Software Design

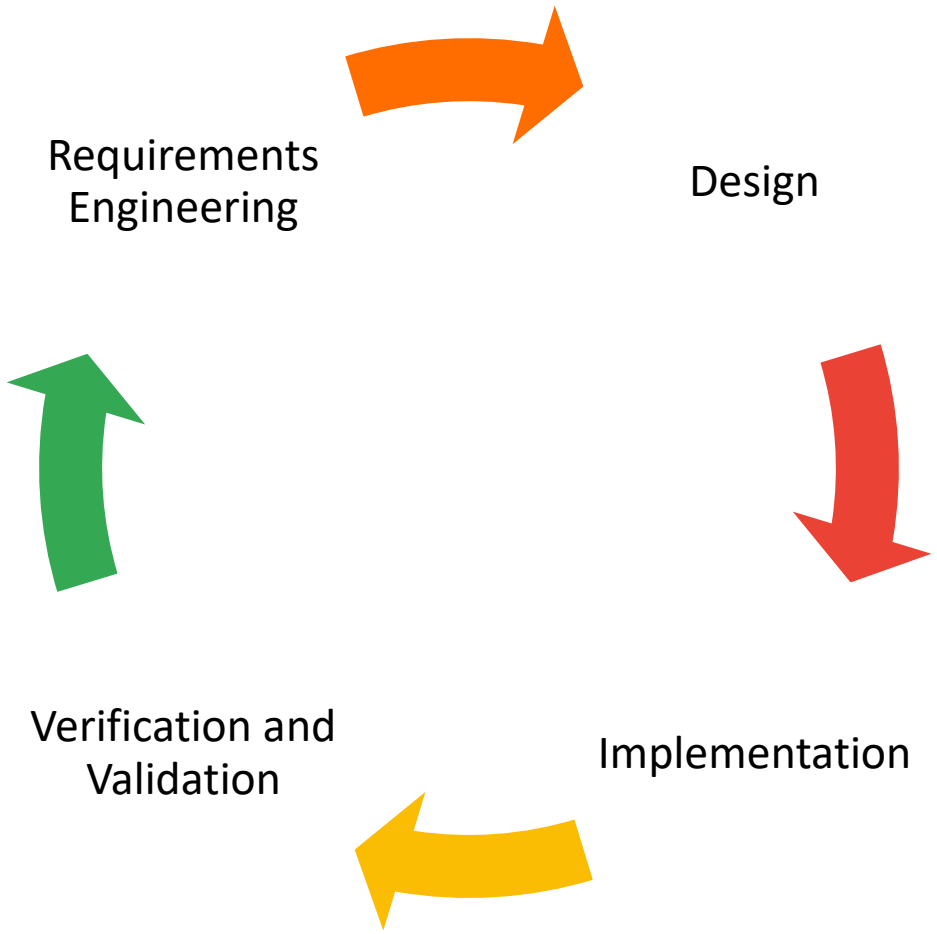
CENG 316 – Software Engineering

23 February 2022, @IZTECH

Agenda

- What is Software Design?
- Software Design Principles
- Software Architecture & Layered Architecture
- Responsibility-driven Design
- UML (Class, Interaction, Package Diagrams)
- A Case Study
- Sample Architectures on Cloud

Design in Software Development Life Cycle



What is Software Design?

a creative activity in which software components and their relationships are identified, based on customers' requirements

Ref: [Sommerville]

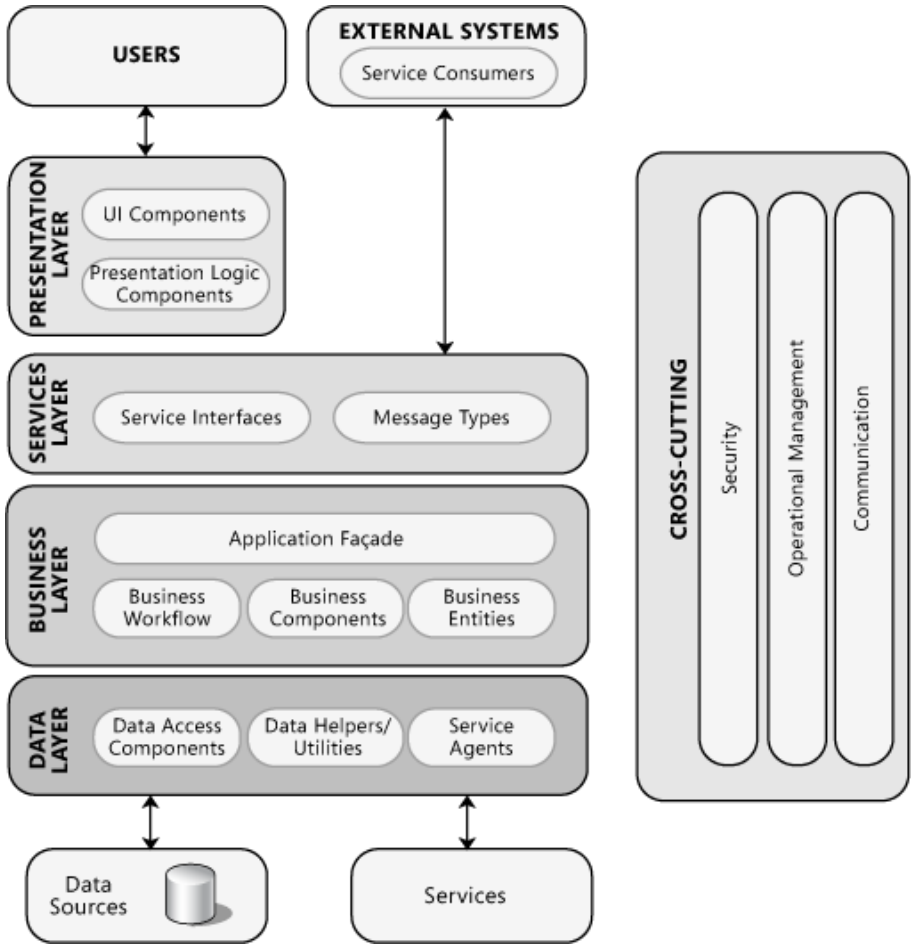
Level of Design

Architectural design describes how software is organized into components.

Detailed design describes the desired behavior of these components.

Ref: [SWEBOK]

A Sample Architecture



Ref: [Microsoft], available at https://msdn.microsoft.com/en-us/library/ee658124.aspx

Approaches to Software Design

Structured Design

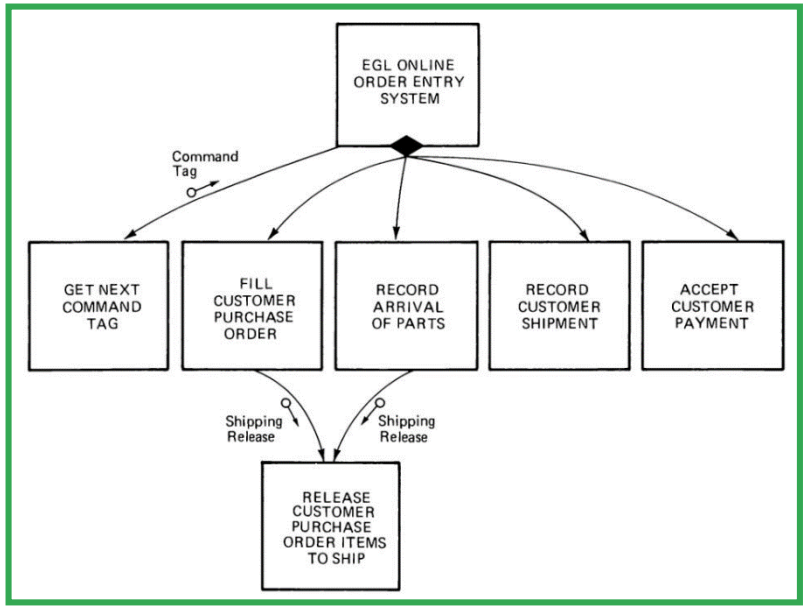
Object-oriented Design

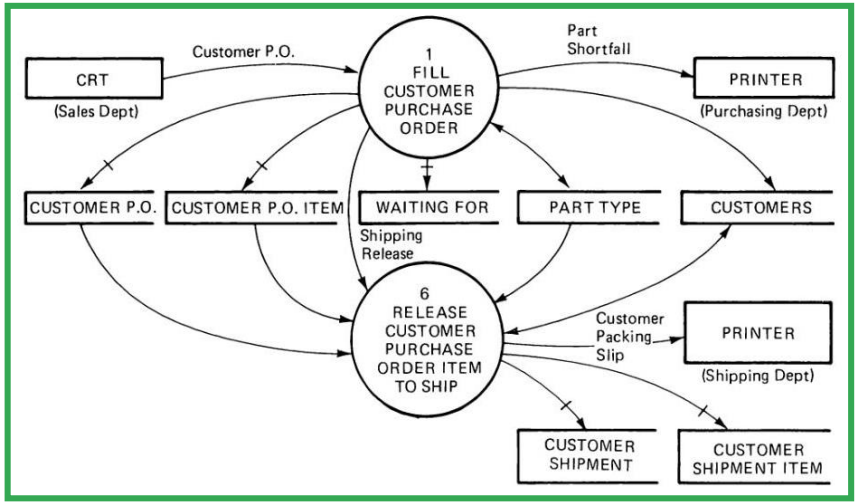
Structured Design

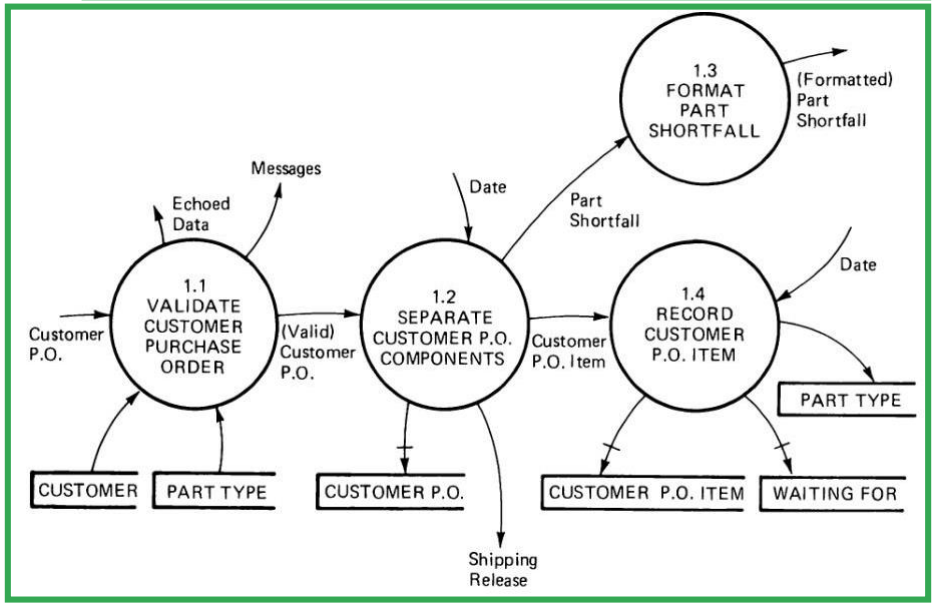
- is a disciplined approach to software design
- offers a set of strategies for developing a design solution from a well-defined statement of a problem
- seeks to cope with the complexity by partitioning and organizing
- concerned with the development of modules and the synthesis of these modules in a so-called “module hierarchy”

Ref: [Page-Jones 88]

Structured Design: Sample Diagrams





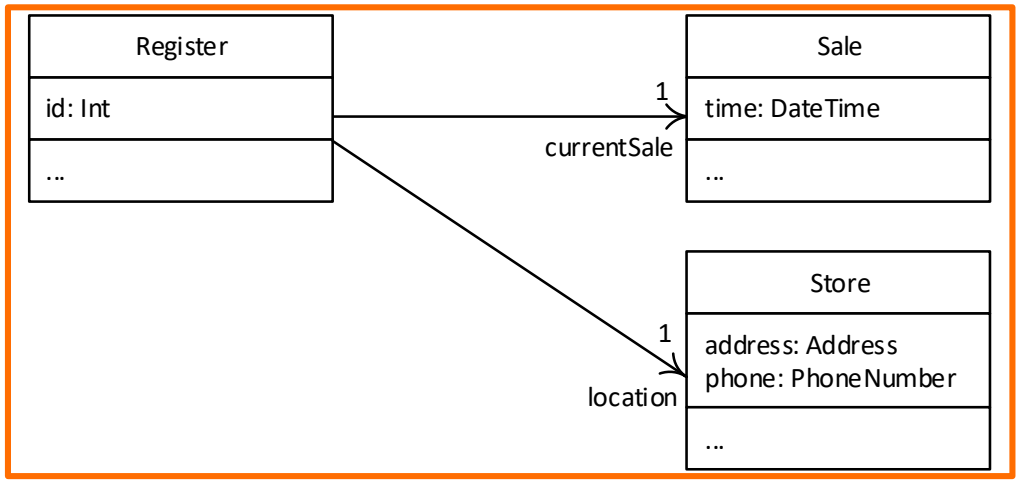


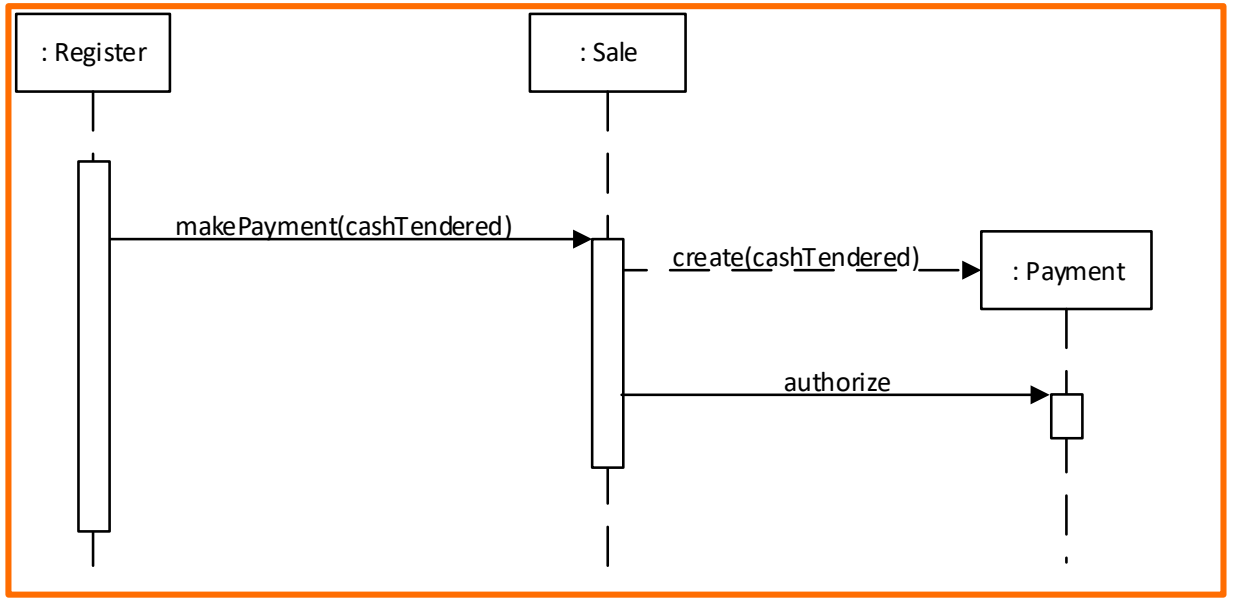
Ref: [Page-Jones 88]

Object-oriented (OO) Design

- is an approach to software design
- aims designing a software consisting of objects that are collaborating to produce an output which fulfills user needs

OO Design: Sample Diagrams





Ref: [Larman]

Software Design Principles

Abstraction

- a compulsory technique associated with problem solving
- reducing information so that one can focus on the “big picture”
  - *reducing → preserving relevant, forgetting irrelevant in a given context*

Decomposition and Modularization

- divide software into a number of smaller named components having well-defined interfaces that describe component interactions
- place different functionalities and responsibilities in different components

Ref: [SWEBOK]

Software Design Principles

Encapsulation and Information Hiding

- grouping and packaging the internal details of an abstraction and making those details inaccessible to external entities

Coupling and Cohesion

- Coupling: a measure of the interdependence among modules in a computer program
- Cohesion: a measure of the strength of association of the elements within a module

Ref: [SWEBOK]

