Software Architecture (UP Elaboration Phase)

Marek Rychlý
rychly@fit.vutbr.cz

Brno University of Technology
Faculty of Information Technology
Department of Information Systems

Information Systems Analysis and Design (AIS)
7 November 2019

Outline

Software Architecture

Definition (Architecture by ISO/IEC/IEEE 24765:2010)

A fundamental organization of a system embodied in its components, their relationships to each other, and to the environment, and the principles guiding its design and evolution.

- However, there are another well-established definitions.
  (e.g., Kruchten's 4+1 View Model of Architecture discussed before)
- According to those definitions, software architecture deals with
  - the organizational structure of a system or component,
  - interfaces and dependencies of its subsystems/subcomponents,
  - their behaviour in their interactions/cooperation,
  - in accordance with architectural styles.

Dealing with Architectural Structure (1)

in “Software Systems Architecture” by Rozanski & Woods, 2005/2011

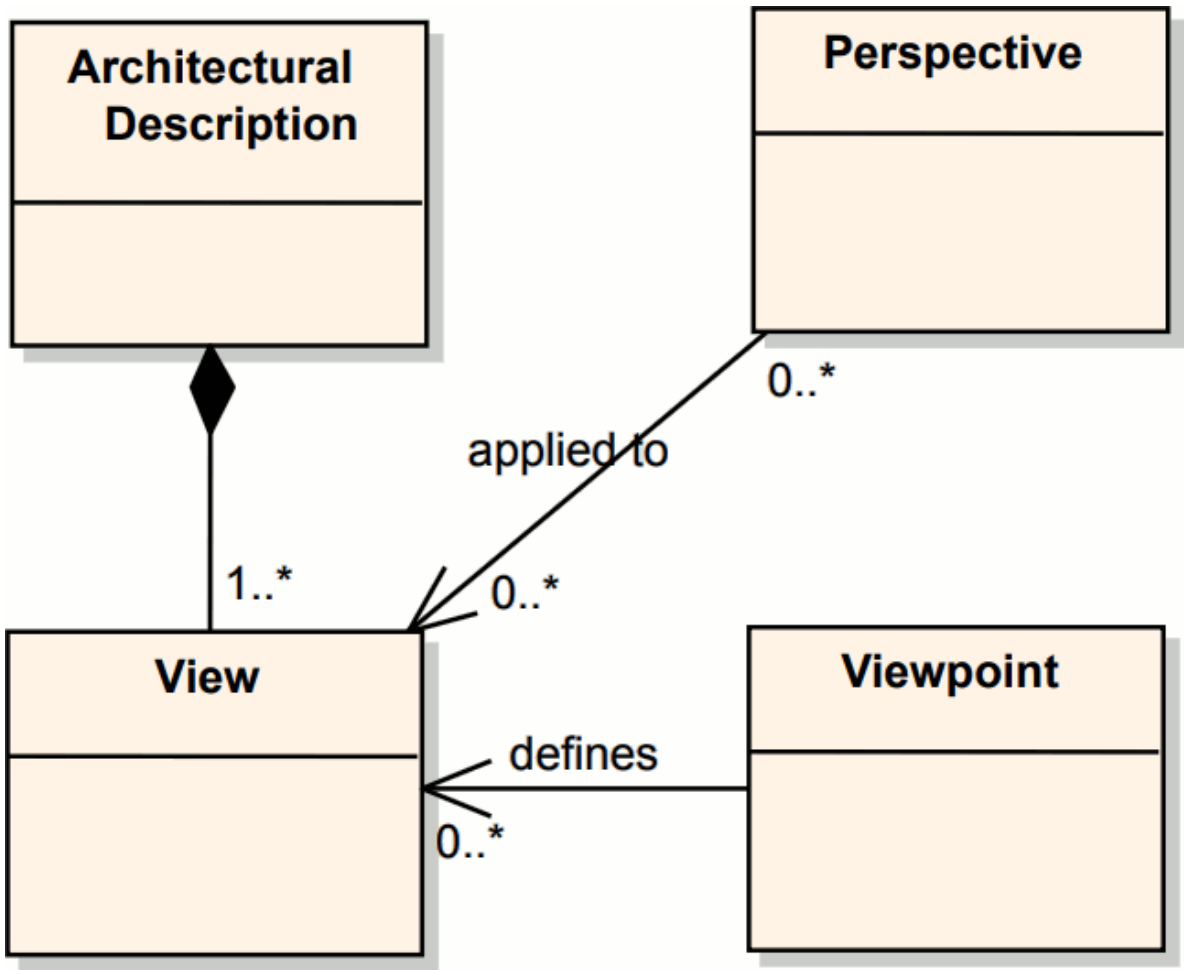
- The architectural description is decomposed into several **views**.
  (each for one specific aspect of the arch. structure; a separation of concerns)
- The development of each view is guided by a **viewpoint**.
  (general; provides knowledge, experience, process templates, guiding architects)
- Rozanski & Woods proposed the seven viewpoints:
  - functional – elements, connectors, interfaces
  - information – entities, constraints, relationship, ownership, usage
  - concurrency – processes, threads, coordination, element mapping
  - development – layers, module structure, standard design, code-line
  - deployment – hardware, network, dependencies, process mapping
  - operational – installation, migration, administration, support
- The concept of the views and the viewpoints is well-established.
  (approaches such as Kruchten’s 4+1V Model, Hofmeister&Nor&Soni, RM-ODP, etc., can be considered as libraries of viewpoints, i.e., guides to build the views)

Dealing with Architectural Structure (2)

in “Software Systems Architecture” by Rozanski & Woods, 2005/2011

- A view described by a set of models conforming a given viewpoint.
  (how a certain stakeholder with particular concerns sees the system, i.e., it leaves out details irrelevant to the concerns)
- However, viewpoints typically do not consider quality properties.
  (those usually need cross-viewpoint consideration that may be available too late)
- **Perspectives** guide in achieving the required quality properties.
  (each addresses one major quality property to be considered by the architect)
- Rozanski & Woods proposed an initial set of arch. perspectives:
  (each defines particular concerns, and techniques to achieve these concerns)
  - performance and scalability
  - security
  - availability and resilience
  - evolution
  - and also location, i18n, usability, regulation, etc.

Arch. Description, View, Viewpoint, and Perspective

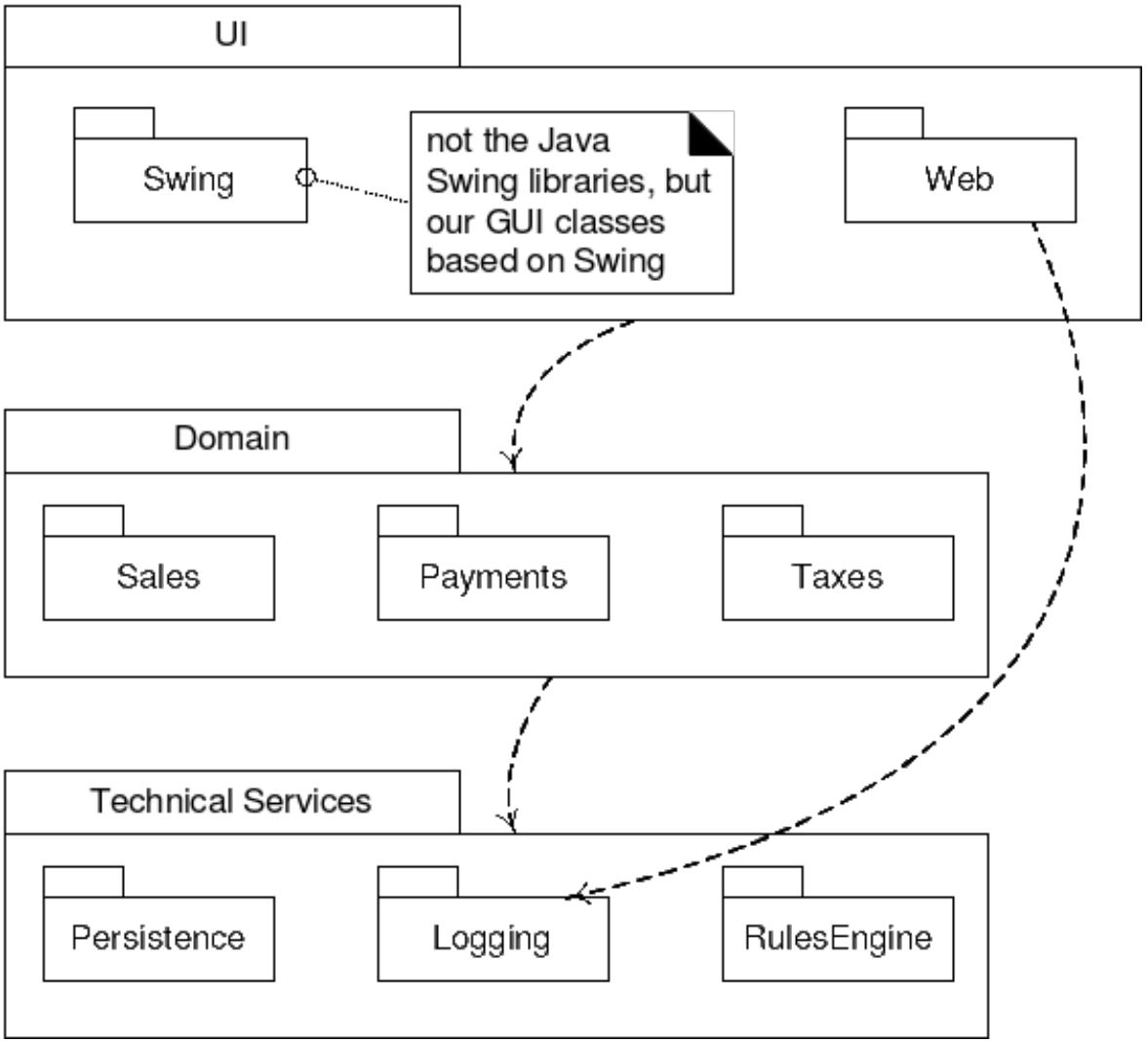


(adopted from “Software Architecture Using Viewpoints and Perspectives” by Eoin Woods)

Software Architecture and Layers

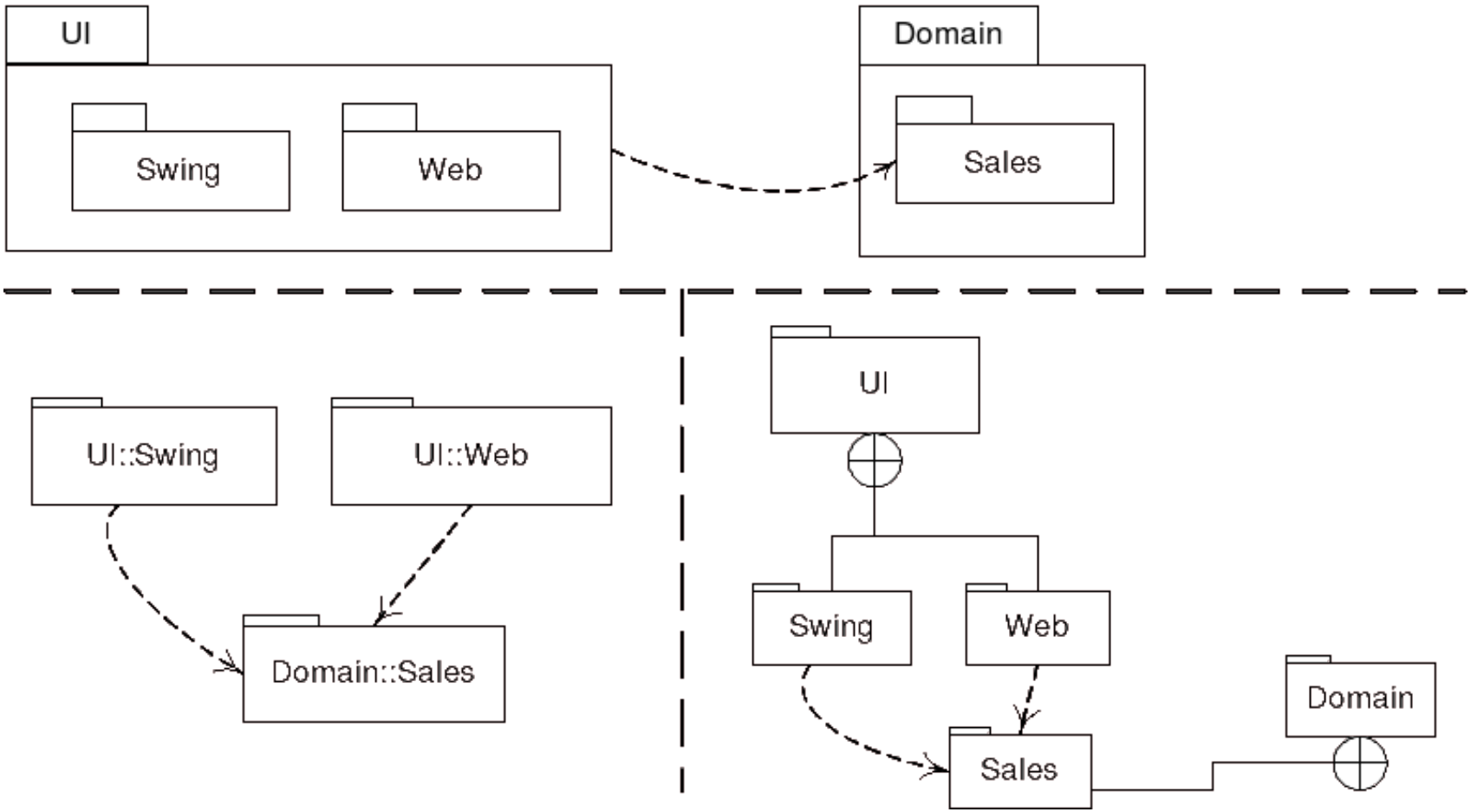
- Nowadays, SW arch. about hierarchically organised layers.
  (that is the “logical” architecture, not a “physical” deployment)
- Layer is a grouping of classes, packages, subsystems, etc., linked together to be responsible for a particular aspect of a SW system.
  (for example, to provide a user interface, to perform a domain logic, to serve as technical means, e.g., to access external systems, databases, etc.)
- A layer is on top of another, because it depends on it.
  (it can exist without the layers above it, not without the layers below)
- According to the dependency, we can distinguish
  - a strict (closed) layered system,
    (each layer strictly depends only on the first adjacent layer below)
  - or a relaxed (open) layered system.
    (a layer can also depend on all the layers below it, not only the first one)

