UNIVERSITÄT DES SAARLANDES

30 YEARS DFKI Human Centric AI.

Modeling for Architects I: UML

Architectural Thinking for Intelligent Systems

Winter 2019/2020

Marcel Köster, Kai Waelti, Jochen Britz

Prof. Dr. habil. Jana Koehler

References & Special Thanks

- Prof. Sven Apel for his slides & material ☺
- https://www.uml-diagrams.org/ for several images
- https://c4model.com/ for some images

Agenda

- Capturing architectural concepts with UML 2
- Basics & class diagrams (repetition)
- Sequence diagrams
- Package & Component diagrams
- State machines
- Use case diagrams

Views and Diagrams

- We will later in this lecture discuss *views*, which help us to communicate architectural concerns and decisions
- There is no standard for the representation of views, but some modeling standards are helpful and commonly used
- Context view – none !
- Component view – UML package and component diagrams
- Distribution view – UML package and component diagrams
- Runtime view - UML sequence diagrams, UML state machins, BPMN collaboration diagrams
- Functional requirements – UML use case diagrams

Learning Objectives

- Know
  - purpose of UML
  - 14 different diagram types

- Being able to
  - capture architectural concepts with UML 2.5.1
  - communicate architectural concerns and decisions using views
  - explain how UML describes structures, processes and states of software

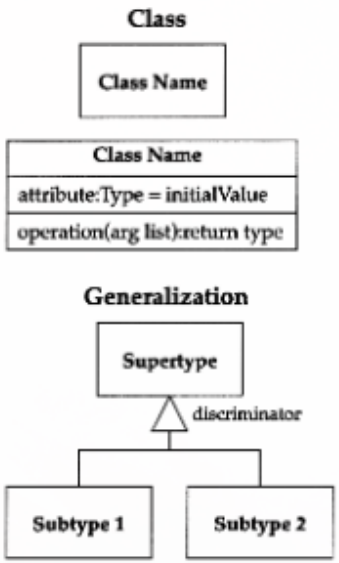
What is UML?

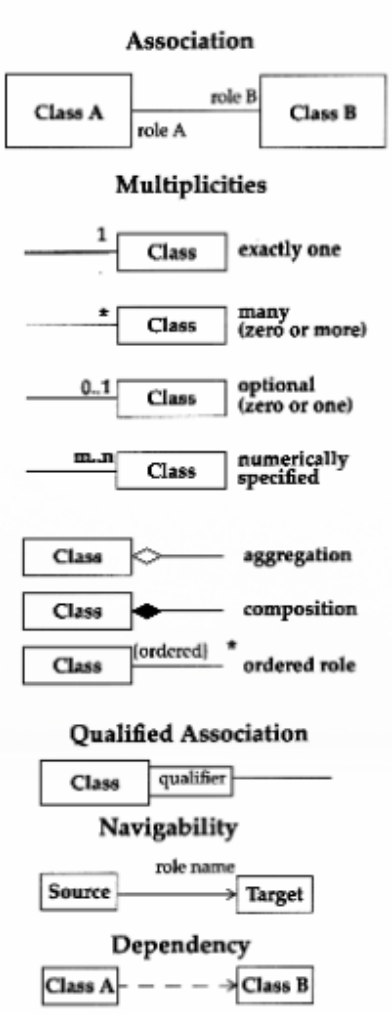
- Uniform notation

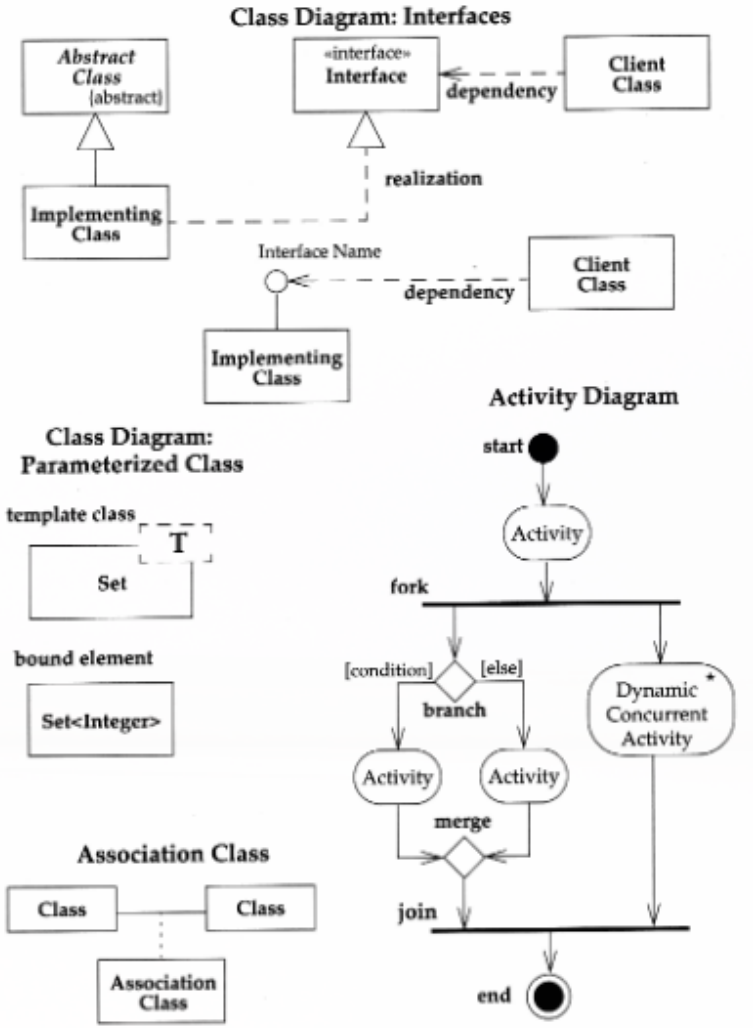
  Booch + OMT + Use Cases (+ state charts)

- UML is *not*
  - A method
  - A process

UML (in a nutshell)







Why UML?

- There are other modeling languages like
  - Systems Modeling Language **SysML**
    - Is less software centric and a lot smaller
  - The Open Group's **ArchiMate**
    - Best for higher-level Enterprise Architectures

- **UML** is the de-facto standard for software modeling
- **UML** fits nicely under the covers
  - Describes the system from various perspectives

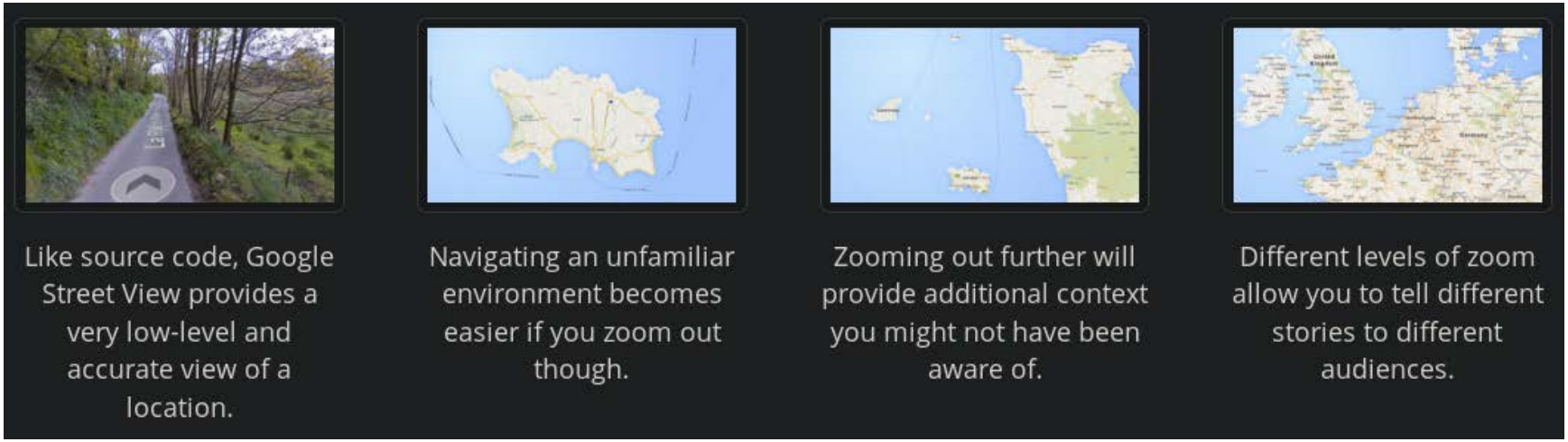
Purpose of UML

- Provides **unified notation** and **semantics** of modeling elements
- Describes **structures** and **processes** of a system
- Offers possibility for **different views** on a system
- Allows people to **understand** and **talk** about the design decisions

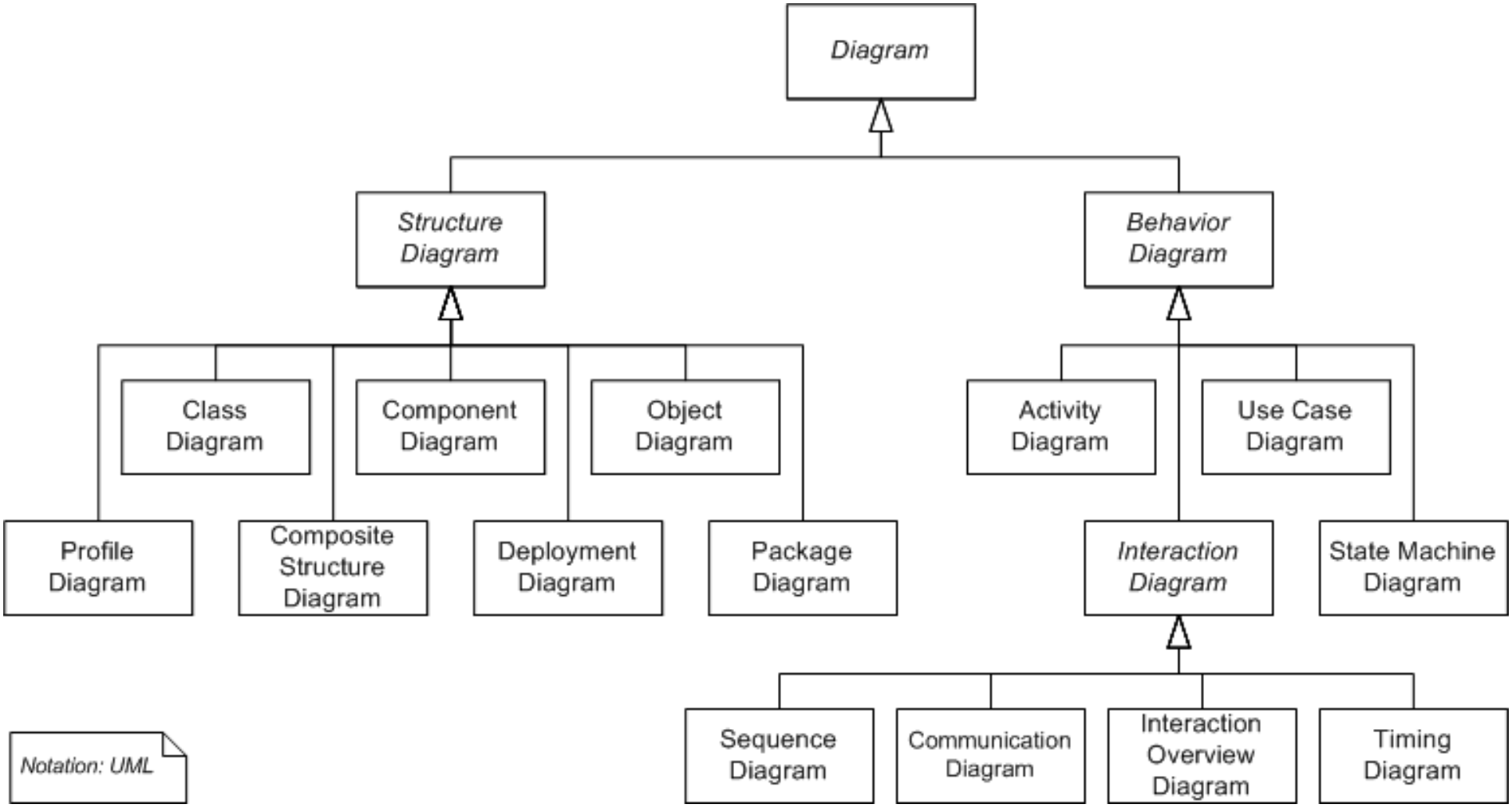
Maps of Your System

- Use **different views** with **different levels of detail**
  - Tell **different stories** to **different types of audiences**