Coupling

Low Coupling

tends to

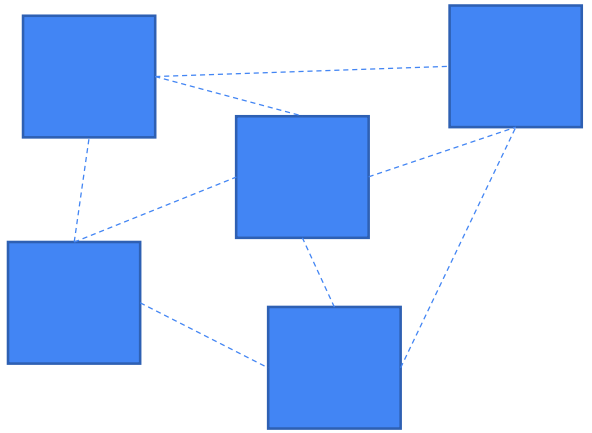
reduce

the time,

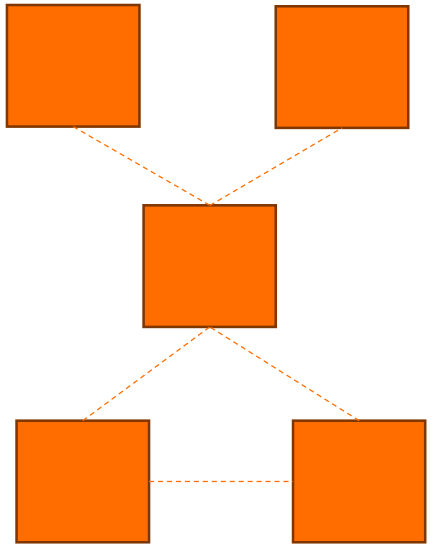
effort, and

defects

in building and modifying software.



highly coupled

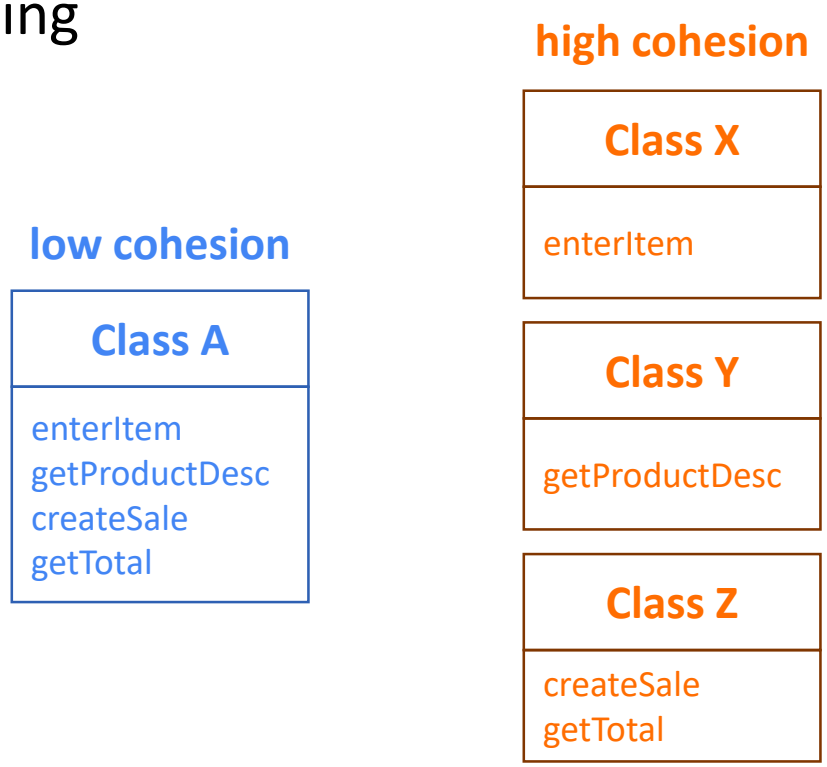


loosely coupled

Ref: [Larman]

Cohesion

- informally measures
  - how functionally related the operations of a software element are
  - how much work a software element is doing
- indicated by
  - the amount of code
  - the relatedness of the code are



Ref: [Larman]

Software Architecture

- is a high-level design representation of a software
- comprise software elements, relations among them and properties of both

Ref: [Bass]

Layer

a very coarse-grained grouping

of

classes, packages, or subsystems

that has

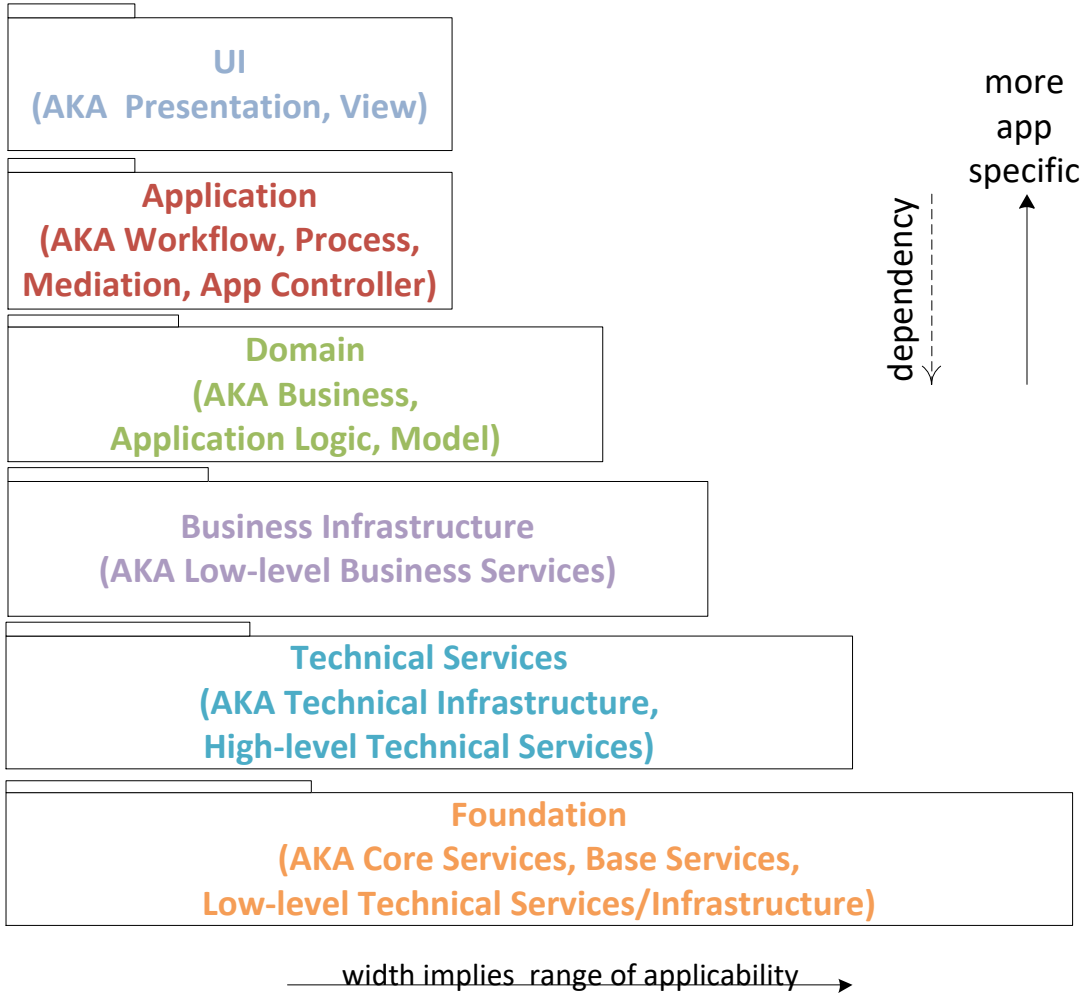
cohesive responsibility

for a

major aspect of the system

Ref: [Larman]

Common layers in an information system logical architecture



Ref: [Larman]

Typical Layers in an OO system

- **User Interface**
- **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.

Ref: [Larman]

Essential Ideas of Using Layers

- Organize the large-scale logical structure of a system into discrete layers of **distinct, related responsibilities**, with a **clean, cohesive separation of concerns** such that the "lower" layers are low-level and general services, and the higher layers are more application specific.
- Collaboration and coupling is from higher to lower layers; lower-to-higher layer coupling is avoided.

Ref: [Larman]

Benefits of Using Layers

- Separation of concerns: a separation of high from low-level services, and of application-specific from general services
  - reduces coupling and dependencies,
  - improves cohesion,
  - increases reuse potential, and
  - increases clarity.
- Related complexity is encapsulated and decomposable.
- Some layers (UI, Application, Domain layers) can be replaced with new implementations.
- Some layers (primarily the Domain and Technical Services) can be distributed.
- Development by teams is aided because of the logical segmentation.

Ref: [Larman]

Responsibility-driven design (RDD)

software objects

have

responsibilities, i.e., an abstraction of what they do

Think of software objects as similar to people with responsibilities who collaborate with other people to get work done. RDD leads to viewing an OO design as a *community of collaborating responsible objects*.

- How should responsibilities be allocated to classes of objects?
- How should objects collaborate?

Ref: [Larman]

Responsibilities

- **Doing** responsibilities of an object include:
  - doing something itself, such as creating an object or doing a calculation
  - initiating action in other objects
  - controlling and coordinating activities in other objects
- **Knowing** responsibilities of an object include:
  - knowing about private encapsulated data
  - knowing about related objects
  - knowing about things it can derive or calculate

a responsibility ≠ a method
Methods fulfill responsibilities

Ref: [Larman]

Unified Modeling Language (UML)

a visual language for

specifying,

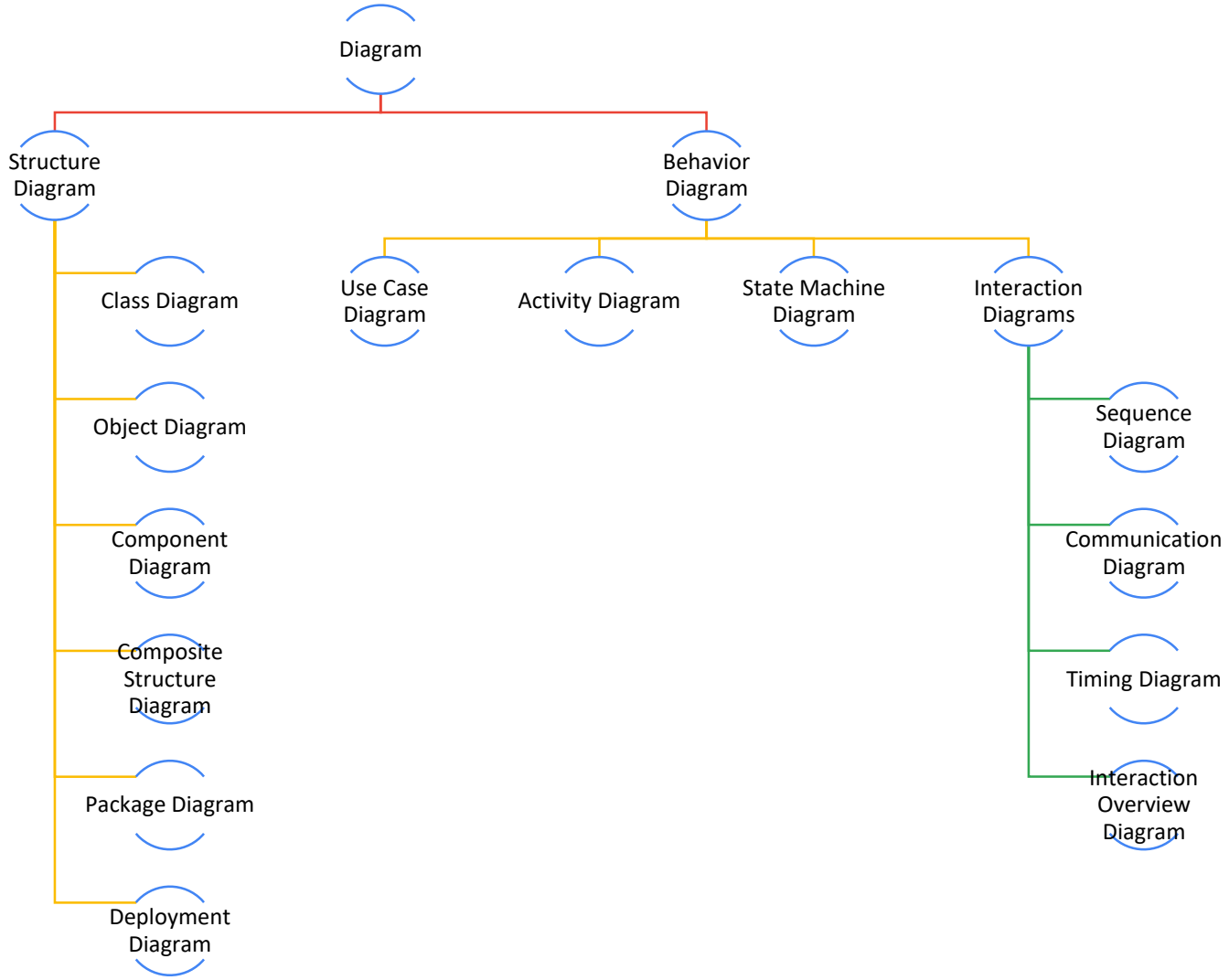
constructing and

documenting

the artifacts of systems

Ref: [Larman]

UML Diagram Types



Ref: [Fowler]

UML in Software Engineering Process

Requirements Analysis

- Use case diagram
- Class diagram
- Activity diagram
- State diagram

Design

- Class diagram
- Interaction (Sequence/Communication) diagram
- Package diagram
- State diagram
- Deployment diagram

Documentation

- How much?

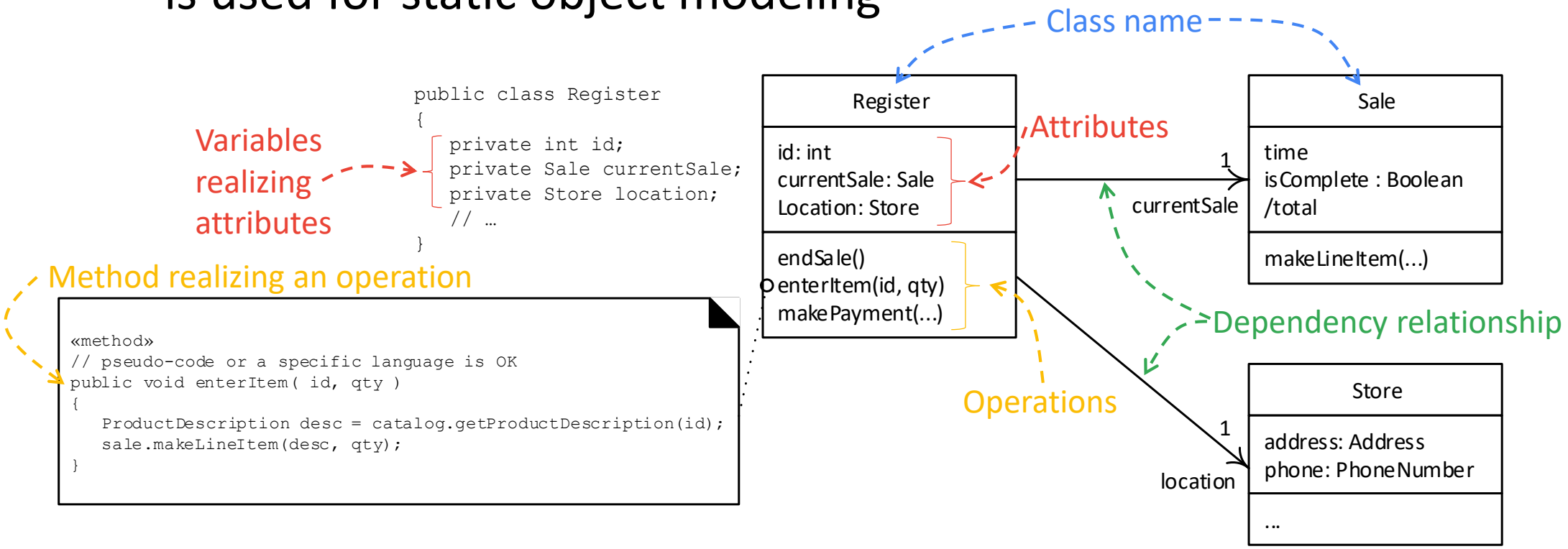
Understanding Legacy Code

- Clarify key points

Ref: [Fowler]

Class Diagram

- illustrates classes, interfaces, and their associations
- is used for static object modeling



