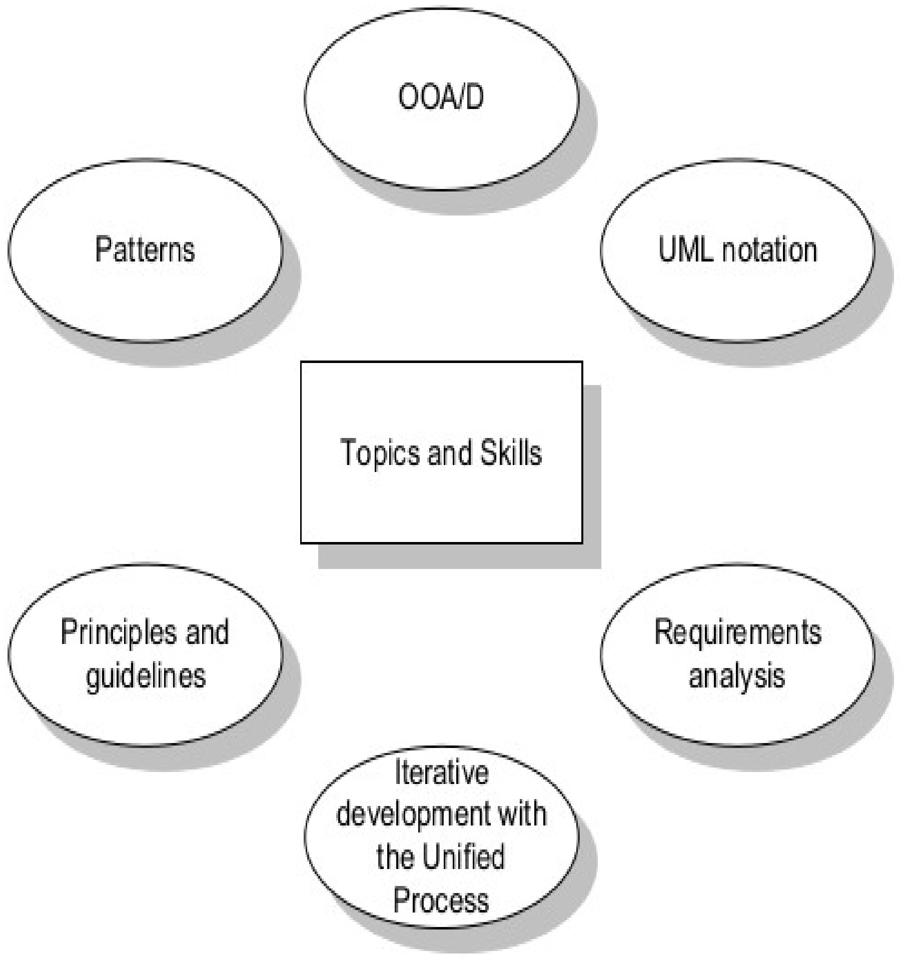
Introduction to OOAD

Rajeev Gupta

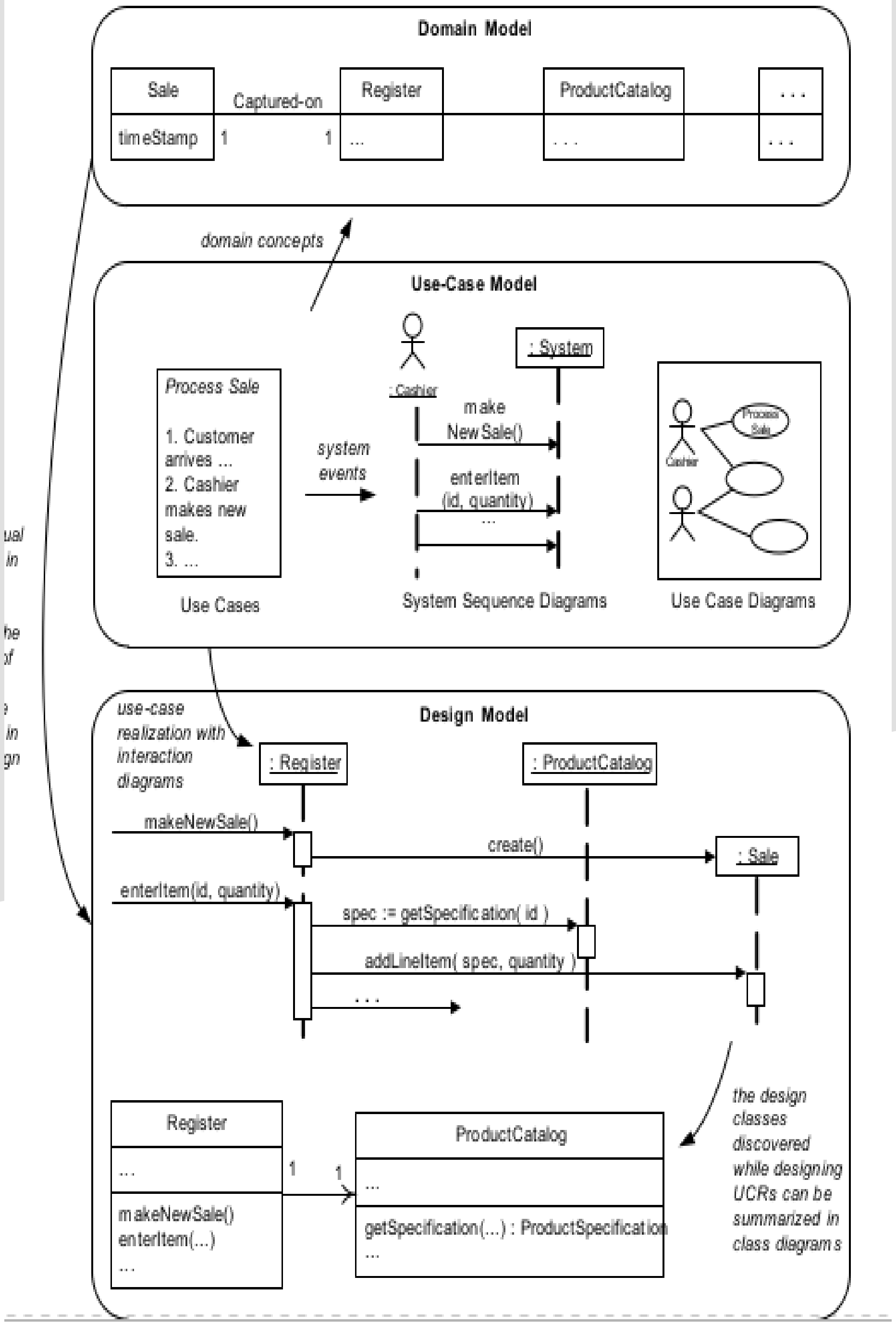
Mtech CS

rgutpa.mtech@gmail.com

Java Trainer & consultant


OOA/D
Patterns
UML notation
Topics and Skills
Principles and guidelines
Requirements analysis
Iterative development with the Unified Process

Sample Unified Process Artifact Relationships

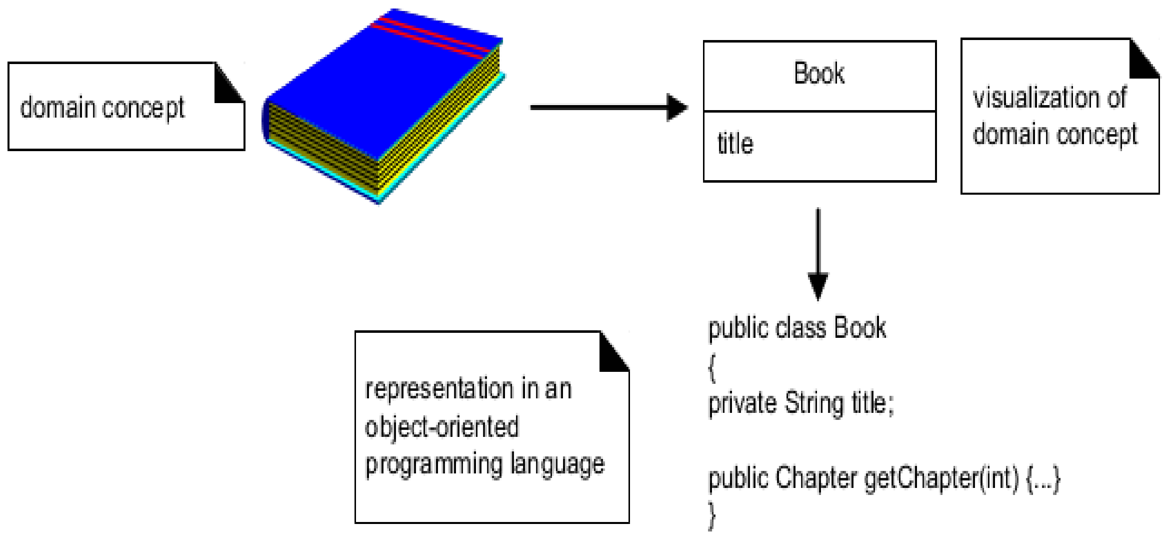


What Is Object-Oriented Analysis and Design?

During **object-oriented analysis,** there is an emphasis on finding and describing the objects—or concepts—in the problem domain. For example, in the case of the library information system, some of the concepts include *Book, Library,* and *Patron.*

During **object-oriented design,** there is an emphasis on defining software objects and how they collaborate to fulfill the requirements. For example, in the library system, a *Book* software object may have a *title* attribute and a *getChap-ter* method (see Figure 1.2).

Finally, during implementation or object-oriented programming, design objects are implemented, such as a *Book* class in Java.

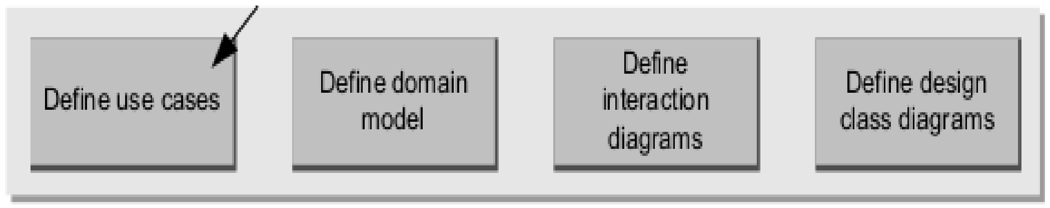

domain concept
Book
title
visualization of domain concept
representation in an object-oriented programming language

```
public class Book
{
private String title;

public Chapter getChapter(int) {...}
}
```

Define Use Cases

Requirements analysis may include a description of related domain processes; these can be written as **use cases.**

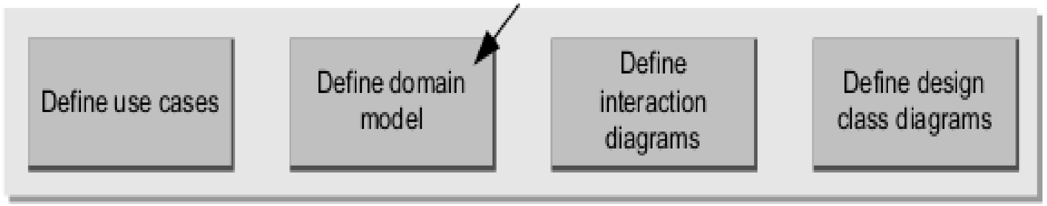


Use cases are not an object-oriented artifact—they are simply written stories. However, they are a popular tool in requirements analysis and are an important part of the Unified Process. For example, here is a brief version of the *Play a Dice Game* use case:

> **Play a Dice Game:** A player picks up and rolls the dice. If the dice face value total seven, they win; otherwise, they lose.

Define a Domain Model

Object-oriented analysis is concerned with creating a description of the domain from the perspective of classification by objects. A decomposition of the domain involves an identification of the concepts, attributes, and associations that are considered noteworthy. The result can be expressed in a **domain model,** which is illustrated in a set of diagrams that show domain concepts or objects.



For example, a partial domain model is shown in Figure 1.3.

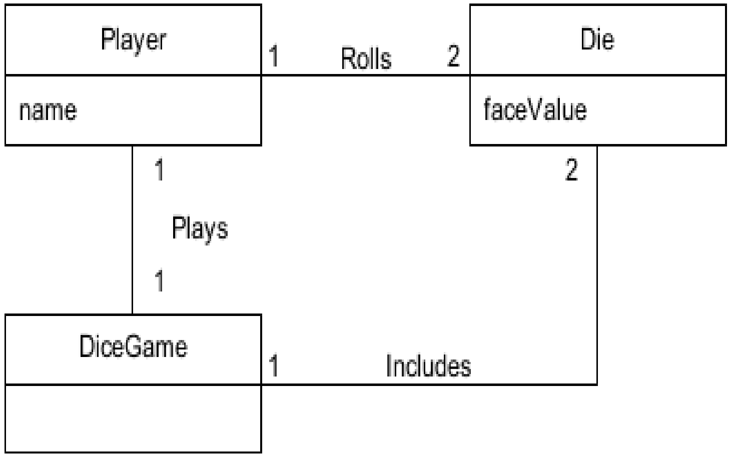
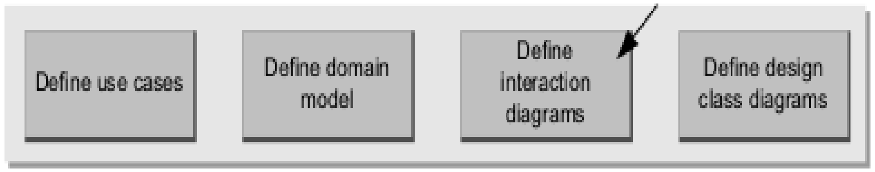


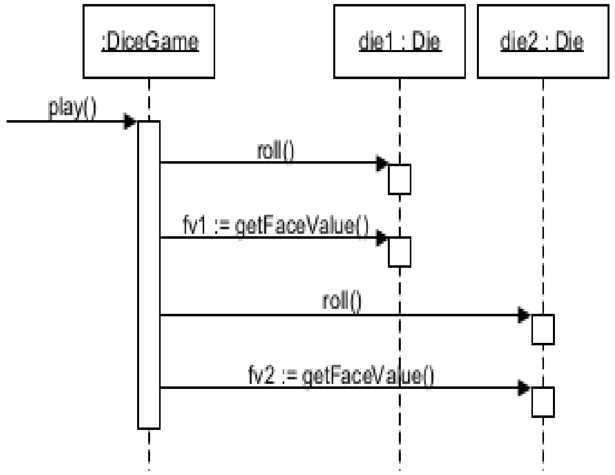
Figure 1.3 Partial domain model of the dice game.

Define Interaction Diagrams

Object-oriented design is concerned with defining software objects and their collaborations. A common notation to illustrate these collaborations is the **interaction diagram.** It shows the flow of messages between software objects, and thus the invocation of methods.

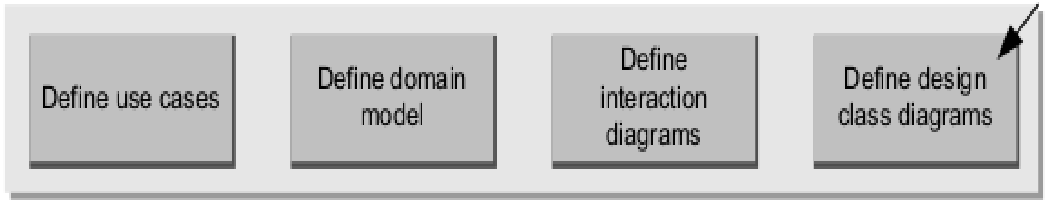


For example, assume that a software implementation of the dice game is desired. The interaction diagram in Figure 1.4 illustrates the essential step of playing, by sending messages to instances of the *DiceGame* and *Die* classes.



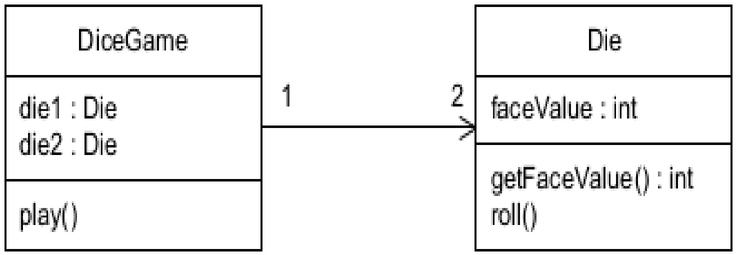
Define Design Class Diagrams

In addition to a *dynamic* view of collaborating objects shown in interaction diagrams, it is useful to create a *static* view of the class definitions with a **design class diagram.** This illustrates the attributes and methods of the classes.



For example, in the dice game, an inspection of the interaction diagram leads to the partial design class diagram shown in Figure 1.5. Since a *play* message is sent to a *DiceGame* object, the *DiceGame* class requires a *play* method, while class *Die* requires a *roll* and *getFaceValue* method.

In contrast to the domain model, this diagram does not illustrate real-world concepts; rather, it shows software classes.



ITERATIVE DEVELOPMENT AND THE UNIFIED PROCESS

People are more important than any process.

Good people with a good process will outperform good people with no process every time.

—Grady Booch

Early iterative process ideas were known as spiral development and evolutionary development [Boehm.88, Gilb88].

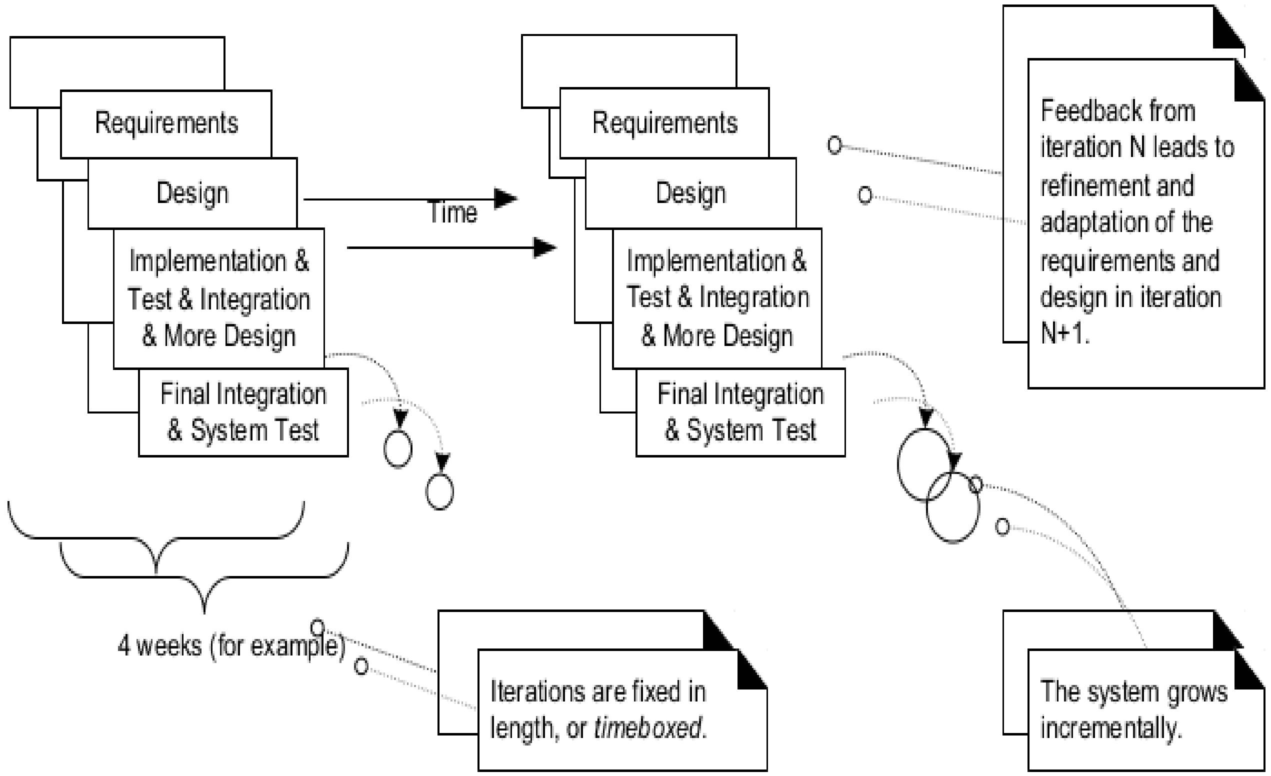
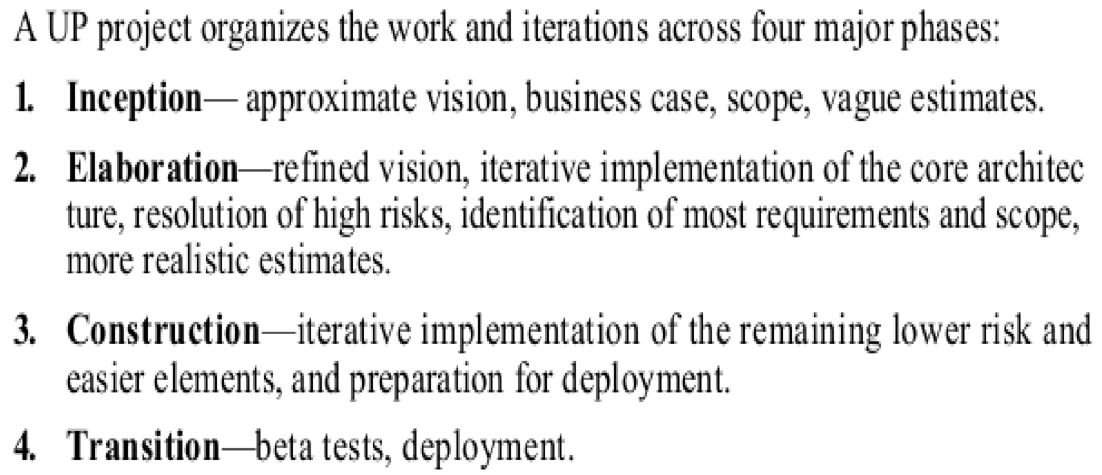

Requirements
Design
Implementation & Test & Integration & More Design
Final Integration & System Test
Time
Requirements
Design
Implementation & Test & Integration & More Design
Final Integration & System Test
Feedback from iteration N leads to refinement and adaptation of the requirements and design in iteration N+1.
4 weeks (for example)
Iterations are fixed in length, or *timeboxed*.
The system grows incrementally.

Figure 2.1 Iterative and incremental development.

The UP Phases and Schedule-Oriented Terms



A UP project organizes the work and iterations across four major phases:

1. **Inception**— approximate vision, business case, scope, vague estimates.
2. **Elaboration**—refined vision, iterative implementation of the core architec ture, resolution of high risks, identification of most requirements and scope, more realistic estimates.
3. **Construction**—iterative implementation of the remaining lower risk and easier elements, and preparation for deployment.
4. **Transition**—beta tests, deployment.

