

# Chapter 13

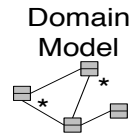
Starting Design:  
Logical Architecture  
and  
UML Package Diagrams

# Logical Architecture

- Logical architecture: Large-scale organization of the software classes into
  - packages (or namespaces)
  - subsystems
  - layers
- Distinct from “deployment architecture”
  - No decision about how the elements are deployed
    - to different OS processes
    - across physical computers in a network
- A layer: A coarse-grained grouping of classes, packages or subsystems that together have responsibility for one major aspect of a system
- Examples of layers:
  - UI layer
  - Application logic and domain objects layer
  - Technical services (interfacing with a database, error logging)
    - Typically application-independent and reusable

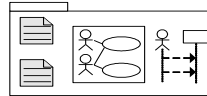
## Sample UP Artifact Relationships

### Business Modeling



### Requirements

#### Use-Case Model



#### Vision



#### Supplementary Specification



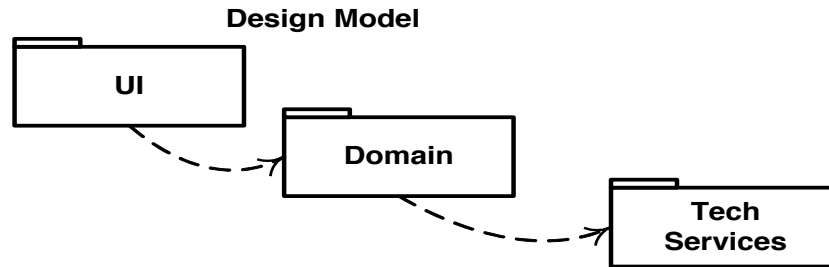
#### Glossary



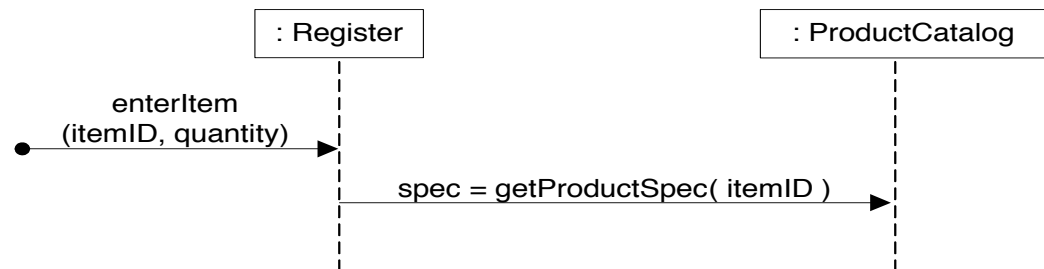
The logical architecture is influenced by the constraints and non-functional requirements captured in the Supp. Spec.

### Design

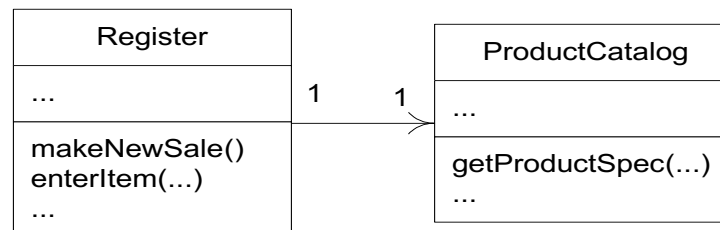
**package diagrams of the logical architecture (a static view)**



