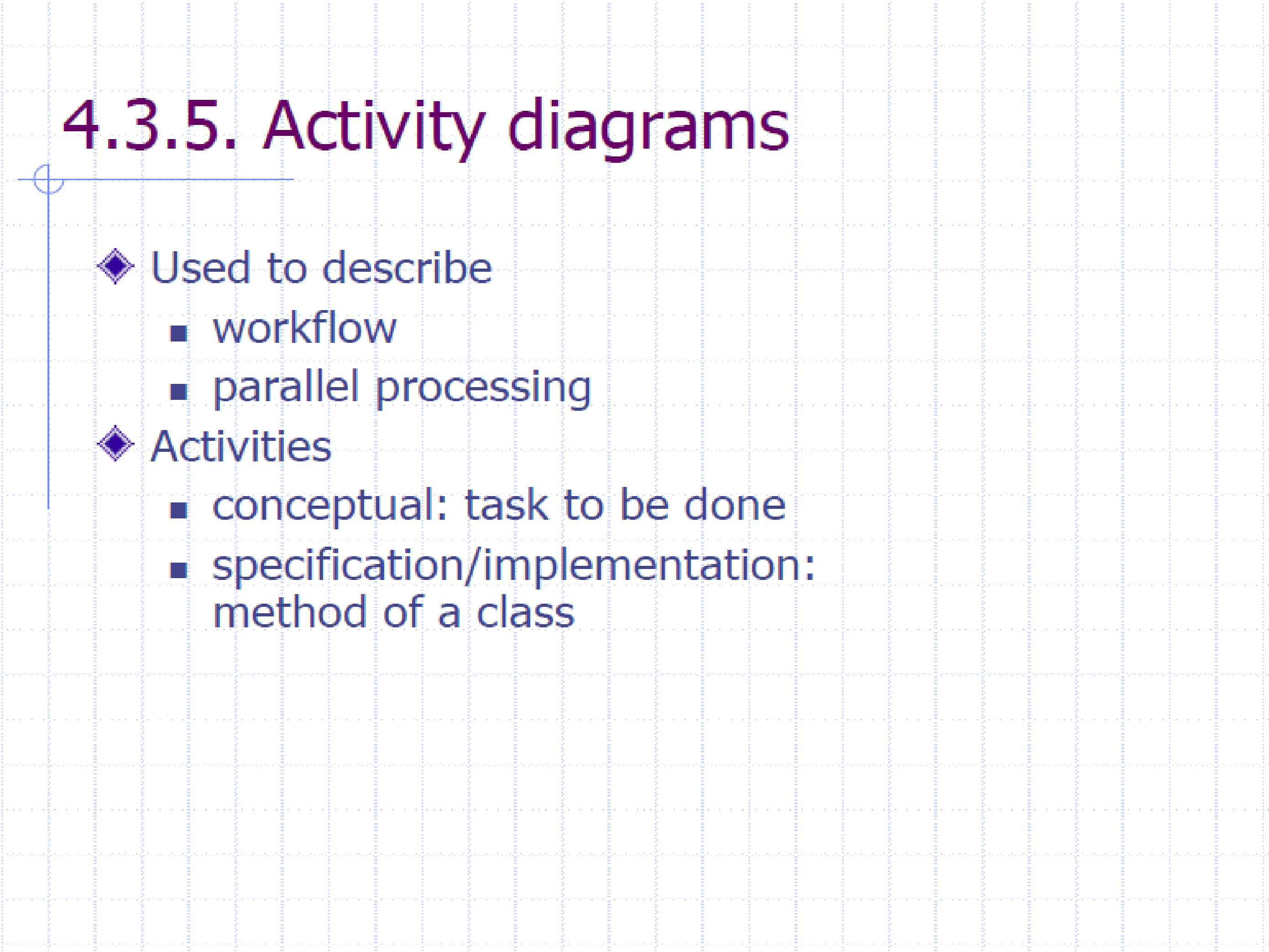
PM tutor

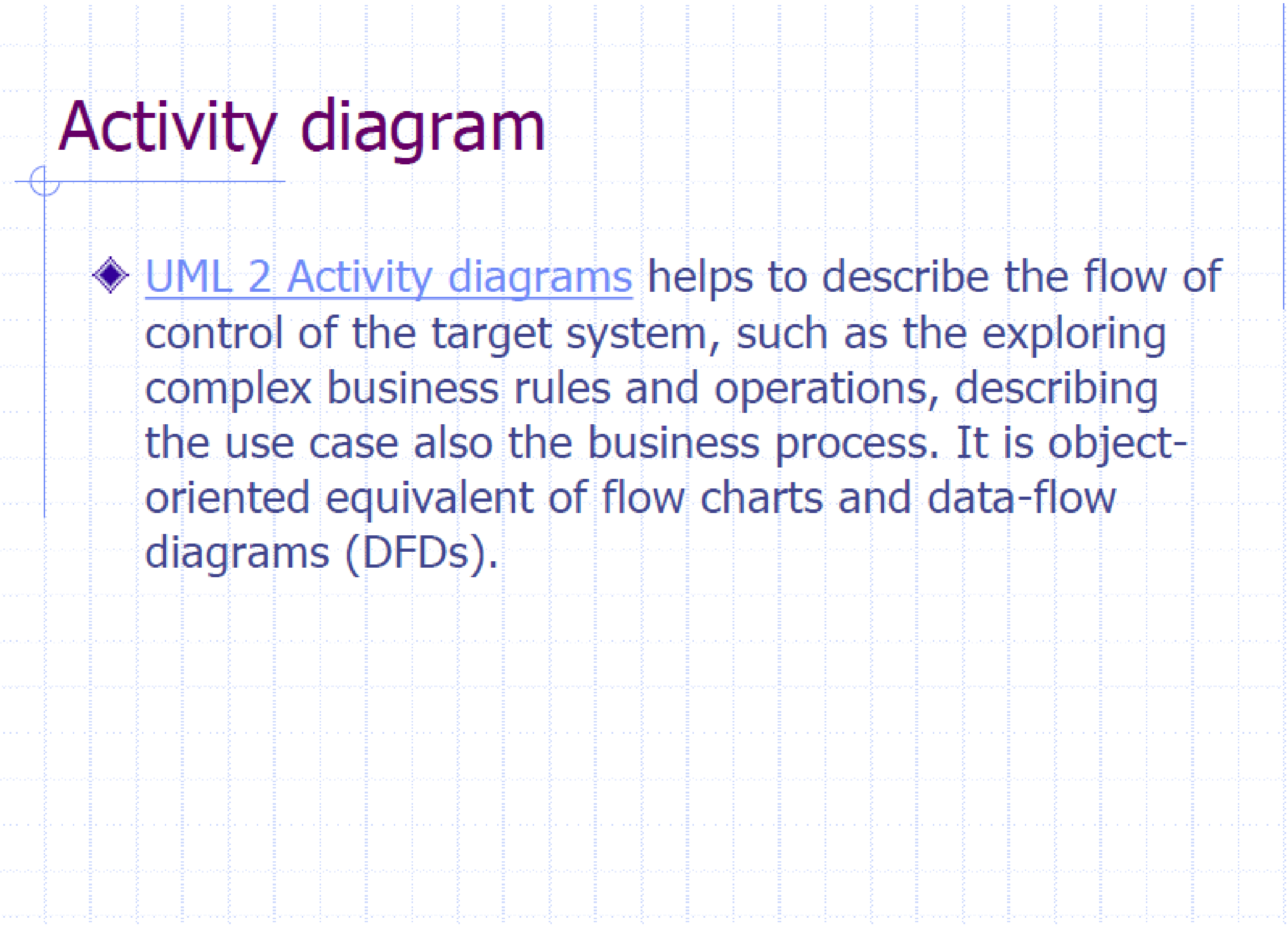
Empowering Excellence

ACTIVITY DIAGRAM



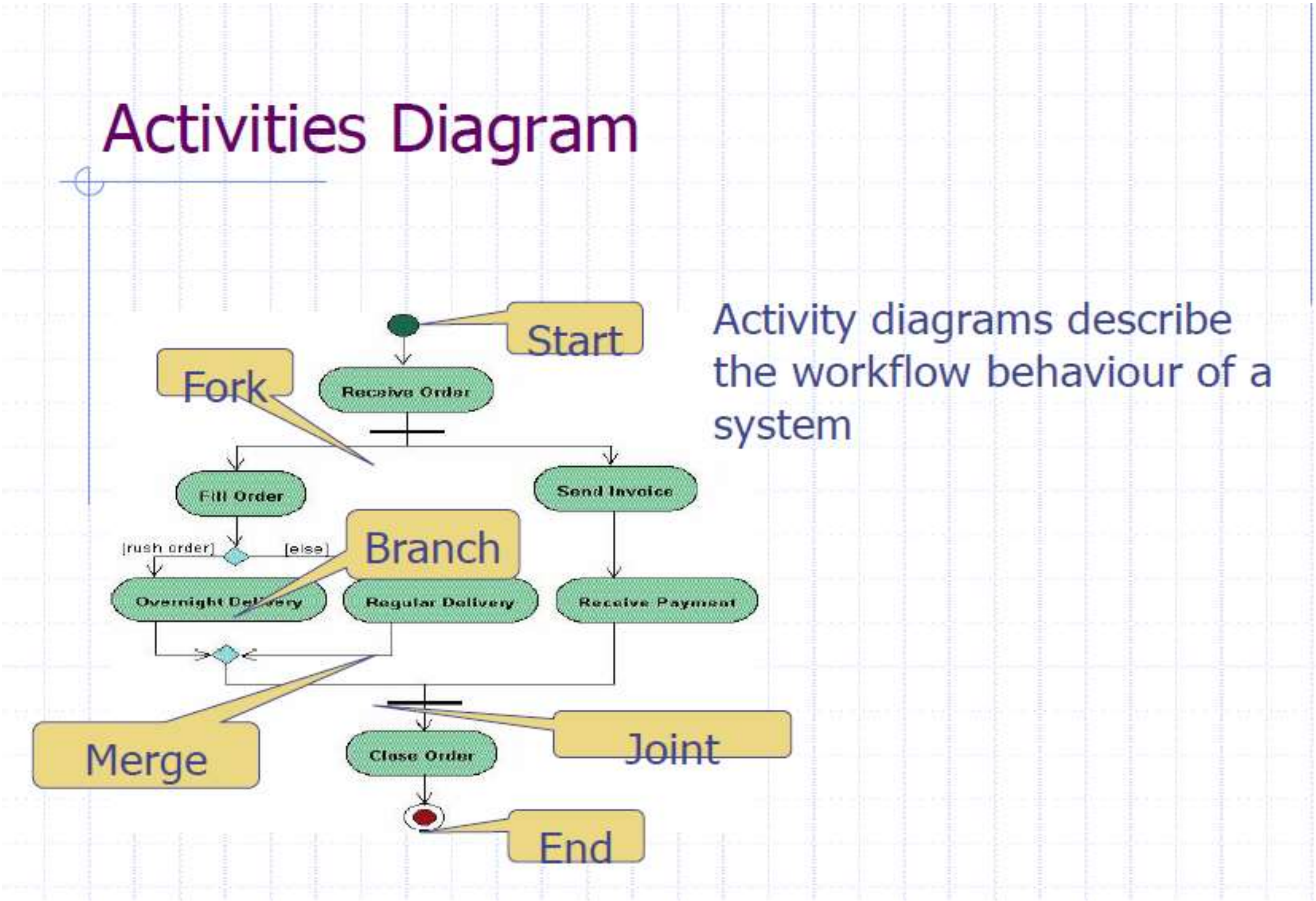
4.3.5. Activity diagrams

- Used to describe
  - workflow
  - parallel processing
- Activities
  - conceptual: task to be done
  - specification/implementation: method of a class



Activity diagram

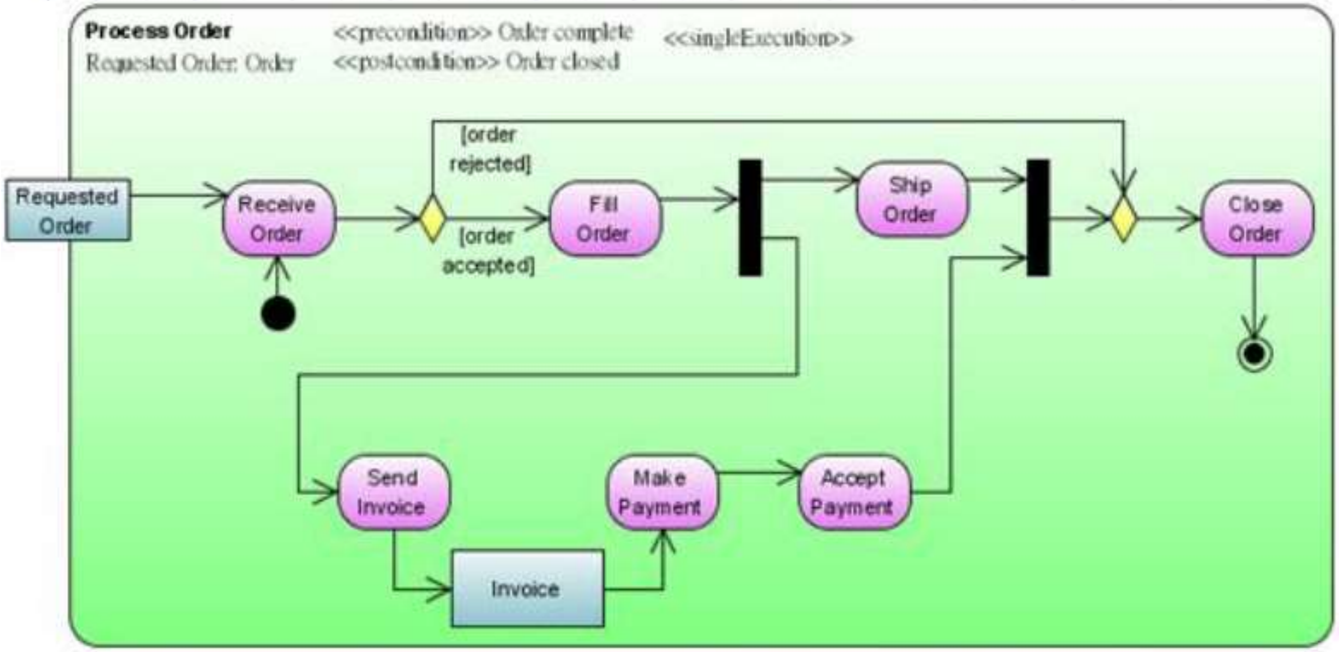
- UML 2 Activity diagrams helps to describe the flow of control of the target system, such as the exploring complex business rules and operations, describing the use case also the business process. It is object-oriented equivalent of flow charts and data-flow diagrams (DFDs).



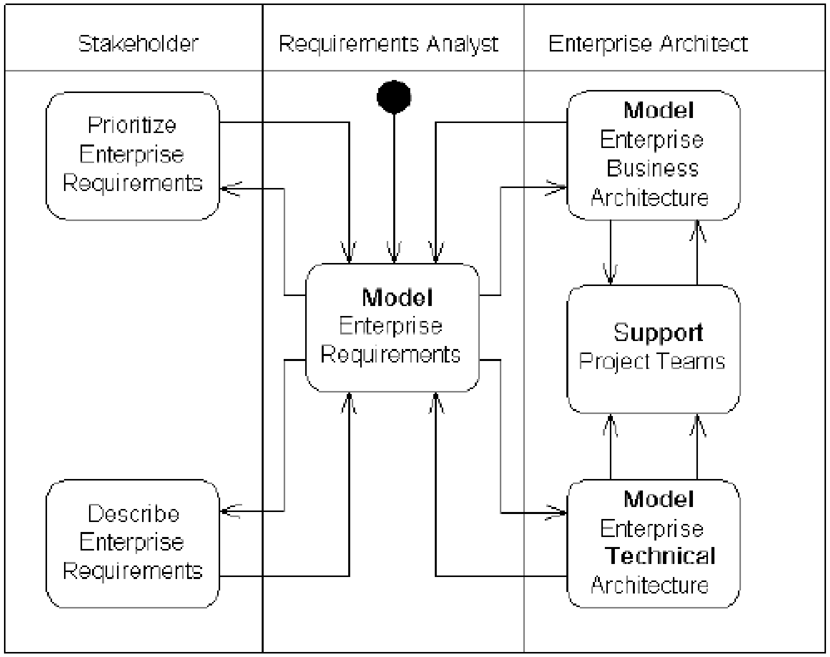
Activities Diagram

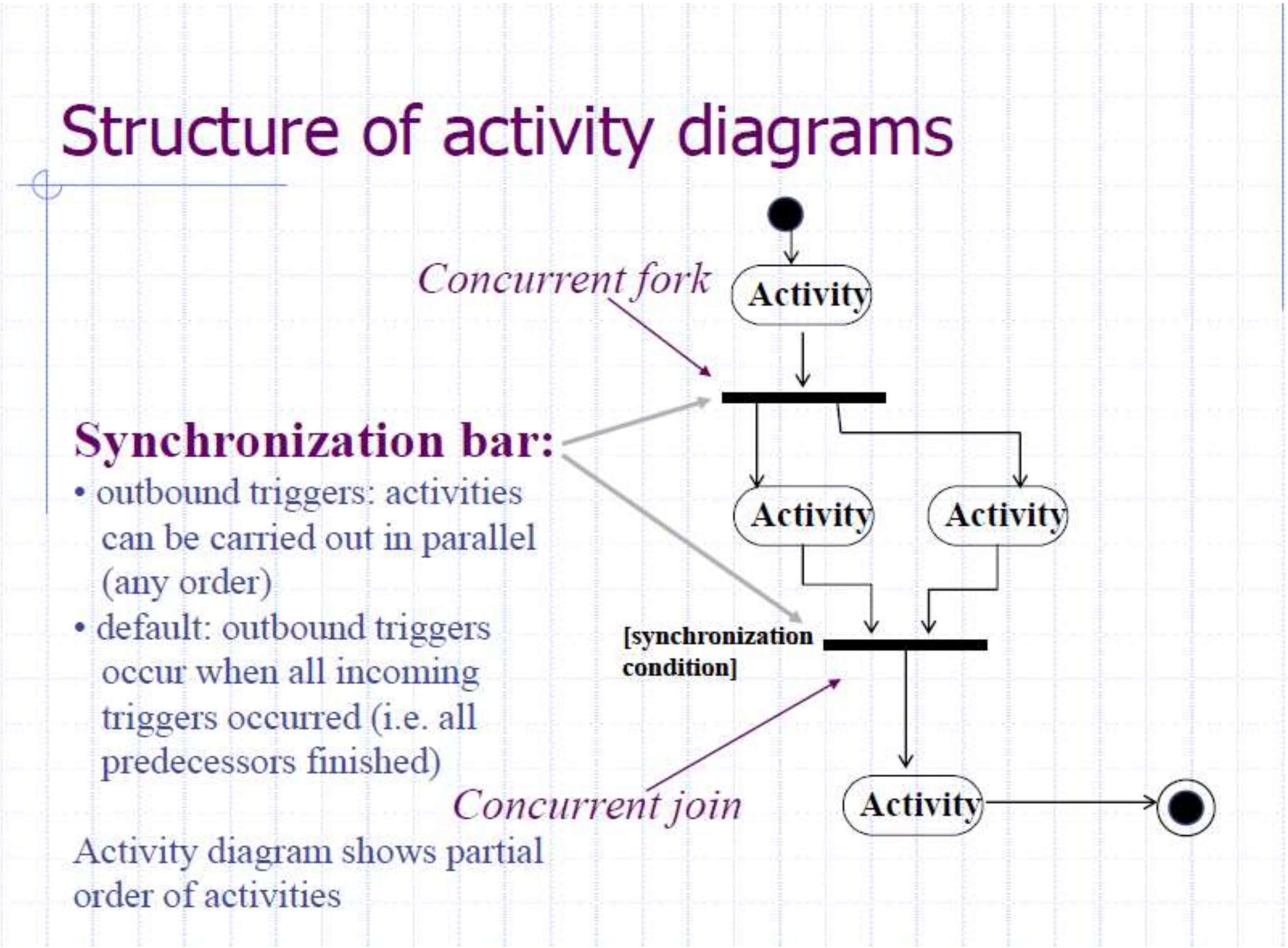
Activity diagrams describe the workflow behaviour of a system

