CS189A Capstone
Fall 2023

Lecture 5: UML

Announcements

Attendance: It is crucial for teams to attend both the lectures and the discussion sections and check in with the instructor and the TAs. This is a requirement for the course. You can use the lecture and discussion time to have team meetings and work on the project while the instructor and the TAs are meeting with other teams.

Use of code generation tools: You can use code generation tools if it is helpful for your project. You have to disclose it. You are still responsible in testing the system and the functionality is implemented correctly.

You can NOT use AI tools for generating the documents (such as PRD) since we want you to practice it in the course.

PRD: Product Requirements Document

- The official statement of what is required of the system developers
- Includes a specification of both user and system requirements

- Defines WHAT the system should do, not HOW it should do it
  - Design comes later

- Agile and extreme SWE processes express requirements as
  - Use cases – how a system will act
  - Or as scenarios called user stories (describe result/benefit of it)

  - **Both** document how the system responds from an *external* perspective (when viewed from the **outside**) – like a black box…

So - we are only interested in describing externally visible behavior

PRDv1: Your **Living** Requirements Document: A Shared Google Doc (due tomorrow)

- Authors, Team, Project Title
- Intro – including problem, innovation, science, core technical advance (2-3 pages)
  - Define project specifics, team goals/objectives, background, and assumptions
- System architecture overview
  - High level diagram (1 page)
  - User interaction and design (1 page)
- Requirements (functional and non-functional)
  - User stories or use cases (links): 10 for PRDv1 prioritized
  - Prototyping code, tests, metrics (5+ user stories): github commits/issues
- System models: contexts, sequences, behavioral/UML, state
- Appendices
  - Technologies employed

On-going Process

- Evolving (aka “living”) requirements document
  - Identify/learn (and teach each other) the technologies required
  - Write user stories in particular; update the requirements as you go:
    - Prioritize stories and mark **mandatory, important, or desirable**
    - Assign time estimates to stories; improve your estimation ability over time
    - Specify **acceptance test** for each story – should be in code
- Concurrently as part of Sprint
  - Break down stories into tasks (begin design/prototyping process)
    - Prioritize tasks
    - Assign timings to tasks
    - Specify what (code) test(s) are to be used as evidence of task completion/acceptance
    - Each member/developer chooses task, implements, and tests task
    - Another member does code review/test and accepts the pull request
      - Test is the one specified above (Acceptance)
    - When a Story is complete, some member performs story test/acceptance

PRDv2: Your **Living** Requirements Document: A Shared Google Doc (due in 2 weeks)

- Authors, Team, Project Title
- Intro: problem, innovation, science, core technical advance
  - Define project specifics, team goals/objectives, background, and assumptions
- System architecture overview
  - High level diagram (1 page)
  - User interaction and design (1 page)
- Requirements (functional and non-functional)
  - User stories or use cases (links): 20+ for PRDv2 prioritized
  - Prototyping code, tests, metrics (10+ user stories): github commits/issues
- System models (1+ pages)
  - Contexts, interactions, structural, behavioral (UML)
  - Use cases, sequencing, event response, system state, classes/objects
- Appendices - Technologies employed

UML (Unified Modeling Language)

- Combines several visual specification techniques
  - use case diagrams, component diagrams, package diagrams, deployment diagrams, class diagrams, sequence diagrams, collaboration diagrams, state diagrams, activity diagrams
- Based on object oriented principles and concepts
  - encapsulation, abstraction
  - classes, objects
- Semi-formal
  - Precise syntax but no formal semantics
  - There are efforts in formalizing UML semantics
- There are tools which support UML
  - Can be used for developing UML models and analyzing them

Examples for UML Tool Support

You can find a list of tools here:
https://en.wikipedia.org/wiki/List_of_Unified_Modeling_Language_tools

- Microsoft Visio has support for UML shapes diagram drawing
- IBM's Rational Rose is a software development tool based on UML
- ArgoUML is an open source tool for developing UML models
  - https://github.com/argouml-tigris-org/argouml

Others:

https://github.com/ModelioOpenSource/Modelio
https://www.lucidchart.com/pages/
http://www.visual-paradigm.com/solution/freeumldesigntool/
http://yuml.me

USE is an open source tool which supports UML class diagrams and Object Constraint Language

- https://sourceforge.net/projects/useocl/

UML References

- There are lots of books on UML. The ones I used are:
  - "UML Distilled," Martin Fowler
    - The examples I use in this lecture are from this book
  - "Using UML," Perdita Stevens
  - "UML Explained," Kendall Scott
  - "UML User Guide," Grady Booch, James Rumbaugh, Ivar Jacobson

- The Object Management Group (OMG, a computer industry consortium) defines the UML standard
  - The current UML language specification is available at:

    http://www.uml.org/

UML

- UML can be used in all phases of software development
  - specification of requirements, architectural design, detailed design and implementation

- There are different types of UML diagrams for specifying different aspects of software:
  - Functionality, requirements
    - Use-case diagrams
  - Architecture, modularization, decomposition
    - Class diagrams (class structure)
    - Component diagrams, Package diagrams, Deployment diagrams (architecture)
  - Behavior
    - State diagrams, Activity diagrams
  - Communication, interaction
    - Sequence diagrams, Collaboration diagrams

UML Class Diagrams

- Class diagram describes
  - Types of objects in the system
  - Static relationships among them
- Two principal kinds of static relationships
  - Associations between classes
  - Subtype relationships between classes
- Class descriptions show
  - Attributes
  - Operations
- Class diagrams can also show constraints on associations

UML Class Diagrams

- Class diagrams can be used at different stages of development
  - For requirements specification, for design specification, and for implementation

- In requirements specification class diagrams can be used to model real world objects or concepts

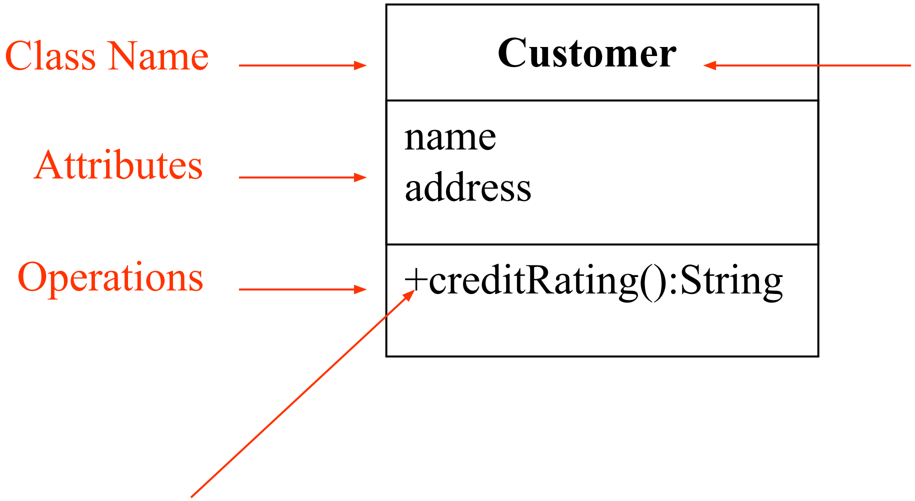
- In design specification it can be used to specify interfaces and classes that will be implemented in an object oriented program

- In implementation they can be used to show the structure of the software by showing the relationships among different classes

Classes

- A class is represented as a three-part box
- **Class Name**
- **Attributes**
  - At conceptual level it is a piece of information associated with the class that can be accessed and possibly modified
  - Corresponds to a field at the implementation level
  - Difference from association: navigability is from class to attribute (not both ways as in association)
- **Operations**
  - The processes the class can carry out (methods at implementation level)
  - Basic operations (such as getValue) on attributes can be omitted (they can be inferred)

Classes



For abstract classes, class name is written in italic: ***Customer***

Visibility:
public + (default) any outside class with visibility to the given class can use the feature
protected # any descendant of the class can use the feature
private – only the class itself can use the feature

Classes

Attribute syntax: *visibility name*[*multiplicity*] : *type* = *initial-value* { *property-string* }

can be:
changeable (is modifiable)
addOnly (for collections, items can be added but cannot be removed)
frozen (no modification is allowed)

Example: − accountName [0..1] : String {changeable}

Operation syntax: *visibility name* (*parameter-list*) : *return-type* { *property-string* }

Parameters can be marked as:
in: input parameter (cannot be modified)
out: output parameter
inout: an input parameter that can be modified

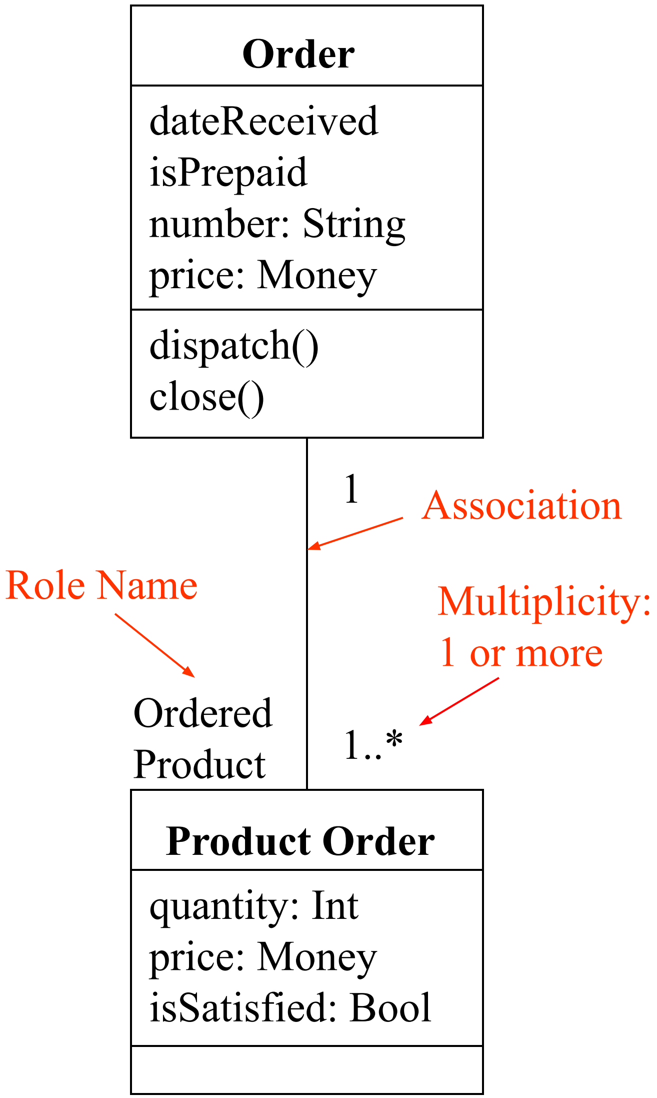
can be:
isQuery (does not change state of the object)
sequential (should not be called concurrently)
guarded (like synchronized in Java)
concurrent (can be executed concurrently)

Example: + getAccountName (number : Integer) : String {isQuery}

Associations

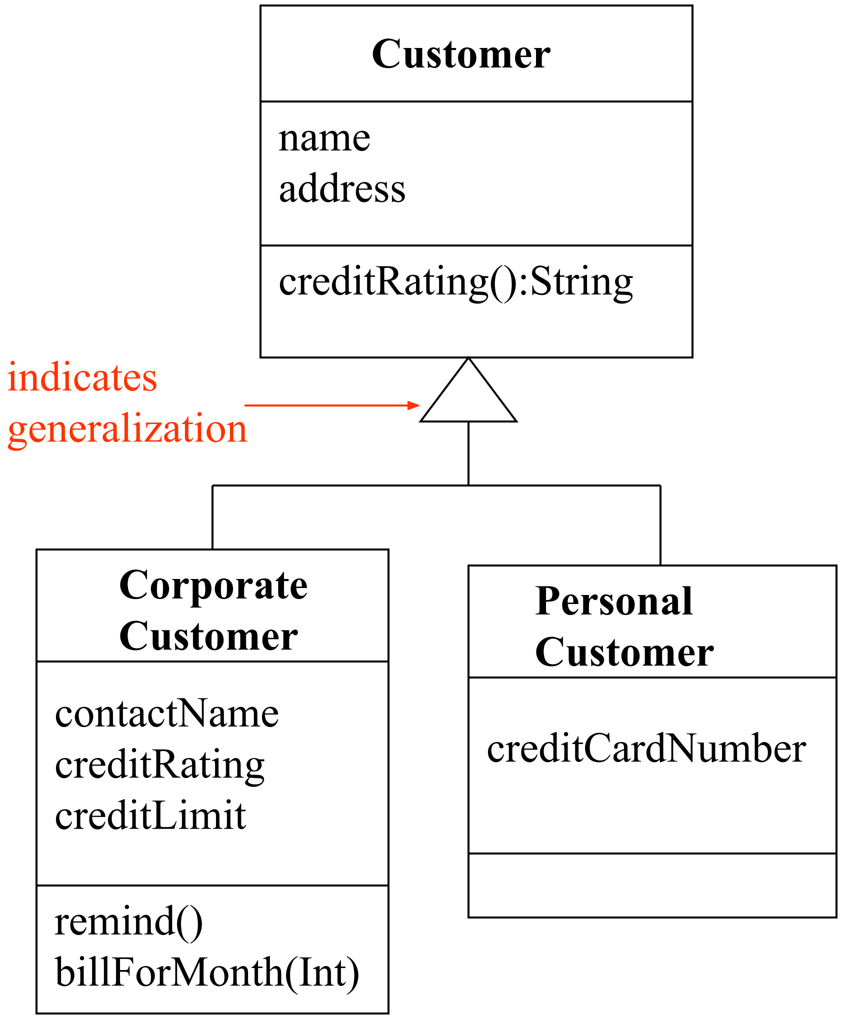
Associations are shown as lines between classes