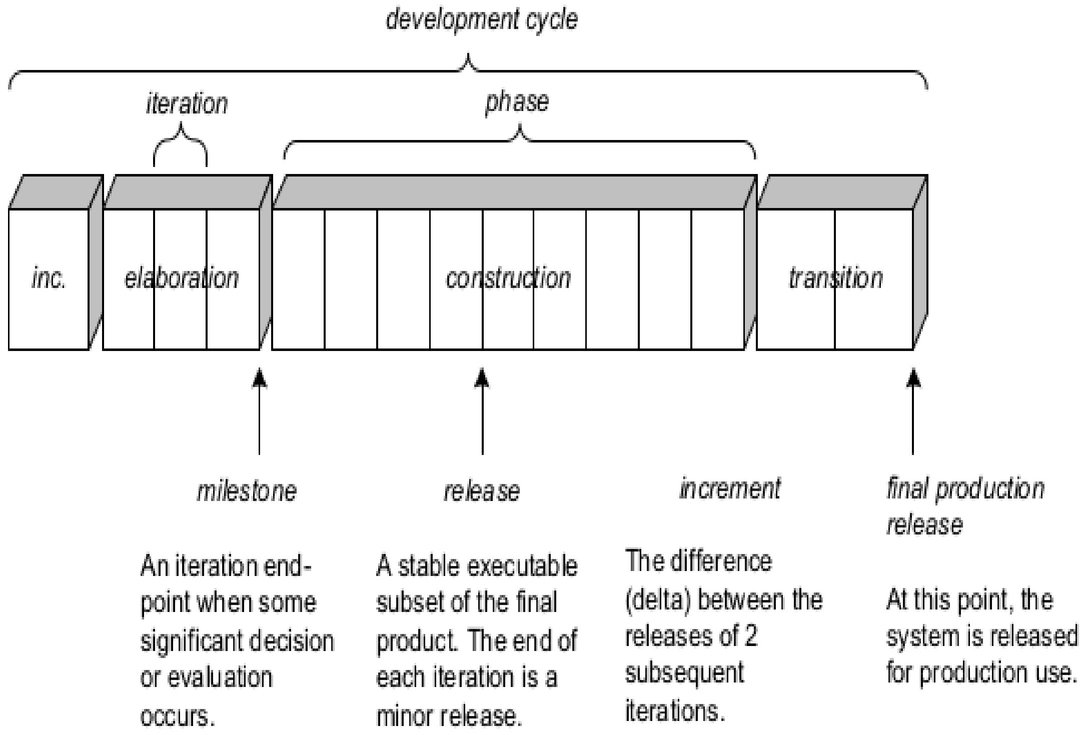

development cycle
iteration
phase
inc.
elaboration
construction
transition
milestone
release
increment
final production release
An iteration end-point when some significant decision or evaluation occurs.
A stable executable subset of the final product. The end of each iteration is a minor release.
The difference (delta) between the releases of 2 subsequent iterations.
At this point, the system is released for production use.

Figure 2.3 Schedule-oriented terms in the UP.

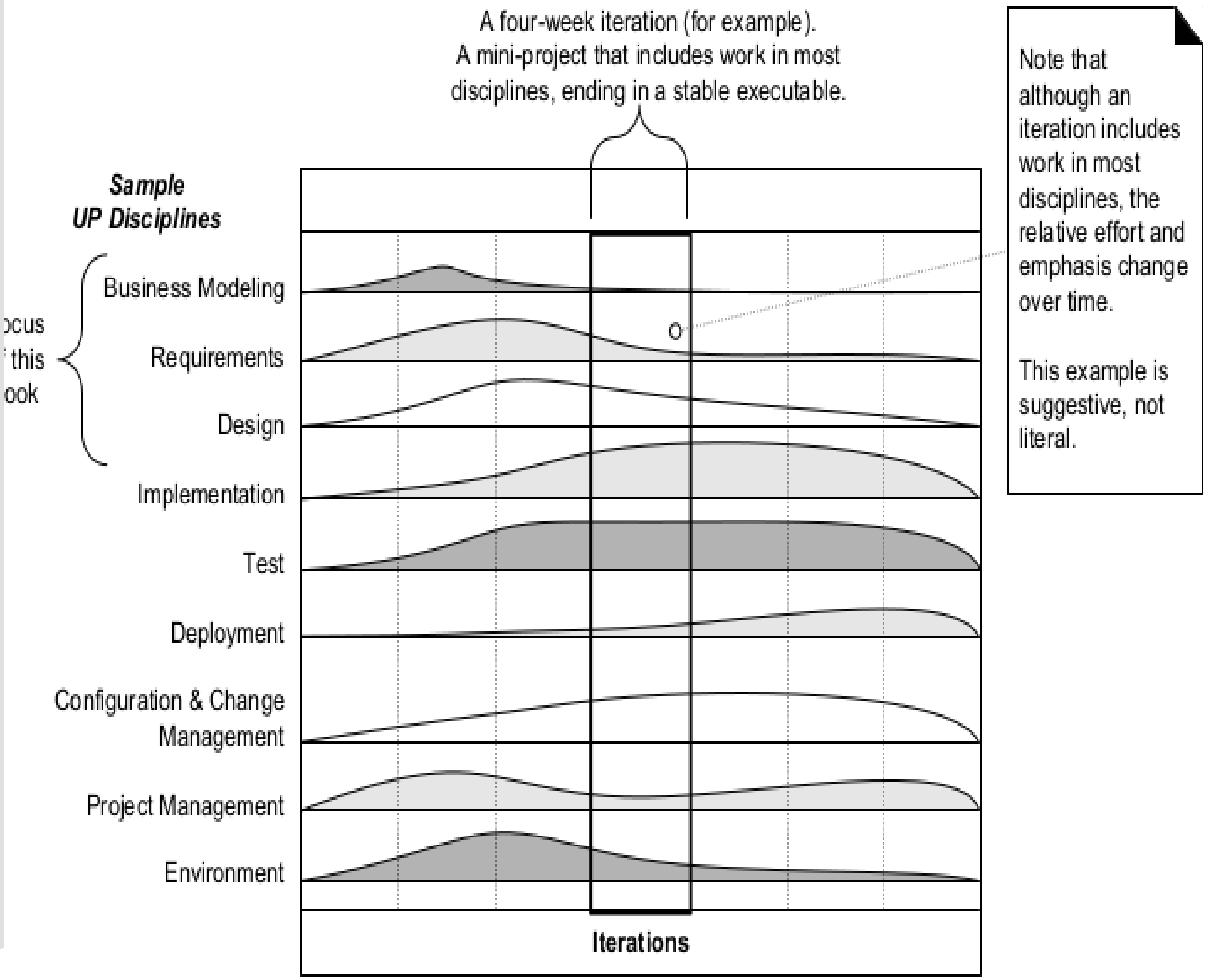

A four-week iteration (for example).
A mini-project that includes work in most disciplines, ending in a stable executable.
Sample UP Disciplines
ocus
f this
ook
Business Modeling
Requirements
Design
Implementation
Test
Deployment
Configuration & Change Management
Project Management
Environment
Iterations
Note that although an iteration includes work in most disciplines, the relative effort and emphasis change over time.
This example is suggestive, not literal.

Figure 2.4 UP disciplines.[4]

In the UP, **Implementation** means programming and building the system, not deployment. The **Environment** discipline refers to establishing the tools and customizing the process for the project—that is, setting up the tool and process environment.

relatively high level of requirements and design work, although definitely some implementation as well. During construction, the emphasis is heavier on implementation and lighter on requirements analysis.

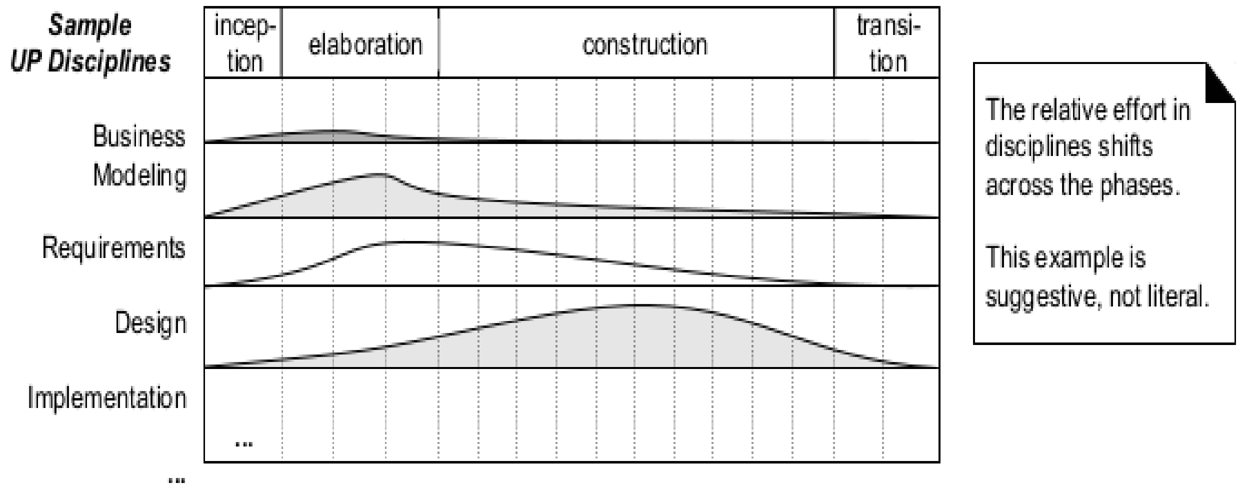
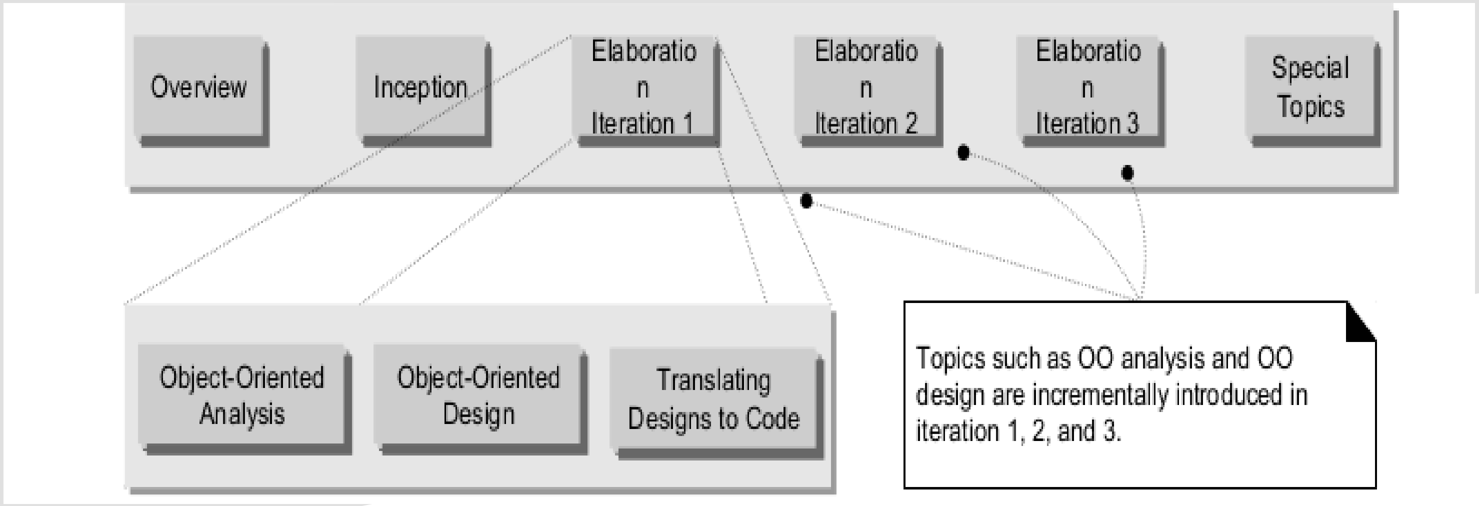


Figure 2.5 Disciplines and phases


Overview
Inception
Elaboratio n Iteration 1
Elaboratio n Iteration 2
Elaboratio n Iteration 3
Special Topics
Object-Oriented Analysis
Object-Oriented Design
Translating Designs to Code
Topics such as OO analysis and OO design are incrementally introduced in iteration 1, 2, and 3.

UNDERSTANDING REQUIREMENTS

Fast, Cheap, Good: Choose any two.

—anonymous

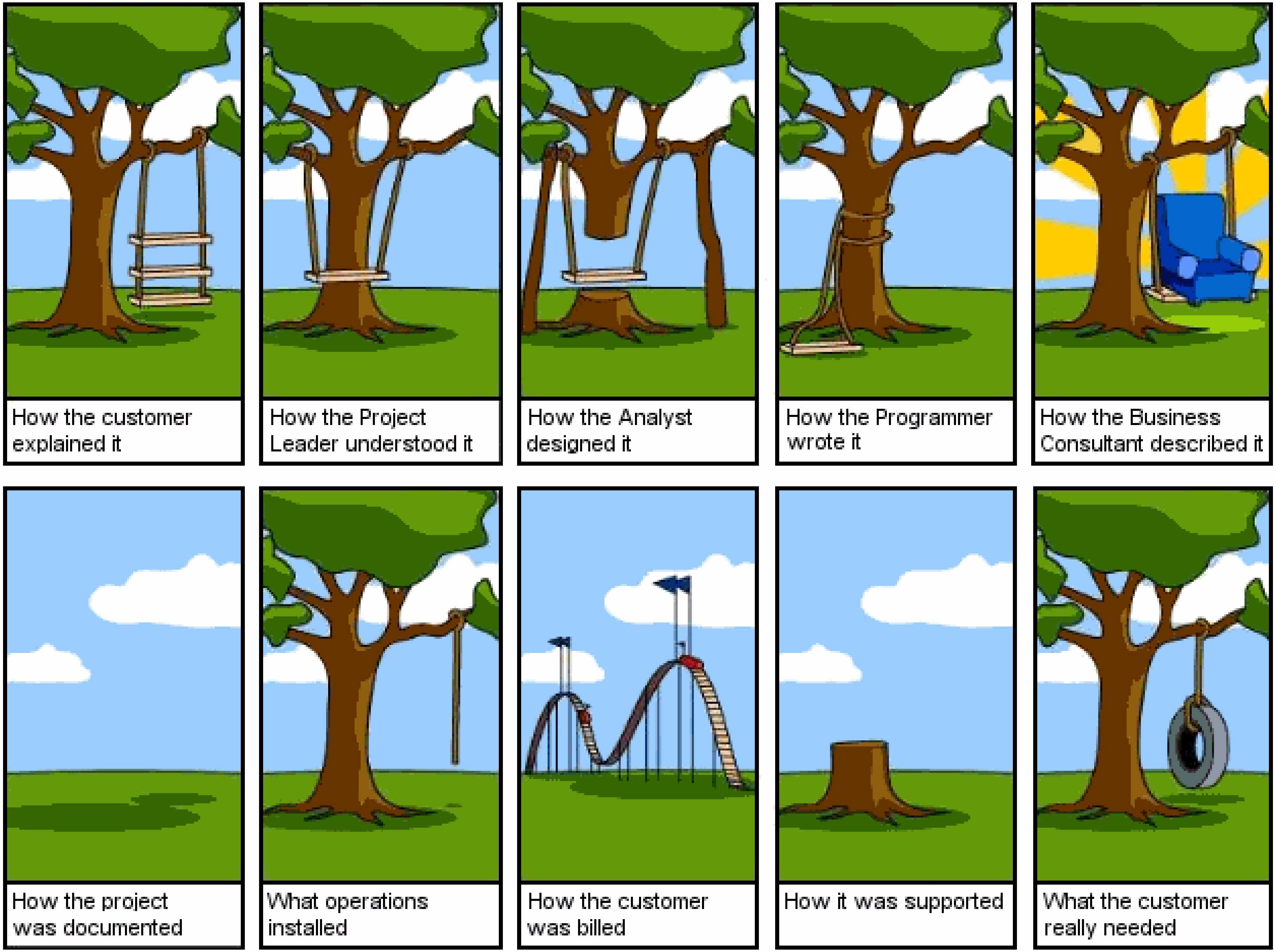
Types of Requirements

In the UP, requirements are categorized according to the FURPS+ model [Grady92], a useful mnemonic with the following meaning:[1]

- **Functional**—features, capabilities, security.
- **Usability**—human factors, help, documentation.
- **Reliability**—frequency of failure, recoverability, predictability.
- **Performance**—response times, throughput, accuracy, availability, resource usage.
- **Supportability**—adaptability, maintainability, internationalization, con
figurability.

The "+" in FURPS+ indicates ancillary and sub-factors, such as:

- **Implementation**—resource limitations, languages and tools, hardware, ...
- **Interface**—constraints imposed by interfacing with external systems.
- **Operations**—system management in its operational setting.
- **Packaging**
- **Legal**—licensing and so forth.


How the customer explained it
How the Project Leader understood it
How the Analyst designed it
How the Programmer wrote it
How the Business Consultant described it
How the project was documented
What operations installed
How the customer was billed
How it was supported
What the customer really needed

What is OOAD?

Analysis — understanding, finding and describing concepts in the problem domain.

Traceability!

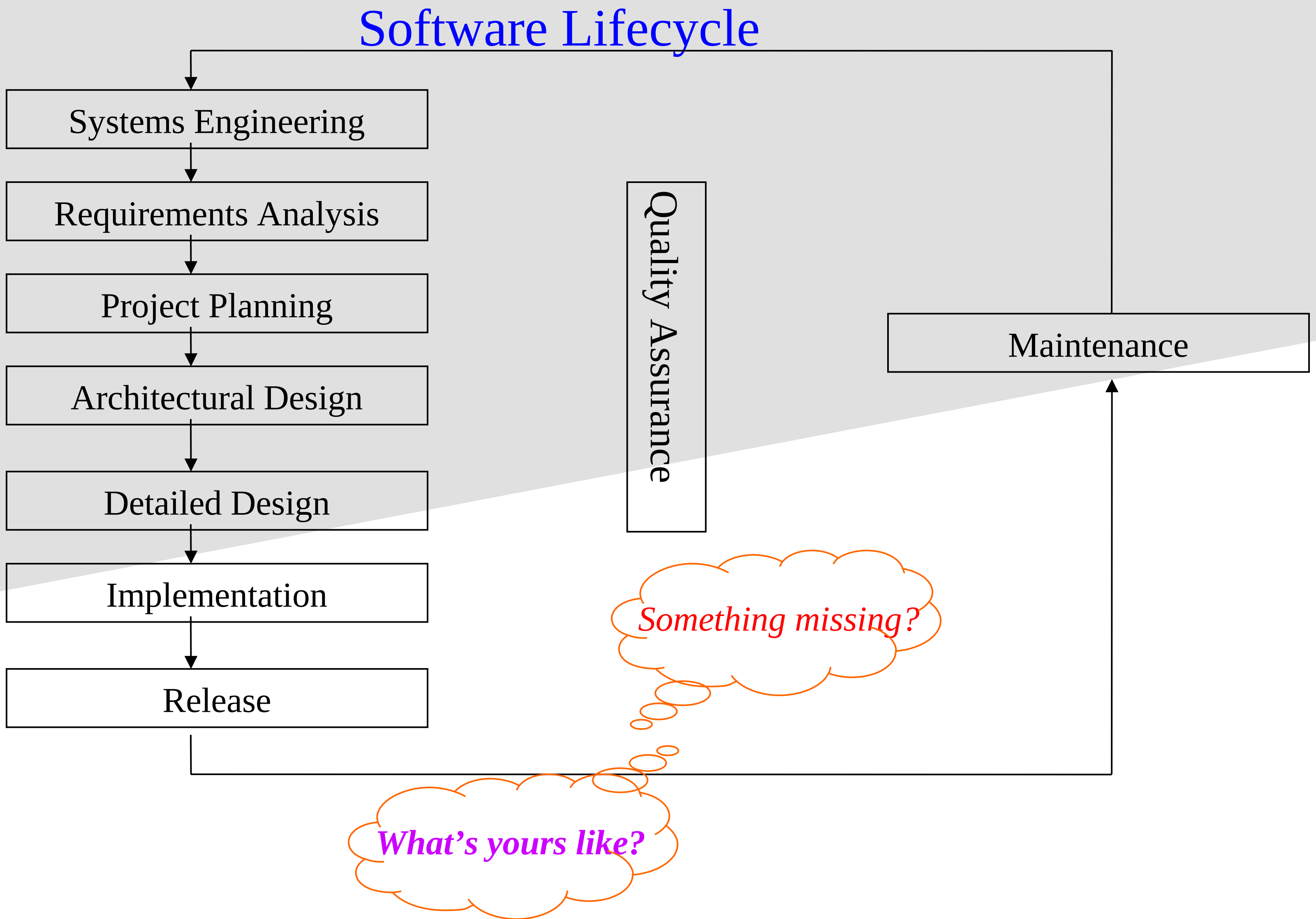
Design — understanding and defining software solution/objects that *represent* the analysis concepts and will eventually be implemented in code.

OOAD — Analysis is object-oriented and design is object-oriented. A software development approach that emphasizes a logical solution based on objects.

Involves both a notation and a process

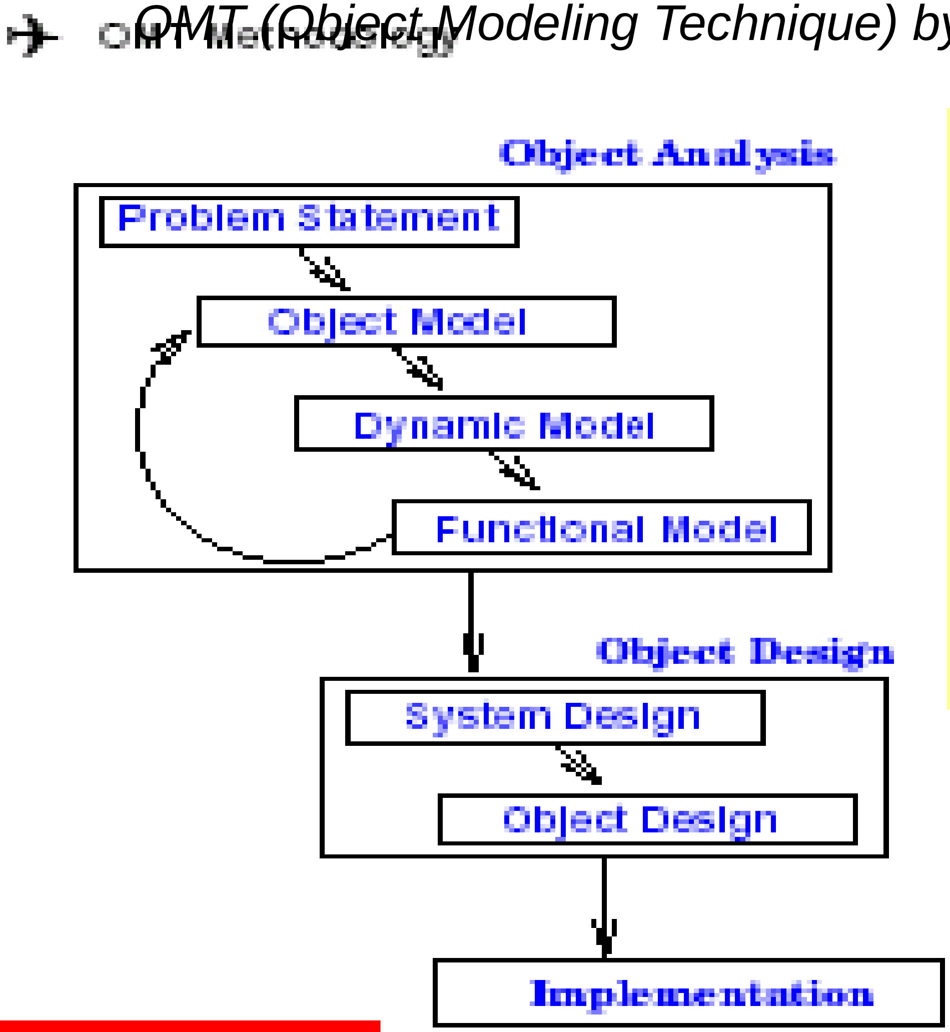
How to Do OOAD
– Where to Use OO?

Traceability!



How to Do OOAD
– OMT as Object-Oriented Methodology

OMT (Object Modeling Technique) by James Rumbaugh



Object Model: describes the ***static*** structure of the objects in the system and their relationships -> Object Diagrams.

