# Software Design & Architecture

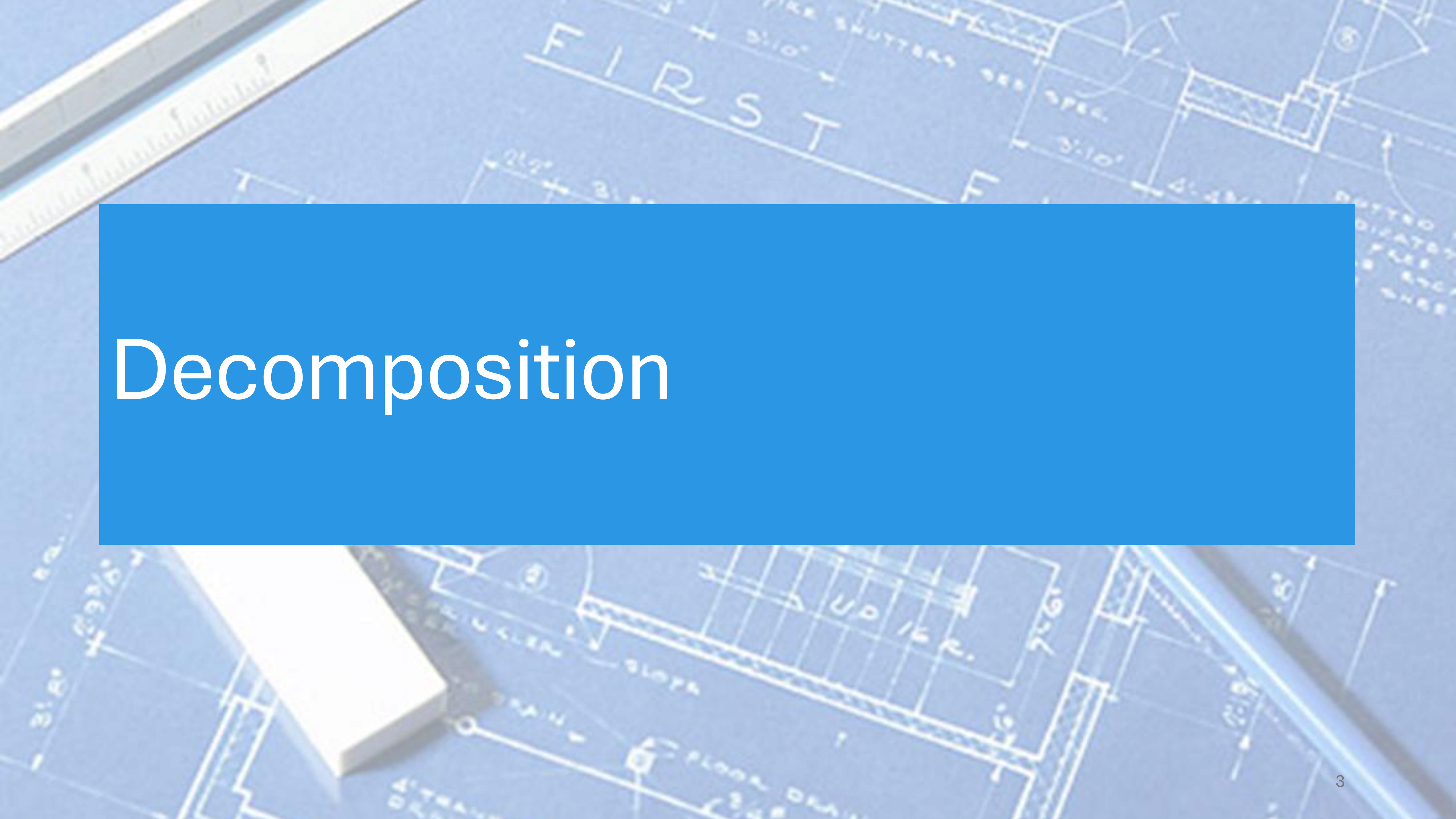
# Decomposition & Architectural Views

Pengyu Nie

Acknowledgements: slides adapted from previous versions by Mei Nagappan and Shane McIntosh, which are adapted from previous versions by Zhen Ming Jiang, Ahmed E. Hassan, Reid Holmes.

# Agenda

- Decomposition
  - key definitions in architecture
  - principles
- Architectural views
  - more UML diagrams



# Decomposition

# What is Software Architecture

- “Architecture is the fundamental organization of a system, embodied in its components, their relationships to each other and the environment, and the principles govering its design and evolution”

-- ANSI/IEEE 1471-200

- Working definition: the set of principal design decisions about the system
- Architectures capture three primary dimensions:
  - Structure: what are the subsystems and components?
  - Communication / Behaviour: how do they interact?
  - Non-functional requirements

# Subsystems

- Definition: architectural entity that
  - encapsulates a subset of functionality
  - restricts access via explicit interface
  - has explicit environmental dependencies
- Elements that encapsulate processing and data at an architectural level
- ➡️ project/subproject, group of packages/modules

# Components

- Definition: architectural/design entity that
  - encapsulates a smaller subset of functionality
  - restricts access via explicit interface
- Elements from which subsystems are composed
- ➡️ package/module, group of classes/files

# Connectors

- Definition: architectural entity tasked with effecting and regulating interactions between subsystems
- Application-independent interaction mechanisms
- Describing connectors can be more challenging than subsystems in large heterogenous systems
- ➡️ method call, RPC (remote procedure call), shared memory, network call, streaming connection, etc.