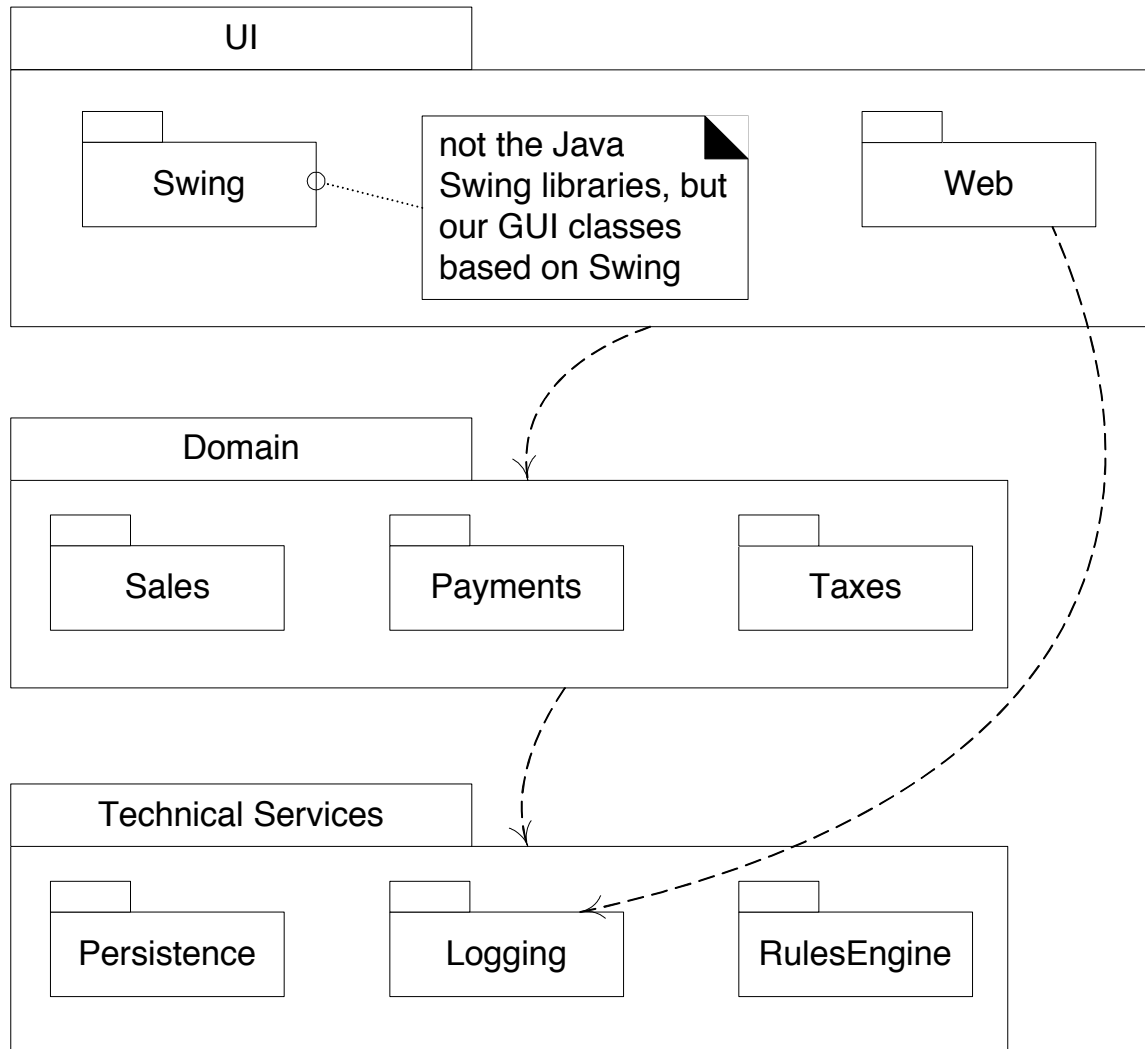
**interaction diagrams (a dynamic view)**