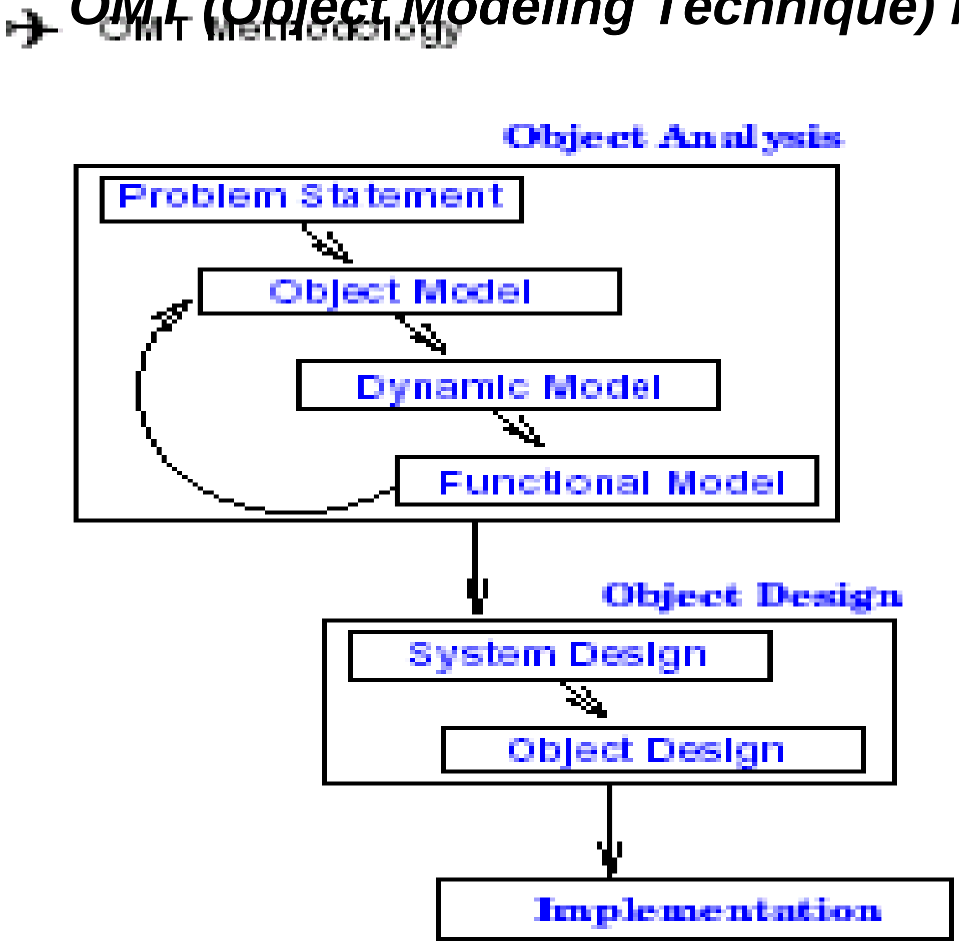
Dynamic Model: describes the ***interactions*** among objects in the system -> State Diagrams.

Functional Model: describes the data ***transformation*** of the system -> DataFlow Diagrams.

Traceability!

How to Do OOAD
– OMT as Object-Oriented Methodology

OMT (Object Modeling Technique) by James Rumbaugh



Analysis:

i) Model the ***real world*** showing its important properties;

ii) Concise model of what the ***system*** will do

System Design:
Organize into subsystems based on analysis structure and propose ***architecture***

Object Design: Based on analysis model but with implementation details; Focus on ***data structures and algorithms*** to implement each class; Computer and domain objects

Implementation: Translate the object classes and relationships into a ***programming*** language

Traceability!

Module 2:
Introduction to UML

- Background
- What is UML for?
- Building blocks of UML
- Appendix: Process for Using UML

UML History

- OO languages appear mid 70's to late 80's (cf. Budd: communication and complexity)
- Between '89 and '94, OO methods increased from 10 to 50.
- Unification of ideas began in mid 90's.
  - Rumbaugh joins Booch at Rational '94
    - v0.8 draft Unified Method '95
  - Jacobson joins Rational '95
    - UML v0.9 in June '96

  preUML

    - UML 1.0 offered to OMG in January '97
    - UML 1.1 offered to OMG in July '97
  - Maintenance through OMG RTF
    - UML 1.2 in June '98
    - UML 1.3 in fall '99
    - UML 1.5 http://www.omg.org/technology/documents/formal/uml.htm

  UML 1.x

    - UML 2.0 underway http://www.uml.org/

  UML 2.0

- IBM-Rational now has *Three Amigos*
  - Grady Booch - Fusion
  - James Rumbaugh – Object Modeling Technique (OMT)
  - Ivar Jacobson – Object-oriented Software Engineering: A Use Case Approach (Objectory)
  - (And David Harel - StateChart)
- Rational Rose http://www-306.ibm.com/software/rational/

Unified Modeling Language (UML)

- An effort by IBM (Rational) – OMG to standardize OOA&D notation

- Combine the best of the best from
  - Data Modeling (Entity Relationship Diagrams); Business Modeling (work flow); Object Modeling
  - Component Modeling (development and reuse - middleware, COTS/GOTS/OSS/…:)

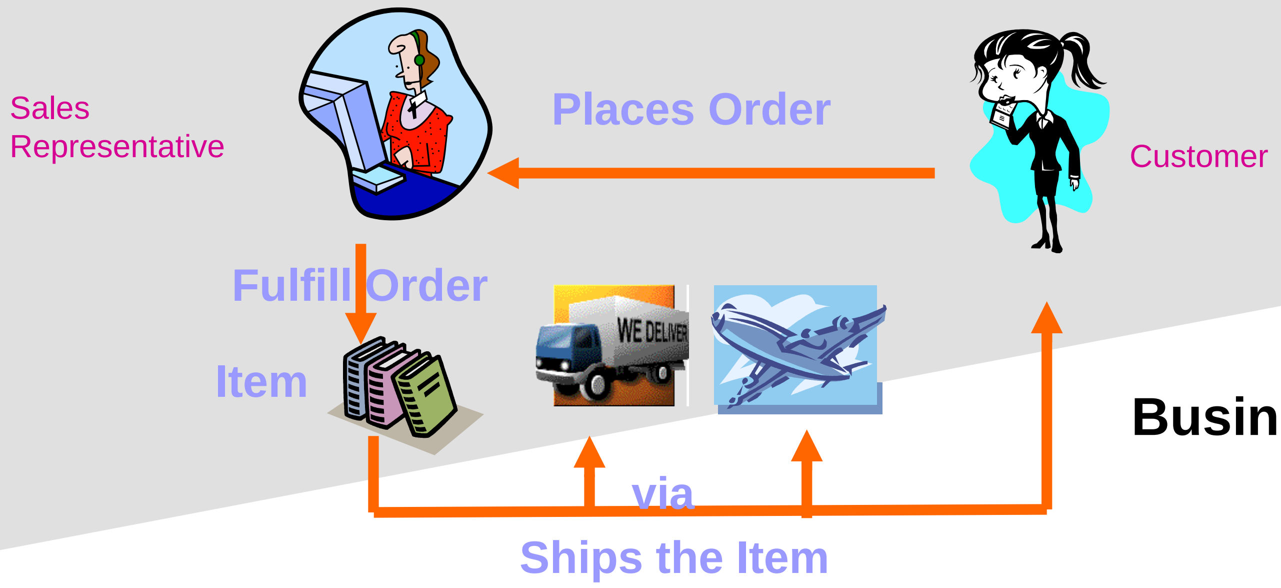
- Offers vocabulary and rules for communication
- *Not* a process but a language

de facto industry standard

UML is for Visual Modeling

A picture is worth a thousand words!

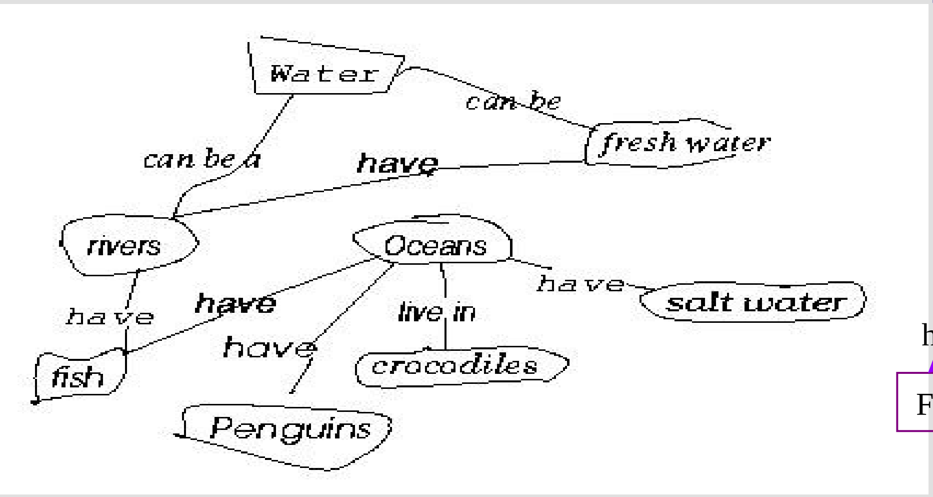
- standard graphical notations: Semi-formal
- for modeling enterprise info. systems, distributed Web-based applications, real time embedded systems, …

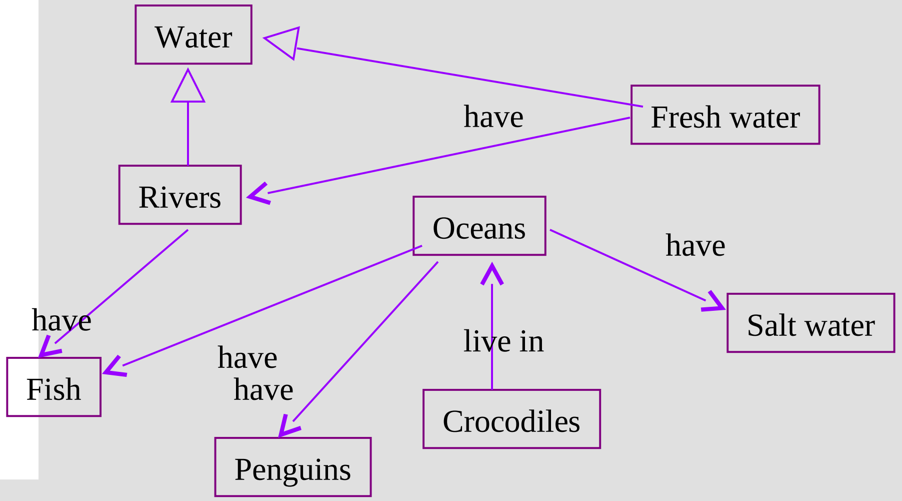


Business Process

- ***Specifying & Documenting:*** models that are precise, unambiguous, complete
  - ☐ UML symbols are based on well-defined syntax and semantics.
  - ☐ analysis, architecture/design, implementation, testing decisions.
- ***Construction:*** mapping between a UML model and OOPL.

Three (3) basic *building blocks* of UML (cf. Harry)





– Things - important modeling concepts

– Relationships - tying individual things

Just glance thru for now

– Diagrams - grouping interrelated collections of things and relationships

3 basic building blocks of UML - Things

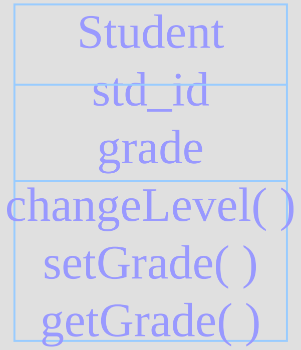
- UML 1.x
  - Structural — nouns/static of UML models (irrespective of time).
  - Behavioral — verbs/dynamic parts of UML models.
  - Grouping — organizational parts of UML models.
  - Annotational — explanatory parts of UML models.

Main

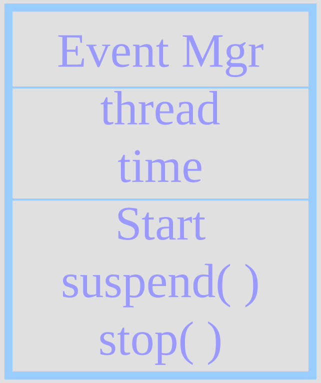
Structural Things in UML- *7 Kinds* *(Classifiers)*

- Nouns.
- Conceptual or physical elements.

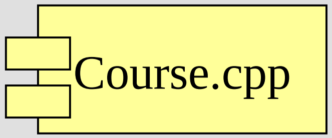
Class



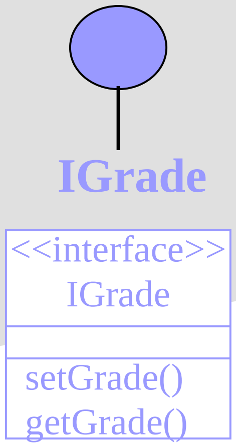
Active Class
(processes/threads)



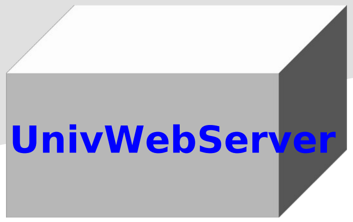
Component
(replaceable part, realizes interfaces)

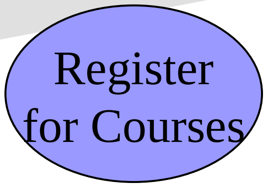


Interface
(collection of externally Visible ops)



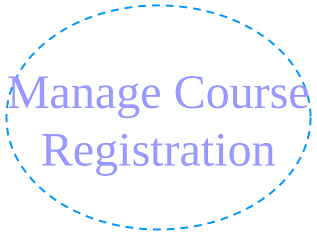
Node
(computational resource at run-time, processing power w. memory)





Use Case

(a system service -sequence of Interactions w. actor)



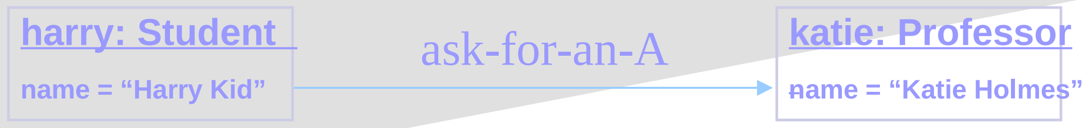
Collaboration
(chain of responsibility shared by a web of interacting objects, structural and behavioral)

Behavioral Things in UML

- Verbs.
- Dynamic parts of UML models: “behavior over time”
- Usually connected to structural things.

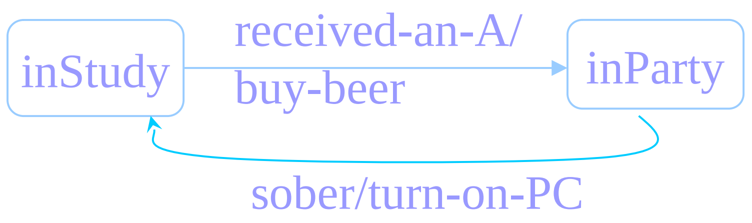
- Two primary kinds of behavioral things:
  - ***Interaction***

    a set of objects exchanging messages, to accomplish a specific purpose.



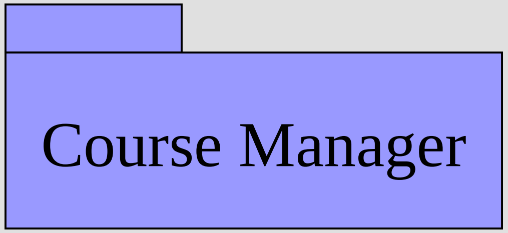
  - ***State Machine***

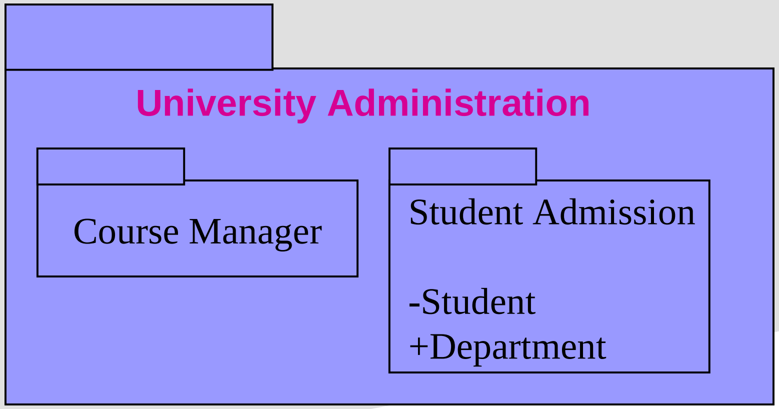
    specifies the sequence of states an object or an interaction goes through during its lifetime in response to events.



Grouping Things in UML: *Packages*

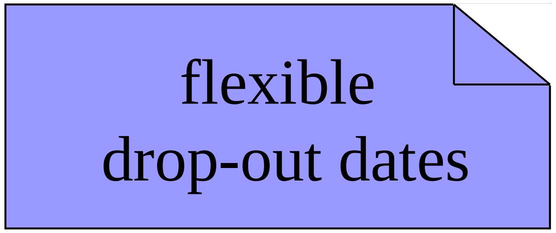
- For organizing elements (structural/behavioral) into groups.
- Purely conceptual; only exists at development time.
- Can be nested.
- Variations of packages are: Frameworks, models, & subsystems.

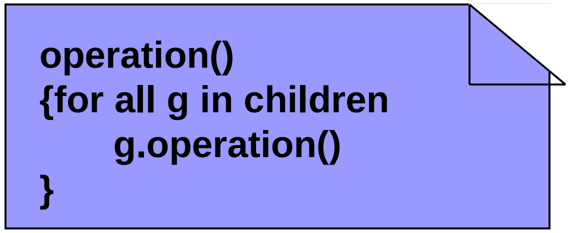




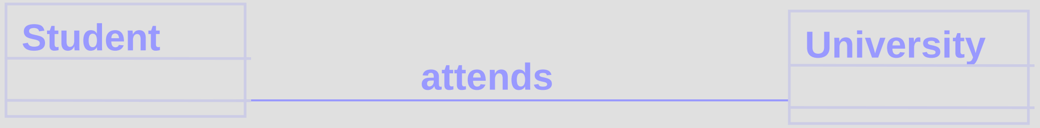
Annotational Things in UML: *Note*

- Explanatory/Comment parts of UML models - usually called adornments
- Expressed in informal or formal text.



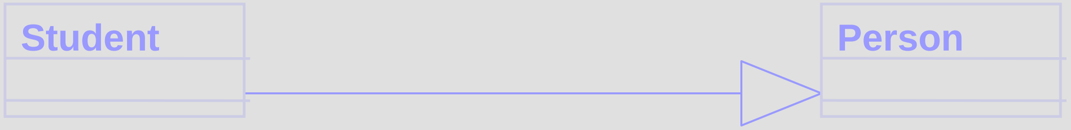


3 basic building blocks of UML - Relationships



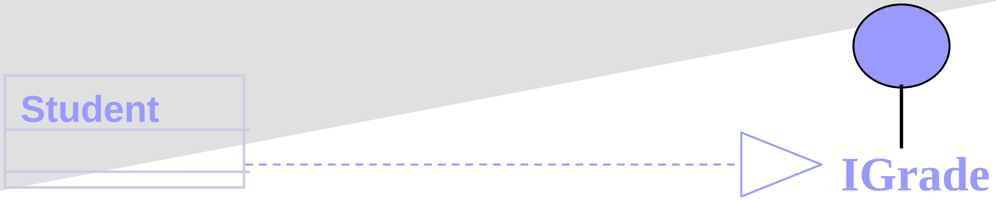
1. *Associations*

Structural relationship that describes a set of links, a link being a connection between objects. *variants: aggregation & composition*



2. *Generalization*

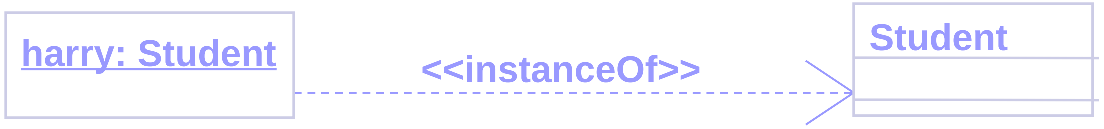
a specialized element (the child) is more specific the generalized element.



3. *Realization*

one element guarantees to carry out what is expected by the other element.

(e.g, interfaces and classes/components; use cases and collaborations)



4. *Dependency*

a change to one thing (independent) may affect the semantics of the other thing (dependent).

(direction, label are optional)

3 basic building blocks of UML - Diagrams

A connected graph: Vertices are things; Arcs are relationships/behaviors.

UML 1.x: 9 diagram types.

Structural Diagrams

Represent the static aspects of a system.

- Class;
  Object
- Component
- Deployment

Behavioral Diagrams

Represent the dynamic aspects.

- Use case
- Sequence;
  Collaboration
- Statechart
- Activity

UML 2.0: 12 diagram types

Structural Diagrams

- Class;
  Object
- Component
- Deployment
- Composite Structure
- *Package*

Behavioral Diagrams

- Use case
- Statechart
- Activity

Interaction Diagrams

- Sequence;
  Communication
- Interaction Overview
- Timing

Diagrams in UML

The UTD wants to computerize its registration system

- The Registrar sets up the curriculum for a semester
- Students select 3 core courses and 2 electives
- Once a student registers for a semester, the billing system is notified so the student may be billed for the semester
- Students may use the system to add/drop courses for a period of time after registration
- Professors use the system to set their preferred course offerings and receive their course offering rosters after students register
- Users of the registration system are assigned passwords which are used at logon validation

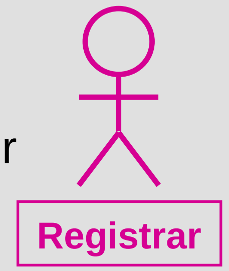
What's most important?