# Configuration/Topology

- Definition: a set of specific associations between the subsystems and the connectors of the system’s architecture
- Bind subsystems and connectors together in a specific way

# Decomposition

- Top-down abstraction
  - focus on the key issues while removing extraneous detail
  - break problem into independent subsystems
  - describe each subsystem
- A good decomposition should make typical cases simple, and exceptional cases possible
- Criteria for decomposition can include
  - implementation teams
  - application domains (aka obvious patitions)
  - parallelization

# Coupling and Cohesion

- Minimize coupling between subsystems
  - the less that subsystems know about each other, the better
  - make future change easier (maintenability)
- Maximize cohesion within each subsystem
  - one subsystem should be responsible for one logical service
  - components of each subsystem are strongly inter-related (they really do belong together)

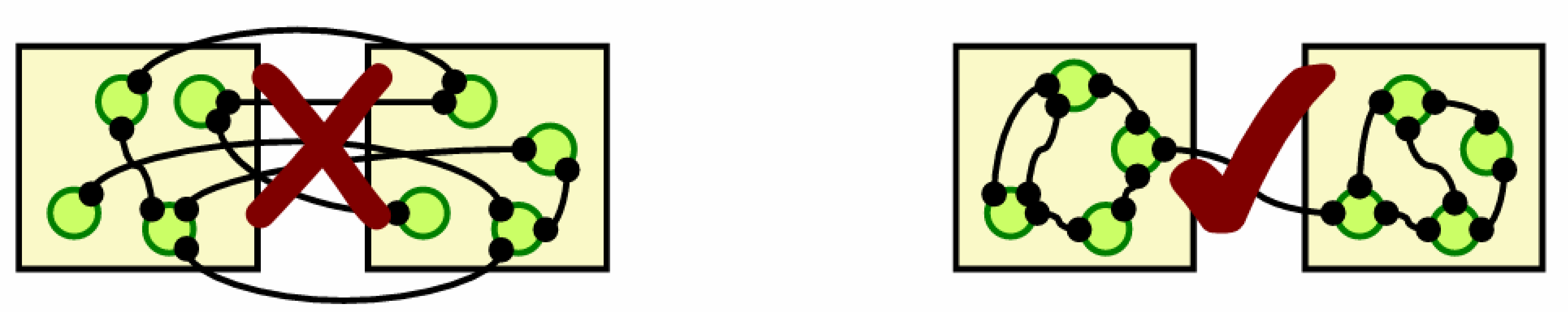


figure source: https://www.cs.toronto.edu/~sme/CSC302/notes/04-package-diagrams.pdf

# Conway’s Law

- The structure of a software system reflects the structure of the organization that built it

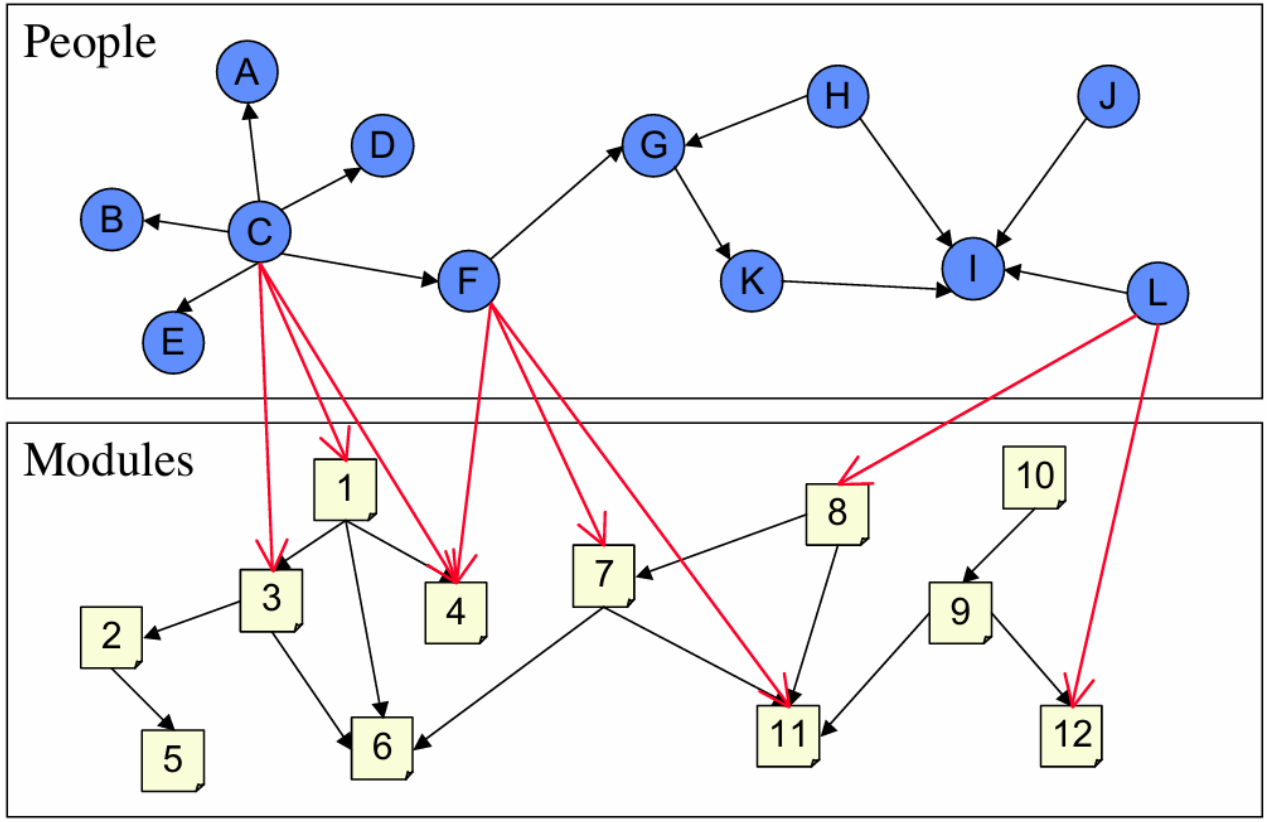


figure source: https://www.cs.toronto.edu/~sme/CSC302/notes/04-package-diagrams.pdf