- An association shows a relationship between instances of two classes
  - Each association has two roles (one for each direction)
  - A **role** can be explicitly named with a label
  - Roles have multiplicity showing how many objects participate in the relationship
  - Associations can have **multiplicities**
    - A fixed value (such as 1 or 3)
    - Many denoted by * (unlimited number, 0 or more)
    - A range of values 0..1 or 3..*
    - A set of values 2,4,8



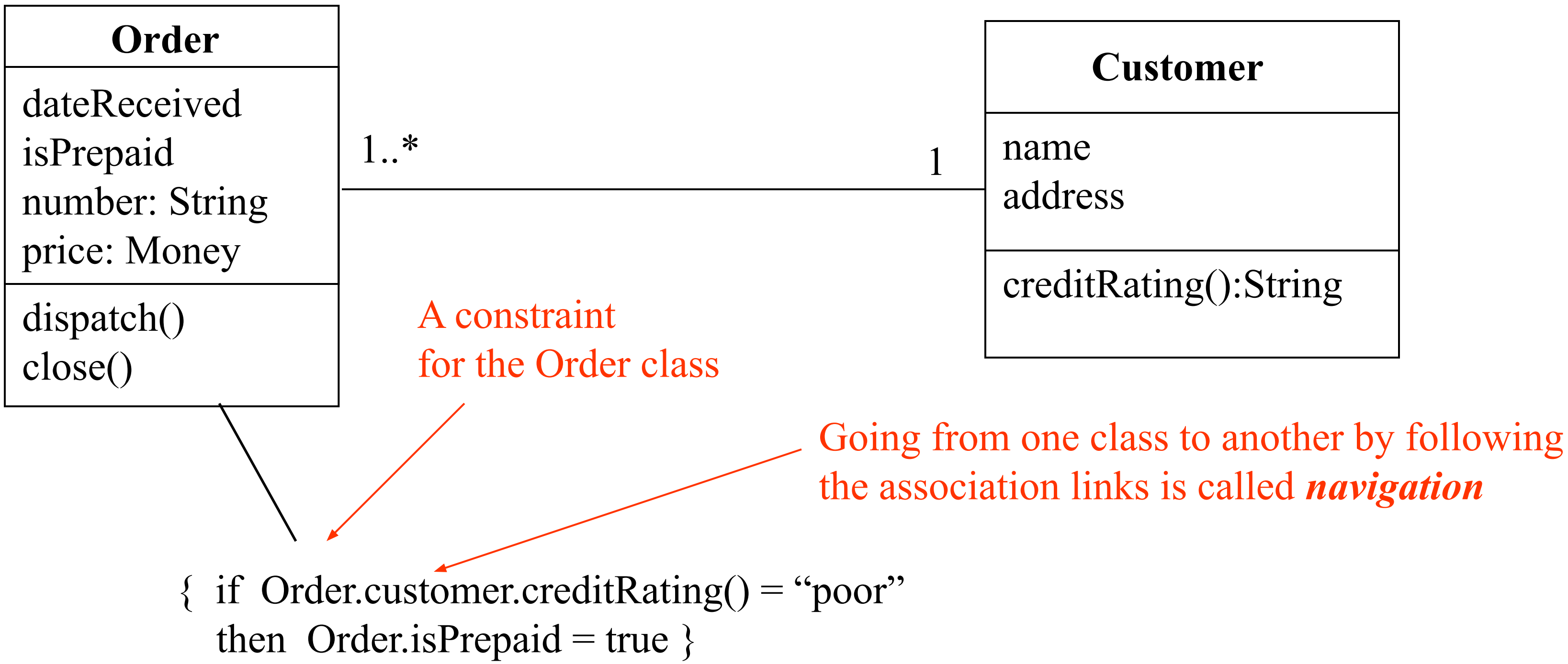
Generalization

Generalization is used to show subtyping between classes
- Subtype is a specialization of the supertype
- Subtype can be substituted for the supertype
- Subtype inherits the interface
- Subtype inherits the operations

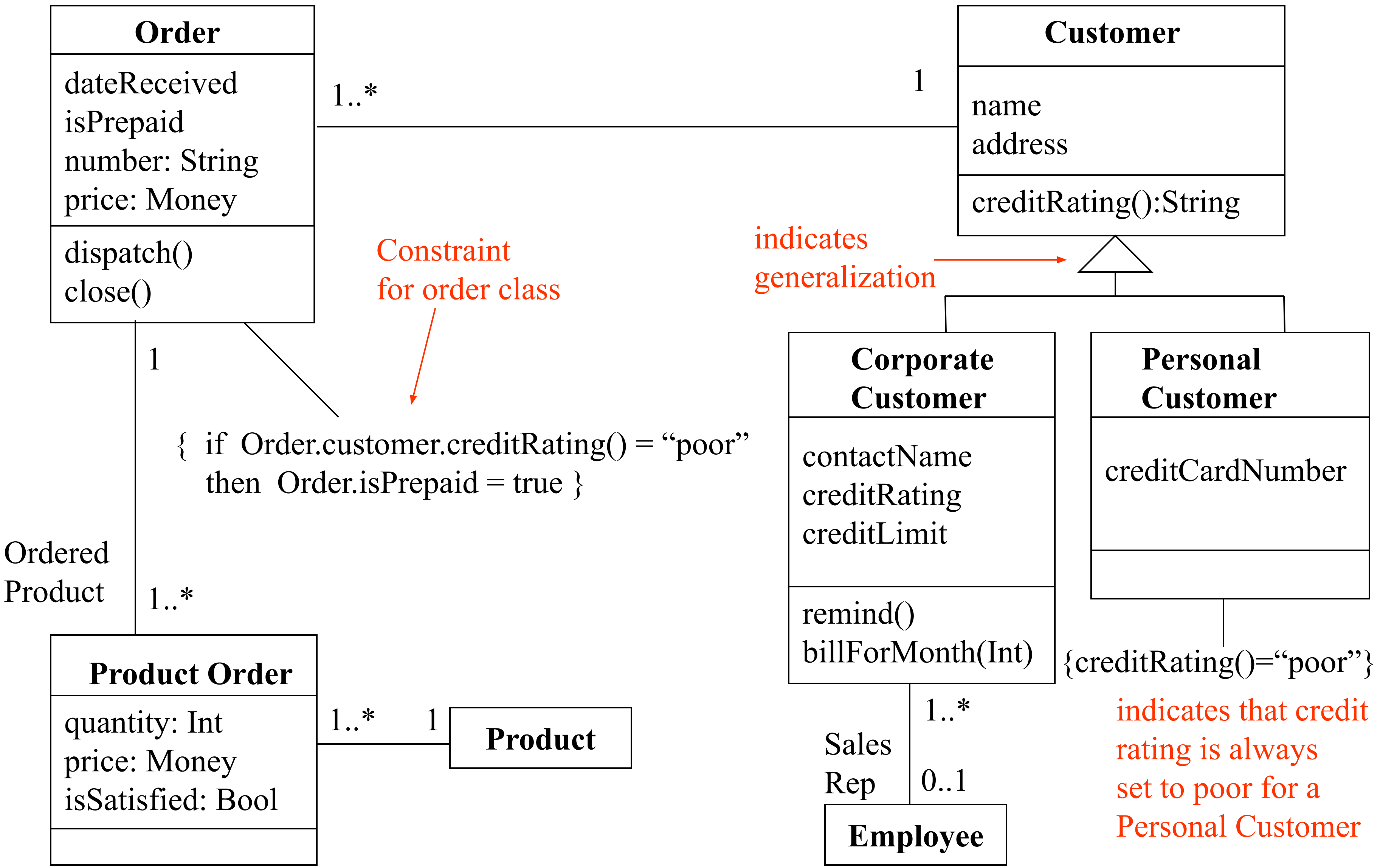


Constraints

- Constraints can be used to represent further restrictions on associations or classes
- Constraints are stated inside braces {}
  - Object Constraint Language (OCL) is a formal language for specifying constraints