**class diagrams (a static view)**



**Fig. 13.2**



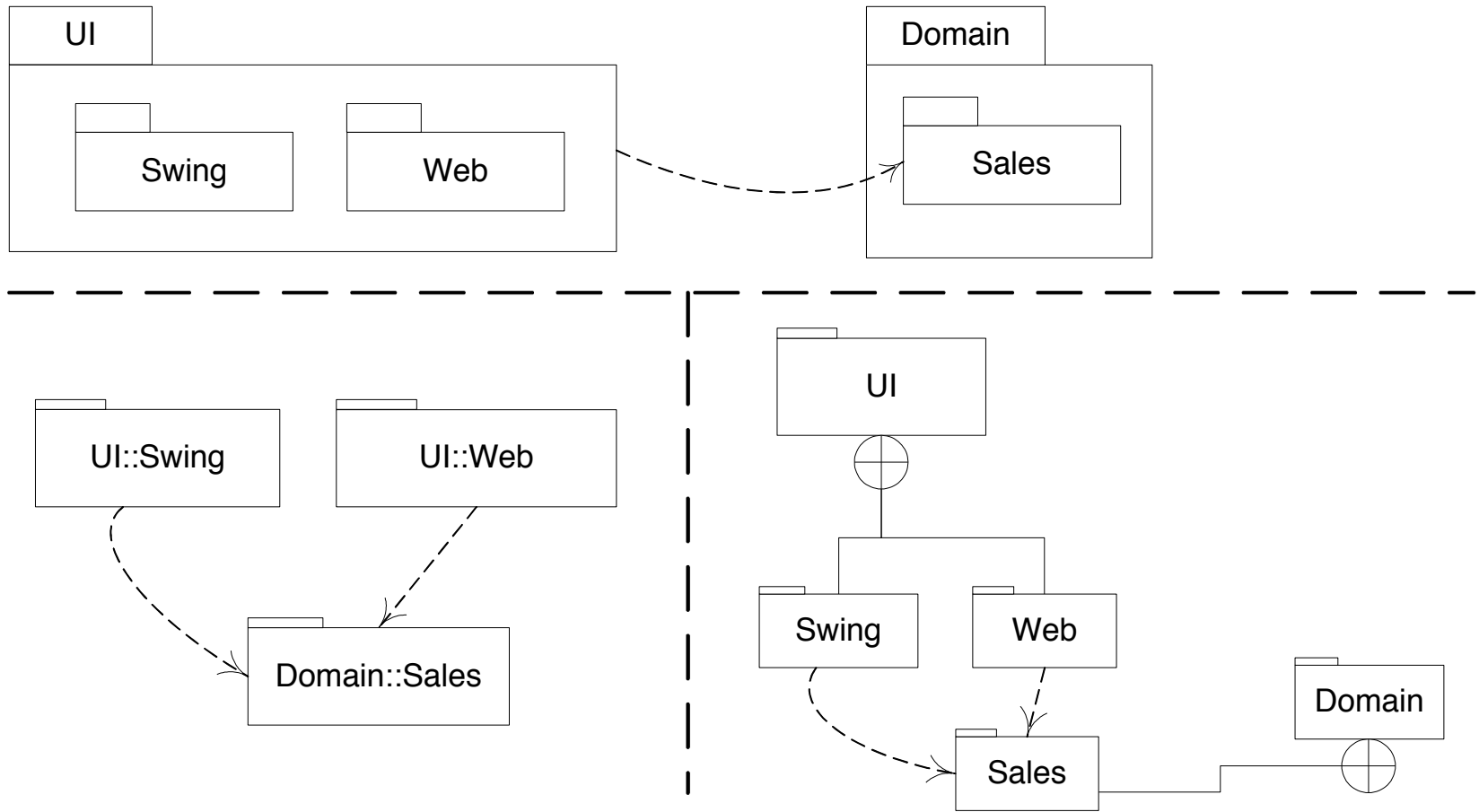
# Architecture

- Strict layered architecture: Each layer only calls upon services of the layer directly below it.
  - Common in network protocol stacks
  - Not so common in information systems
- You do NOT have to use a layered architecture
  - But it is very common to do so
- What is architecture then?
  - The set of significant decisions about
    - the organization of a software system
      - hierarchical composition of smaller subsystems to progressively larger subsystems
      - the selection of structural elements and interfaces
    - the style that guides this organization
- Architecture: Related to large scale, not implementation details.

# UML Package Diagrams

- UML Package Diagrams:
  - Used to illustrate the logical architecture of a system
    - Layers, subsystems, Java packages
  - Provides a way to group elements
    - Different from (more general than) a Java package
    - Can group anything
      - Classes, other packages, diagrams, use cases, ...
    - Nesting packages is very common

# Alternative UML Package Diagram Notations



## UML Packages

- A package represents a “namespace”
  - Example: A Date class can be defined in two packages
    - Fully qualified names: `java::util::Date`



## Layers: Why?

- Two key architectural principles
  - Separation of concerns
  - Maintaining high cohesion
- Separation of concerns:
  - Discrete layers of distinct, related responsibilities
  - Clean cohesive separation of duties:
    - Lower layers: Low-level, general services
    - Higher layers: More application-specific services
  - Easier to define boundaries for different developers
- Collaboration and coupling from higher to lower layers

## Layers: Why? (continued)

- Limiting dependencies between subsystems:
  - Source code changes ripple throughout the system if many parts are tightly coupled
  - Example: If application logic is intertwined with UI,
    - it cannot be distributed to another physical node
    - It cannot be used with a different UI
- General technical services and business logic can be re-used, replaced or moved to another physical node

