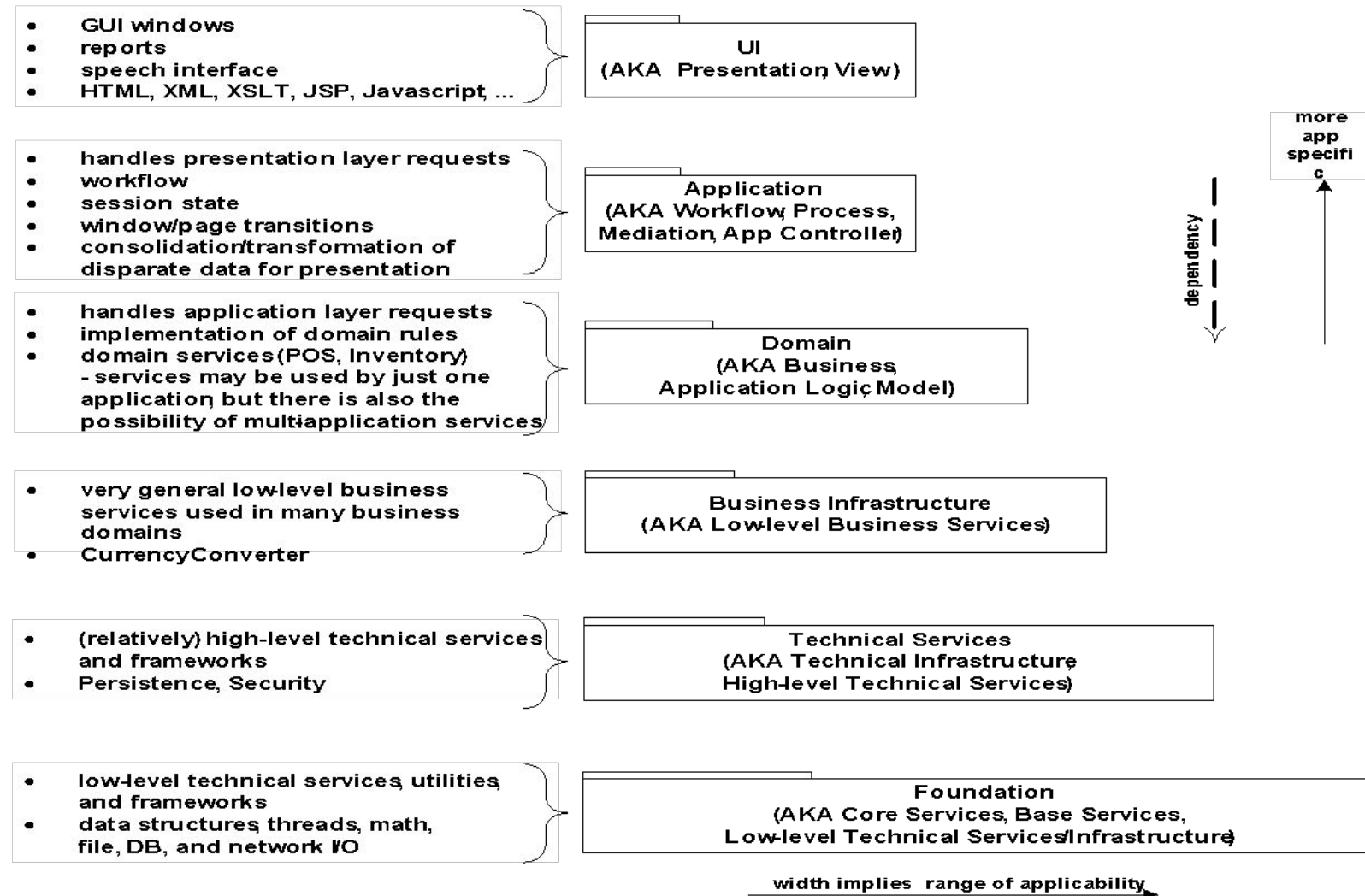
Various UML notations for package nesting



Design with Layers

- Organize the large-scale logical structure of a system into discrete layers of distinct, related responsibilities.
 - Cohesive separation of concerns.
 - Lower layers are general services.
 - Higher layers are more application-specific.
- Collaboration and coupling is from higher to lower layers.
 - Lower-to-higher layer coupling is avoided.

Common Layers in an Information System Logical Architecture



Benefits of a Layered Architecture

- Separation of concerns:

E.g., UI objects should not do application logic (a window object should not calculate taxes) nor should a domain layer object create windows or capture mouse events.

- Reduced coupling and dependencies.
- Improved cohesion.
- Increased potential for reuse.
- Increased clarity.

Benefits of a Layered Architecture (con..)