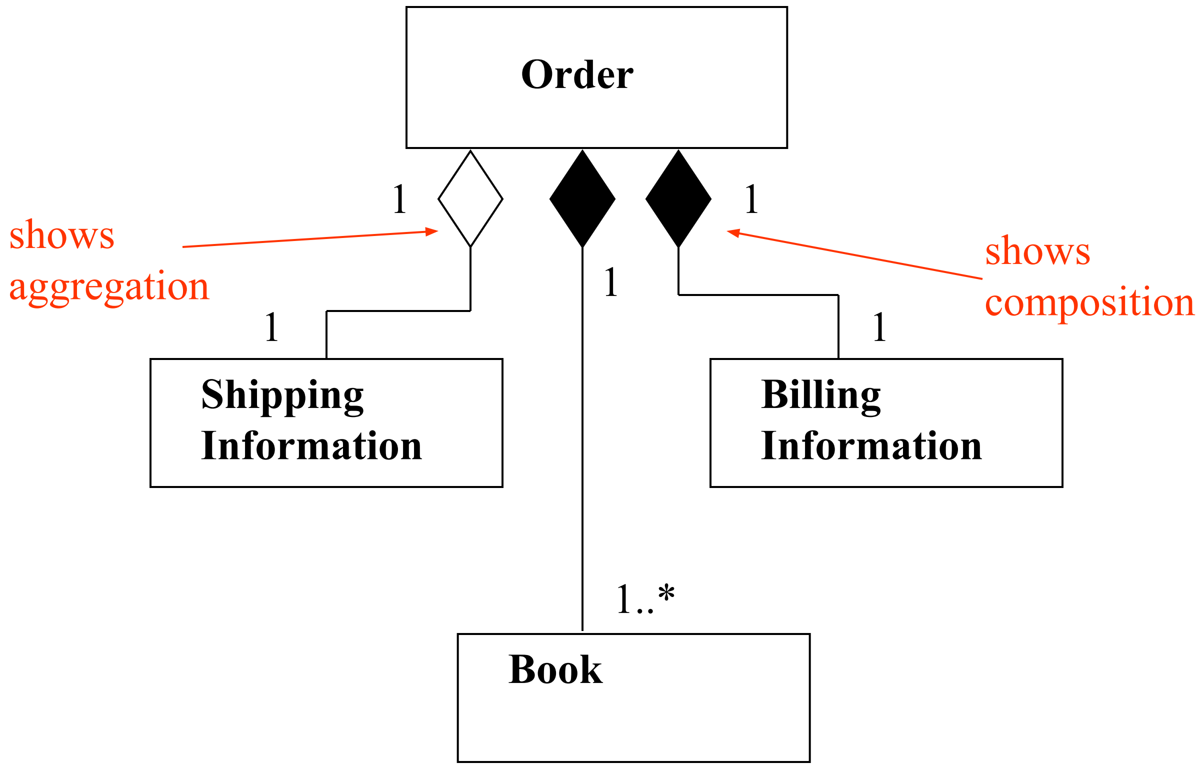
Example Class Diagram



Aggregation and Composition

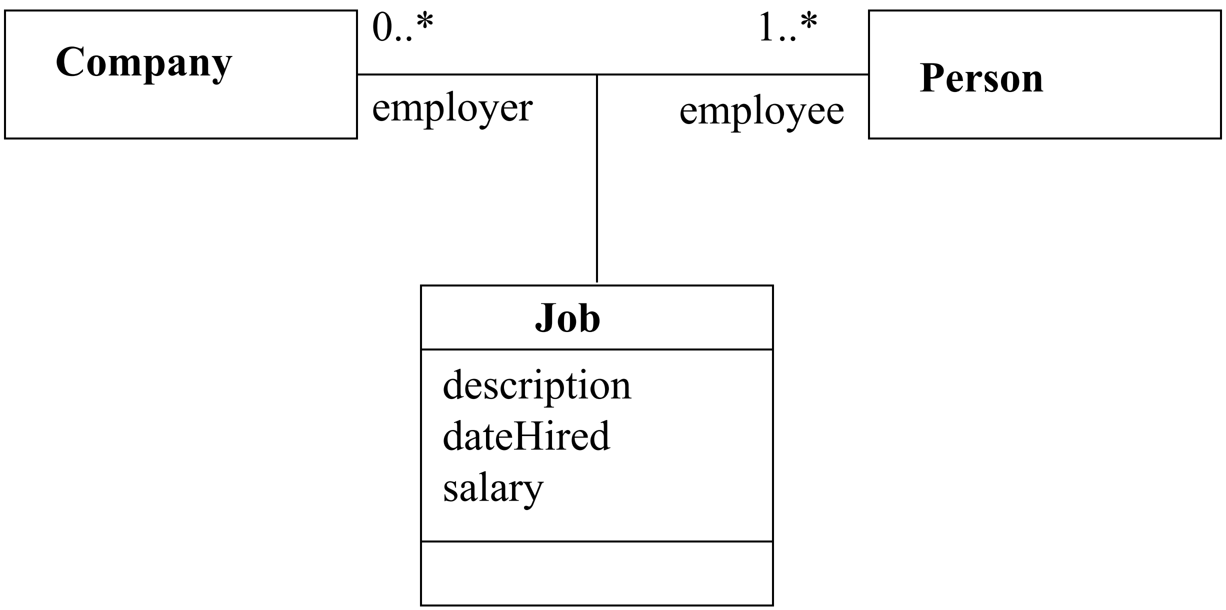
- Aggregation is a part-of relationship

- Composition is also a part-of relationship, but part and whole created and destroyed together



Association Classes

- Adds attributes and operations to an association
  - Allows exactly one instance of the association class between any two objects
    - Can use an actual class instead if you need more instances



Sequence Diagrams

- A sequence diagram shows a particular sequence of messages exchanged between a number of objects
- Sequence diagrams also show behavior by showing the ordering of message exchange
- A sequence diagram shows some particular communication sequences in some run of the system
  - it is not characterizing all possible runs

Sequence Diagrams

- Sequence diagrams can be used in conjunction with use-cases
  - At the requirements phase they can be used to visually represent the use cases
  - At the design phase they can be used to show the system's behavior that corresponds to a use-case

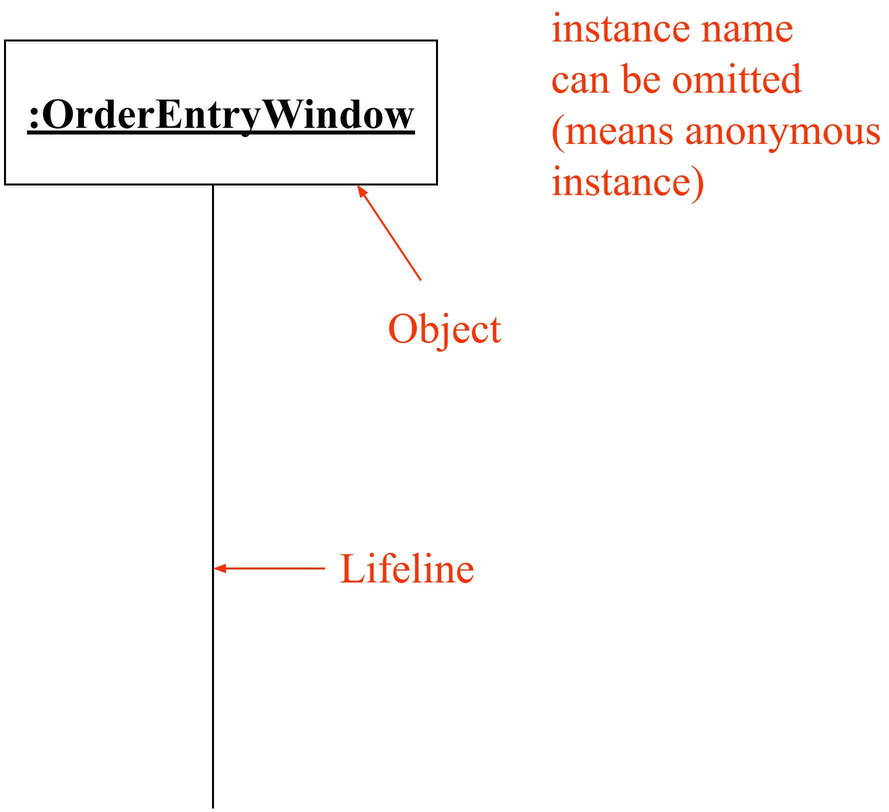
- During the testing phase sequence diagrams from the requirements or design phases can be used to generate test cases for the software product

- Sequence diagrams are similar to MSCs (Message Sequence Charts) which are a part of SDL(Specification and Description Language) and have formal semantics

Components of Sequence Diagrams

- Object (an instance of a class)
  - shown as a box at the top of a vertical dashed line
  - instance syntax
    <u>*instanceName:ClassName*</u>

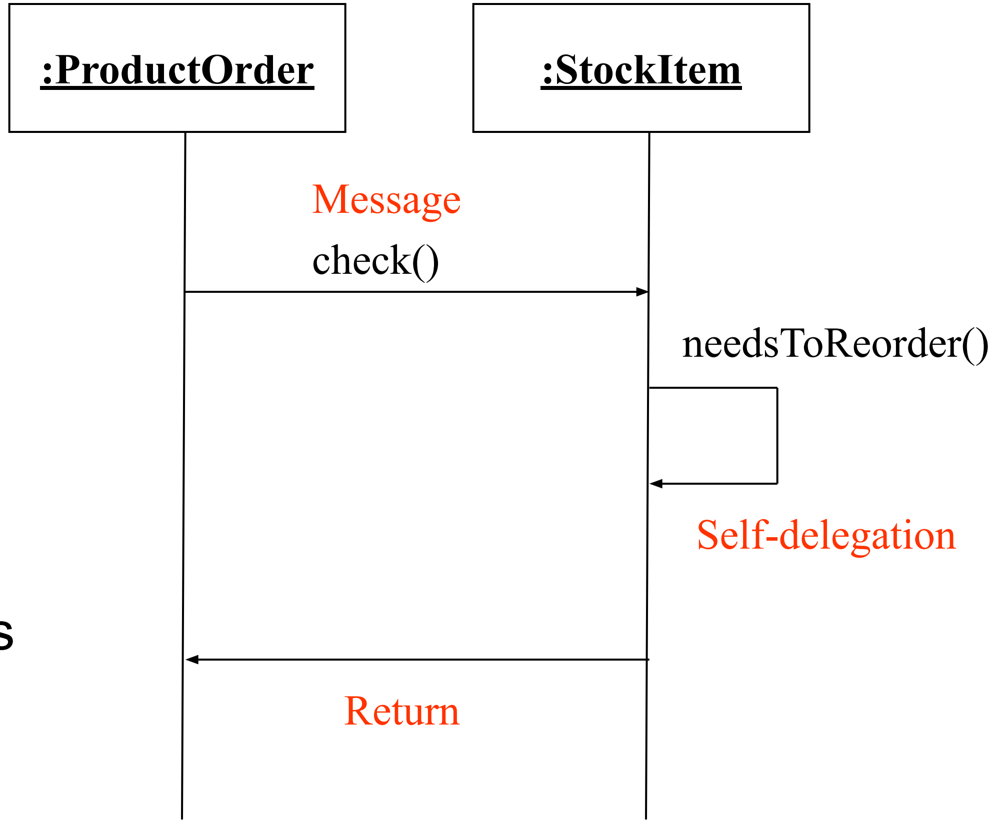
- Lifeline
  - represents time flow



Components of Sequence Diagrams

- Messages
  - communication between objects
  - correspond to method calls at the implementation level

- Special message types
  - self-delegation
  - return
    - show returns only if it adds to clarity
  - <<create>>
  - <<destroy>>

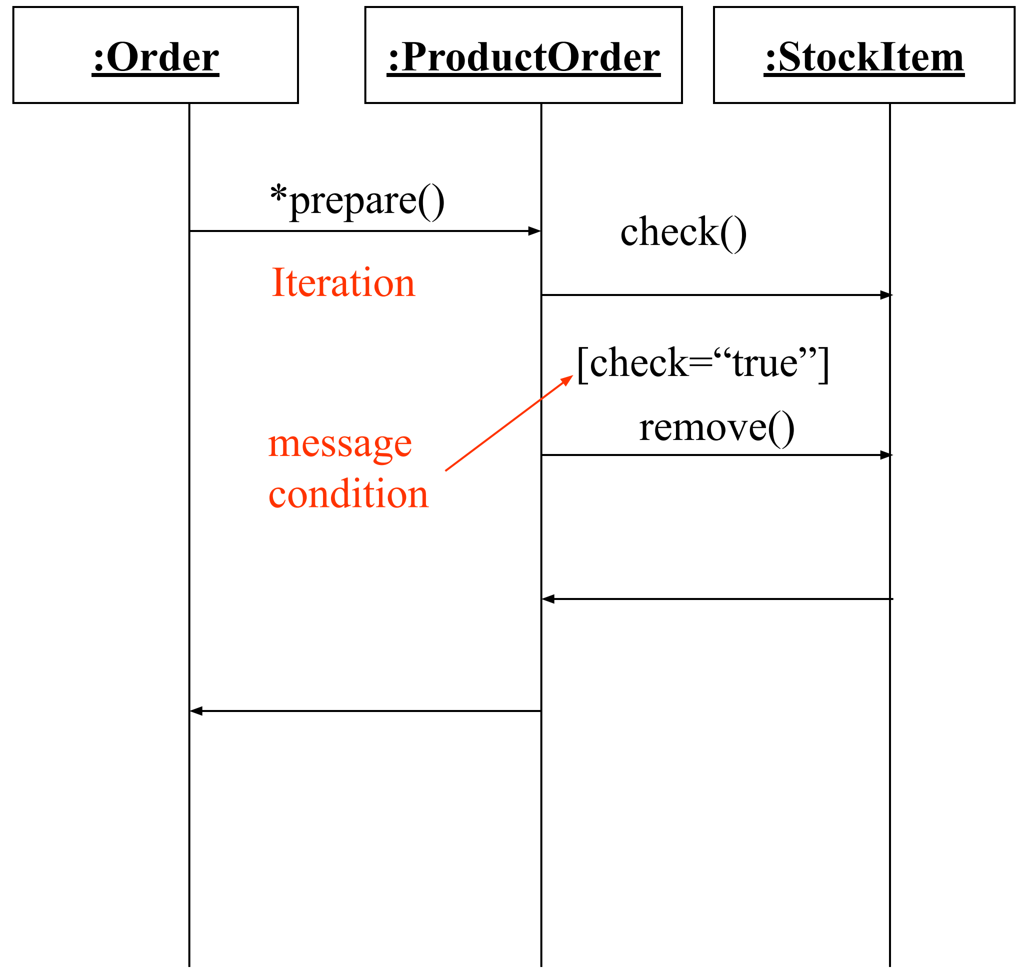


 Denotes procedure call (control flow passes from caller to callee)