Activity diagram



Activity diagram


Stakeholder
Requirements Analyst
Enterprise Architect
Prioritize Enterprise Requirements
Model Enterprise Requirements
Model Enterprise Business Architecture
Support Project Teams
Describe Enterprise Requirements
Model Enterprise Technical Architecture

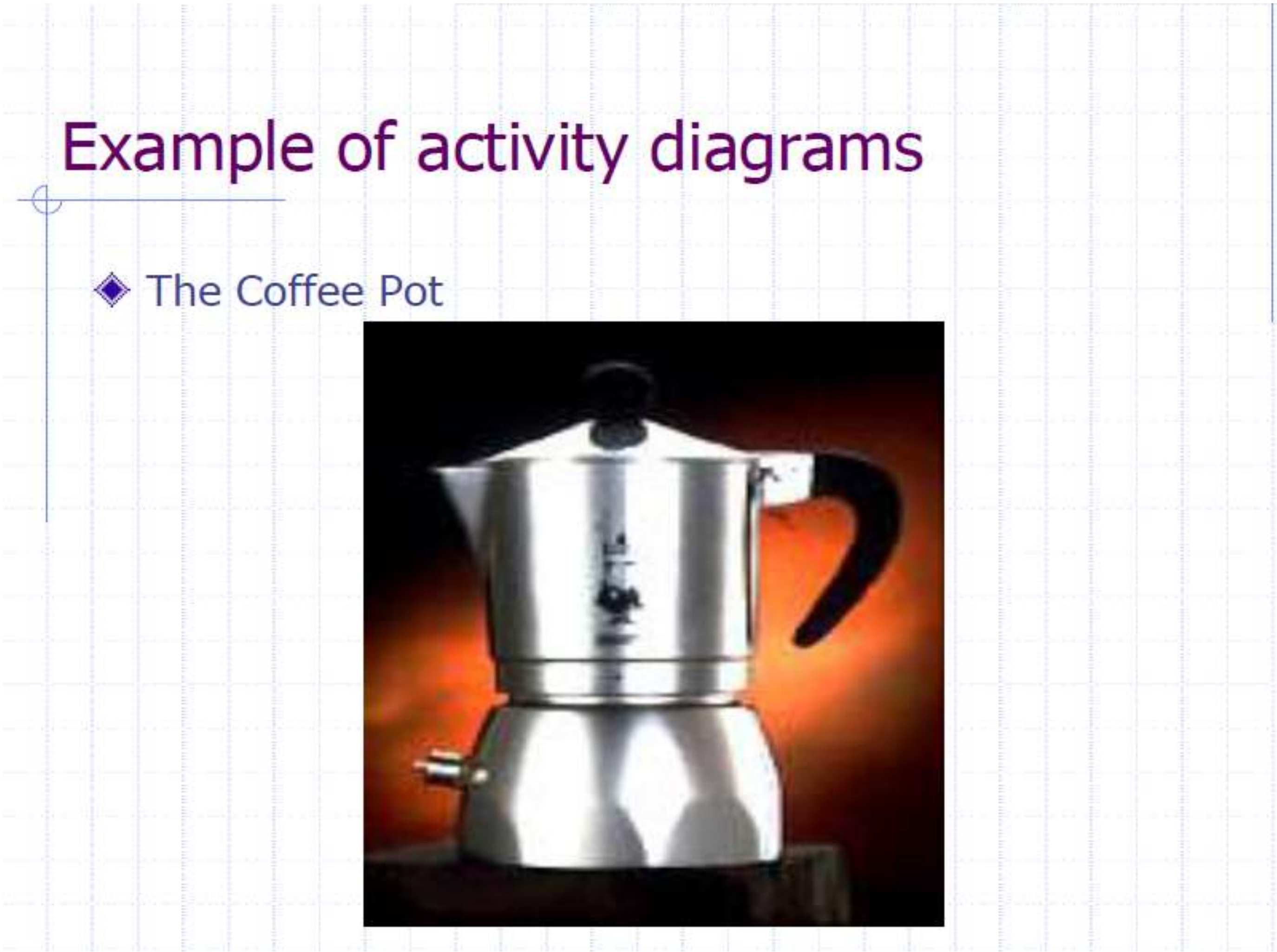


Structure of activity diagrams

Synchronization bar:

- outbound triggers: activities can be carried out in parallel (any order)
- default: outbound triggers occur when all incoming triggers occurred (i.e. all predecessors finished)

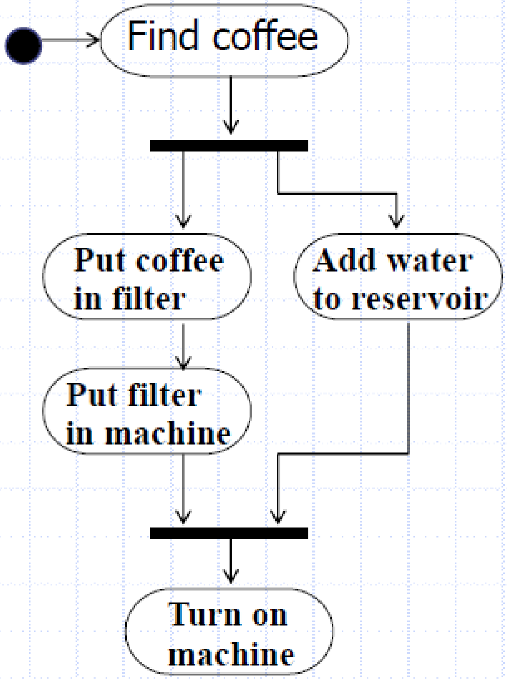
Activity diagram shows partial order of activities



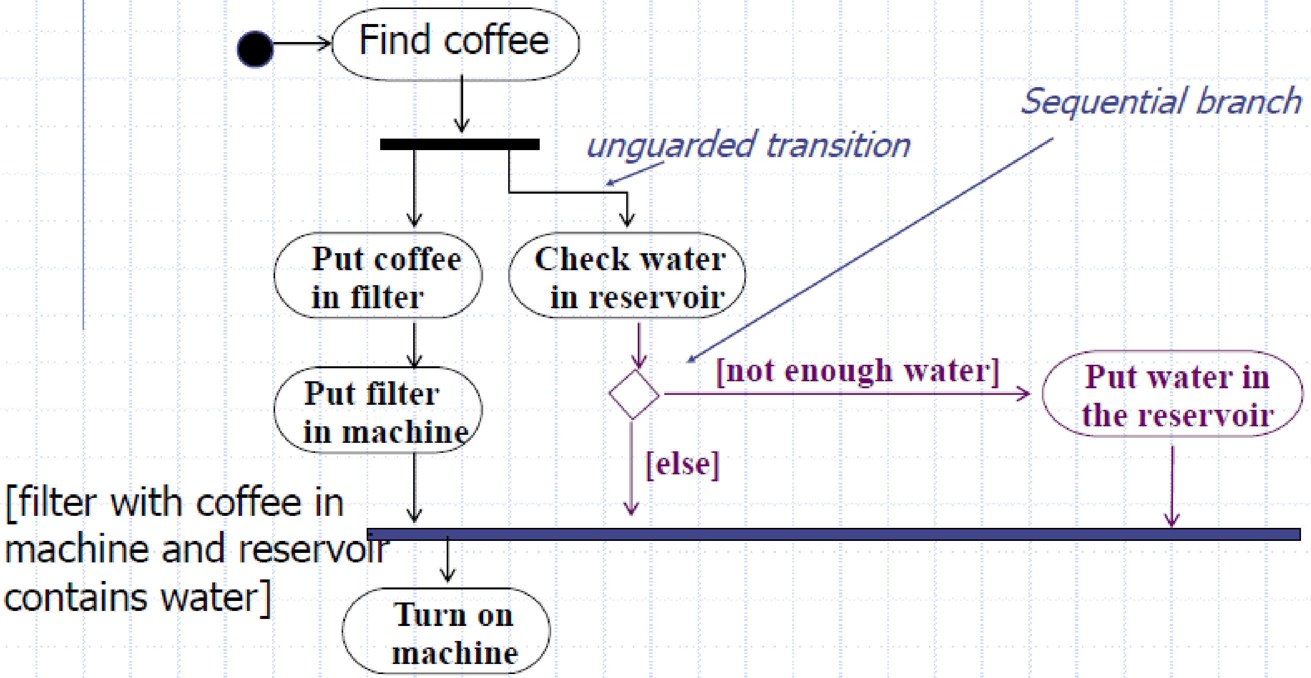
Example of activity diagrams

- The Coffee Pot

Activity diagram



Conditions in activity diagrams



Find coffee

unguarded transition

Sequential branch

Put coffee in filter

Check water in reservoir

Put filter in machine

[not enough water]

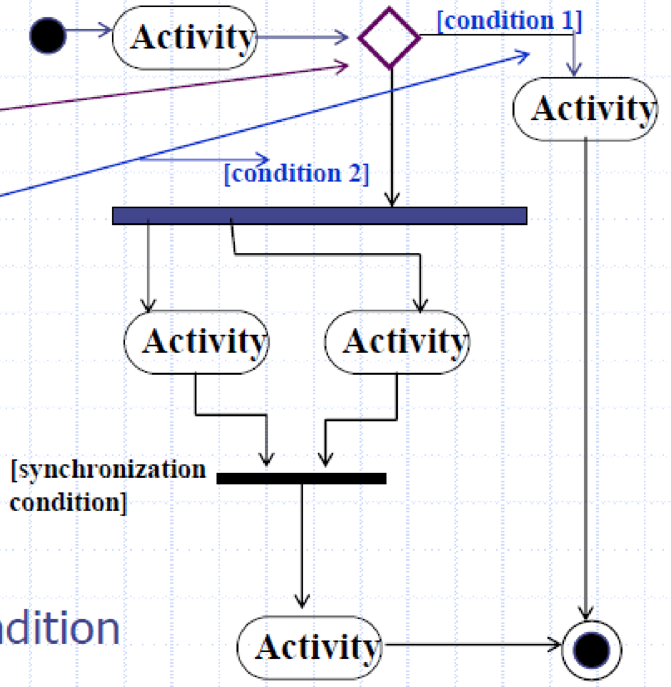
Put water in the reservoir

[else]

[filter with coffee in machine and reservoir contains water]