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

**UI**  
(AKA **Presentation**, View)

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

**Application**  
(AKA Workflow, Process, Mediation, App Controller)

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

**Domain**  
(AKA Business, Application Logic, Model)

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

**Business Infrastructure**  
(AKA Low-level Business Services)

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

**Technical Services**  
(AKA Technical Infrastructure, High-level Technical Services)

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

**Foundation**  
(AKA Core Services, Base Services, Low-level Technical Services/Infrastructure)

# Typical set of layers for information systems

more  
app  
specific  
↑  
dependency  
↓

width implies range of applicability →

# Mapping Code Organization to Layers and UML Packages

```
//---- PRESENTATION
```

```
com.foo.nextgen.ui.swing  
com.foo.nextgen.ui.text
```

```
//---- DOMAIN
```

```
    // packages relatively specific to the NextGen project  
com.foo.nextgen.domain.sales com.foo.nextgen.domain.pricing  
com.foo.nextgen.domain.serviceaccess  
com.foo.nextgen.domain.posruleengine
```

```
    // packages that can easily be designed as  
    // multi-application common business services  
com.foo.domain.inventory  
com.foo.domain.creditpayment
```

```
// --- TECHNICAL SERVICES
```

```
    // our team creates  
com.foo.service.persistencelite
```

```
    // third party  
org.apache.log4j  
org.apache.soap.rpc  
jess
```