- Helps to **describe architecture** during **up-front design** sessions as well as **retrospectively documenting** an existing code base

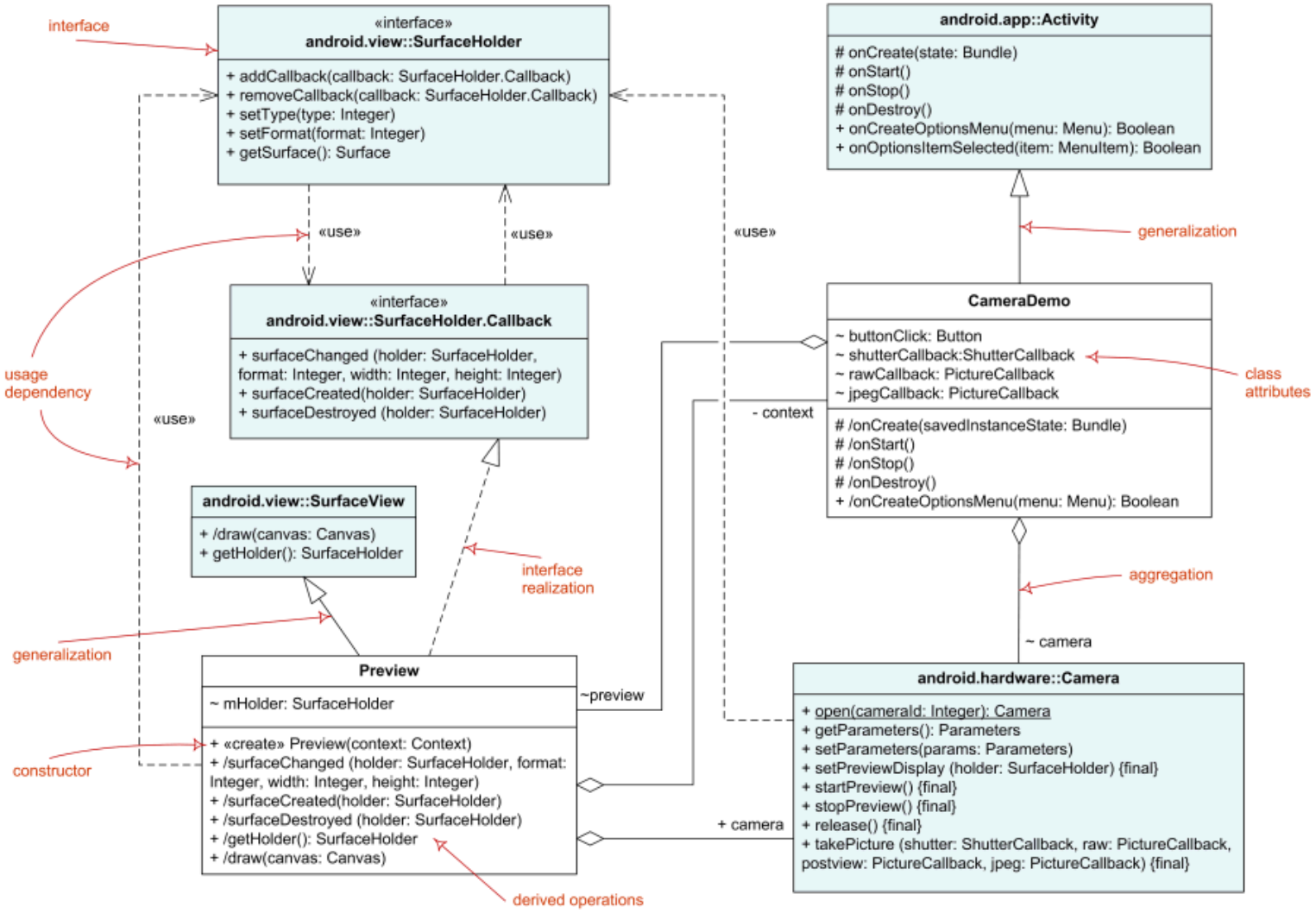


UML 2.5 Hierarchy from Paulo Merson



STRUCTURAL DIAGRAM TYPES

Class Diagram – building blocks of object-oriented systems

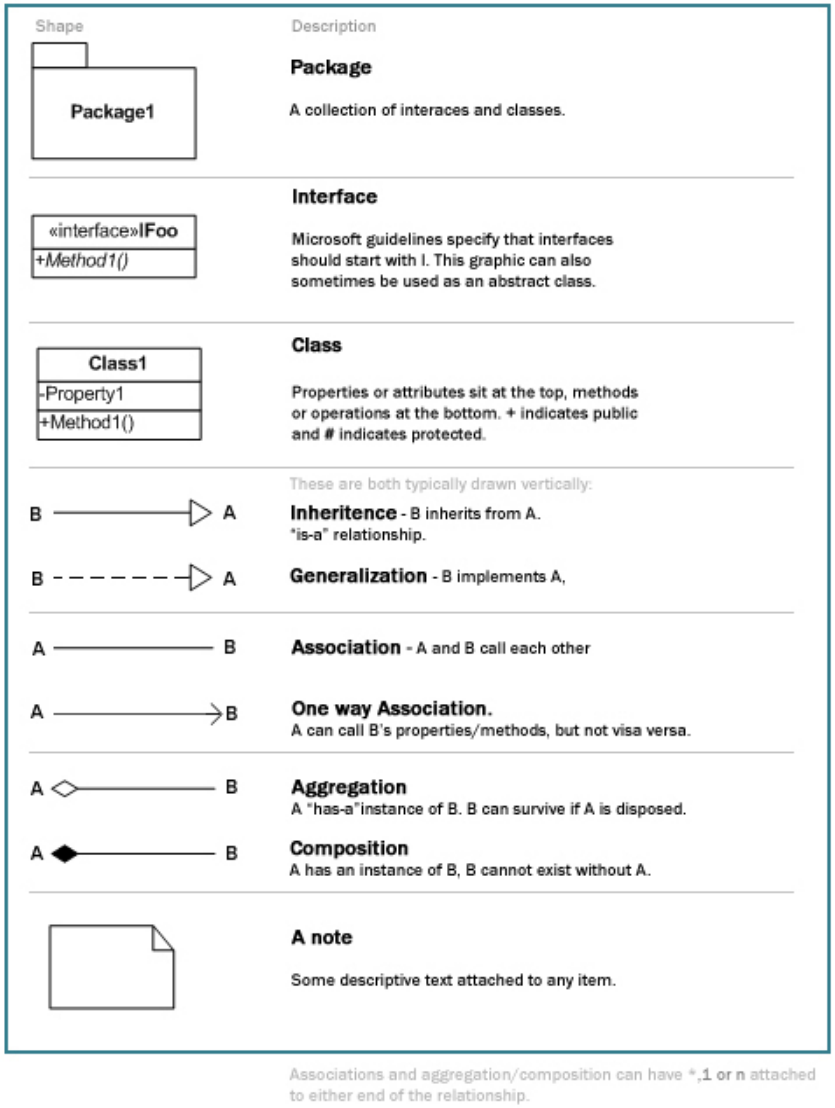


Class Diagram Focus on Behavior

- *Class diagrams* show **generic descriptions** of **possible systems**
- *Object diagrams* show particular **instantiations of systems** and **their behavior**
- *Attributes and operations* are also collectively **called features**
- Risk of turning into **data models**