An Example of the Layered System



(adopted from “Applying UML and Patterns” by Craig Larman)

Modelling Architecture by UML Package Diagrams



(adopted from “Applying UML and Patterns” by Craig Larman)

Architecture Design with Layers (1)

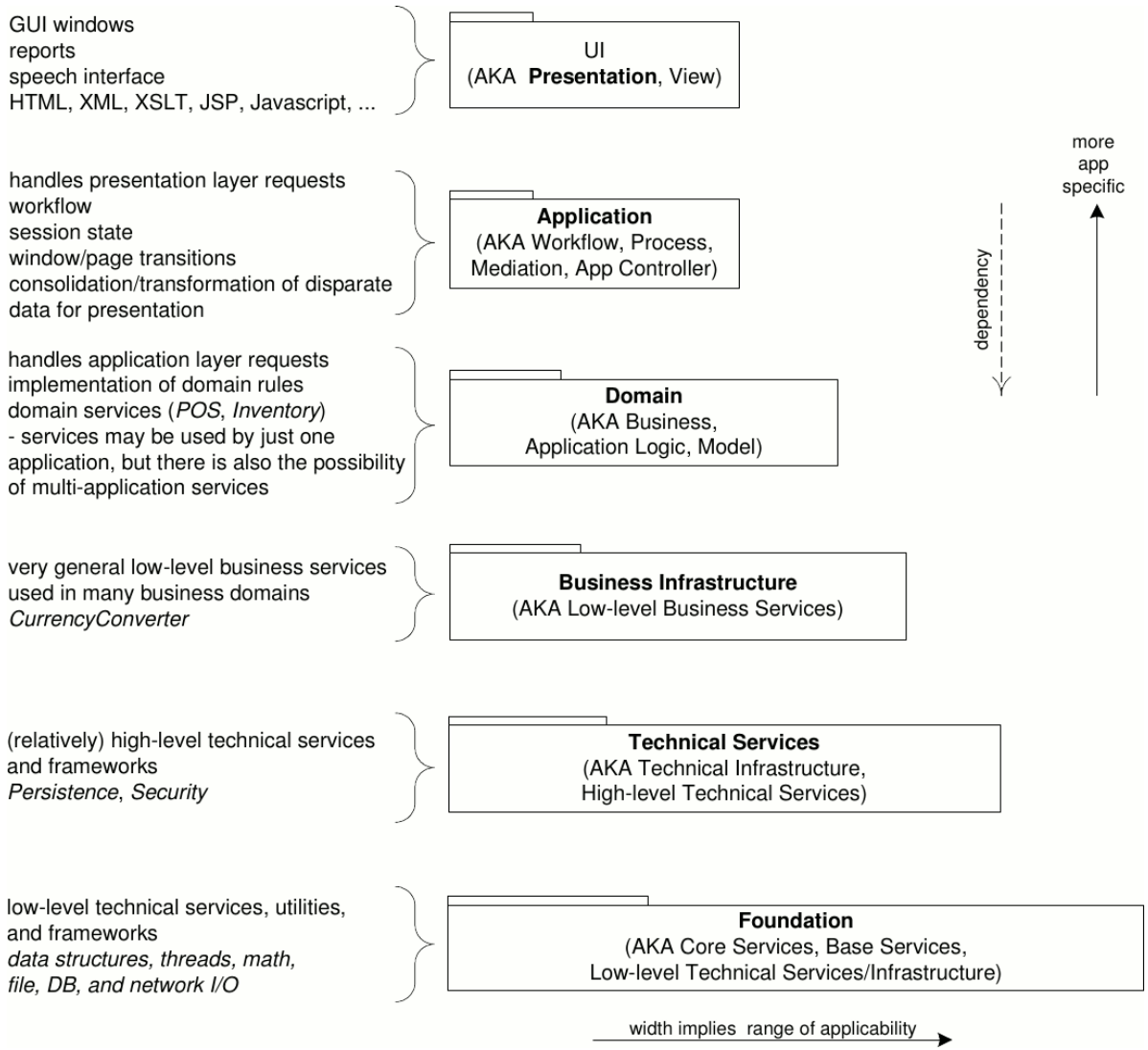
- By the “separation of concerns” (SoC) design principle.
  (each layer addresses a separate concern, a particular component of the system)
  - Lower layers provide more general low-level services.
    (technical services; e.g., data access layer, persistence layers, etc.)
  - Higher layers provided more specific high-level services.
    (application services; e.g., business logic layer, UI/presentation layer, etc.)
- Cooperation is initiated always from the higher to the lower layers.
  (higher layers utilise/depend on the lower layers, not in the reverse direction)

Architecture Design with Layers (2)

This layered approach requires that

- updates are propagated in the right bottom-up direction,
  (how to ensure an update of lower layers without breaking the higher layers?)
- application logic and UI are independent in two separate layers,
  (how to let app. logic to update UI and UI to pass actions to app. logic?)
- general technical services are linked to specific app. services,
  (how to develop/build/test the app. services without the tech. services?)
- there is a high cohesion and low coupling in/between the layers.
  (how to ensure these properties and how to assign responsibilities in a development team without breaking the layers?)

Common Layers of a Software System



(adopted from “Applying UML and Patterns” by Craig Larman)

Advantages of the Layered Architecture

- SoC design principle results into high cohesion and low coupling.
  (the architecture is easier to understand and its components are more reusable)
- The software system complexity is encapsulated in the layers.
  (it is easier to build the layers, to define interactions between the layers)
- Layers can be updated individually without breaking the system.
  (with the respect to their dependencies and interfaces)
- Better scalability of individual layers.
  (layers can be deployed individually, the system can be distributed)
- The structure ensures easier team development and integration.
  (high cohesion, low coupling, interfaces, clear dependencies, etc.)

Domain Objects

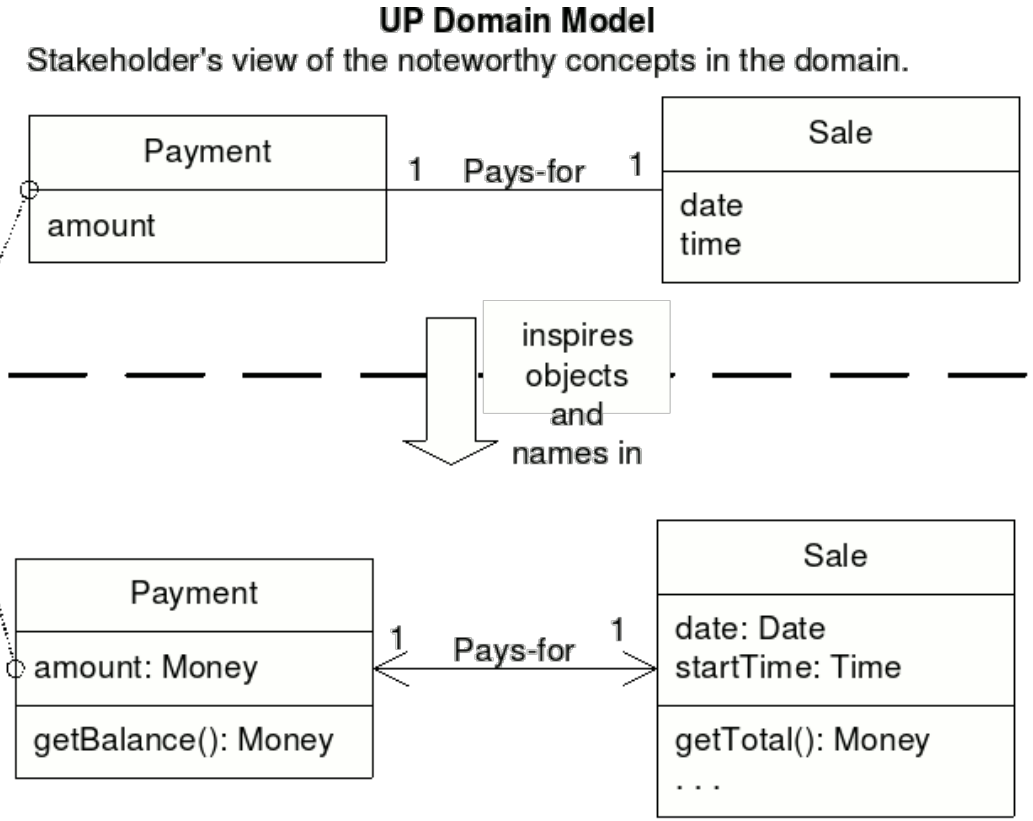
- SW systems composed of UI and application/business layers.
  (e.g., GUI components in the UI layer and a tax calculator in the app. layer)
- The questions is how to design the application layer?
  - There can be one class providing all necessary operations,
  - or domain classes with their attributes and operations according to responsibilities of their objects in the application logic.
    (there will be domian objects, therefore it is a “domain layer”)
- Domain classes are usually adopted from a domain model.
  (an object-oriented developer takes inspiration from the real-world domain)

Domain Objects in Domain Model and Design 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.

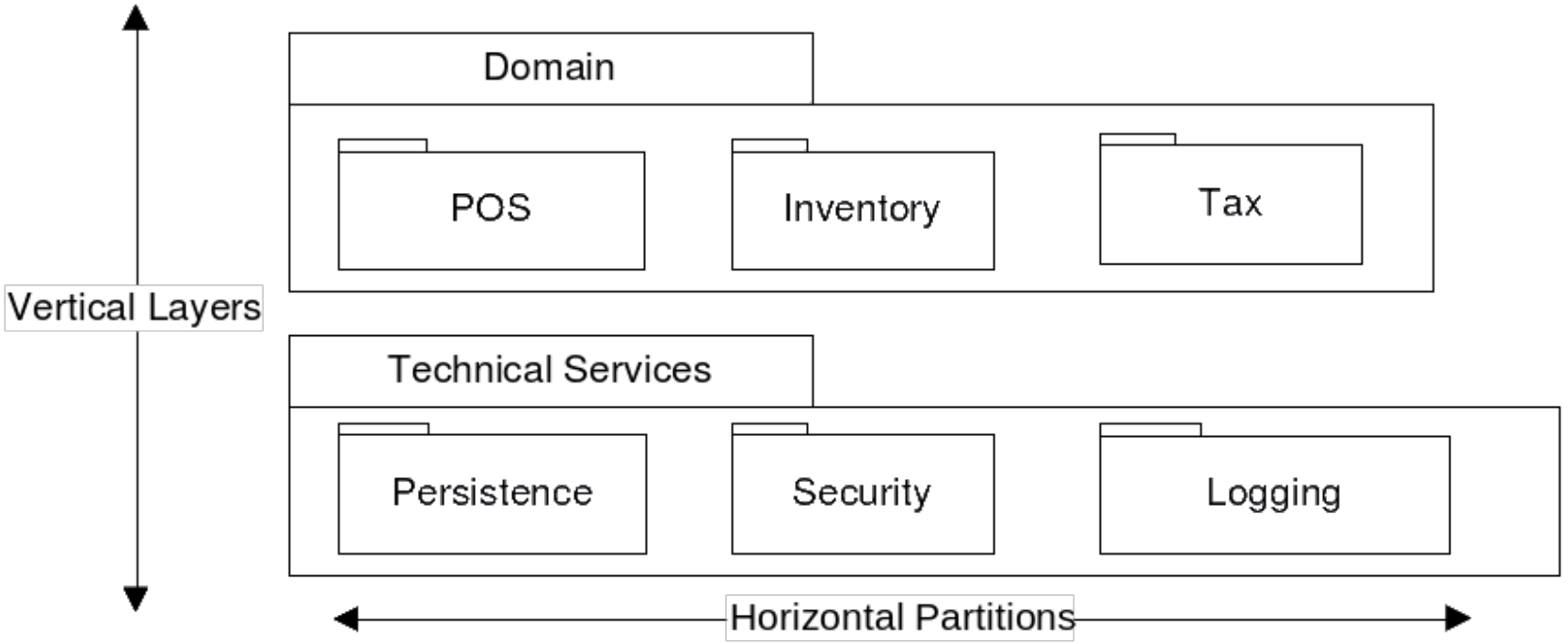


(adopted from “Applying UML and Patterns” by Craig Larman)