Turn on machine

Structure of activity diagrams



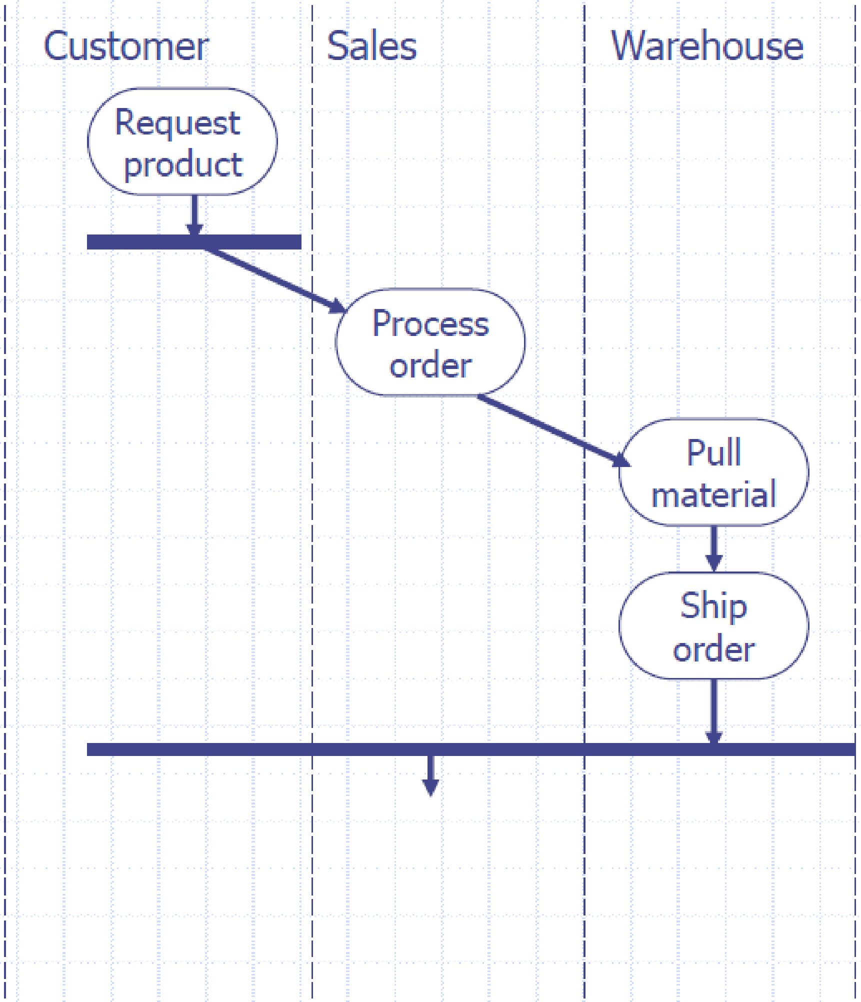
Branch

Guard expression

Strictly UML:
- forks ↔ joins
- branch ↔ merge
- No synchronization condition

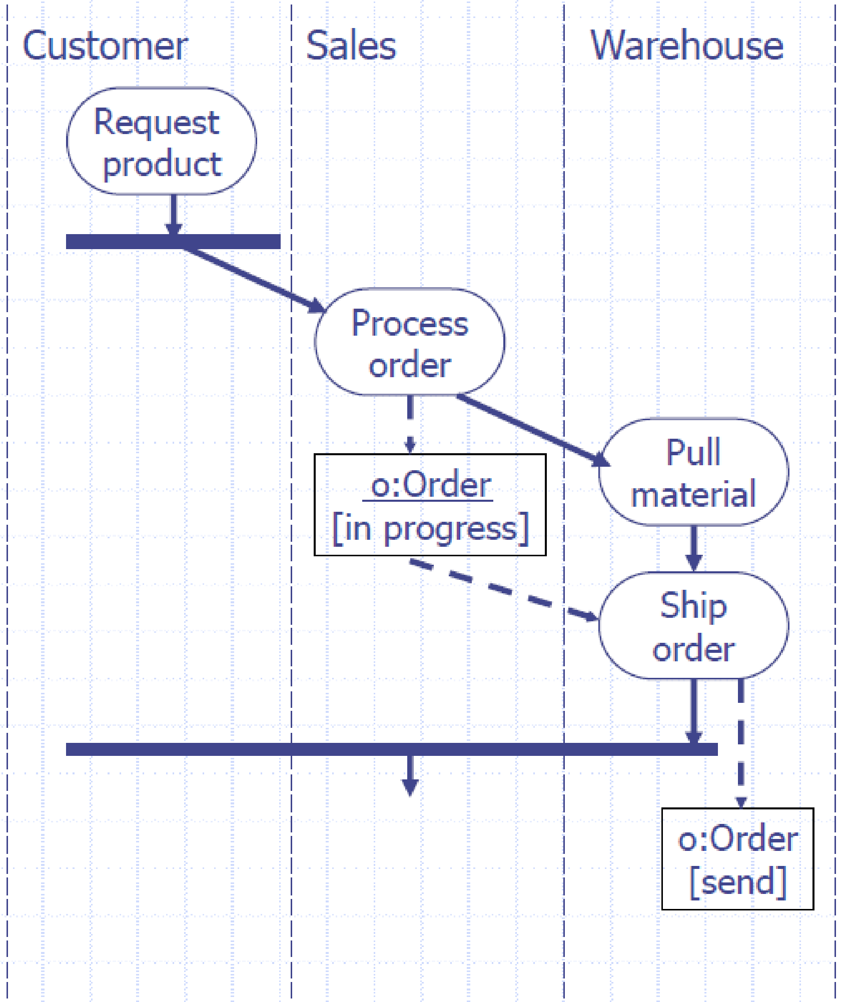
Swimlanes

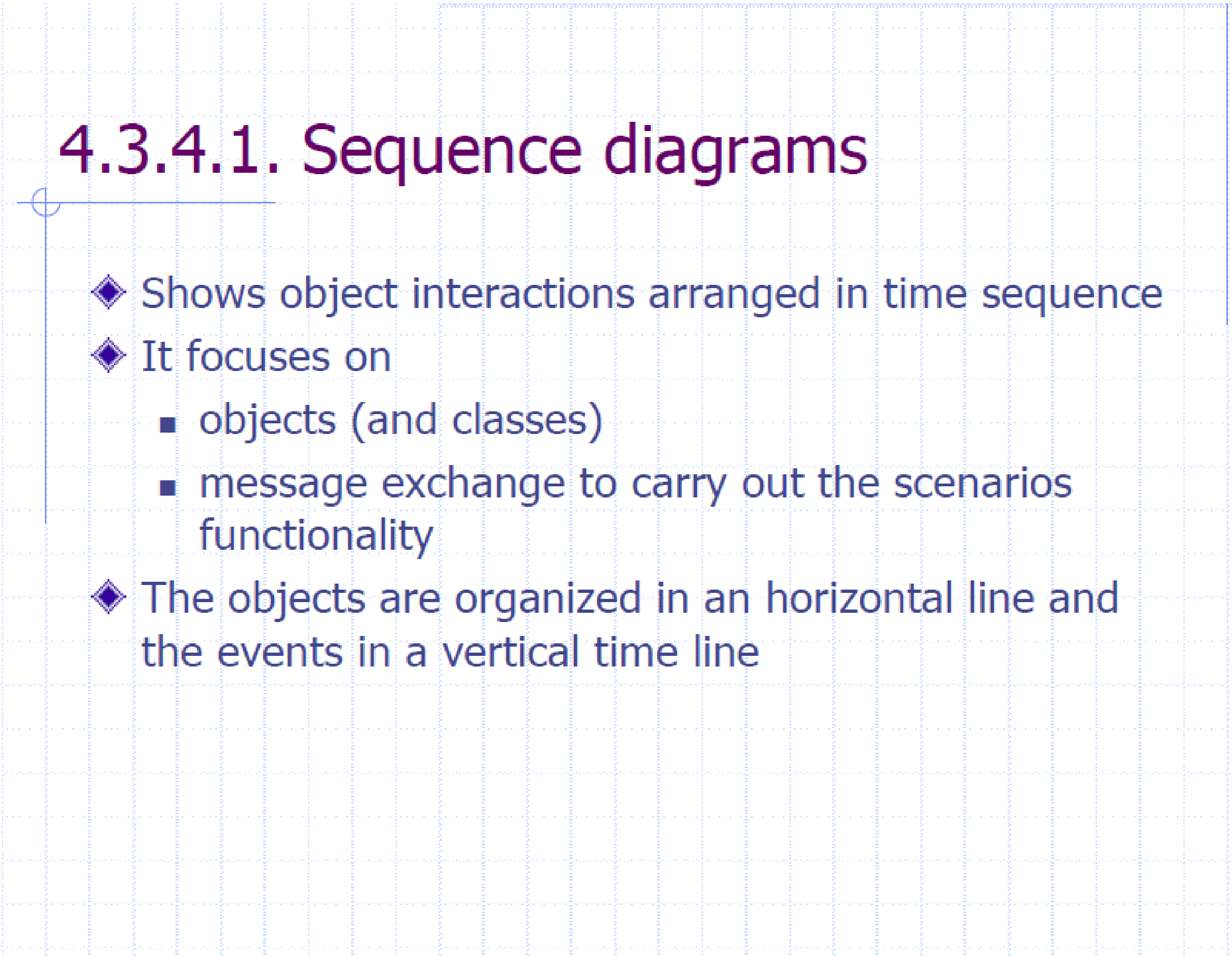
- Identify responsibilities
- Example: Modeling workflows for business units



Object flow

- Shows output objects and (optionally) their state



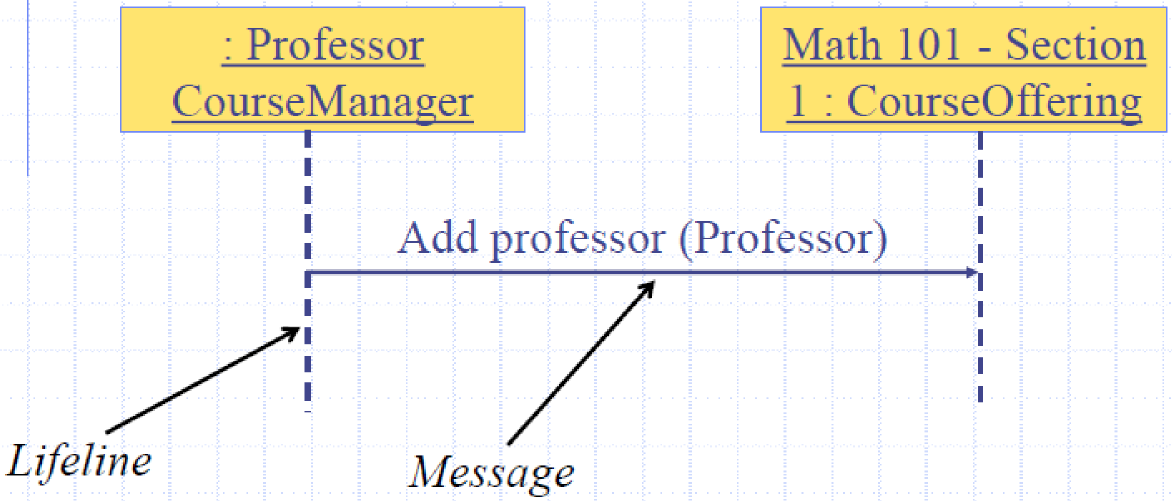


4.3.4.1. Sequence diagrams

- Shows object interactions arranged in time sequence
- It focuses on
  - objects (and classes)
  - message exchange to carry out the scenarios functionality
- The objects are organized in an horizontal line and the events in a vertical time line

Notation example (simple version)

- Messages point from client to supplier



: Professor CourseManager

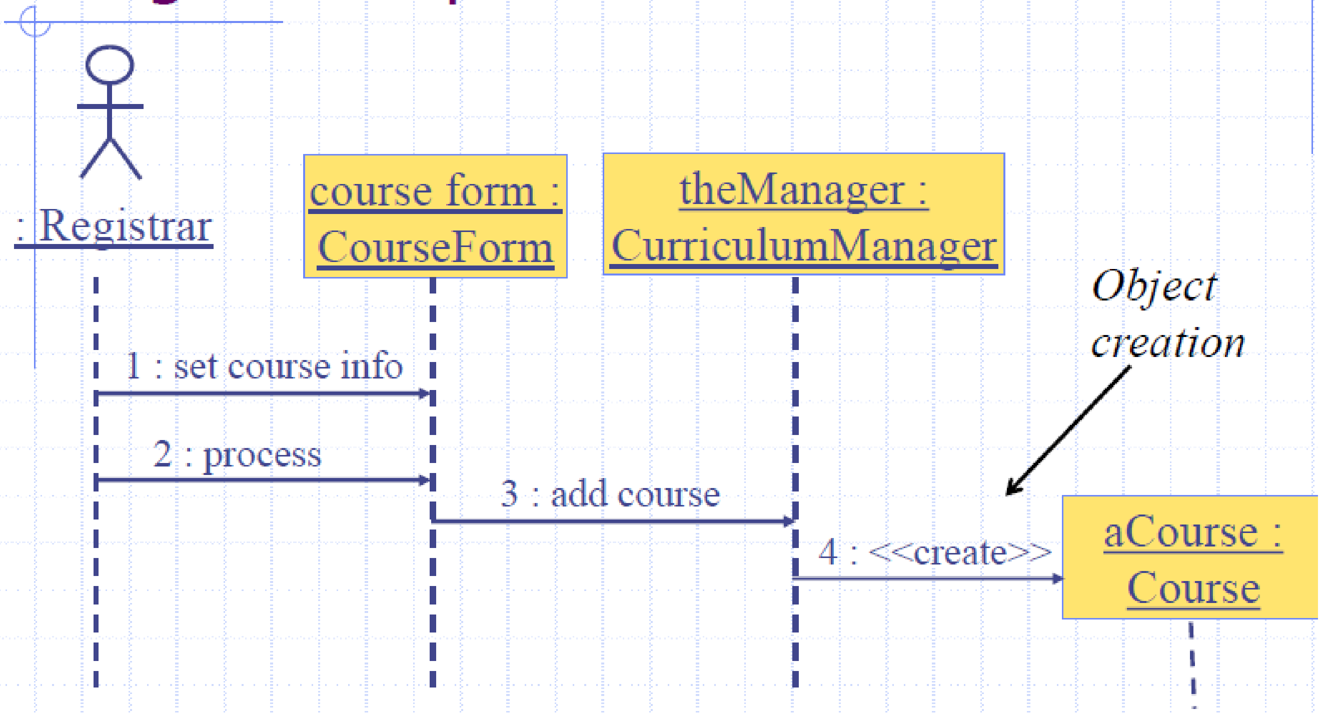
Math 101 - Section 1 : CourseOffering

Add professor (Professor)