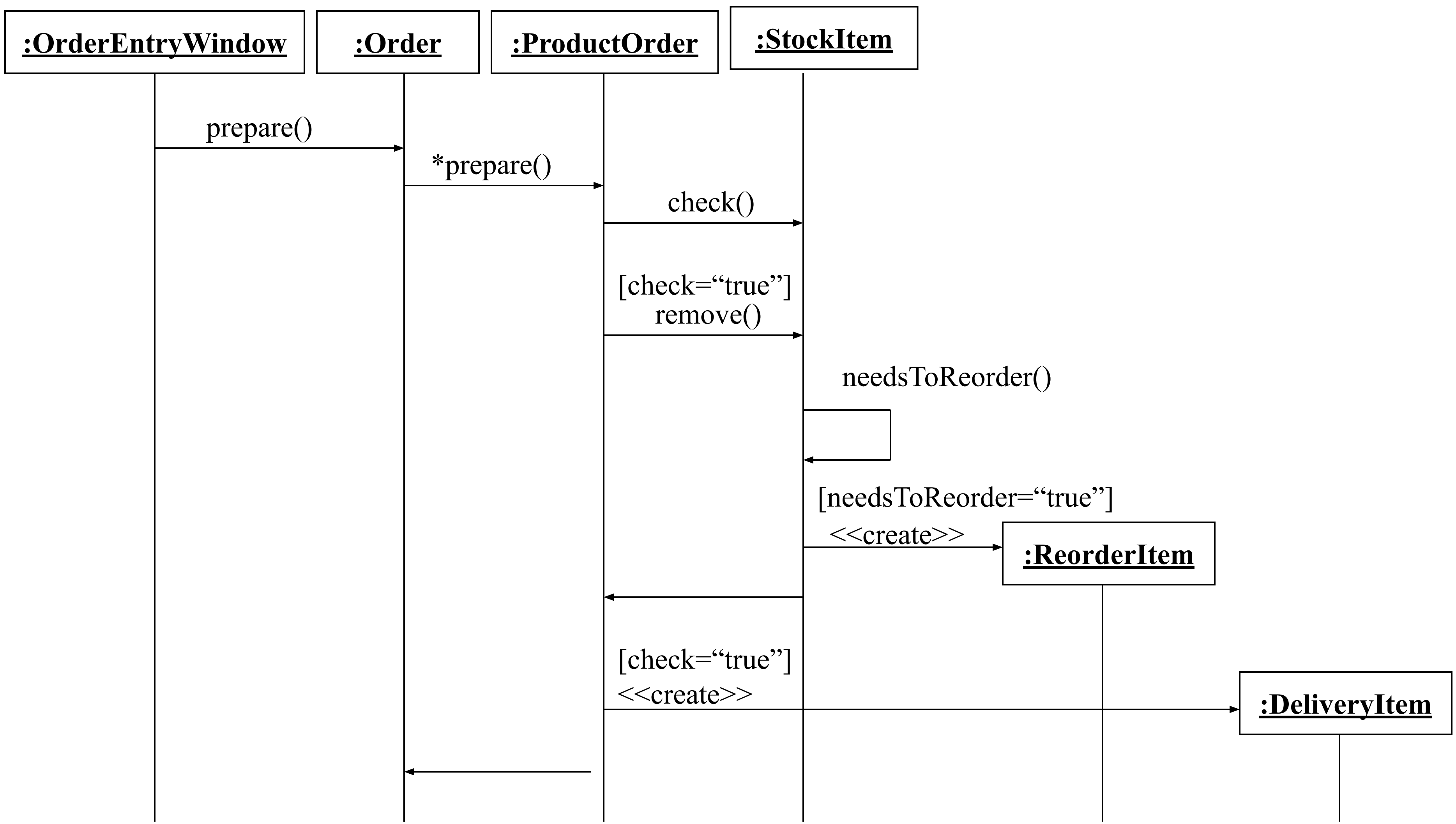
 Denotes interaction among two threads of control (no transfer of control)

Components of Sequence Diagrams

- Two kinds of control information:
  - message conditions
    - message is sent only if the condition is true
  - iteration marker: *
    - message sent to multiple receiver objects



Example Sequence Diagram



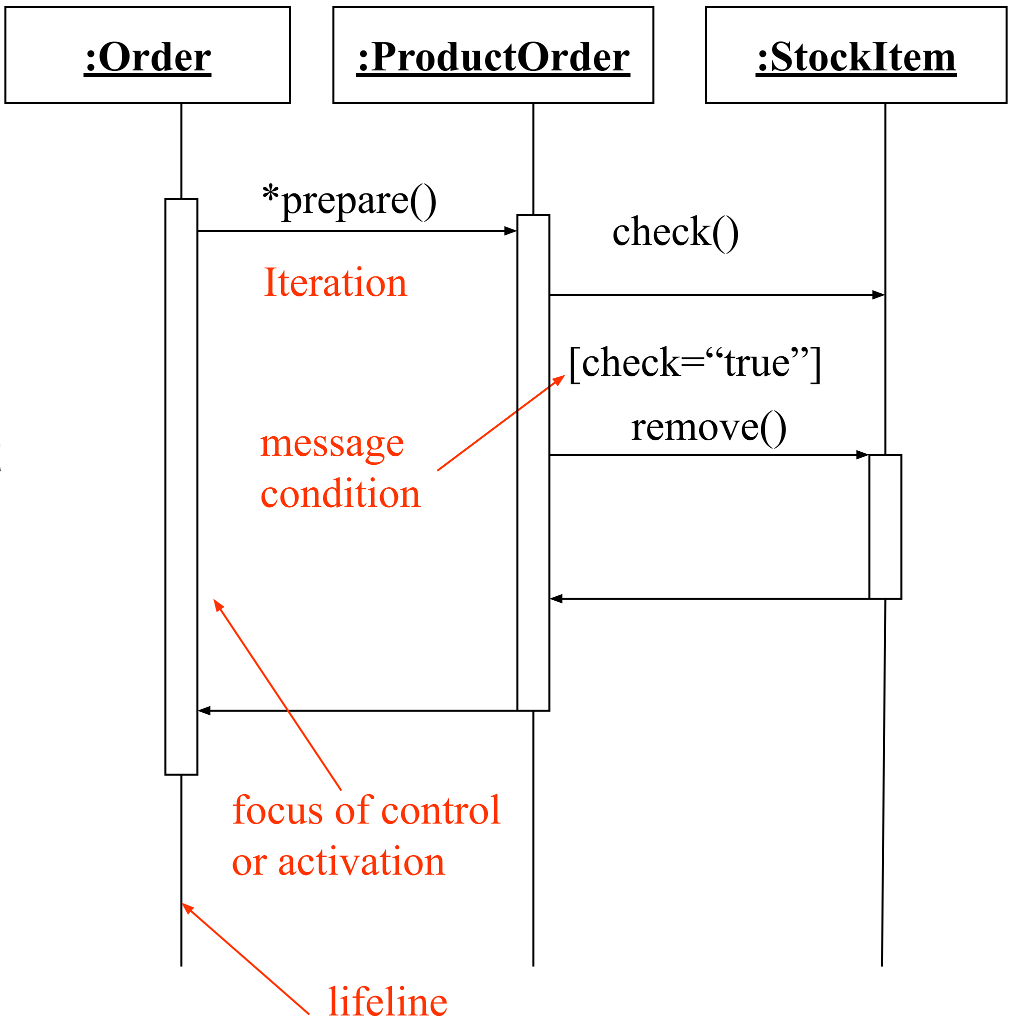
Sequence diagrams

- Show conditional behavior on separate diagrams to keep them understandable
  - for example for a use case you can give the basic path as one sequence diagram and have separate sequence diagrams for alternative paths or exceptions

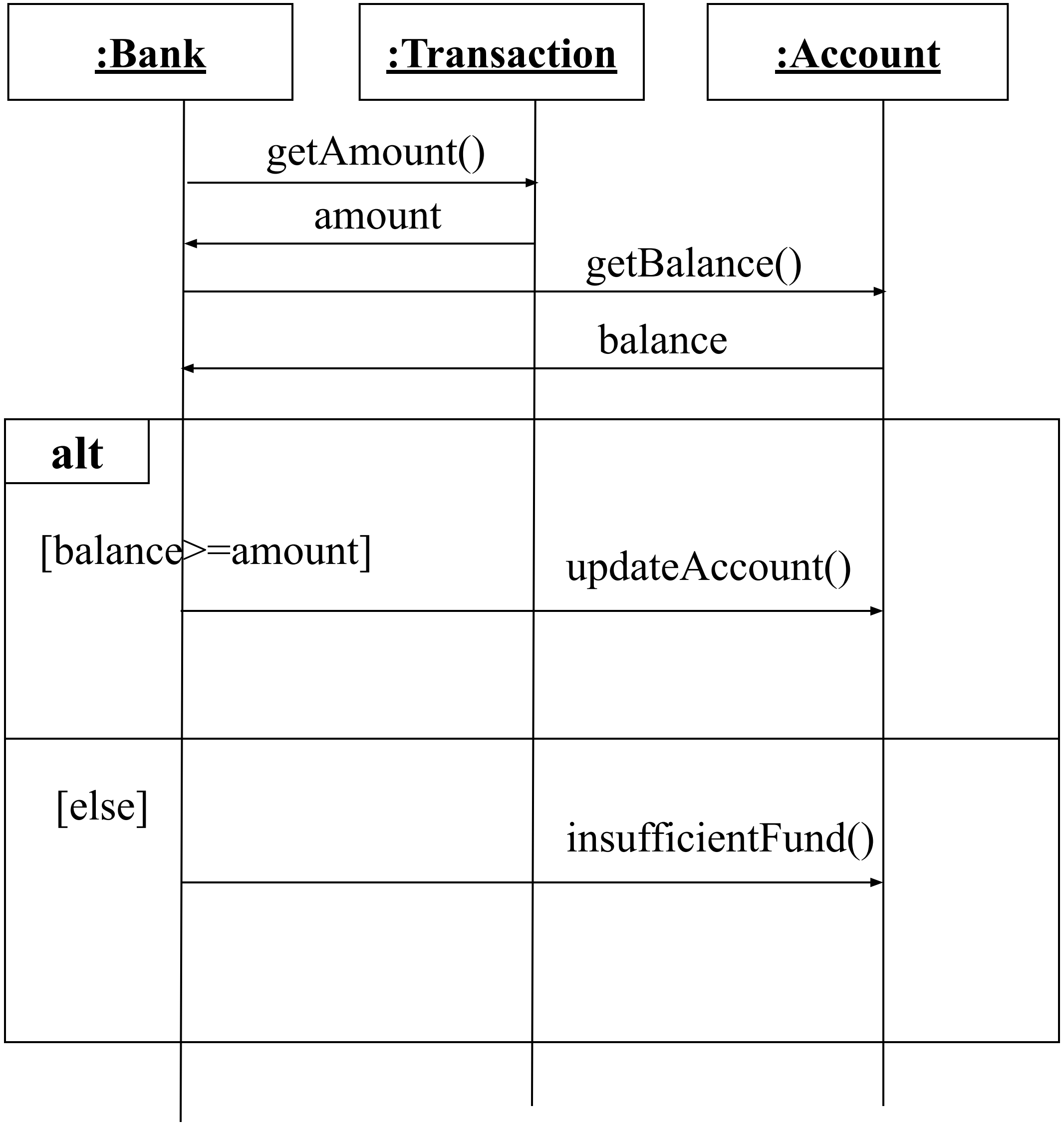
- Use sequence diagrams to show the behavior of several objects within a use case
  - use a state diagram when you want to show the behavior of an object across many use cases

Sequence Diagrams

- Focus of control (or activation) can be shown in sequence diagrams as a thin rectangle put on top of the lifeline of an object
- Shows the period of time during which the given object is in control of the flow
  - From an implementation point of view, you can think of it as showing how long an activation record stays in the control stack
- It is optional to use focus of control rectangles in a sequence diagram
  - use it when it adds to clarity