Diagrams in UML – Actors in Use Case Diagram

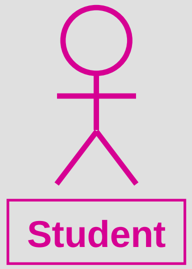
- An actor is someone or some thing that must interact with the system under development

The UTD wants to computerize its registration system

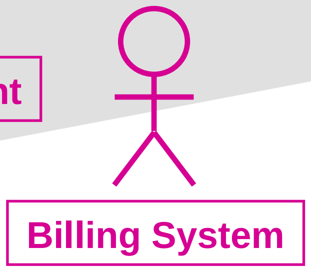
- The **Registrar** sets up the curriculum for a semester



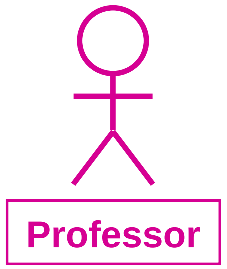
- **Students** select 3 core courses and 2 electives



- Once a student registers for a semester, the **billing system** is notified so the student may be billed for the semester



- Students may use the system to add/drop courses for a period of time after registration
- **Professors** use the system to set their preferred course offerings and receive their course offering rosters after students register



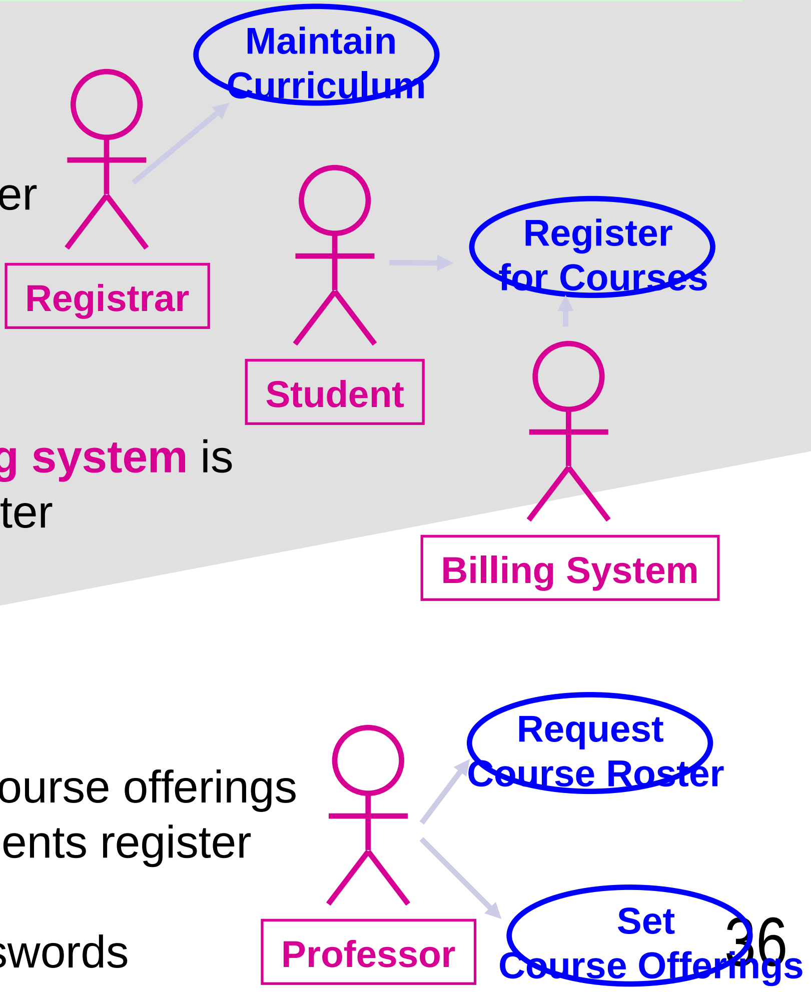
- Users of the registration system are assigned passwords which are used at logon validation

Diagrams in UML – Use Cases in Use Case Diagram

- A use case is a sequence of interactions between an actor and the system

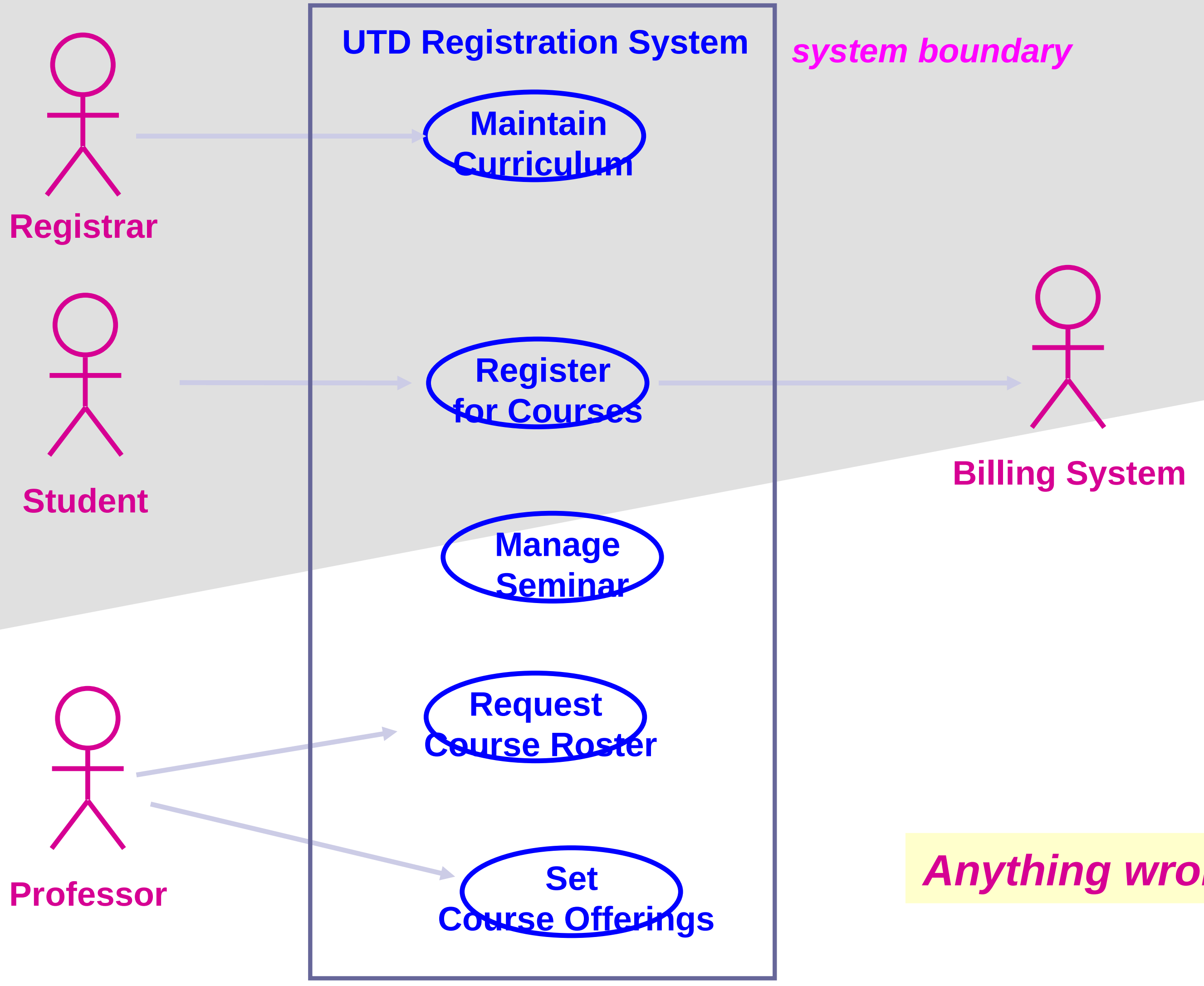
The UTD wants to computerize its registration system

- The **Registrar** sets up the curriculum for a semester
- **Students** select 3 core courses and 2 electives
- Once a student registers for a semester, the **billing system** is notified so the student may be billed for the semester
- Students may use the system to add/drop courses for a period of time after registration
- **Professor**s use the system to set their preferred course offerings and receive their course offering rosters after students register
- Users of the registration system are assigned passwords which are used at logon validation



Diagrams in UML – Use Case Diagram

- Use case diagrams depict the relationships between actors and use cases

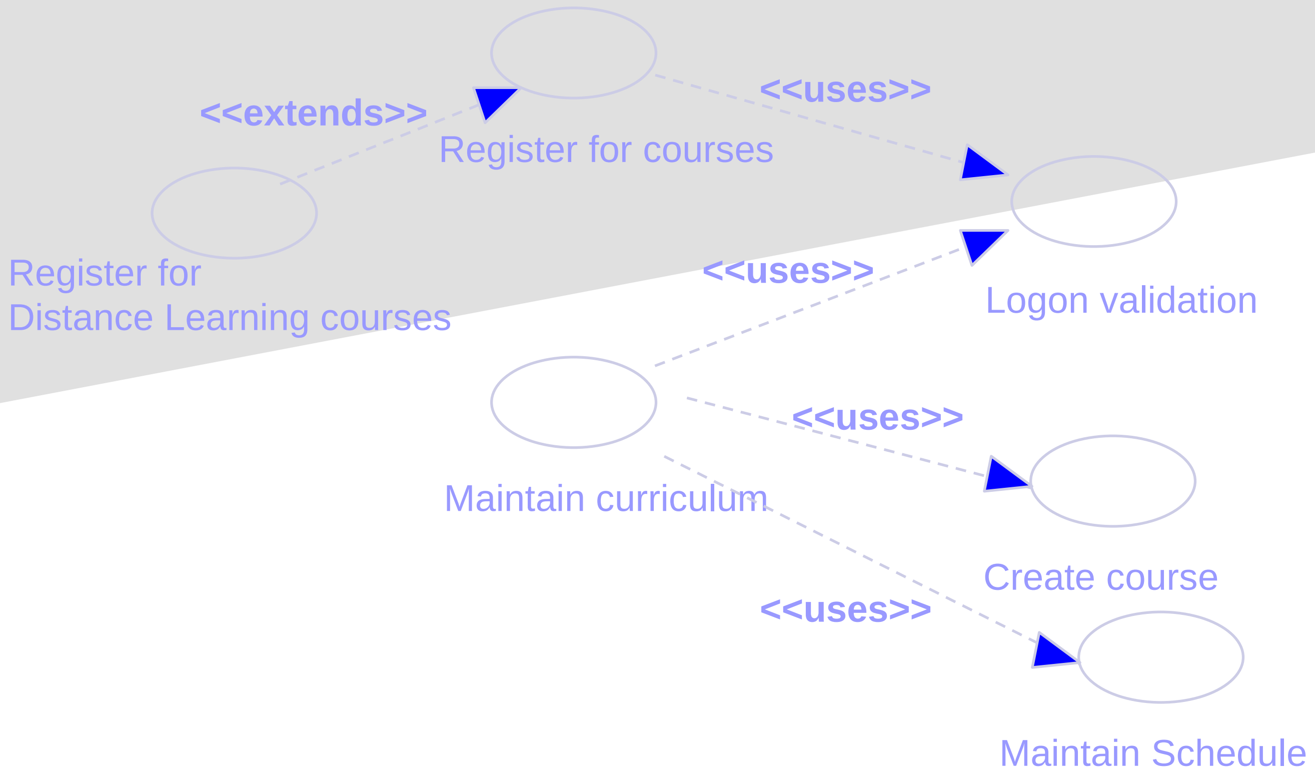


Anything wrong?

Diagrams in UML - Uses and Extends in Use Case Diagram

A uses relationship shows behavior common to one or more use cases

An extends relationship shows optional/exceptional behavior



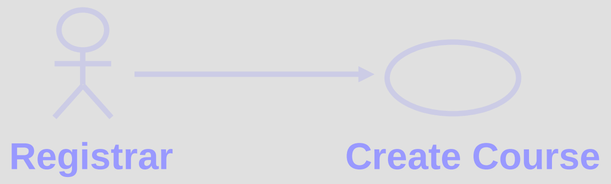
Diagrams in UML – Flow of Events for each use case:

Typical contents:

How the use case starts and ends

Normal flow of events (focus on the normal first!)

Alternate/Exceptional flow of events



Flow of Events for Creating a Course

- This use case begins after the Registrar logs onto the Registration System with a valid password.
- The registrar fills in the course form with the appropriate semester and course related info.
- The Registrar requests the system to process the course form.
- The system creates a new course, and this use case ends

Diagrams in UML – Interaction Diagrams

*A use case diagram presents an **outside** view of the system.*

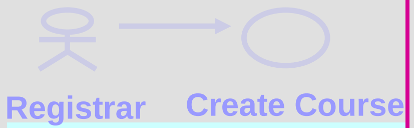
*Then, how about the **inside** view of the system?*

- Interaction diagrams describe how use cases are ***realize***d in terms of interacting objects.

- Two types of interaction diagrams
  - Sequence diagrams
  - Collaboration *(Communication)* diagrams

Diagrams in UML - Sequence Diagram

- A sequence diagram displays object interactions arranged in a time sequence



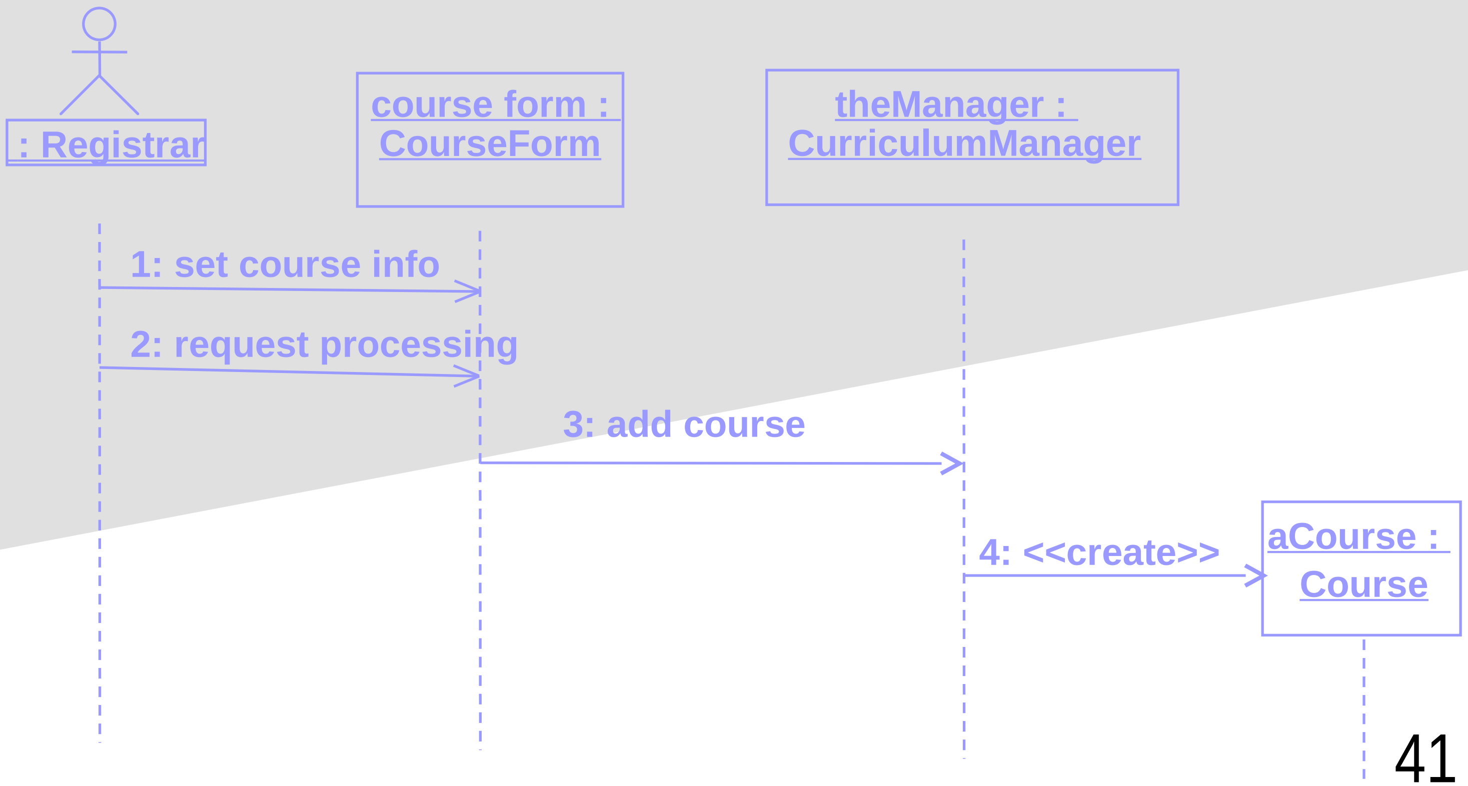
This use case begins after the Registrar logs onto the Registration System with a valid password.

The registrar fills in the course form with the appropriate semester and course related info.

The Registrar requests the system to process the course form.

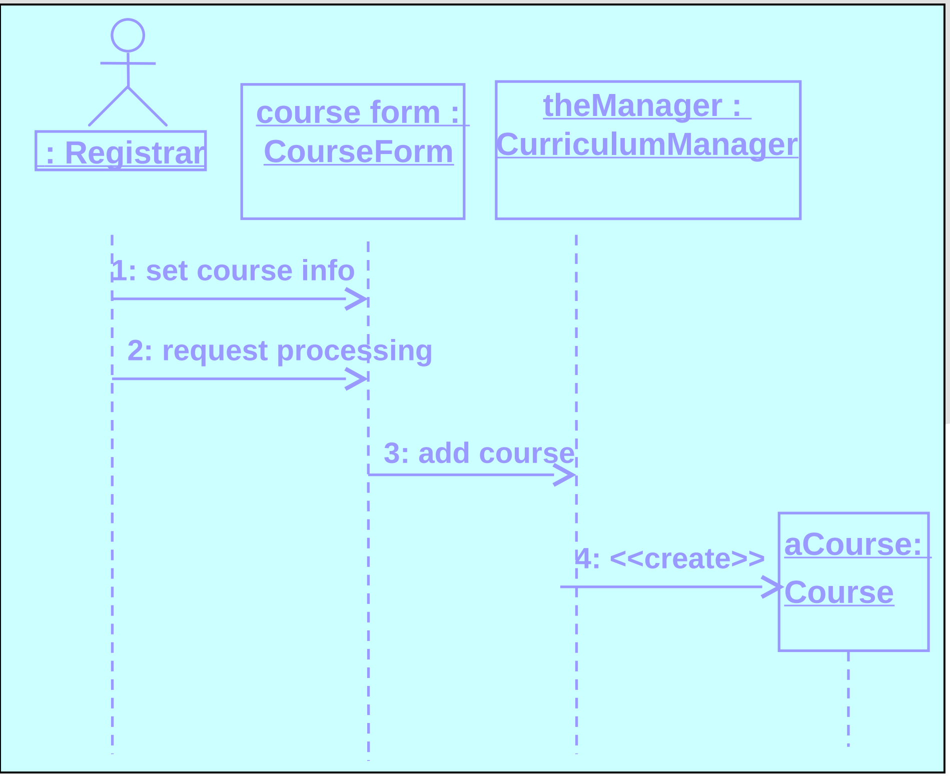
The system creates a new course, and this use case ends

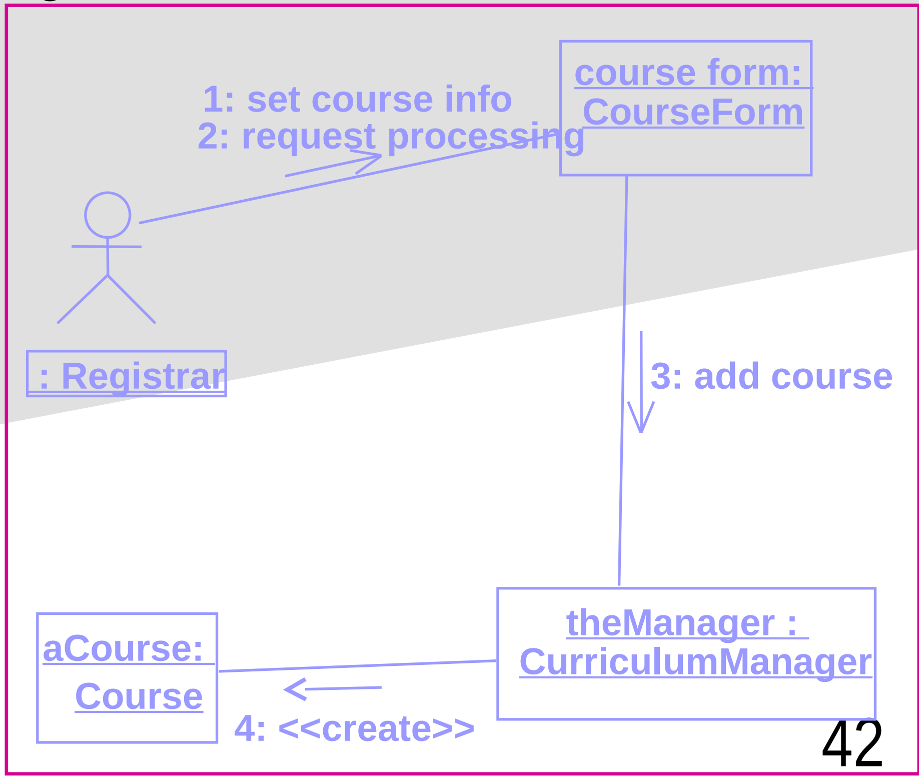
Traceability!



Diagrams in UML – Collaboration (Communication)

- Displays object interactions organized around objects and their direct links to one another.
- Emphasizes the structural organization of objects that send and receive messages.

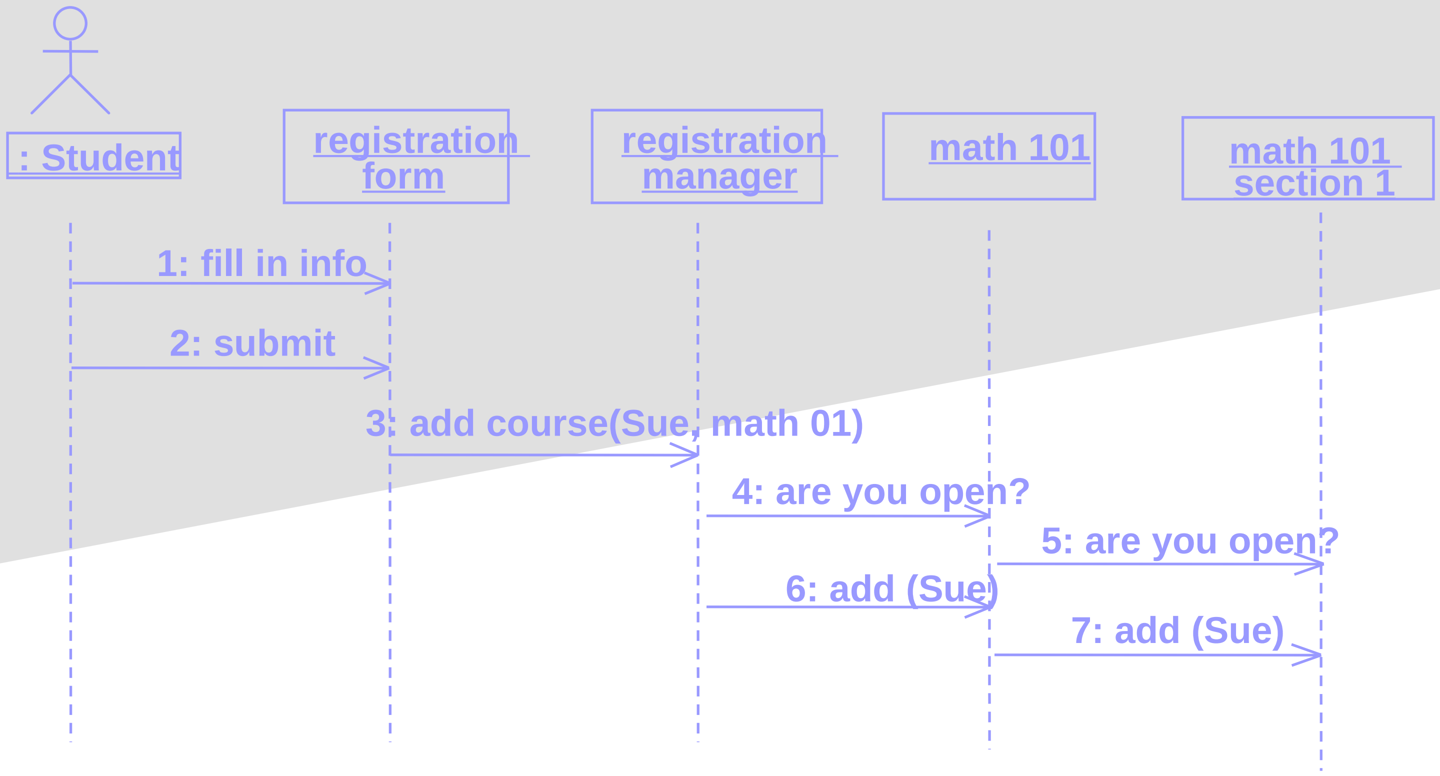




Traceability!

Diagrams in UML – Collaboration (Communication)

- What would be the corresponding collaboration diagram?