# Law of Demeter / Principle of Least Knowledge

- Each unit should have only limited knowledge about other units: only units "closely" related to the current unit
- Each unit should only talk to its friends; don't talk to strangers
- Only talk to your immediate friends

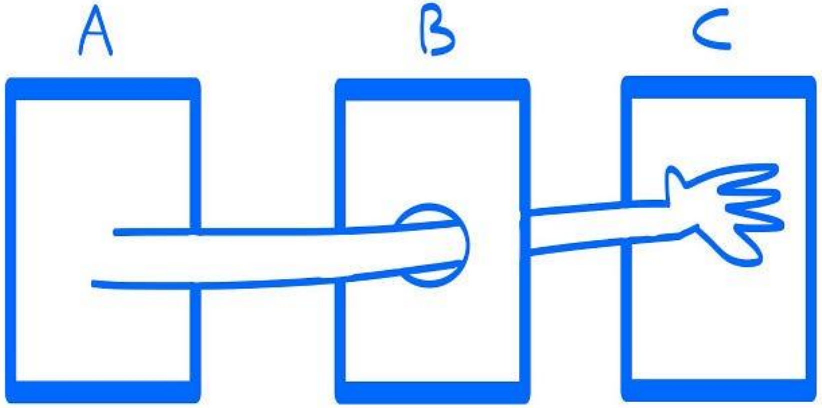


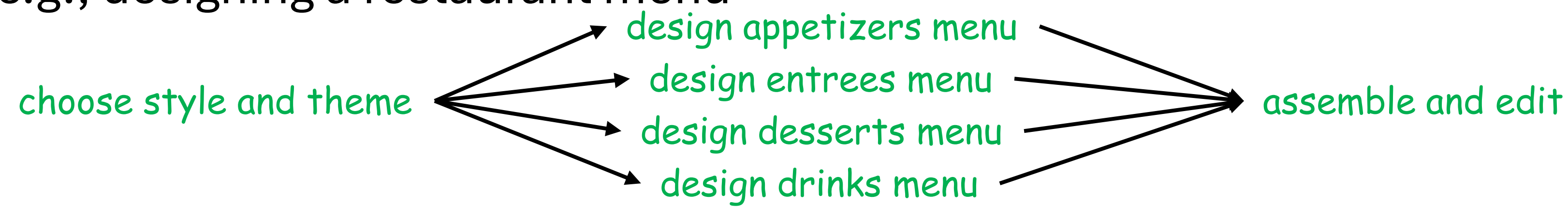
figure source: https://levelup.gitconnected.com/the-law-of-demeter-4bd40aa21cbe

# SOLID Principles

- Single responsibility principle
  - There should never be more than one reason for a class to change
- Open-closed principle
  - Software entities should be open for extension but closed for modification
- Liskov substitution principle
  - Functions that use pointers or references to base classes must be able to use objects of derived classes without knowing it
- Interface segregation principle
  - Clients should not be forced to depend upon interfaces that they do not use
- Dependency inversion principle
  - Depend upon abstractions, not concretes

# Decomposition isn’t always great

- Decomposition can work well
  - e.g., designing a restaurant menu



choose style and theme

design appetizers menu

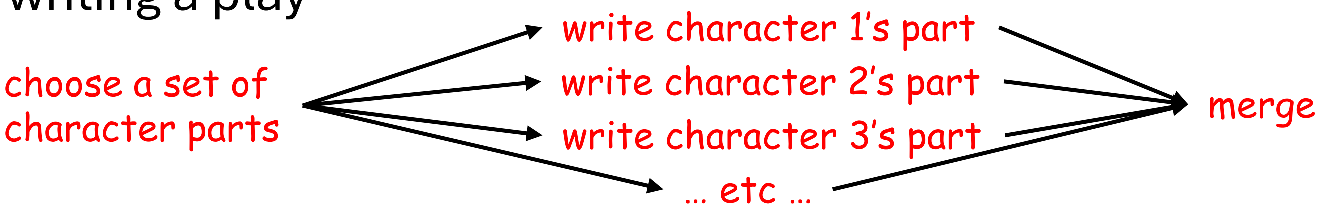
design entrees menu

design desserts menu

design drinks menu

assemble and edit

- Decomposition doesn’t always work
  - e.g., writing a play



choose a set of character parts

write character 1’s part

write character 2’s part

write character 3’s part

… etc …

merge

- Decomposition isn’t always possiple
  - for very complex problems (e.g., managing the economy)
  - for impossible problems (e.g., turning water into wine)
  - for atomic problems (e.g., adding 1 and 1)