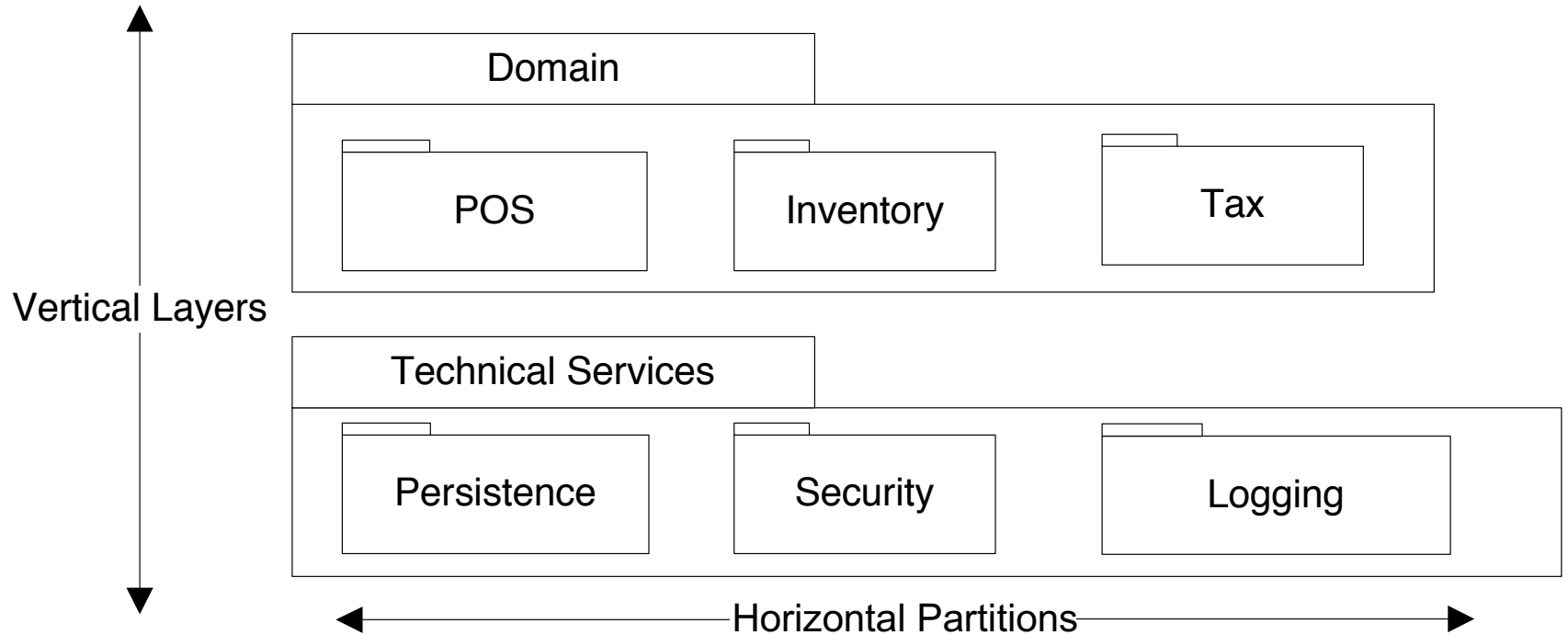
```
// --- FOUNDATION
```

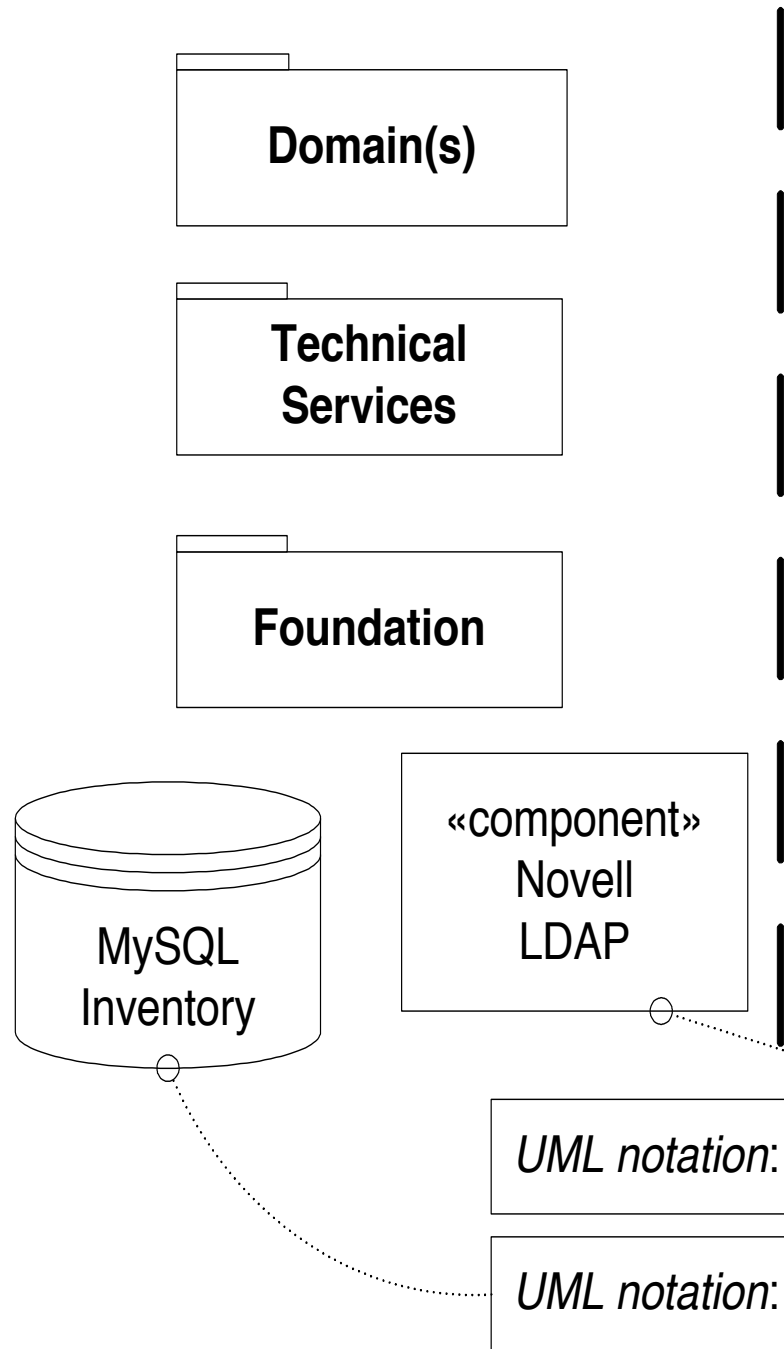
```
    // our team creates  
com.foo.util  
com.foo.stringutil
```