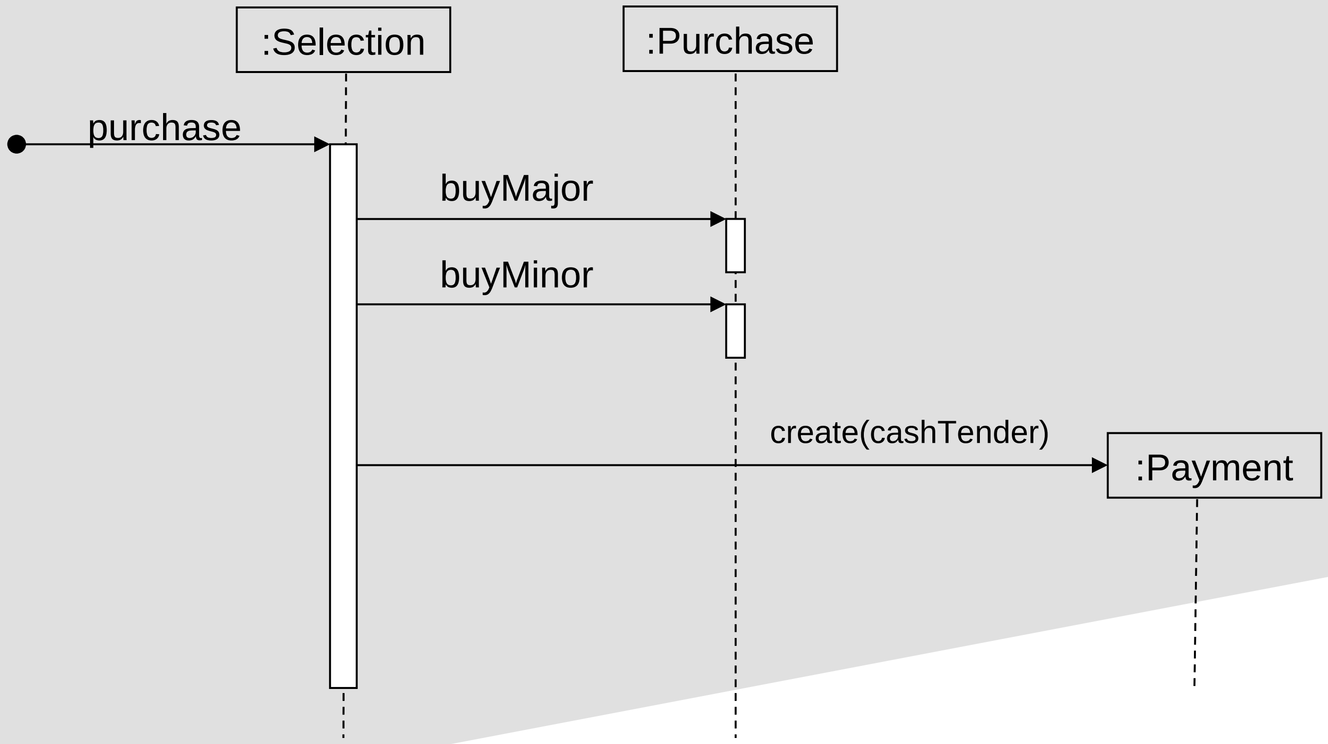


Which use case could this be for?

How about <----------

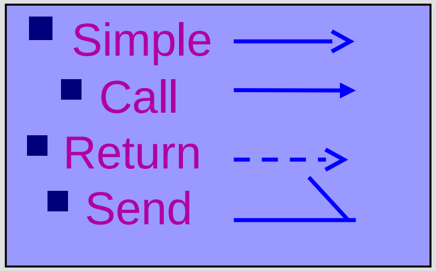
Skip the following for now: In M3.2

Sequence Diagrams & Some Programming



```
public Class Selection
                { private Purchase myPurchase = new Purchase();
                  private Payment myPayment;
                   public void purchase()
                                  { myPurchase.buyMajor();
                                    myPurchase.buyMinor():
                                    myPayment = new Payment( cashTender );
                                    //. .
                                  }
                  // . .
                }
```

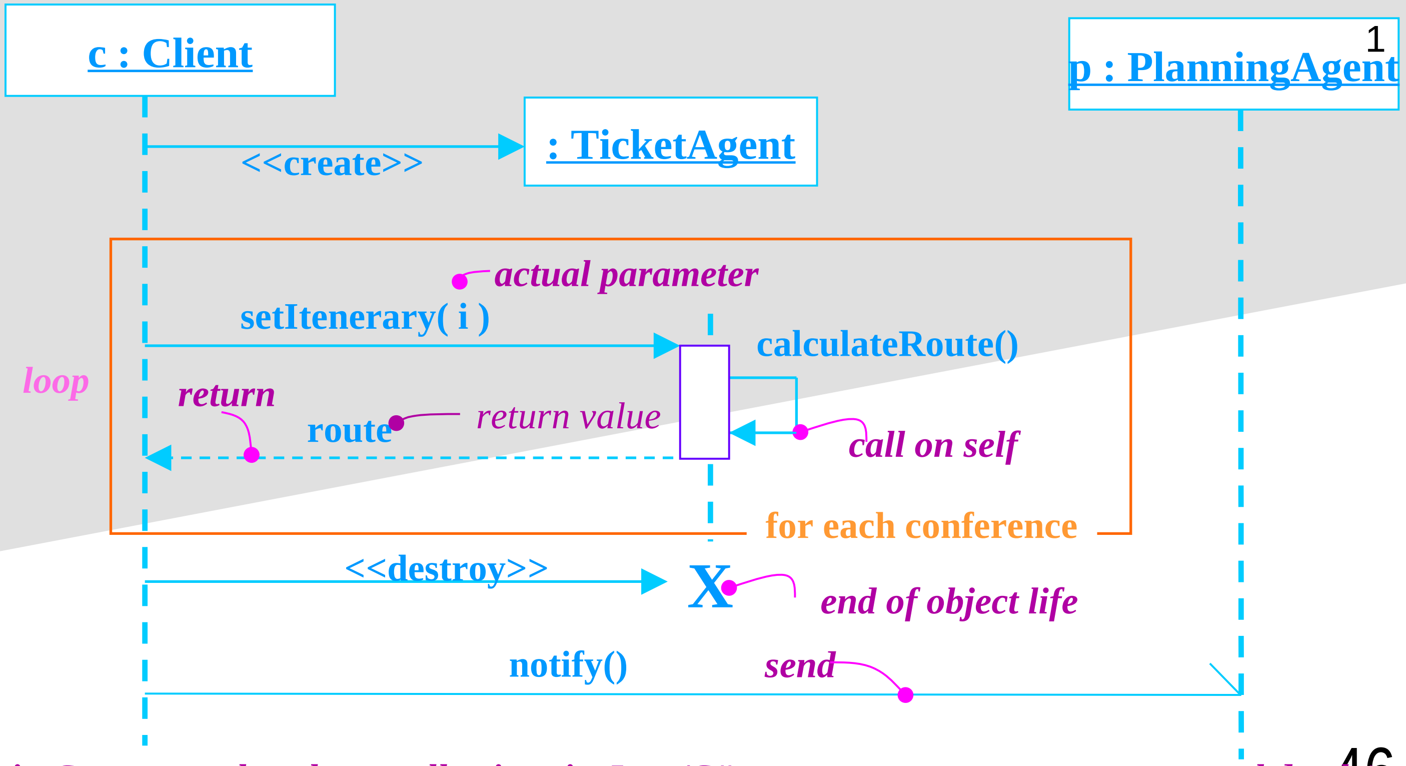
Interactions - Modeling Actions



asynchronous in 2.0 (stick arrowhead) – no return value expected at end of callee activation

activation of caller may end before callee's

half arrow in 1.x

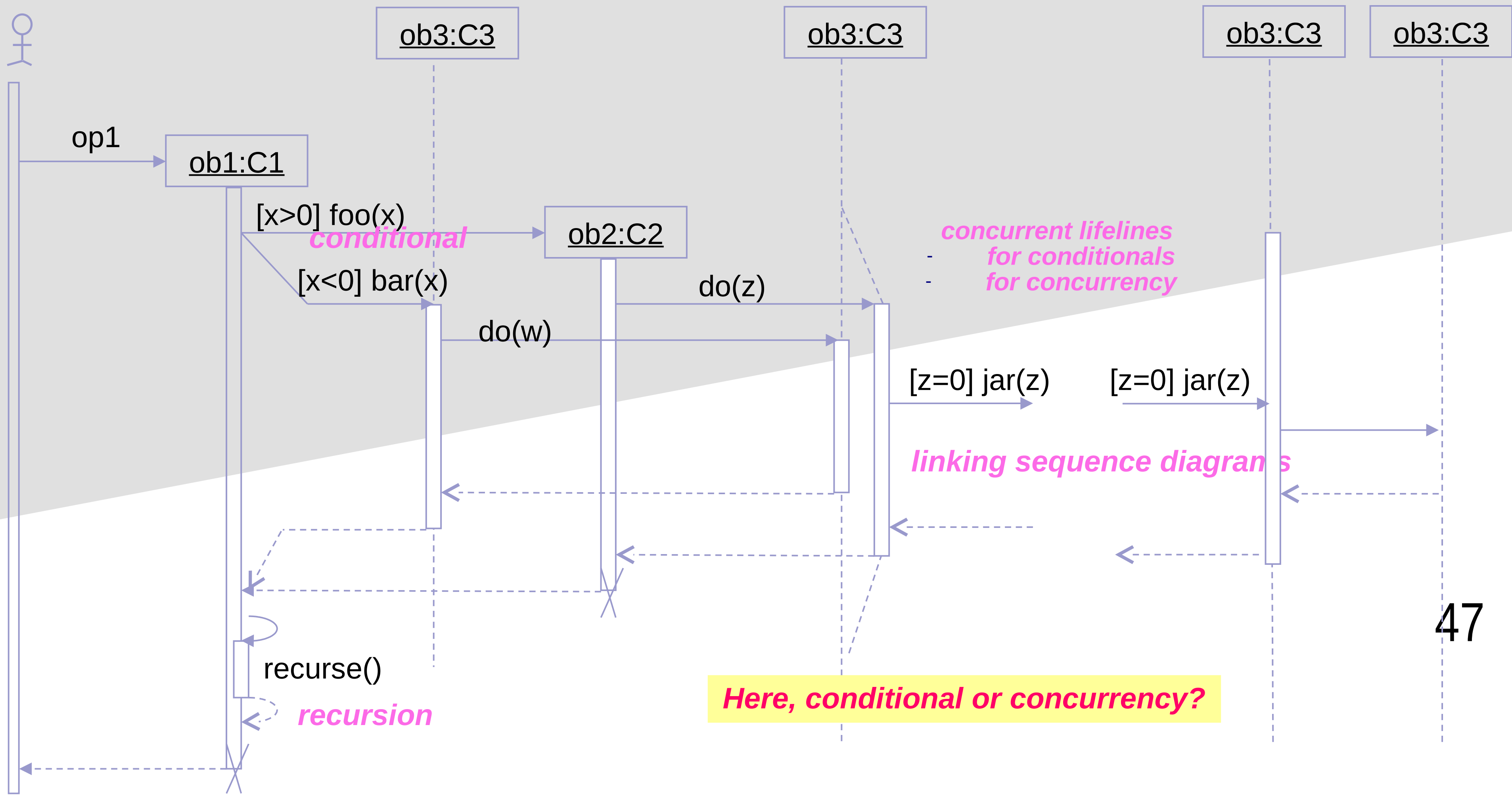


destroy: e.g., in C++ manual garbage collection; in Java/C#, unnecessary

natural death/ self destruction

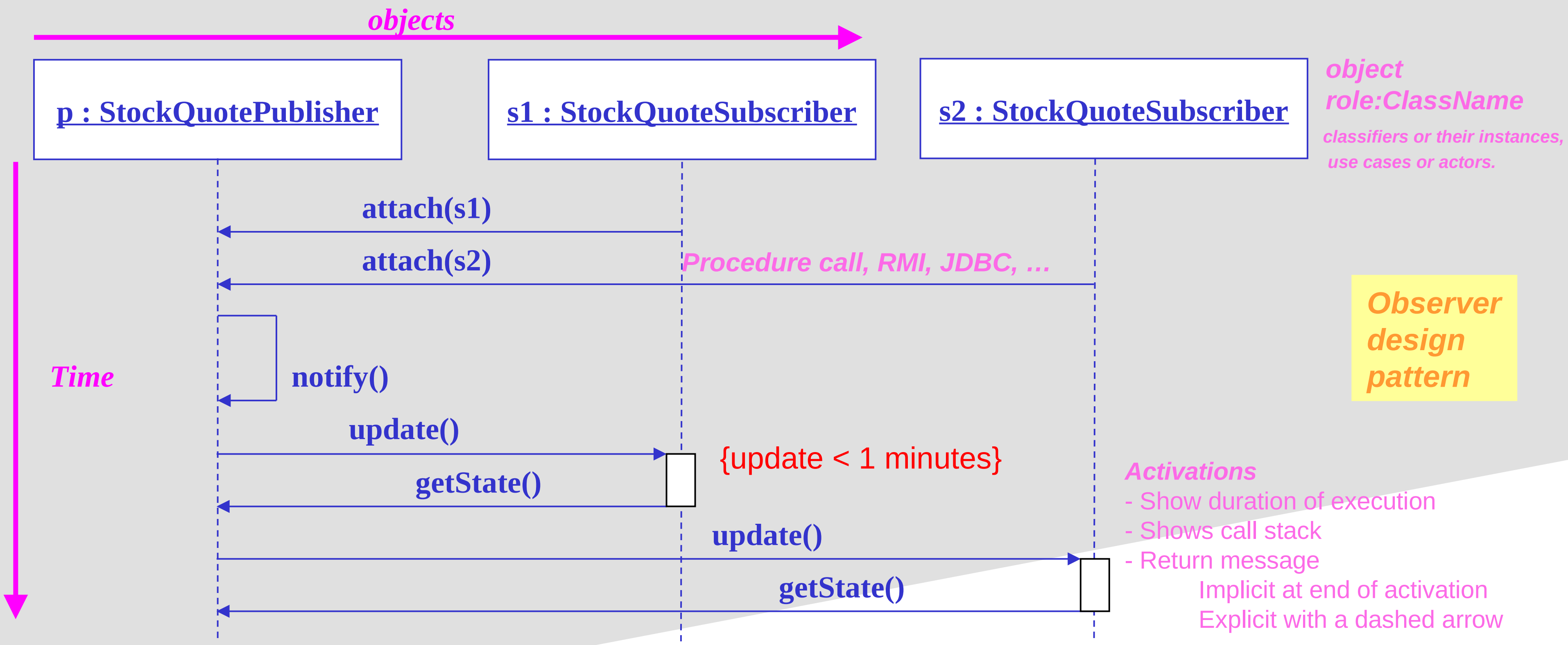
Sequence Diagrams – Generic vs. Instance

- 2 forms of sd:
  - **Instance** sd: describes a specific scenario in detail; no conditions, branches or loops.
  - **Generic** sd: a use case description with alternative courses.



Here, conditional or concurrency?

Interaction Diagram: sequence vs communication

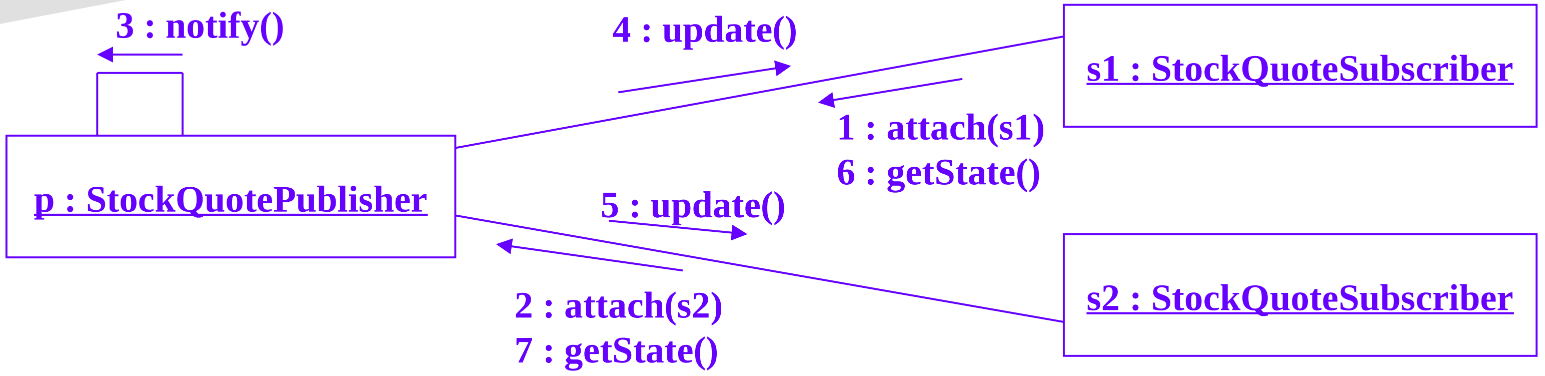


object role:ClassName

classifiers or their instances, use cases or actors.

Observer design pattern

Activations
- Show duration of execution
- Shows call stack
- Return message
  Implicit at end of activation
  Explicit with a dashed arrow

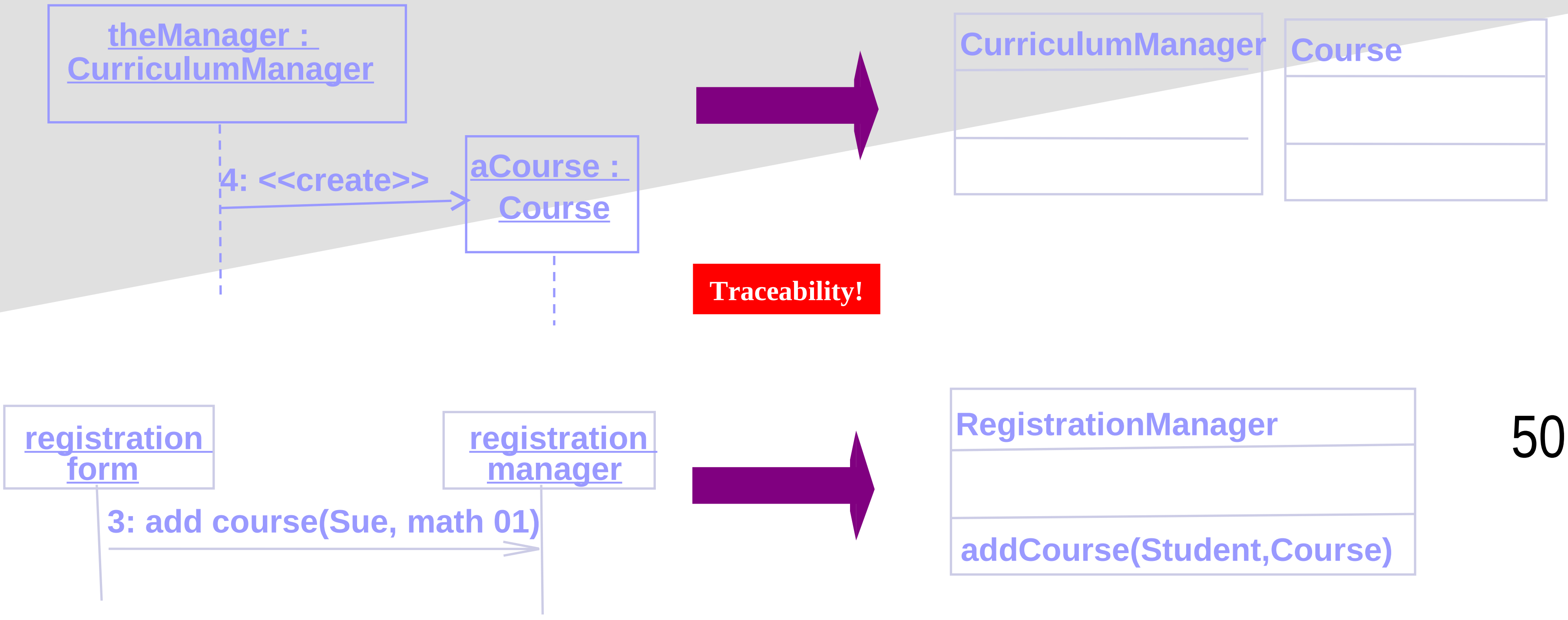


Skip end: resume

Diagrams in UML - Class Diagrams

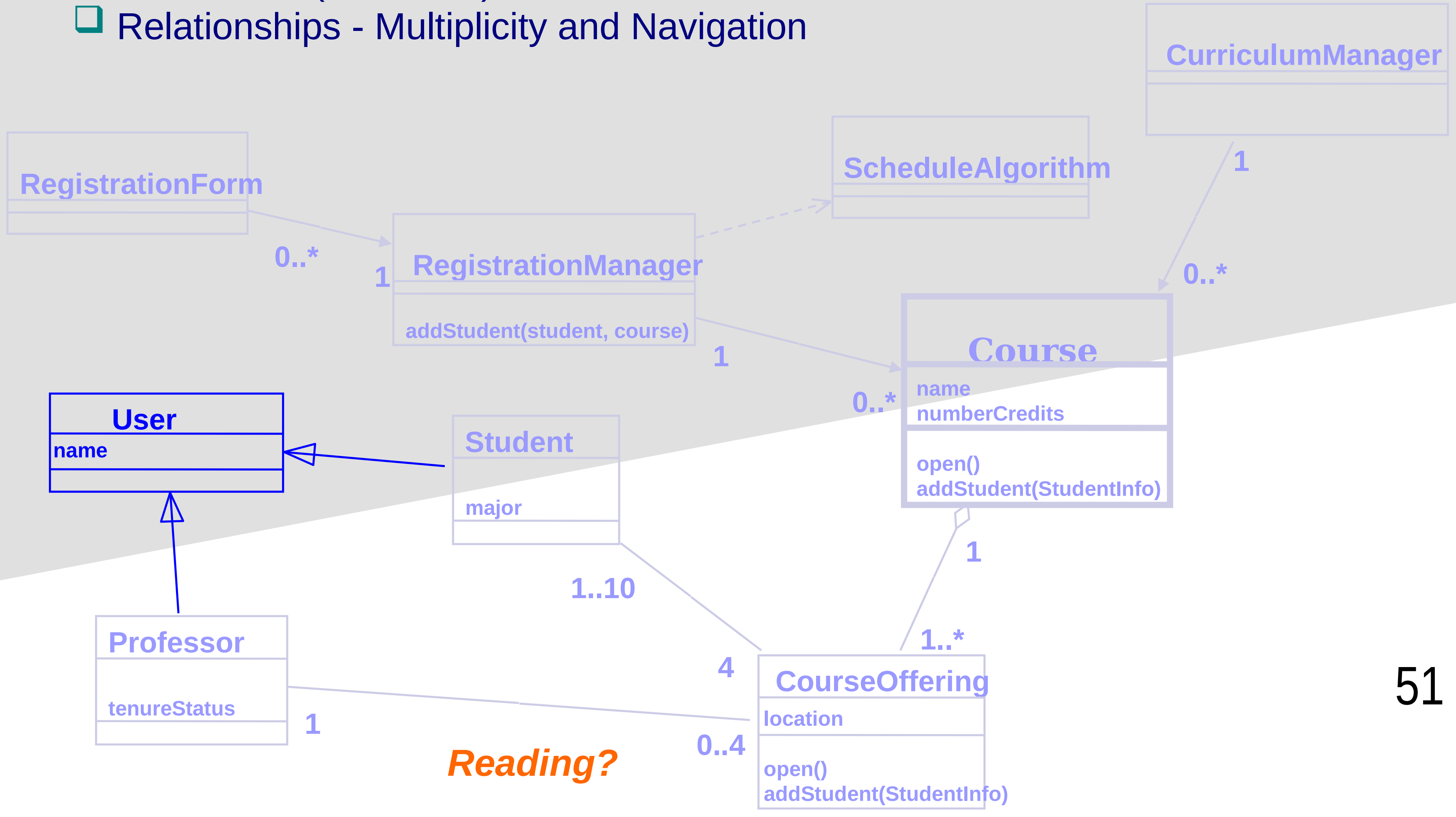
- A class diagram shows the existence of classes and their relationships

- Recall: A class is a collection of objects with common structure, common behavior, common relationships and common semantics
- Some classes are shown through the objects in sequence/collaboration diagram