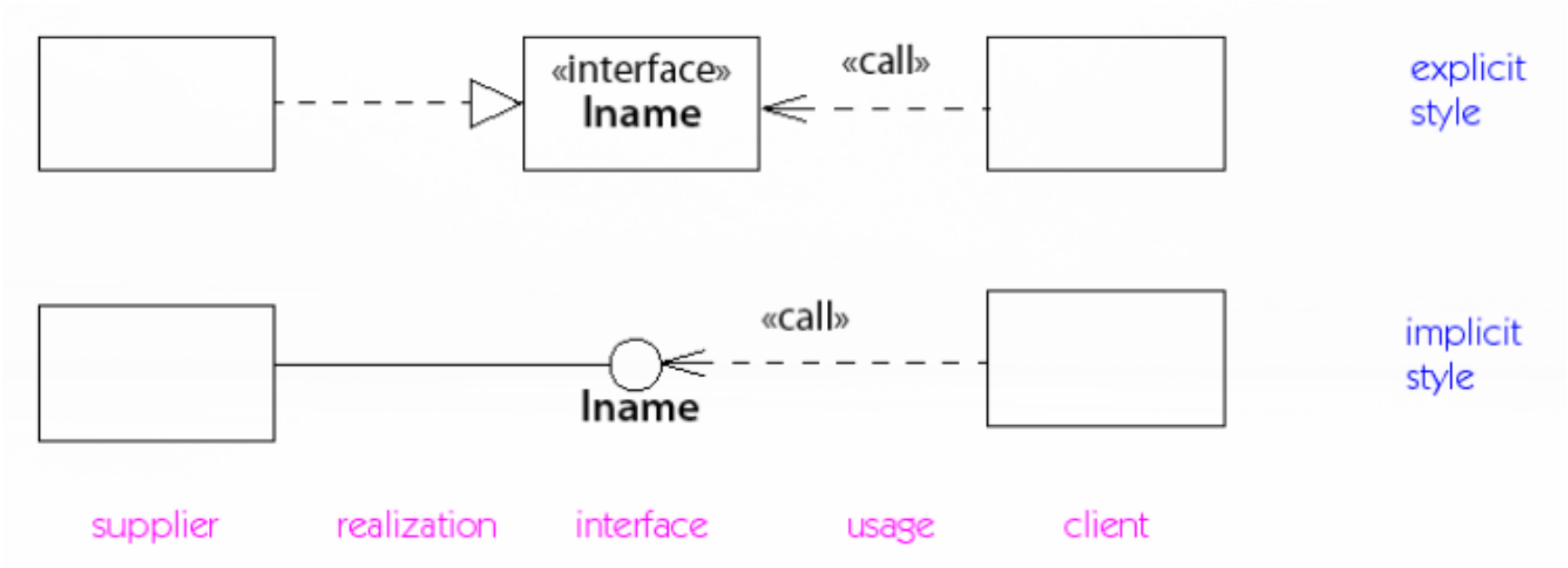
  → be sure to **focus on behavior!**

Class Diagram UML 2.5 Reference



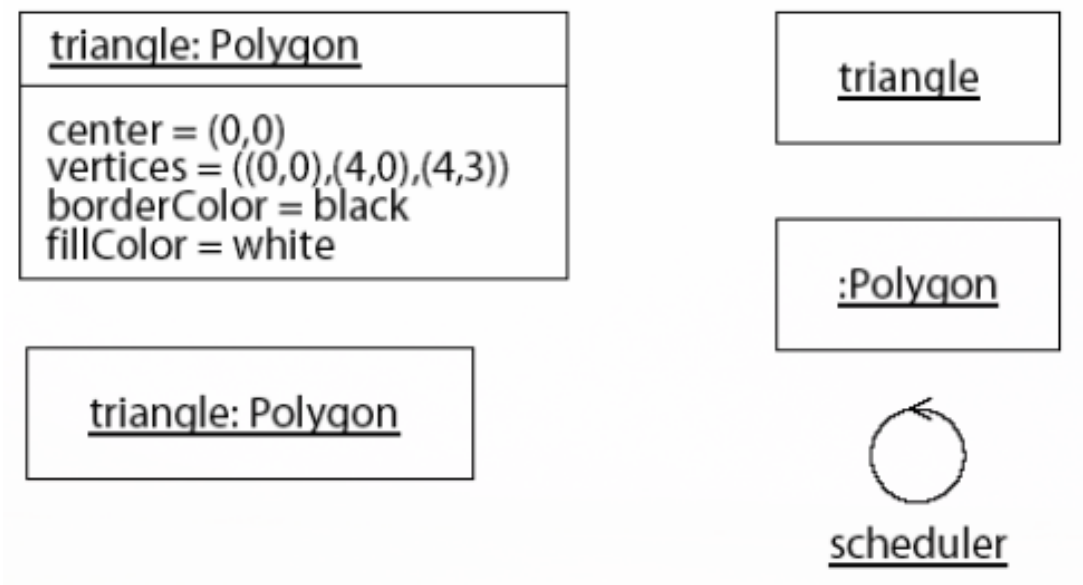
Interfaces

- Equivalent to abstract classes **minus the attributes**
- Represented as classes with **explicit stereotype «interface»** or implicit **lollipop notation**



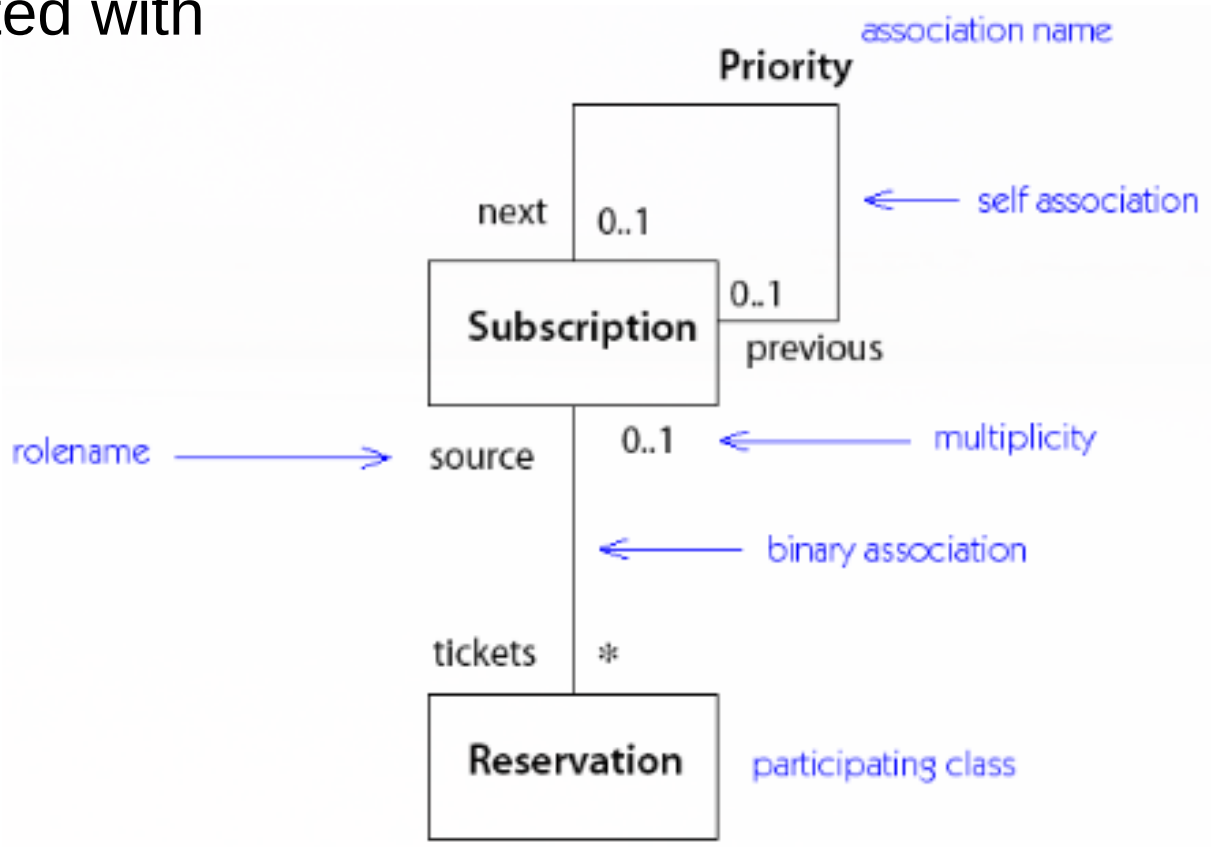
Objects

- Class is a blueprint from which objects are created
  - **Class:** Human
  - **Object:** Man, Woman
- Shown as rectangles with their name and type underlined



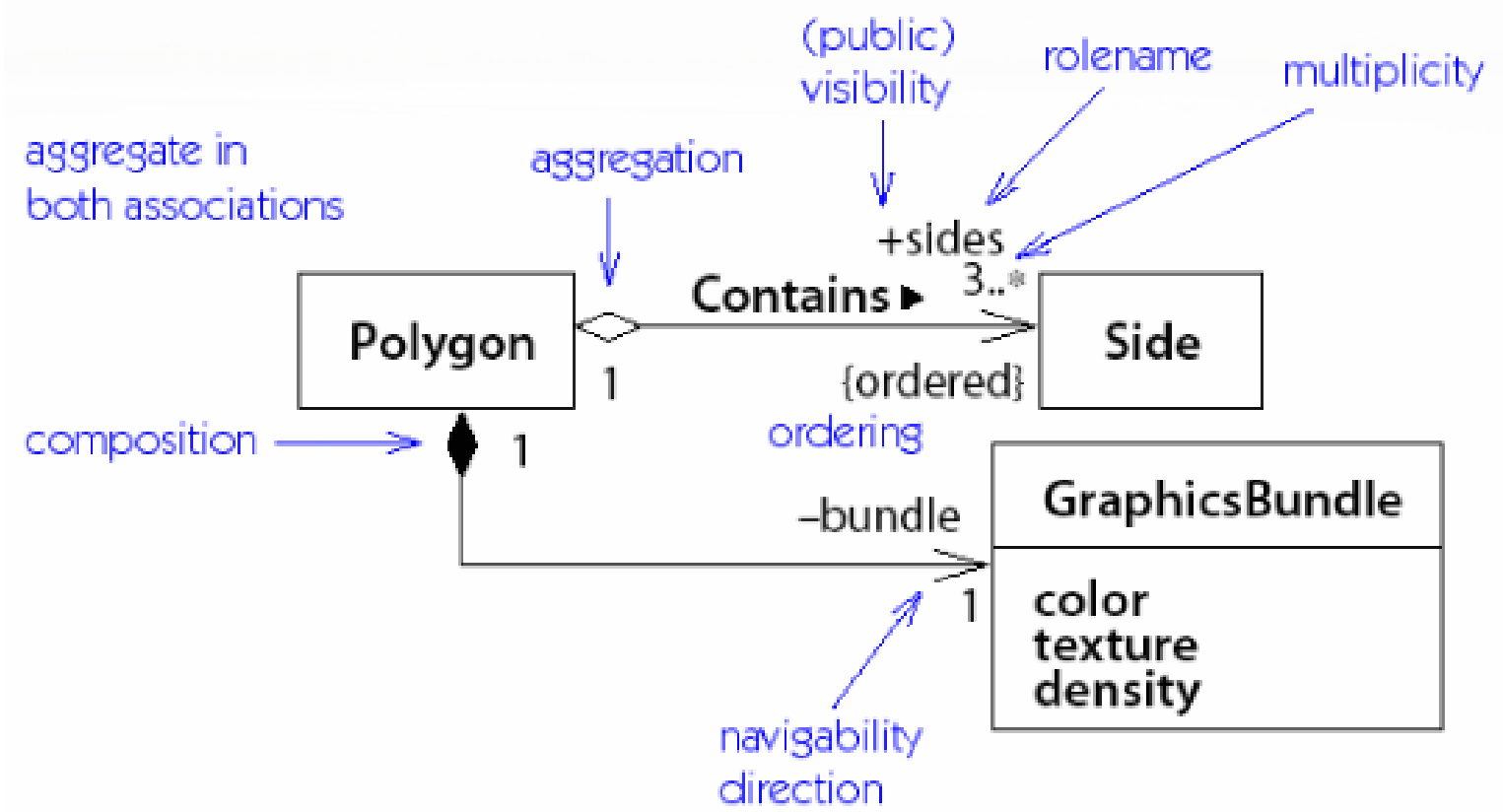
Associations

- Represent structural relationships between objects
- Multiplicity constraints how many entities one may be associated with