Architectural Layer, Tier, and Partition

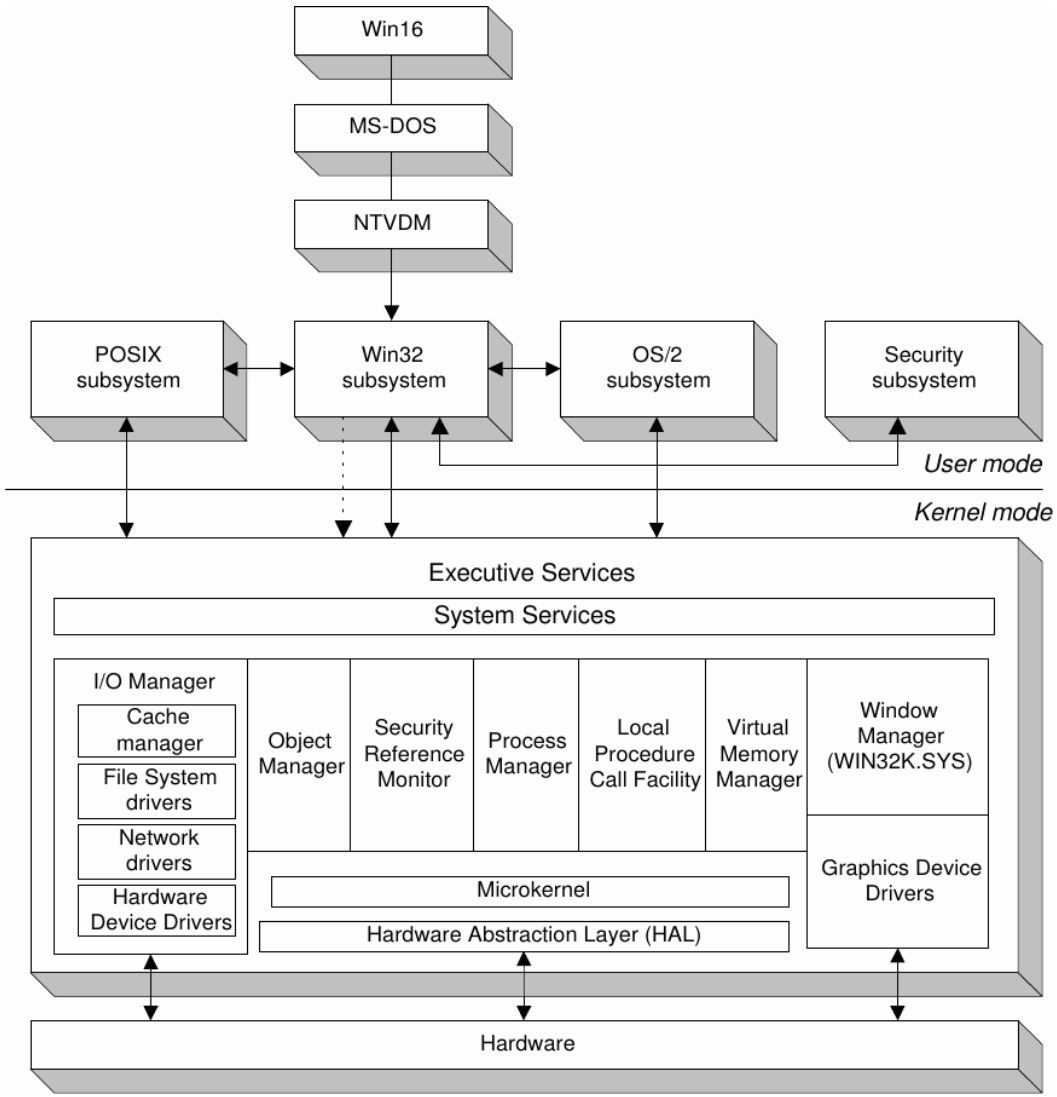
- Layers – to separate responsibilities and manage dependencies:
  - each layer addresses a separate concern;
    (SoC design principle, the layer has a specific responsibility)
  - a higher layer can use services in the first or all lower layers
    (not the other way around; it depends on strict-/relaxedness of the system)
- Tiers – to separate software artefacts in their deployment:
  - tiers are physically separated, running on separate machines;
    (to improve scalability & resiliency, however, with a communication overhead)
  - tiers can be in particular relationships of their interaction;
    (e.g., client-server, peer-to-peer, etc.)
  - communication is not restricted, a tier can call to any other tier;
    (directly, or using asynchronous messaging by a message queue/bus)
  - each layer might be hosted in its own tier (one to one),
    or several layers might be hosted on the same tier (many to one).
- To split a (vertical) layer into several tiers, the layer can be split into several (horizontal) partitions that will be distributed into the tiers.

Layers and Partitions in UML Package Diagram



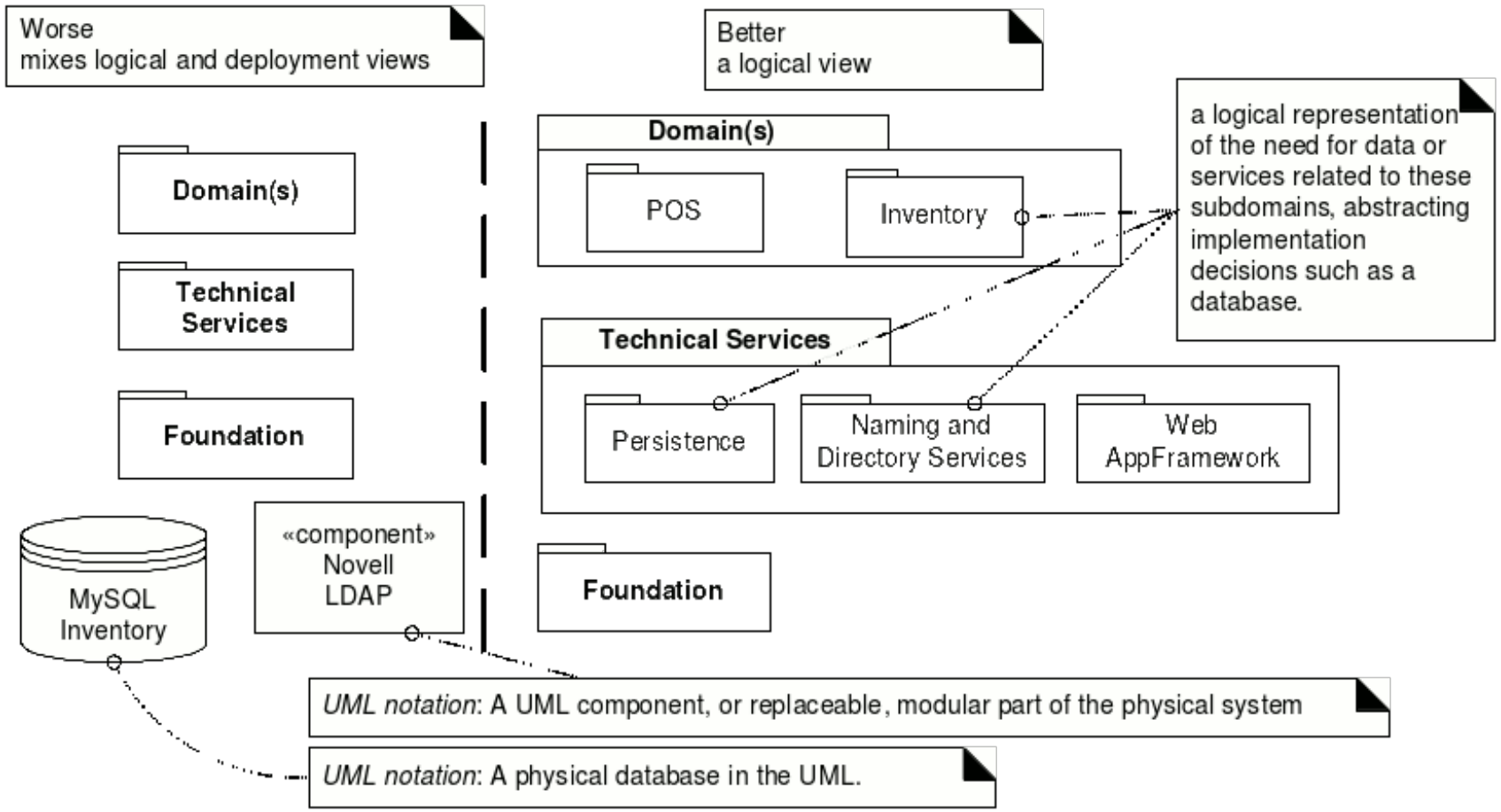
(adopted from “Applying UML and Patterns” by Craig Larman)

Windows NT Operating System Architecture



(adopted from “Windows NT Networking Architecture” by Microsoft)

Best Practice: No External Resources and Services

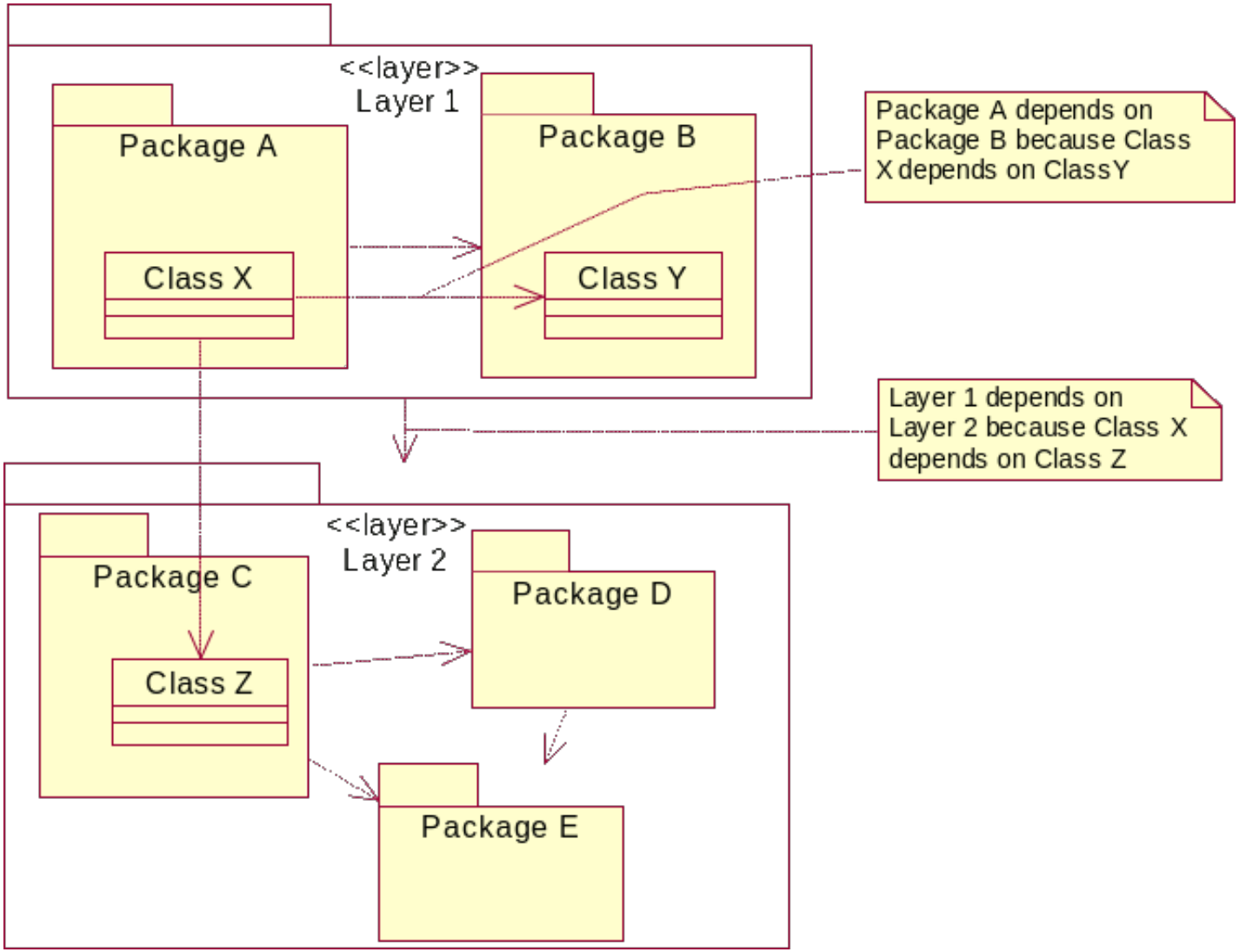


(adopted from “Applying UML and Patterns” by Craig Larman)

Package Dependencies

- Package *A* depends on package *B* if
  a modification of *B* may result into a modification of *A*.
- The package dependency is a transitive relation.
  (a circular dependency of mutually recursive packages, either directly or indirectly)
- Class dependency may result into package dependency
  that may result into layer dependency.
  (it can be an issue because there are rules for layer dependencies)
- The dependency is critical for supportability.
  (i.e., serviceability, maintainability, testability, flexibility, install-ability, scalability, etc.)