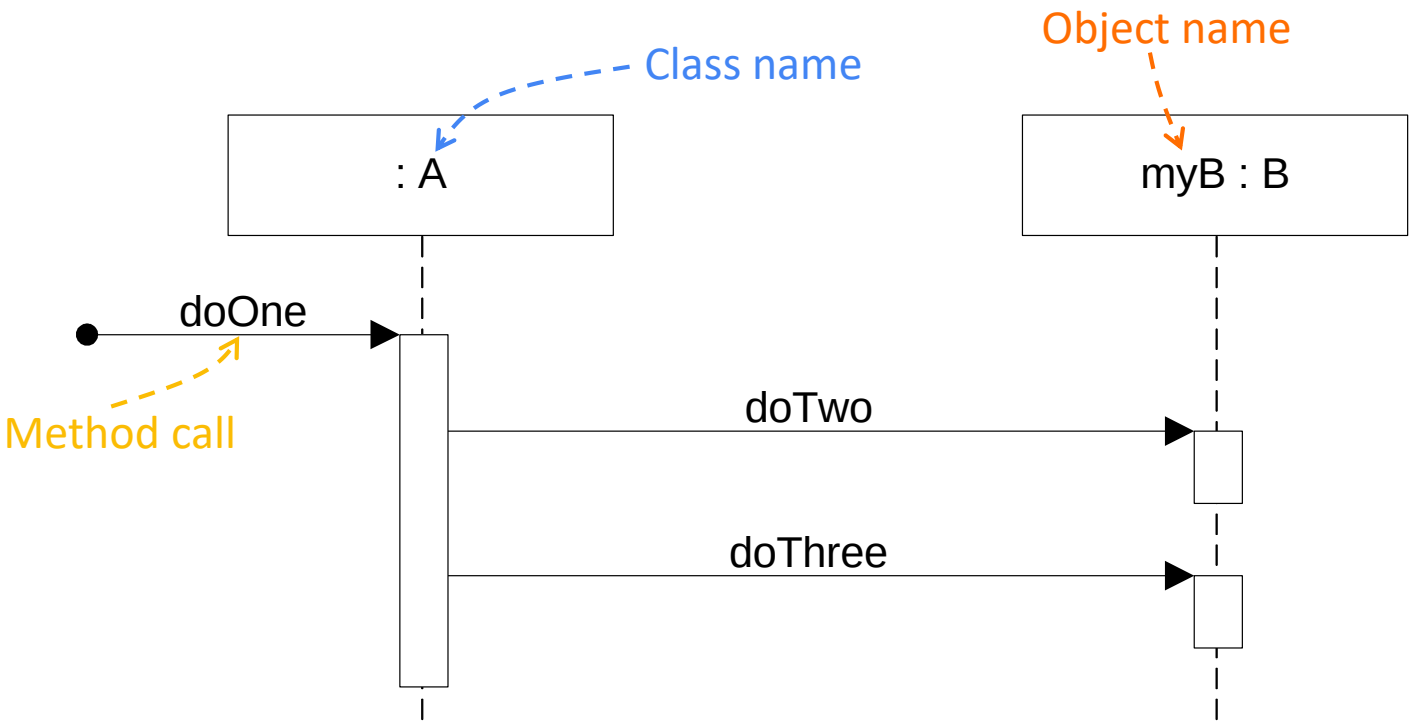
Ref: [Larman]

Interaction Diagram

- illustrates how objects interact via messages
- is used for dynamic object modeling
- has two common types: sequence and communication interaction diagrams

Ref: [Larman]

A Sample Sequence Diagram

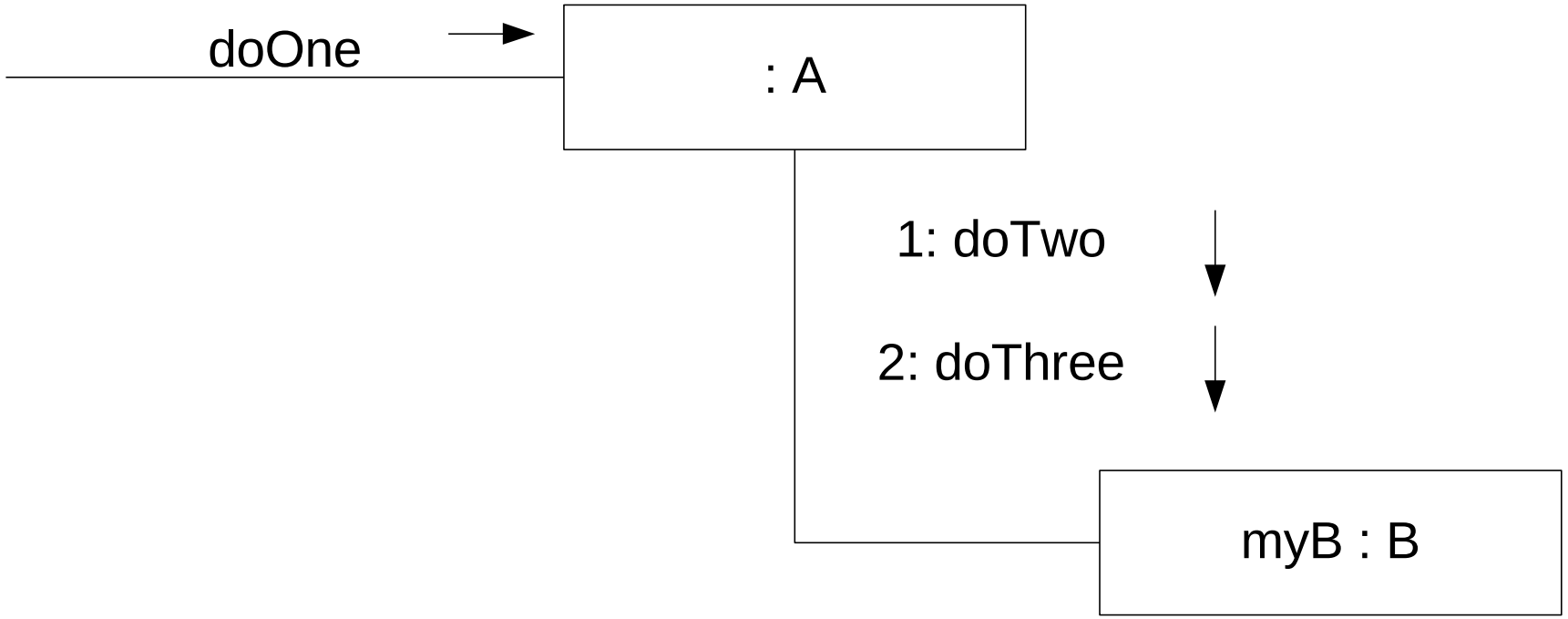



```
public class A
{
        private B myB = new B();

        public void doOne()
        {
                myB.doTwo();
                myB.doThree();
        }
// …
}
```

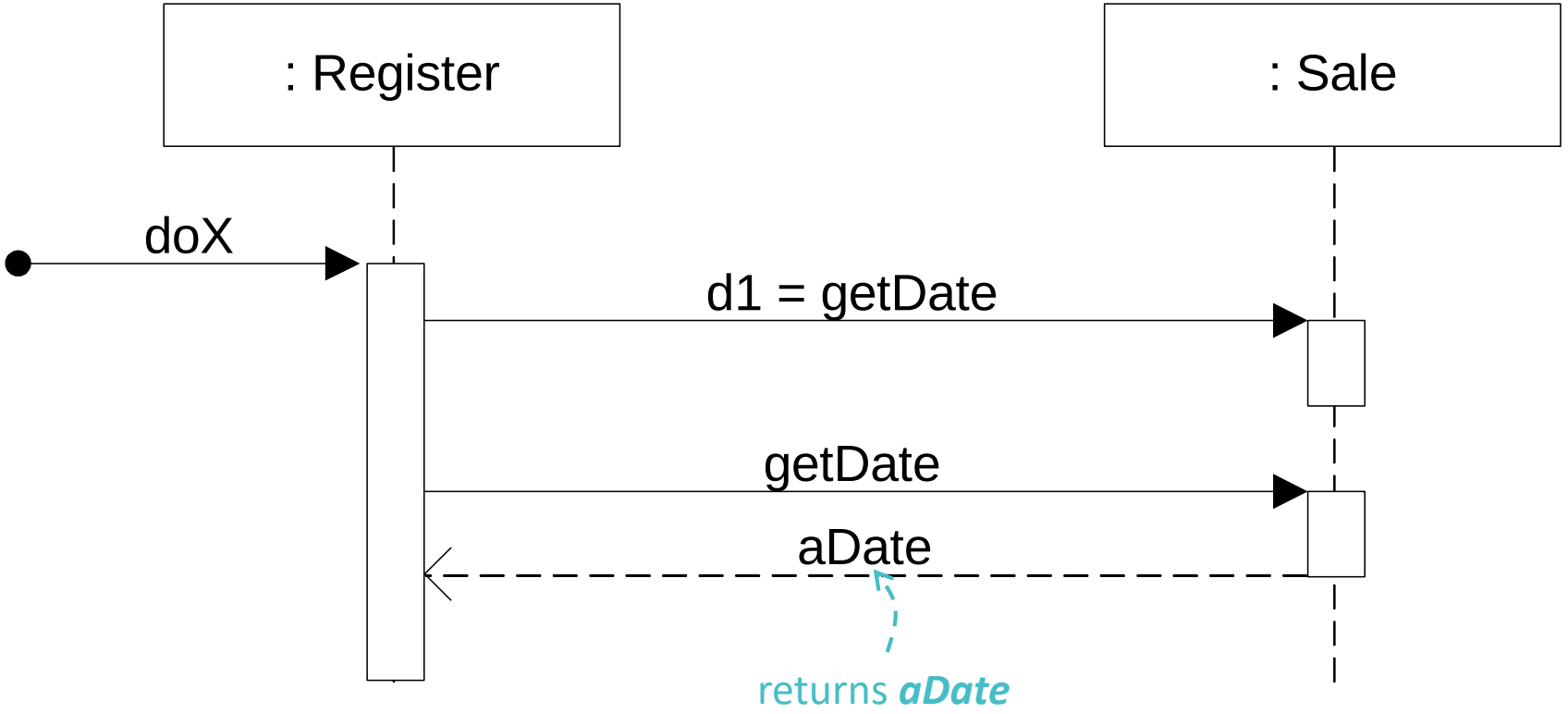
Ref: [Larman]

A Sample Communication Diagram



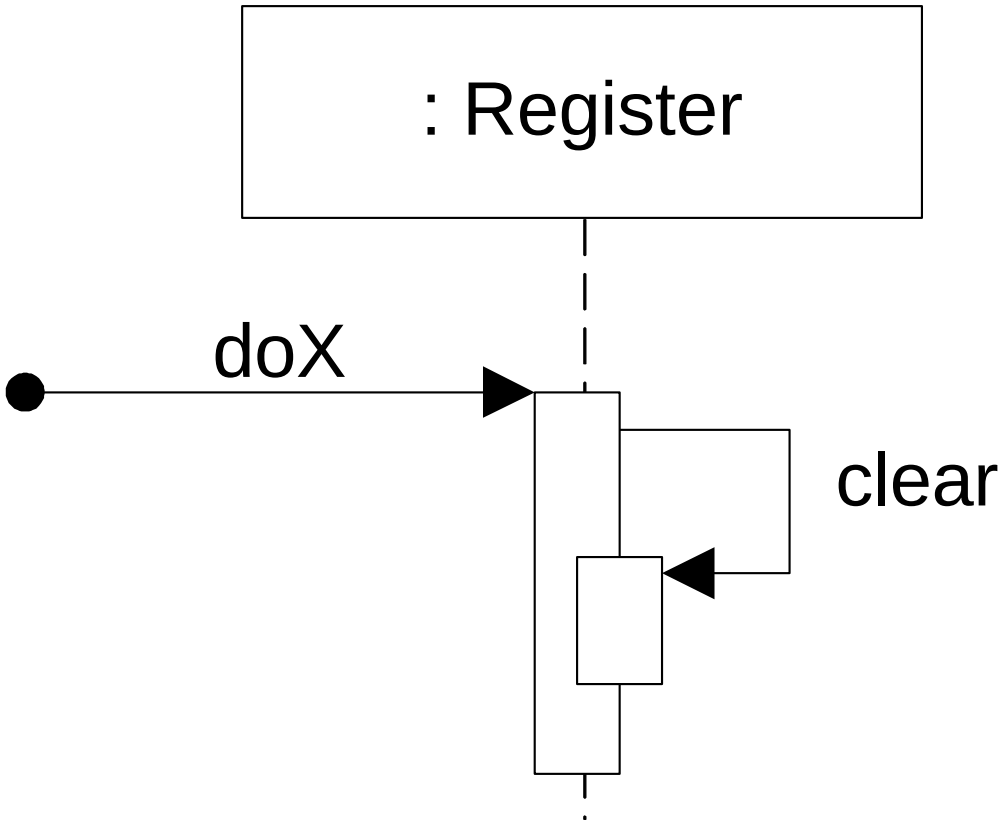
Ref: [Larman]

Illustrating Returns



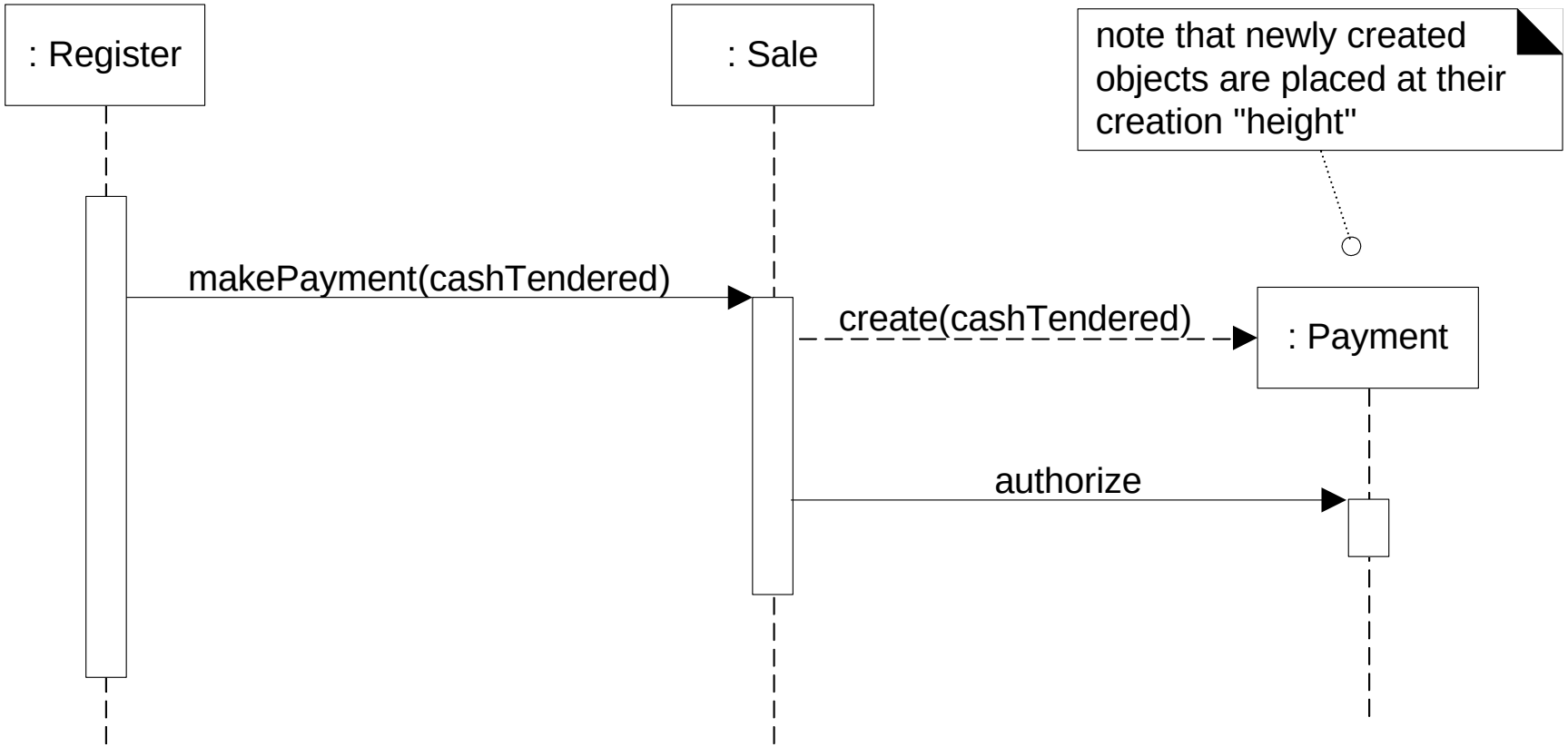
Ref: [Larman]