Lifeline

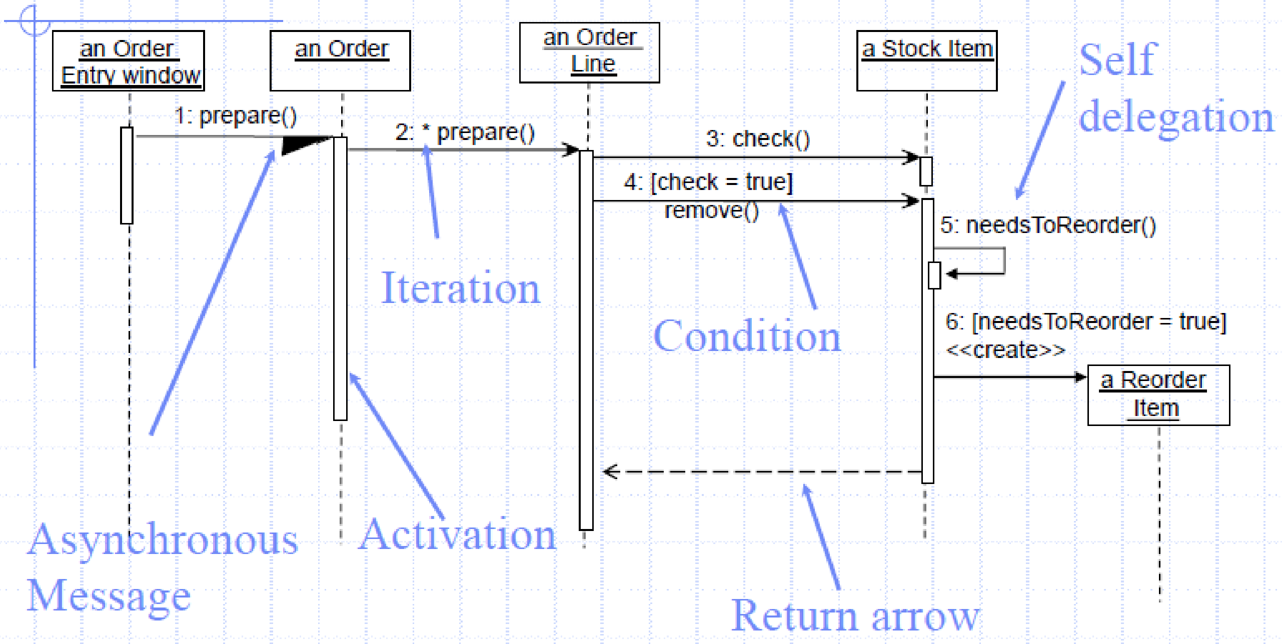
Message

Sequence diagram: Larger Example

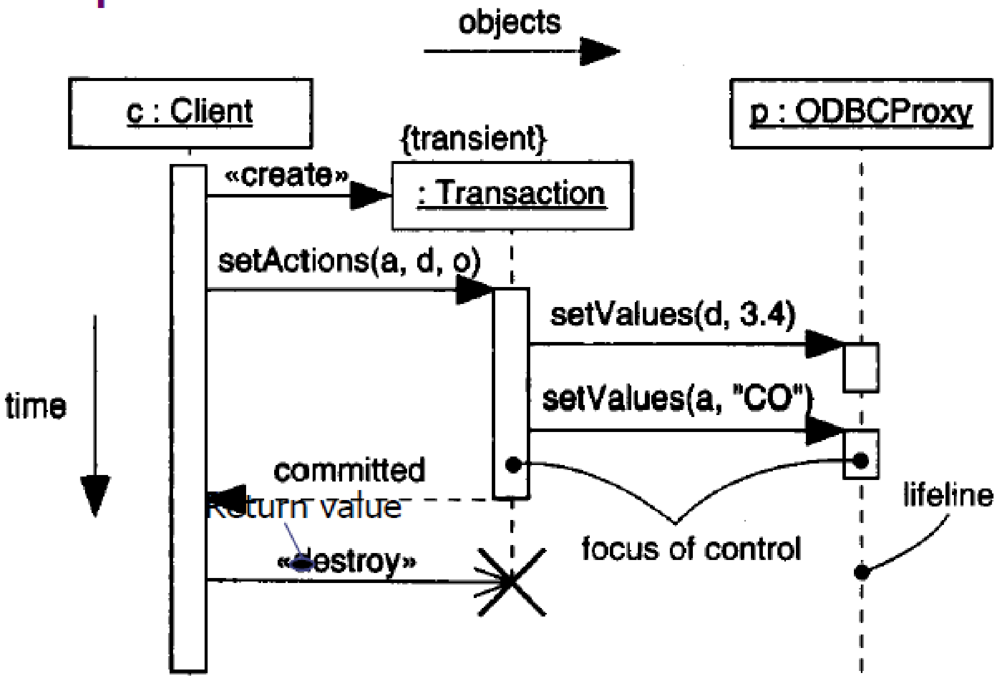


Object creation

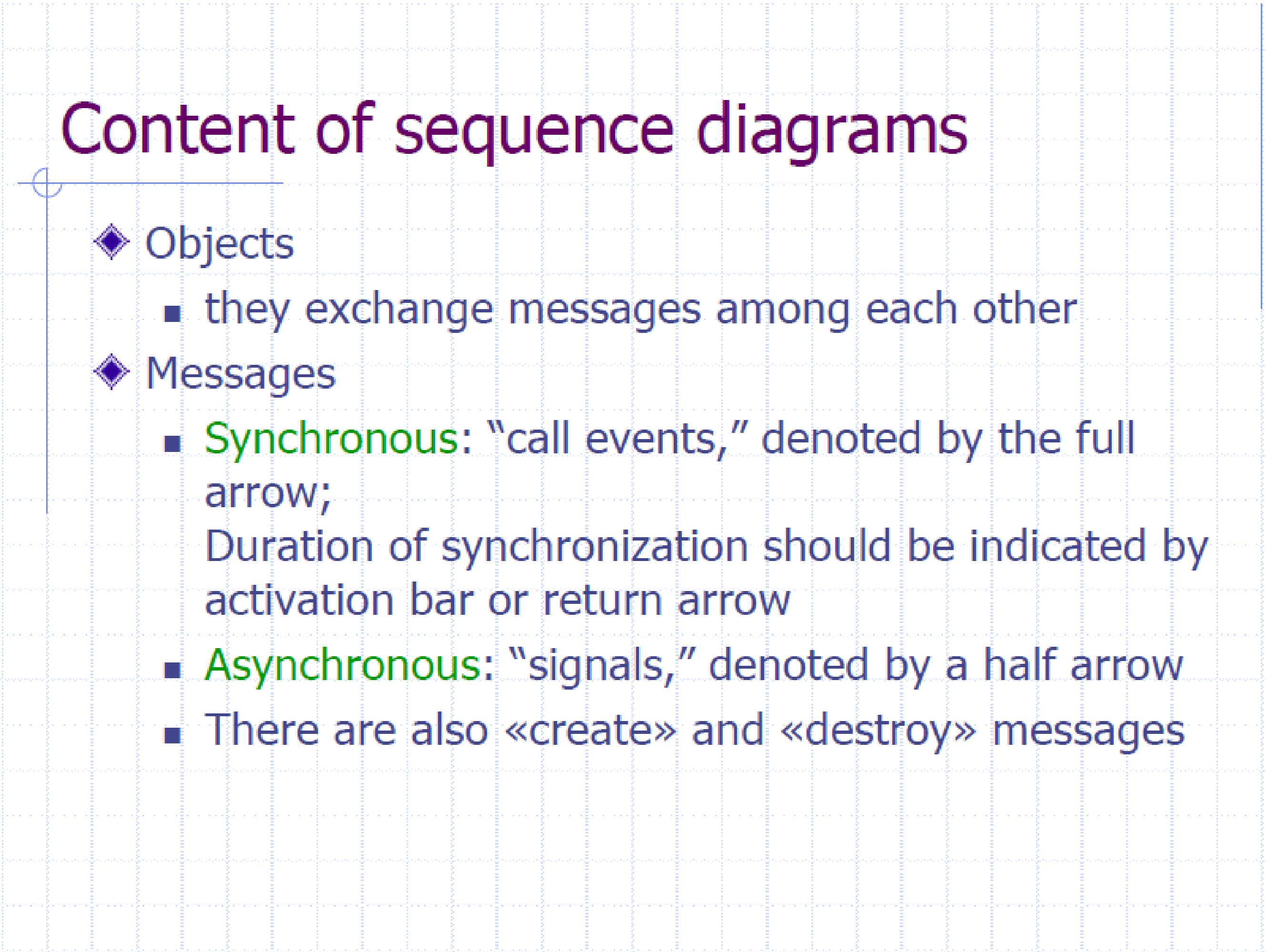
Sequence diagrams: More details



Example of a transaction



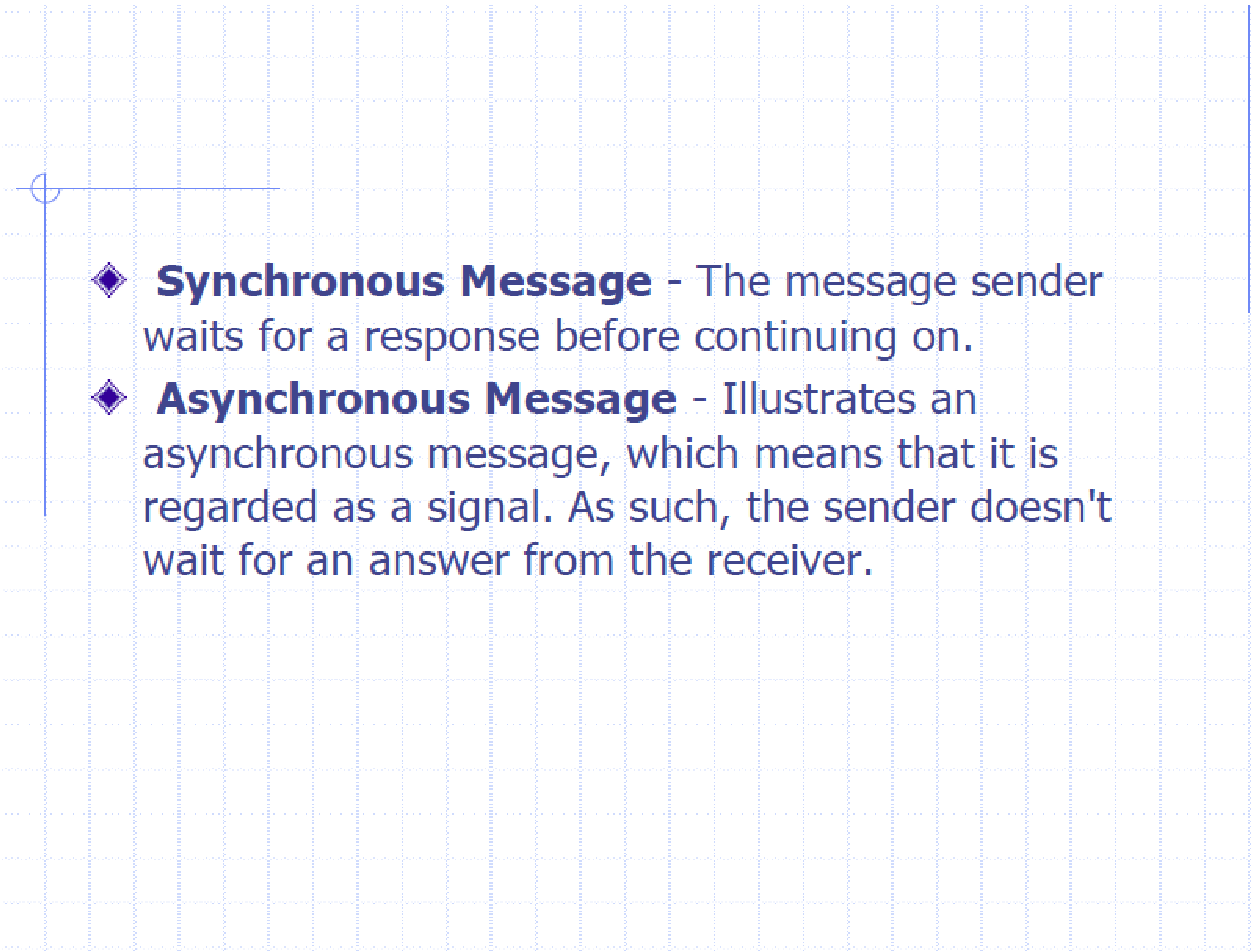
From: G. Booch, J.Rumbaugh, I. Jacobson: The Unified Modeling Language User Guide.
Addison Wesley, 1999, fig 18-2 page 247



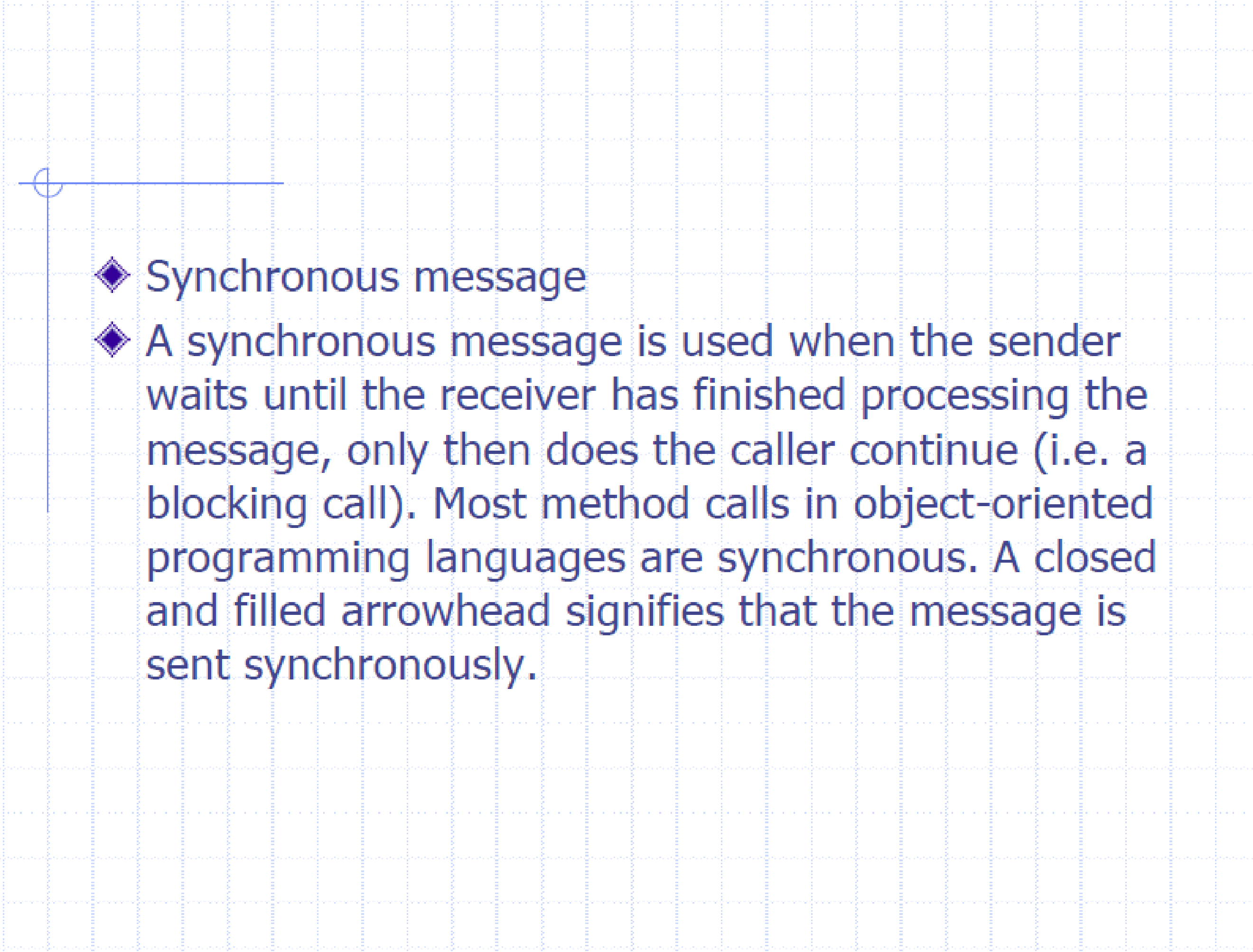
Content of sequence diagrams

- Objects
  - they exchange messages among each other
- Messages
  - Synchronous: "call events," denoted by the full arrow;

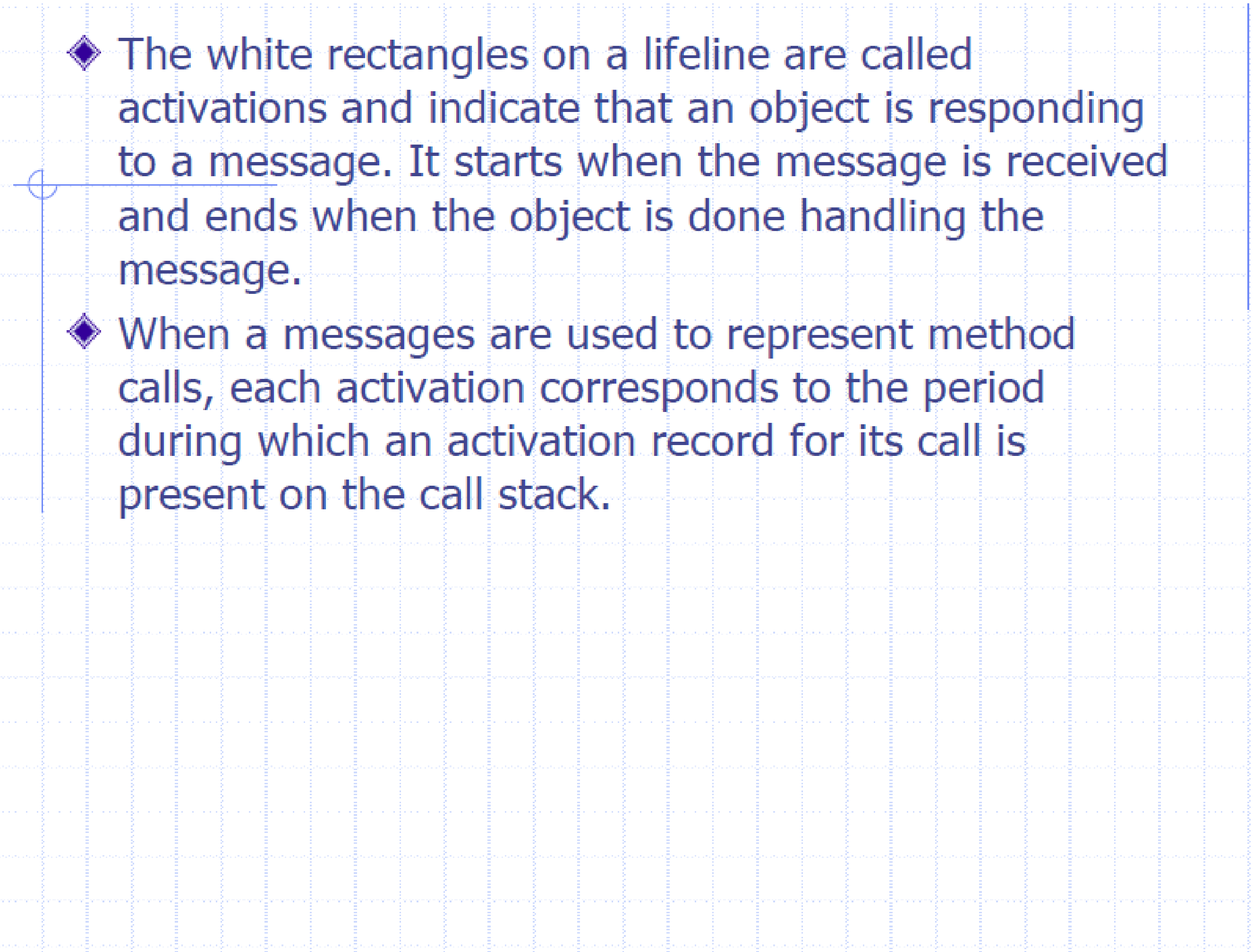
    Duration of synchronization should be indicated by activation bar or return arrow
  - Asynchronous: "signals," denoted by a half arrow
  - There are also «create» and «destroy» messages



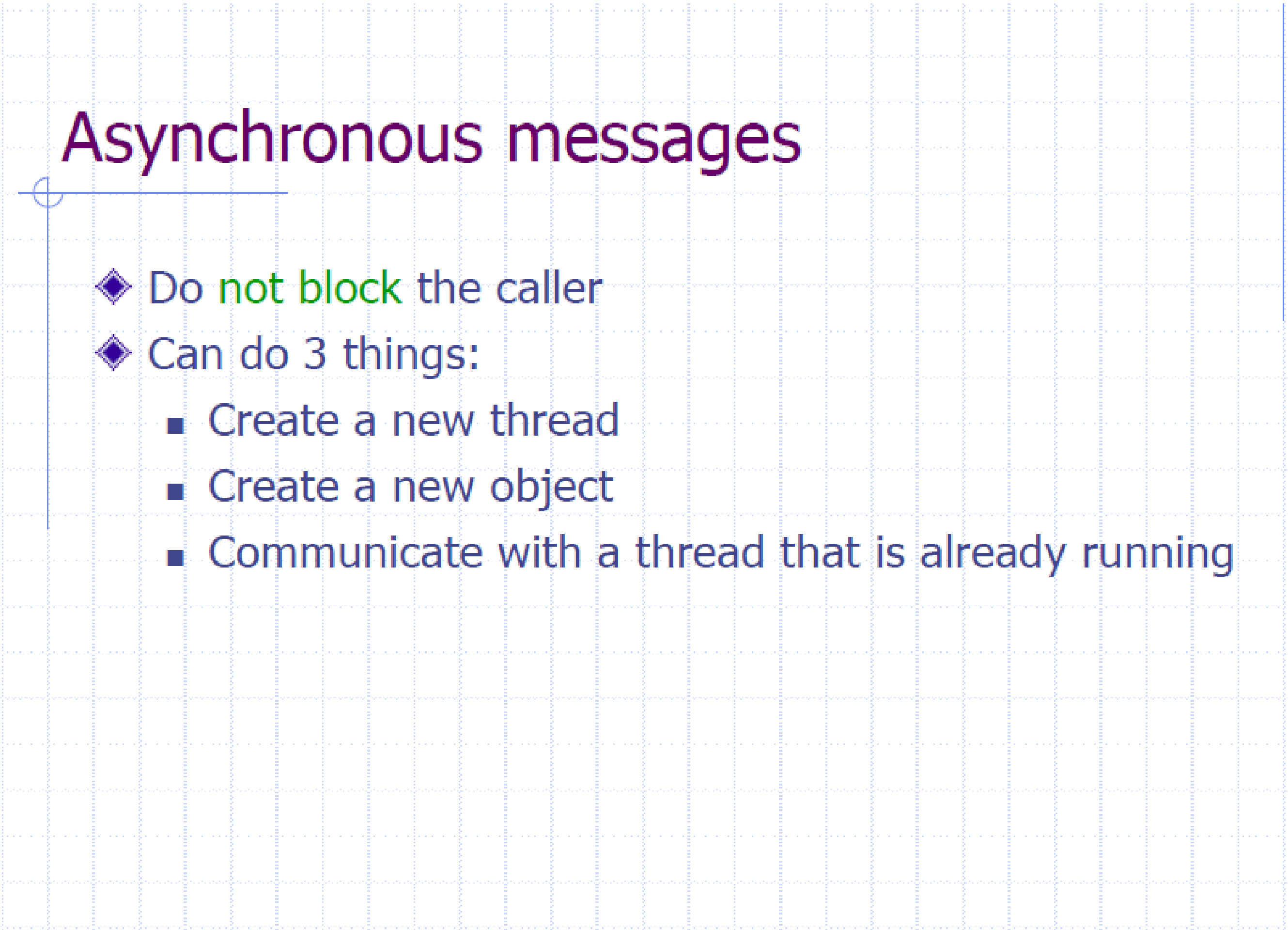
- **Synchronous Message** - The message sender waits for a response before continuing on.
- **Asynchronous Message** - Illustrates an asynchronous message, which means that it is regarded as a signal. As such, the sender doesn't wait for an answer from the receiver.



- Synchronous message
- A synchronous message is used when the sender waits until the receiver has finished processing the message, only then does the caller continue (i.e. a blocking call). Most method calls in object-oriented programming languages are synchronous. A closed and filled arrowhead signifies that the message is sent synchronously.

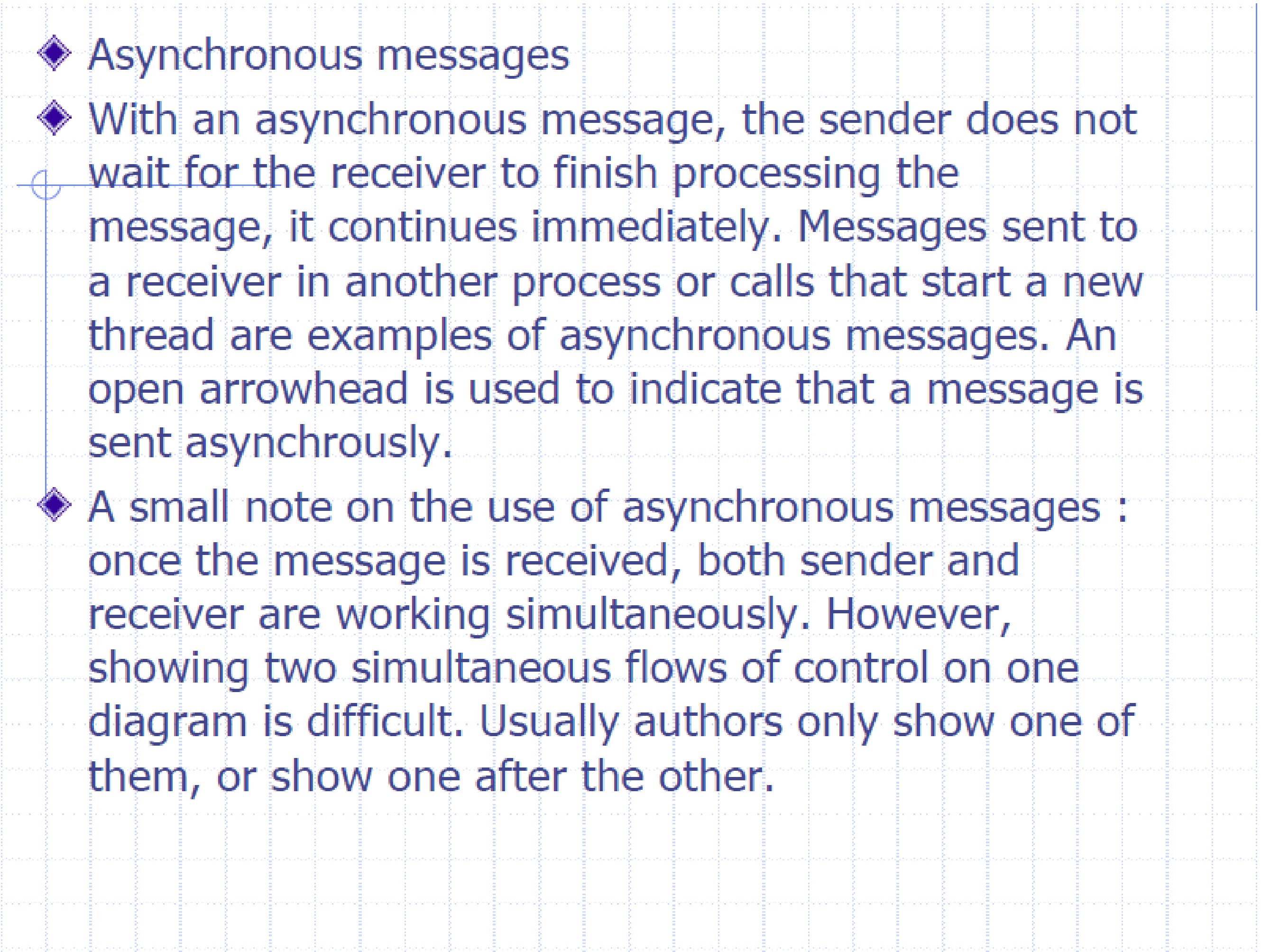


- The white rectangles on a lifeline are called activations and indicate that an object is responding to a message. It starts when the message is received and ends when the object is done handling the message.
- When a messages are used to represent method calls, each activation corresponds to the period during which an activation record for its call is present on the call stack.