An Example of Class Dependencies



(adopted from “Practical Software Engineering” by Maciaszek&Liong)

Class Dependencies (1)

according to “On the congruence of modularity and code coupling” by Beck&Diehl, 2011

Class/object *A* depends on class/object *B* if

SD there is a **structural dependency**

inheritance: One of the classes extends the other class,

aggregation: aggregates another by having class variables using the other class,

usage: or uses the other class in a method as a local variable, a method parameter, or by calling a method of the other class.

FO there is a **fan-out similarity**

inheritance: The classes extend the same class or implement a similar set of interfaces,

aggregation: aggregate a similar set of classes,

usage: or use a similar set of classes.

Class Dependencies (2)

according to “On the congruence of modularity and code coupling” by Beck&Diehl, 2011

Class/object *A* depends on class/object *B* if

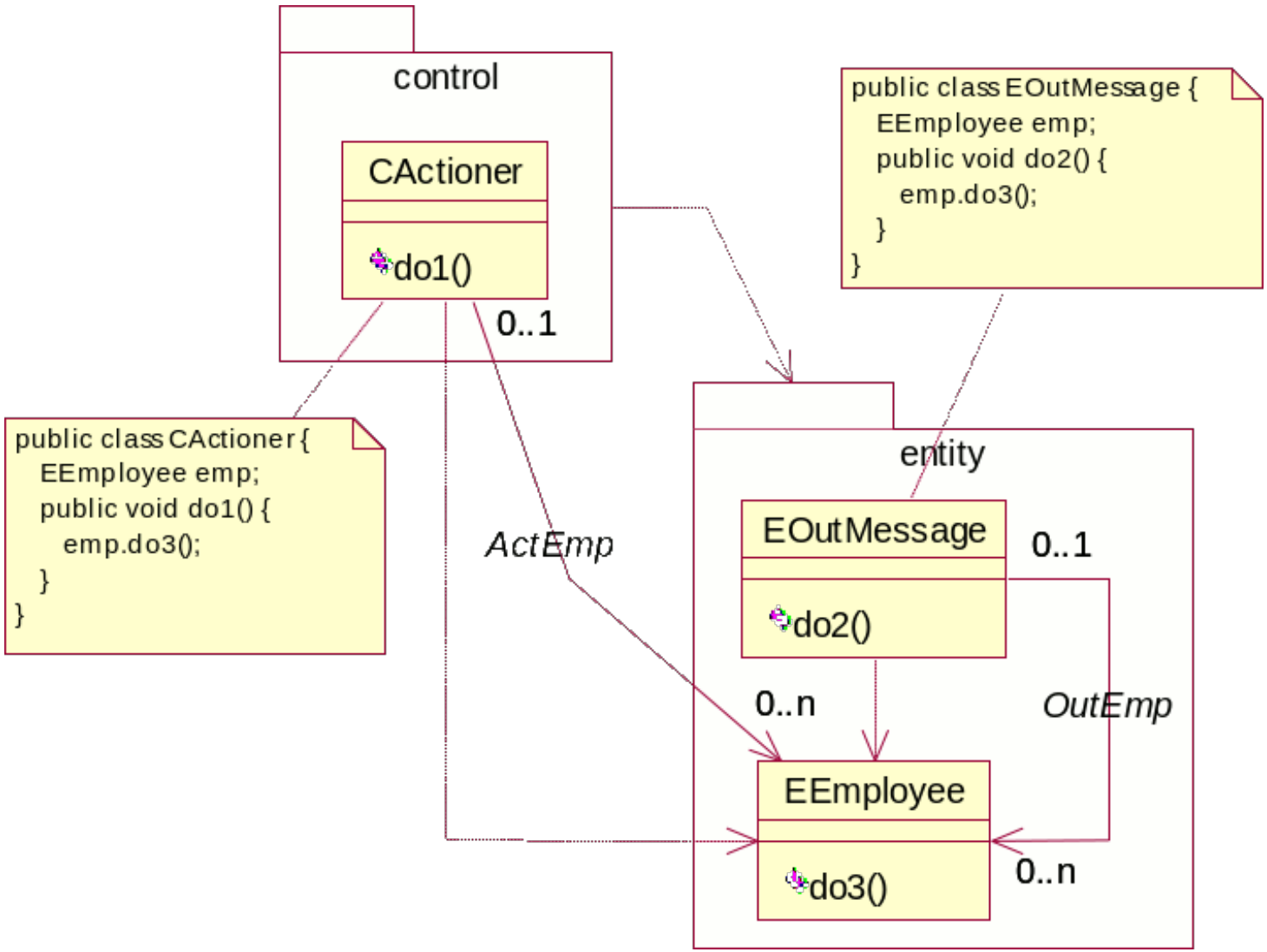
EC there is an **evolutionary coupling**
(the classes are frequently changed together during development)

CC there are **code clones**
(the classes contain equivalent or similar code, that is the exact matches of code fragments ignoring code layout and comments, or similarities when generalizing from identifier names and also ignoring code layout and comments)

CO there is the same **code ownership**
(the classes share common authors, the experts to be contacted when there is a problem or question concerning their code)

SS there is a **semantic similarity**
(the classes use the same vocabulary, that are identifiers used in their code and words in the corresponding comments)

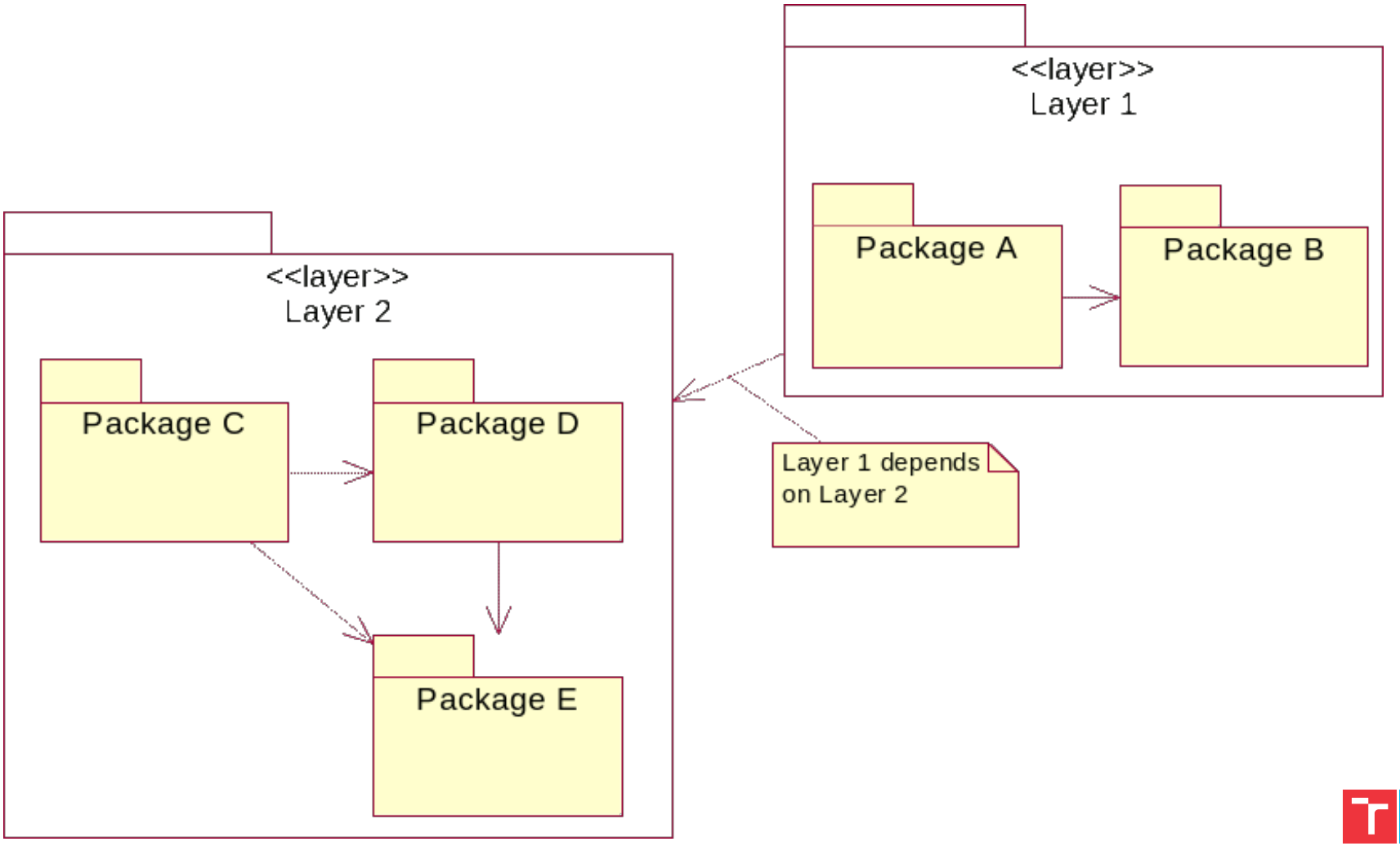
An Example of the Usage Structural Dependency



(adopted from “Practical Software Engineering” by Maciaszek&Liong)

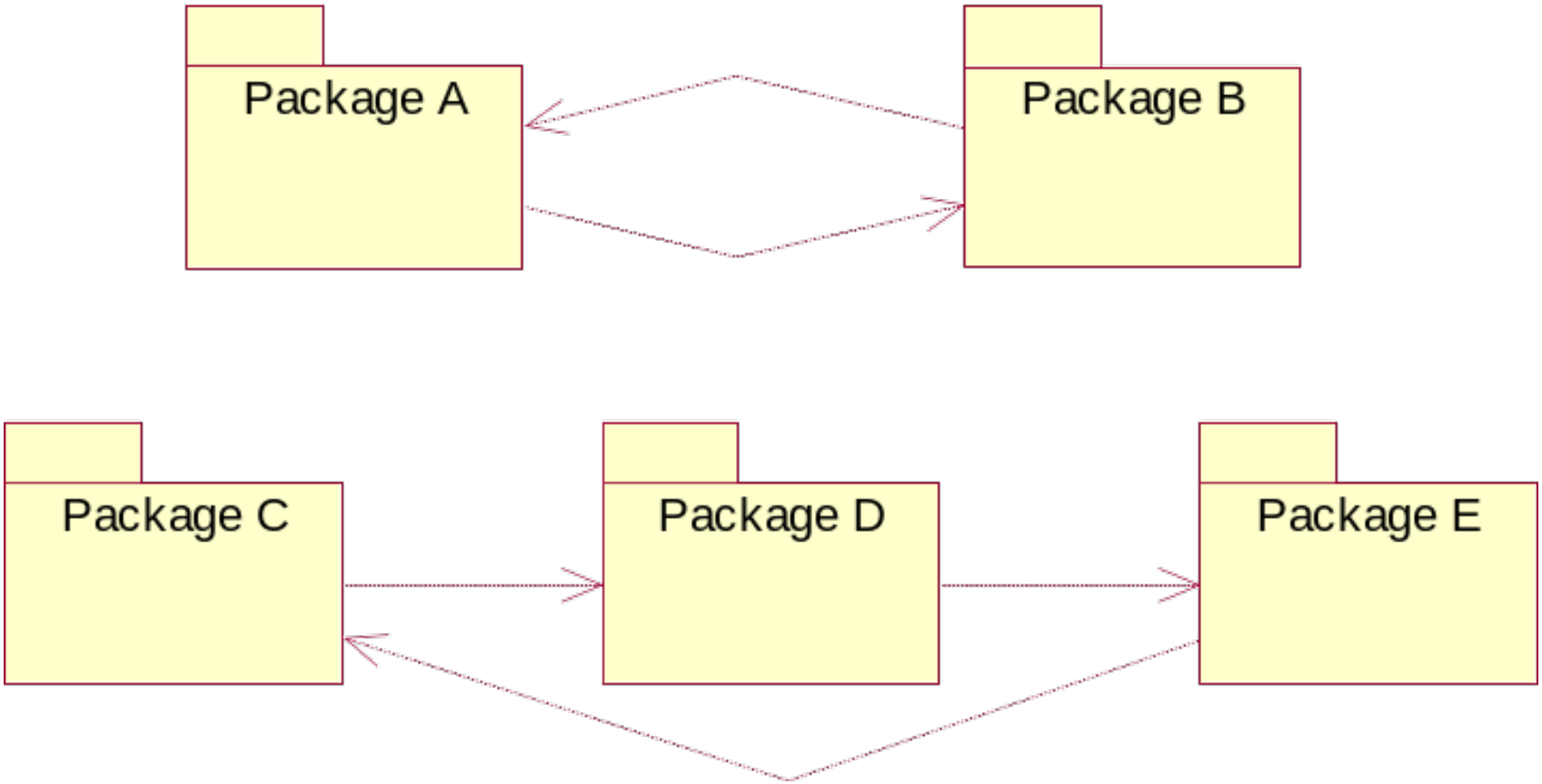
An Example of Layer Dependencies

(the lower layers should be stable)