Diagrams in UML - Class Diagrams: static structure in the system

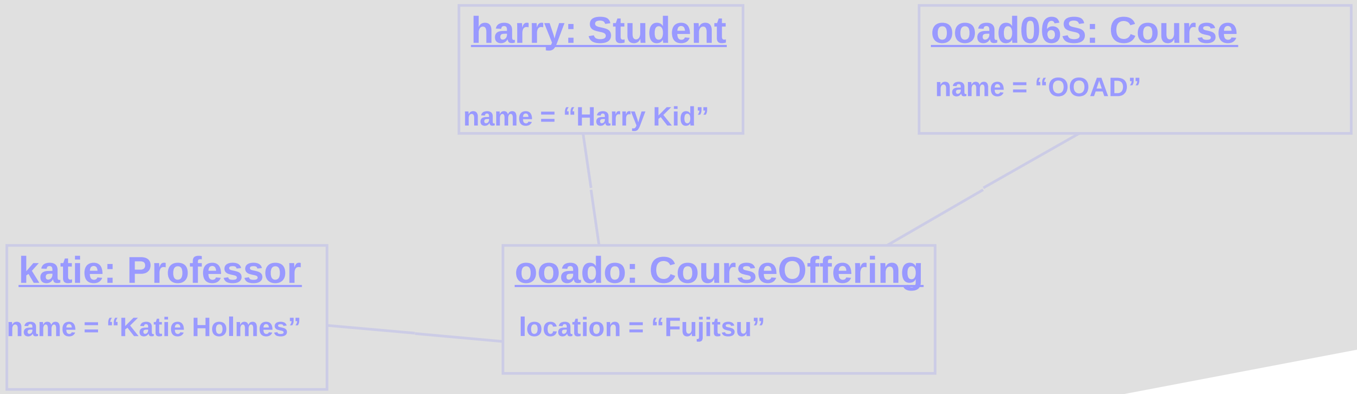
- Naming & (often) 3 Sections;
- Inheritance (as before);
- Relationships - Multiplicity and Navigation



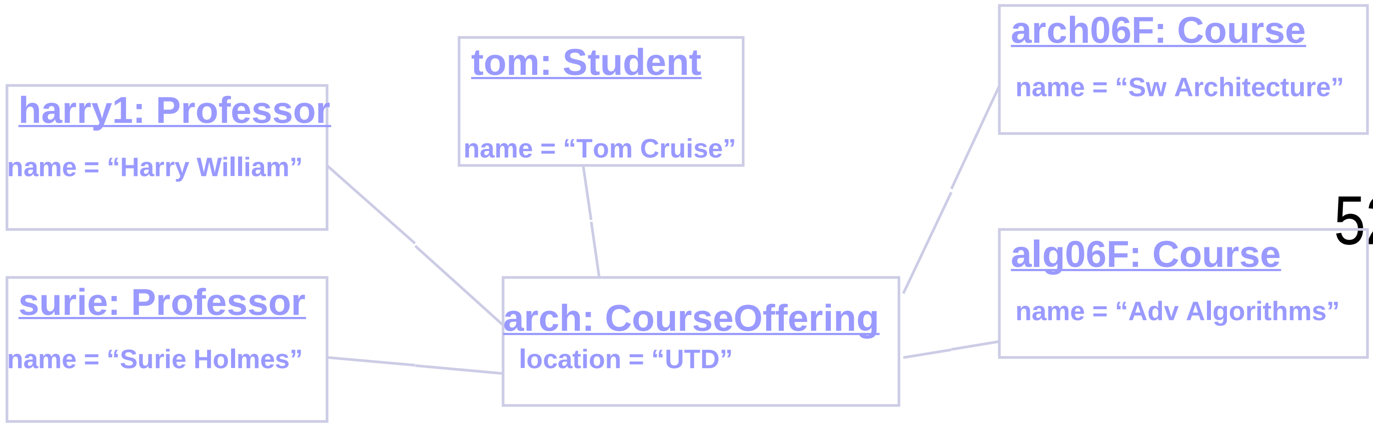
Reading?

Diagrams in UML – Object Diagrams

- Shows a set of objects and their relationships.
- As a static snapshot.

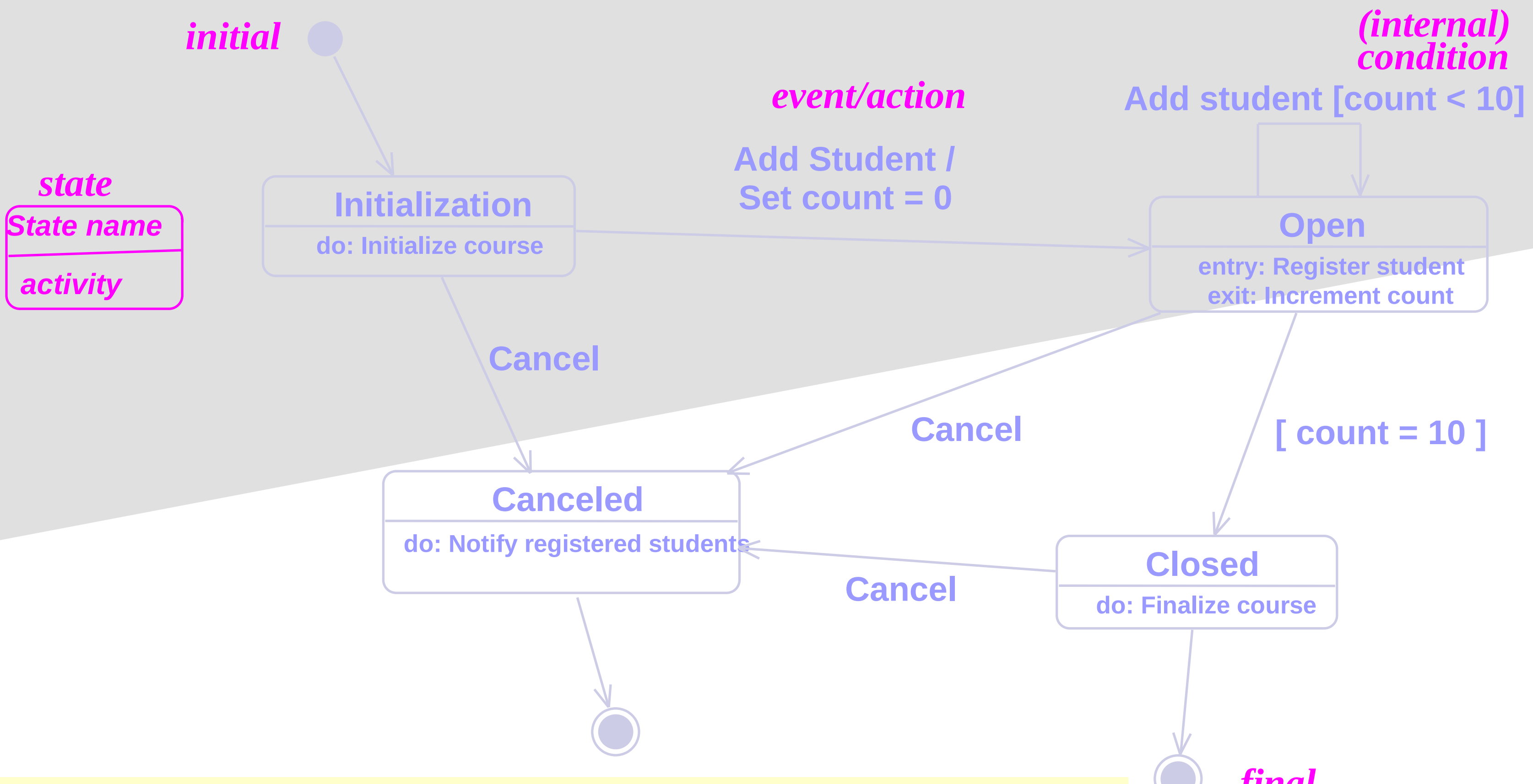


Anything wrong?



Diagrams in UML – State Transition Diagram (Statechart Diagram)

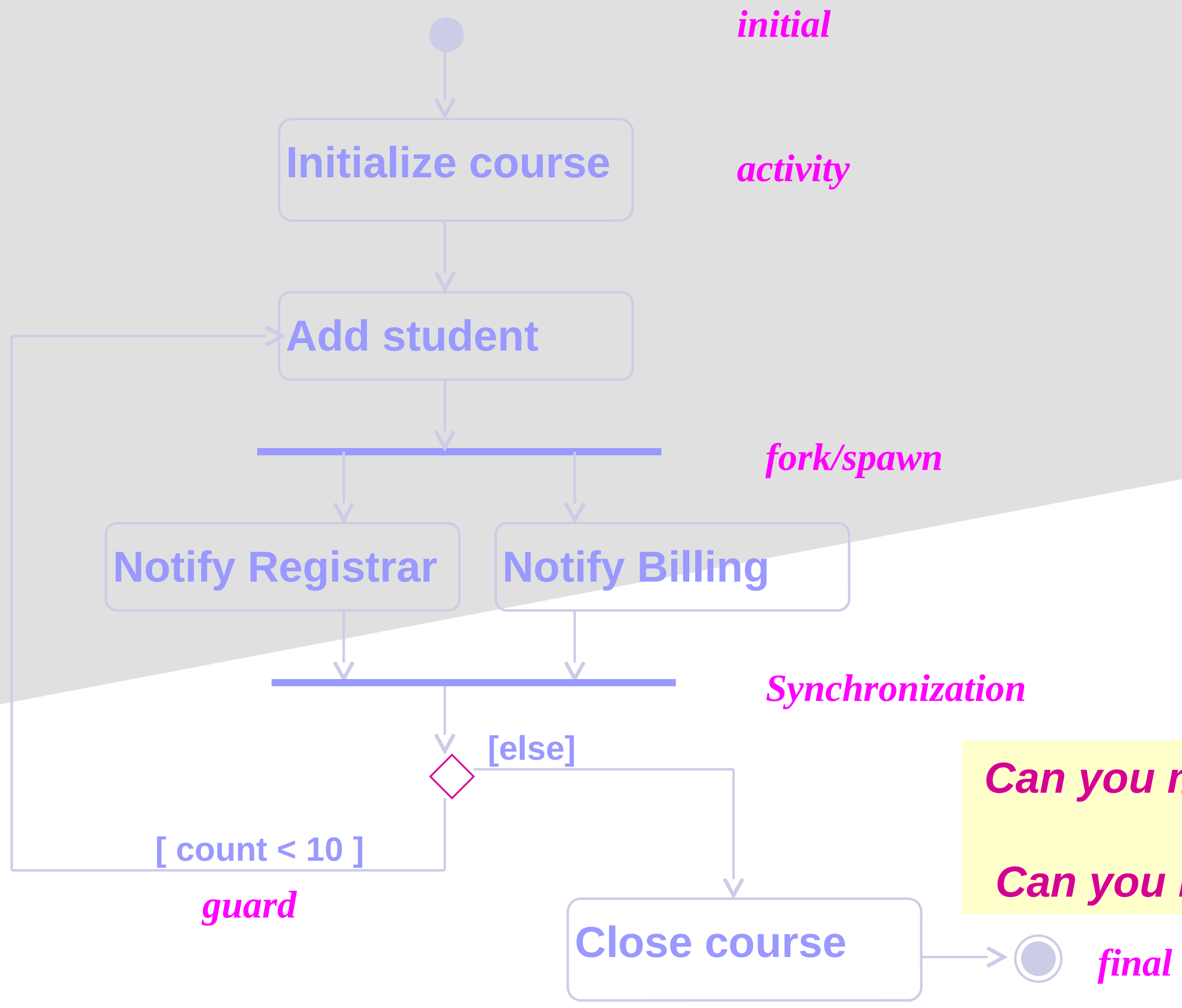
- The life history (often of a given class: from class to object behavior)
- States, transitions, events that cause a transition from one state to another
- Actions that result from a state change



What life history/class is this for? Anything wrong?
…until the drop date?

Diagrams in UML – Activity Diagrams

- A special kind of statechart diagram that shows the flow ***from activity to activity***.



What is this for? Traceability???

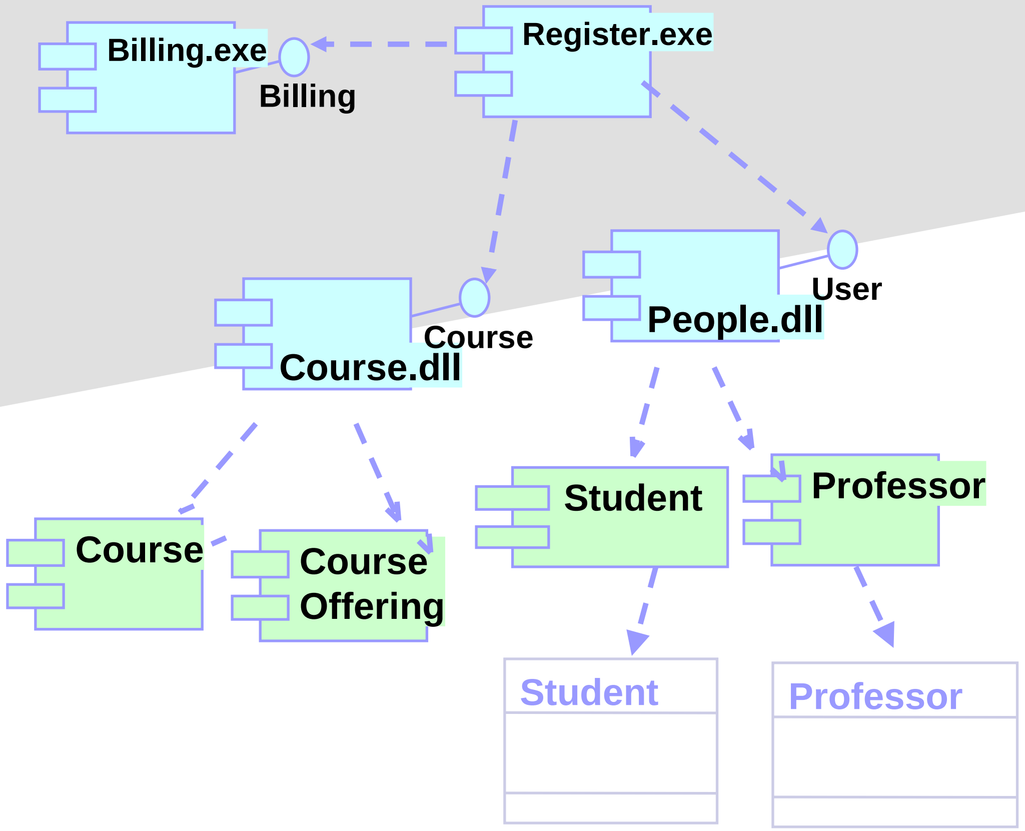
Can you model this using SD?

Can you model this using CD?

Diagrams in UML – Component Diagram

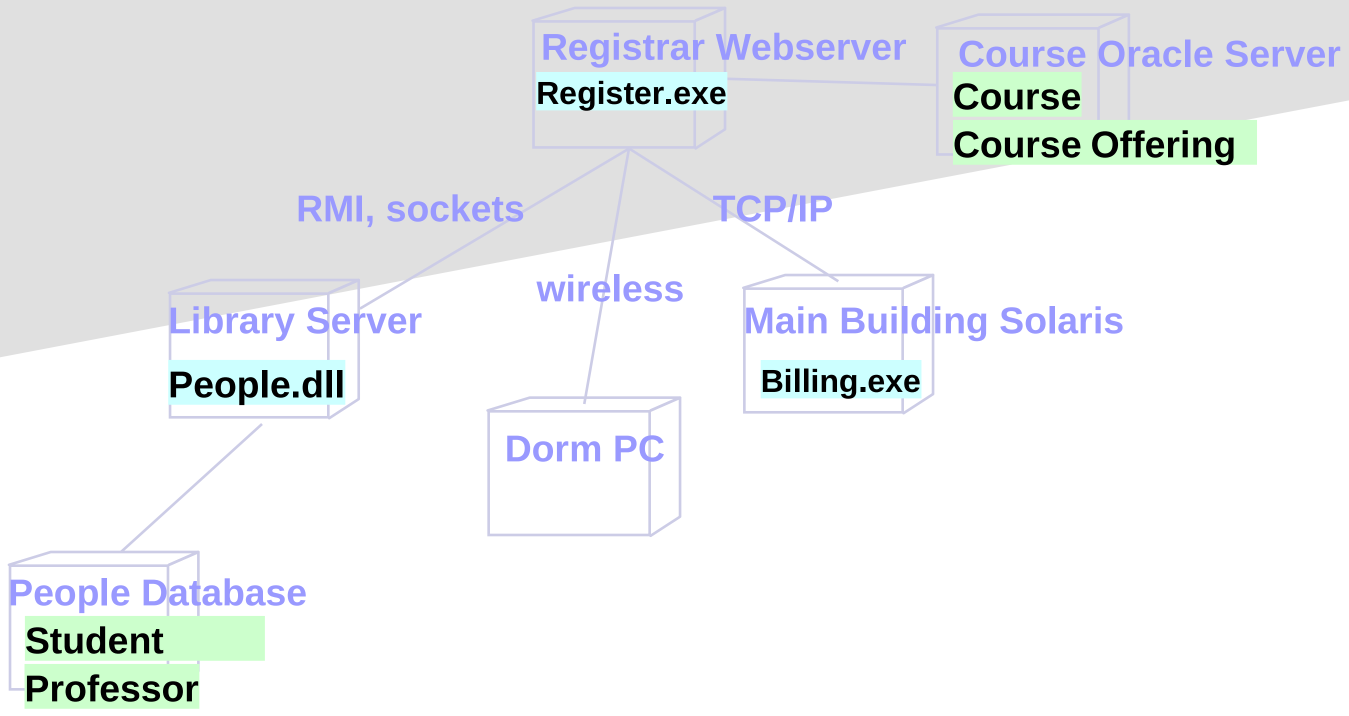
shows the organizations and dependencies among a set of components *(mostly <<uses>>).*

In UML 1.1, a component represented implementation items, such as files and executables;
…
In ***UML 2.0***, a component is a replaceable/reusable, ***architecture***/design-time construct w. interfaces



Diagrams in UML – Deployment Diagram

- shows the configuration of run-time processing elements and the software processes living on them.
- visualizes the distribution of components across the enterprise.



3 basic building blocks of UML - Diagrams

Here, UML 1.x first (UML 2.0 later)

Use case

Sequence; Collaboration *(Communication)*

Class; Object

Statechart Activity

Component Deployment

Using UML Concepts in a Nutshell

- Display the boundary of a system & its major functions using use cases and actors
- Illustrate use case realizations with interaction diagrams
- Represent a static structure of a system using class diagrams
- Model the behavior of objects with state transition diagrams
- Reveal the physical implementation architecture with component & deployment diagrams
- Extend your functionality with stereotypes

Summary

- Background

- What is UML for (both 1.x and 2.0)?

  for visualizing, specifying, constructing, and documenting models

- Building blocks of UML

  Things, Relationships (4 kinds) and Diagrams (9 different kinds)

Module 2:
Introduction to UML - Appendix

Extensibility of UML

- Stereotypes (<< >>) can be used to extend the UML notational elements
- Stereotypes may be used to classify and extend associations, inheritance relationships, classes, and components
- Examples:
  - Class stereotypes: boundary, control, entity, utility, exception
  - Inheritance stereotypes: uses and extends
  - Component stereotypes: subsystem

Stereotypes *— extends vocabulary (metaclass in UML metamodel)*
Tagged values *— extends properties of UML building blocks (i.e., metamodel)*
Constraints *— extend the semantics of UML building blocks.*

More on this later

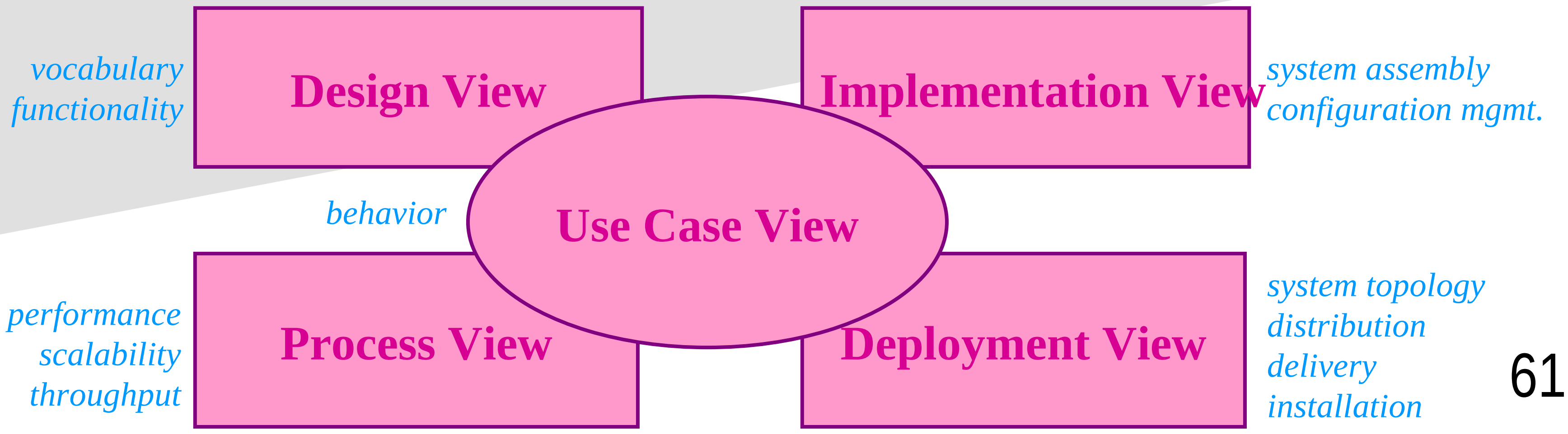
Architecture & Views

(You can skip this part on the first reading)

UML is for visualizing, specifying, constructing, and documenting with emphasis on system architectures (things in the system and relationships among the things) from five different views

Architecture - set of significant decisions regarding:

- Organization of a software system.
- Selection of structural elements & interfaces from which a system is composed.
- Behavior or collaboration of elements.
- Composition of structural and behavioral elements.
- Architectural style guiding the system.



Views

Use Case View

- Use Case Analysis is a technique to capture business process from user’s perspective.
- Encompasses the behavior as seen by users, analysts and testers.
- Specifies forces that shape the architecture.
- Static aspects in use case diagrams; Dynamic aspects in interaction (statechart and activity) diagrams.

Design View

- Encompasses classes, interfaces, and collaborations that define the vocabulary of a system.
- Supports functional requirements of the system.
- Static aspects in class and object diagrams; Dynamic aspects in interaction diagrams.

Process View

- Encompasses the threads and processes defining concurrency and synchronization.
- Addresses performance, scalability, and throughput.
- Static and dynamic aspects captured as in design view; emphasis on active classes.

Implementation View

- Encompasses components and files used to assemble and release a physical system.
- Addresses configuration management.
- Static aspects in component diagrams; Dynamic aspects in interaction diagrams.

Deployment View

- Encompasses the nodes that form the system hardware topology.
- Addresses distribution, delivery, and installation.
- Static aspects in deployment diagrams; Dynamic aspects in interaction diagrams.

Rules of UML

- Well formed models — *semantically self-consistent and in harmony with all its related models*.
  - Semantic rules for:
    - Names — what you can call things.
    - Scope — context that gives meaning to a name.
    - Visibility — how names can be seen and used.
    - Integrity — how things properly and consistently relate to one another.
    - Execution — what it means to run or simulate a dynamic model.

- Avoid models that are

  Elided — certain elements are hidden for simplicity.

  Incomplete — certain elements may be missing.

  Inconsistent — no guarantee of integrity.

Process for Using UML

How do we use UML as a notation to construct a good model?