Sequence Diagram Frames

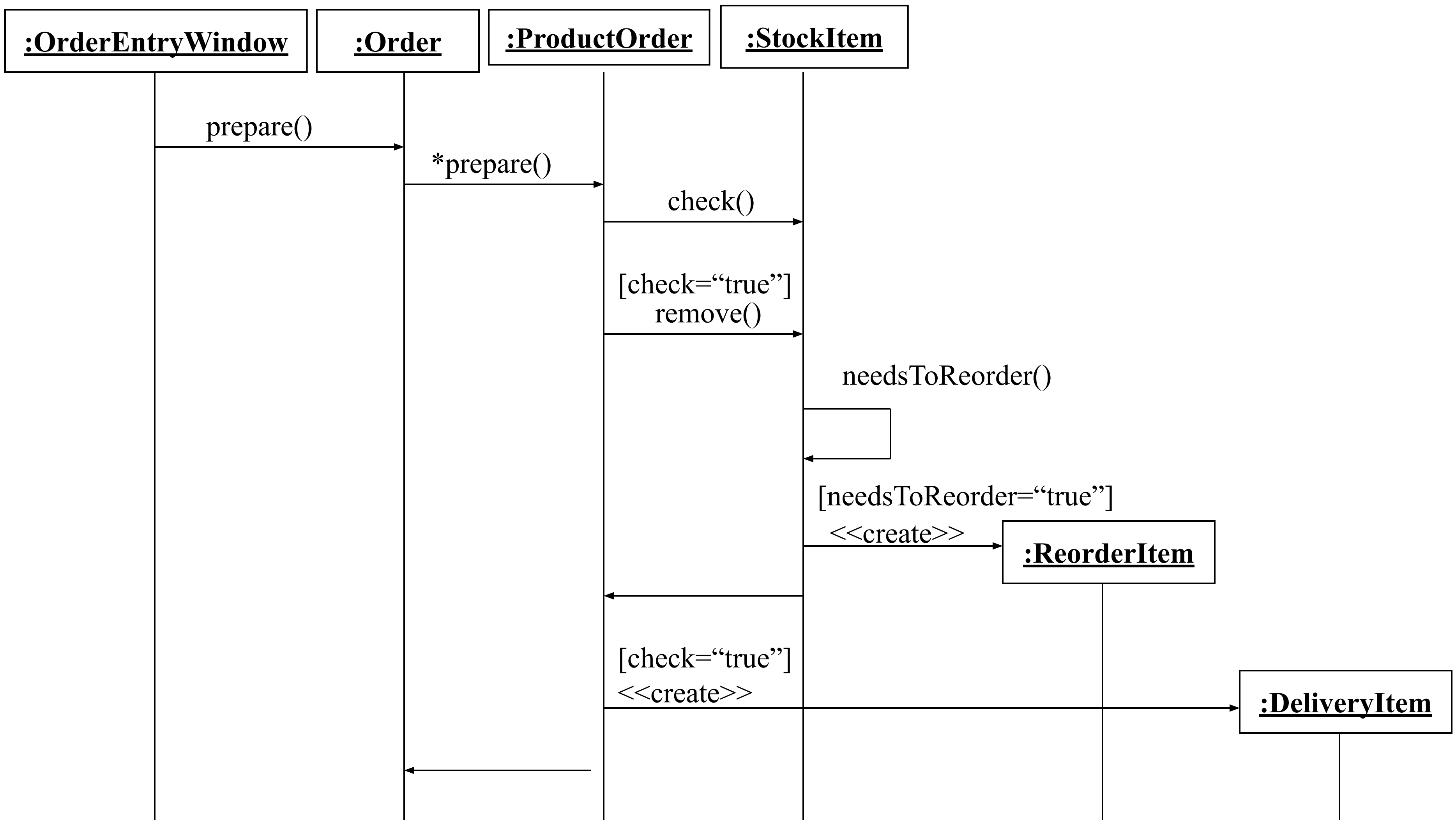


- Frames can be used to specify conditional behavior (as seen in the example), loops, optional behavior etc. in sequence diagrams

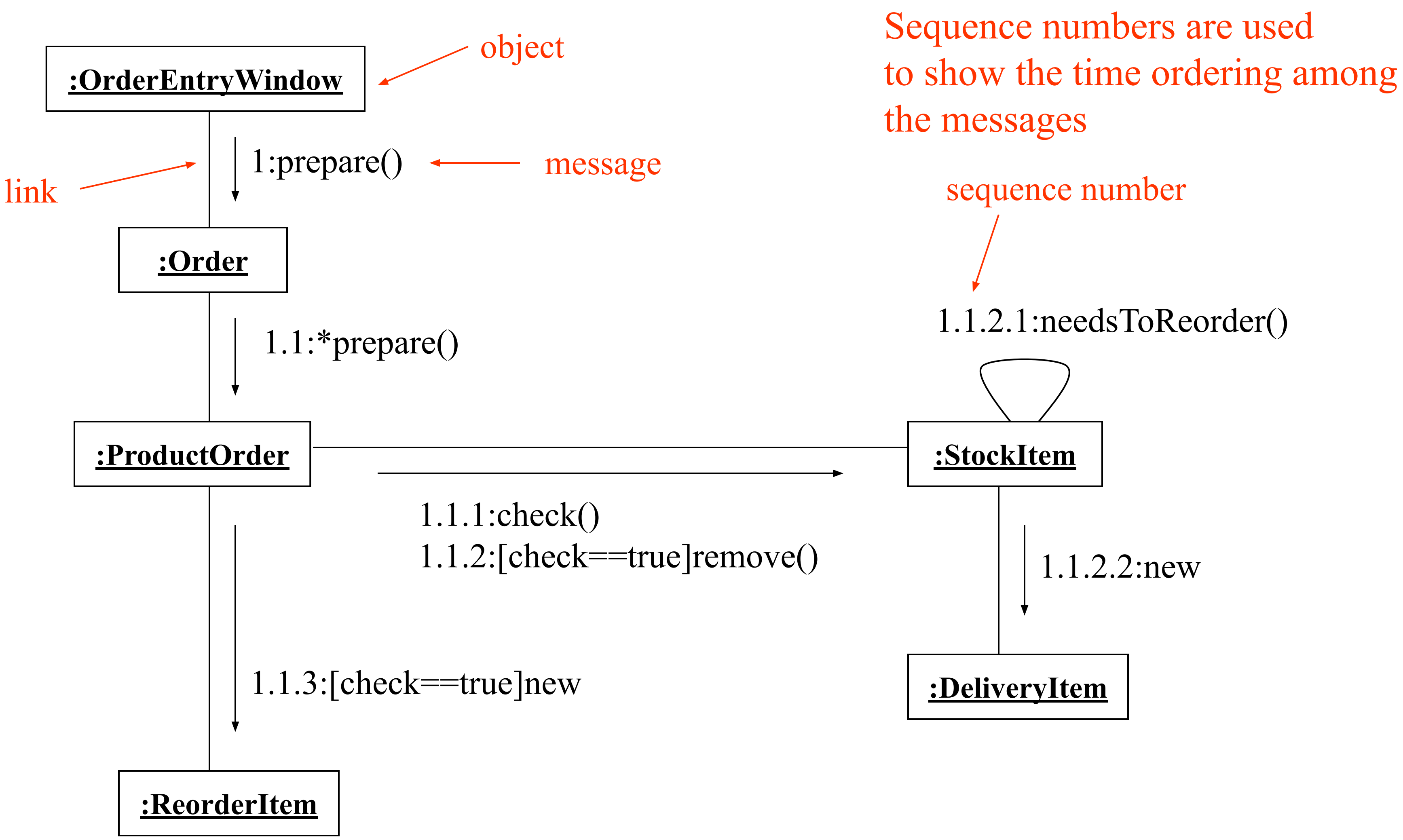
Collaboration (Communication) Diagrams

- Collaboration diagrams (aka Communication diagrams) show a particular sequence of messages exchanged between a number of objects
  - this is what sequence diagrams do too!
- Use sequence diagrams to model flows of control by time ordering
  - sequence diagrams can be better for demonstrating the ordering of the messages
  - sequence diagrams are not suitable for complex iteration and branching
- Use collaboration diagrams to model flows of control by organization
  - collaboration diagrams are good at showing the static connections among the objects while demonstrating a particular sequence of messages at the same time

Example Sequence Diagram



Corresponding Collaboration Diagram



State Diagrams (Statecharts a la UML)

- State diagrams are used to show possible states a single object can get into
  - shows states of an object

- How object changes state in response to events
  - shows transitions between states

- UML state diagrams are a variation of Statecharts
  - "A Visual Formalism for Complex Systems," David Harel, Science of Computer Programming, 1987
  - Statecharts are basically hierarchical state machines
  - Statecharts have formal semantics

State Diagrams

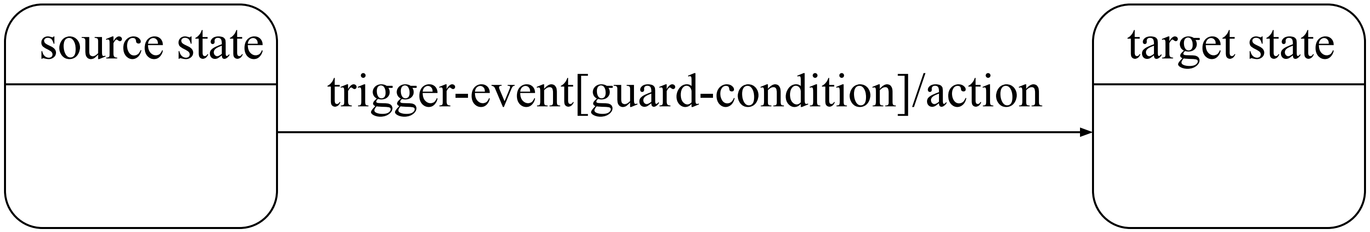
- State diagrams are used to show possible states a single object can get into
  - shows states of an object
- How object changes state in response to events
  - shows transitions between states
- Uses the same basic ideas from statecharts and adds some extra concepts such as internal transitions, deferred events etc.

State Diagrams

- Hierarchical grouping of states
  - composite states are formed by grouping other states
  - A composite state has a set of sub-states
- Concurrent composite states can be used to express concurrency
  - When the system is in a concurrent composite state, it is in all of its substates at the same time
  - When the system is in a normal (non-concurrrent) composite state, it is in only one of its substates
  - If a state has no substates it is an atomic state
- Synchronization and communication between different parts of the system is achieved using events

State Diagrams: Transitions

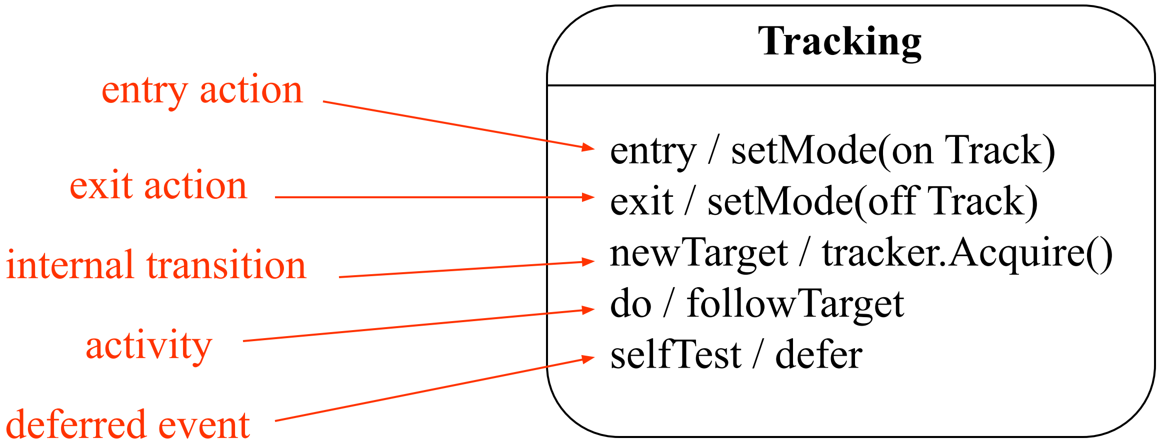
- Transitions consist of
  - **source state** and **target states**: shown by the arrow representing the transition
  - **trigger event**: the event that makes the transition fire, for example it could be receipt of a message
  - **guard condition**: a boolean expression that is evaluated when the trigger event occurs, the transition can fire only if the guard condition evaluates to true
  - **action**: an executable atomic computation that can directly act on the object that owns the state machine or indirectly on other objects that are visible to the object such as sending a message