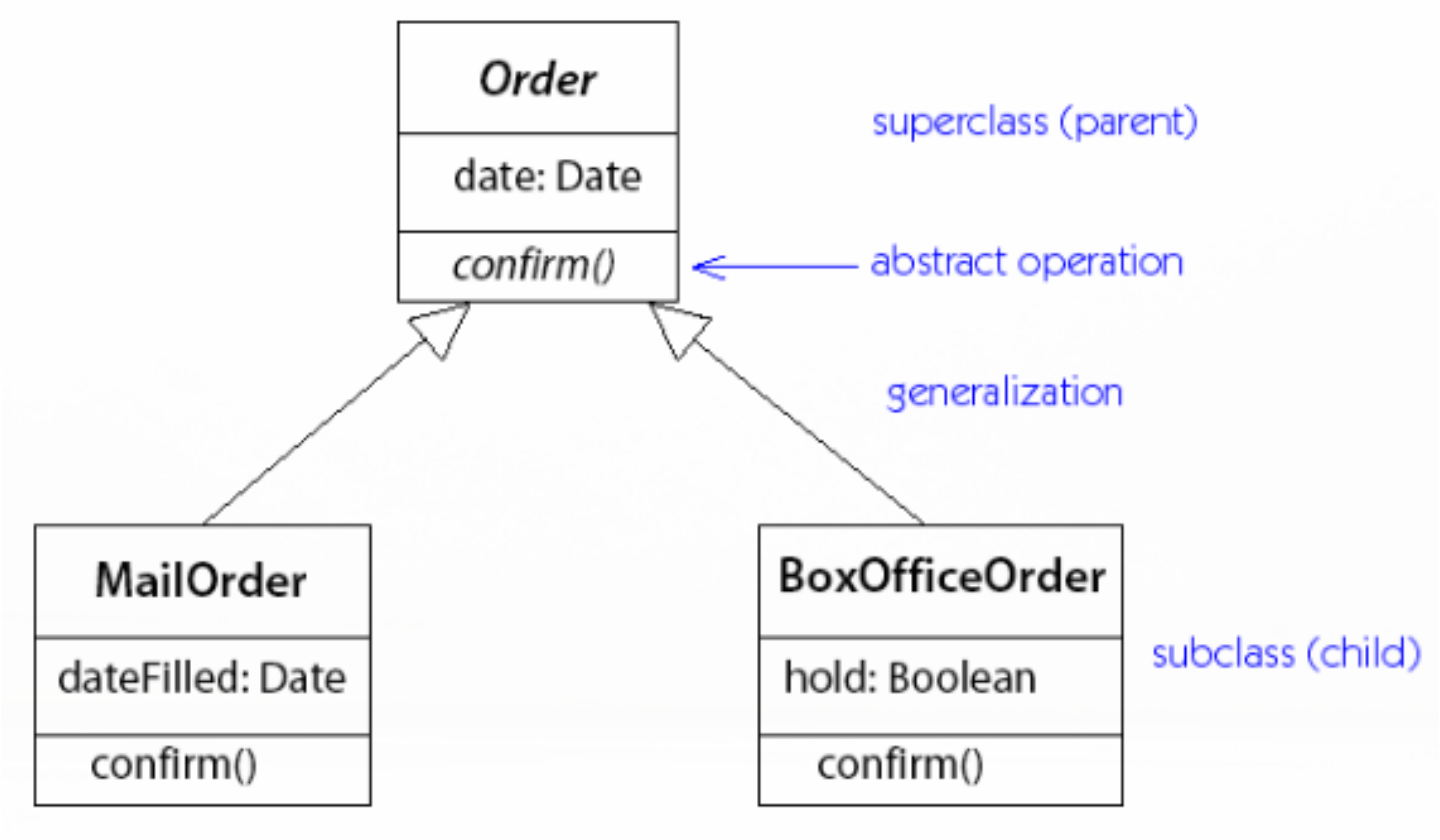
Aggregation vs. Composition

- **Aggregation** → parts may be shared
- **Composition** → one part belongs to one whole



Generalization

- *MailOrder* and *BoxOfficeOrder* specialize their superclass *Order*



Why Inheritance?

- **New software** often builds on **old software** by imitation, refinement, or combination

- Similarly, **classes may be extensions, specializations or combinations**, of existing classes

Generalization Expresses…

- Conceptual hierarchy
  - conceptually related classes can be organized into a **specialization** hierarchy
    - *people, employees, managers*
    - *geometric objects*
- Polymorphism
  - objects of distinct, but related classes may be **uniformly treated** by clients
    - *array of geometric objects*
- Software reuse
  - related classes may **share** interfaces, data structures or behavior
    - *geometric objects*

Component Diagram

- Shows **components**, provided and required **interfaces**, **ports**, and **relationships** between them
- Based on assumptions, that **previously constructed components** could be **reused**
  - or be **replaced** by some other ***equivalent*** **component**
- Artifacts that implement the component are intended to be capable of being deployed independently
  - e.g. for updating an existing system

Components Could Represent…

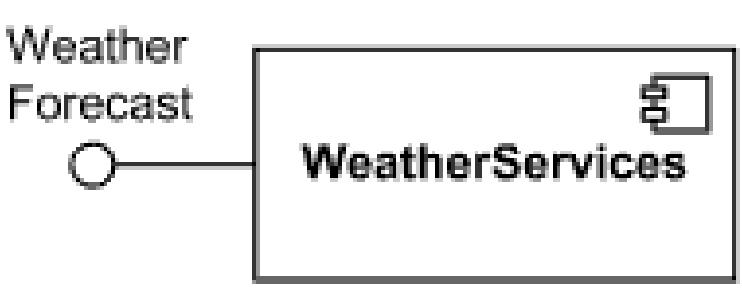
- **Logical components**
  - e.g. business components, process components, etc.
- **Physical components**
  - e.g. EJB components, COM+ and .NET components, WSDL components, etc.

- A **component** is a **replaceable part** of a system that **conforms to** and **provides the realization** of a set of **interfaces**

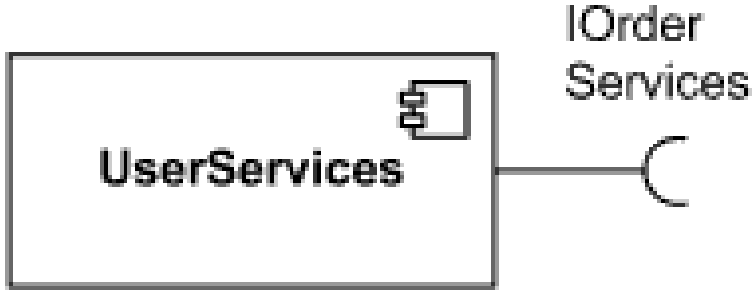
Component Notation

UserService Component

- An **interface** is a **collection of operations** that specify a service that is **provided by or requested from** a component



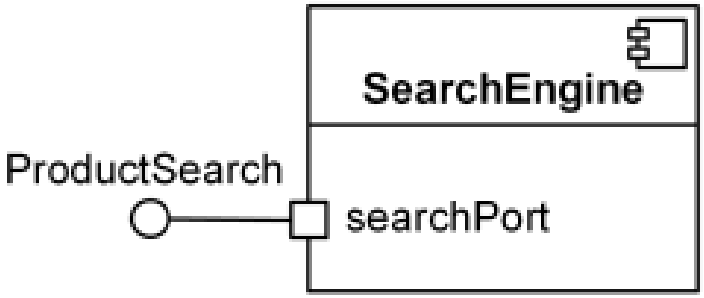
Provided Interface



Required Interface

Components Notation: Ports

- A **port** is a **specific window** into an encapsulated component **accepting messages**
  - **to and from** the component



Simple Port

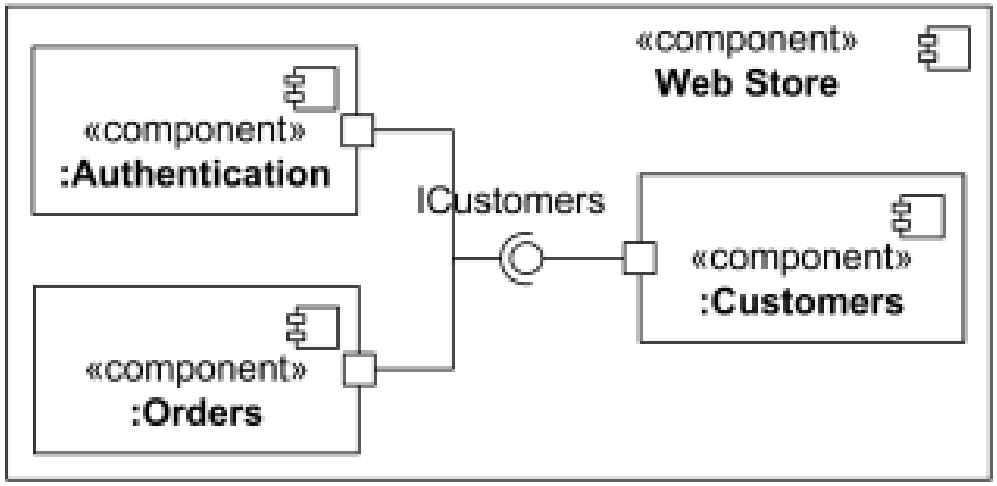
Components: Parts and Connectors

- A **part** is a **specification** of a **role** that **composes part of the implementation** of a component

- A **connector** is a **communication relationship between** two **parts or ports** within the context of a component
  - Connector linking could be either **delegation** or **assembly** connector

Components: Assembly Connectors

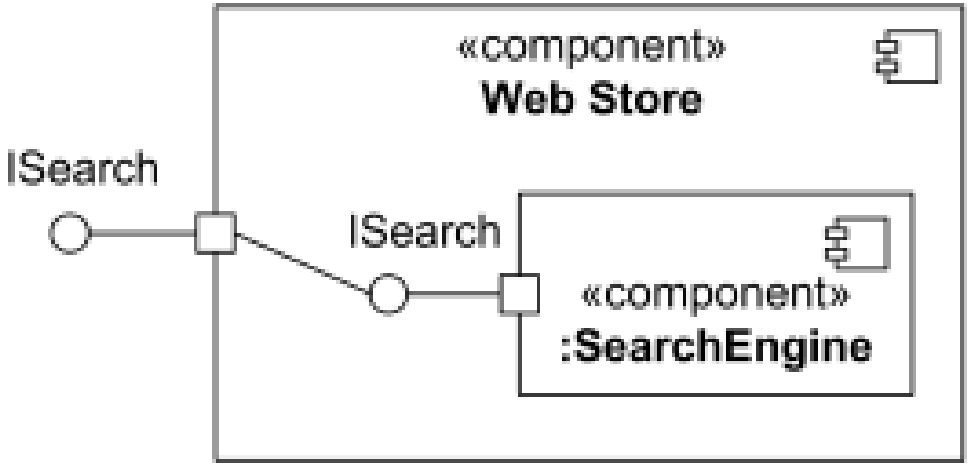
- Connector between **two or more parts or ports**
- Defines that **one or more parts** provide the **services that other parts use**