- Use case driven — use cases are primary artifact for defining behavior of the system.
- Architecture-centric — the system's architecture is primary artifact for conceptualizing, constructing, managing, and evolving the system.
- Iterative and incremental — managing streams of executable releases with increasing parts of the architecture included.

The Rational Unified Process (RUP)

Process for Using UML - Iterative Life Cycle

- It is planned, managed and predictable …almost
- It accommodates changes to requirements with less disruption
- It is based on evolving executable prototypes, not documentation
- It involves the user/customer throughout the process
- It is risk driven

Primary phases

- Inception — seed idea is brought up to point of being a viable project.
- Elaboration — product vision and architecture are defined.
  (http://www.utdallas.edu/~chung/OOAD_SUMMER04/HACS_vision_12.doc)
- Construction — brought from architectural baseline to point of deployment into user community.
- Transition — turned over to the user community.

Process for Using UML - Iterative Approach

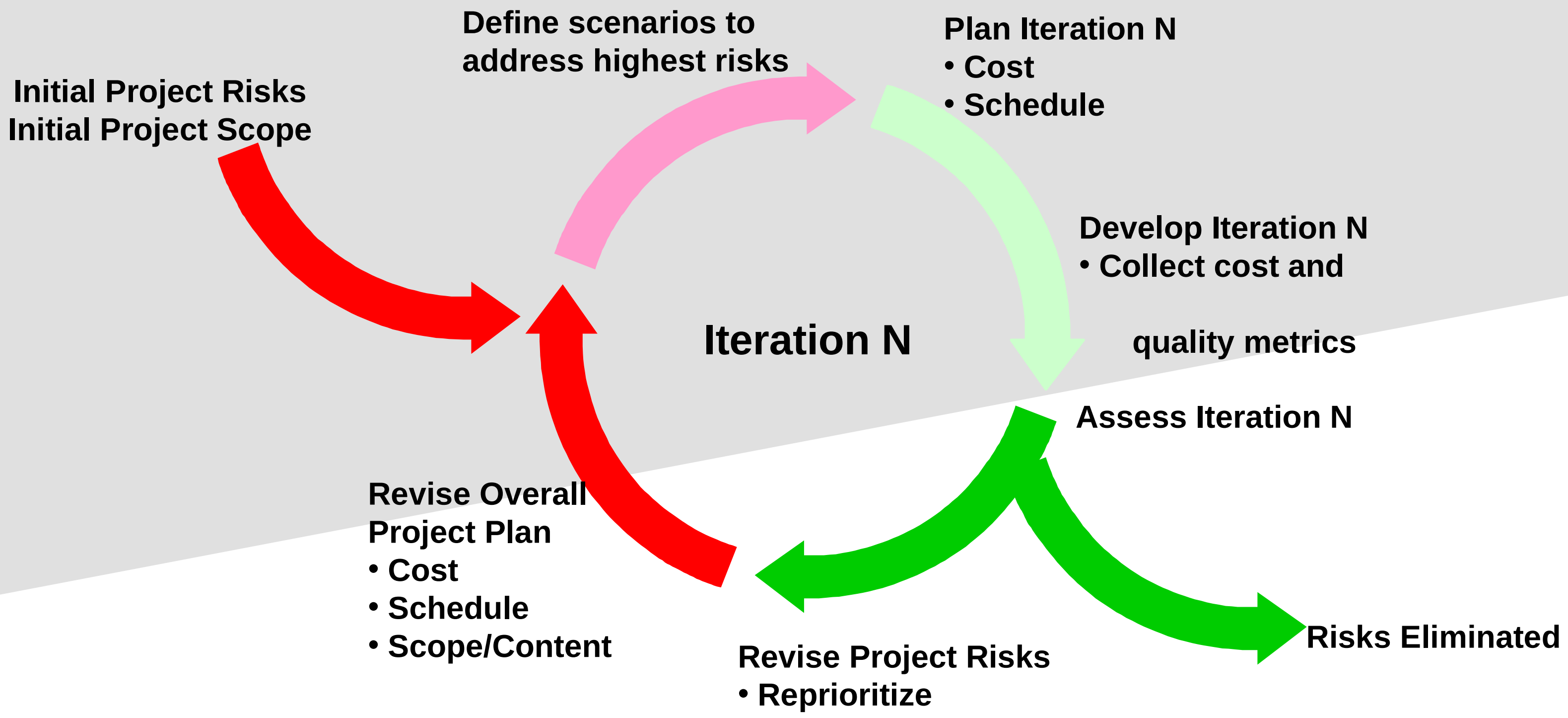
Three Important Features

- Continuous integration - Not done in one lump near the delivery date
- Frequent, executable releases - Some internal; some delivered
- Attack risks through demonstrable progress - Progress measured in products, not documentation or engineering estimates

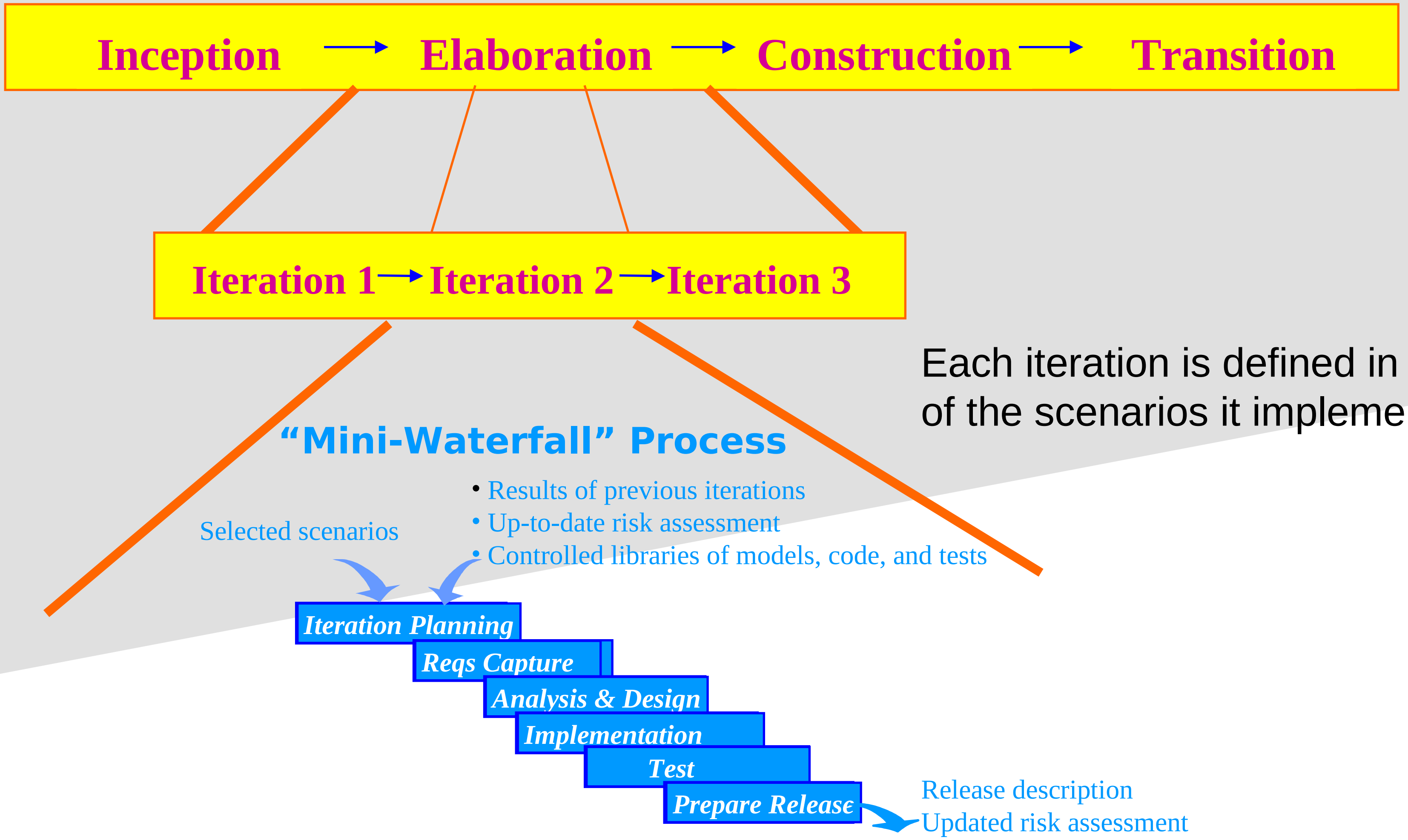
Resulting Benefits

- Releases are a forcing function that drives the development team to closure at regular intervals - Cannot have the “90% done with 90% remaining” phenomenon
- Can incorporate problems/issues/changes into future iterations rather than disrupting ongoing production
- The project’s supporting elements (testers, writers, toolsmiths, QA, etc.) can better schedule their work

Process for Using UML - Risk Reduction Drives Iterations



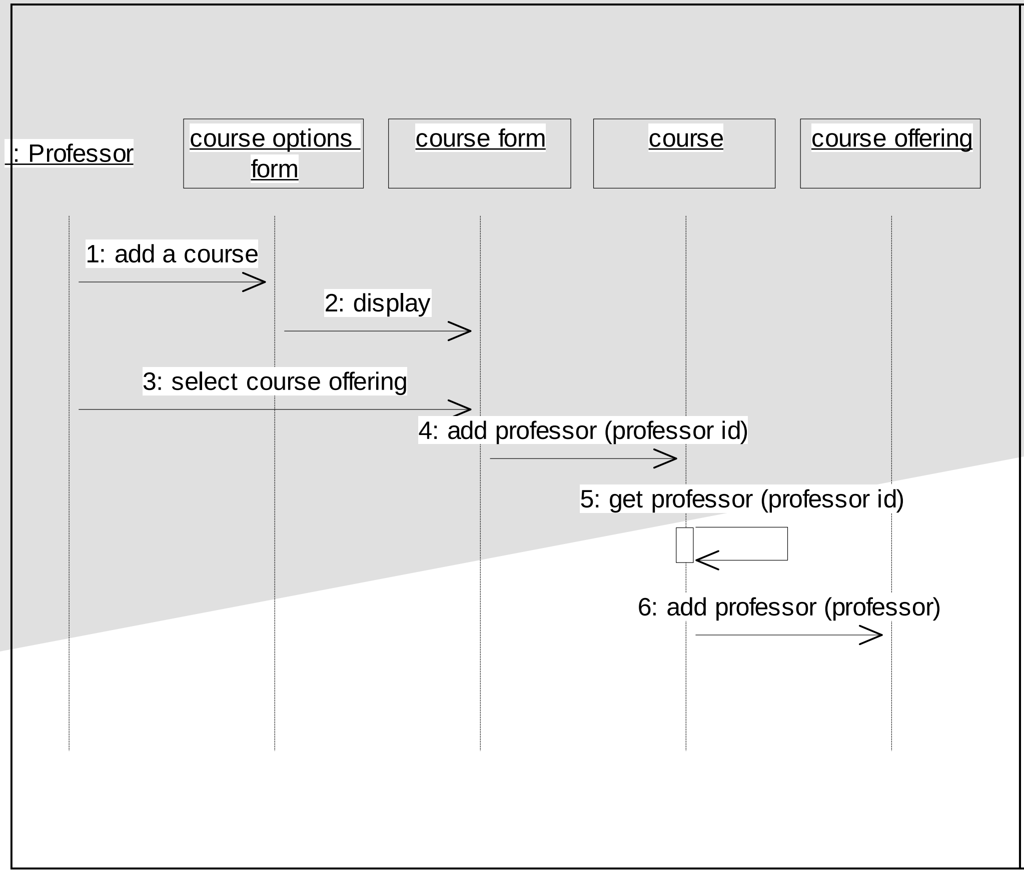
Process for Using UML - Use Cases Drive the Iteration Process



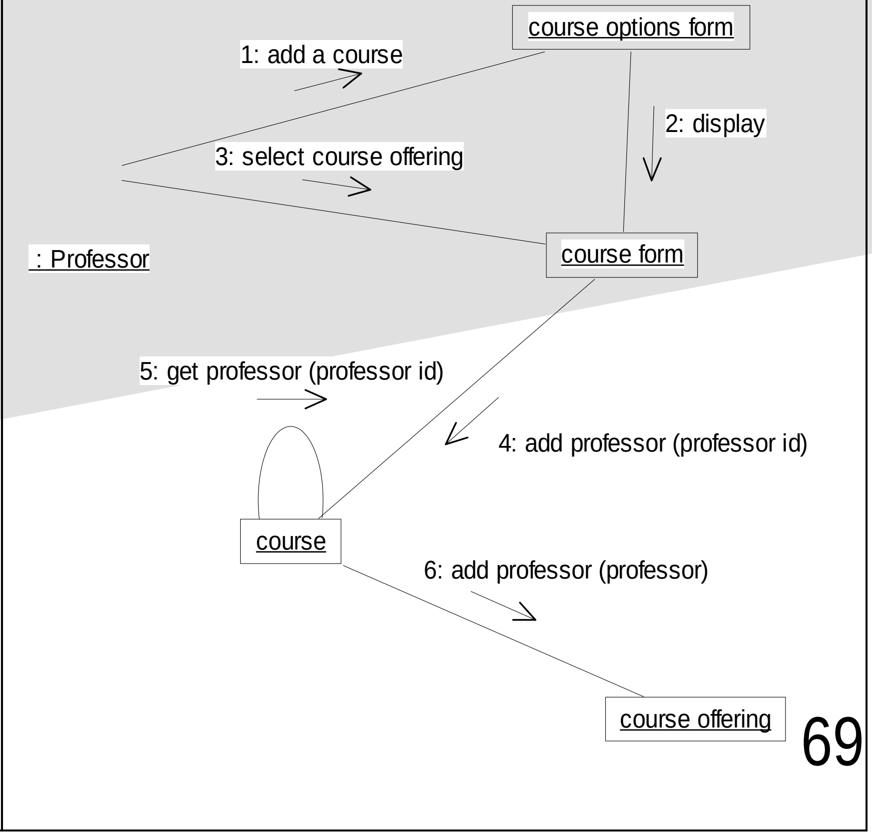
Each iteration is defined in terms of the scenarios it implements

Points to Ponder

Are Sequence and Collaboration Diagrams Isomorphic?



: Professor
course options form
course form
course
course offering
1: add a course
2: display
3: select course offering
4: add professor (professor id)
5: get professor (professor id)
6: add professor (professor)



course options form
1: add a course
2: display
3: select course offering
: Professor
course form
5: get professor (professor id)
4: add professor (professor id)
course
6: add professor (professor)
course offering

Points to Ponder

- How much unification does UML do?

Consider the Object Model Notation on the inside cover on the front and back of the textbook "Object Oriented Modeling and Design" by Rumbaugh, et.al.

1. List the OMT items that do not exist in UML
2. List the UML items that do not exist in OMT
3. For those items of OMT for which UML equivalents exist, map the notation to UML.

- Where would you want to use stereotypes?
- Model the “Business Process” on page 6 in UML.
- Map the four (4) phases of the RUP to the traditional software lifecycle.
- If an object refers to a concept, can an object refer to a concept of an concept? Consider some examples.
- What would be the essential differences between a property and an attribute? Consider some examples.
- What is the syntax and semantics of a class diagram?
- In Component-Based Software Engineering (CBSE), components are the units, or building blocks, of a (distributed) software system.

What kind of building blocks of UML can be components for CBSE?

Module 3 – Advanced Features: Part I - Structural Diagrams

3 basic building blocks of UML - Diagrams

Graphical representation of a set of elements.
Represented by a connected graph: Vertices are things; Arcs are relationships/behaviors.
5 most common views built from

UML 1.x: 9 diagram types.

Structural Diagrams

Represent the static aspects of a system.

- Class;
  Object
- Component
- Deployment

Behavioral Diagrams

Represent the dynamic aspects.

- Use case
- Sequence;
  Collaboration
- Statechart
- Activity

UML 2.0: 12 diagram types

Structural Diagrams

- Class;
  Object
- Component
- Deployment
- Composite Structure
- Package

Behavioral Diagrams

- Use case
- Statechart
- Activity

Interaction Diagrams

- Sequence;
  Communication
- Interaction Overview
- Timing

Class Diagrams

Structural Diagrams

- Class;
  Object
- Component
- Deployment
- **Composite Structure**
- Package

Class Diagram

The basis for all object modeling
All things lead to this

- Most common diagram.
- Shows a set of classes, interfaces, and collaborations and their relationships (dependency, generalization, association and realization); notes too.
- Represents the static view of a system (With active classes, static process view)

Three modeling perspectives for Class Diagram

- ***Conceptual***: the diagram reflects the domain
- ***Specification***: focus on interfaces of the software (Java supports interfaces)
- ***Implementation***: class (logical database schema) definition to be implemented in code and database.

Most users of OO methods take an implementation perspective, which is a shame because the other perspectives are often more useful. *-- Martin Fowler*

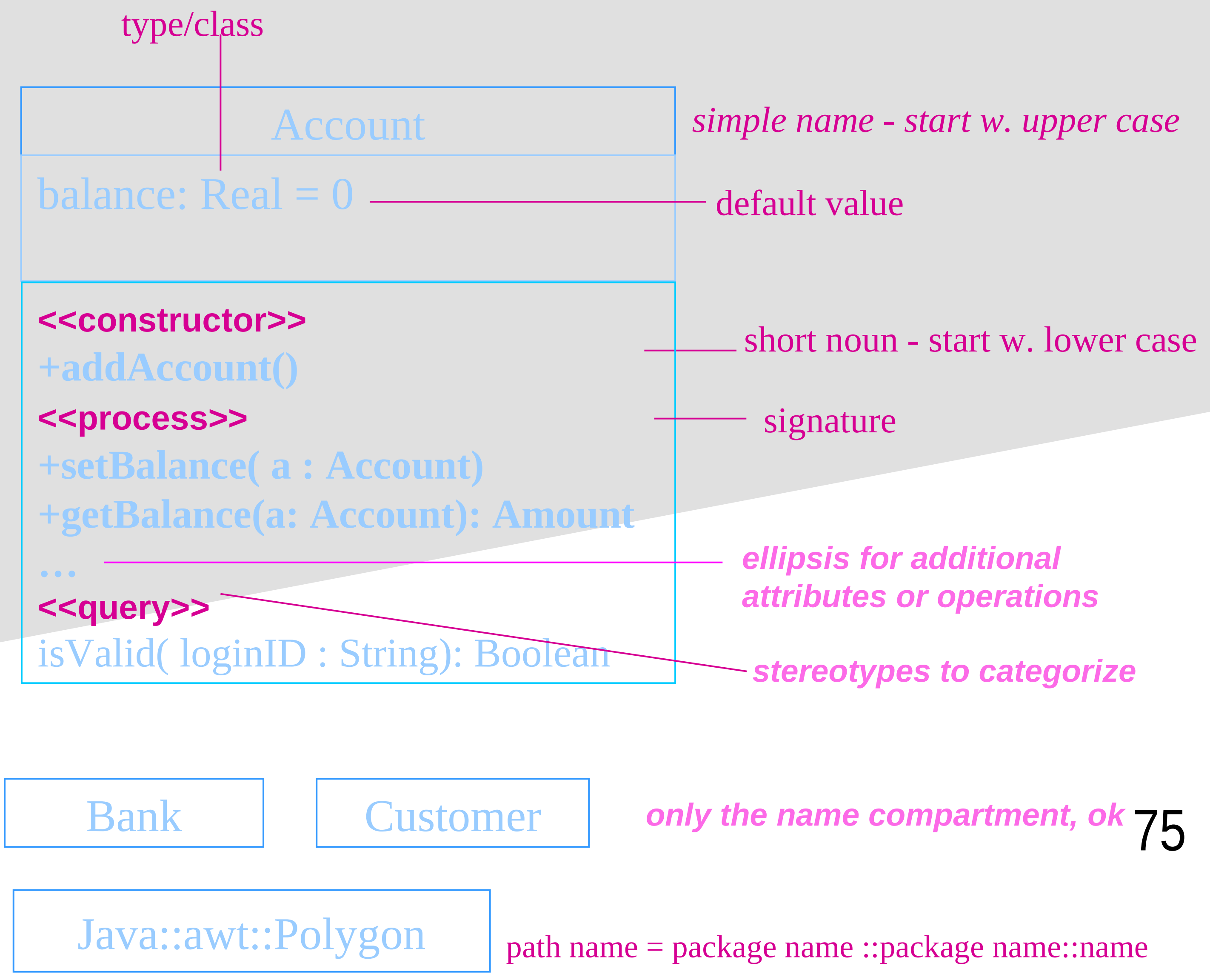
Classes

Names

Attributes

Operations

may cause object to change state


type/class
Account
simple name - start w. upper case
balance: Real = 0
default value
<<constructor>>
+addAccount()
short noun - start w. lower case
<<process>>
signature
+setBalance(a : Account)
+getBalance(a: Account): Amount
…
ellipsis for additional attributes or operations
<<query>>
isValid(loginID : String): Boolean
stereotypes to categorize
Bank
Customer
only the name compartment, ok
Java::awt::Polygon
path name = package name ::package name::name

Responsibilities

Account
Responsibilities -- handles deposits -- reports fraud to managers

- anything that a class knows or does
  (Contract or obligation)

- An optional 4th item carried out by attributes and operations.
- Free-form text; one phrase per responsibility.
- Technique - CRC cards (Class-Responsibility-Collaborator); Kent Beck and Ward Cunningham'89

- A collaborator is also a class which the (current) class interacts with to fulfill a responsibility

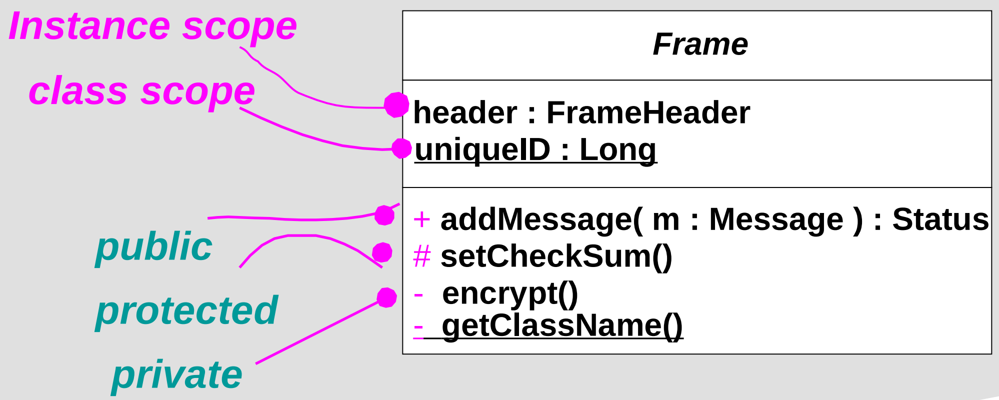
Class Name	
Responsibilities	Collaborators

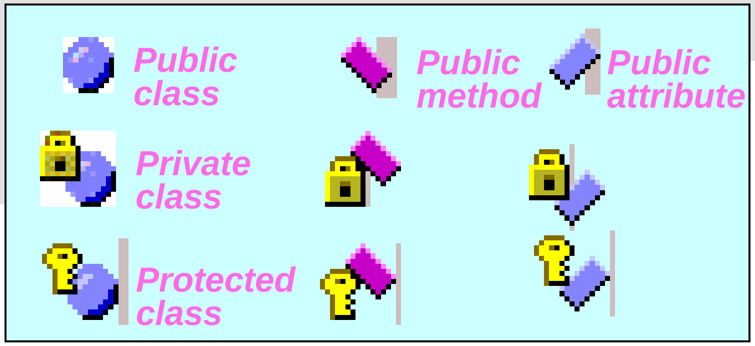
Customer	
Opens account Knows name Knows address	Account

Account	
Knows interest rate Knows balance Handles deposits Reports fraud to manager	Manager

Scope & Visibility

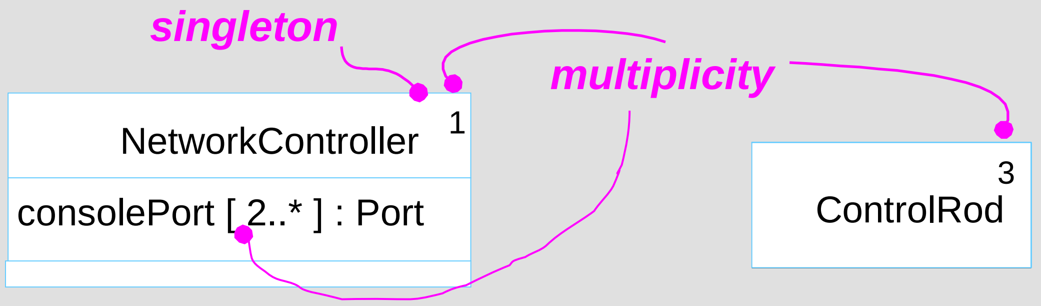
- *Instance* Scope — each instance of the classifier holds its own value.
- *Class* Scope — one value is held for all instances of the classifier (underlined).