Messages to “self” or “this”



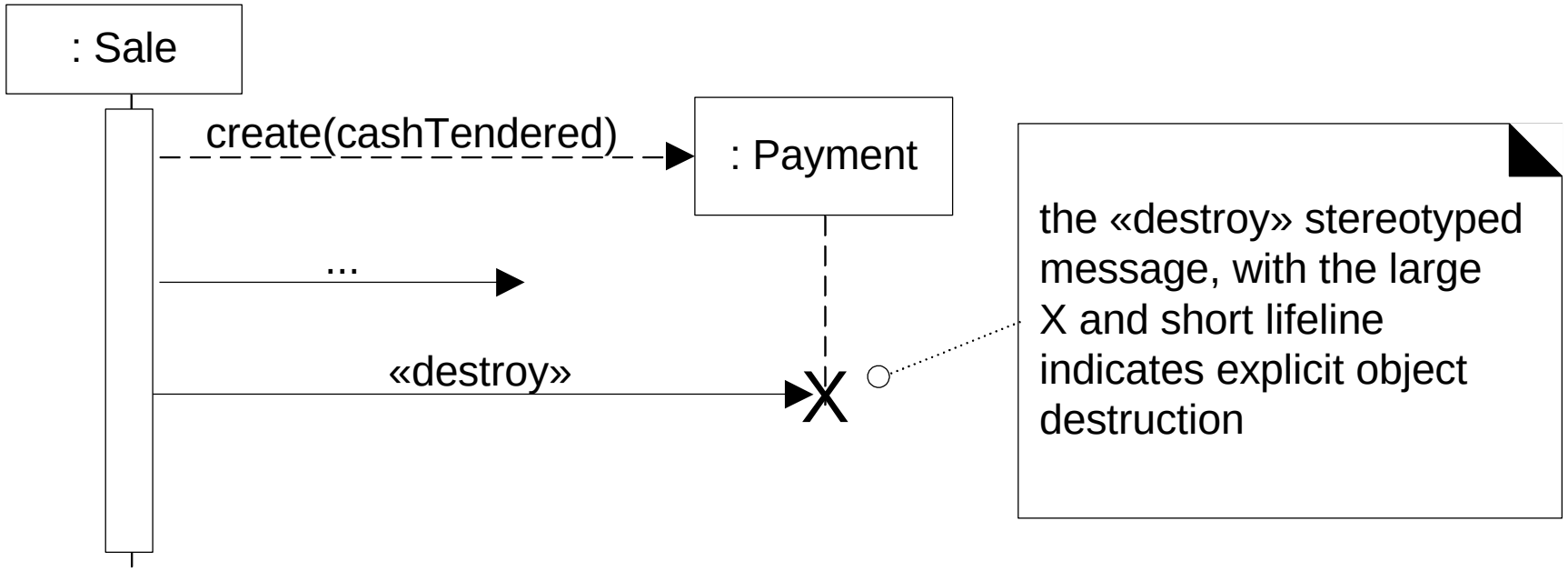
Ref: [Larman]

Creation of Instances



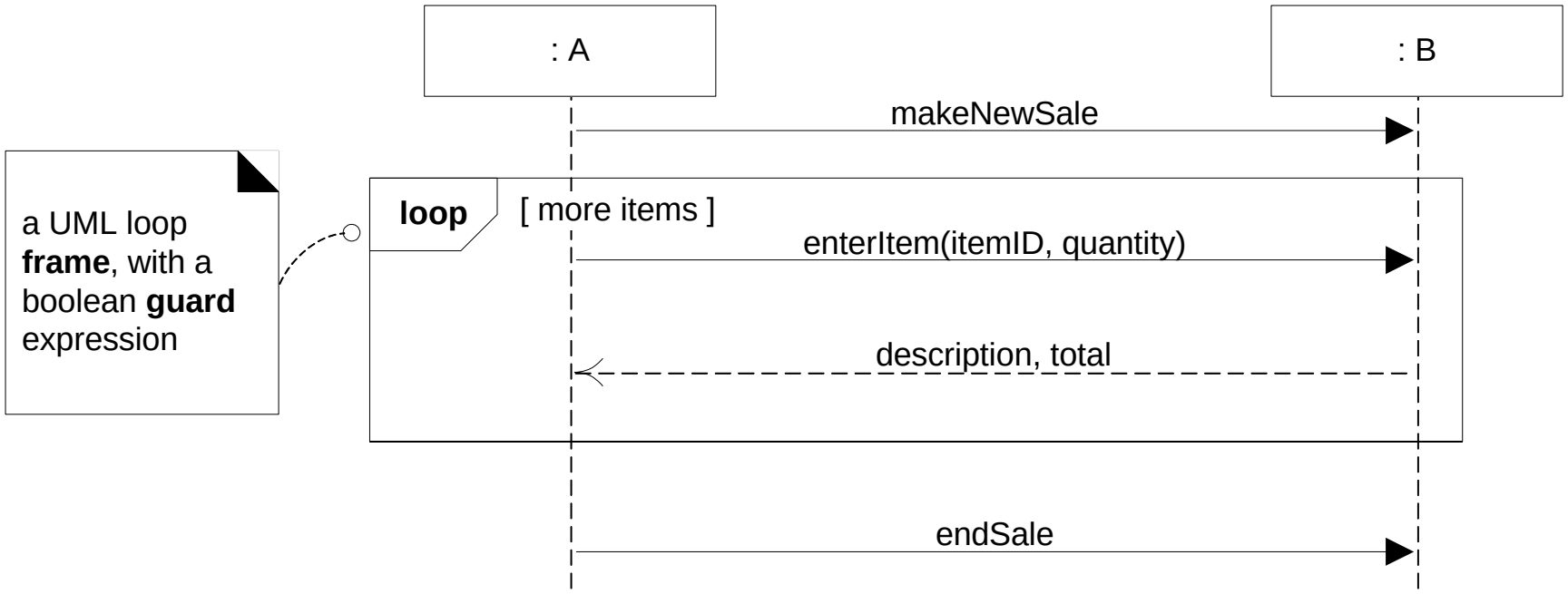
Ref: [Larman]

Object Destruction



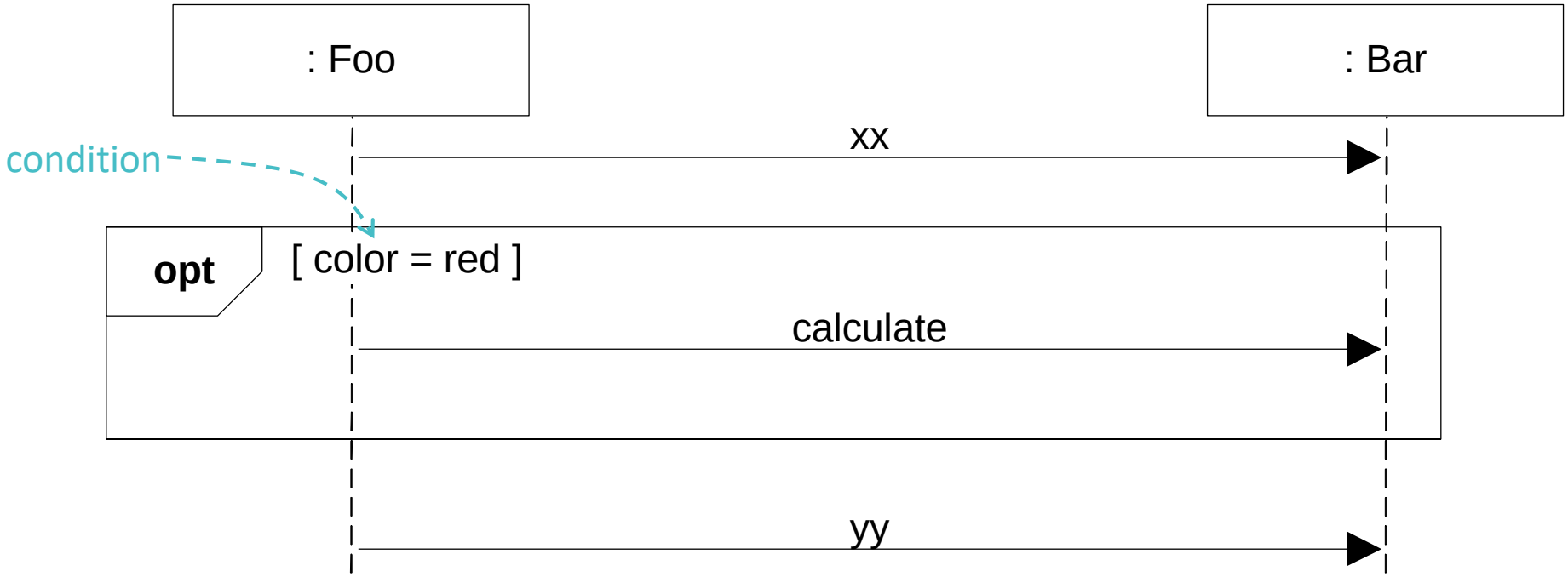
Ref: [Larman]

Loop Frame Notation



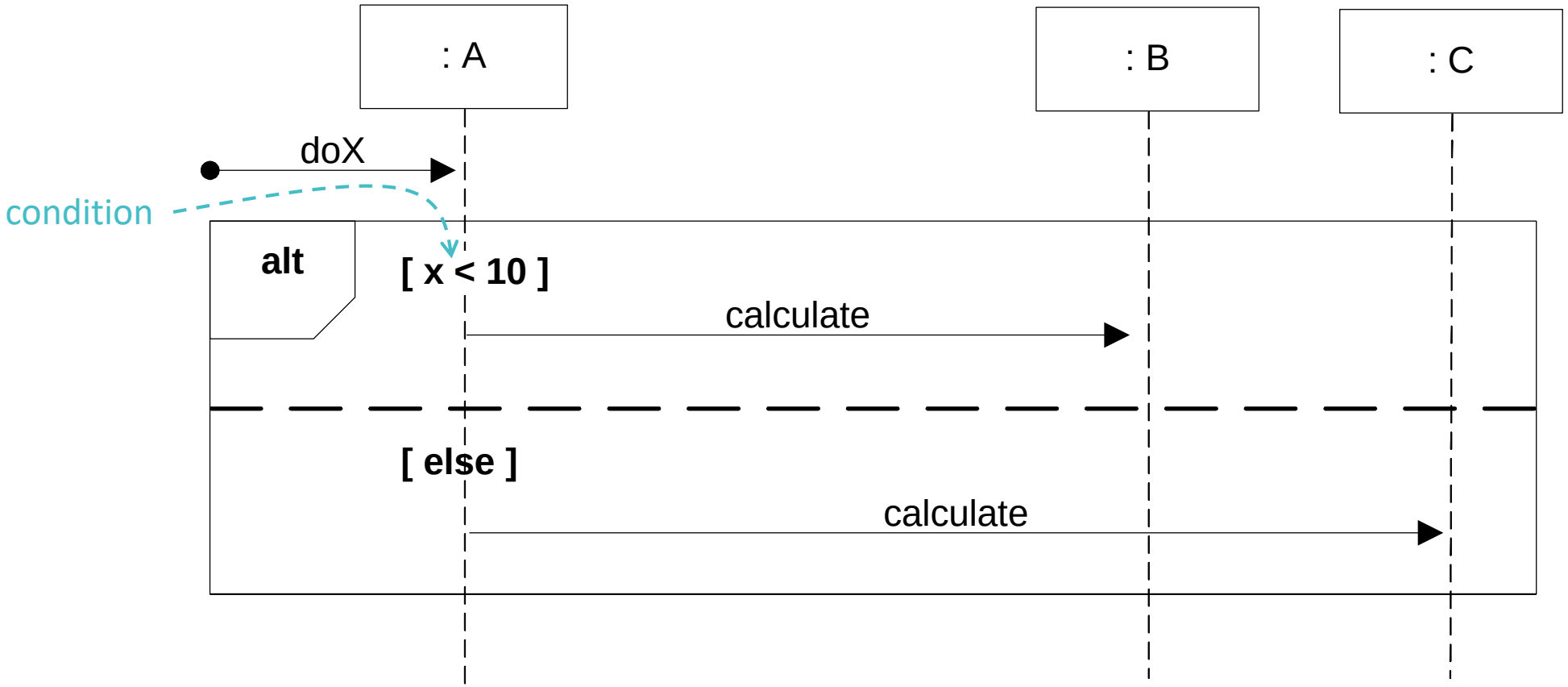
Ref: [Larman]

Conditional Messages



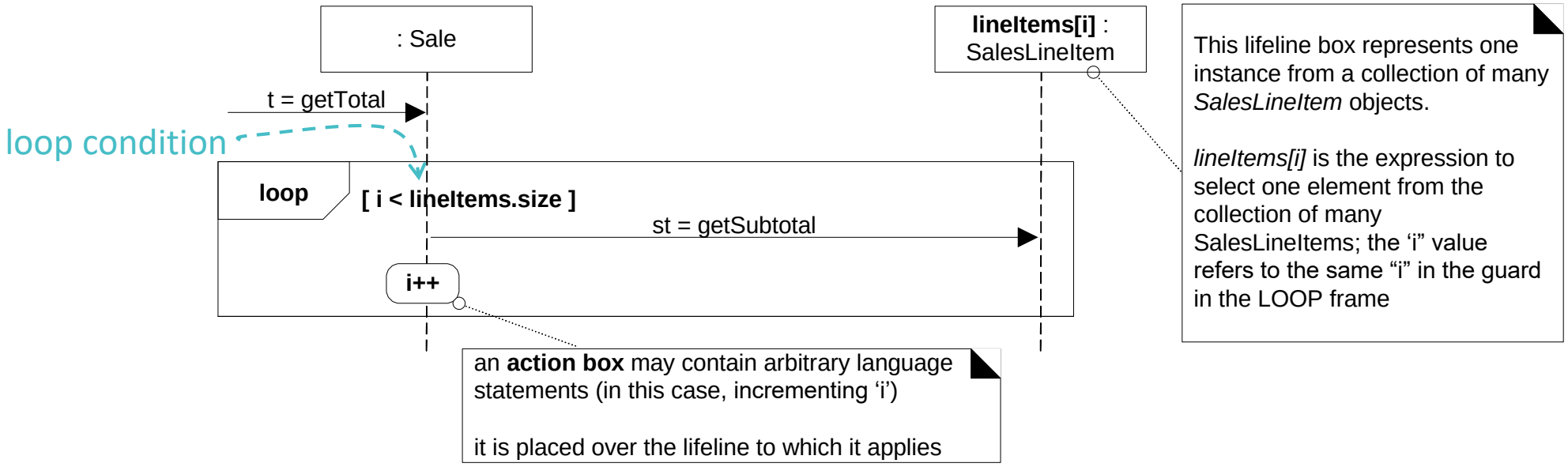
Ref: [Larman]