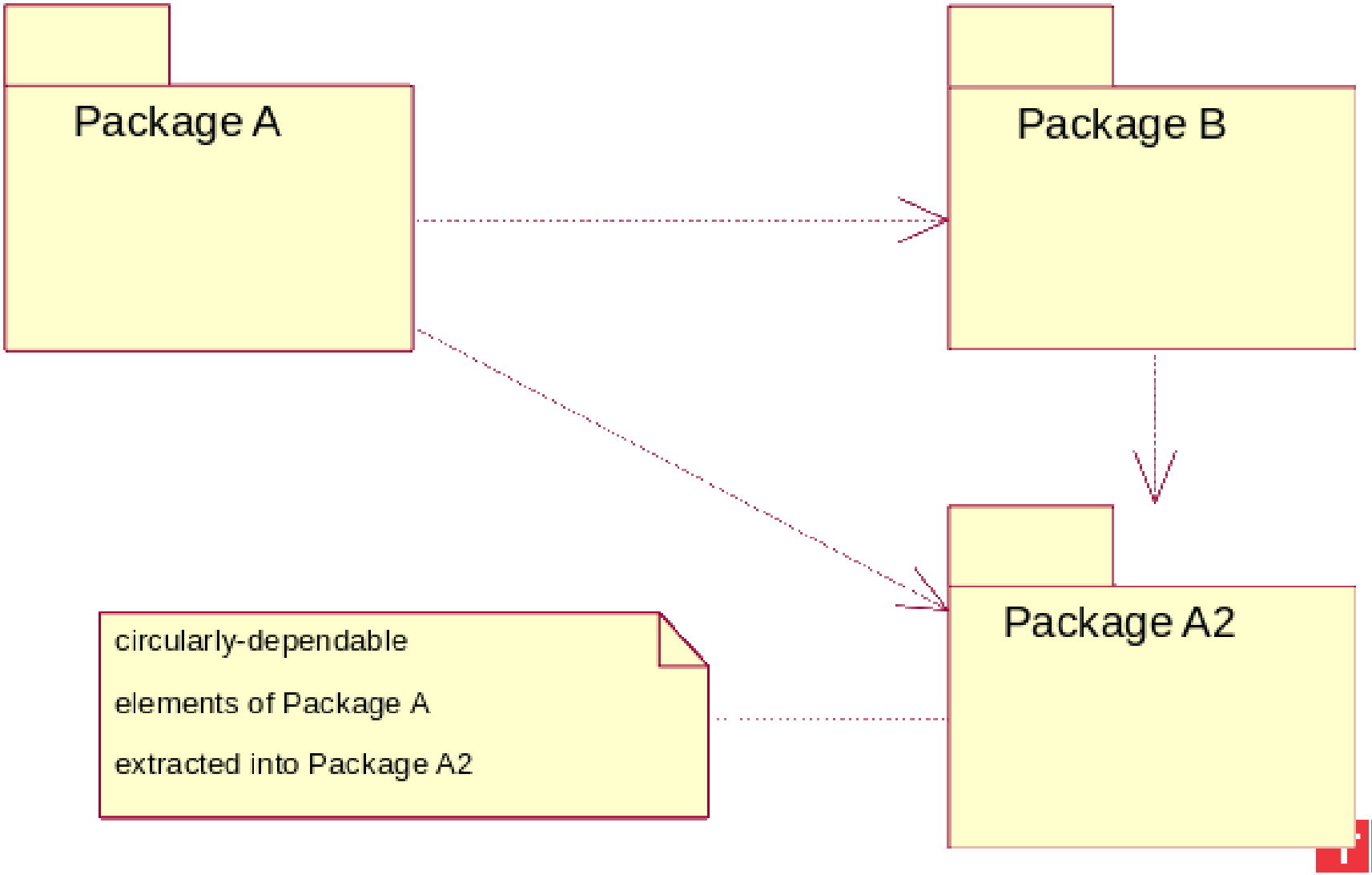
(adopted from “Practical Software Engineering” by Maciaszek&Liong)

An Example of Package Circular Dependencies



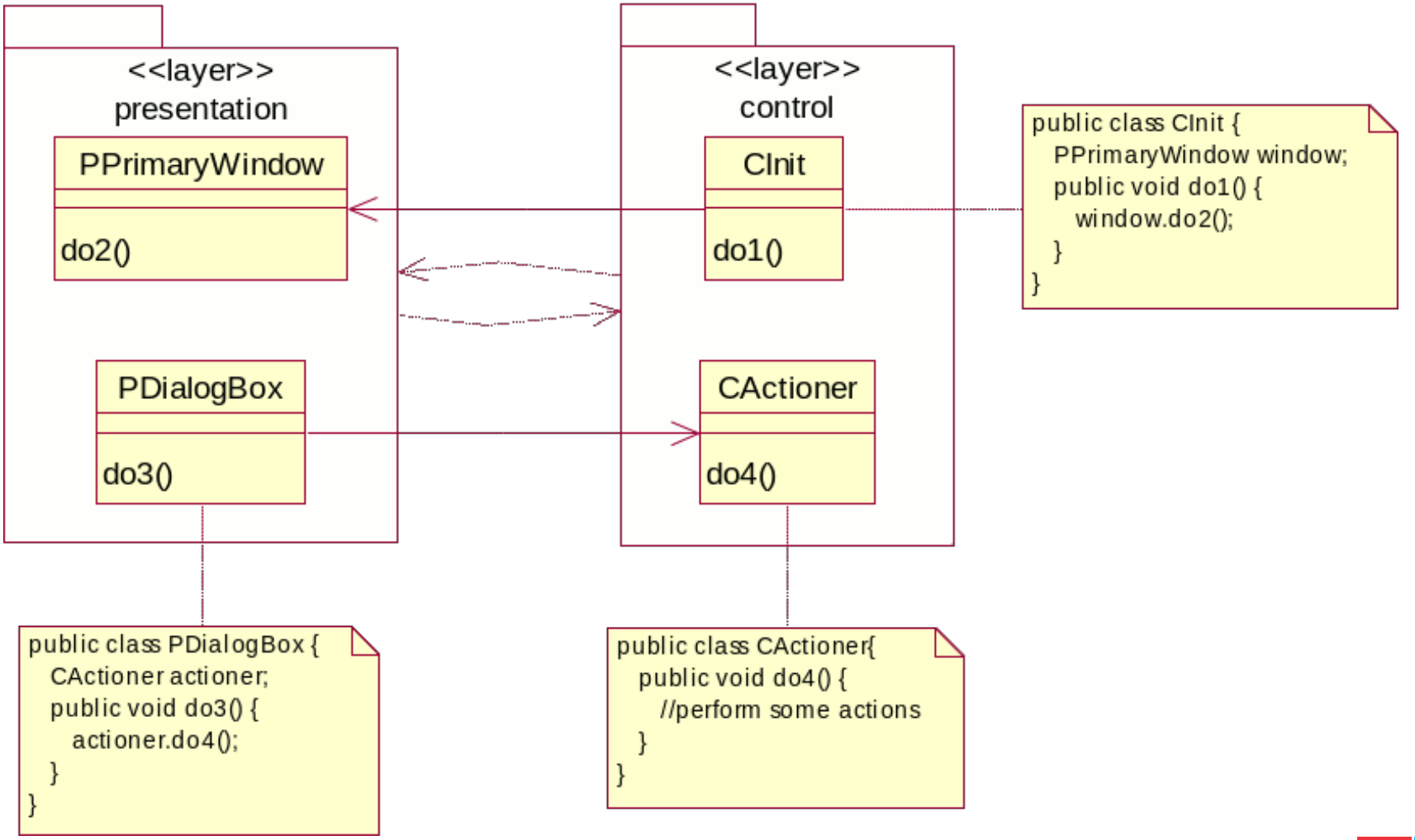
(adopted from “Practical Software Engineering” by Maciaszek&Liong)

Elimination of the Package Circular Dependencies



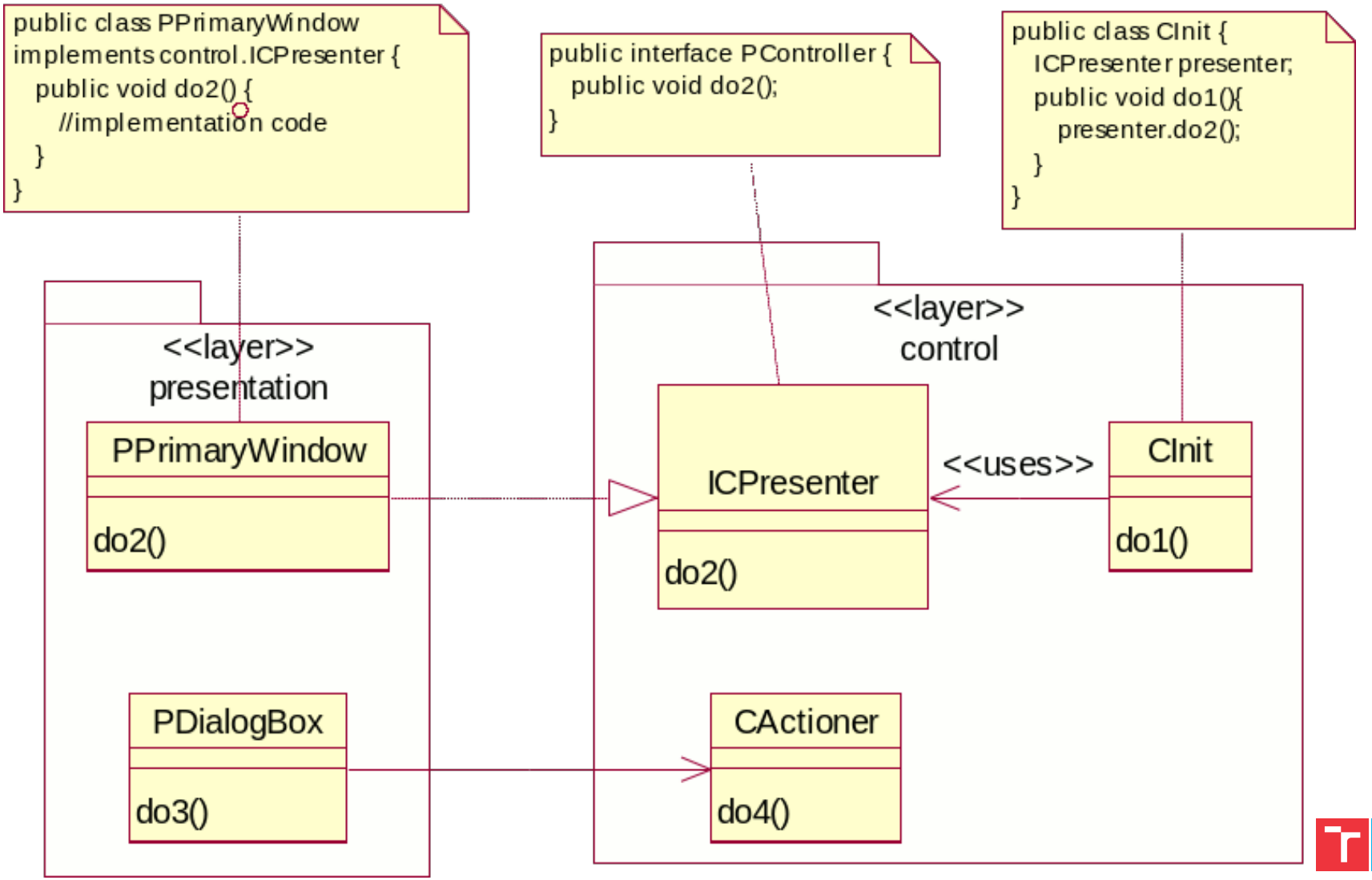
(adopted from “Practical Software Engineering” by Maciaszek&Liong)

Elimination of Circular Dependencies by Interfaces (1)



(adopted from “Practical Software Engineering” by Maciaszek&Liong)

Elimination of Circular Dependencies by Interfaces (2)



(adopted from “Practical Software Engineering” by Maciaszek&Liong)

