Logical Architecture

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.

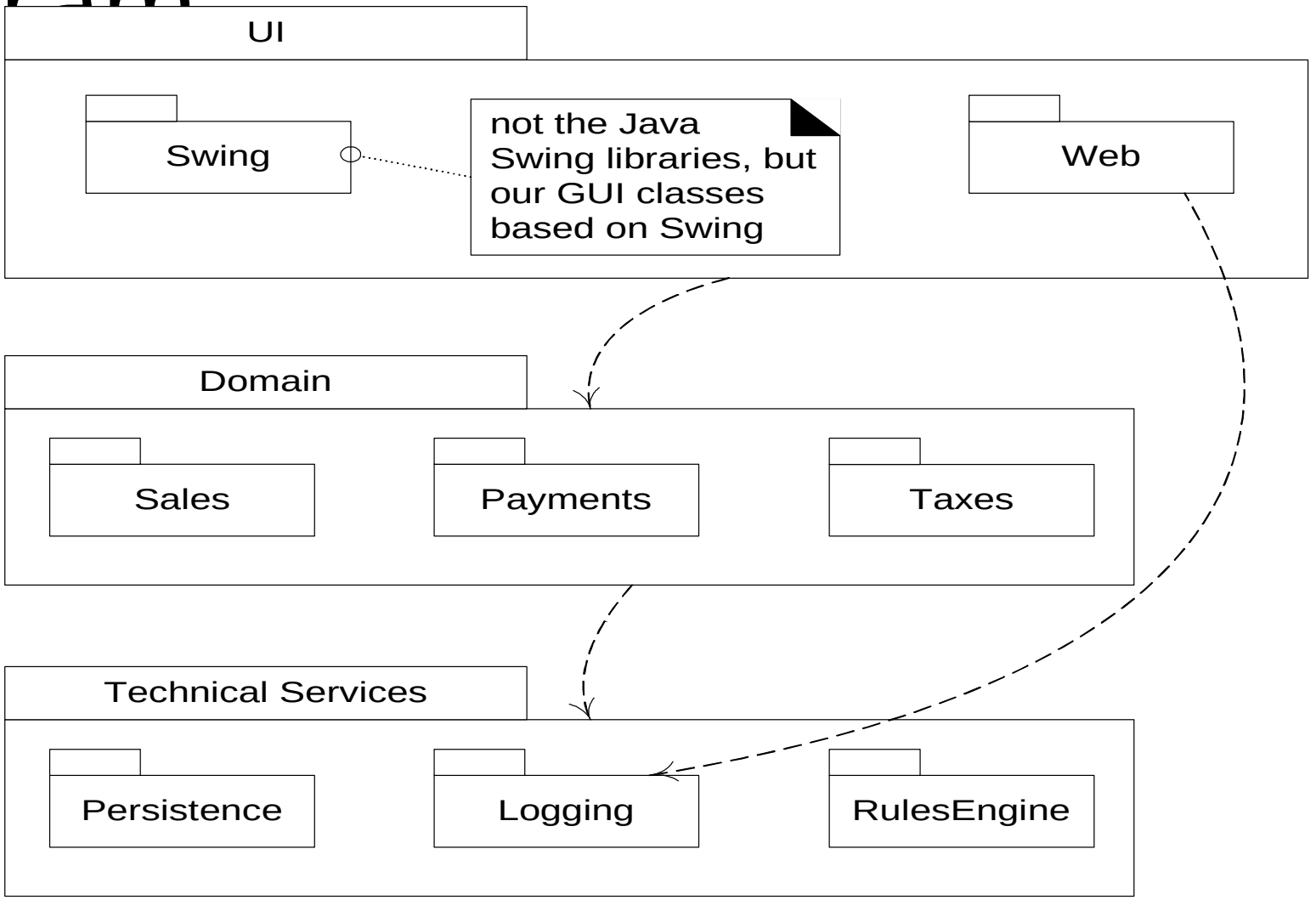
Layers in an OO system include:

- *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.