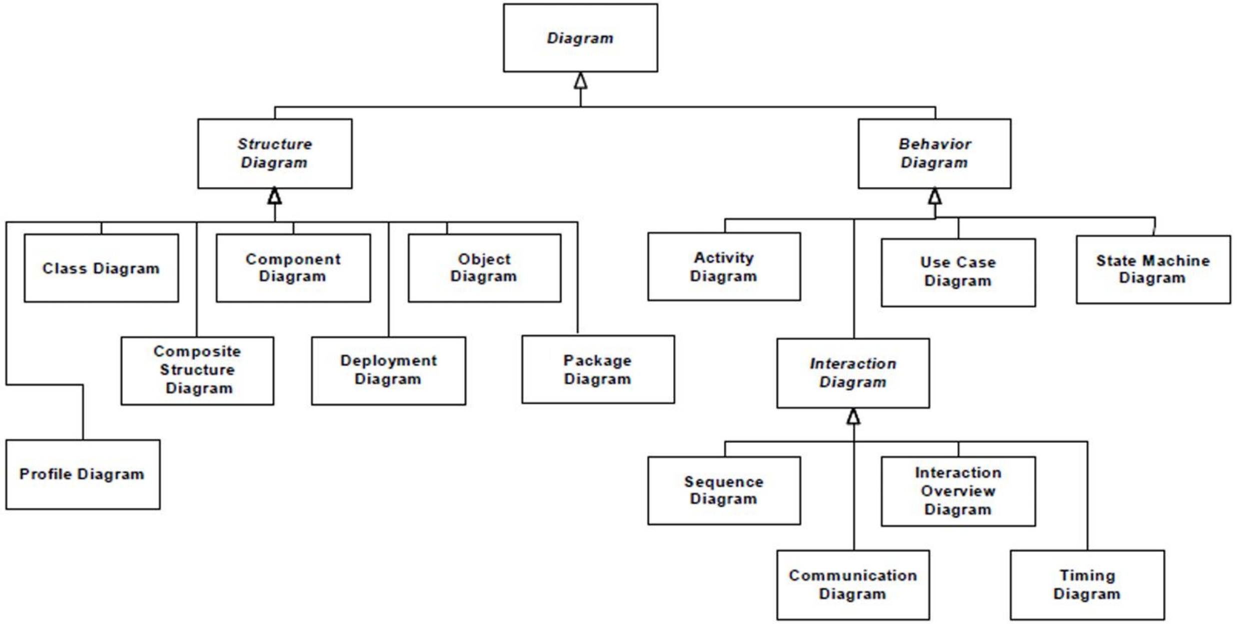
source: https://www.cs.toronto.edu/~sme/CSC444F/slides/L05-DecompositionAbstraction.pdf

# Architectural Views

Acknowledgements: UML examples from UML Distilled – Applying the Standard Object Modeling Language by Martin Fowler and Kendall Scott

# Architectural Views

- Architectural models can be overwhelming
  - different views focus on specific subsets of elements or subsets of relationships
  - views often focus on specific concerns or scenarios within a system
- Views overlap
  - maintaining consistency between views is challenging



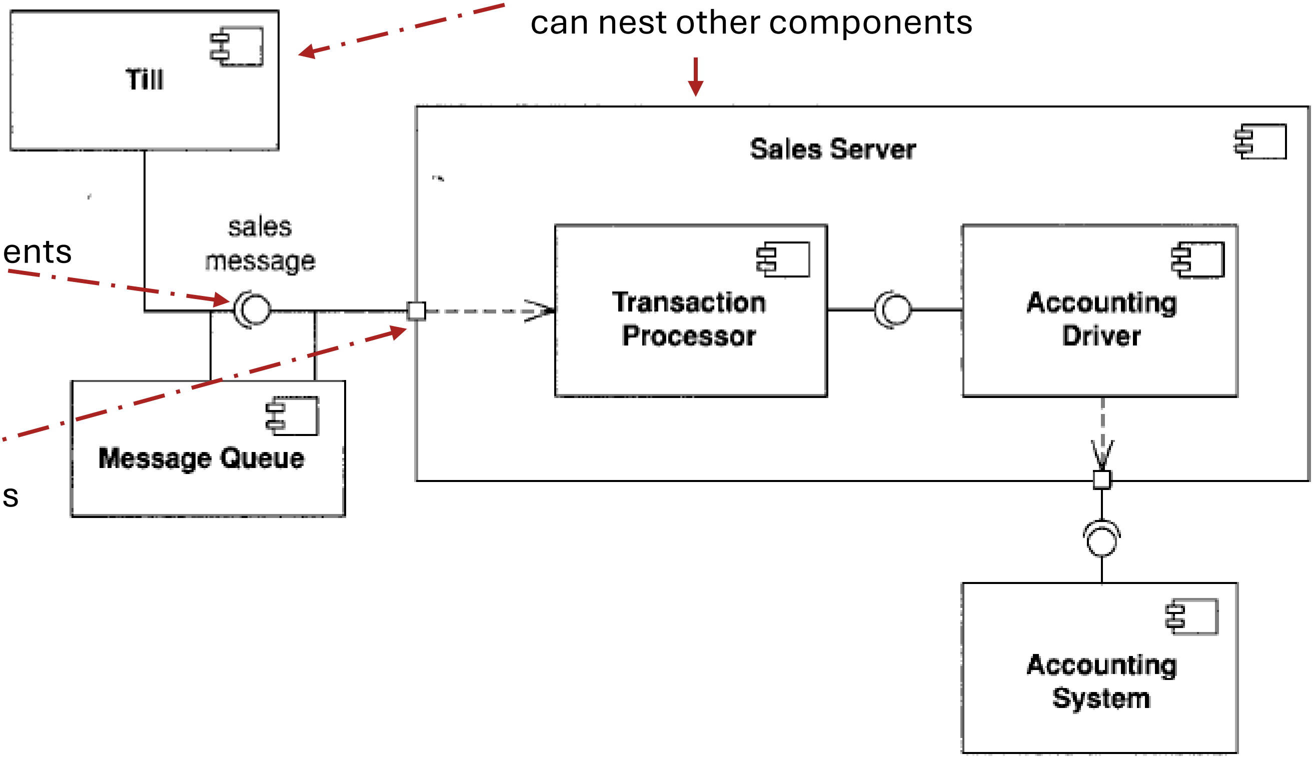
# Component Diagram

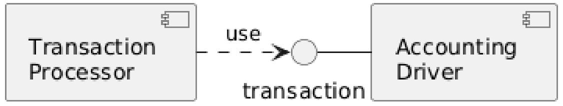
- Shows the organization and dependencies between subsystems/components

*component*
rectangle with the top-right “component” symbol
can nest other components

*interface*
connectors between two components
lollipop: provide an interface
socket: use an interface