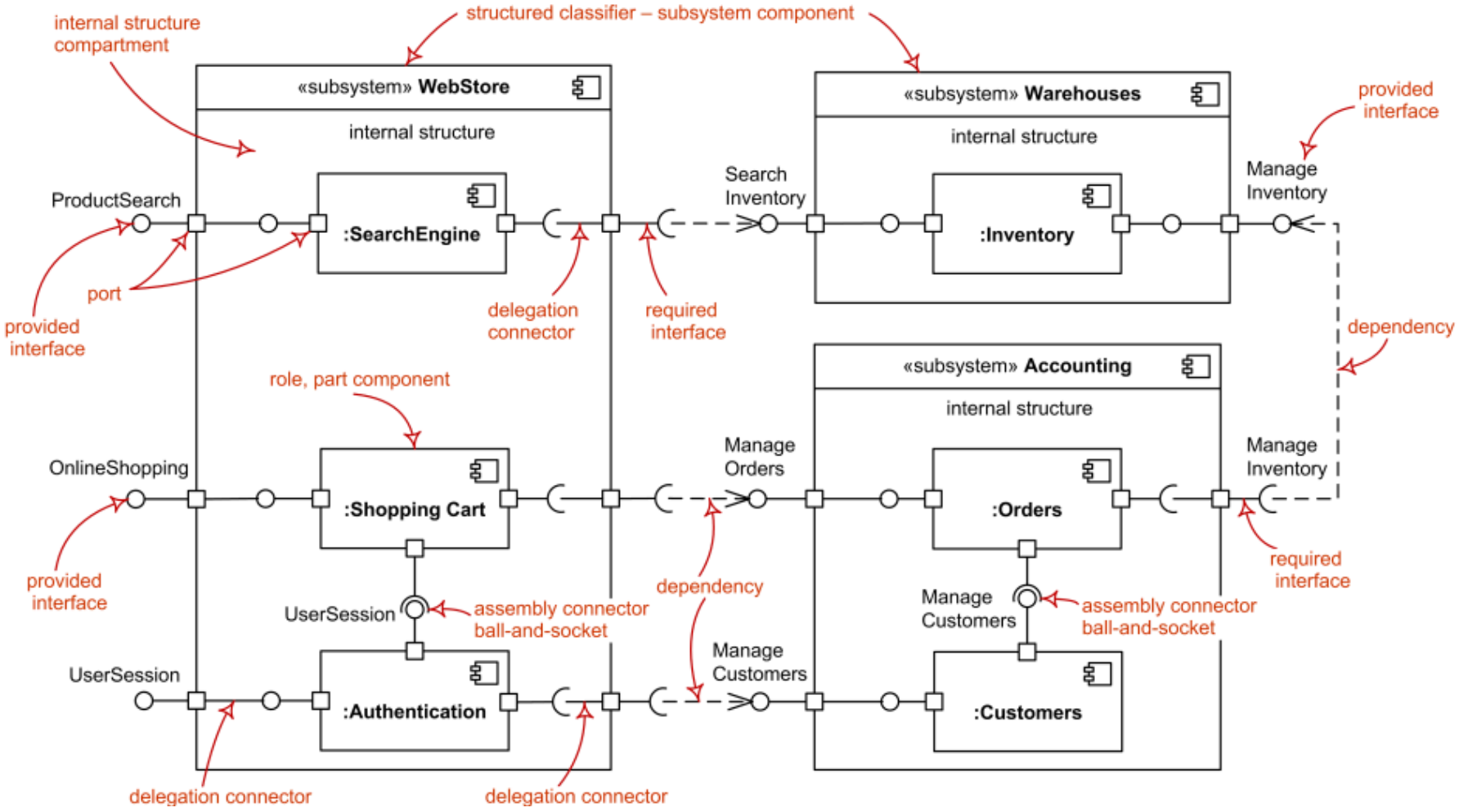
Assembly connector that assembles 3 parts

Components: Delegation Connector

- **Connector** that links the **external contract** of a component to **the realization of that behavior**
- Represents the **forwarding of events**
- Can be used to model **hierarchical decomposition of behavior**
- A **port may delegate** to **a set of ports** on subordinate components



Component Diagram: A Reference

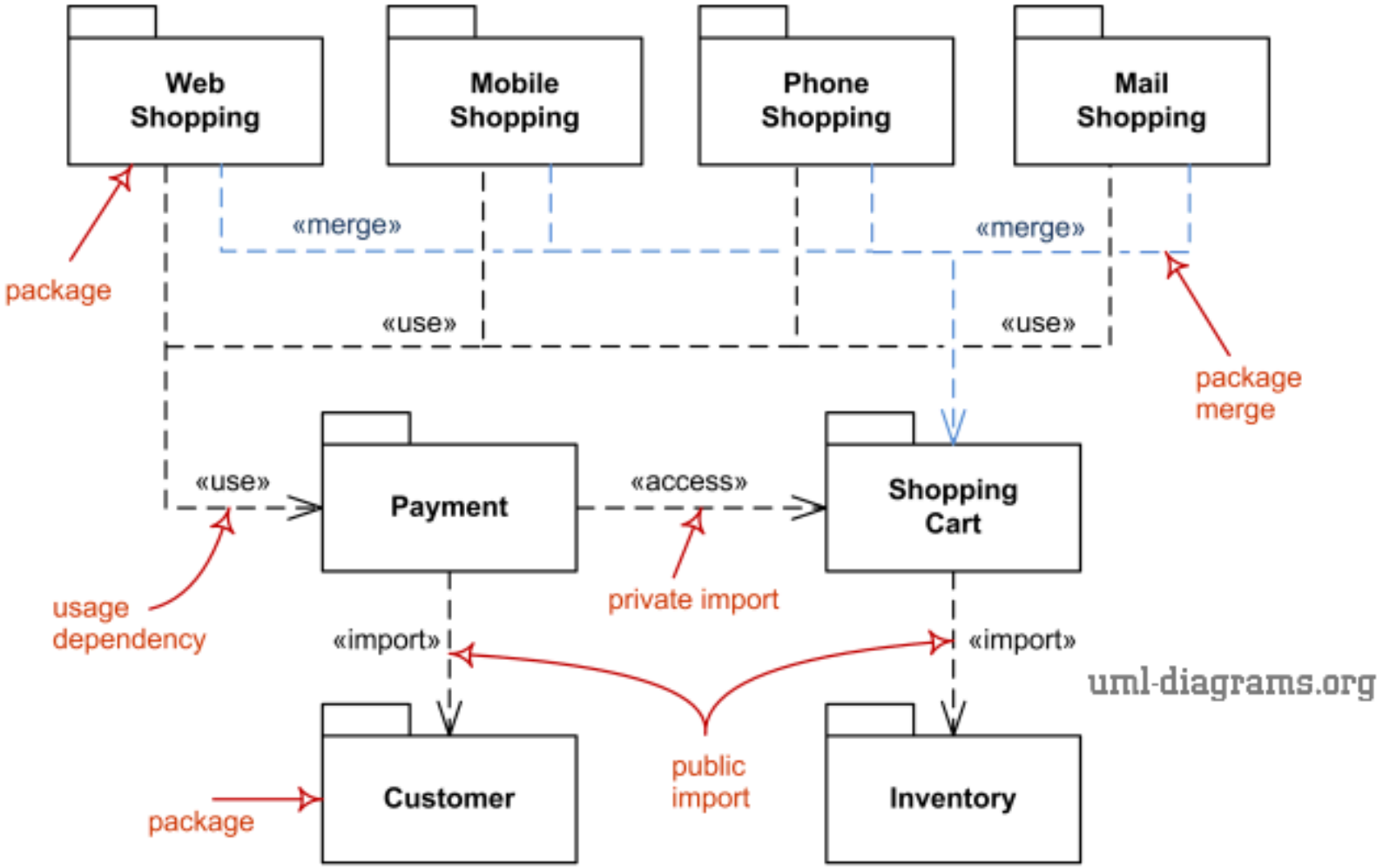


Packages Notation

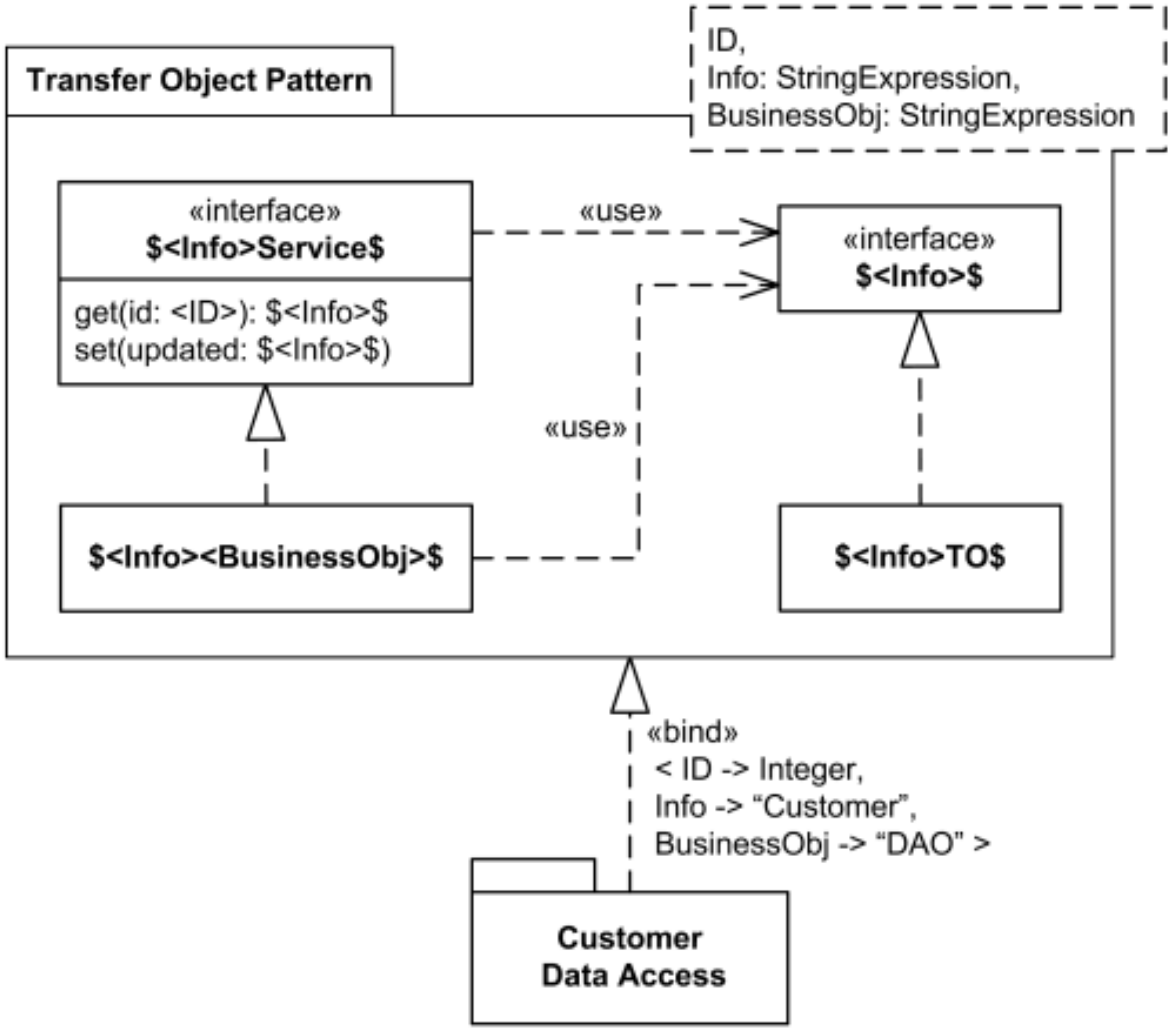
- A **package diagram** shows **structure** of the designed system at level of packages

- **Package** is a **namespace** used to group together elements that are semantically related and might change together
  - May own **packageable elements** like *Type, Classifier, Use Case*, etc.
  - Can be used as a **template** for other packages
    - **Template parameters** can be offered through **packageable elements**
  - Different directed relationships
    - *use, import, merge*