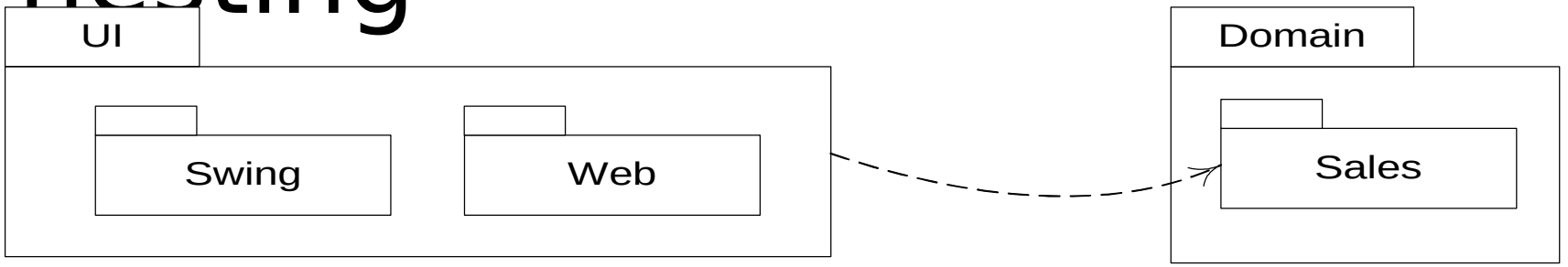
Layered Architectures

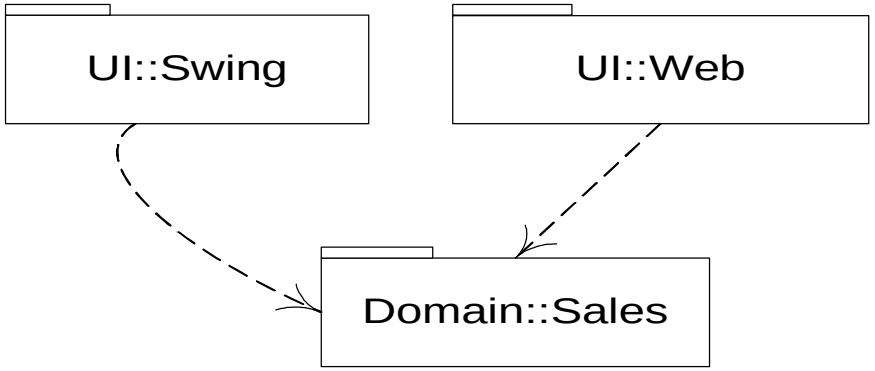
- Relationships between layers:
  - **Strict** layered architecture: a layer only calls upon services of the layer directly below it e.g. TCP/IP Stack
  - **Relaxed** layered architecture: a higher layer calls upon several lower layers.