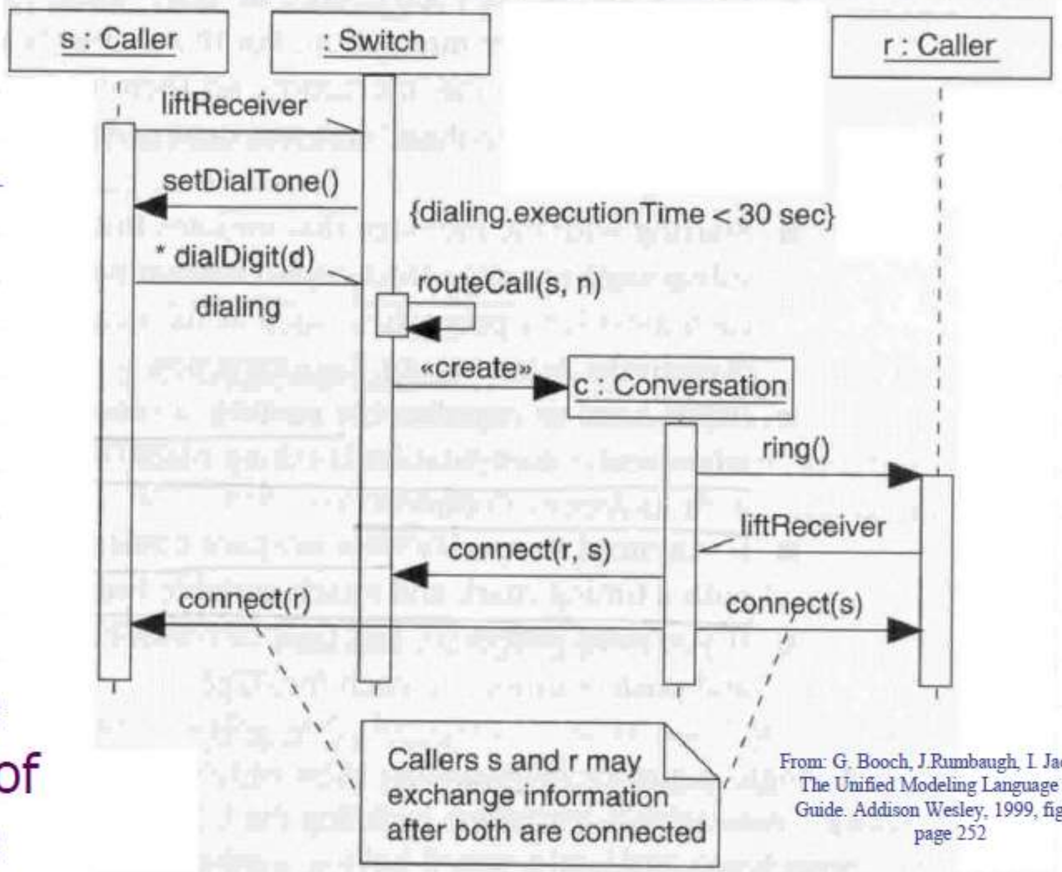


Asynchronous messages

- Do not block the caller
- Can do 3 things:
  - Create a new thread
  - Create a new object
  - Communicate with a thread that is already running

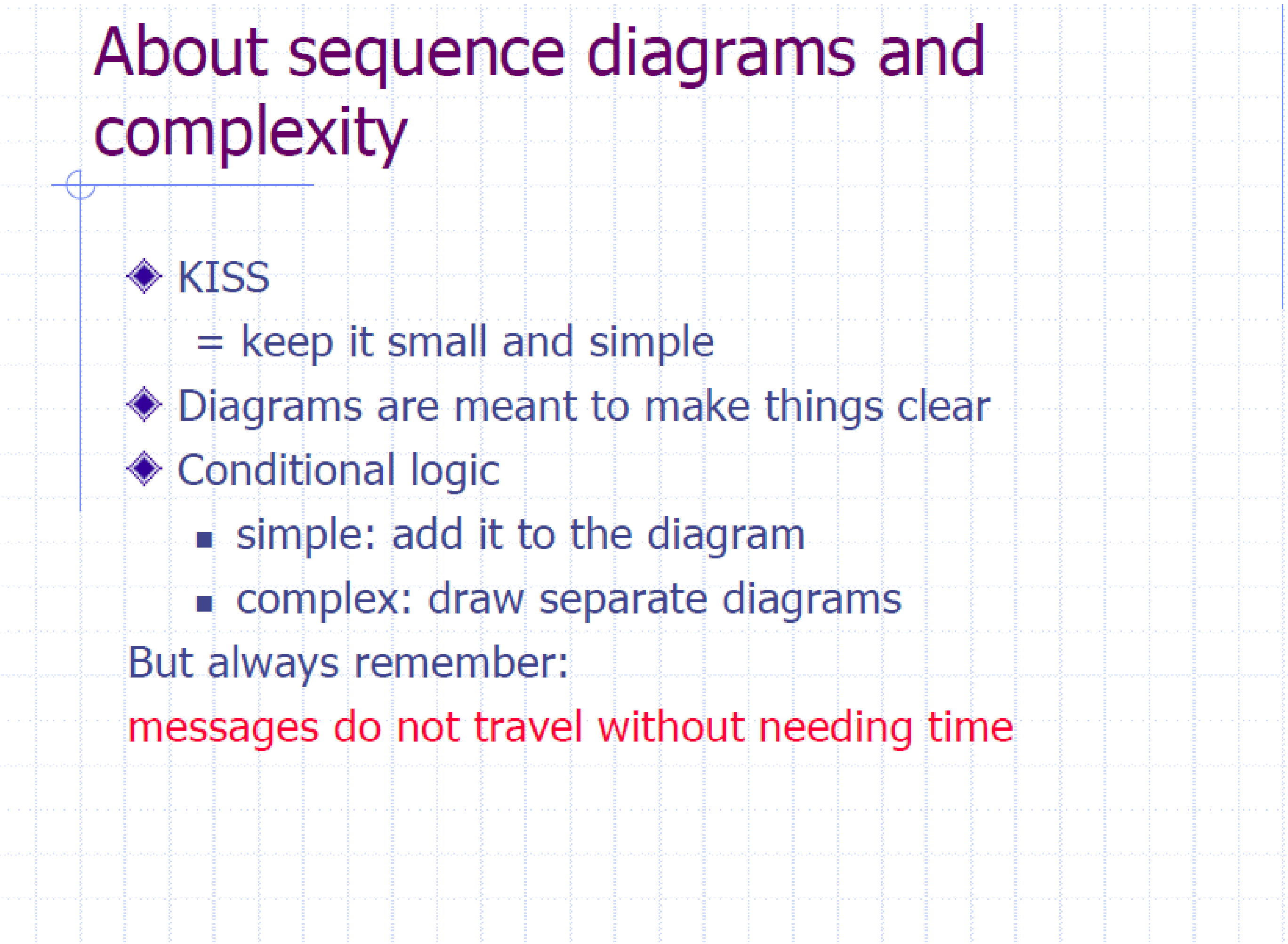


- Asynchronous messages
- With an asynchronous message, the sender does not wait for the receiver to finish processing the message, it continues immediately. Messages sent to a receiver in another process or calls that start a new thread are examples of asynchronous messages. An open arrowhead is used to indicate that a message is sent asynchrously.
- A small note on the use of asynchronous messages : once the message is received, both sender and receiver are working simultaneously. However, showing two simultaneous flows of control on one diagram is difficult. Usually authors only show one of them, or show one after the other.



From: G. Booch, J.Rumbaugh, I. Jacobson: The Unified Modeling Language User Guide. Addison Wesley, 1999, fig 18-4 page 252

Complete example of sequence diagrams



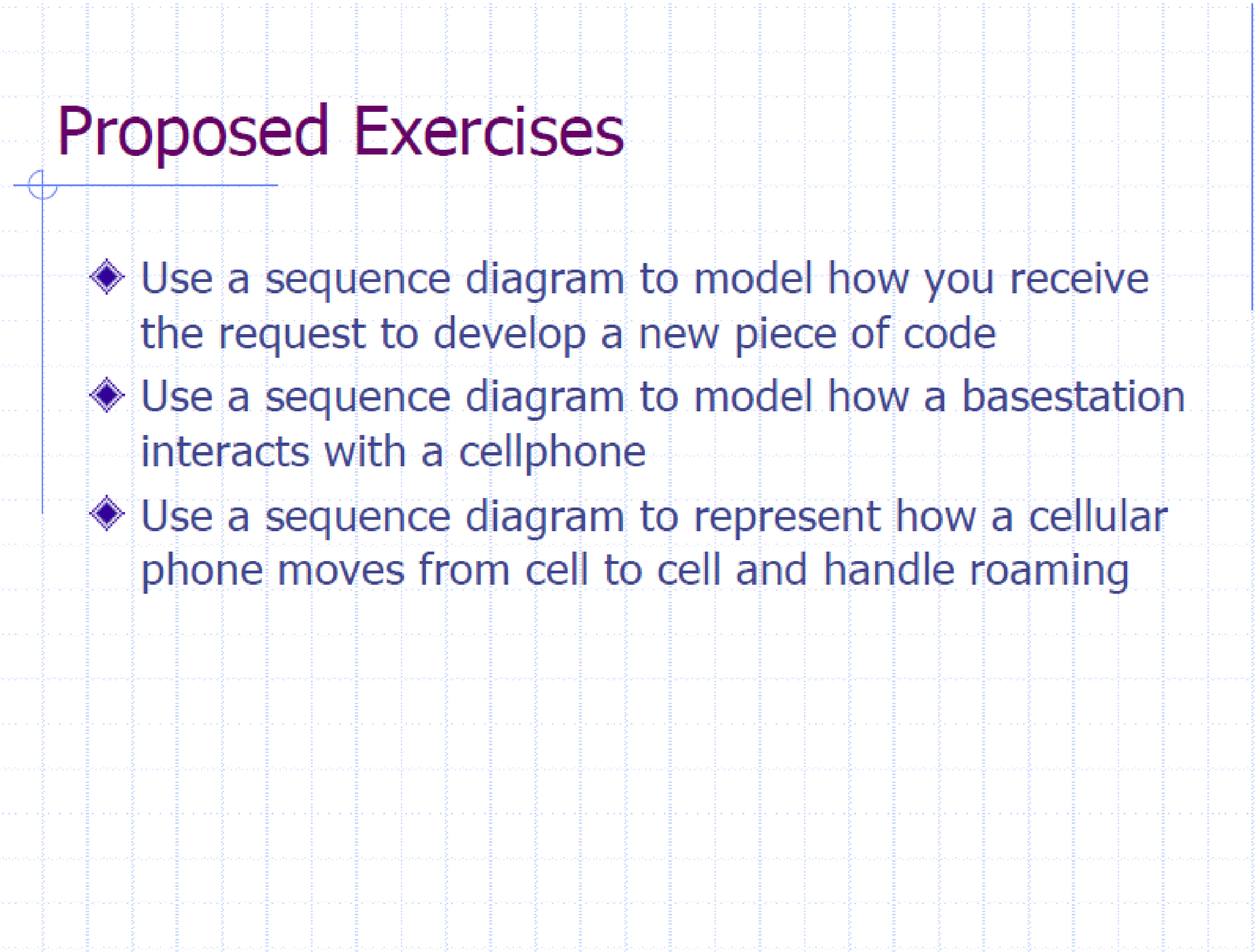
About sequence diagrams and complexity

- KISS

  = keep it small and simple
- Diagrams are meant to make things clear
- Conditional logic
  - simple: add it to the diagram
  - complex: draw separate diagrams

But always remember:

messages do not travel without needing time

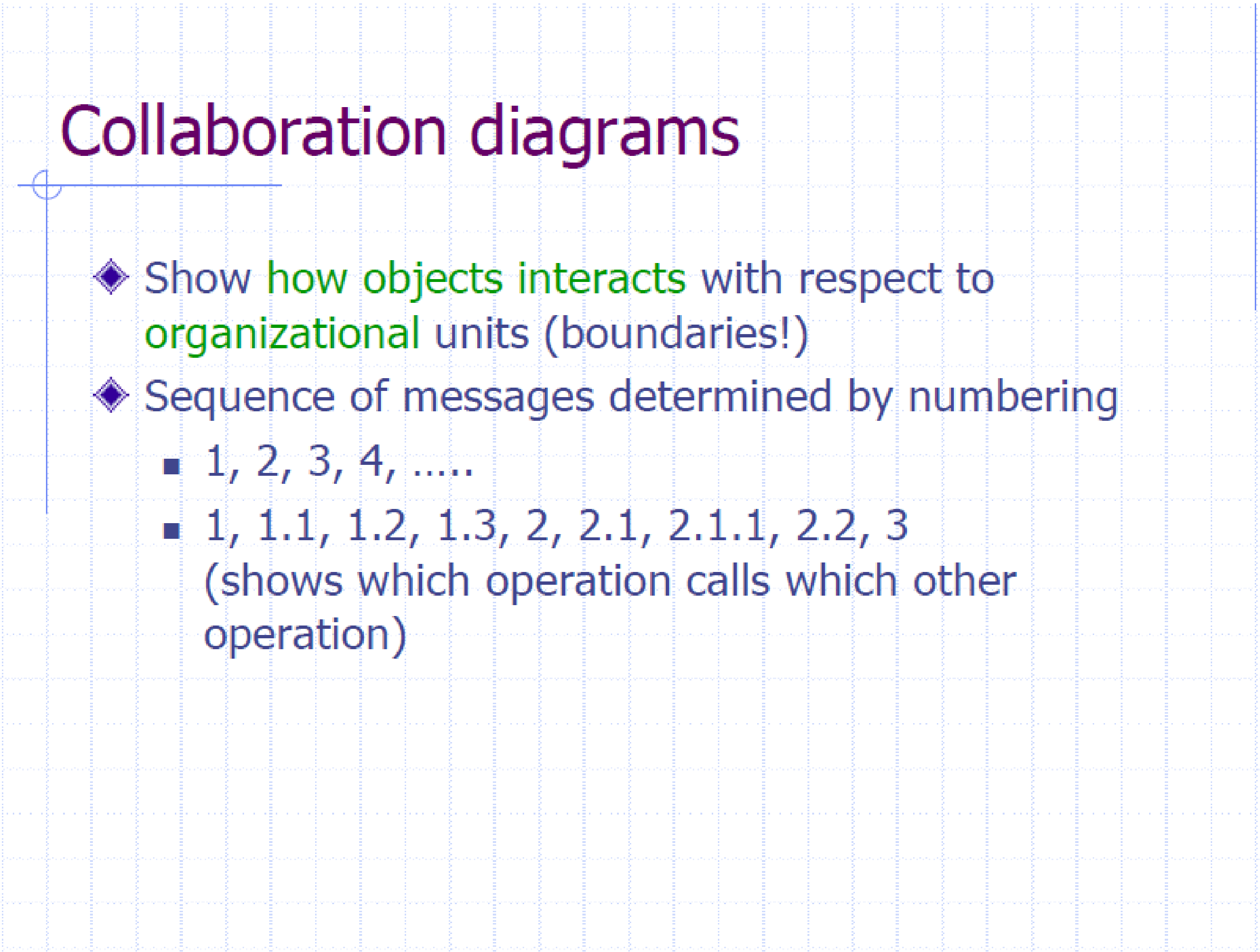


Proposed Exercises

- Use a sequence diagram to model how you receive the request to develop a new piece of code
- Use a sequence diagram to model how a basestation interacts with a cellphone
- Use a sequence diagram to represent how a cellular phone moves from cell to cell and handle roaming

4.3.4.2. Collaboration diagrams: Where are the boundaries?

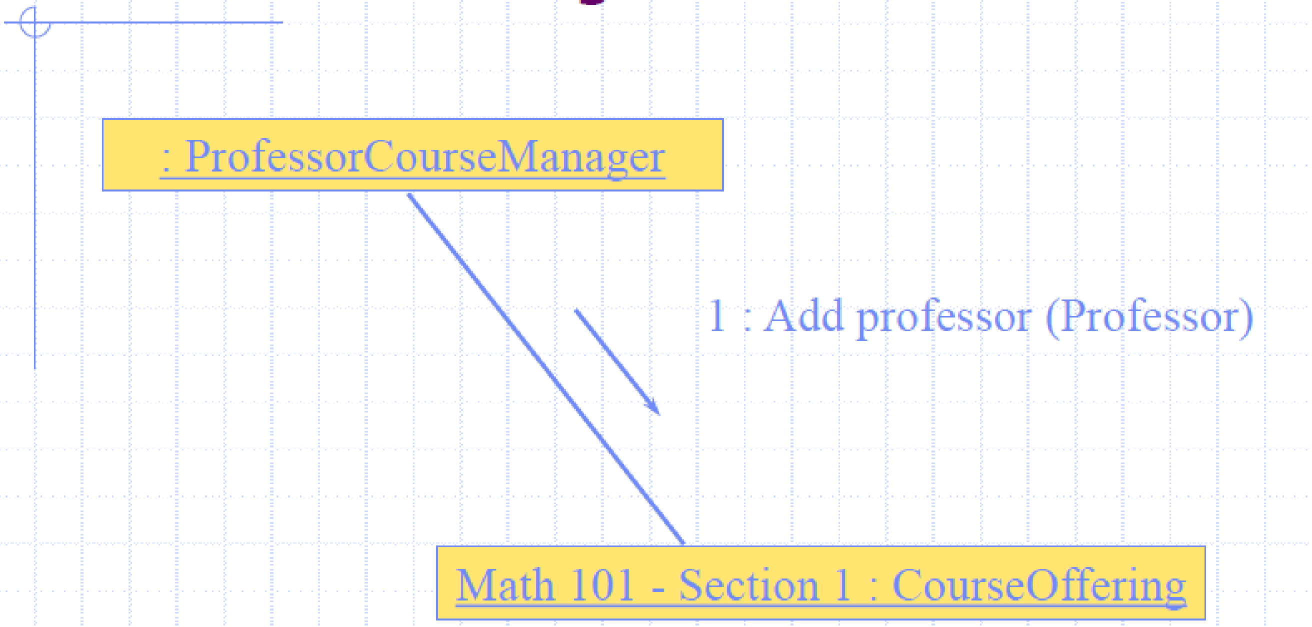
- A boundary shapes communication between system and outside world
  - e.g. user interface or other system
  - Represent actors from use case diagrams
- It may be useful to show boundary classes in interaction diagrams:
  - capture interface requirements
  - do NOT show how the interface will be implemented



