Chapter 13

Logical Architecture and UML Package Diagrams

Logical Architecture and Layers

- Logical architecture: the large-scale organization of software classes into packages, subsystems, and layers.
  - "Logical" because no decisions about deployment are implied. (See Chap. 37.)
- Layer: a very coarse-grained grouping of classes, packages, or subsystems that has cohesive responsibility for a major aspect of the system.

Layered Architectures

- Typical layers in an OO system:
  - User Interface
  - Application Logic and Domain Objects
  - Technical Services
    - Application-independent, reusable across systems.
- Relationships between layers:
  - Strict layered architecture: a layer only calls upon services of the layer directly below it.
  - Relaxed layered architecture: a higher layer calls upon several lower layers.

Fig. 13.2 Layers shown with UML package diagram.

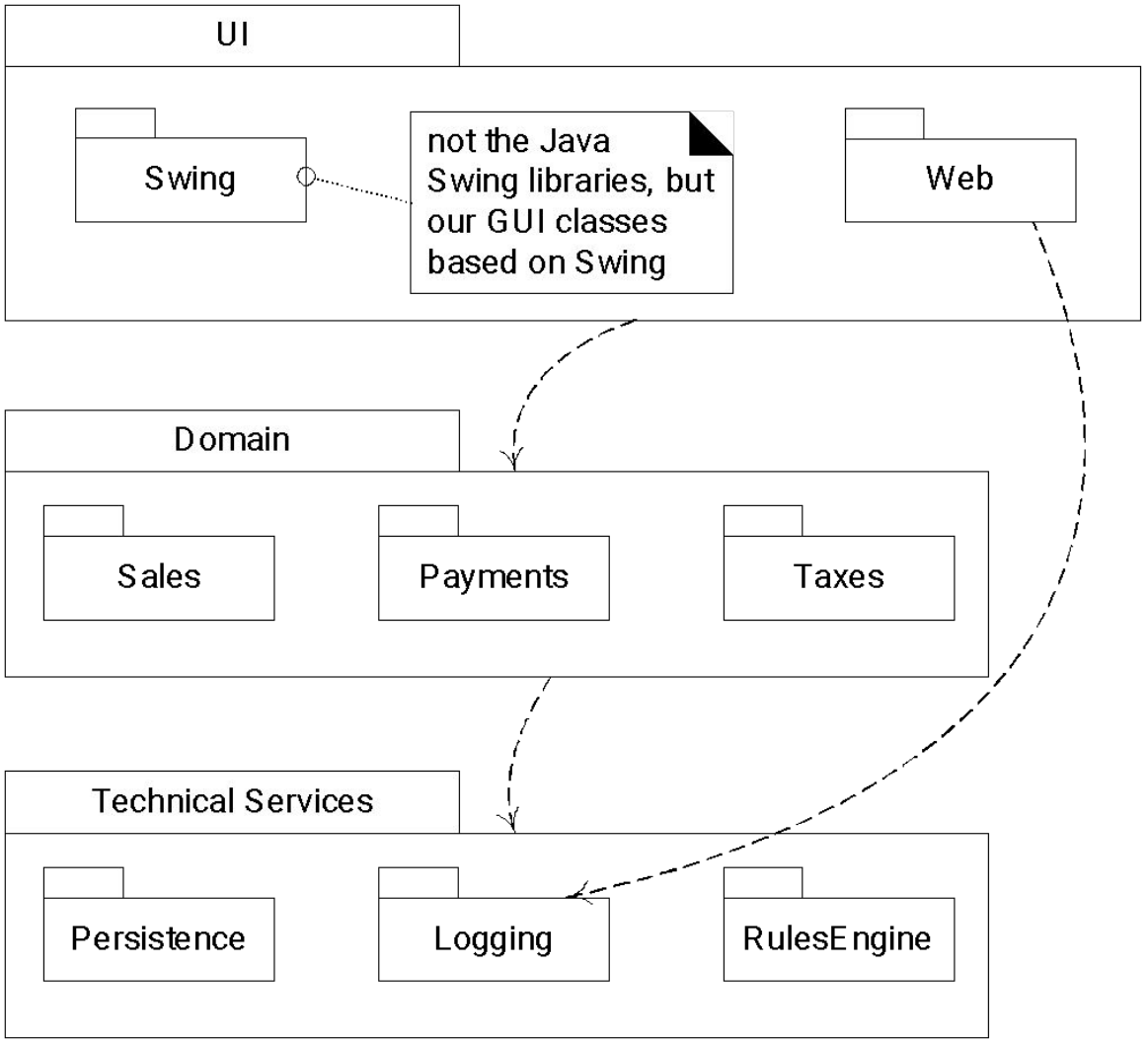
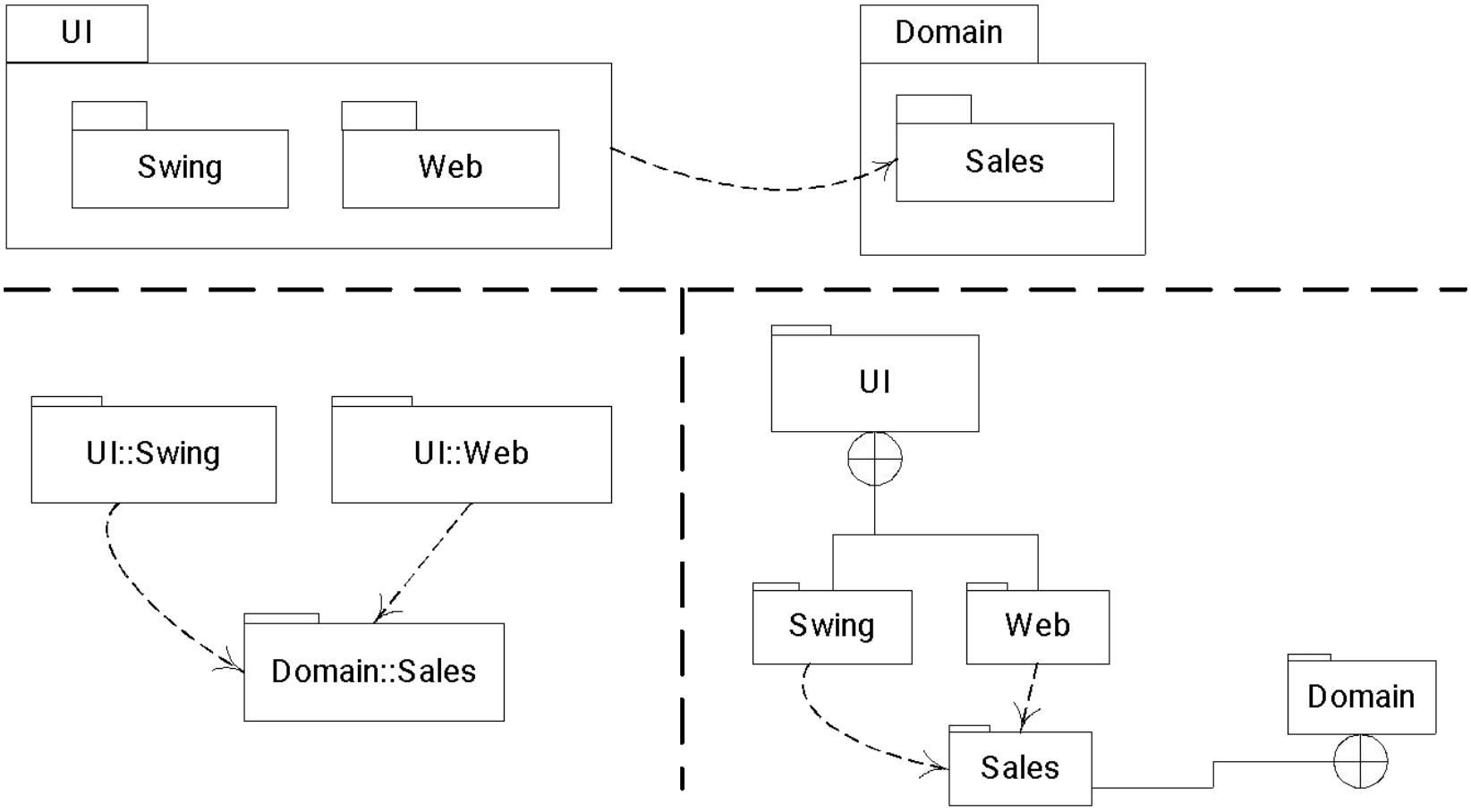