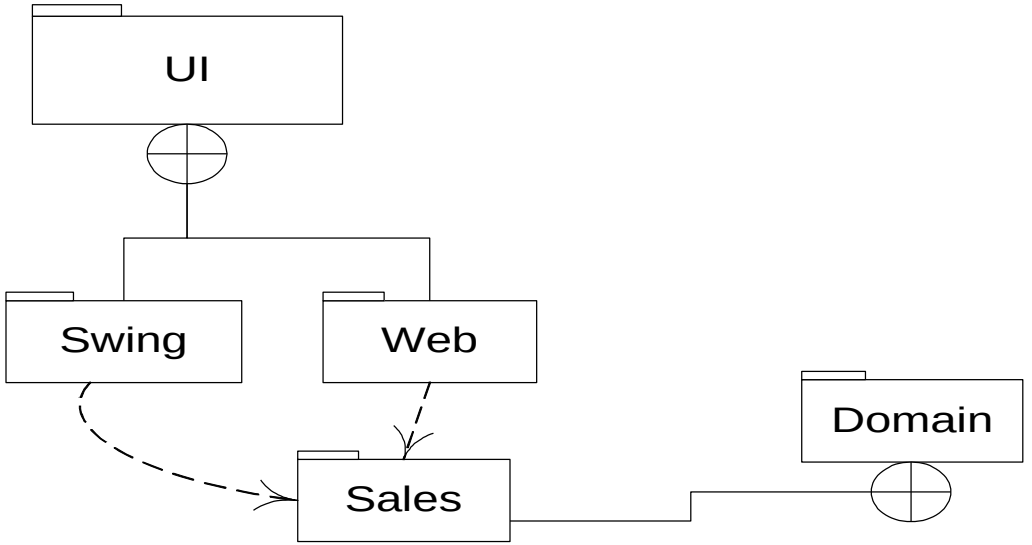
Layers shown with UML package diagram



Various UML notations for package nesting



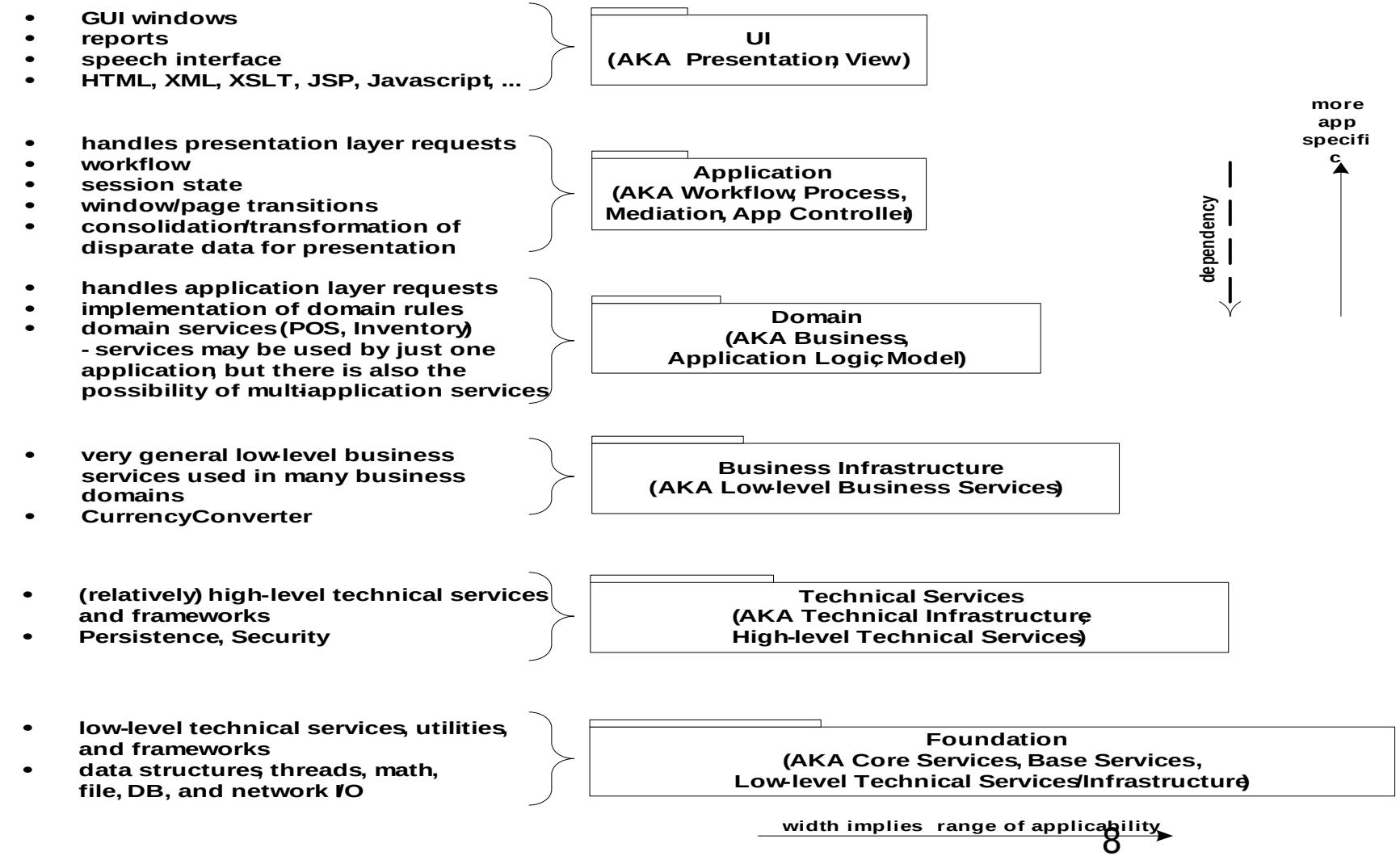




Design with Layers

- 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.

Designing the Domain Layer

How do we design the application logic with objects?

- Create software objects with names and information similar to the real-world domain.
- Assign **application logic** responsibilities to these **domain objects**.
  - E.g., a *Sale* object is able to calculate its total.