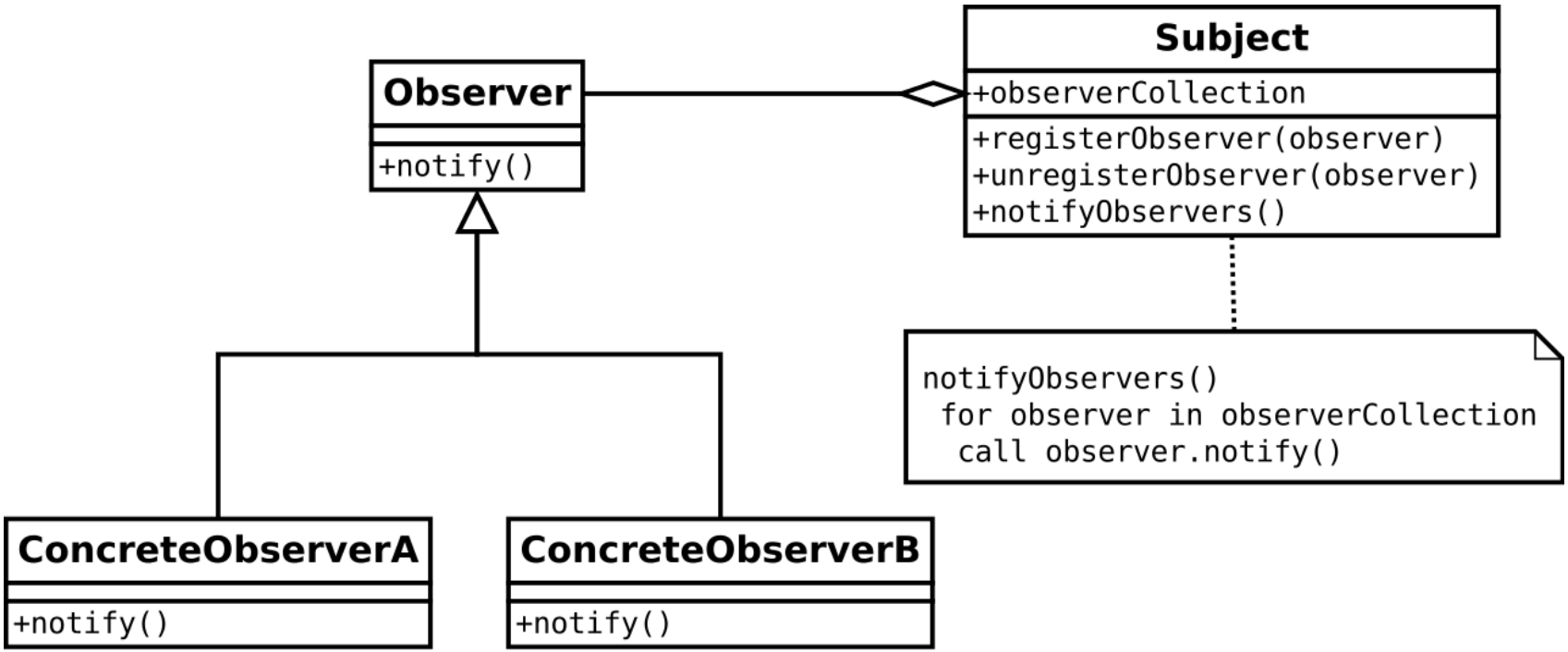
Model-View Separation

- The model-view separation is one of the core design principles.
  (it results into architectural patterns, such as MVC, MPV, or PCMEF)
- Some reasons for the model-view separation:
  - UI is quite volatile, business/application logic is stable;
  - there are different data views, specific to particular users;
  - UI design requires UX specialists, not developers;
  - different layers require various approach to testing.
- It reflects best practices of UI/application development:
  (these are goals of the MVC architectural pattern)
  - low-level application object should not be linked to UI components;
    (no such dependencies, e.g., “Sale” class on “JFrame” Java Swing UI)
  - UI components should not operate directly with business objects;
    (e.g., “Cancel the sale” button click should not perform a “delete” SQL query, not in the UI layer; it should delegate the operation to lower layers)
  - business/low-level application object should not know and communicate directly with the objects of UI components.

Observer Design Pattern

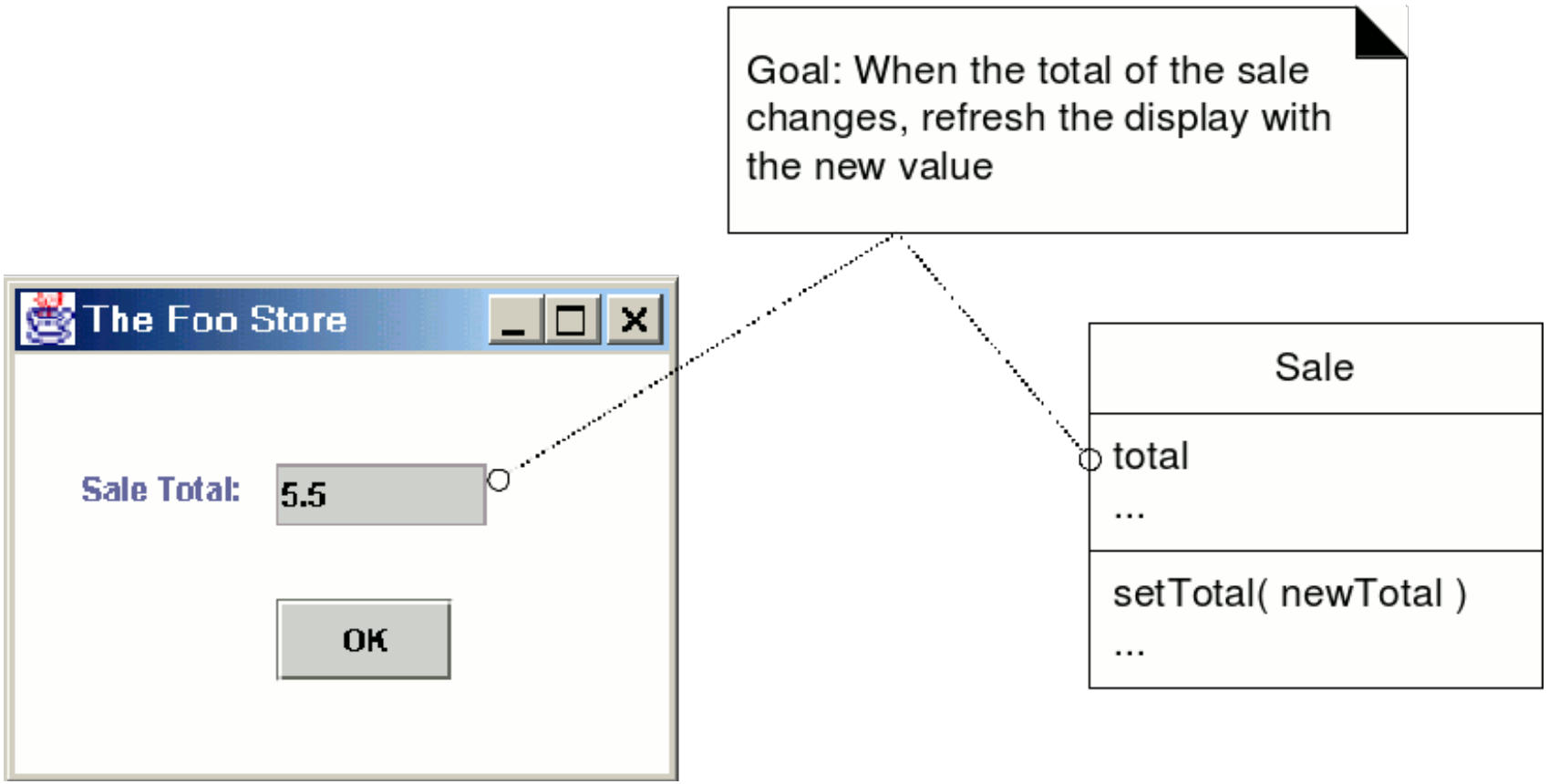
When a lower-level object wants to inform a higher-level object.

(in the cases of an information-flow going in the opposite direction than dependencies)



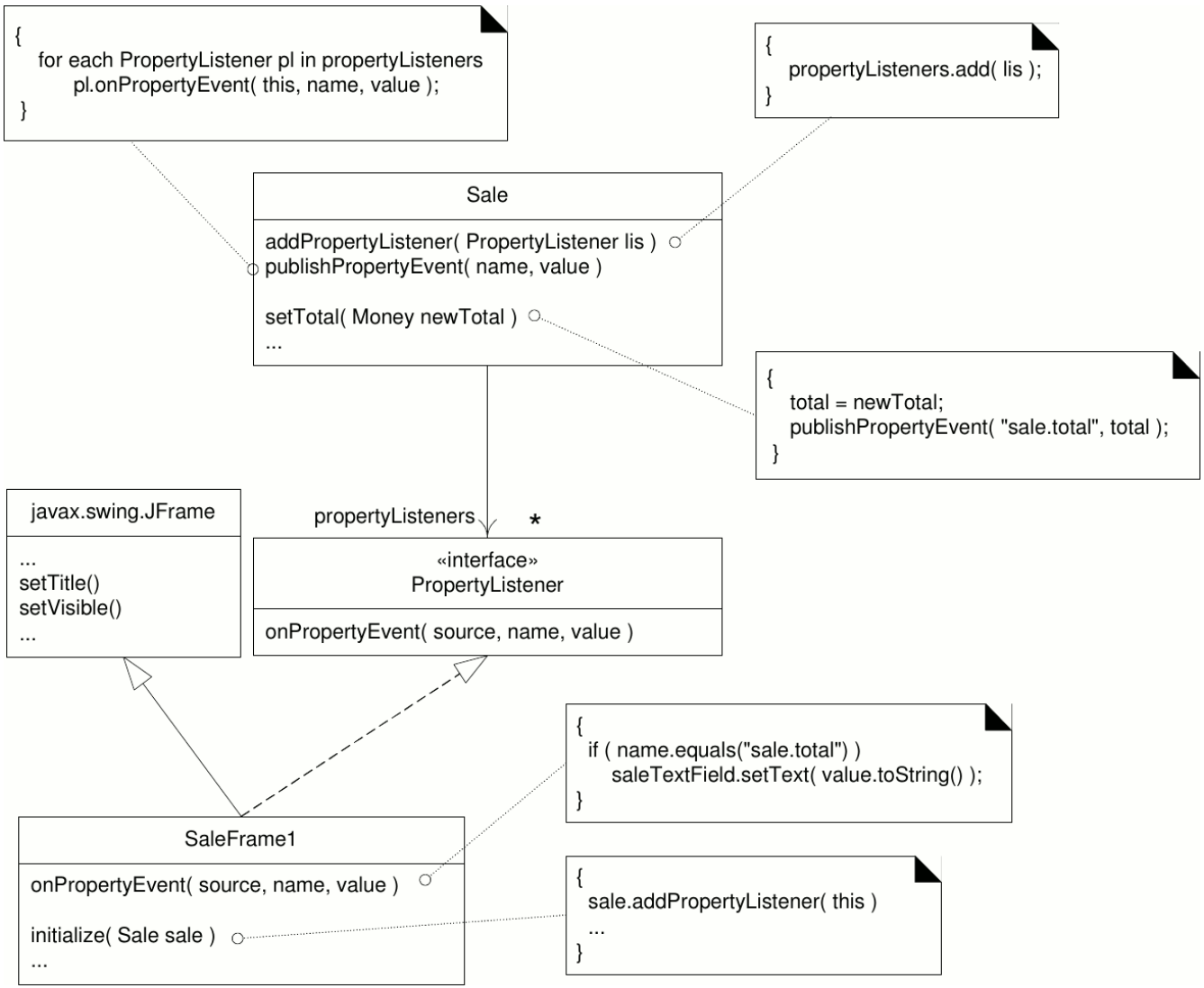
(adopted from “Observer pattern” by Wikipedia)

An Example of the Observer Design Pattern (1)