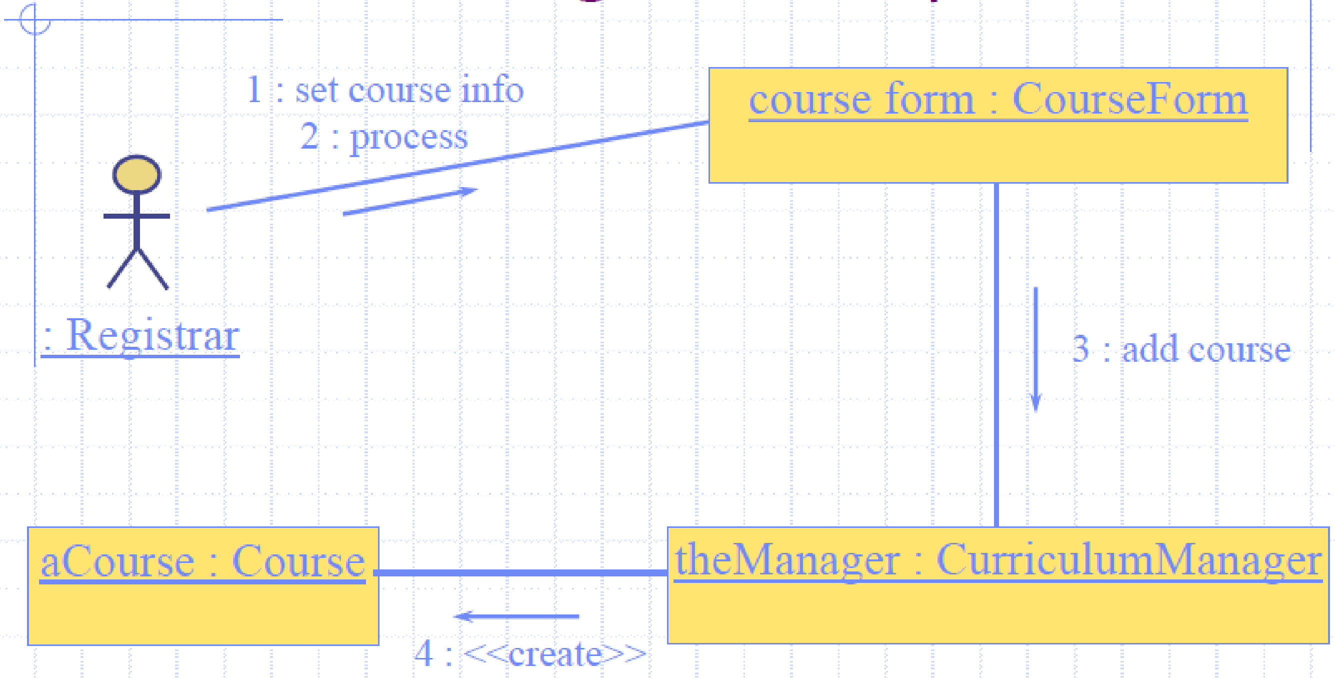
Collaboration diagrams

- Show how objects interacts with respect to organizational units (boundaries!)
- Sequence of messages determined by numbering
  - 1, 2, 3, 4, …..
  - 1, 1.1, 1.2, 1.3, 2, 2.1, 2.1.1, 2.2, 3 (shows which operation calls which other operation)

Collaboration diagram basics


: ProfessorCourseManager
1 : Add professor (Professor)
Math 101 - Section 1 : CourseOffering

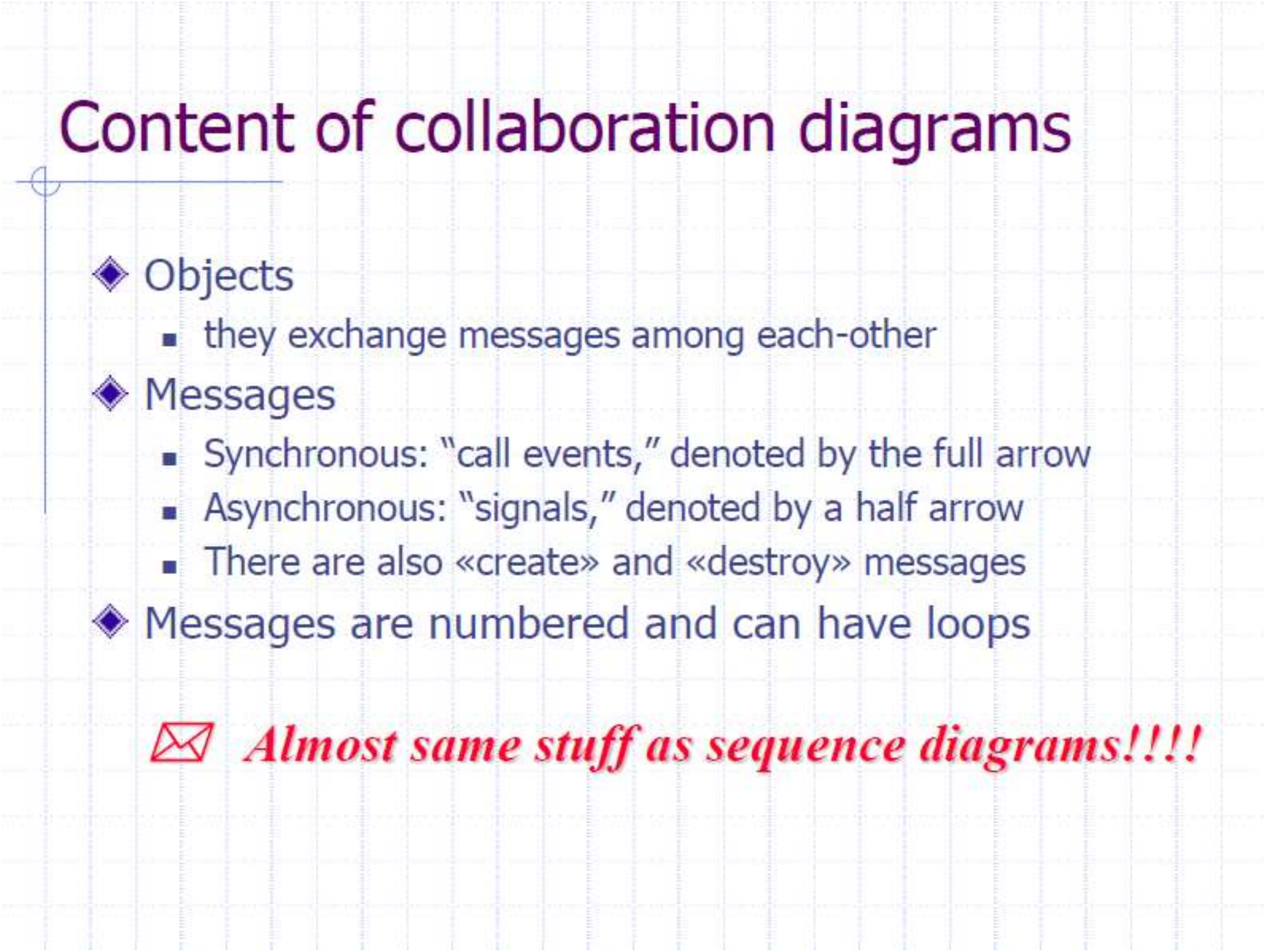
Collaboration diagram example



Comparing sequence & collaboration diagrams

- Layout of collaboration diagrams may show static connections of objects
- Sequence diagrams are best to see the flow of time
  - Sequence of messages more difficult to understand in collaboration diagrams
- Object organization with control flow is best seen through collaboration diagrams
  - Layout of collaboration diagrams may show static connections of objects

Note: Complex control is difficult to express anyway!!!

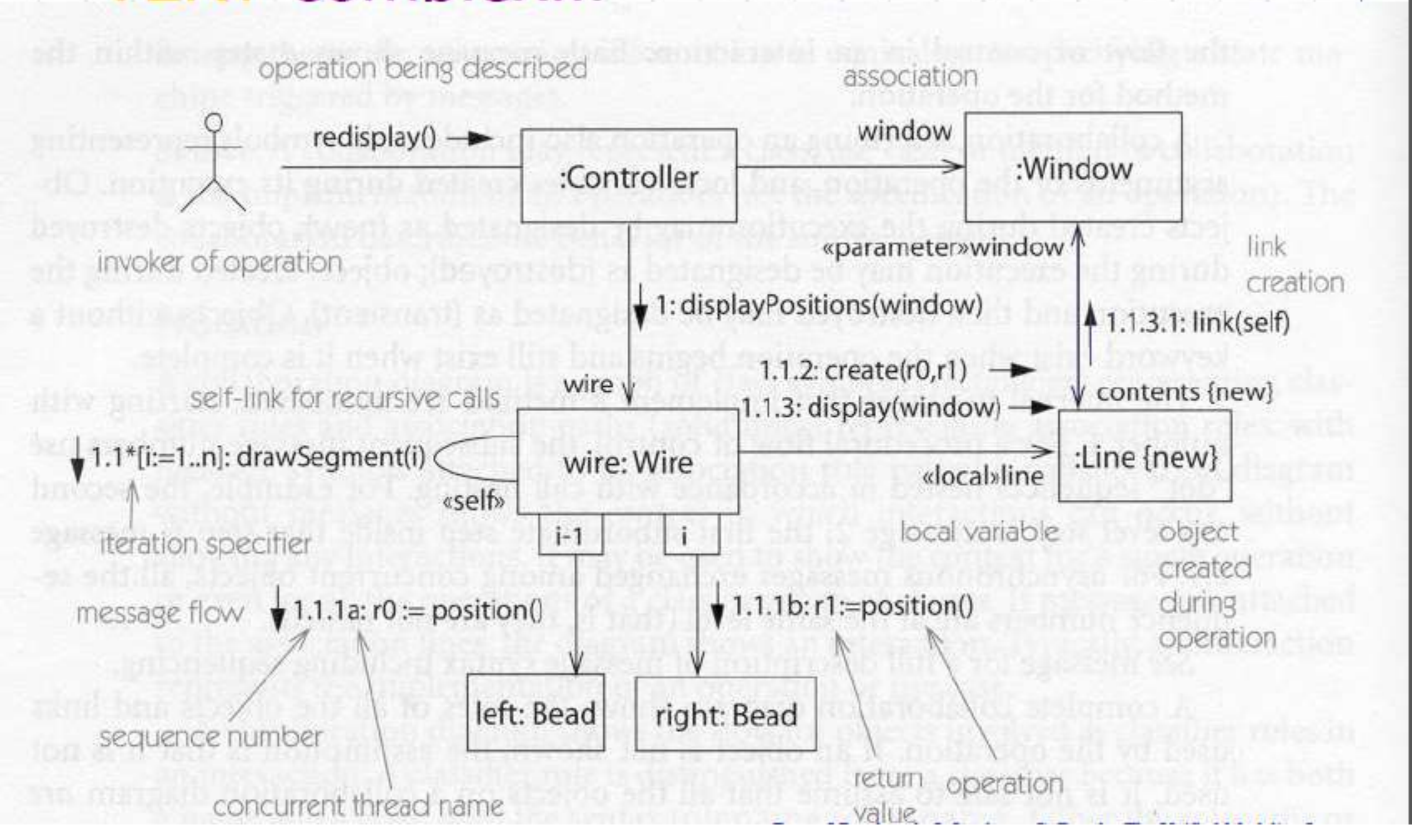


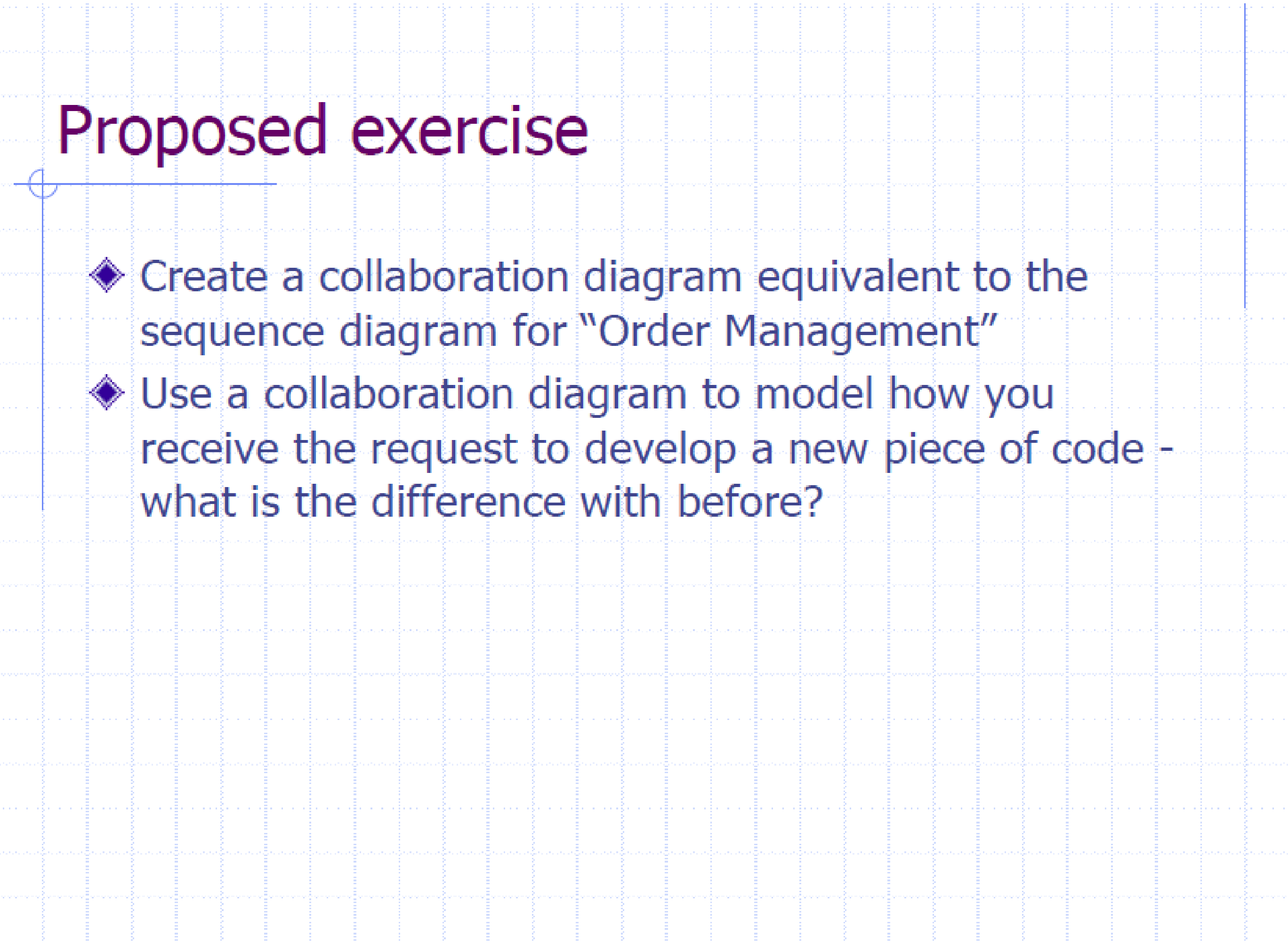
Content of collaboration diagrams

- Objects
  - they exchange messages among each-other
- Messages
  - Synchronous: "call events," denoted by the full arrow
  - Asynchronous: "signals," denoted by a half arrow
  - There are also «create» and «destroy» messages
- Messages are numbered and can have loops

✉ ***Almost same stuff as sequence diagrams!!!!***

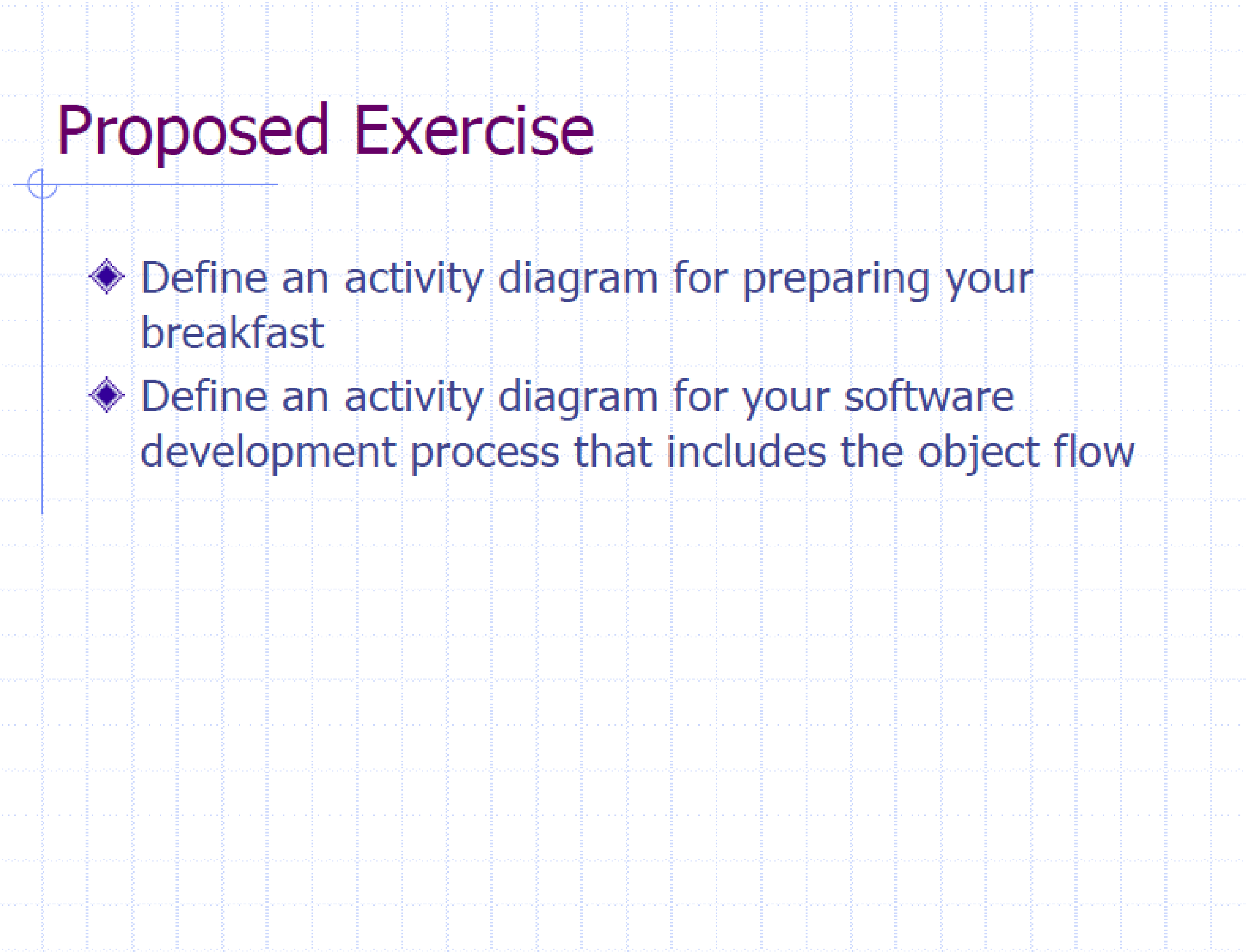
Collaboration diagrams can become VERY complex…





Proposed exercise

- Create a collaboration diagram equivalent to the sequence diagram for “Order Management”
- Use a collaboration diagram to model how you receive the request to develop a new piece of code - what is the difference with before?