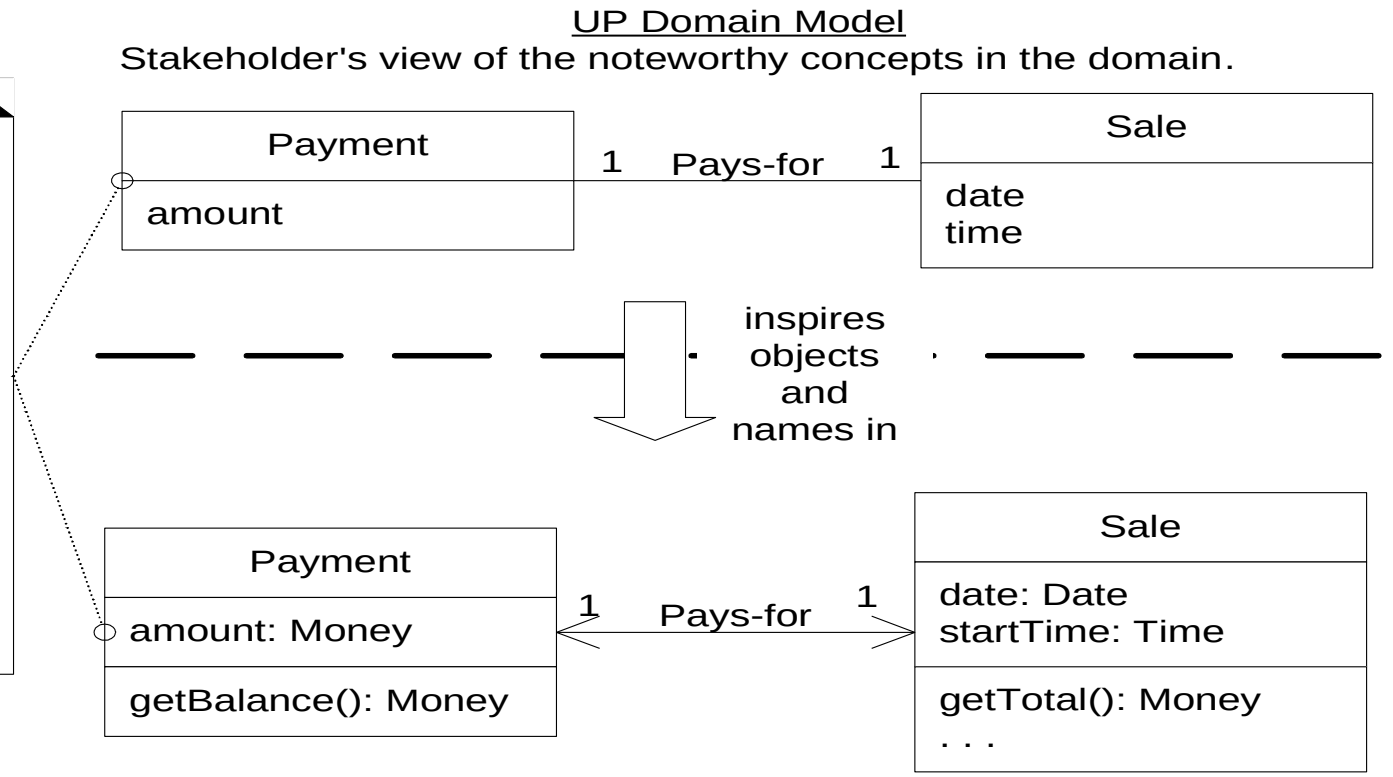
The application logic layer is more accurately called a **domain layer** when designed this way.

Domain Model Related to Domain Layer

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.