Mutually Exclusive Conditional Messages



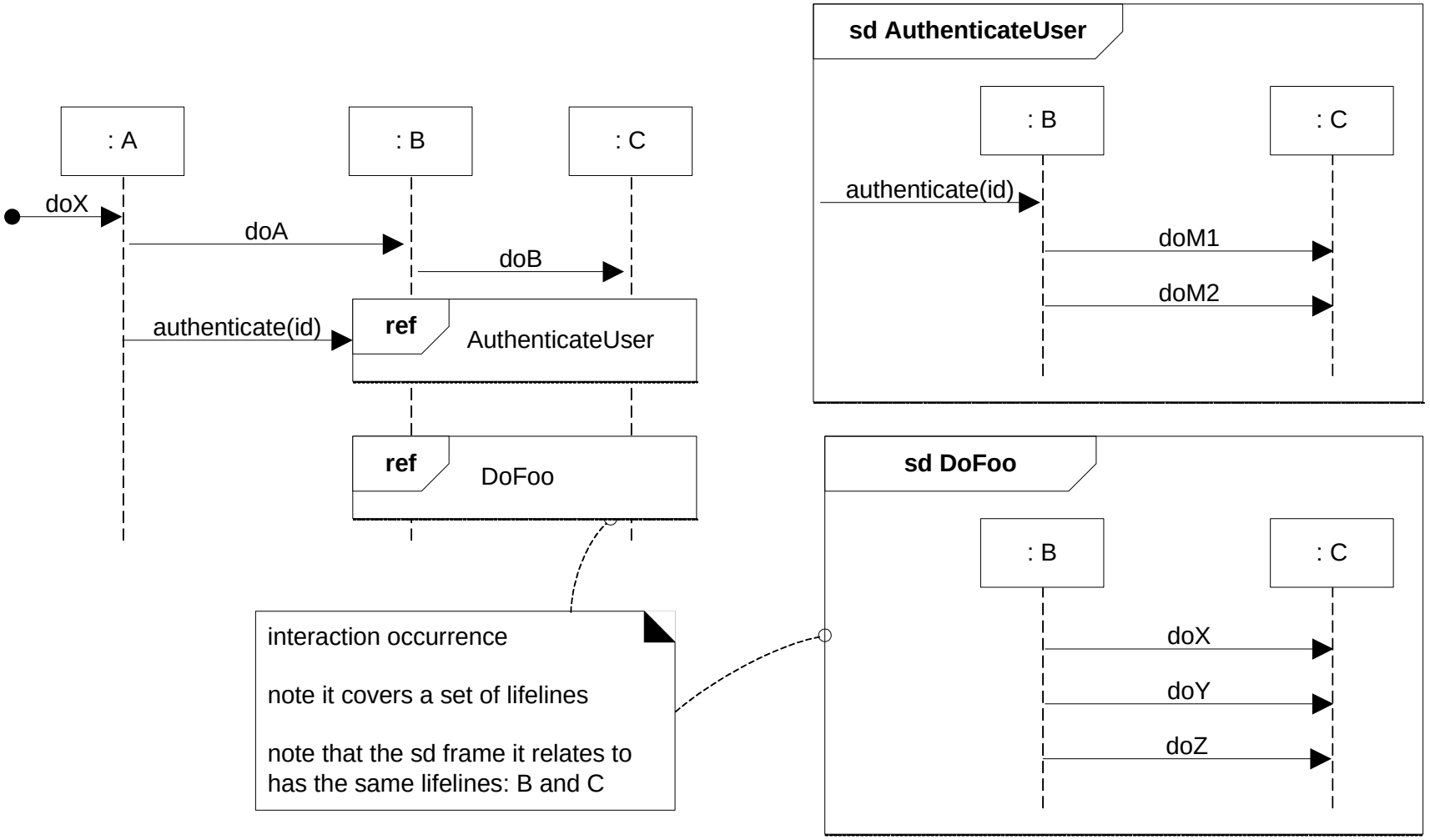
Ref: [Larman]

Iteration over a Collection



Ref: [Larman]

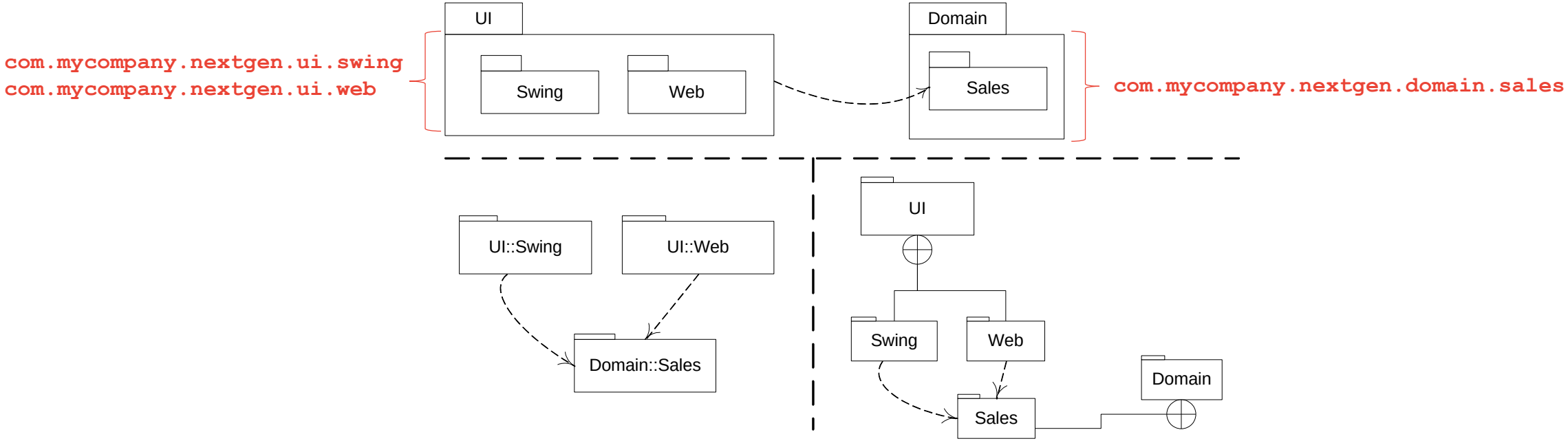
Relating Interaction Diagrams



Ref: [Larman]

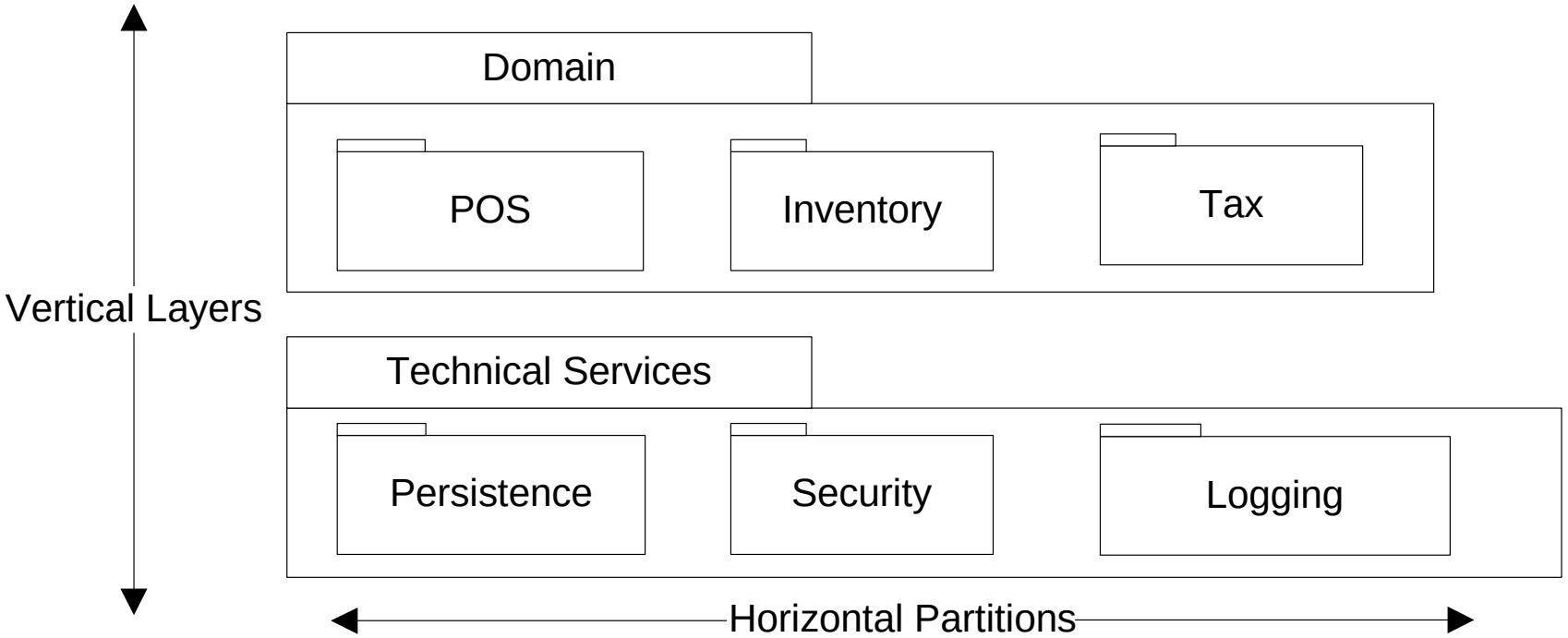
UML Package Diagram

- provides a way to group elements (classes, other packages, use cases, etc.)
- is often used to illustrate the logical architecture of a system



Ref: [Larman]

Layers and Partitions



Ref: [Larman]

UML versus Design Principles

The critical design tool

for

software development

is

a mind well educated in design principles.

It is not the UML or any other technology.

Ref: [Larman]

Case Study
NextGen Point-of-Sale (POS) System

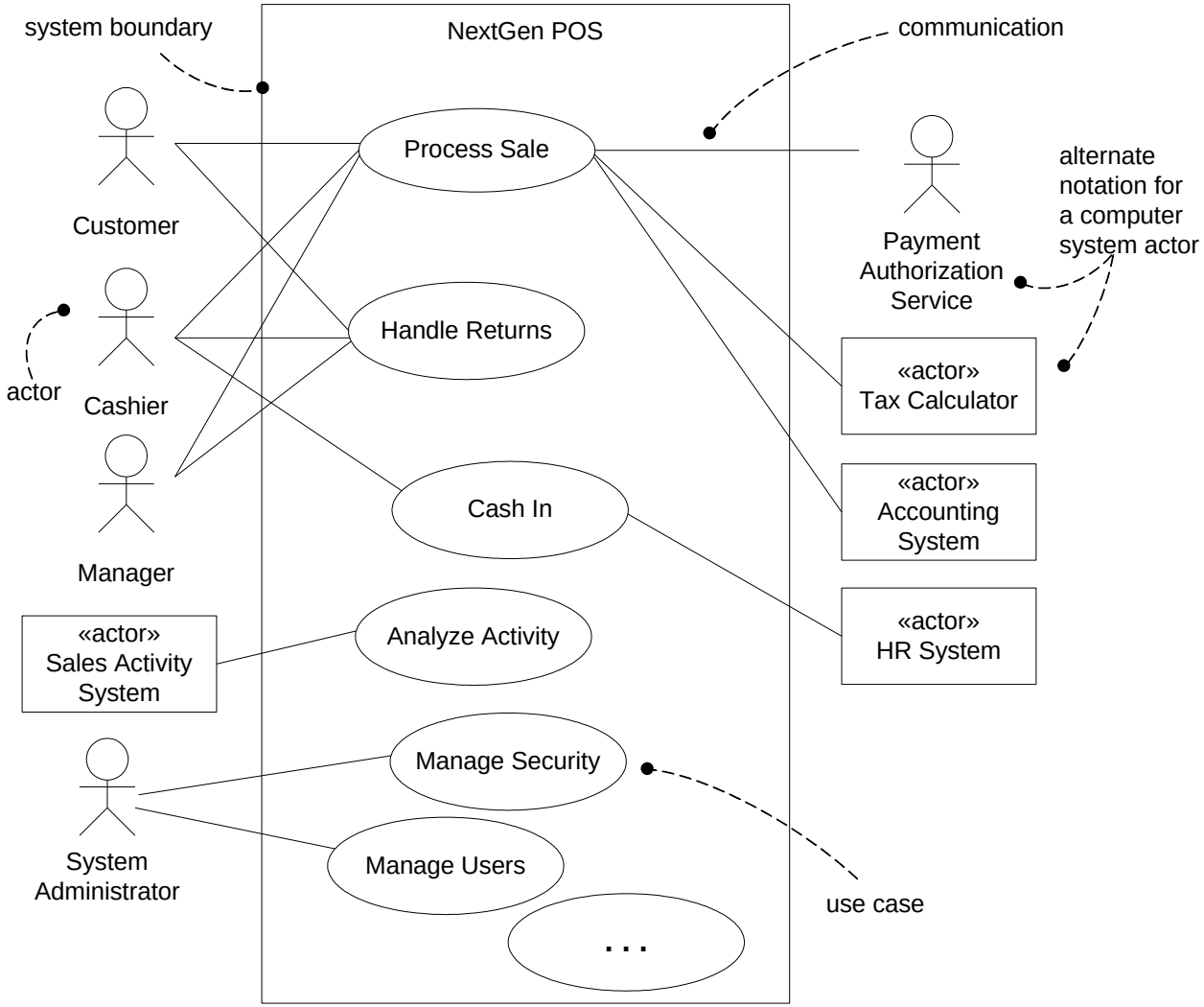
A POS system is a computerized application used (in part) to record sales and handle payments; it is typically used in a retail store. It includes hardware components such as a computer and bar code scanner, and software to run the system. It interfaces to various service applications, such as a third-party tax calculator and inventory control. These systems must be relatively fault tolerant; that is, even if remote services are temporarily unavailable (such as the inventory system), they must still be capable of capturing sales and handling at least cash payments (so that the business is not crippled).

A POS system increasingly must support multiple and varied client-side terminals and interfaces. These include a thin-client Web browser terminal, a regular personal computer with something like a Java Swing graphical user interface, touch screen input, wireless PDAs, and so forth.

Image credit:
https://www.chetu.com/blogs/payments/point-of-sale-software-key-features-development-cost.php

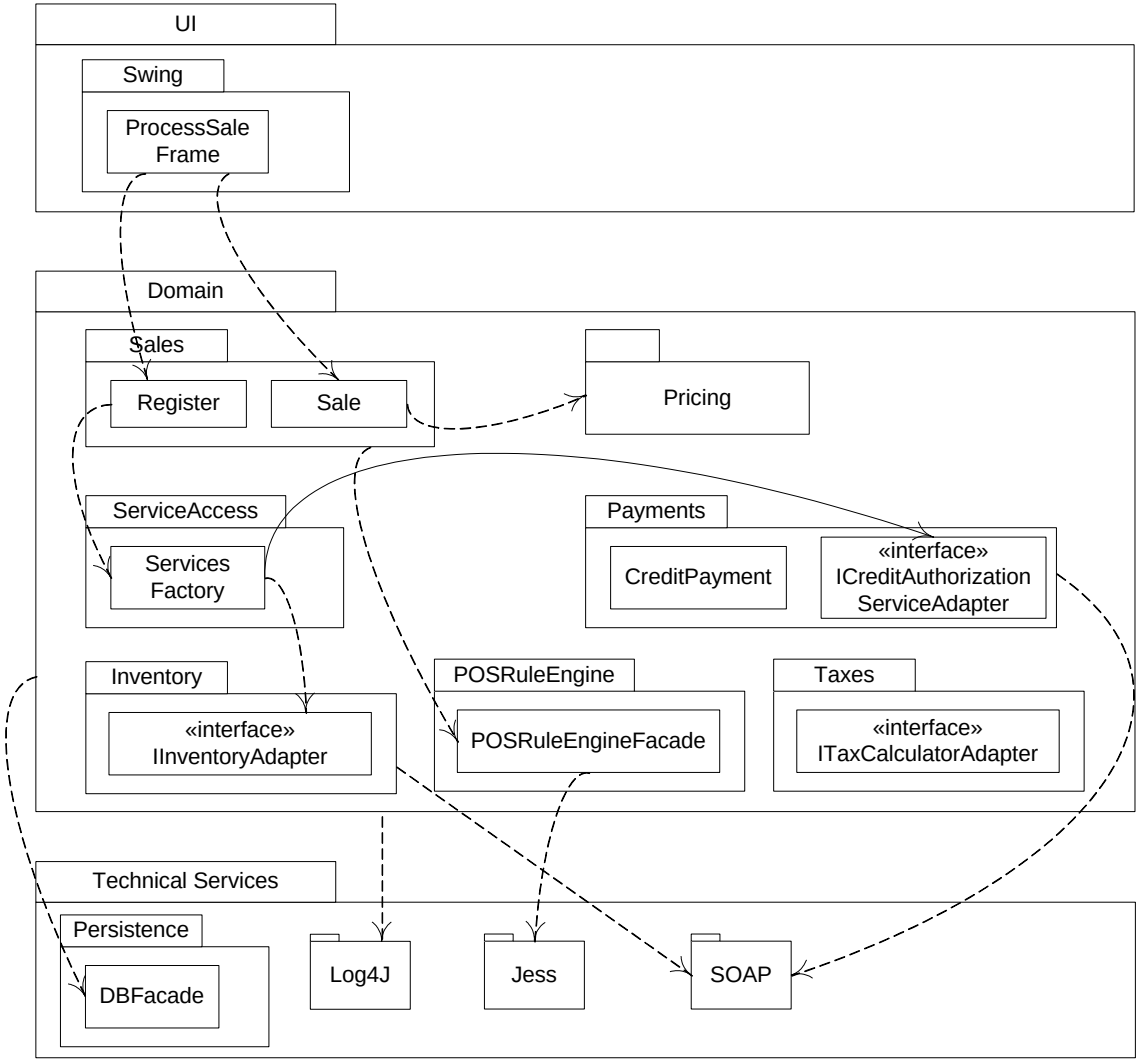
Ref: [Larman]

Requirements: Partial Use Case Diagram



Ref: [Larman]