*port*
group required/provided interfaces

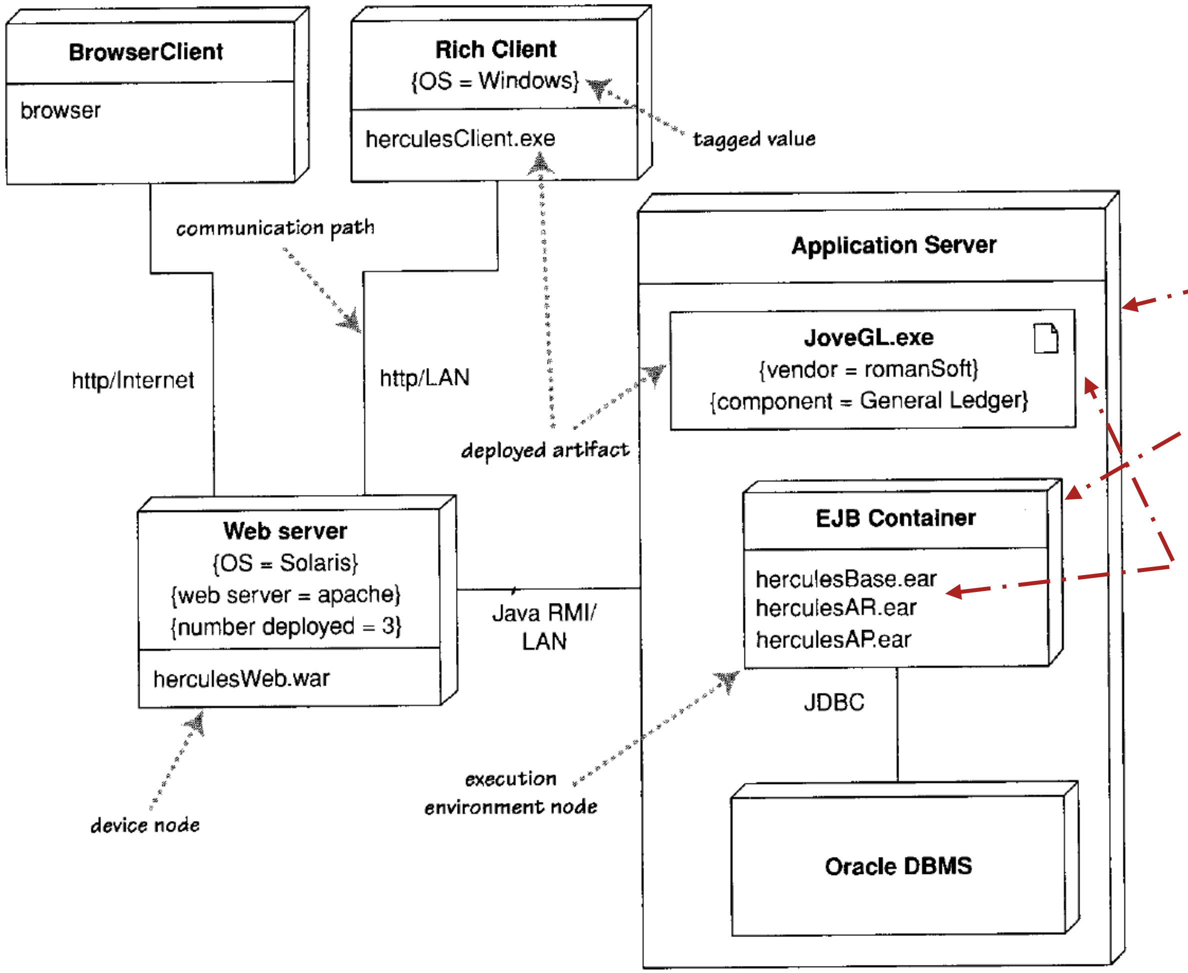




an alternative way to
represent interface
use and provide

# Deployment Diagram

- Shows a system’s physical layout



*node*

something that can host some software

can be device or execution environment

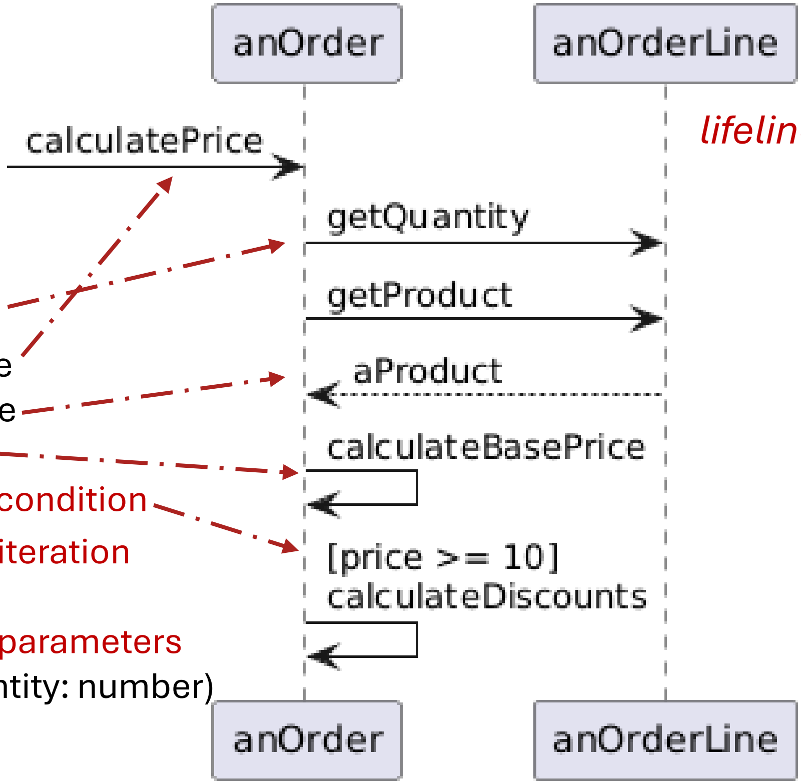
*artifact*

physical manifestations of software

executables, data files, configuration files, etc.