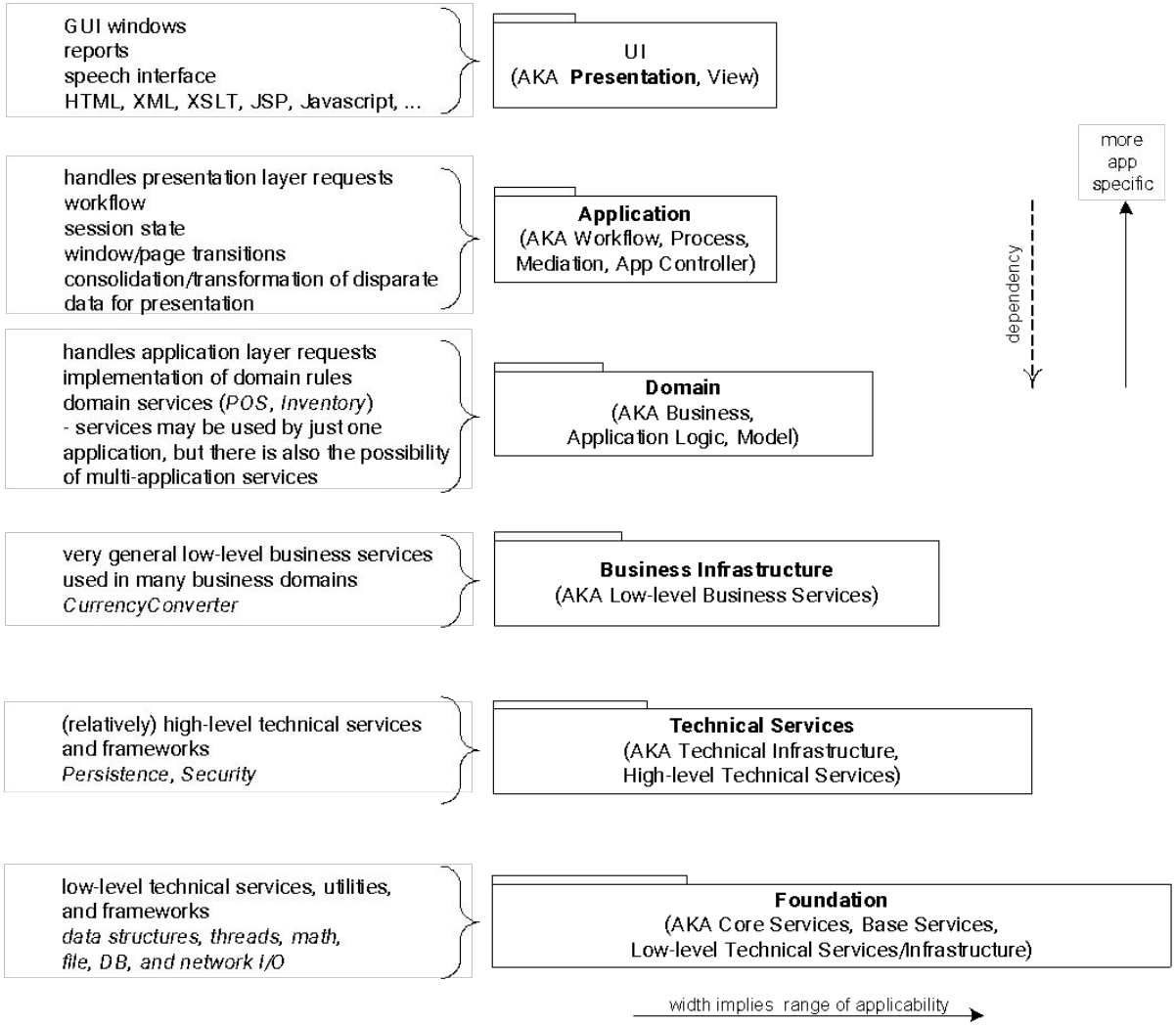
Fig. 13.3 Various UML notations for package nesting



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.