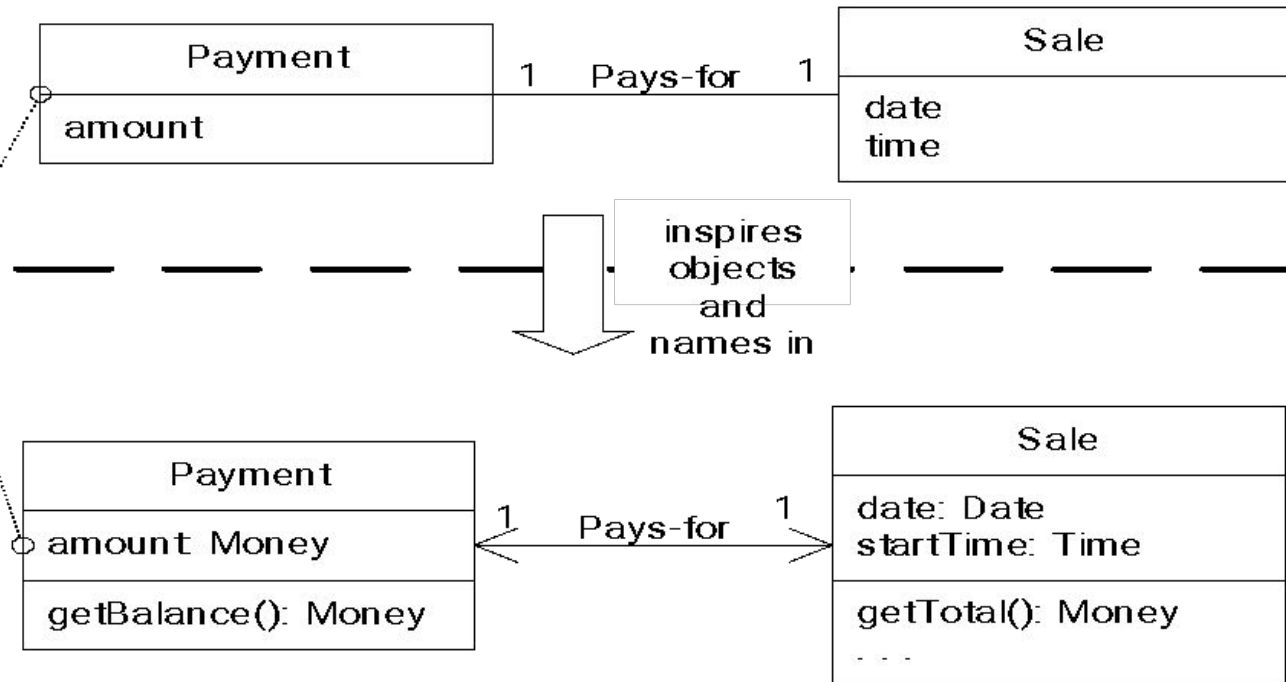
- Related complexity is encapsulated and decomposable.
- Some layers can be replaced with new implementations.
- Lower layers contain reusable functions.
- Some layers can be distributed.
 - Especially Domain and Technical Services.
- Development by teams is aided by logical segmentation.

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.

UP Domain Model Stakeholder's view of the noteworthy concepts in the domain.

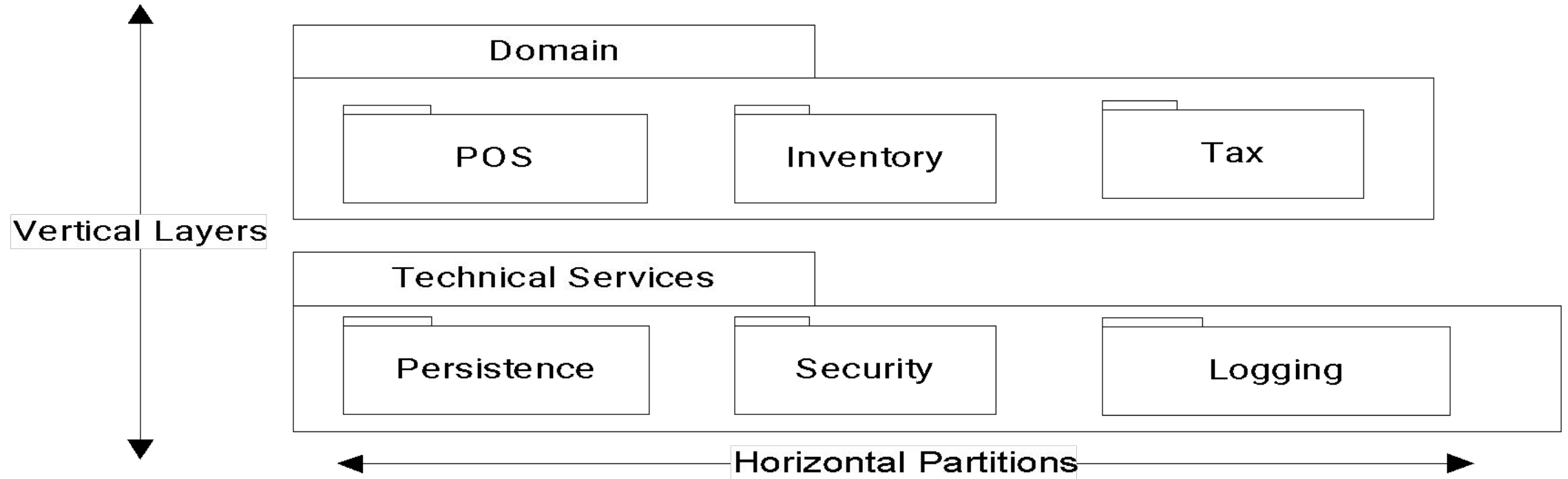


Domain layer of the architecture in the UP Design Model

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.

Layers vs. Partitions



The Model-View Separation Principle

- Model: the domain layer of objects.
- View: user interface (UI) objects.
- Model objects should not have direct knowledge of view objects.
 - Do not connect or couple non-UI objects directly to UI objects.
 - E.g., don't let a *Sale* object have a reference to a Java Swing *JFrame* window object.
 - Do not put application logic in a UI object.
 - UI objects should receive UI events and delegate requests for application logic to non-UI objects.

The Observer Pattern

- If model (domain) objects do not have direct knowledge of view (UI) objects, how can a *Register* or *Sale* object get a window to refresh its display when a total changes?
- The *Observer* pattern allows domain objects to send messages to UI objects viewed only in terms of an **interface**.
 - E.g., known not as concrete window class, but as implementation of *PropertyListener* interface.
- Allows replacement of one view by another.