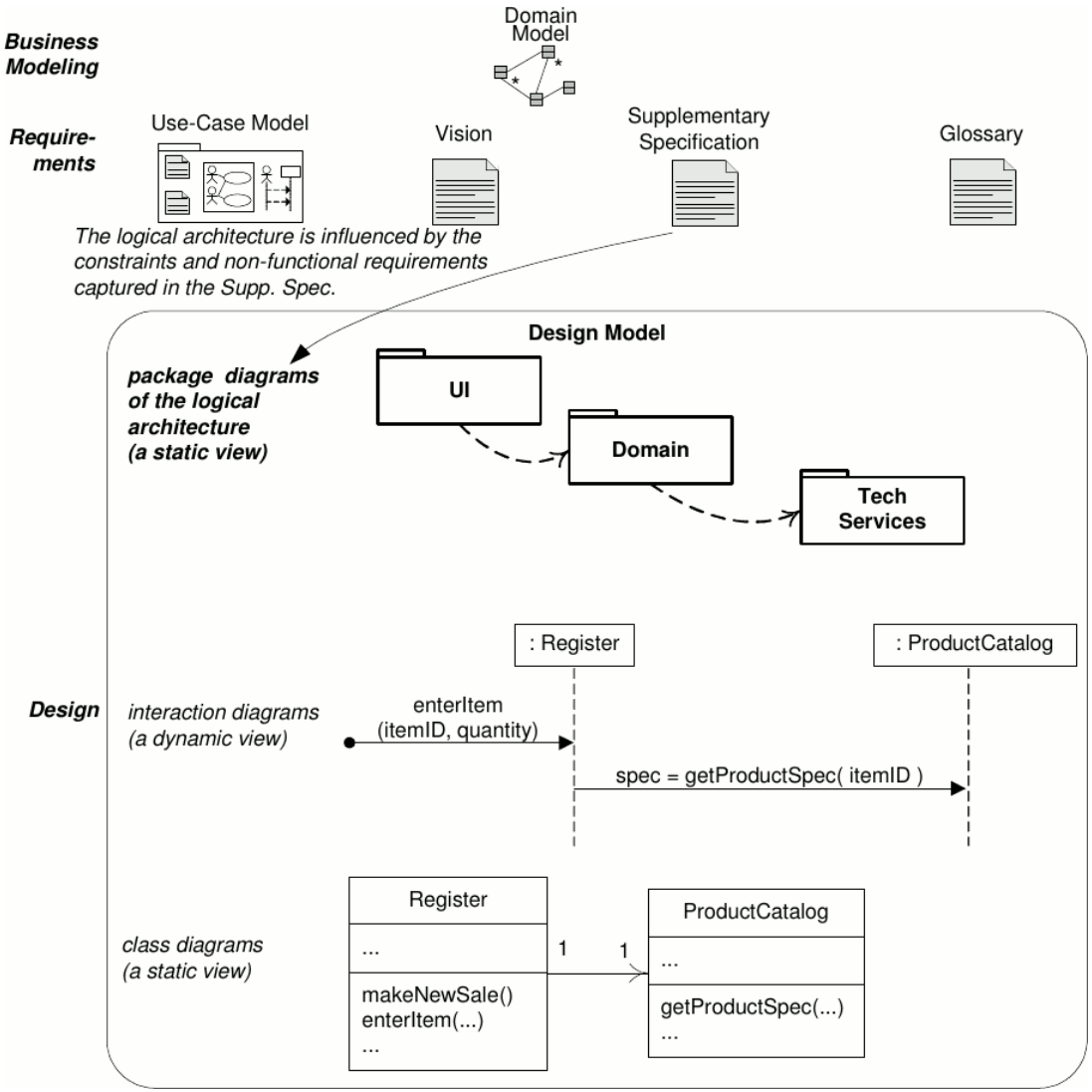
(adopted from “Applying UML and Patterns” by Craig Larman)

An Example of the Observer Design Pattern (2)



(adopted from “Applying UML and Patterns” by Craig Larman)

Sample UP Artefact Relationships

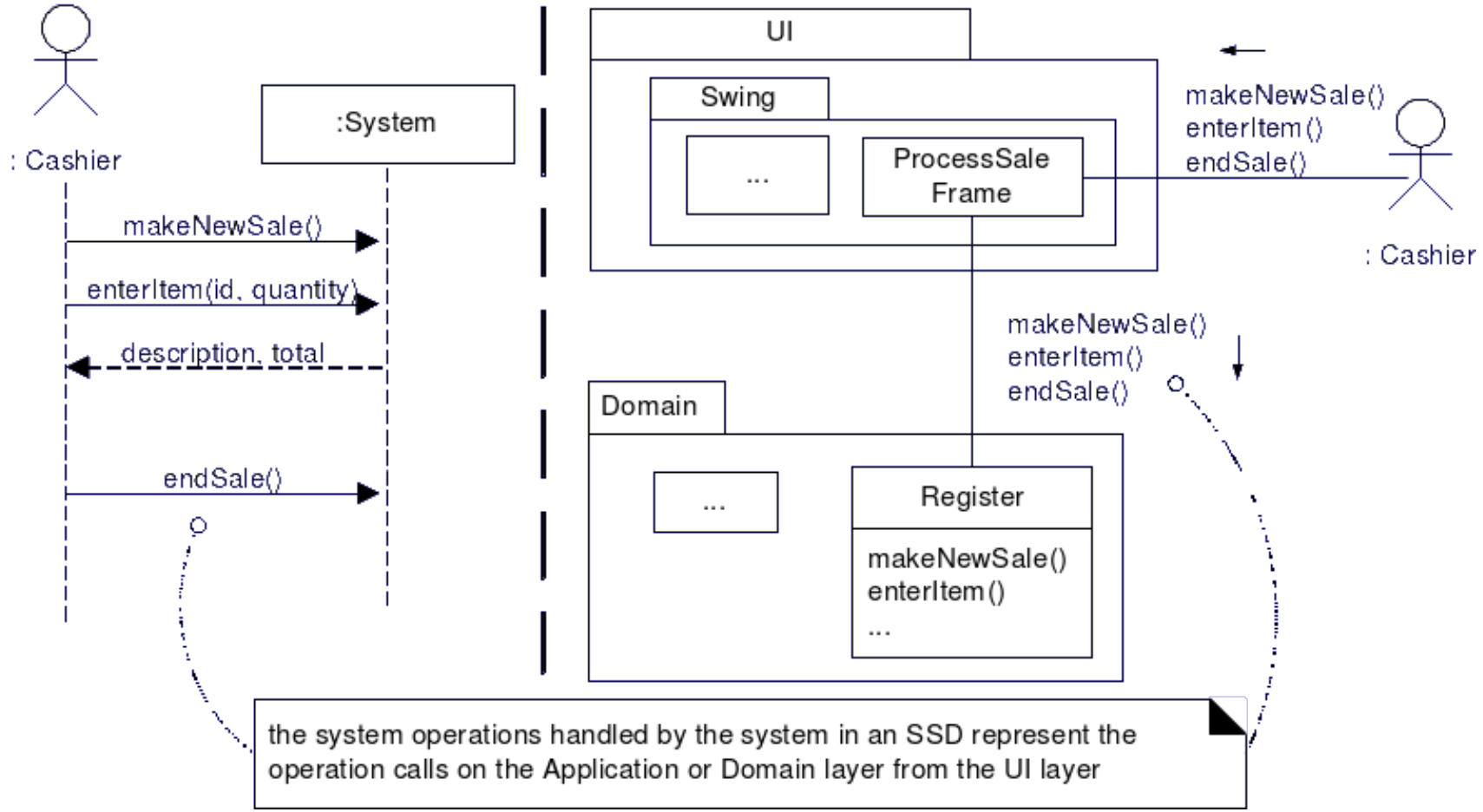


(adopted from “Applying UML and Patterns” by Craig Larman)

SSD, System Operation, and Architectural Layers

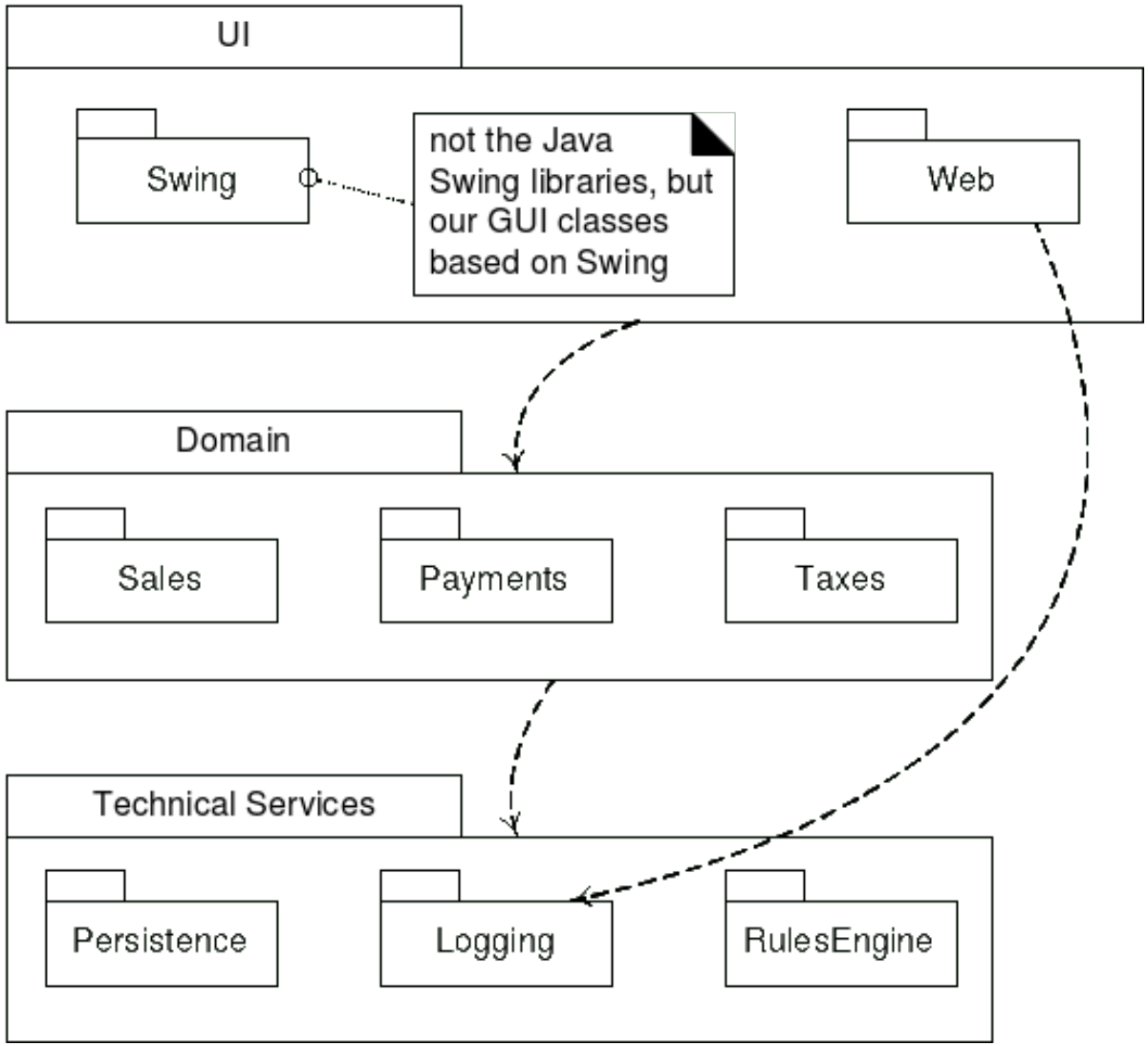
- System Sequence Diagram (SSD) shows system operations.
  (the operations invoked by actors on a system's interface)
- The architecture shows where the operations will be performed.
- In the layered architecture, the operations will be delegated to corresponding objects of the domain layer.
- The top layer is responsible for the presentation only, all actions need to be delegated to layers below.

System Operation in Architectural Layers



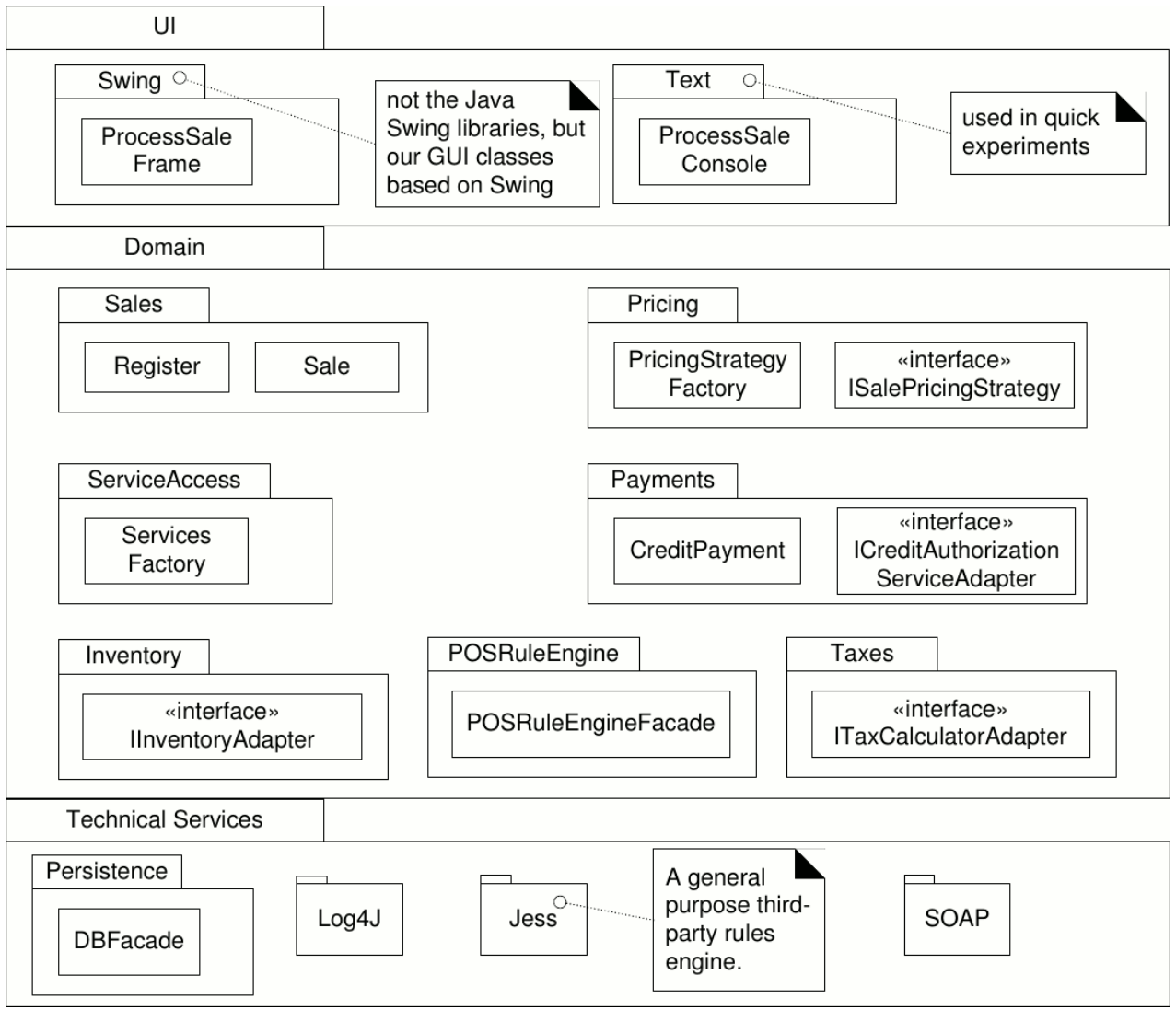
(adopted from “Applying UML and Patterns” by Craig Larman)