## Some more issues, terms

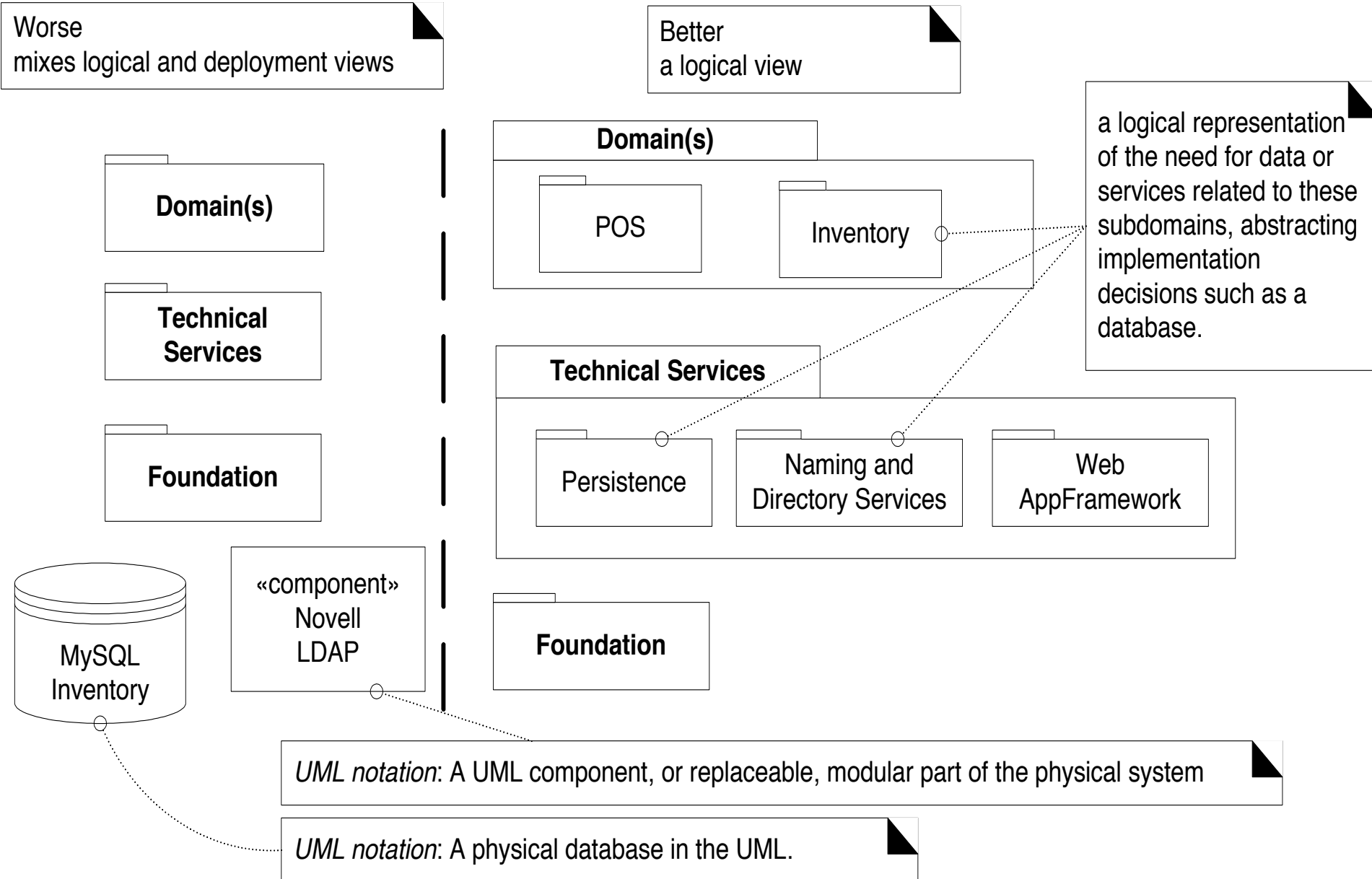
- What if the organization of code into packages changes later on?
  - Follow a good package naming convention
  - Use CASE tools to reverse-engineer code into UML packages
- What is a tier?
  - Originally, it meant a logical layer
  - Common usage today: Physical processing node (or cluster of nodes)
- What is a partition?
  - Division into relatively parallel subsystems for a layer

# Layers vs. Partitions





# Don't mix physical implementation components and logical architecture components





# The Model-View Separation Principle

- Old terminology
  - Model: Domain layer (application logic)
  - View: UI objects (windows, web pages, reports, ...)
- Model-view separation:
  - Do not put application logic (such as tax calculation code) in the UI objects
    - UI objects should only
      - initialize UI elements,
      - receive UI events (mouse click, etc.)
      - delegate requests for application logic on to non-UI objects
  - Do not connect non-UI objects to UI objects
  - Example: a Sale object should not have a reference to a JFrame window object
    - Why not?
    - What if we want to use the application logic with different windows or multiple views?

## What if domain objects need to notify the UI?

- A relaxation of the model-view principle:  
the Observer pattern
  - Domain objects send messages to UI objects  
viewed only in terms of an interface
  - Example: PropertyListener interface
    - Domain object is aware of existence of object  
implementing PropertyListener
      - But not of particular object
    - Notification happens using interface methods

## Motivation for the model-view principle

- Domain objects focus on domain processes
  - Not on user interfaces
- Domain objects and UI can be developed separately
- Effect of requirements changes in one component to the other one minimized
- New views (UIs) connected to existing domain layer
- Multiple simultaneous views (e.g. a GUI and a text-based interface) on the same model object
- Domain layer operable without needing UI layer to be on
- Basically, modularity.

# Relationship between System Sequence Diagrams System Operations and Layers

- Messages on SSDs: Messages sent from UI layer to the domain layer