Partial Software Architecture for NextGen POS System



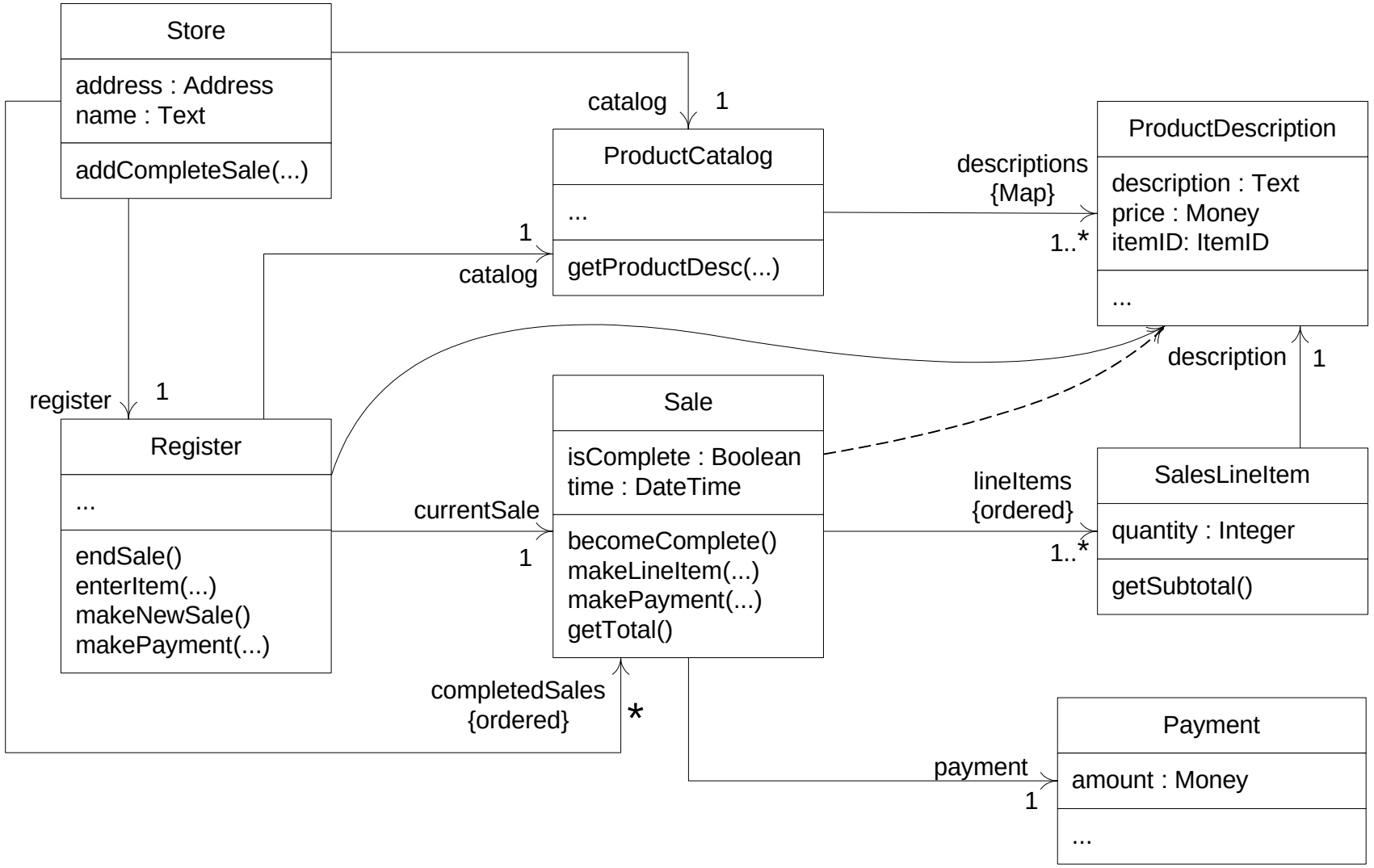
Ref: [Larman]

Use Case UC1: Process Sale
Main Success Scenario

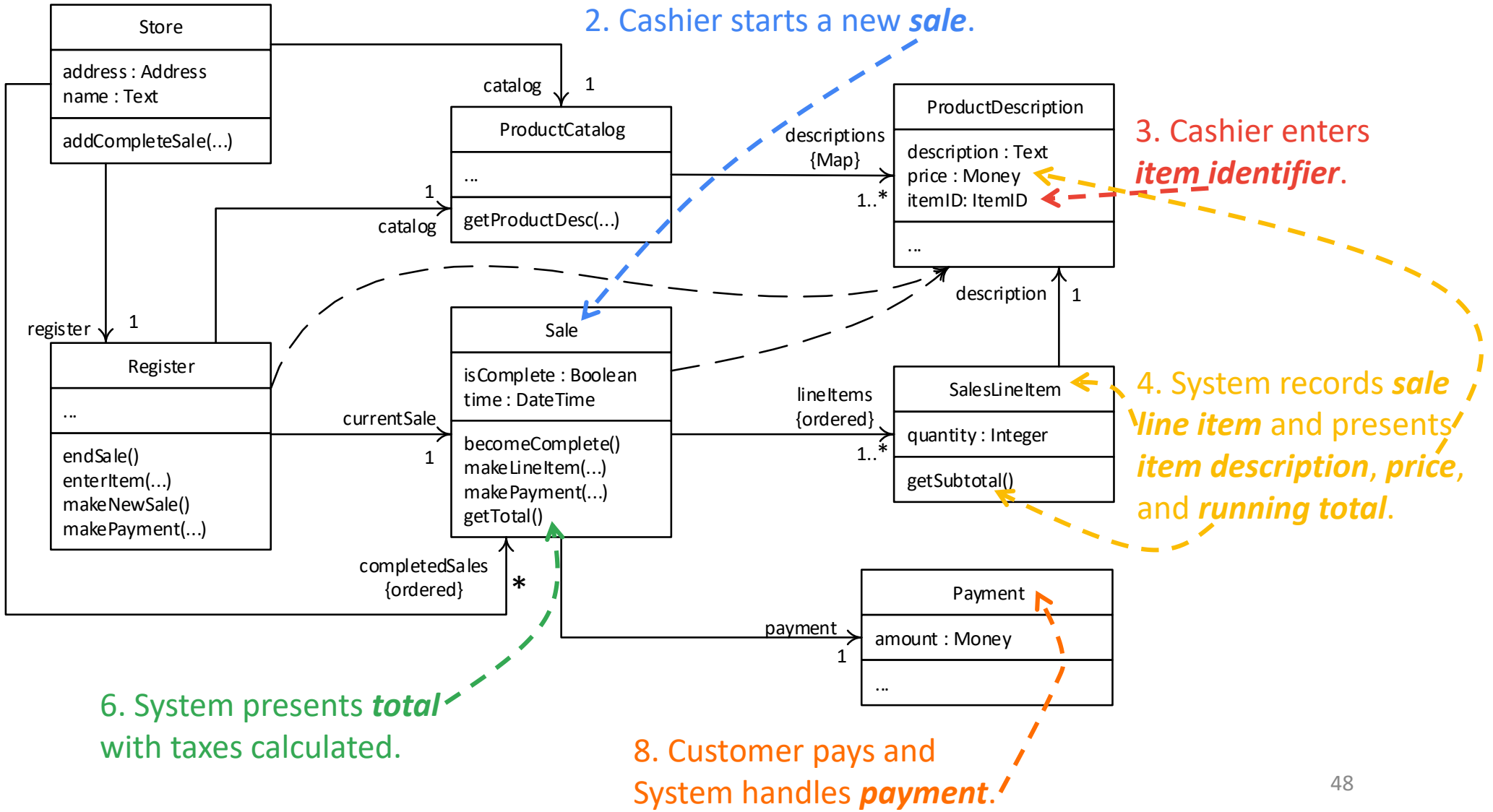
1. Customer arrives at POS checkout with goods and/or services to purchase.
2. Cashier starts a new sale.
3. Cashier enters item identifier.
4. System records sale line item and presents item description, price, and running total. Price calculated from a set of price rules.
5. Cashier repeats steps 3-4 until indicates done.
6. System presents total with taxes calculated.
7. Cashier tells Customer the total, and asks for payment.
8. Customer pays and System handles payment.
9. System logs completed sale and sends sale and payment information to the external Accounting system (for accounting and commissions) and Inventory system (to update inventory).
10. System presents receipt.
11. Customer leaves with receipt and goods (if any).

Ref: [Larman]

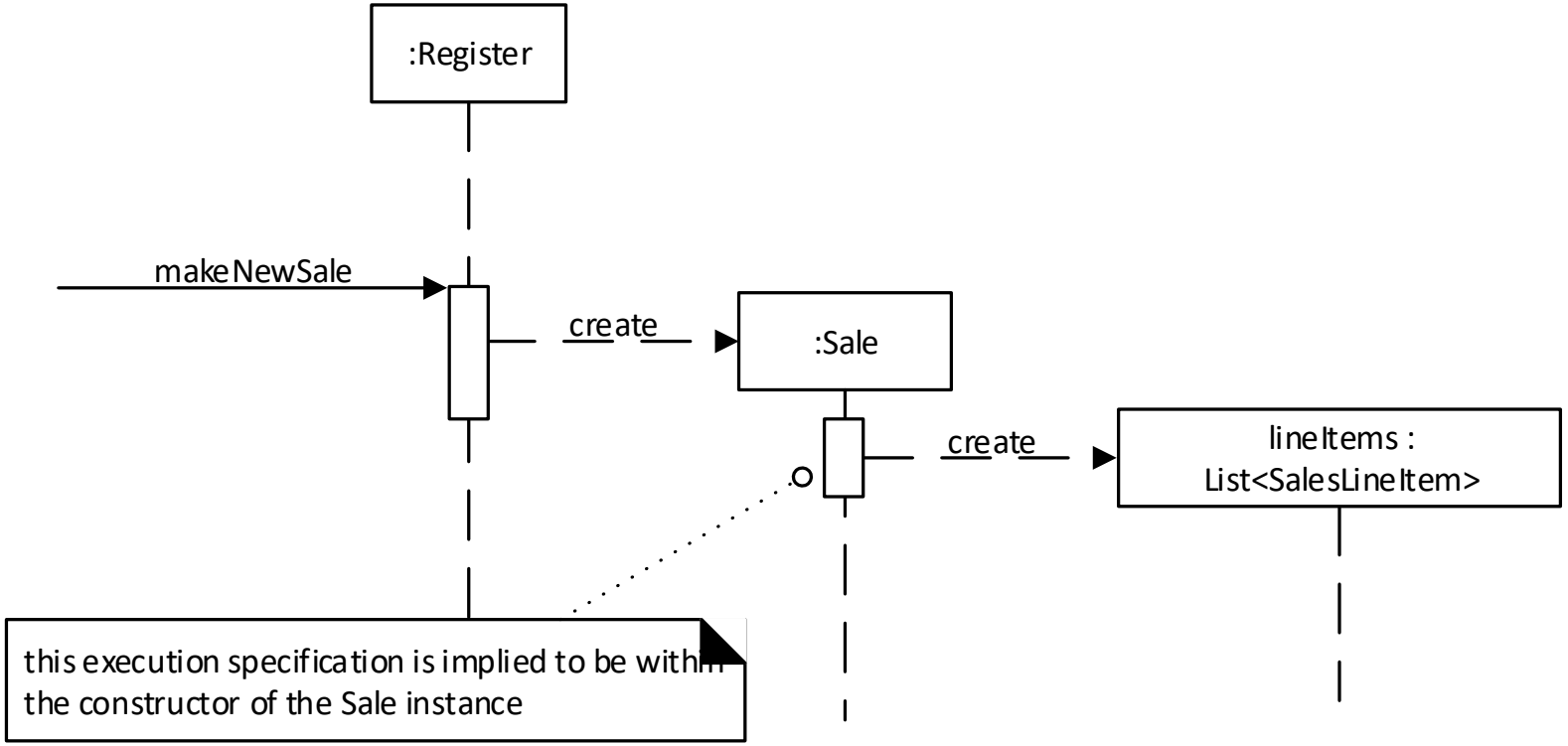
Partial Class Diagram



Partial Class Diagram mapped to Use Case Steps



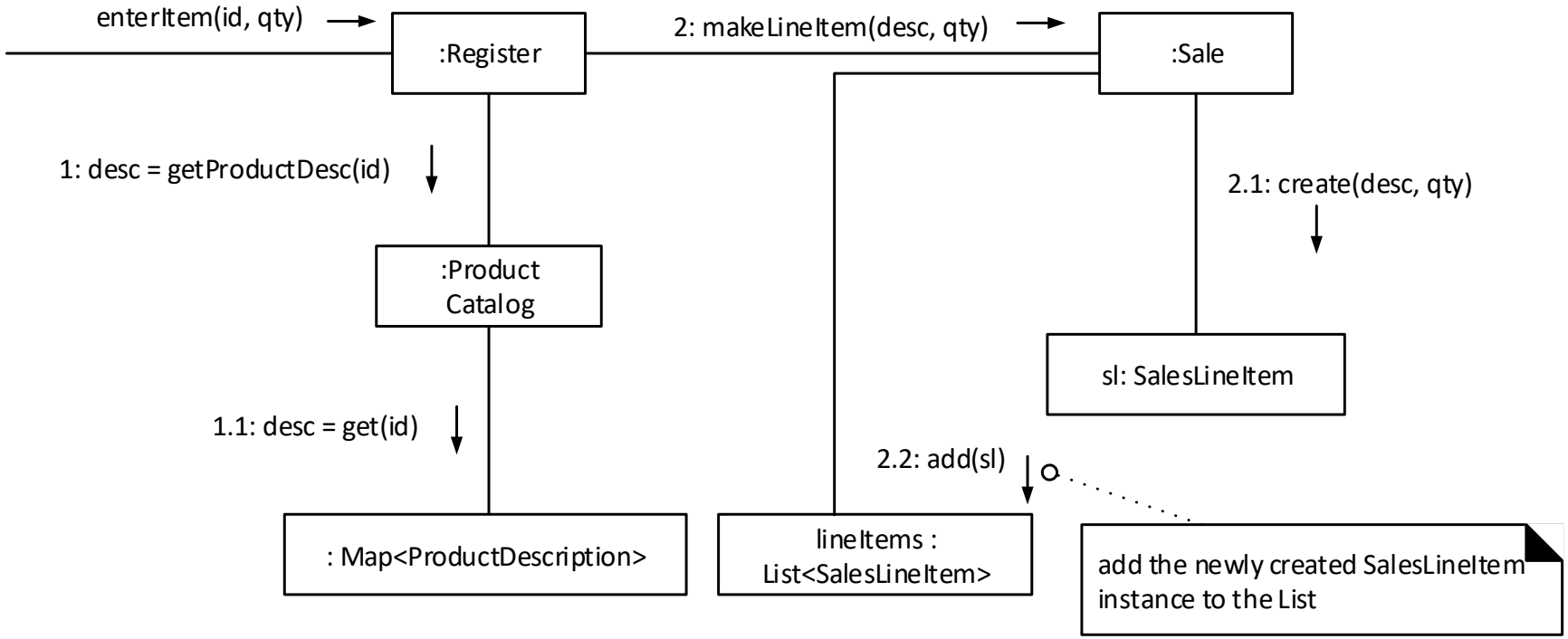
Interaction Diagram for "*starting a sale*"



1. Customer arrives at POS checkout with goods and/or services to purchase.
2. Cashier starts a new sale.

Ref: [Larman]

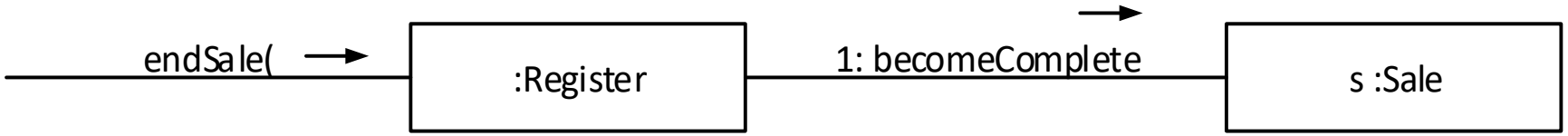
Interaction Diagram for "*entering Items*"



3. Cashier enters item identifier.
4. System records sale line item and presents item description, price, and running total. Price calculated from a set of price rules.
5. Cashier repeats steps 3-4 until indicates done.