State Diagrams: States

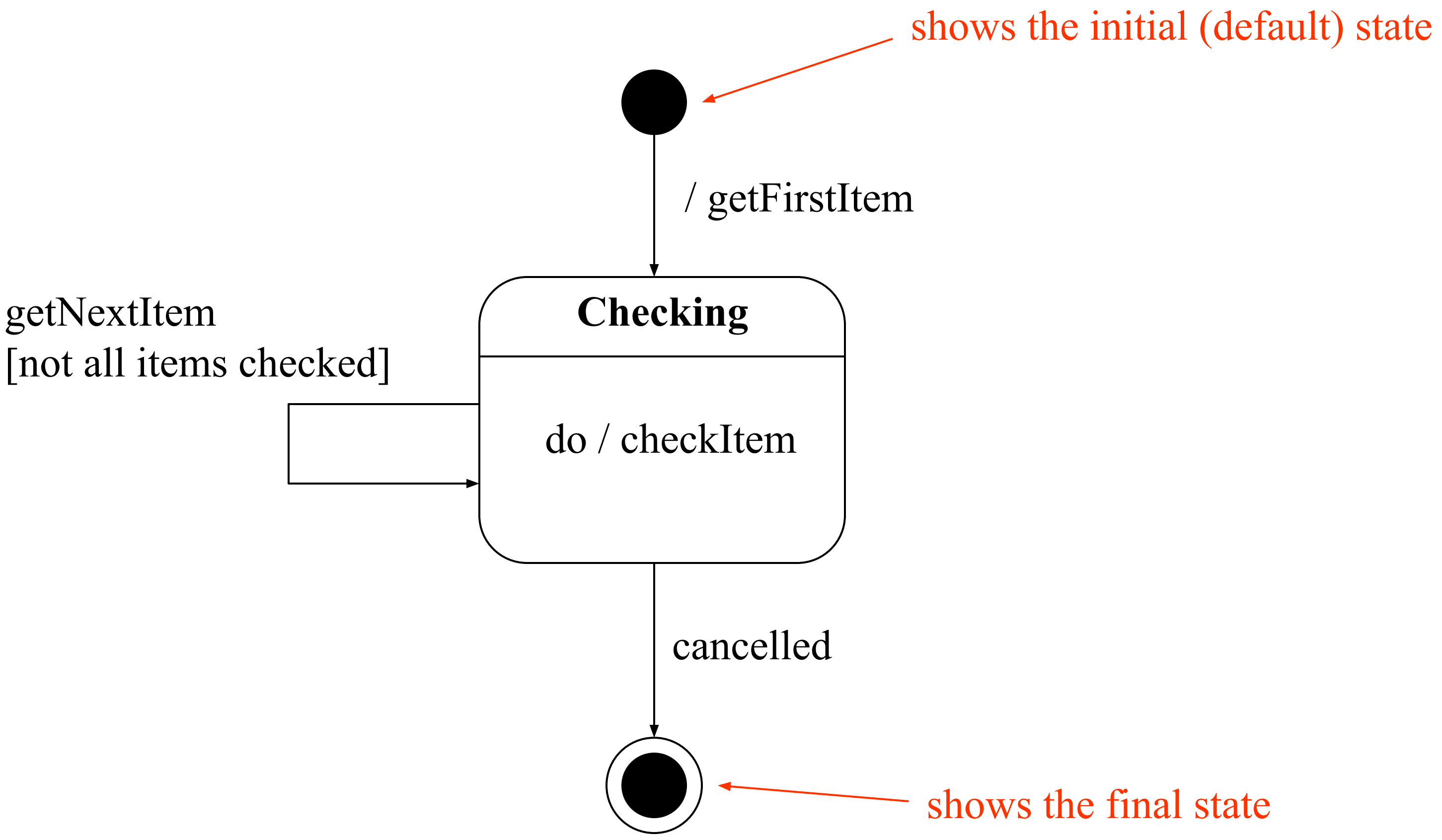
- States are represented as rounded boxes which contain:
  - the **state name**
  - and the following optional fields
    - **entry and exit actions**: entry and exit actions are executed whenever the state is entered or exited, respectively
    - **internal transitions**: internal transitions do not activate the entry and exit actions (different than self-transitions which activate the entry and exit actions).
    - **activities**: Typically, once the system enters a state it sits idle until an event triggers a transition. Activities help you to model situations where while in a state, the object does some work that will continue until it is interrupted by an event
    - **deferred events**: If an event does not trigger a transition in a state, it is lost. In situations where you want to save an event until it triggers a transition, use deferred events

State Diagrams: States



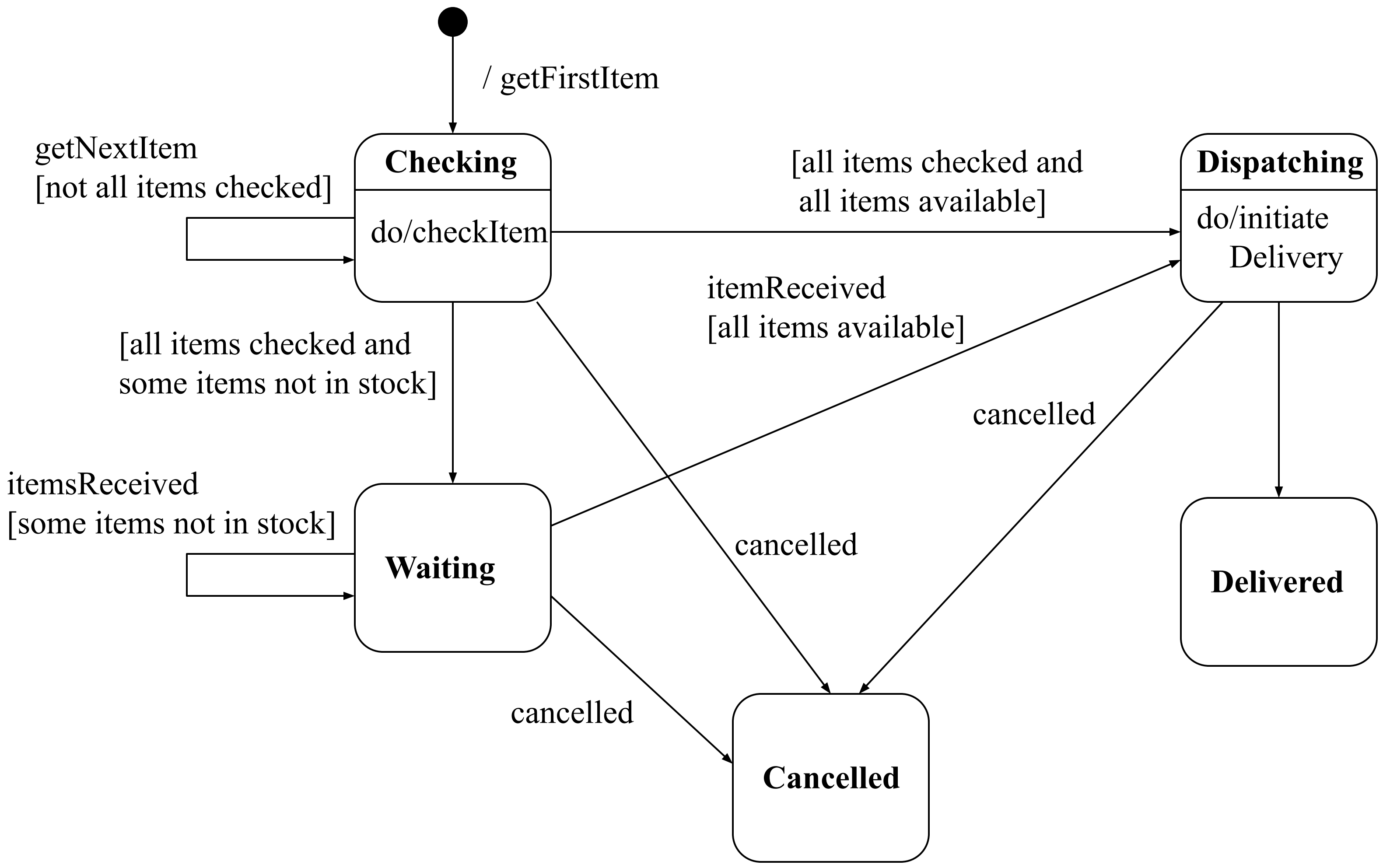
Note that, “entry”, “exit”, “do”, and “defer” are keywords

State Diagrams



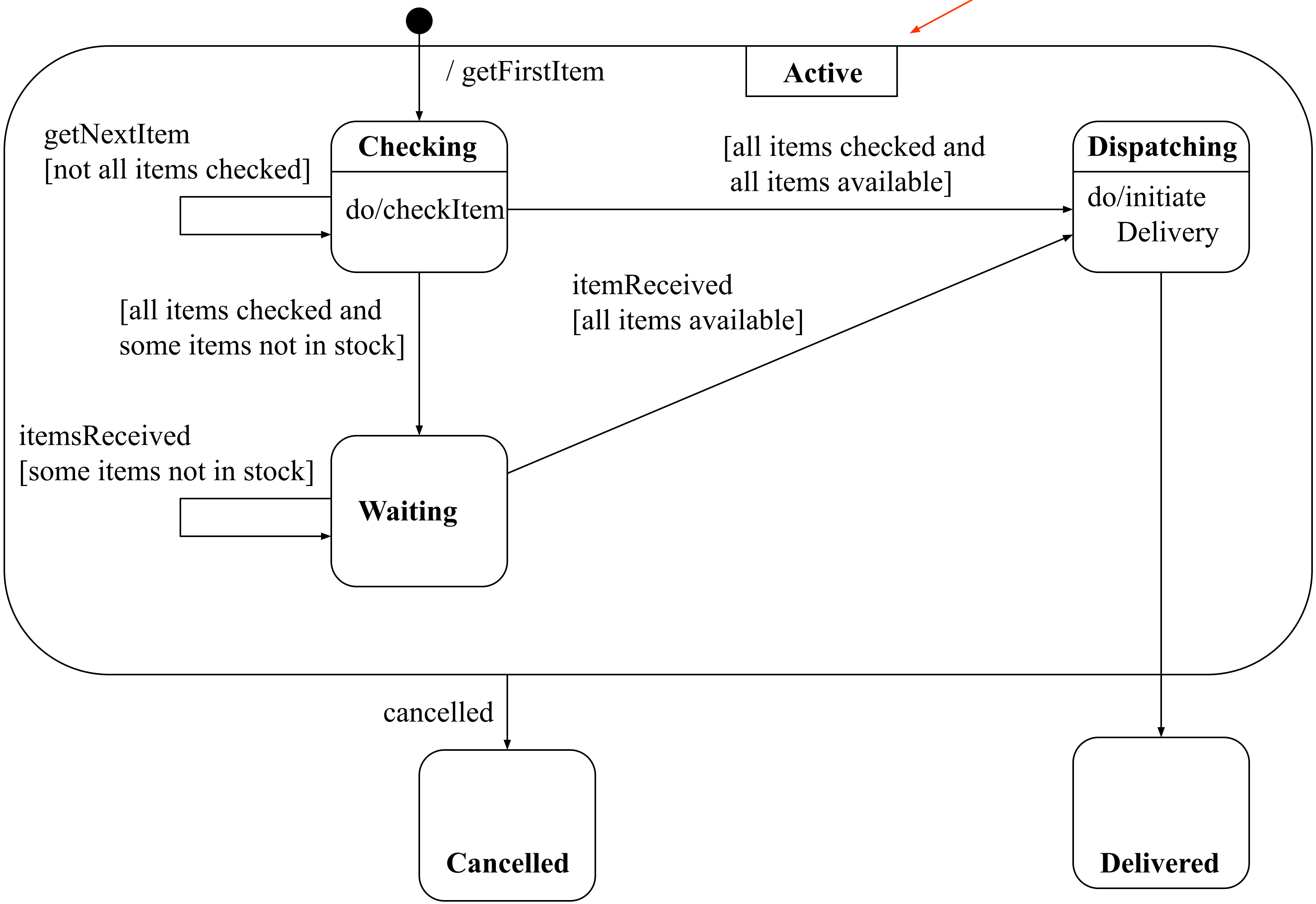
initial and final states: shown as filled black circle
and a filled black circle surrounded by an unfilled circle, respectively