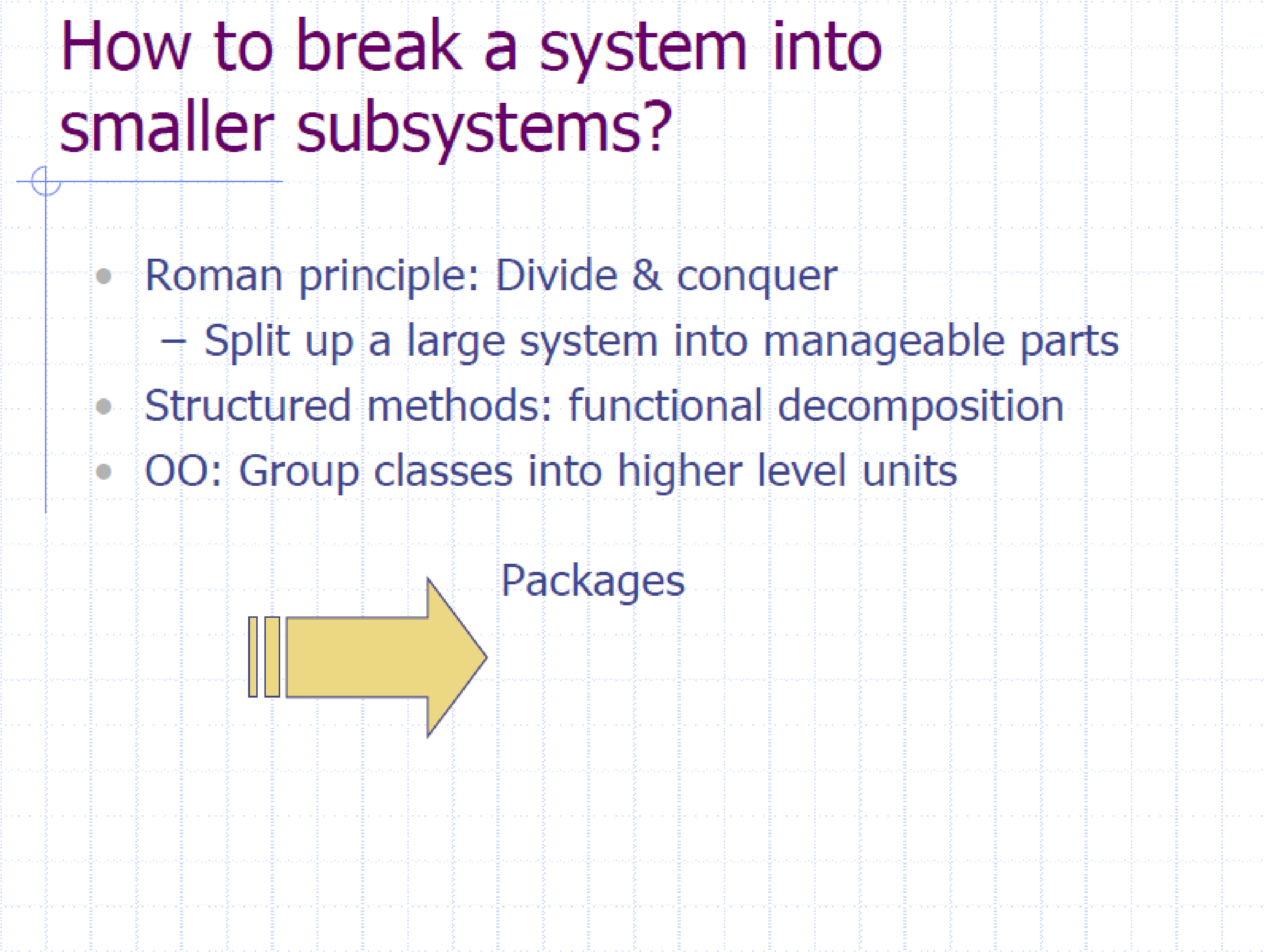


Proposed Exercise

- Define an activity diagram for preparing your breakfast
- Define an activity diagram for your software development process that includes the object flow

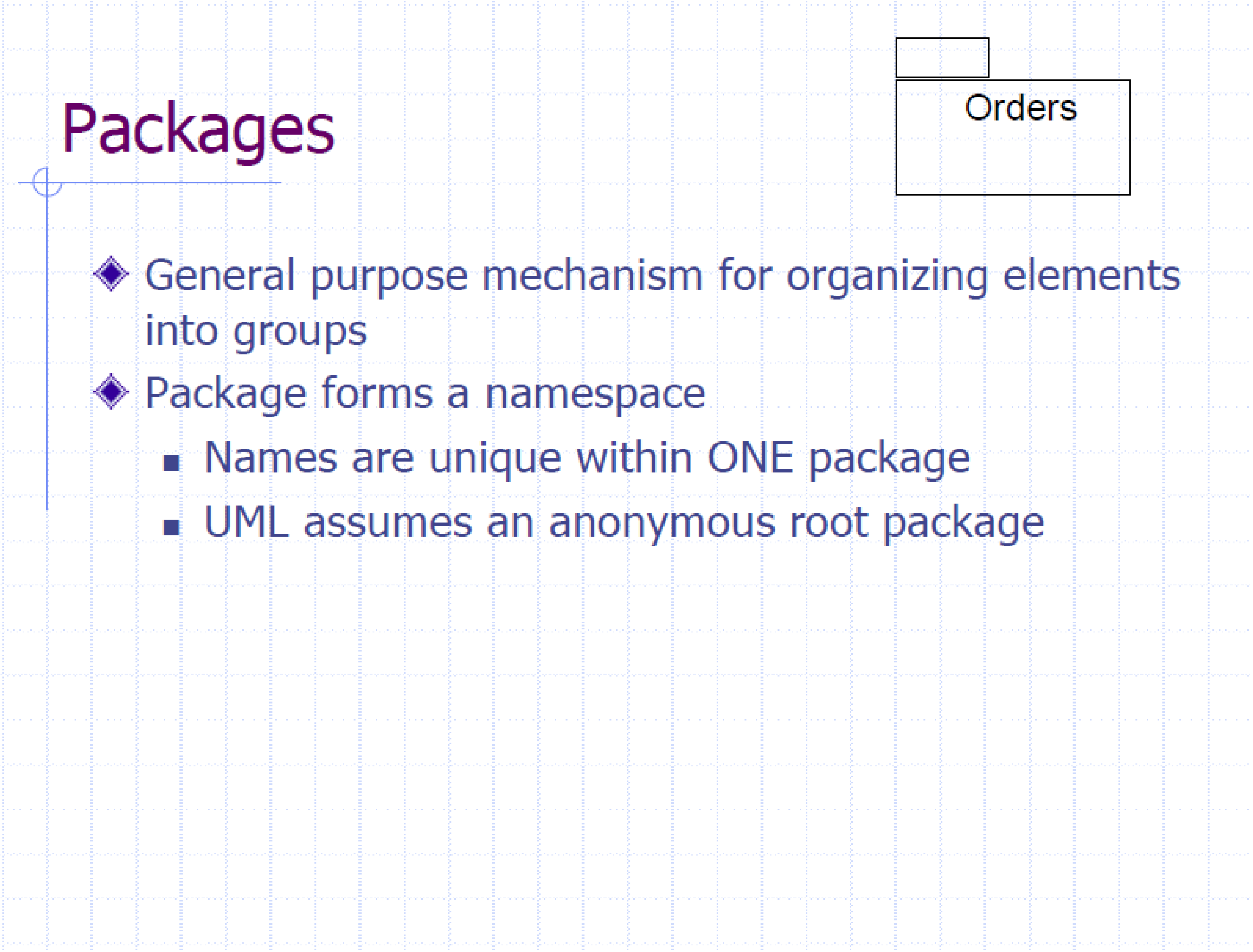
4.4. Package Diagrams

- Remember: packages
- Package Diagrams
- Dependencies
- Ownership



How to break a system into smaller subsystems?

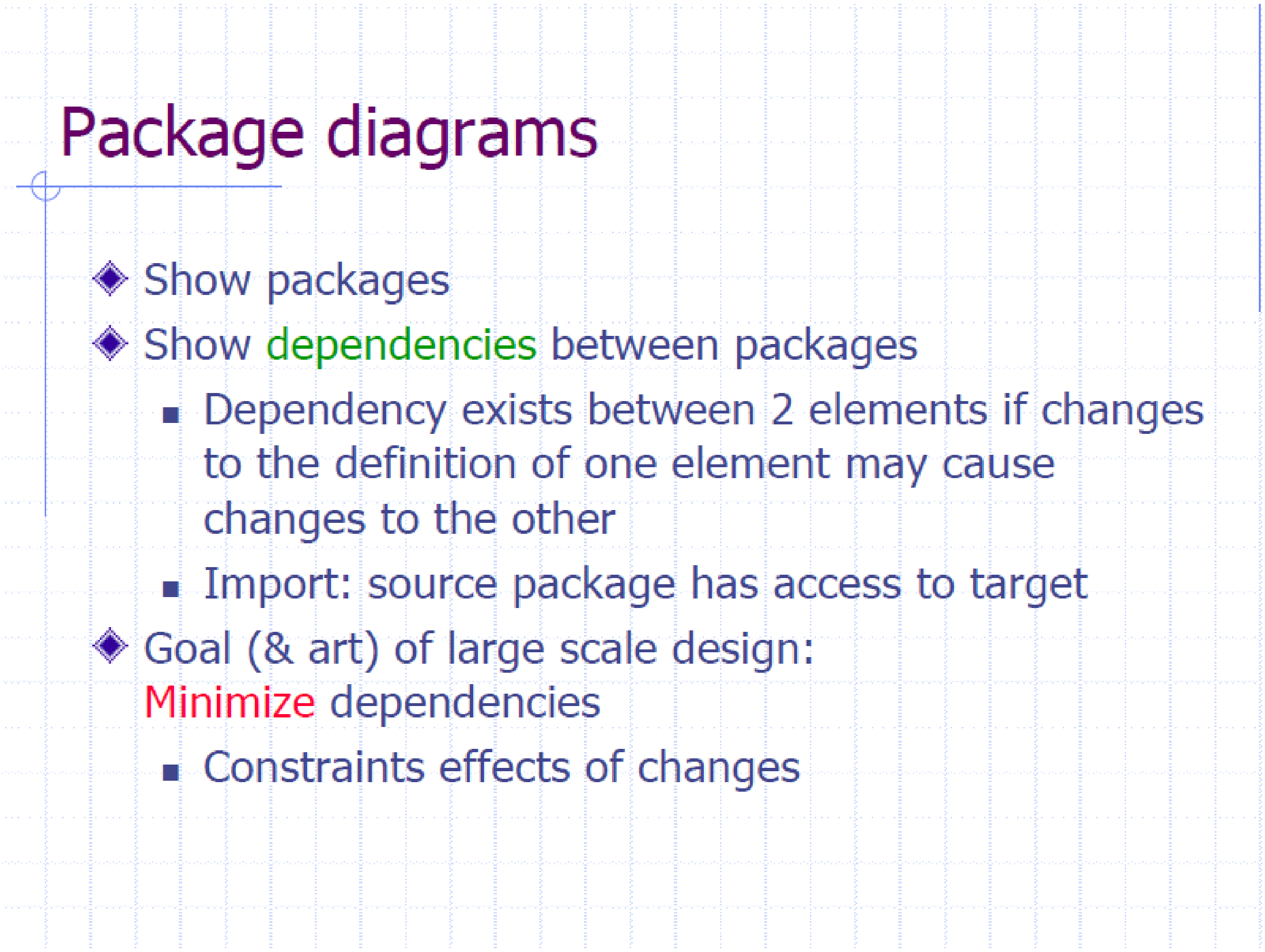
- Roman principle: Divide & conquer
  - Split up a large system into manageable parts
- Structured methods: functional decomposition
- OO: Group classes into higher level units



Packages

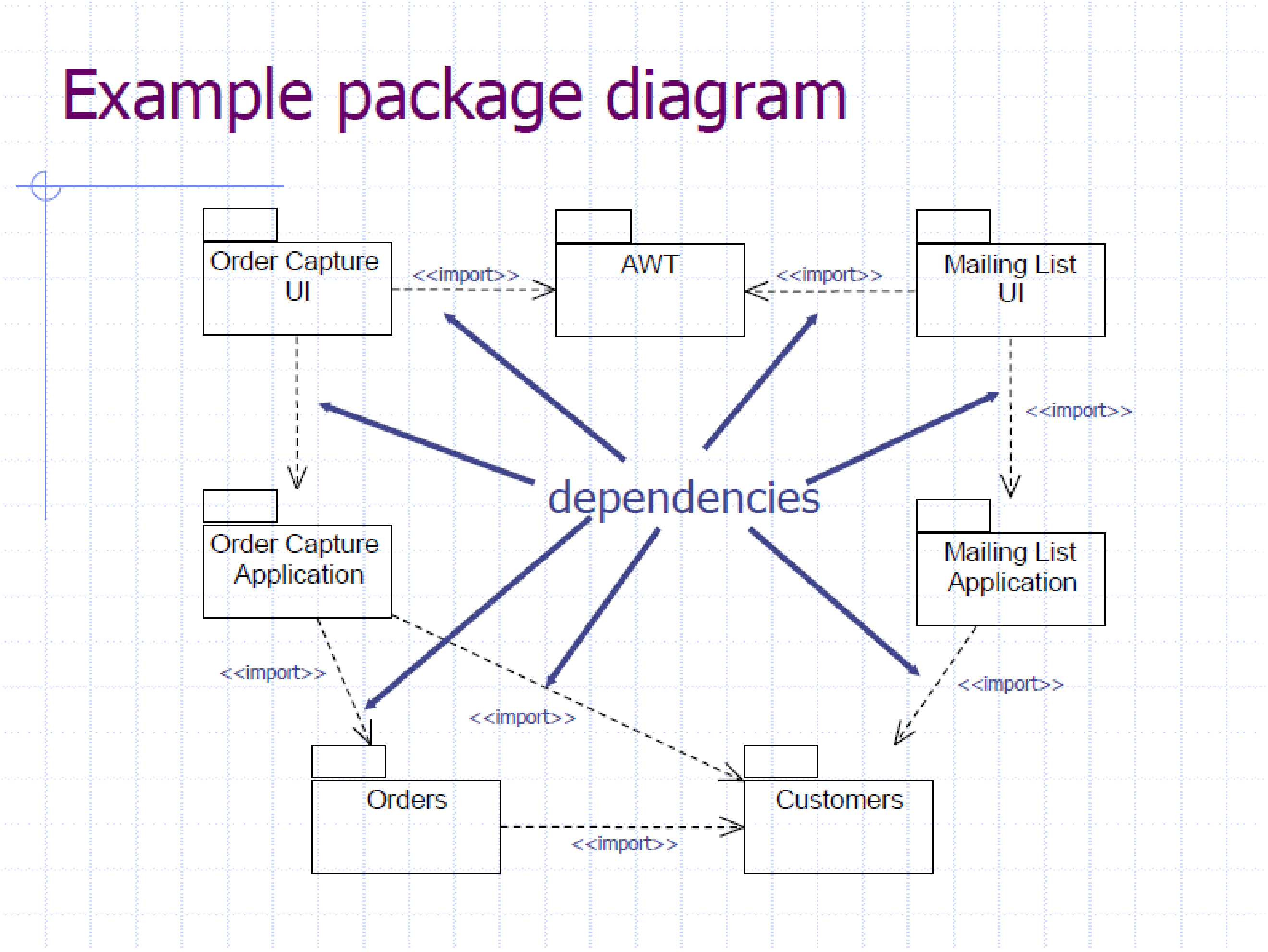
Orders

- General purpose mechanism for organizing elements into groups
- Package forms a namespace
  - Names are unique within ONE package
  - UML assumes an anonymous root package