Fig. 13.4 Common layers in an IS 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, continued

- 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.

Fig. 13.5 Domain Model Related to Domain Layer

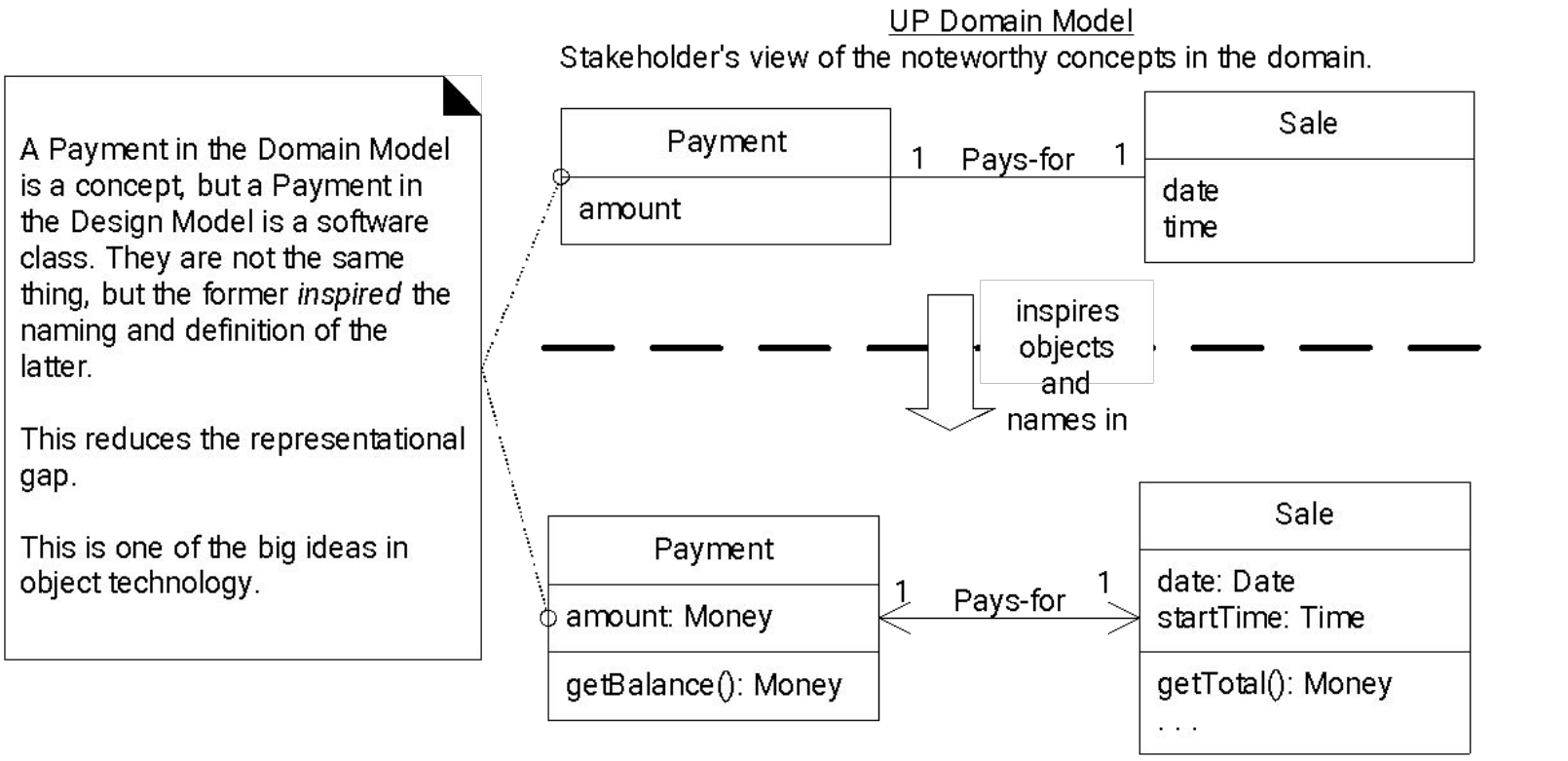