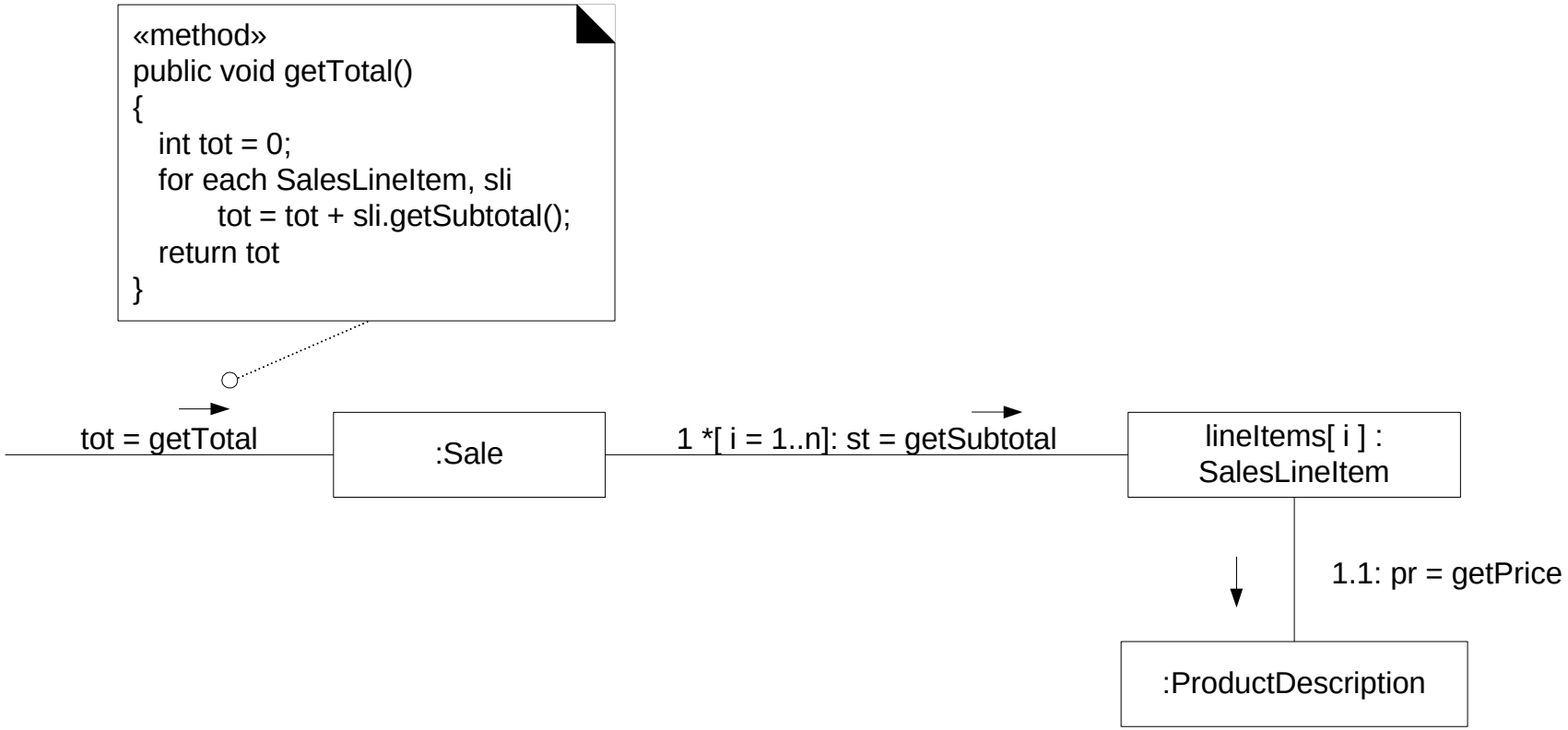
Ref: [Larman]

Interaction Diagram for "*ending sale*"



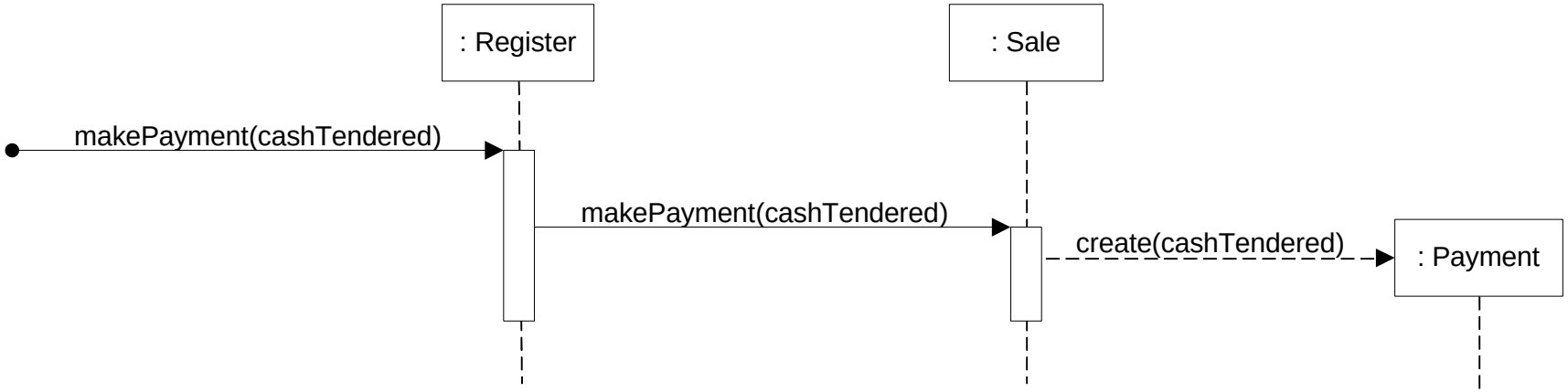
5. Cashier repeats steps 3-4 until indicates done.

Interaction Diagram for "*calculating sale total*"



6. System presents total with taxes calculated.

Interaction Diagram for "*making payment*"



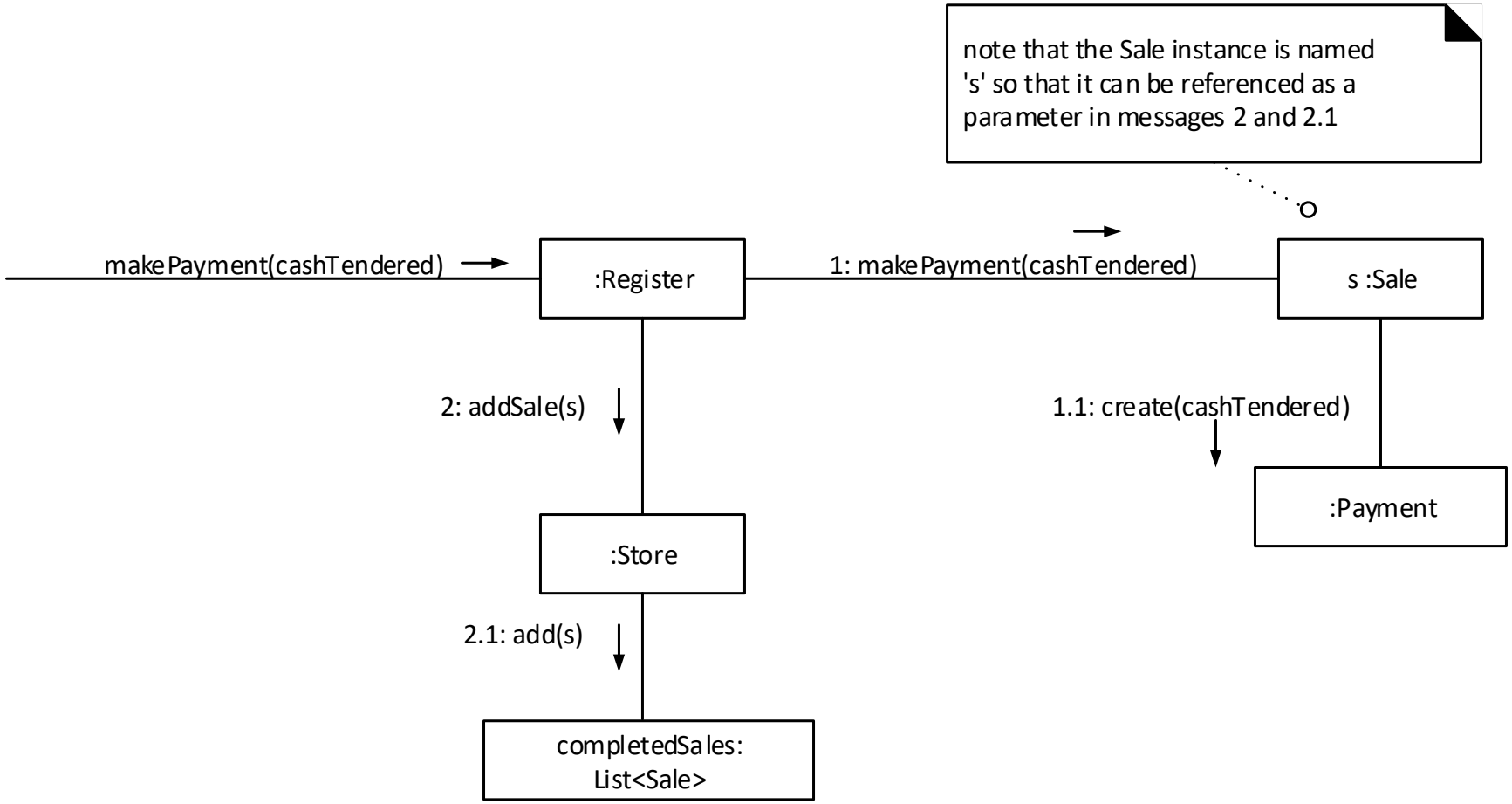
7. Cashier tells Customer the total, and asks for payment.
8. Customer pays and System handles payment.

```
public class Sale
{
  private Payment payment;

  public void makePayment( Money cashTendered )
  {
    payment = new Payment( cashTendered );
    //…
  }
  // …
}
```

Ref: [Larman]

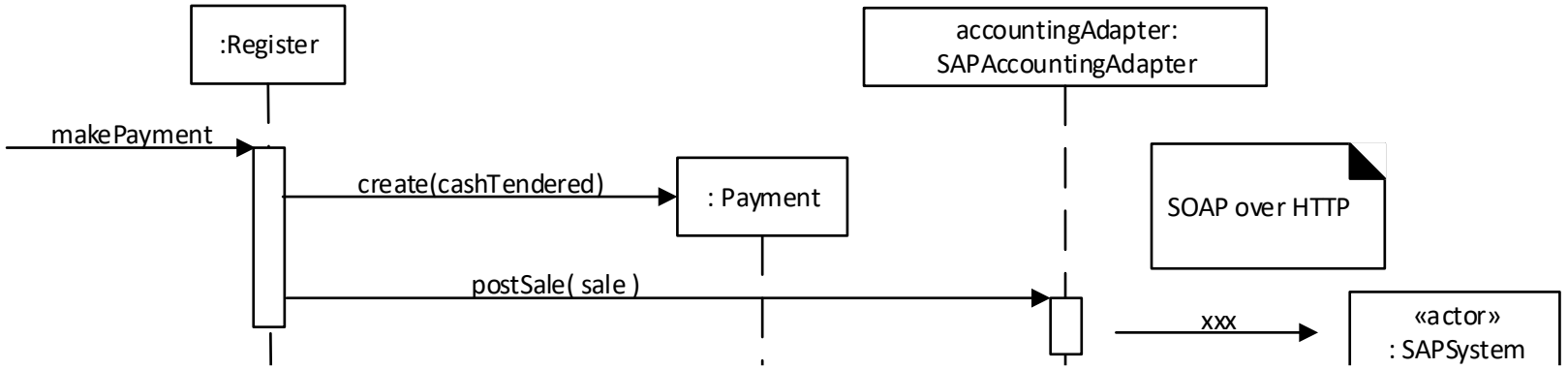
Interaction Diagram for "*logging a completed sale*"



9. **System logs completed sale** and sends sale and payment information to the external Accounting system (for accounting and commissions) and Inventory system (to update inventory).

Ref: [Larman]

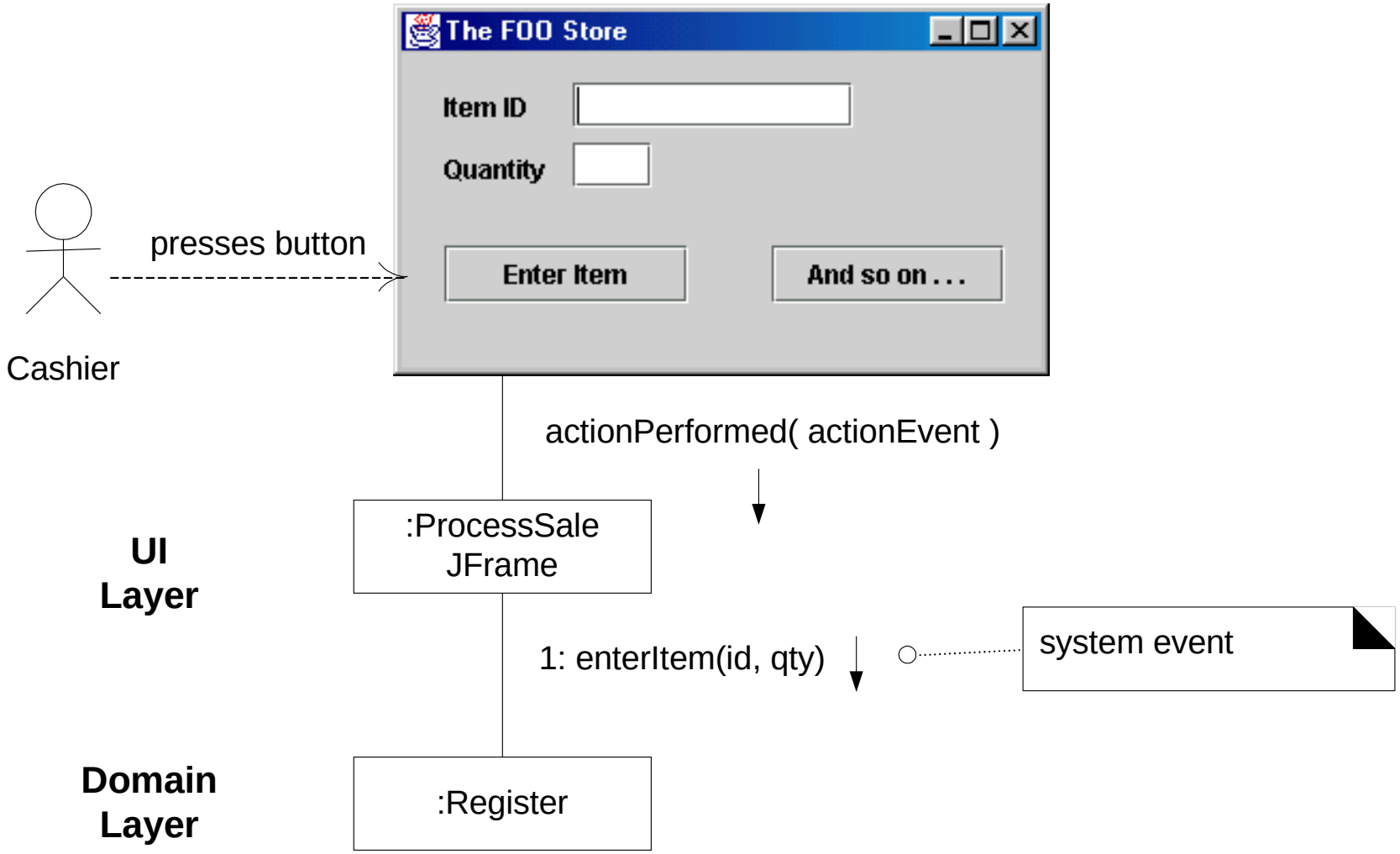
Interaction Diagram for "*sending sale info to accounting system*"



9. System logs completed sale and **sends sale and payment information to the external Accounting system (for accounting and commissions)** and Inventory system (to update inventory).