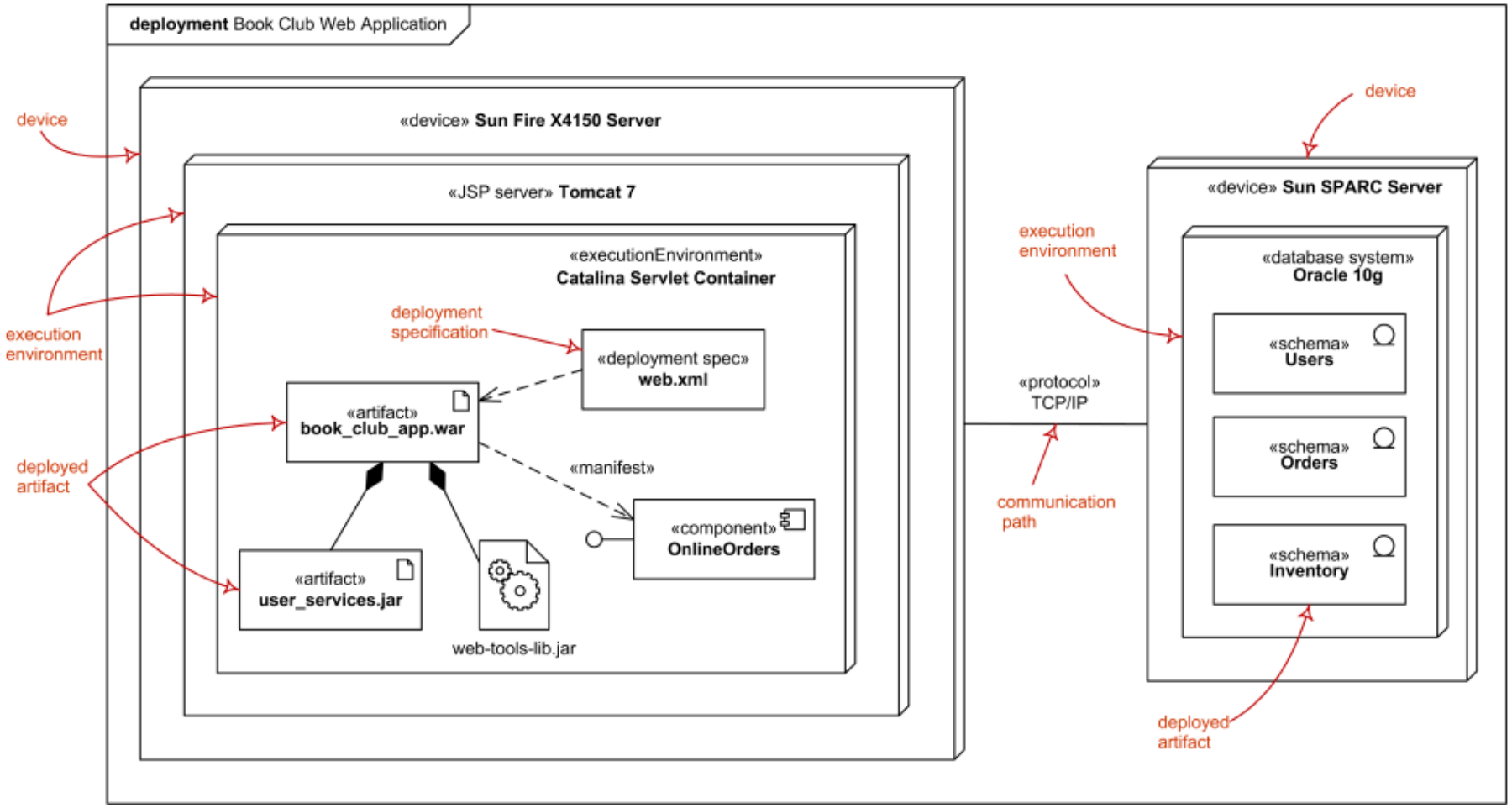
Package Diagram: A Reference



Package Diagram: Design Pattern known as Transfer Obj.

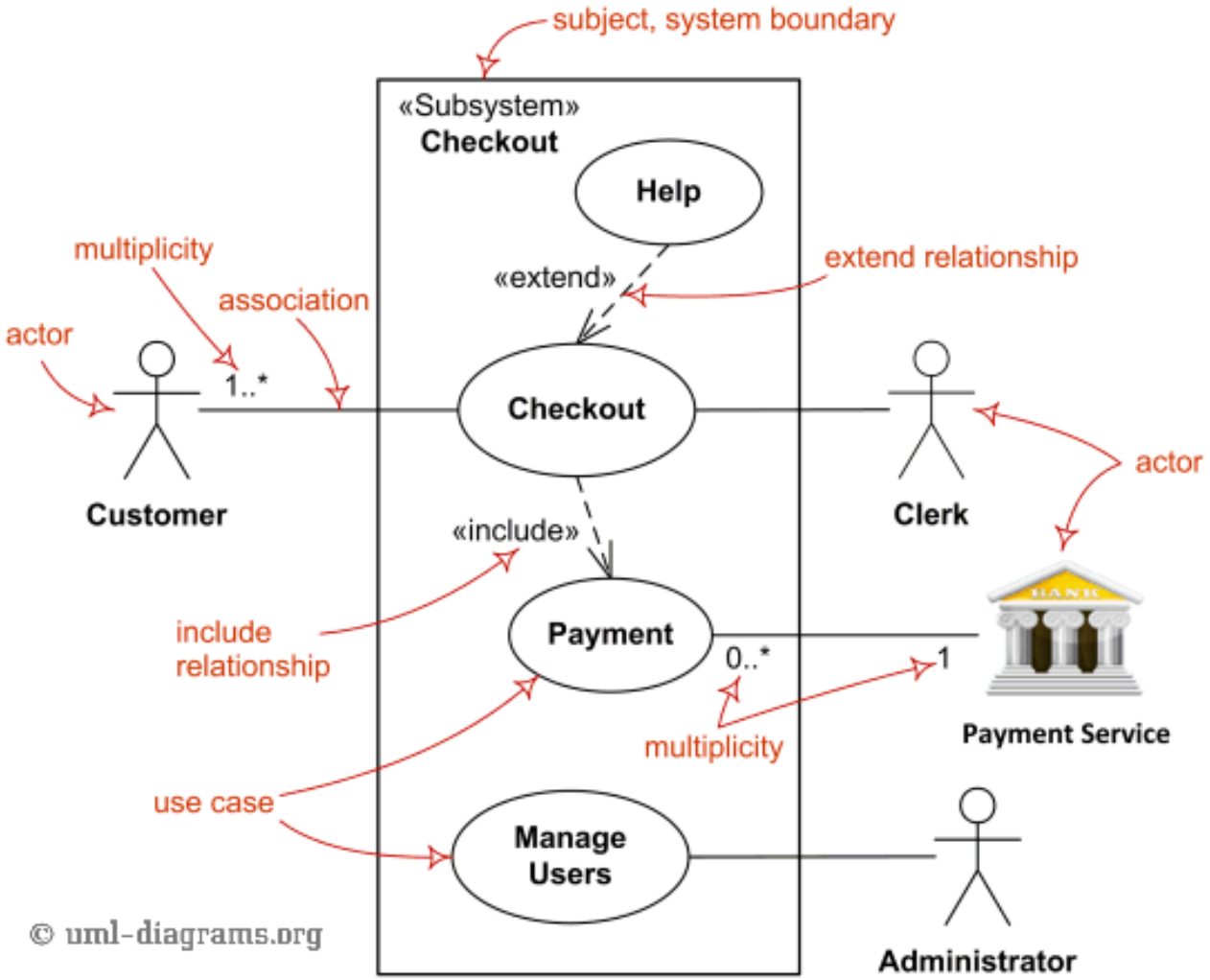


Deployment Diagram



BEHAVIORAL DIAGRAM TYPES

Use Case Diagram

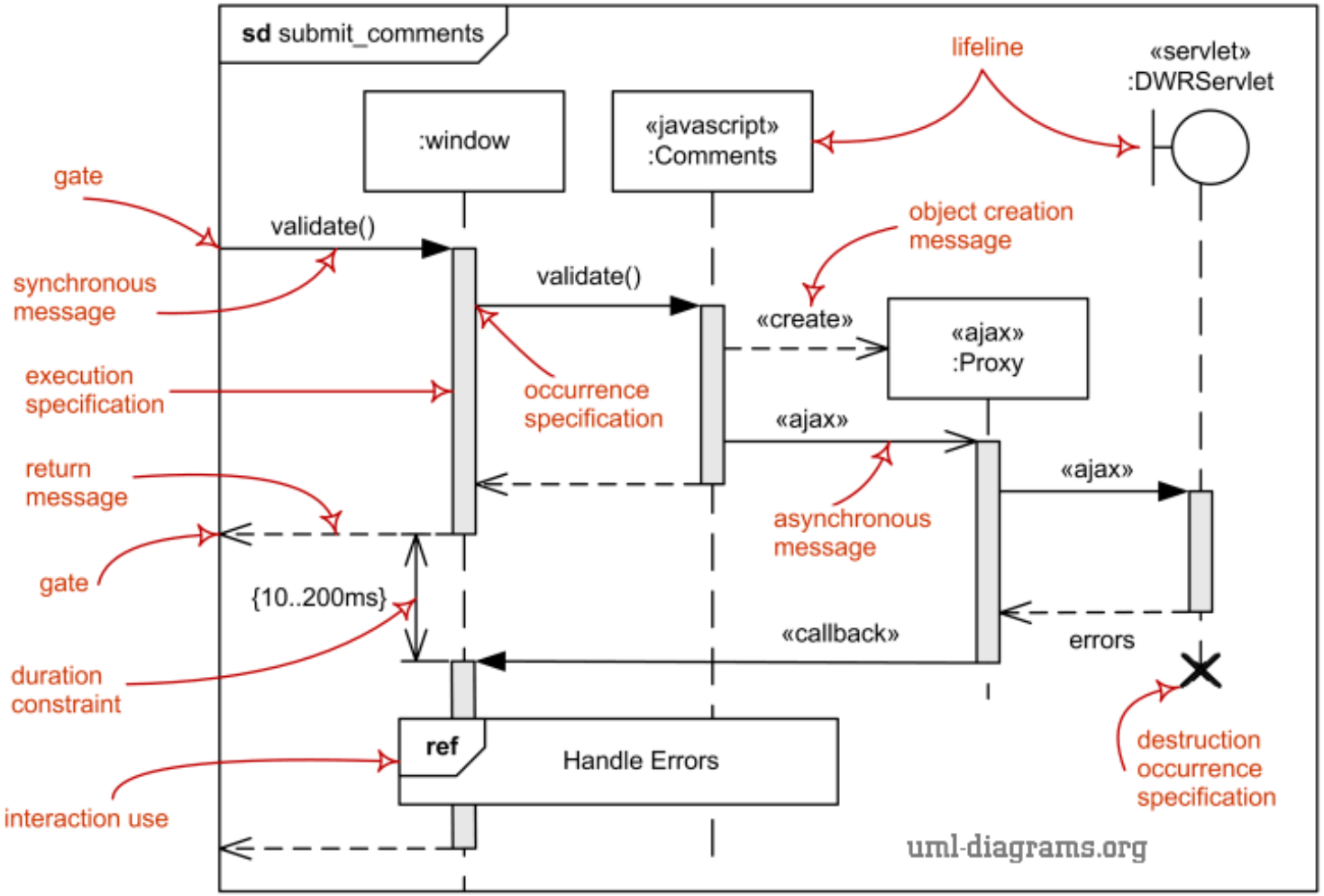


Using Use Case Diagrams

- **Generic description** of an **entire transaction** involving several actors
- **Presents** a set of **use cases** (ellipses) and the **external actors** that **interact with the system**
- **Dependencies** and **associations** between use cases may be indicated

- "A use case is a **snapshot of one aspect** of your system. The sum of all use cases is the **external picture** of your system"