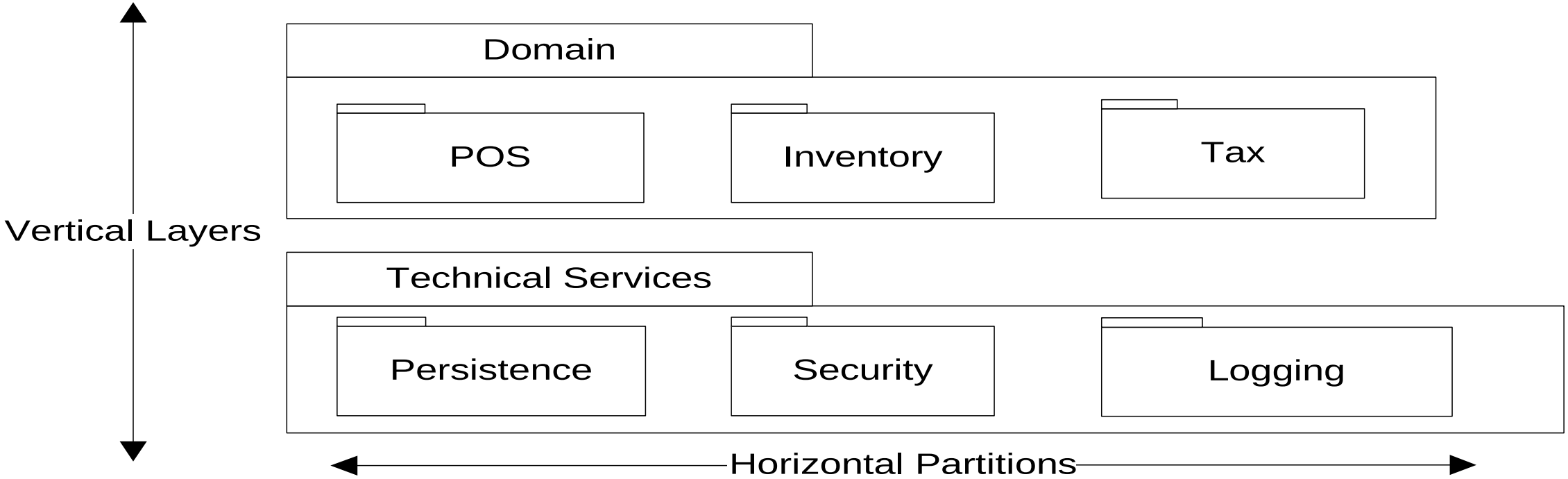


Domain layer of the architecture in the UP Design Model

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

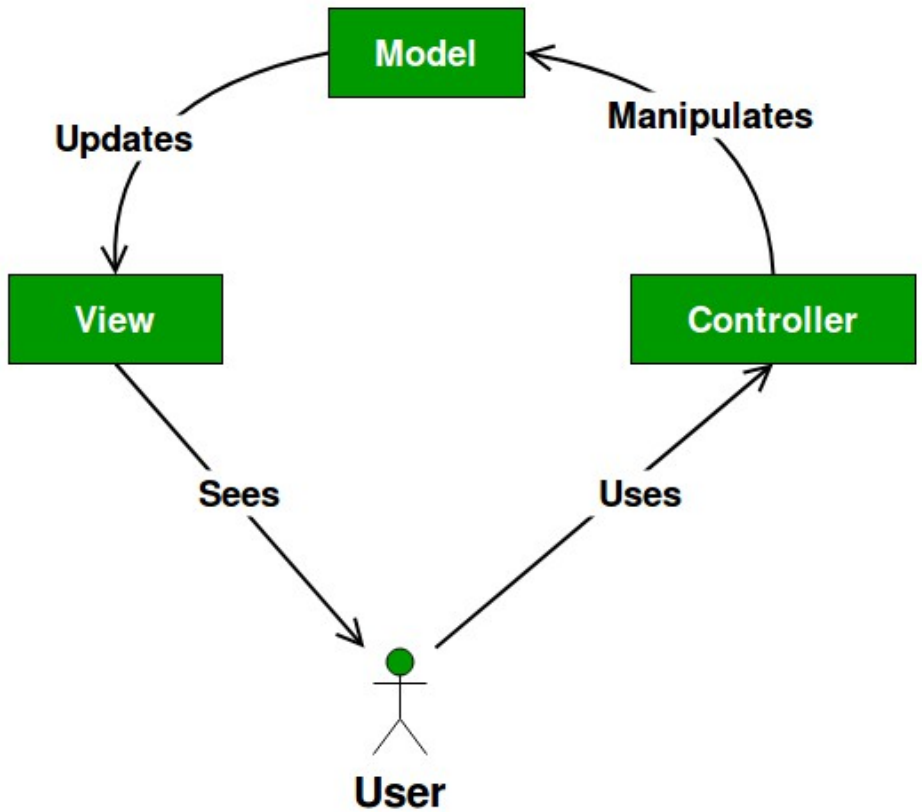
Therefore, the representational gap between how stakeholders conceive the domain, and its representation in software, has been lowered.