# Sequence Diagram

- Shows the interaction between objects, emphasizing the time ordering of messages



*participant/object*

*lifeline*

*message*

found message

return message

self message

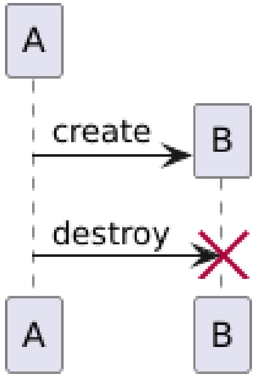
message with condition

message with iteration

* [i = 1..N]

message with parameters

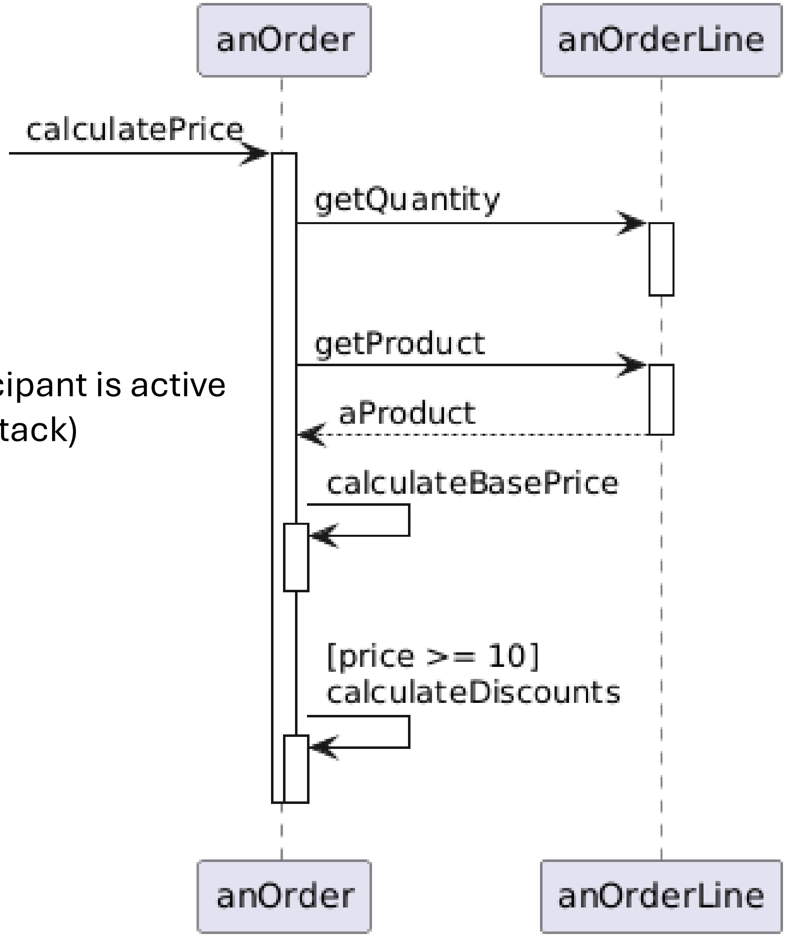
getPrice(quantity: number)



*creation*
and
*destroy*

# Sequence Diagram (cont.)

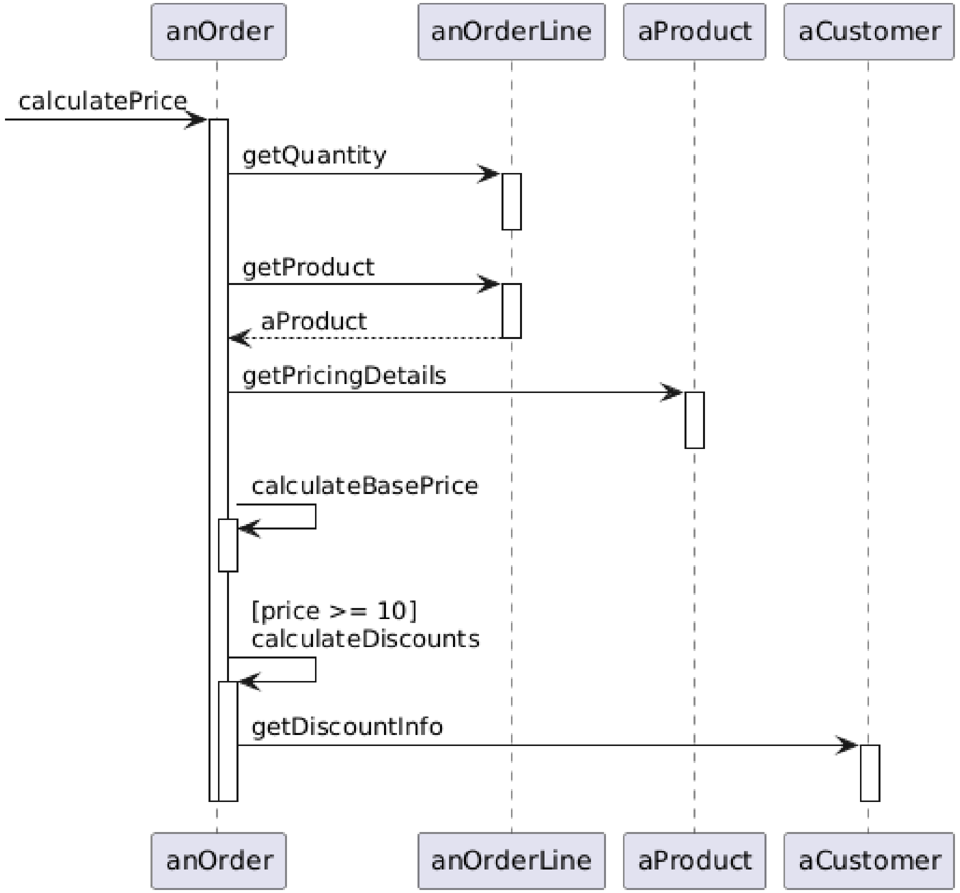
- Shows the interaction between objects, emphasizing the time ordering of messages