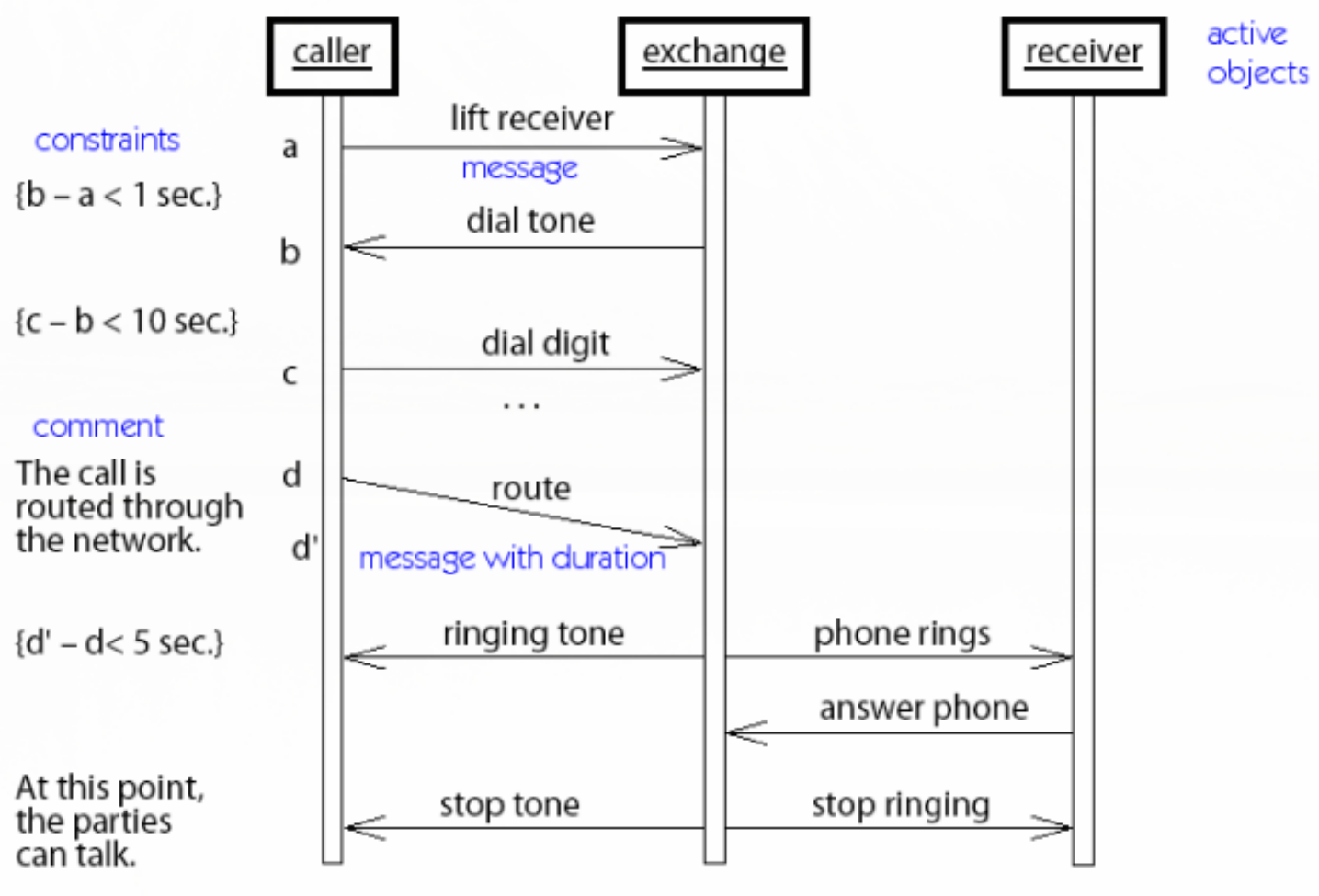
Sequence Diagram



Using Sequence Diagrams

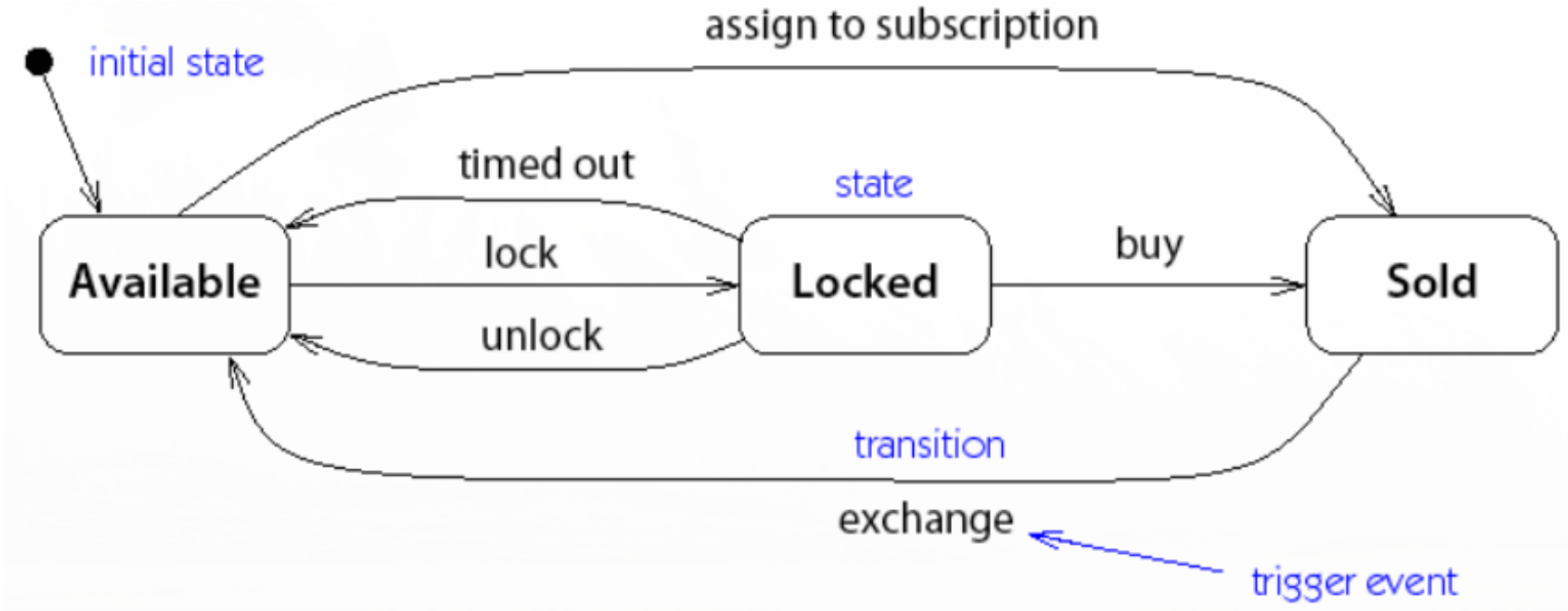
- Depicts a scenario by showing the **interactions** among a set of **objects** in **temporal order**
- **Objects** (not classes!) are shown in **vertical bars**
- **Events** or message dispatches are shown as **horizontal arrows** from the sender to the receiver
- **Avoid returns** in sequence diagrams, unless they add clarity

Asynchrony and Constraints in Sequence Diagrams



State Diagram

- Describes the **temporal evolution** of an object of a given class in **response to interactions** with other objects inside or outside the systems



State Diagram: States and Events

- A state is a **period of time** during which an object is **waiting for an event to occur**
  - may be **nested**
  - depicted as rounded box with (up to) three sections
    - *name*
    - *state variables*
    - *triggered operations*
- An **event** is a one-way asynchronous communication from one object to another
  - **atomic** (non-interruptible)
  - may cause object to make a **transition** between states

Transitions

- A **transition** is an **response to an external event** received by an object in a **given state**
  - May **invoke** an **operation**, and cause the object to **change state**
  - May **send** an **event** to an external object
  - Internal transitions are part of the triggered operations of a state
  - External transitions label arcs between states

Operations and Activities

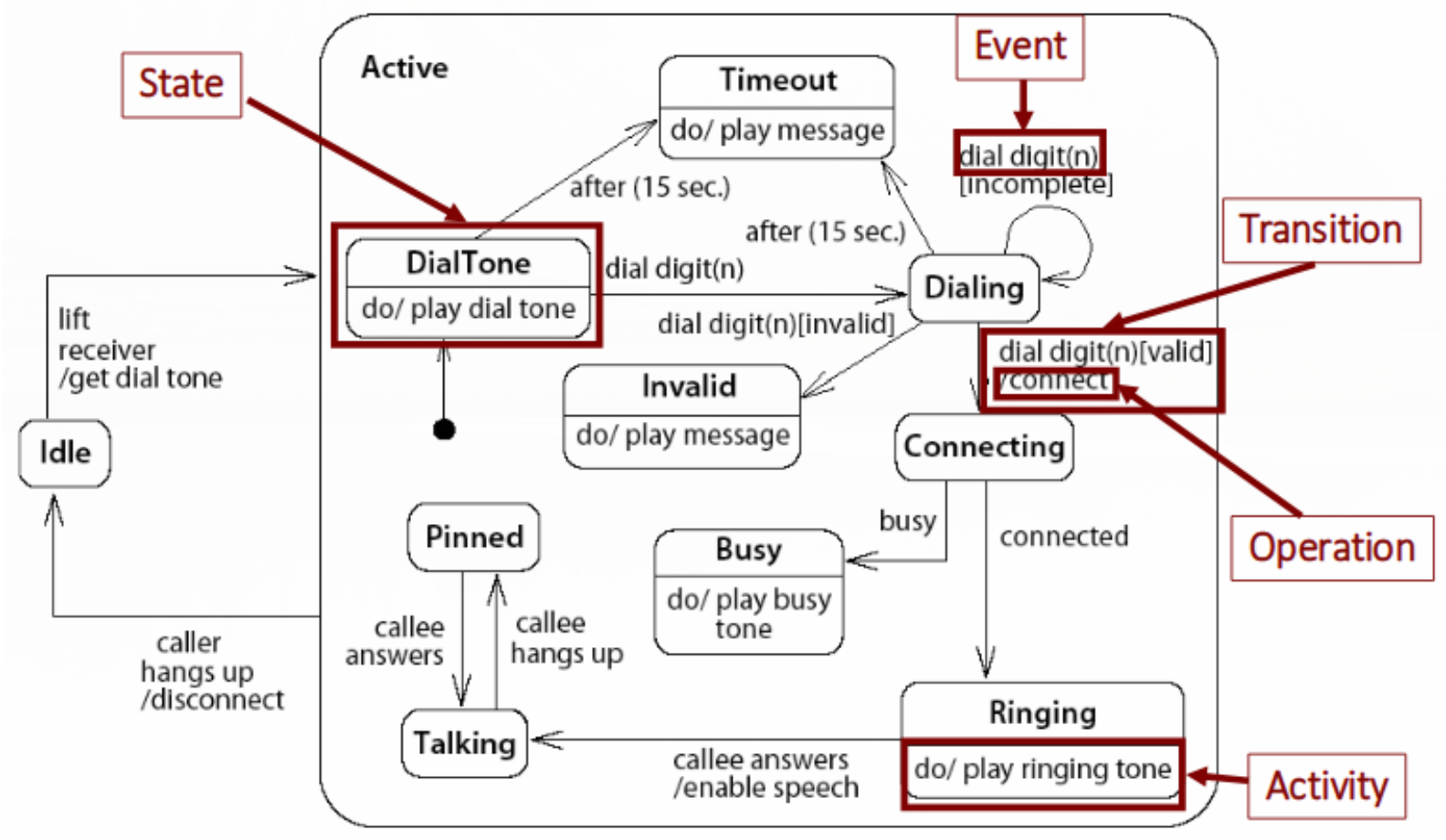
Operation

- **Atomic action** invoked by a transition
  - **Entry and exit operations** can be associated with states