Layers vs. Partitions

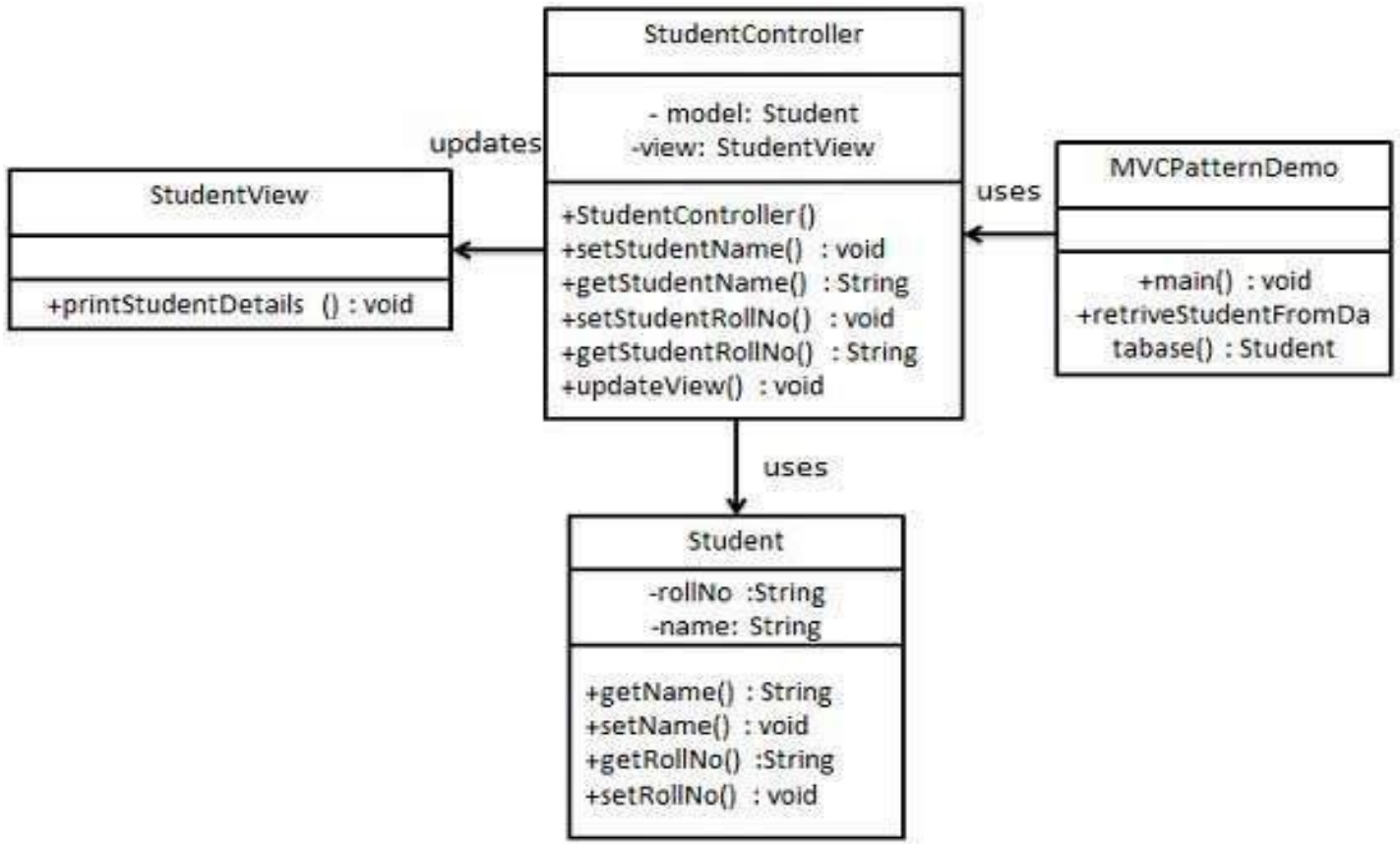


MVC

- The **Model View Controller** (MVC) design pattern specifies that an application consist of a data model, presentation information, and control information. The pattern requires that each of these be separated into different objects.
- MVC is more of an architectural pattern, but not for complete application. MVC mostly relates to the UI / interaction layer of an application. You're still going to need business logic layer, maybe some service layer and data access layer.


Model
Updates
Manipulates
View
Controller
Sees
Uses
User

- The **Model** contains only the pure application data, it contains no logic describing how to present the data to a user.
- The **View** presents the model's data to the user. The view knows how to access the model's data, but it does not know what this data means or what the user can do to manipulate it.
- The **Controller** exists between the view and the model. It listens to events triggered by the view (or another external source) and executes the appropriate reaction to these events. In most cases, the reaction is to call a method on the model. Since the view and the model are connected through a notification mechanism, the result of this action is then automatically reflected in the view.


StudentController
- model: Student
-view: StudentView
+StudentController()
+setStudentName() : void
+getStudentName() : String
+setStudentRollNo() : void
+getStudentRollNo() : String
+updateView() : void
updates
StudentView
+printStudentDetails () : void
uses
MVCPatternDemo
+main() : void
+retriveStudentFromDatabase() : Student
uses
Student
-rollNo :String
-name: String
+getName() : String
+setName() : void
+getRollNo() :String
+setRollNo() : void

Messages from UI layer to domain layer