State Diagram Example: States of an Order object



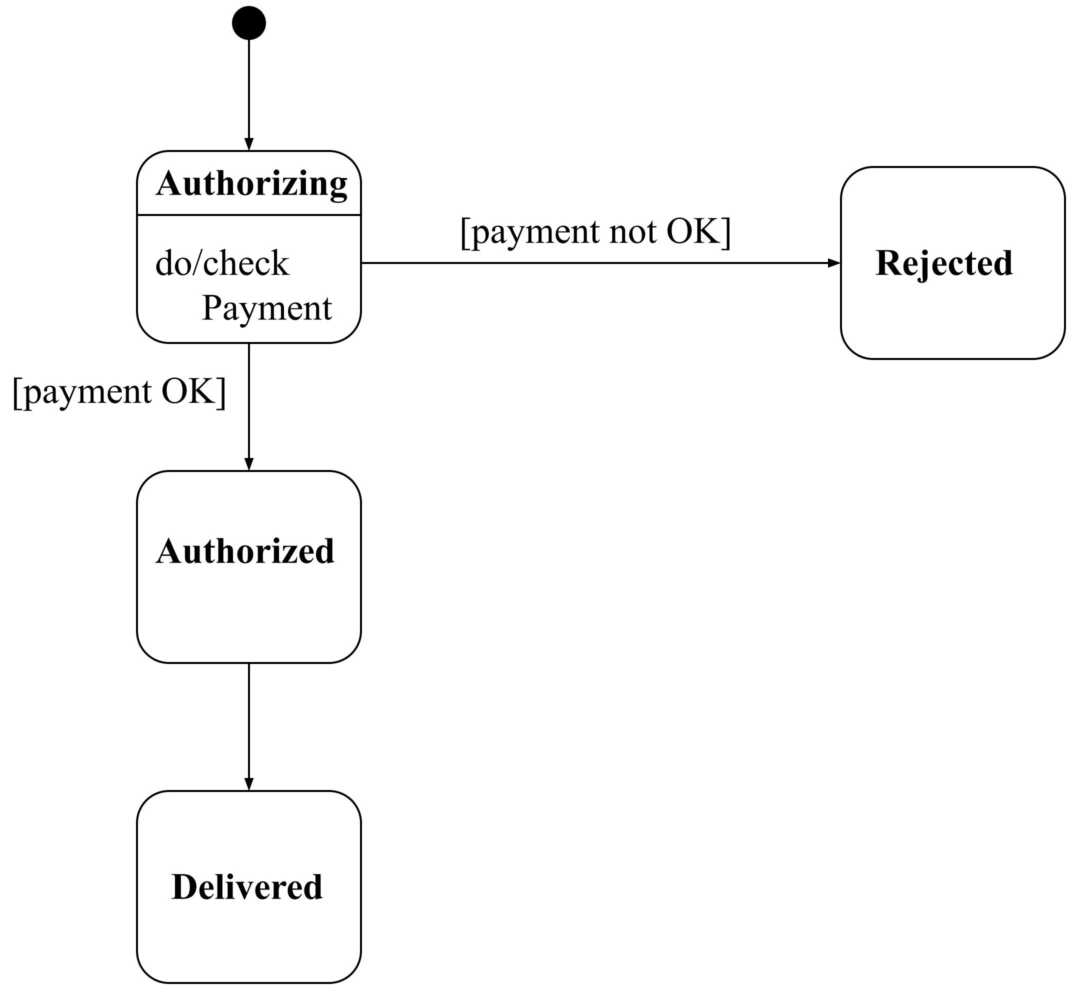
State Diagrams: Superstates

Active is a superstate with substates Checking, Waiting and Dispatching

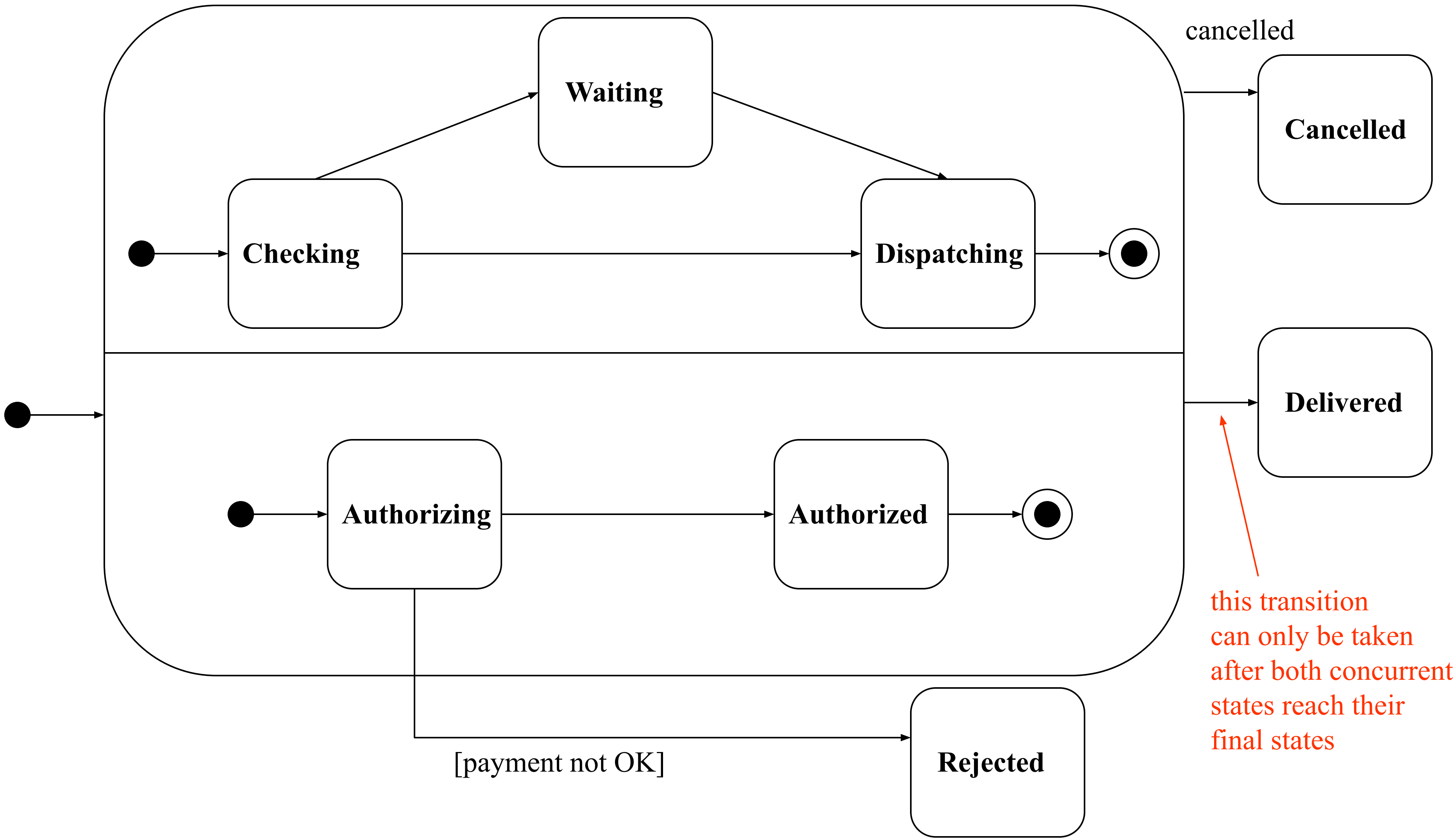


State Diagrams: Concurrent states

- Payment authorization is done concurrently with the order processing



State Diagrams: Concurrent States



State Diagrams

- Good at describing behavior of an object across several use-cases
- Use them to show the behavior of a single object not many objects
  - for many objects use interaction diagrams
- Do not try to draw state diagrams for every class in the system, use them to show interesting behavior and increase understanding

Activity Diagrams

- Activity diagrams show the flow among activities and actions associated with a given object using:
  - activity and actions
  - transitions
  - branches
  - merges
  - forks
  - joins

- Activity diagrams are similar to SDL state diagrams, SDL state diagrams have formal semantics

- Activity diagrams are basically an advanced version of flowcharts

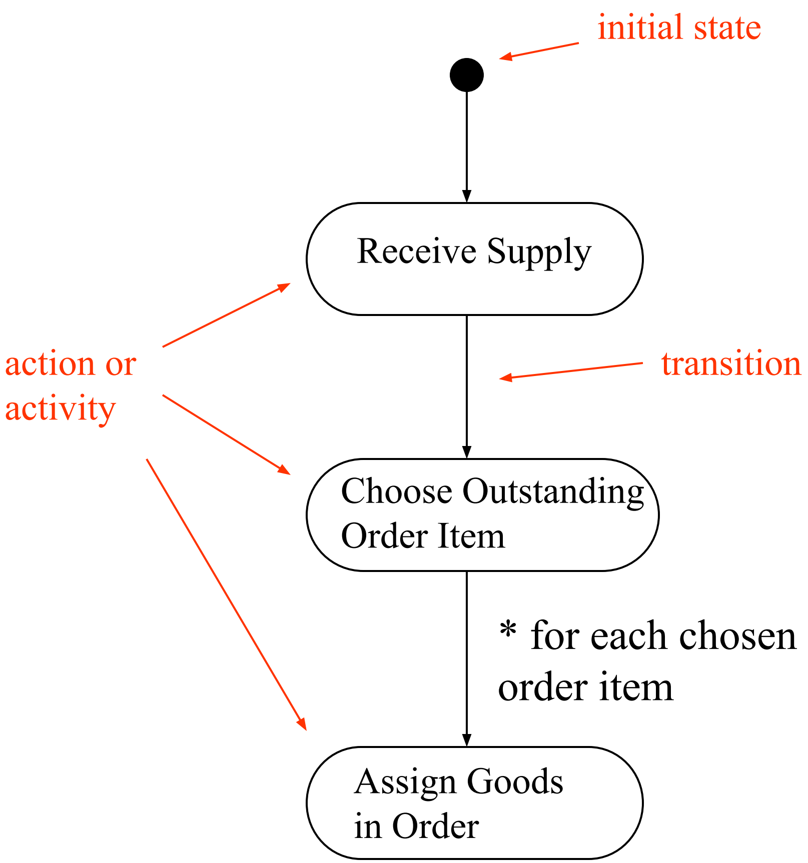
Activity Diagrams

- Activity
  - represents a task that has to be performed, a non-atomic execution within a state machine
  - from an implementation perspective it can represent a method

- Action
  - an atomic computation that changes the state of the system or returns a value

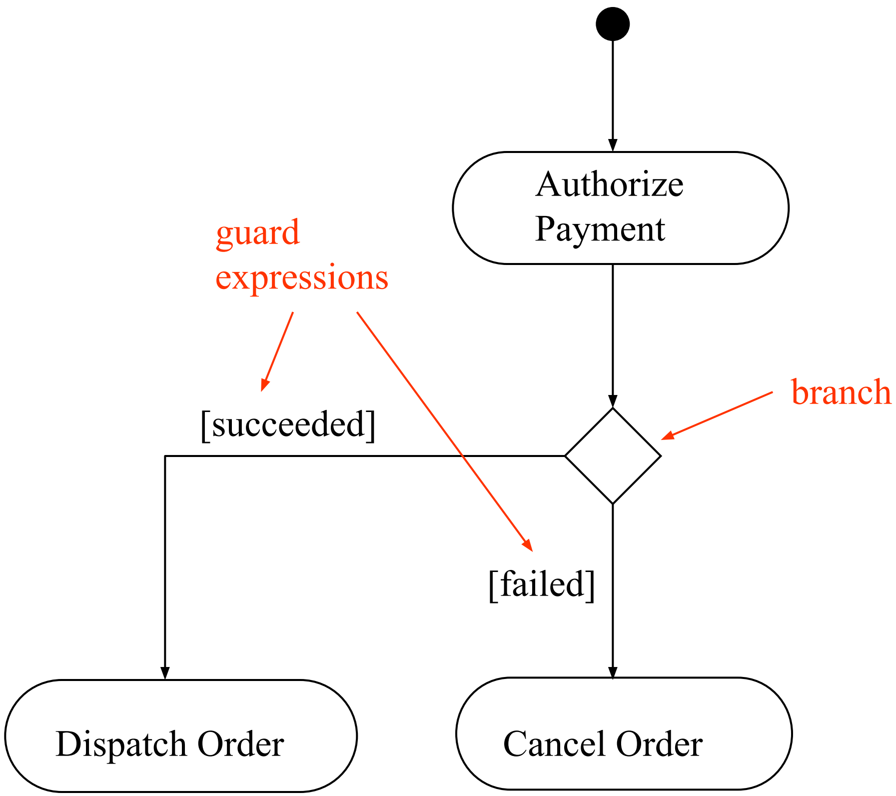
Activity Diagrams

- When an activity or action is completed the control passes immediately to the next action or activity
- Transitions can have guard conditions
- Multiple trigger symbol * is used to show iteration