*activation bar*
shows when each participant is active
(the function call is on stack)

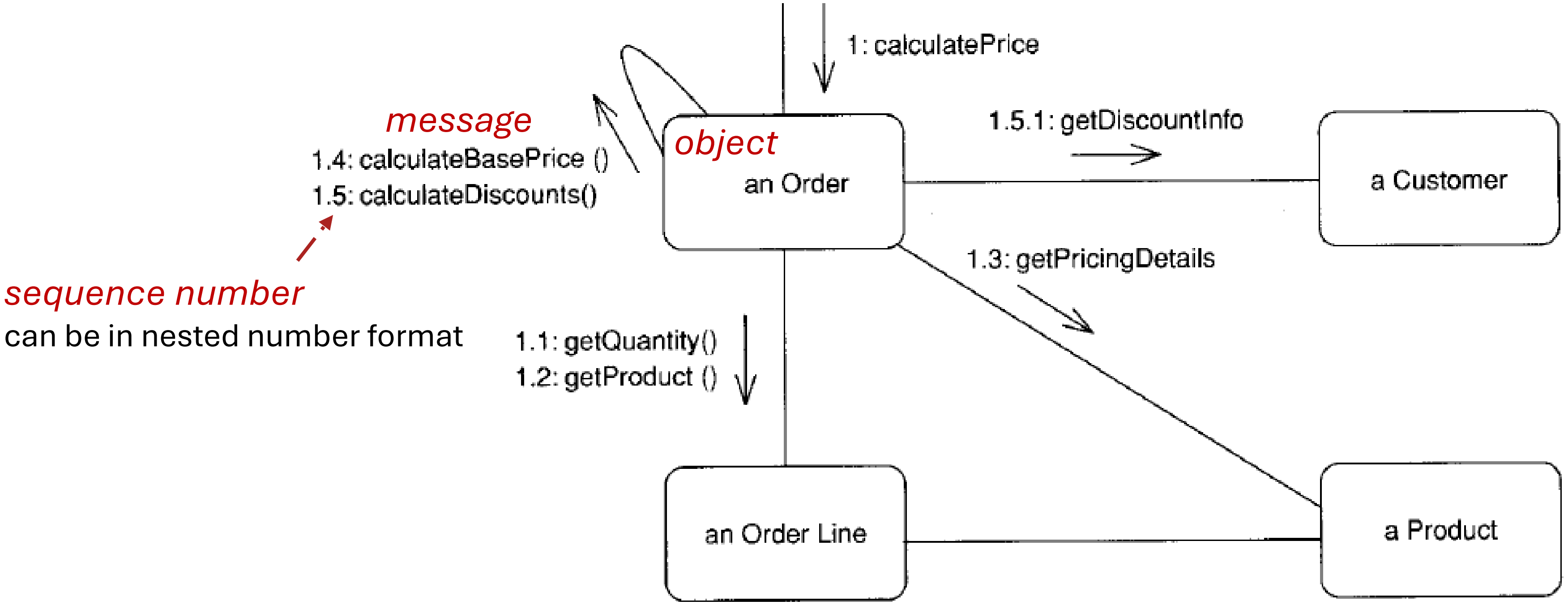
# Sequence Diagram (cont.)

- Shows the interaction between objects, emphasizing the time ordering of messages