Package diagrams

- Show packages
- Show dependencies between packages
  - Dependency exists between 2 elements if changes to the definition of one element may cause changes to the other
  - Import: source package has access to target
- Goal (& art) of large scale design: Minimize dependencies
  - Constraints effects of changes



Example package diagram

Order Capture UI

AWT

Mailing List UI

Order Capture Application

Mailing List Application

Orders

Customers

<<import>>

<<import>>

<<import>>

<<import>>

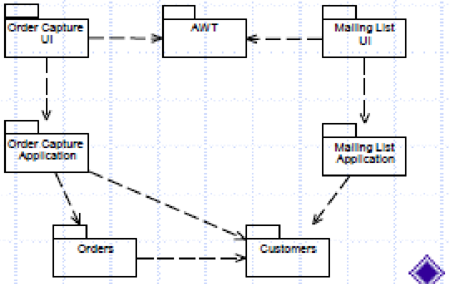
<<import>>

<<import>>

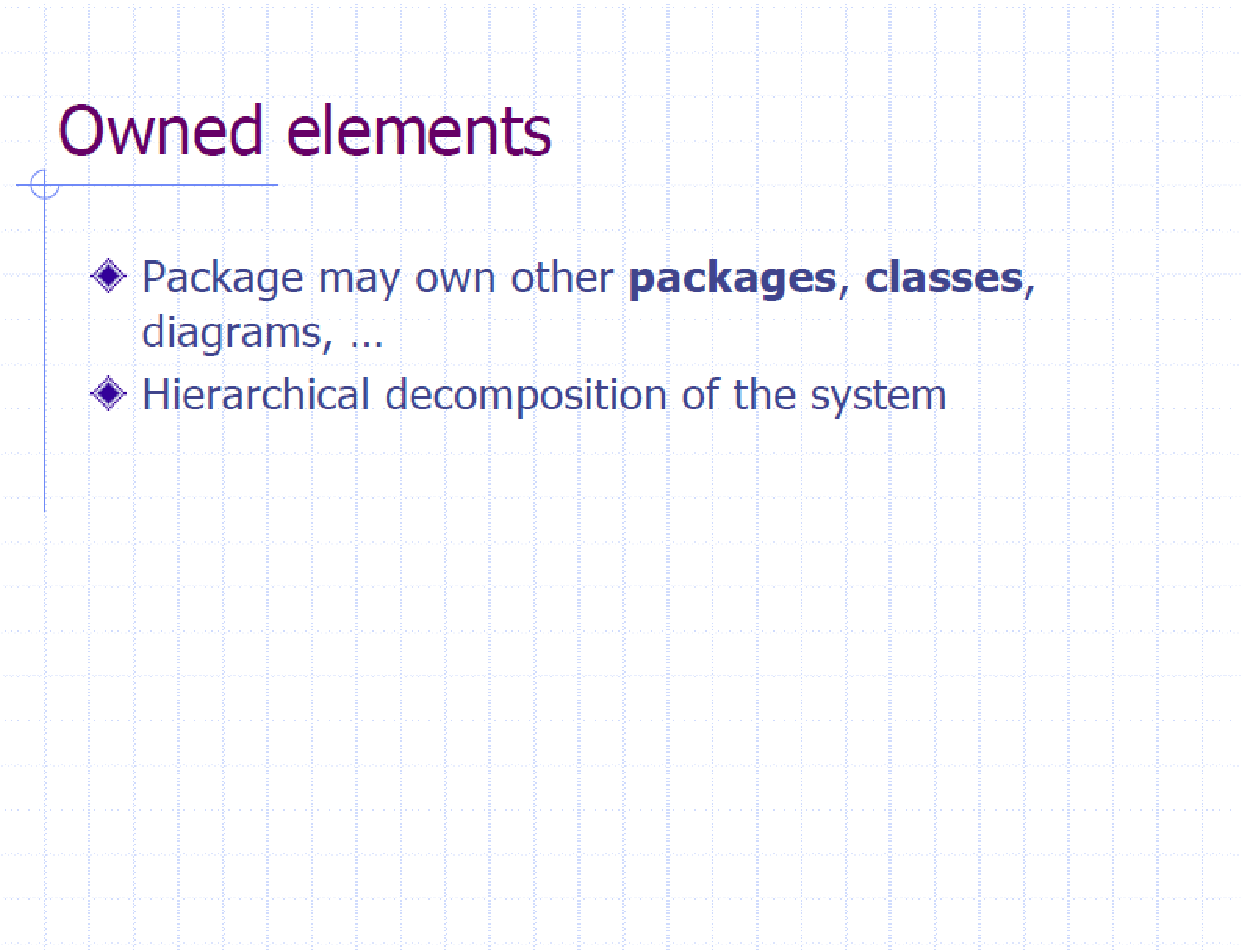
<<import>>

dependencies

Dependencies between packages

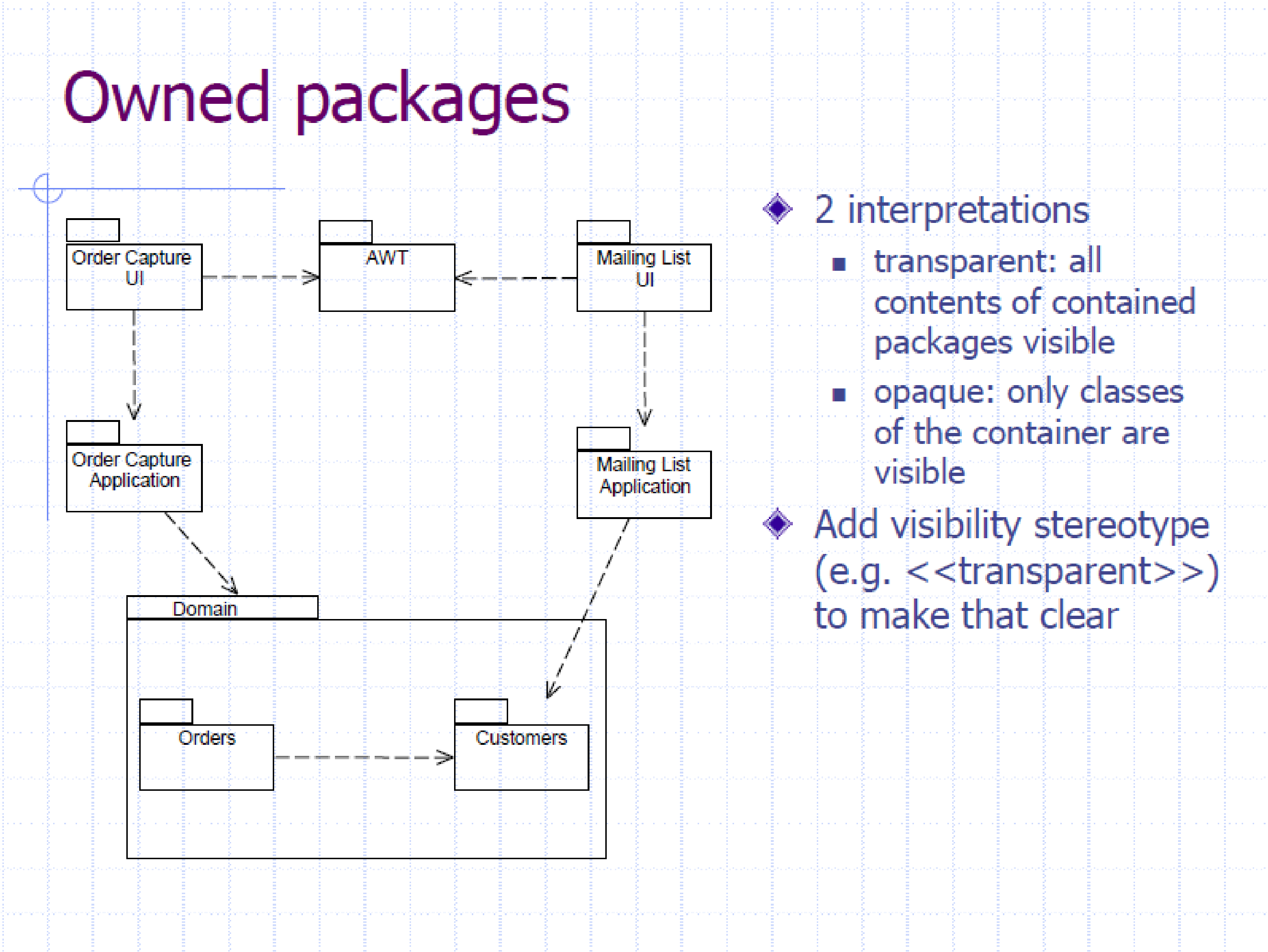


- Exist whenever there exist a dependency between classes of both packages
  - class A calls method of class B
  - class A has instance of B as part
  - method of A has parameter of type B
- Dependencies are not transitive (compilation dependencies are)



Owned elements

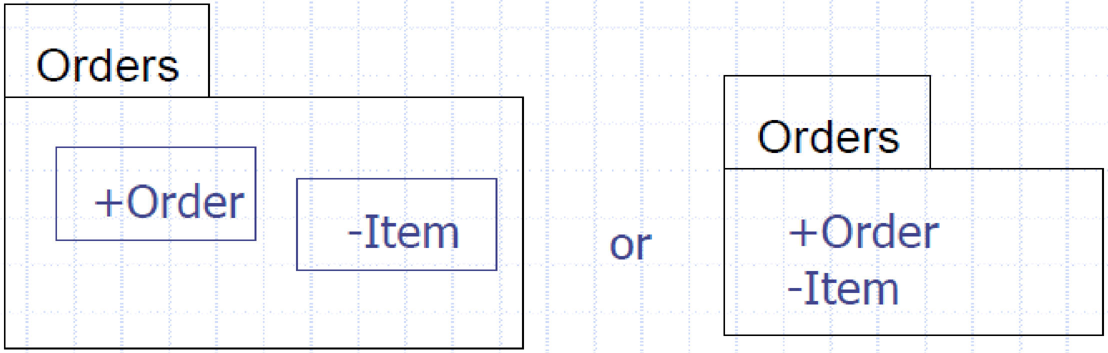
- Package may own other **packages**, **classes**, diagrams, …
- Hierarchical decomposition of the system



Owned packages

- 2 interpretations
  - transparent: all contents of contained packages visible
  - opaque: only classes of the container are visible
- Add visibility stereotype (e.g. <<transparent>>) to make that clear

Owned classes



Orders

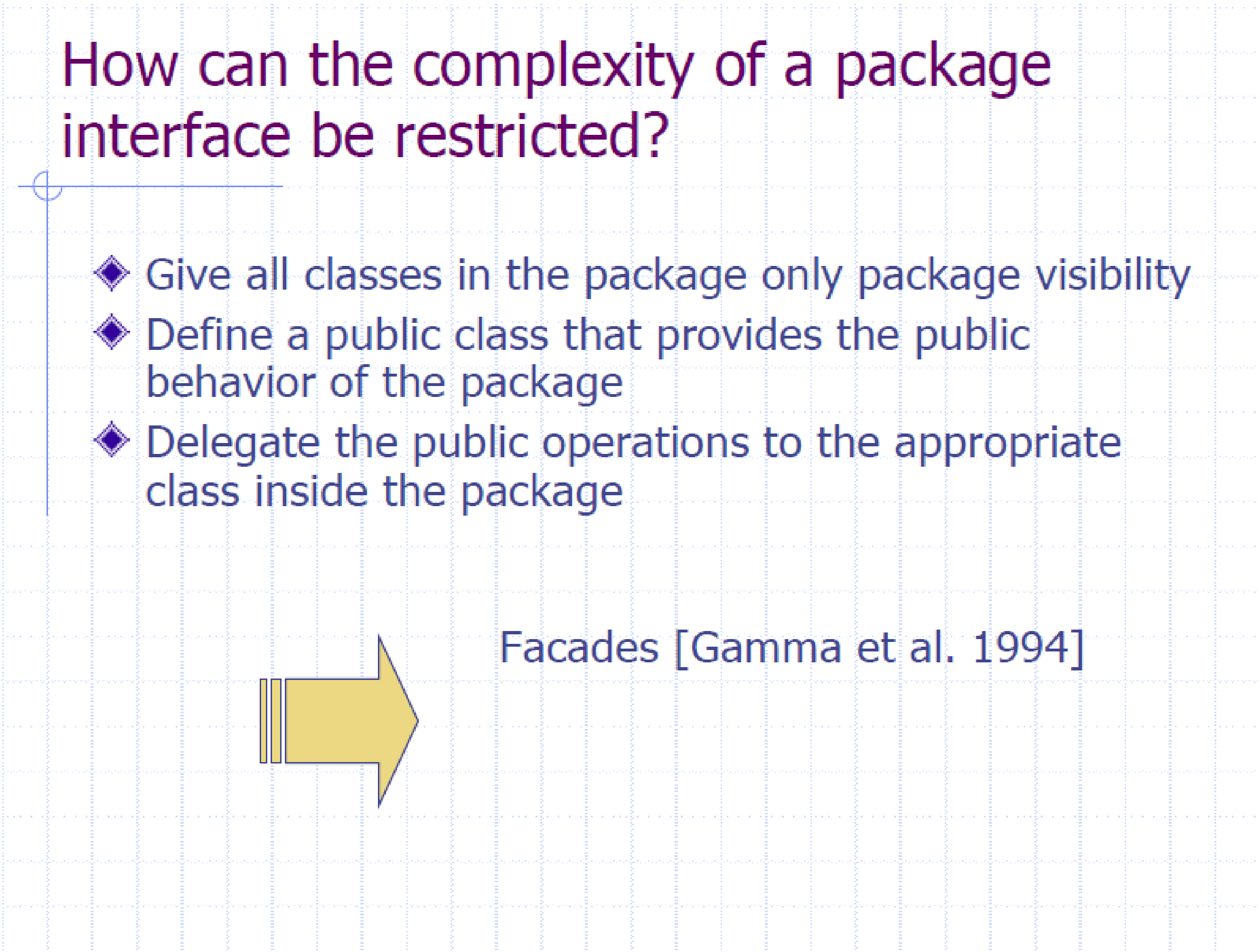
+Order

-Item

or

Orders

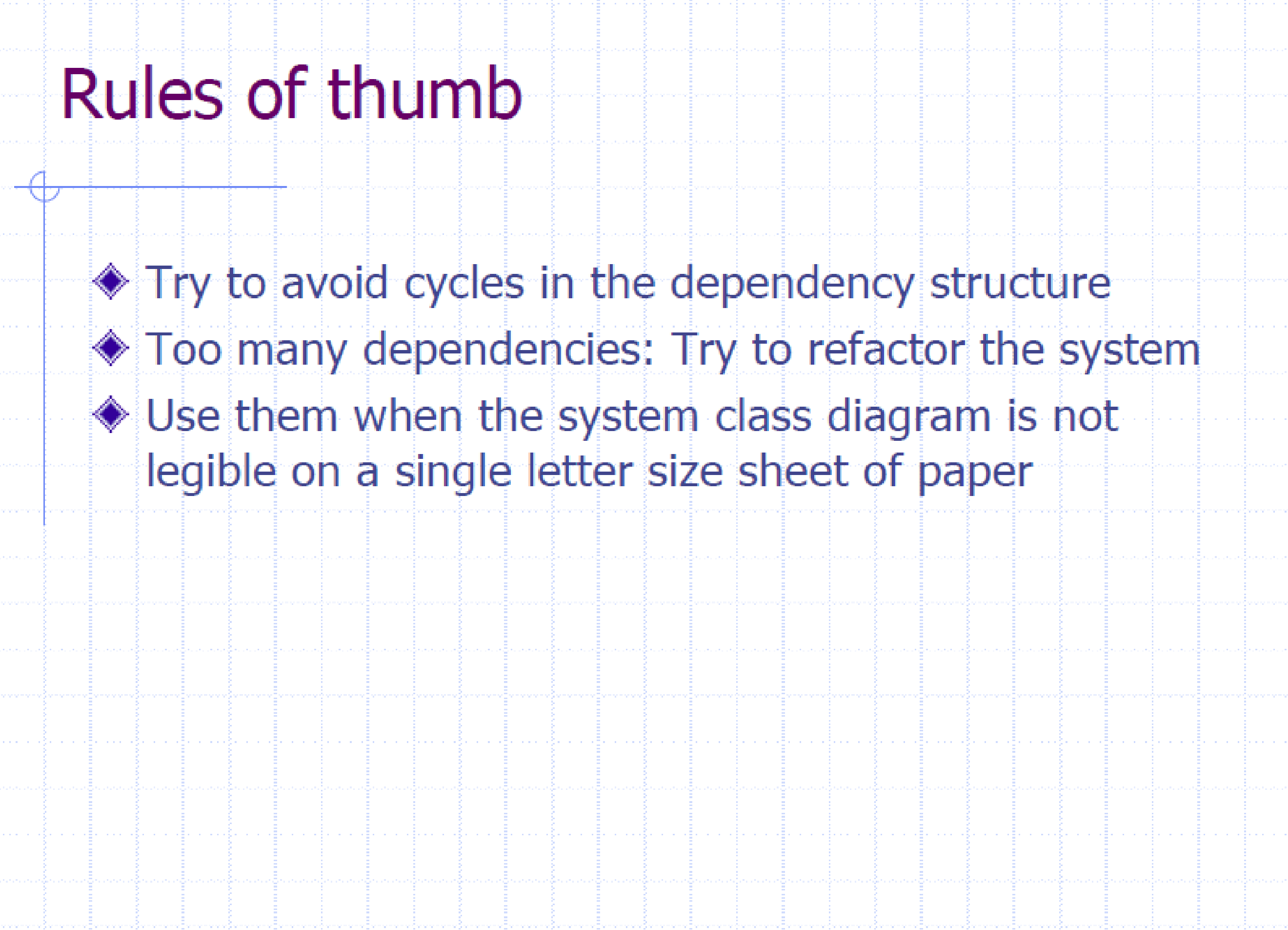
+Order
-Item



How can the complexity of a package interface be restricted?

- Give all classes in the package only package visibility
- Define a public class that provides the public behavior of the package
- Delegate the public operations to the appropriate class inside the package

Facades [Gamma et al. 1994]



Rules of thumb

- Try to avoid cycles in the dependency structure
- Too many dependencies: Try to refactor the system
- Use them when the system class diagram is not legible on a single letter size sheet of paper