Fig. 13.6 Layers vs. Partitions

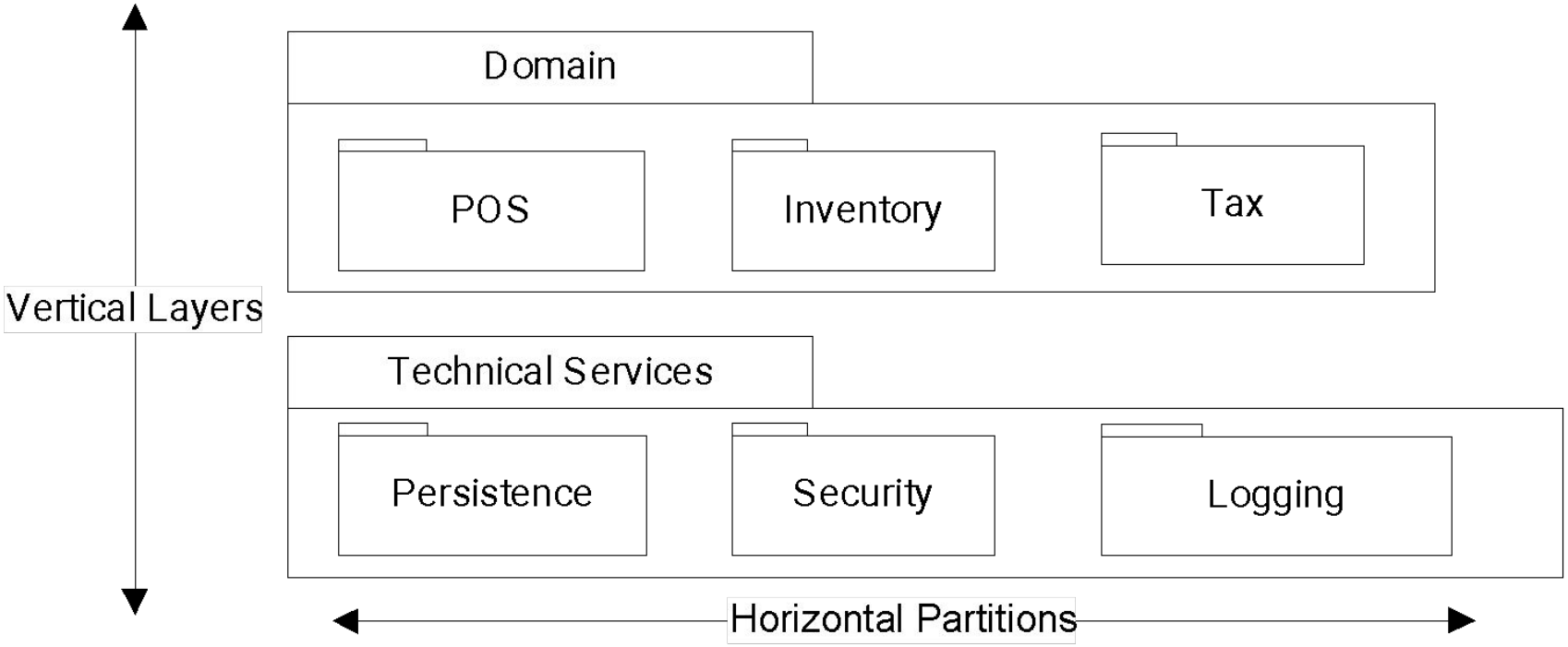
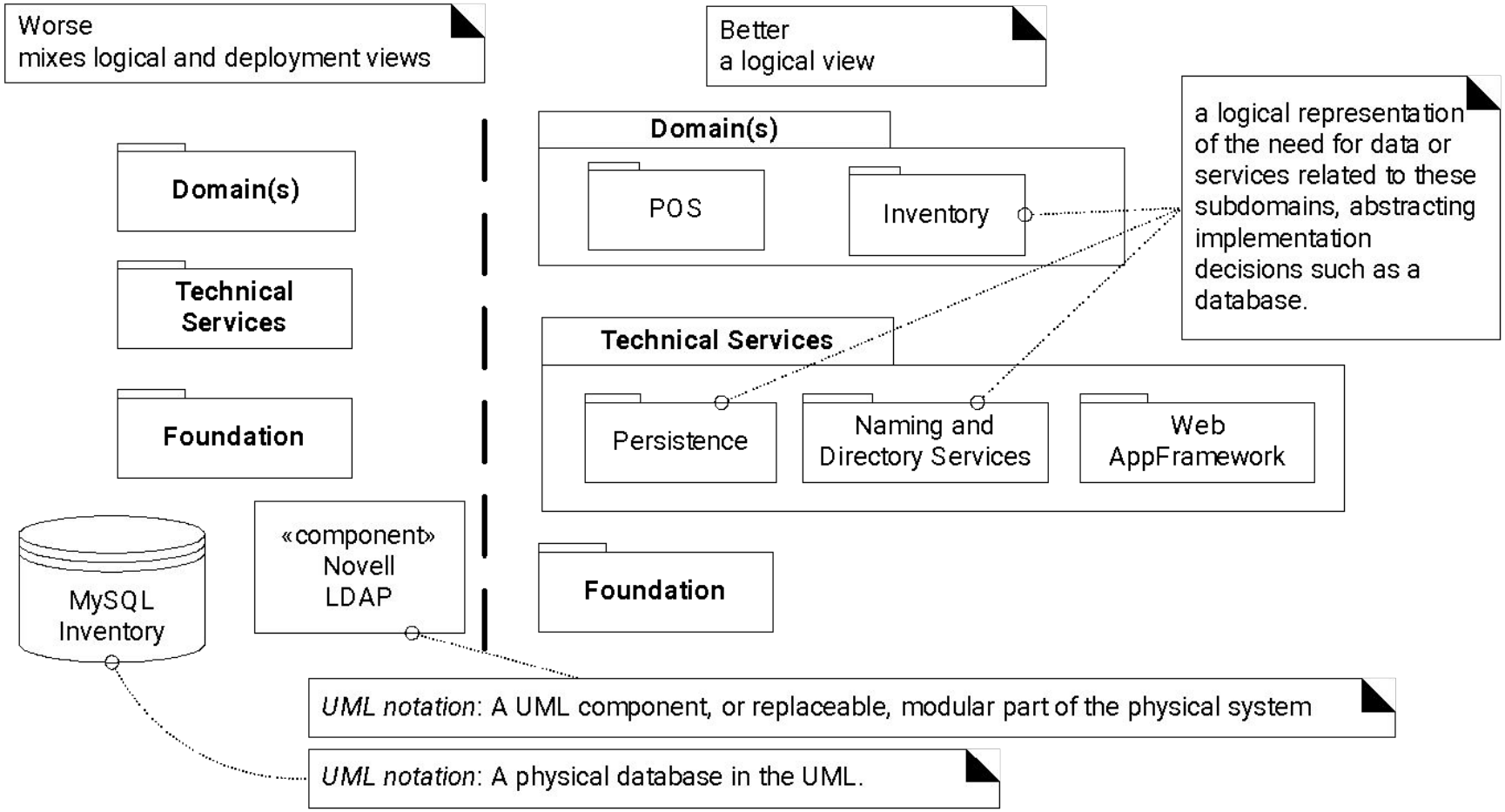