NextGen POS Architecture in the 1st Iteration



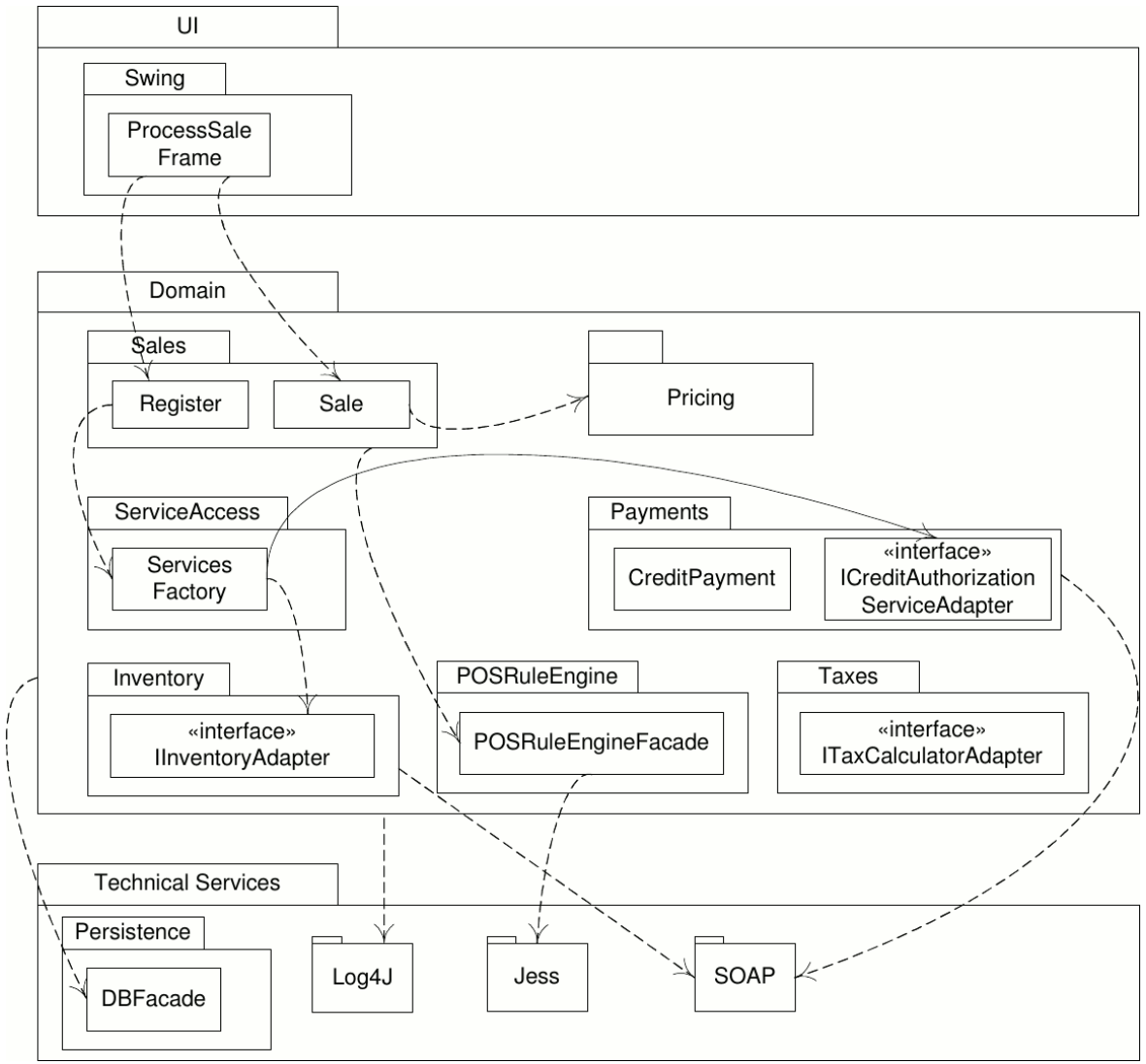
(adopted from “Applying UML and Patterns” by Craig Larman)

NextGen POS Architecture in the Next Iterations (1)



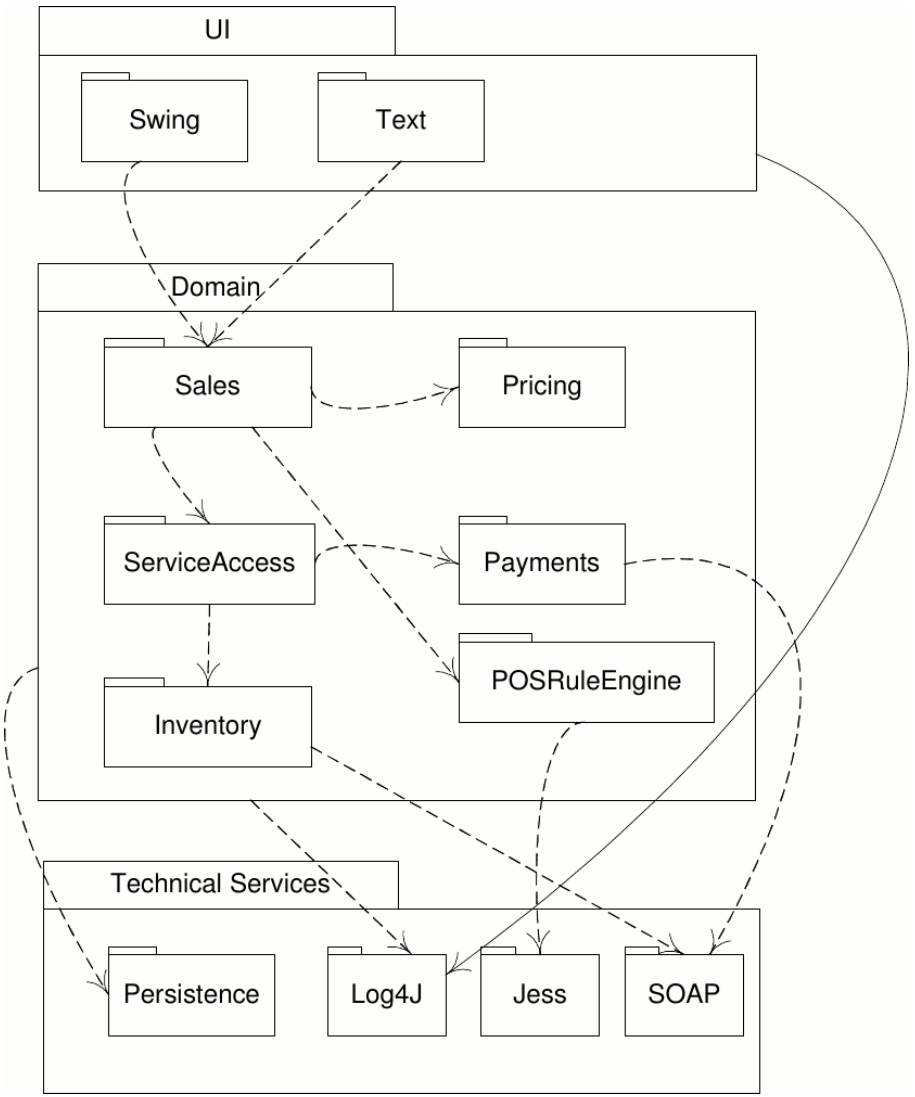
(adopted from “Applying UML and Patterns” by Craig Larman)

NextGen POS Architecture in the Next Iterations (2)



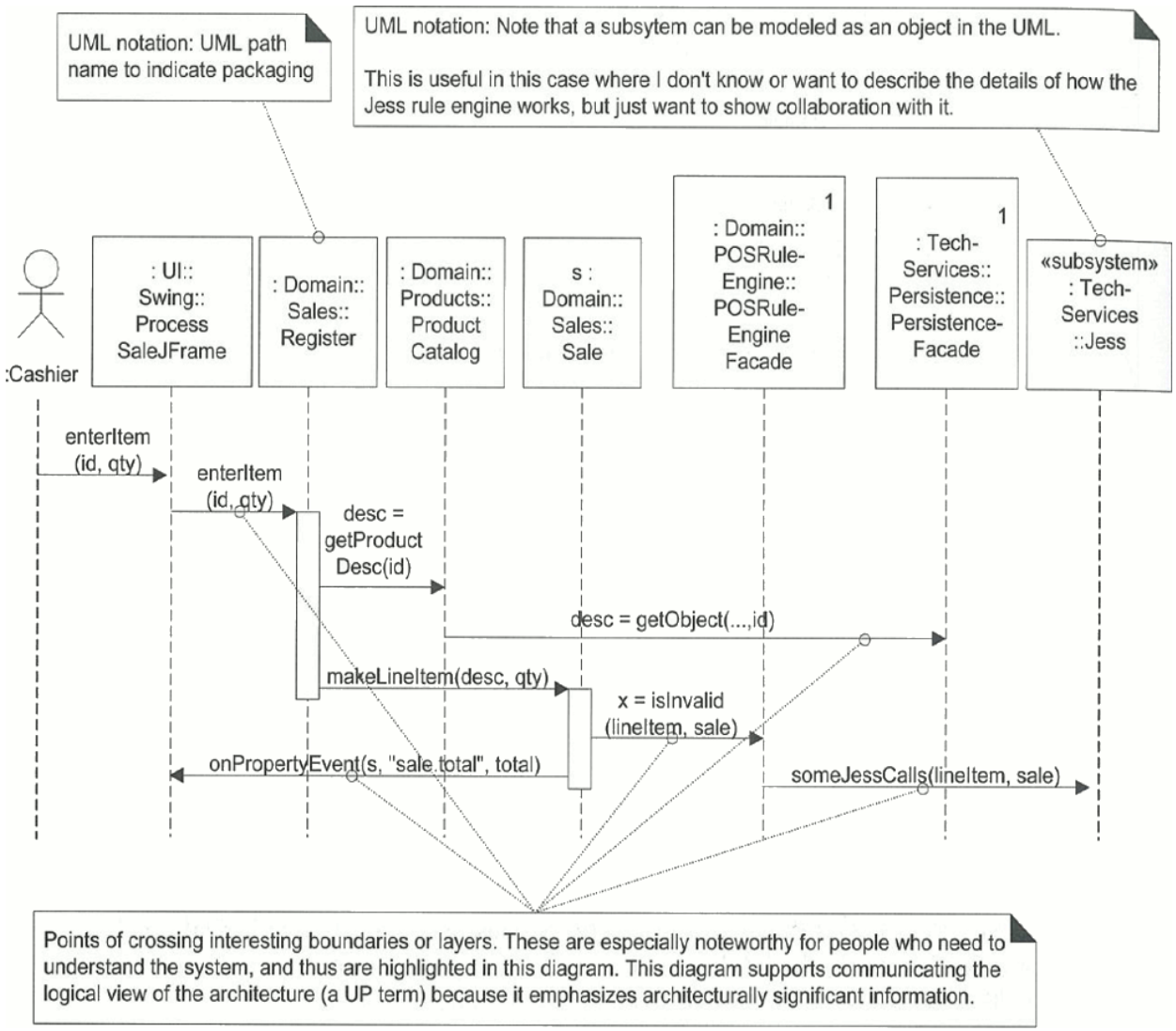
(adopted from “Applying UML and Patterns” by Craig Larman)

NextGen POS Architecture in the Next Iterations (3)



(adopted from “Applying UML and Patterns” by Craig Larman)

NextGen POS: Interactions in Packages and Layers



(adopted from “Applying UML and Patterns” by Craig Larman)

Thank you for your attention!

Marek Rychlý
<rychly@fit.vutbr.cz>