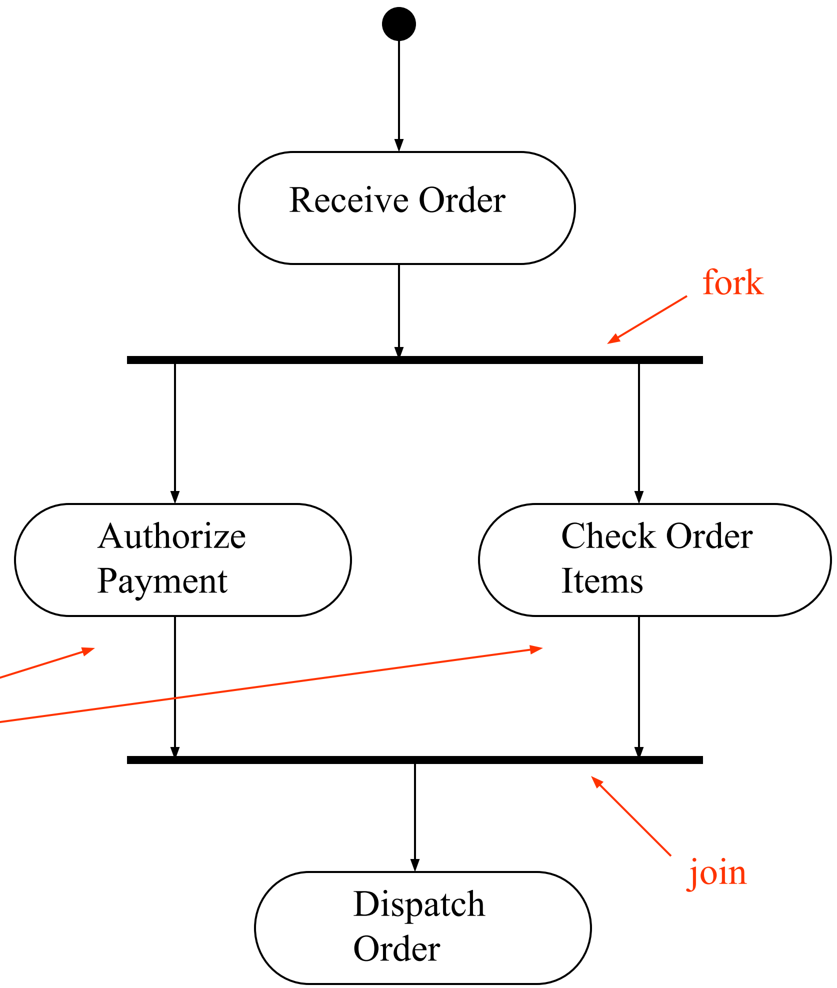
Activity Diagrams: Branches

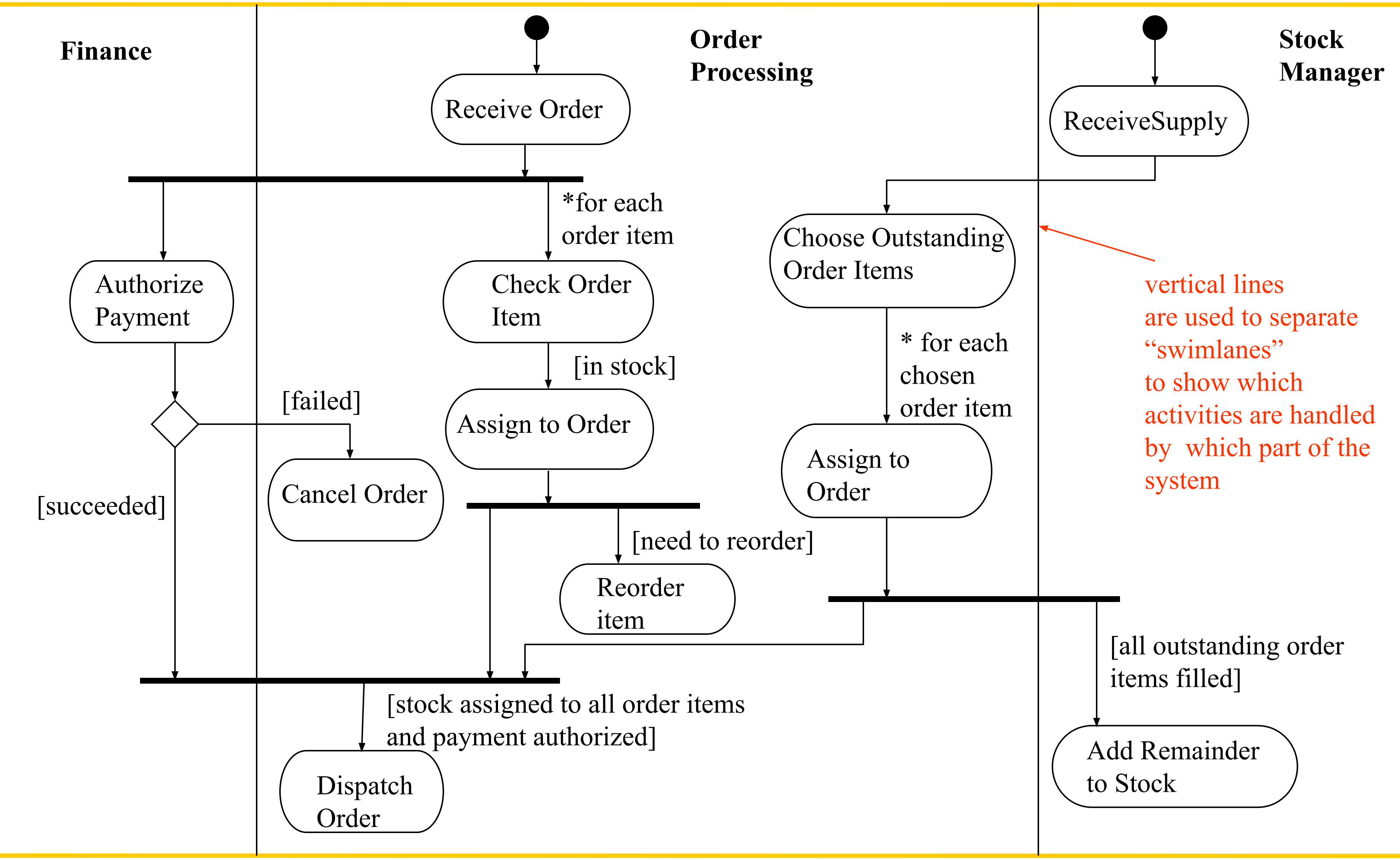
- Conditional branches
  - correspond to if-then-else or switch statements at the implementation level
- a branch is shown as a diamond
- a branch can have one incoming transition and two or more outgoing
- the guard conditions on different outgoing transitions should not overlap to prevent nondeterminism
- guard conditions on different outgoing transitions should cover all the possibilities so that the control flow does not get stuck at the branch



Activity Diagrams: Forks and Joins

- Forks and joins are used to model concurrent execution paths
- They can be used to express parallelism and synchronization
  - forks create concurrent threads
  - joins merge different threads

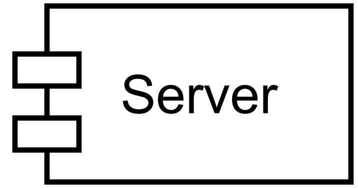


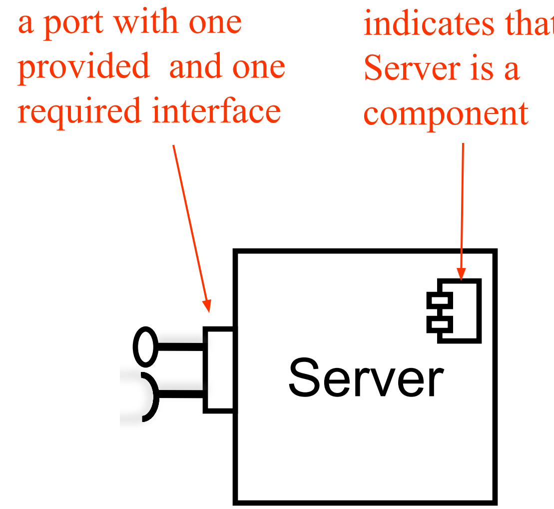

Finance
Order Processing
Stock Manager
Receive Order
*for each order item
Authorize Payment
Check Order Item
[in stock]
[failed]
Assign to Order
Cancel Order
[succeeded]
[need to reorder]
Reorder item
[stock assigned to all order items and payment authorized]
Dispatch Order
ReceiveSupply
Choose Outstanding Order Items
* for each chosen order item
Assign to Order
[all outstanding order items filled]
Add Remainder to Stock
vertical lines are used to separate "swimlanes" to show which activities are handled by which part of the system

Architecture Specification

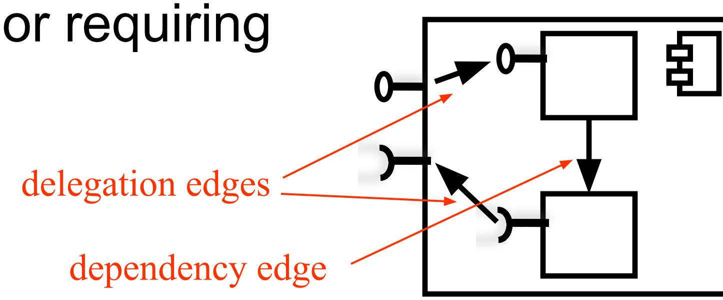
- The basic concepts in specification of software architecture are:
  - **Components:** Components represent either major computational elements or data stores. They are usually represented with boxes in visual representation of architectures.
  - **Connectors:** Connectors represent interactions among components. They are usually represented as lines in visual representations
- Most architecture specification languages support
  - **hierarchical specification** where one component can contain a sub-architecture of components
  - specification of **component interfaces**
  - **connectors** that connect component interfaces
  - both graphical and textual specification of systems that consist of components and connectors
  - specification of additional constraints on components and connectors

UML Component Diagrams

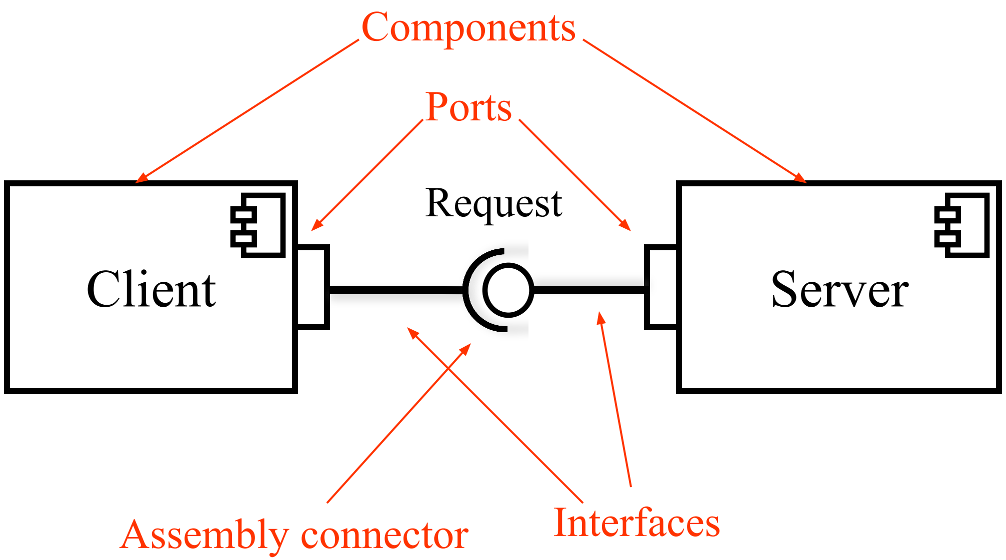




- Components
- Interfaces
  - Provided interfaces 
  - Required interfaces 
- Ports
  - Used to group interfaces
- Assembly connector 
  - Connects a required interface to a provided interface
- Delegation connector 
  - Used for showing internal structure of a component. Connects the handling part to a provided interface or requiring part to a required interface
- Dependencies 
  - Show dependencies



Client Server Architecture



In addition to this diagram, the architecture specification should

- Explain the basic functionality of the Server and the Client
  - What do they do? What do they compute? What do they store?
- Explain the type of the connector
  - Is it an RPC connection or is it a socket connection, etc.?
- Explain the contents of the data (messages) exchanged between the Client and the Server

UML Diagrams

- Functionality, requirements
  - use case diagrams
- Architecture, modularization, decomposition
  - class diagrams (class structure)
  - component diagrams, package diagrams, deployment diagrams (architecture)
- Behavior
  - state diagrams, activity diagrams
- Communication, interaction
  - sequence diagrams, collaboration diagrams

How do they all fit together?

- Requirements analysis and specification
  - use-cases, use-case diagrams, sequence diagrams
- Design and Implementation
  - Component diagrams, package diagrams and deployment diagrams can be used to show the high level architecture
  - Class diagrams can be used for showing the decomposition of the design
  - Activity diagrams can be used to specify behaviors described in use cases
  - State diagrams are used to specify behavior of individual objects
  - Sequence and collaboration diagrams are used to show interaction among different objects
  - Use cases and sequence diagrams can be used to derive test cases