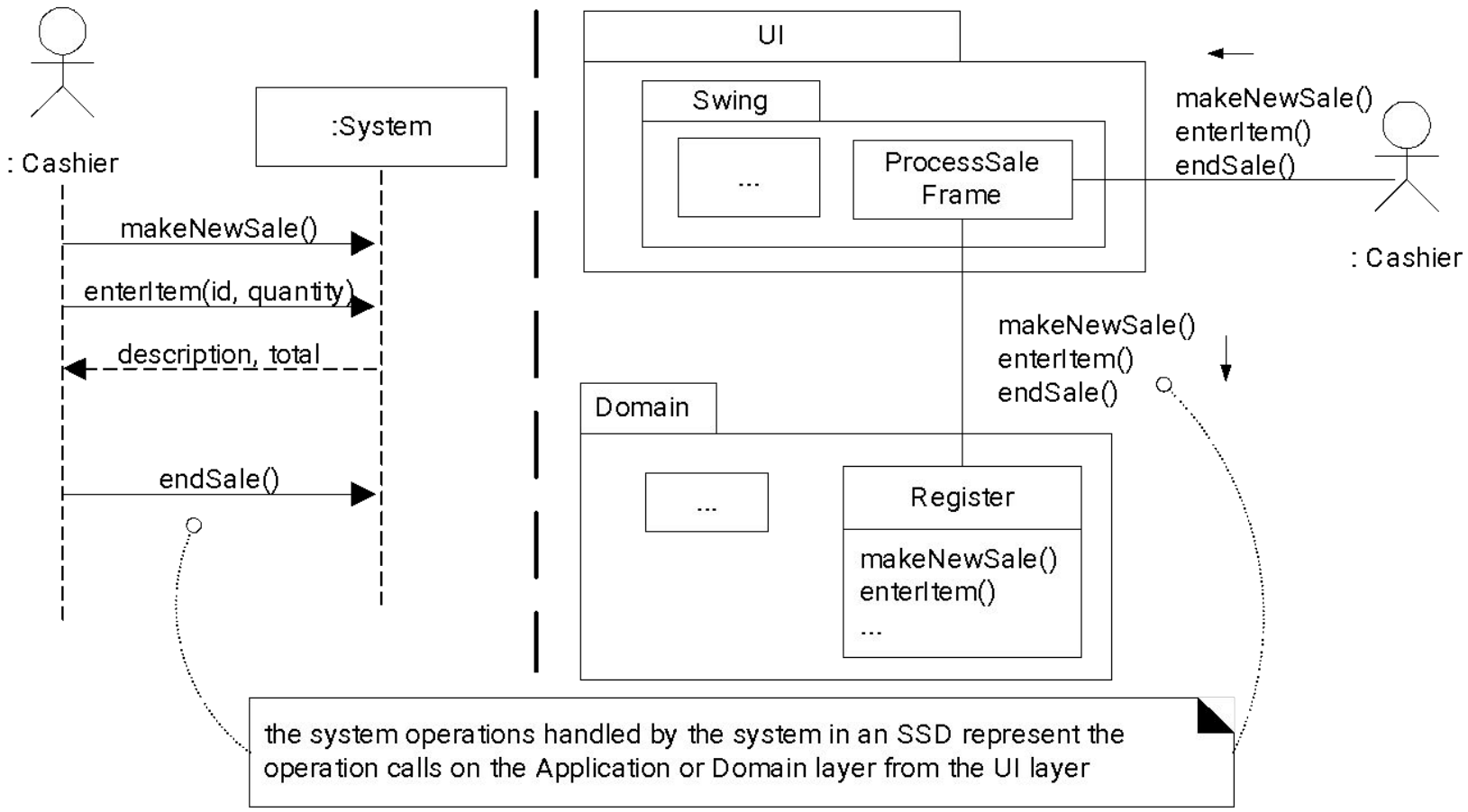
Fig. 13.7 Don't mix logical and deployment views.



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.

Fig. 13.8 Messages from UI layer to domain layer



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 (p. 463) 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.