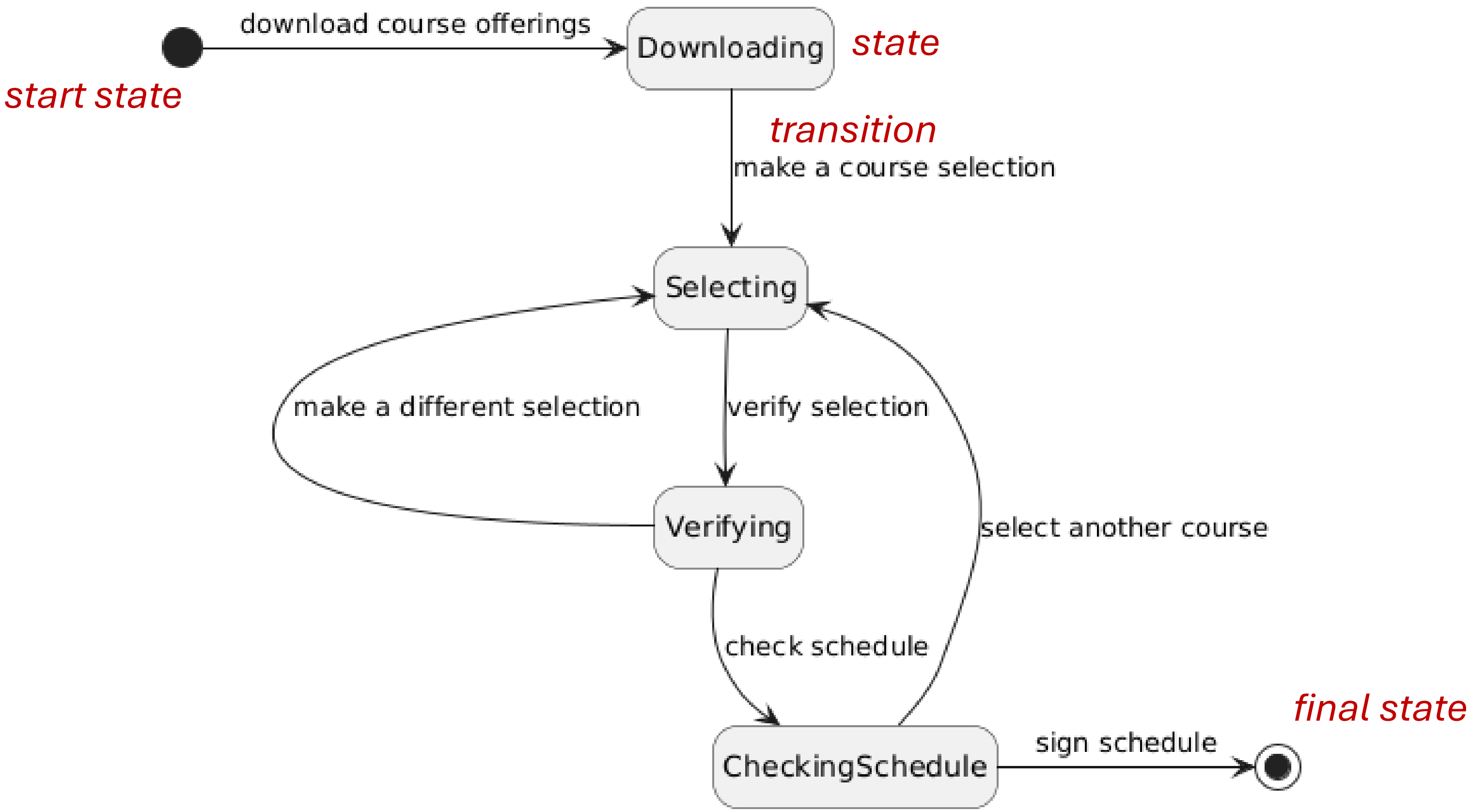
# Communication Diagram

- Shows the interaction between objects, emphasizing their relationships
- Alternative name: collaboration diagram (in UML v1)



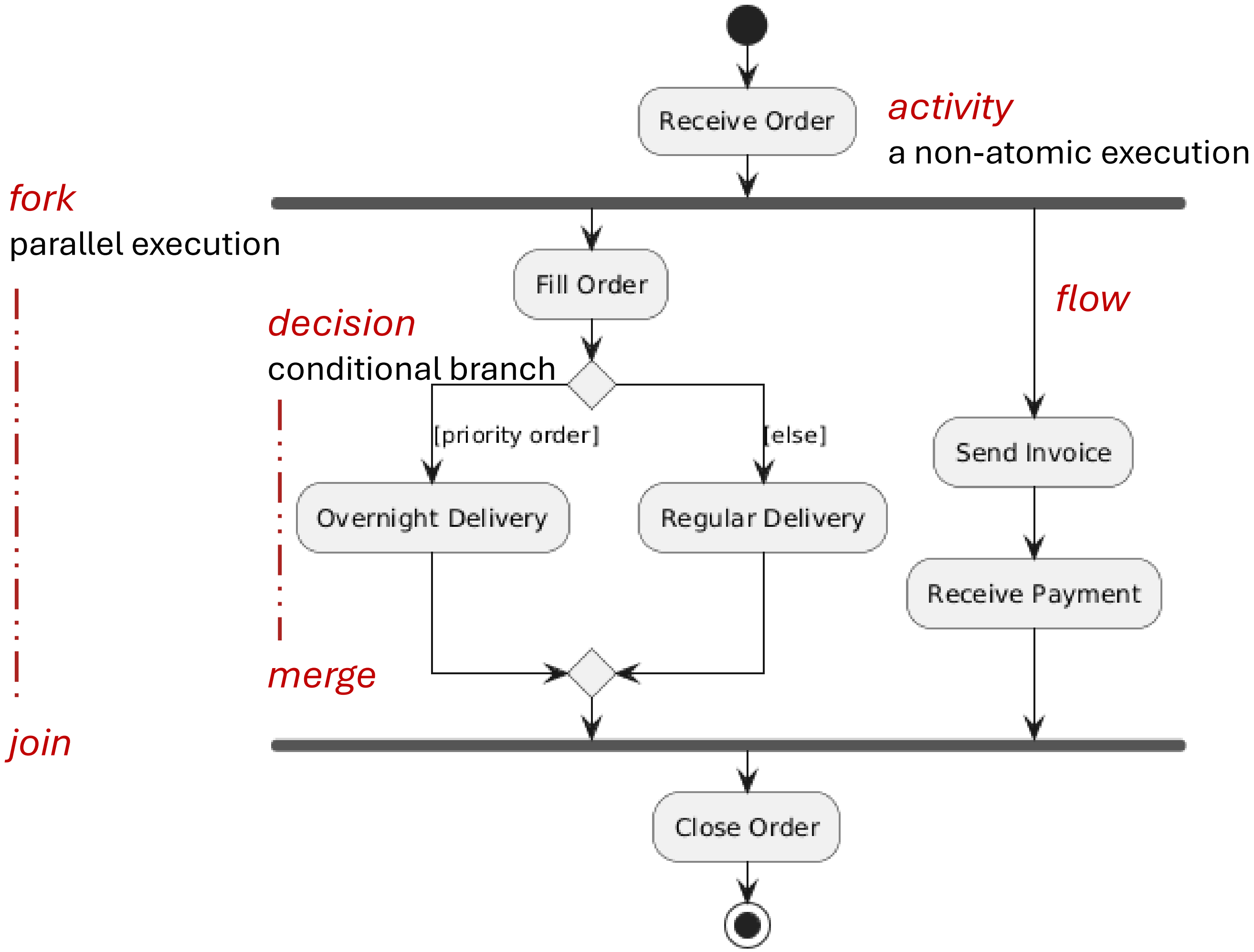
# State Machine Diagram

- Shows the lifecycle of an object, as transitions between states
- Alternative names: state diagram, state machine



# Activity Diagram

- Shows the flow of control (procedural logic) from activity to activity



# Agenda (recap)

- Decomposition
  - key definitions in architecture
  - principles
- Architectural views
  - more UML diagrams
- P1 due this Friday!
  Don’t forget to add `wat-cs446` as collaborator to your repo