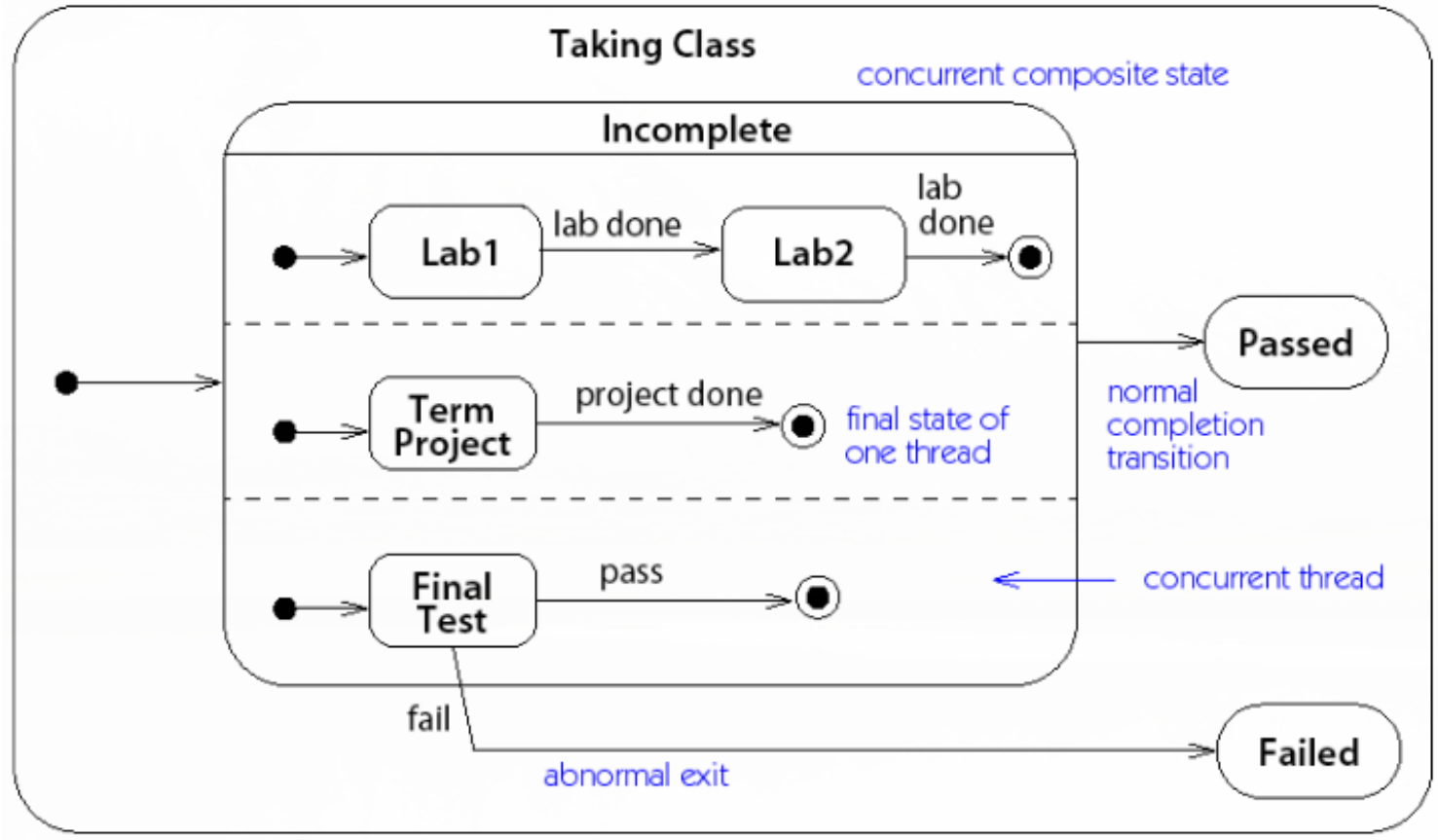
Activity

- **Ongoing operation** that takes place while object is in a given state
  - Modelled as "*internal transitions*" labelled with the pseudo-event **do**

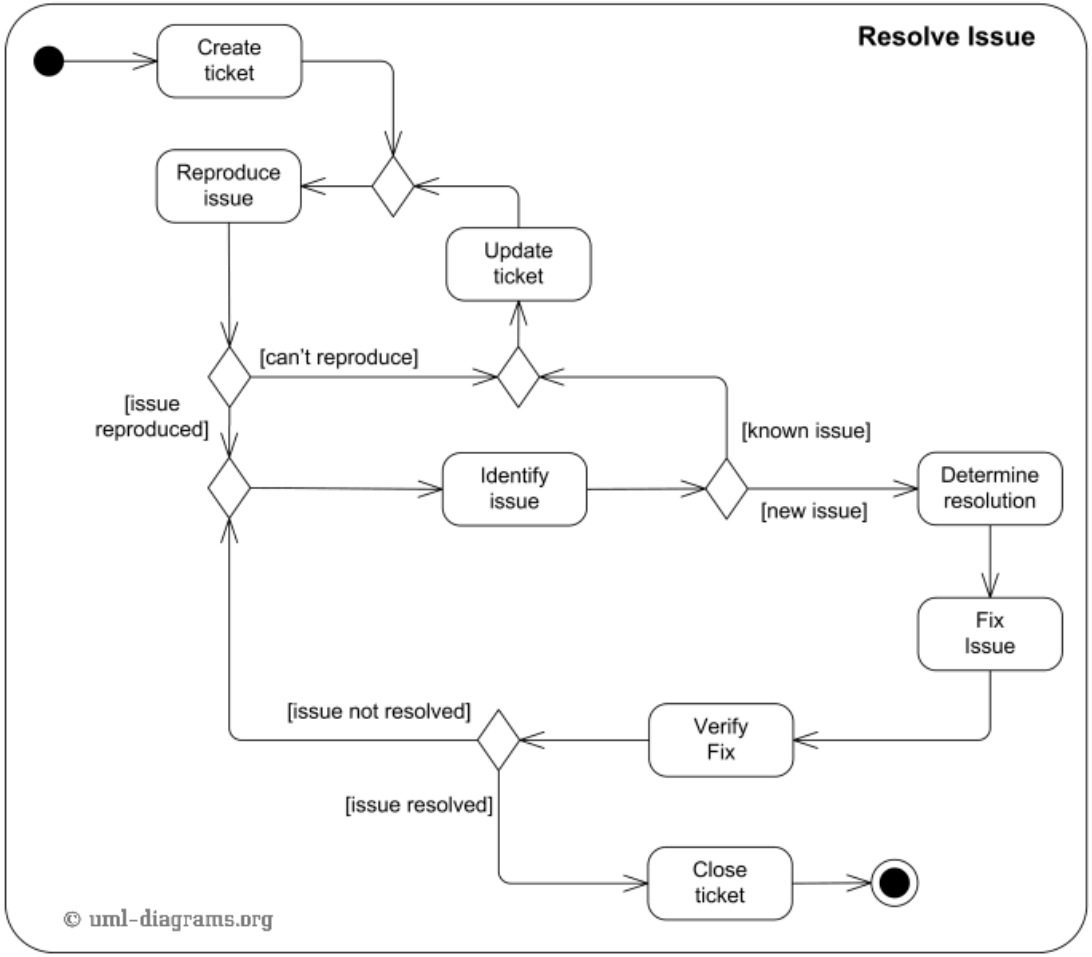
Nested Statechart



Concurrent Substates



Activity Diagram: Resolve an issue in software design



USING UML

Perspectives

- Conceptual
  - Represent domain concepts: *Ignore software issues*
- Specification
  - Focus on visible interfaces and behavior: *Ignore internal implementation*
- Implementation
  - Document implementation choices: *Most common, but least useful perspective(!)*

More Than Creating Blueprints

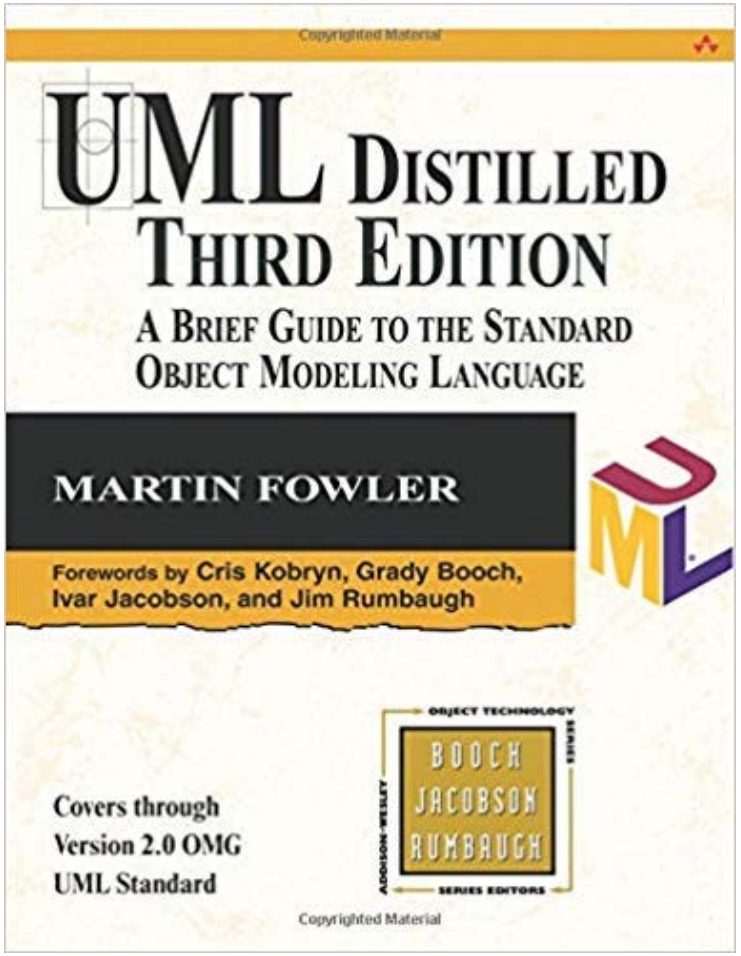
- Create **Use Case** diagrams **to reason** about the **desired behavior** of your system
- Specify the **vocabulary** of your domain using **class diagrams**
- Specify the **sentences** of your domain using **component and package diagrams**
- Use **sequence diagrams**, **statechart diagrams** and **activity diagrams** (or BPMN) to show the way the things in your domain work together to carry out this behavior

OUTLOOK

UML Tools

- StarUML
  - Sophisticated standalone software modeler
- draw.io
  - Online draw app
- UMLet
  - Standalone or Eclipse Plugin
- yEd
  - Standalone graph editor
- astah UML
  - Lightweight UML diagramming tool
- Microsoft Visio
  - Diagramming and vector graphics application

Further Reading



Summary

- UML 2.5 in a nutshell
  - The general purpose of UML
  - Several diagram types for different tasks
  - The different notations depending on the diagram
  - The semantics of these diagrams

- Beeing able to use UML to model
  - Classes, Packages, States, «Control Flow» , etc.

Some Working Questions

1. What was the motivation behind UML?
2. Which UML diagrams exist and what are they used for?
3. Can diagram type X be used to model thing Y in a domain?
4. How can you use diagram X to model a problem description Y (See assignment ☺)