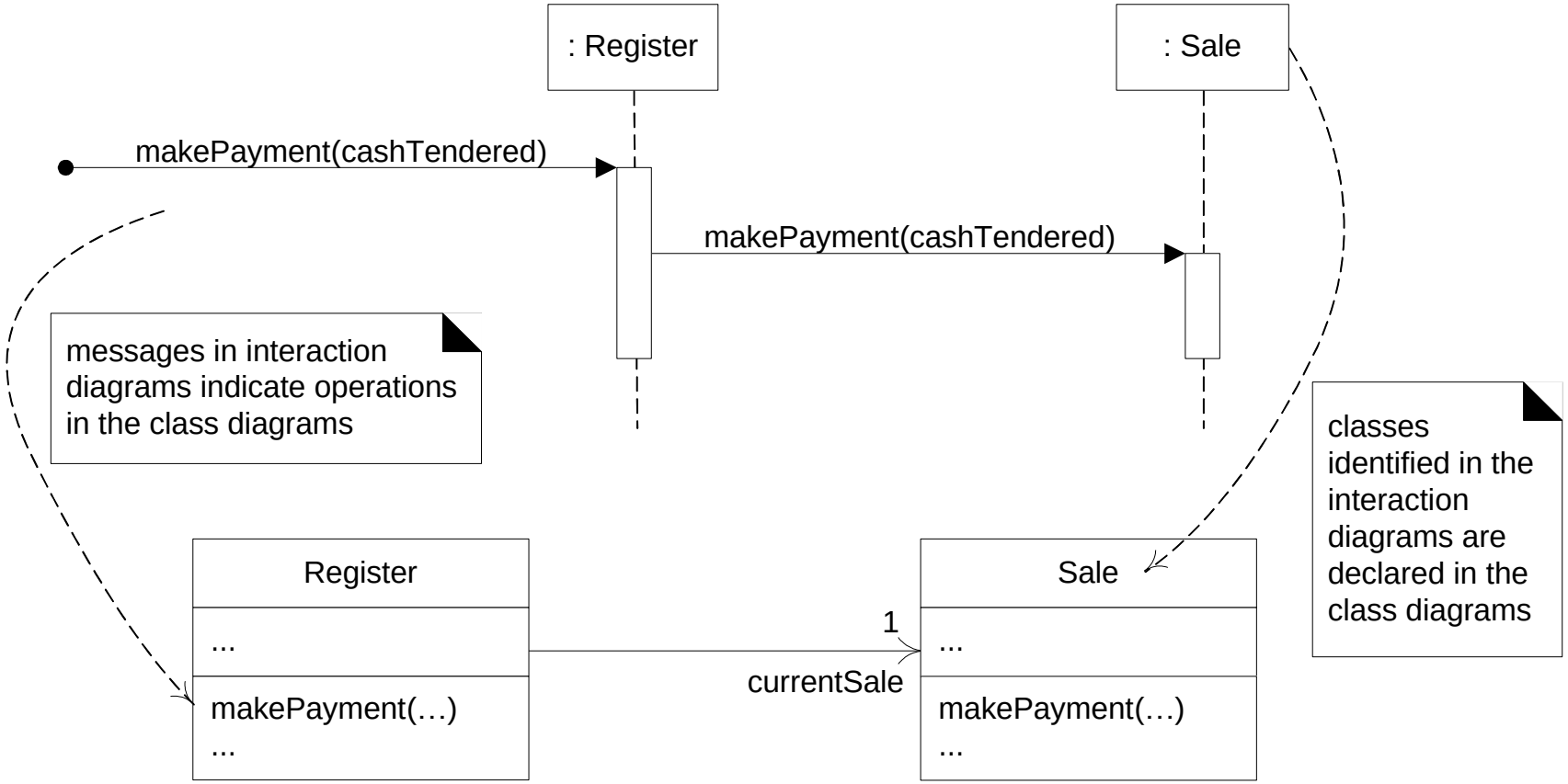
Ref: [Larman]

Connecting the UI and Domain Layers



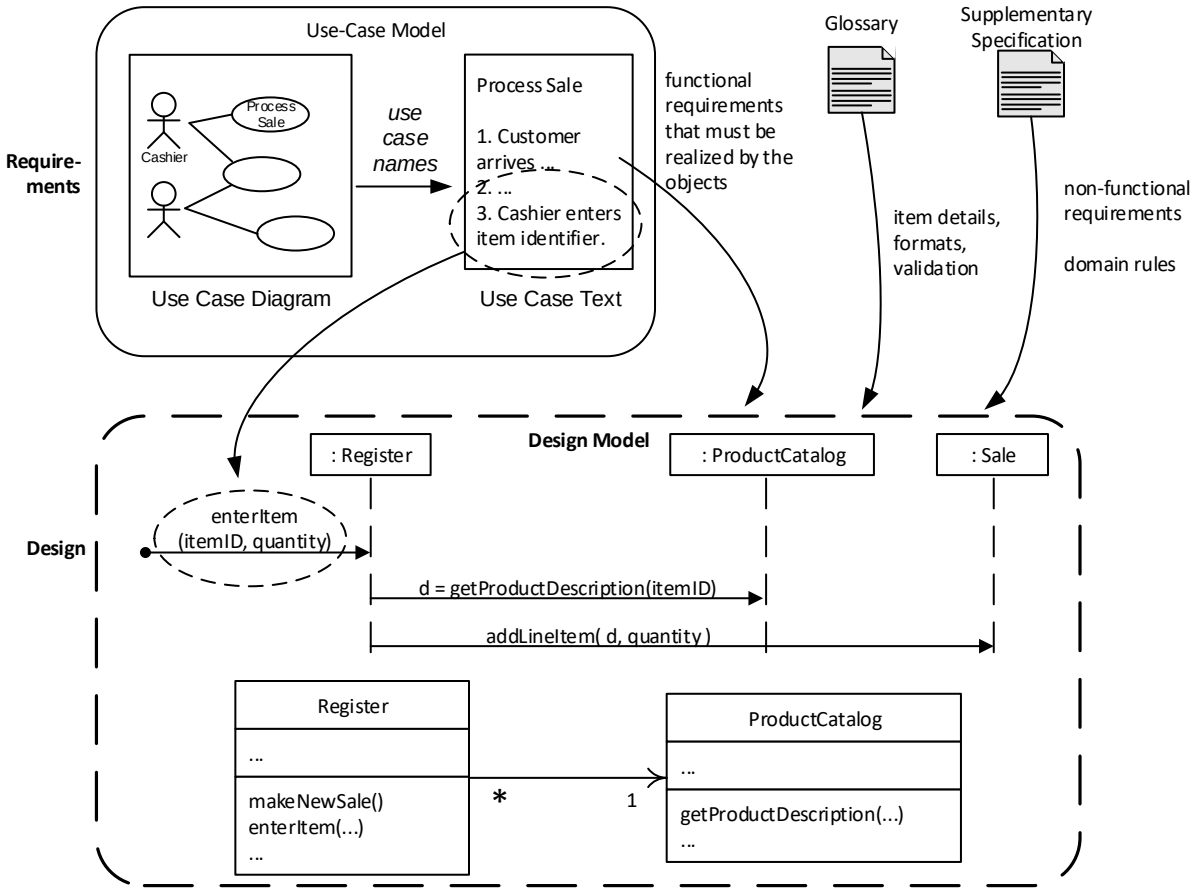
Ref: [Larman]

Relationship Between Interaction and Class Diagrams



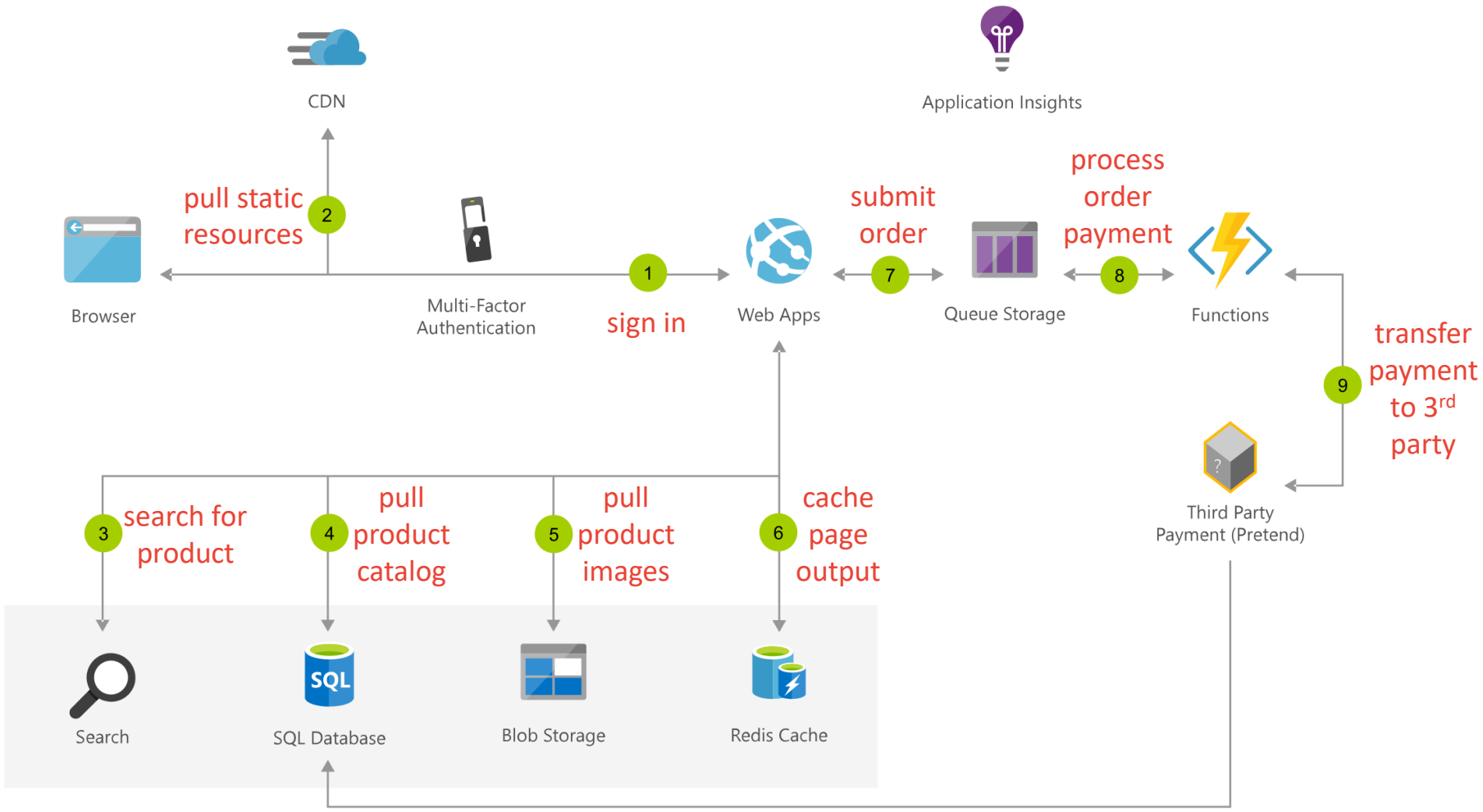
Ref: [Larman]

Artifact Relationships



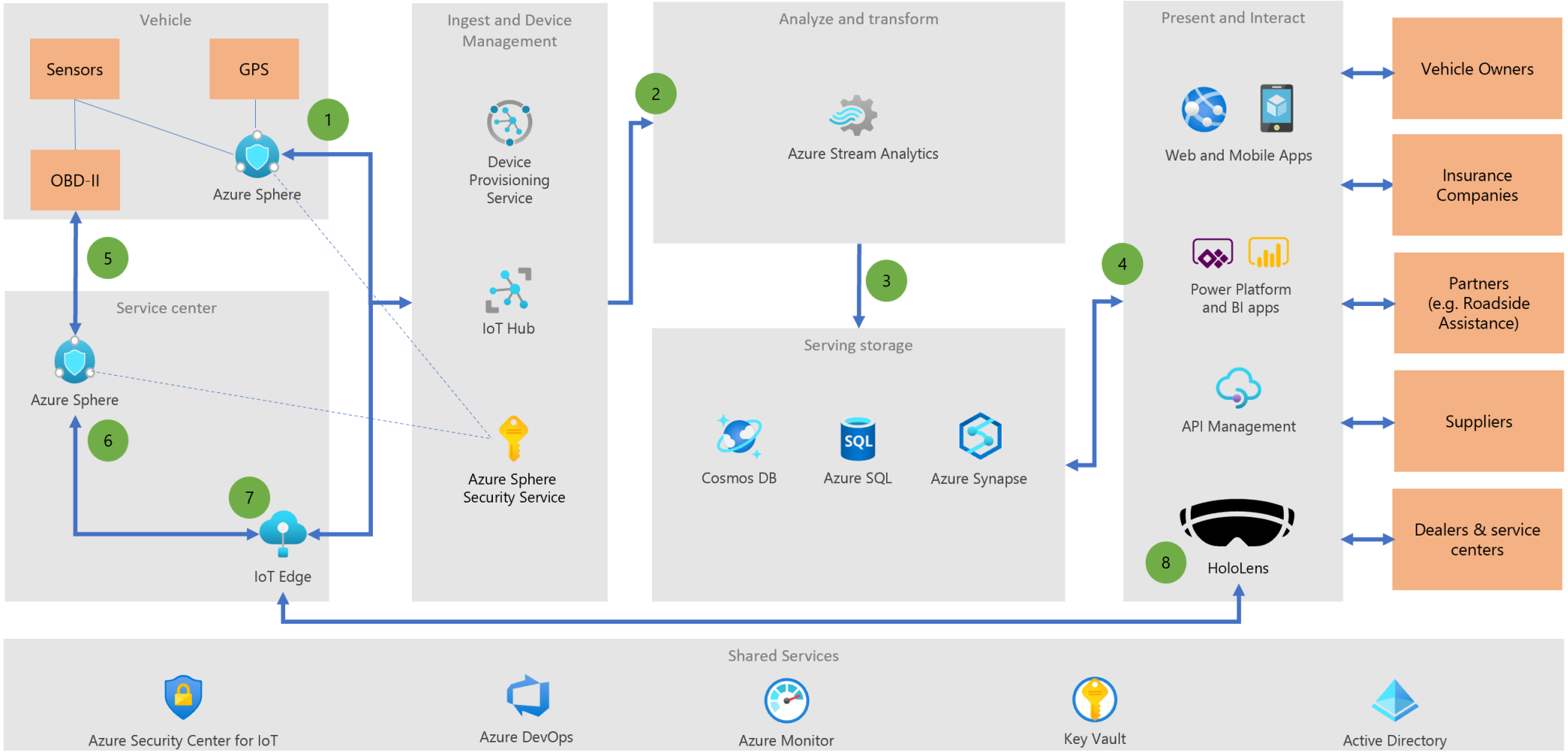
Ref: [Larman]

Sample Scalable e-Commerce Web App Architecture on Azure



Ref: https://docs.microsoft.com/en-us/azure/architecture/solution-ideas/articles/scalable-ecommerce-web-app

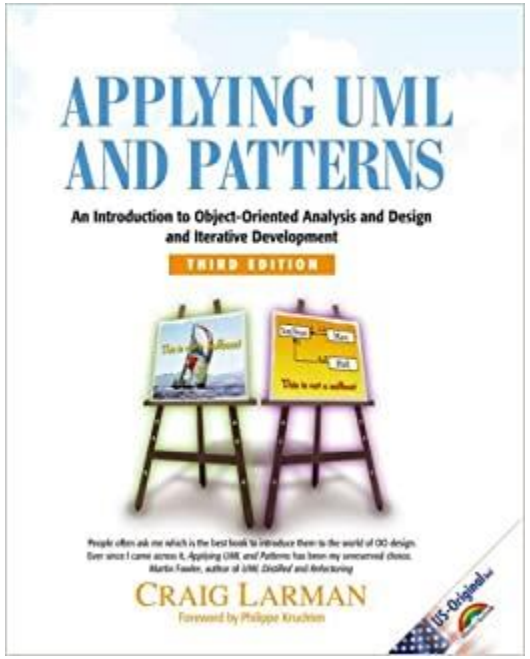
Sample Architecture for processing real-time vehicle data using IoT



Ref: https://docs.microsoft.com/en-us/azure/architecture/example-scenario/data/realtime-analytics-vehicle-iot

More Information

For more detailed information on object-oriented analysis and design



Azure Architecture Center

Guidance for architecting solutions on Azure using established patterns and practices.

https://docs.microsoft.com/en-us/azure/architecture/

AWS Architecture Center

https://aws.amazon.com/architecture/

https://dotnet.microsoft.com/download/e-book/cloud-native-azure/pdf

References

[Bass]	Software Architecture in Practice (3rd Ed.), Len Bass, Paul Clements, Rick Kazman, Addison-Wesley Professional, 2012.
[Fowler]	UML Distilled: A Brief Guide to the Standard Object Modeling Language (3rd Ed.), Martin Fowler, Addison-Wesley Professional, 2003.
[Larman]	Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development (3rd Ed.), Craig Larman, Prentice Hall, 2004.
[Microsoft]	Microsoft Application Architecture Guide (2nd Ed.), 2009.
[Page-Jones 88]	The Practical Guide to Structured Systems Design (2nd Ed.), Meilir Page-Jones, Prentice-Hall, 1988.
[Sommerville]	Software Engineering (9th Ed.), Ian Sommerville, Addison-Wesley, 2010.
[SWEBOK]	Guide to the Software Engineering Body of Knowledge, Version 3.0, P. Bourque and R.E. Fairley, eds., IEEE Computer Society, 2014; www.swebok.org.