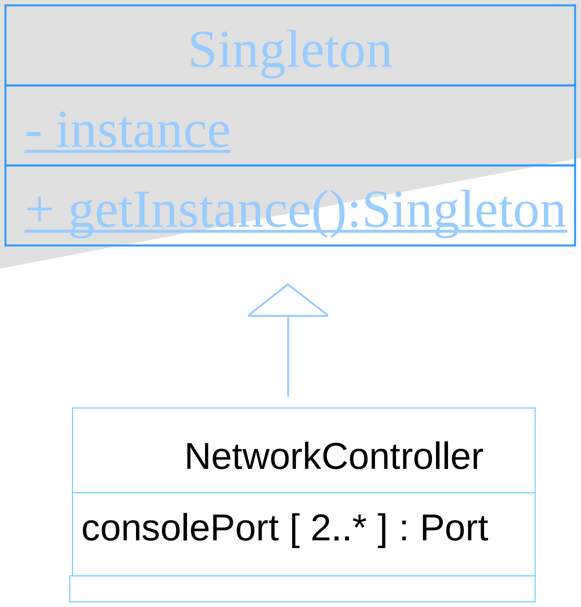


- Public - access allowed for any outside classifier (+).
- Protected - access allowed for any descendant of the classifier (#).
- Private - access restricted to the classifier itself (-).
- (using adornments in JBuilder)

Multiplicity



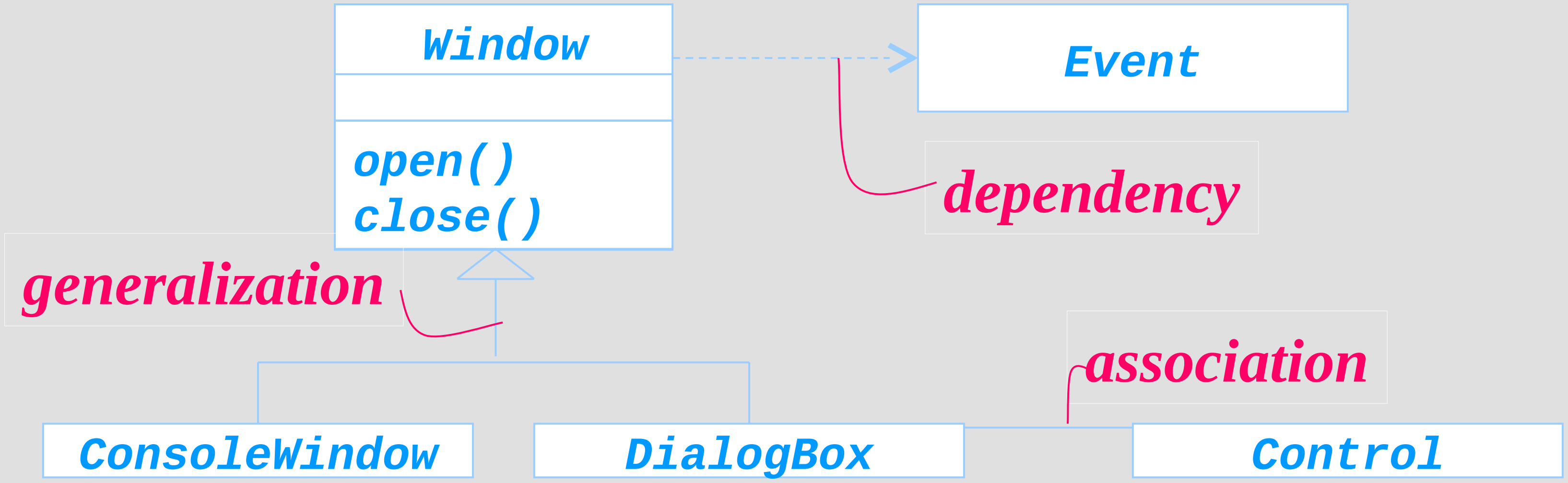
Using Design Pattern

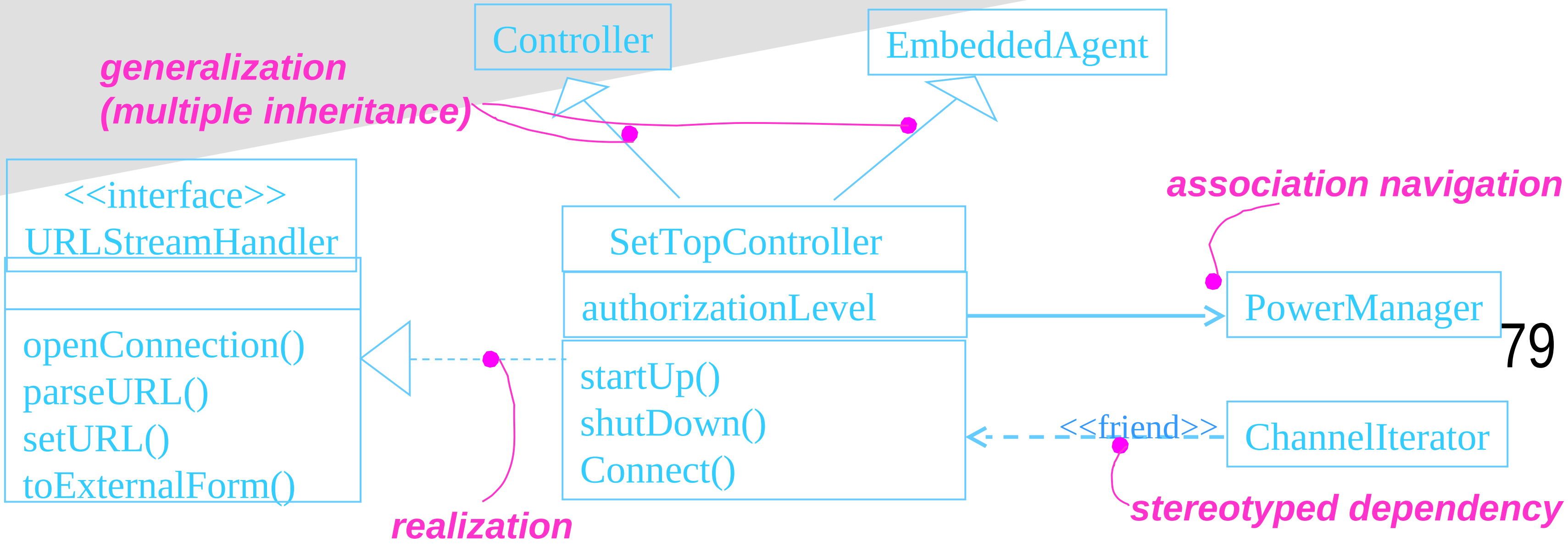


```
public class Singleton {
   private static Singleton instance = null;

   private Singleton() {}
   public static Singleton getInstance() {
      if (instance == null) {
         instance = new Singleton();
      }
      return instance;
   }
}
```

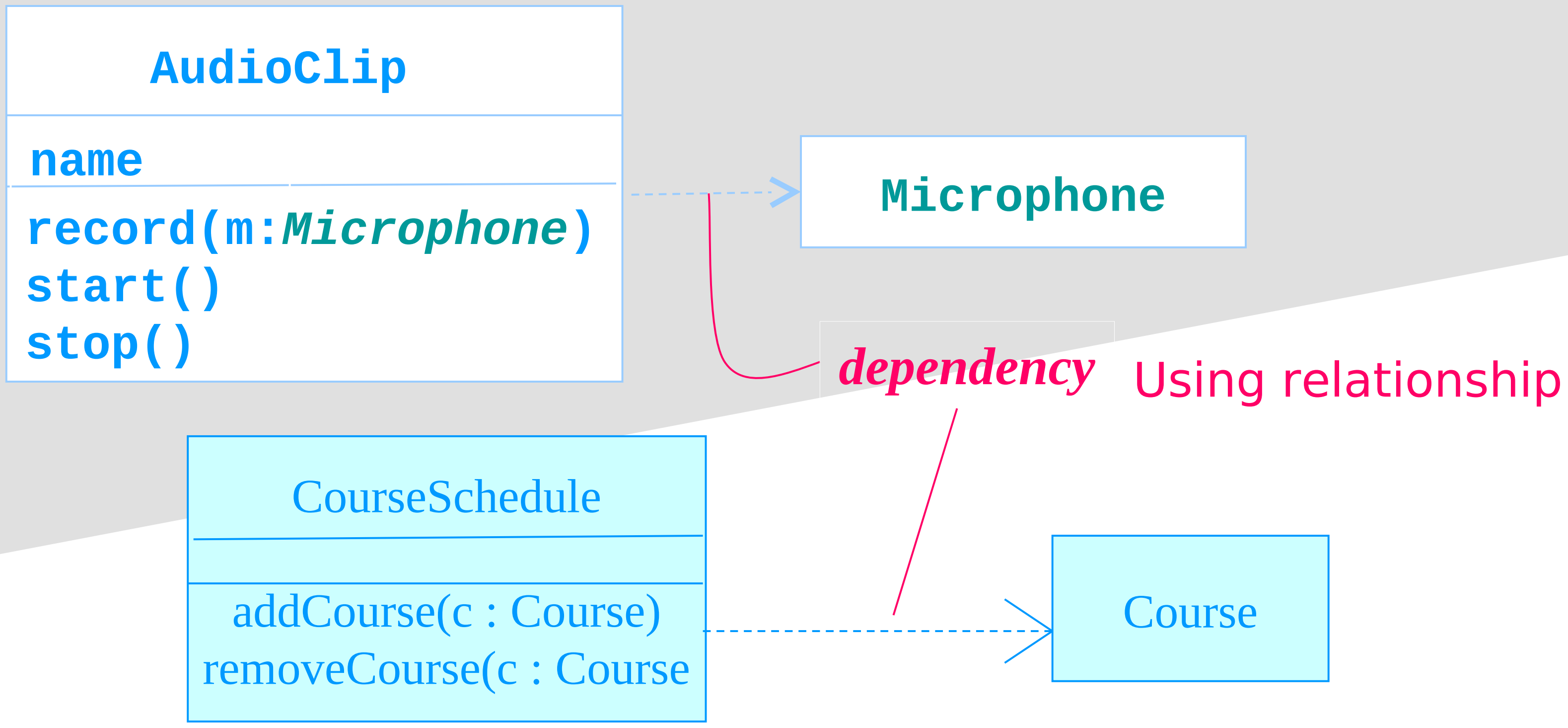
Relationships





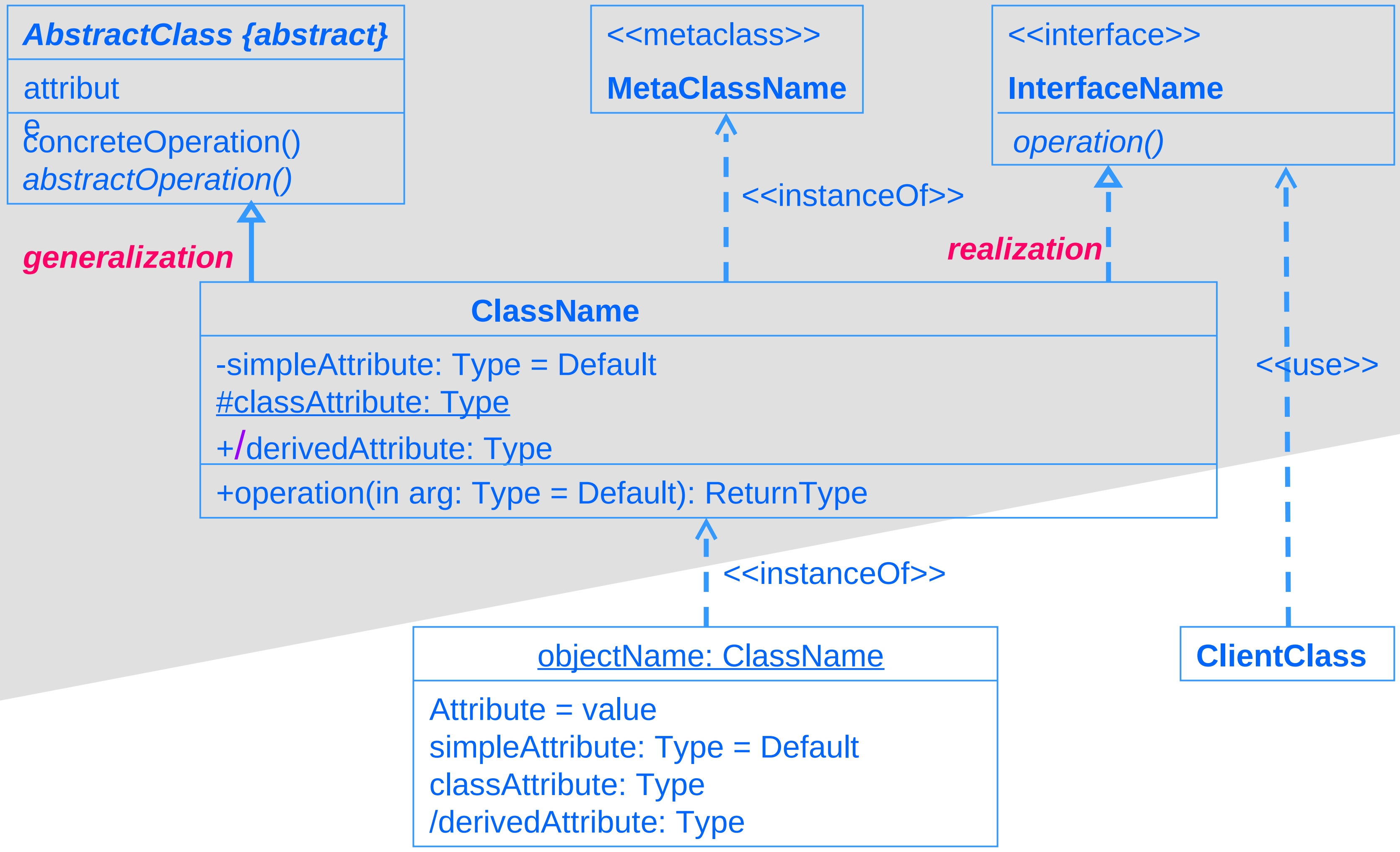
Dependency

- A change in one thing may affect another.
- The most common dependency between two classes is one where one class <<use>>s another as a *parameter to an operation*.



Usually initial class diagrams will not have any significant number of dependencies in the beginning of analysis but will as more details are identified.

Dependency – Among Classes

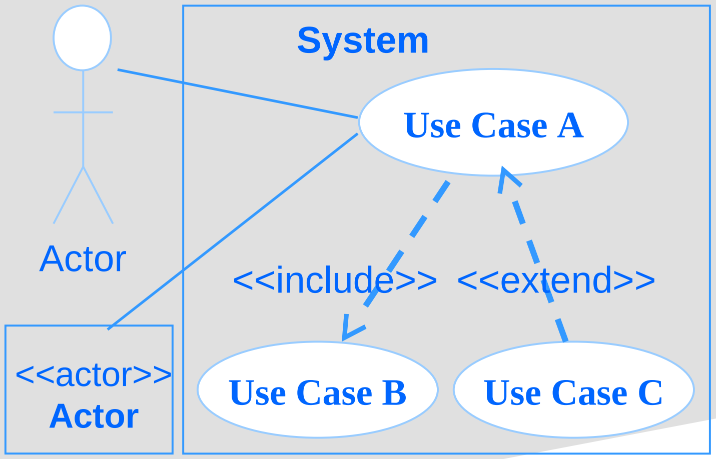


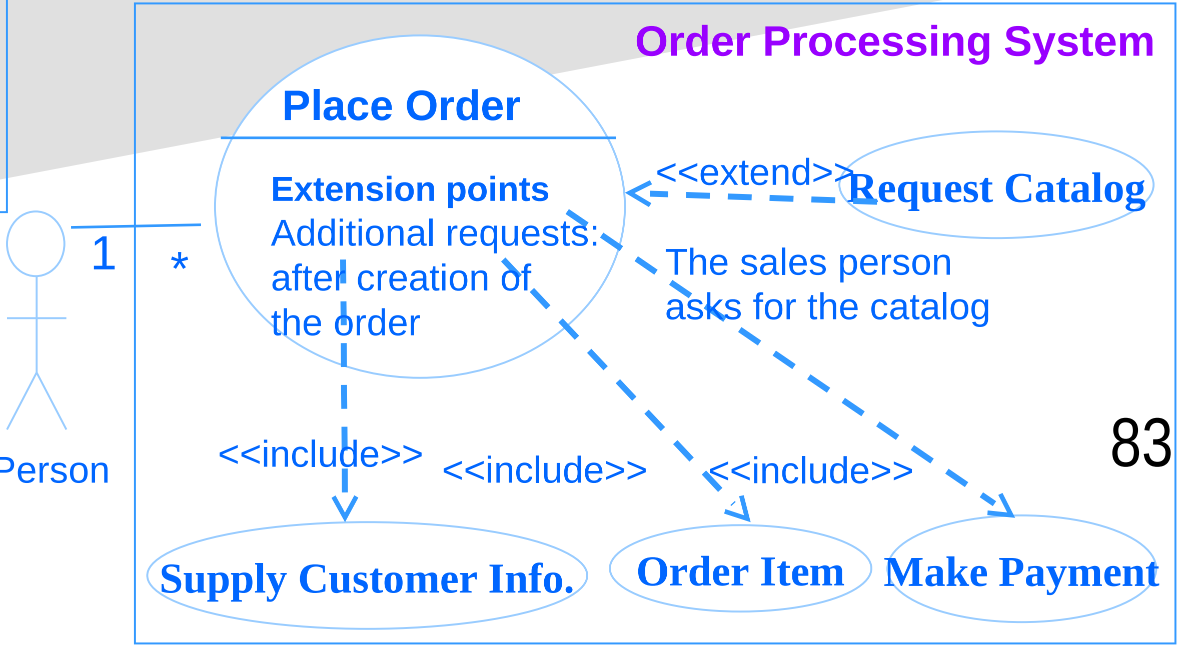
Dependency –Among Classes

- Eight Stereotypes of Dependency Among Classes
  - *bind*: the source instantiates the target template using the given actual parameters
  - *derive*: the source may be computed from the target
  - *friend*: the source is given special visibility into the target
  - ***instanceOf*** *:* the source object is an instance of the target classifier
  - *instantiate*: the source creates instances of the target
  - *powertype*: the target is a powertype of the source; a powertype is a classifier whose objects are all the children of a given parent
  - *refine*: the source is at a finer degree of abstraction than the target
  - *use*: the semantics of the source element depends on the semantics of the public part of the target

Dependency –Among Use Cases

- Two Stereotypes of Dependency Among Use Cases:
  - *extend*: the target use case extends the behavior of the source
  - *include*: the source use case explicitly incorporates the behavior of another use case *at a location* specified by the source



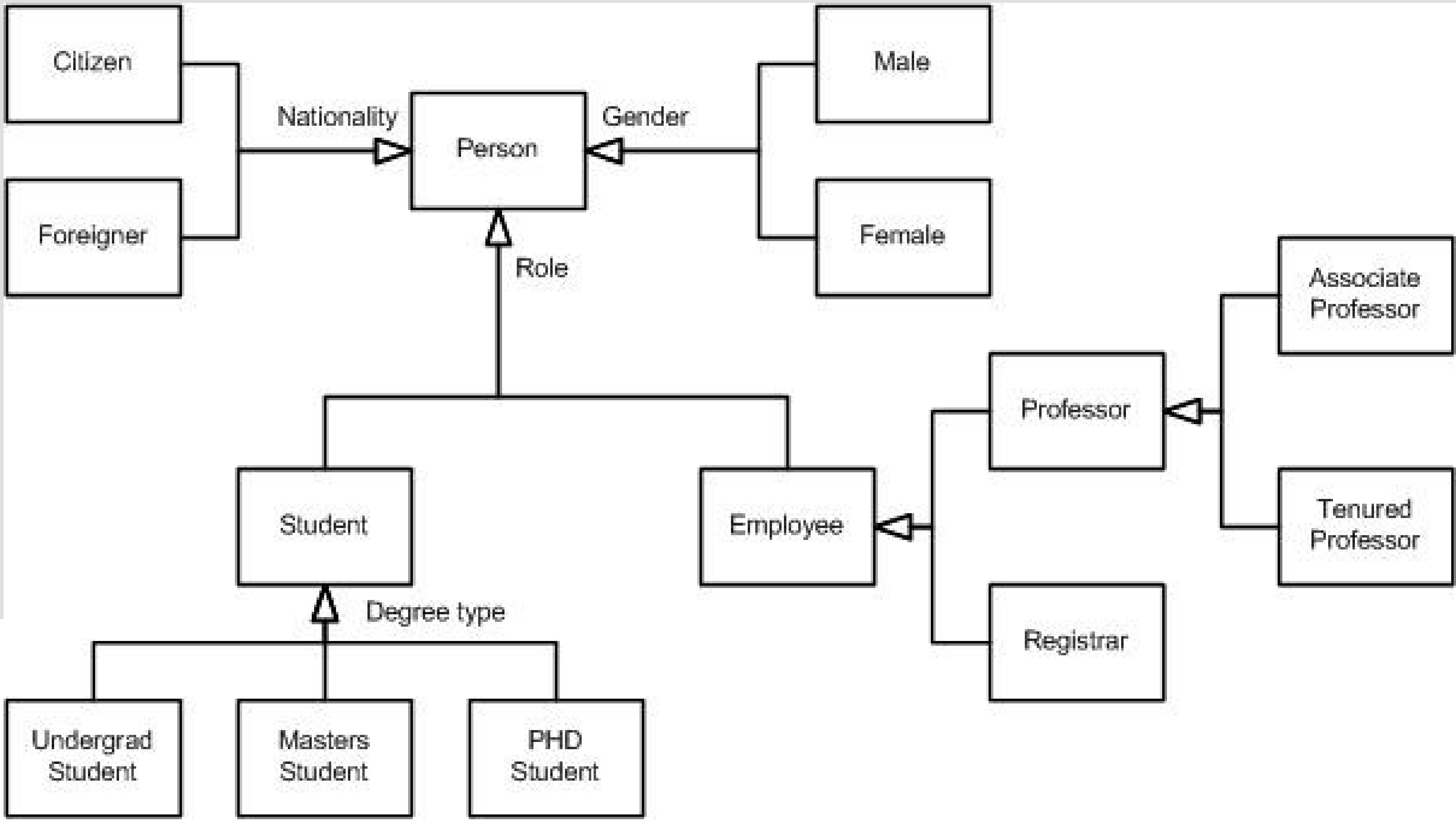


Generalization

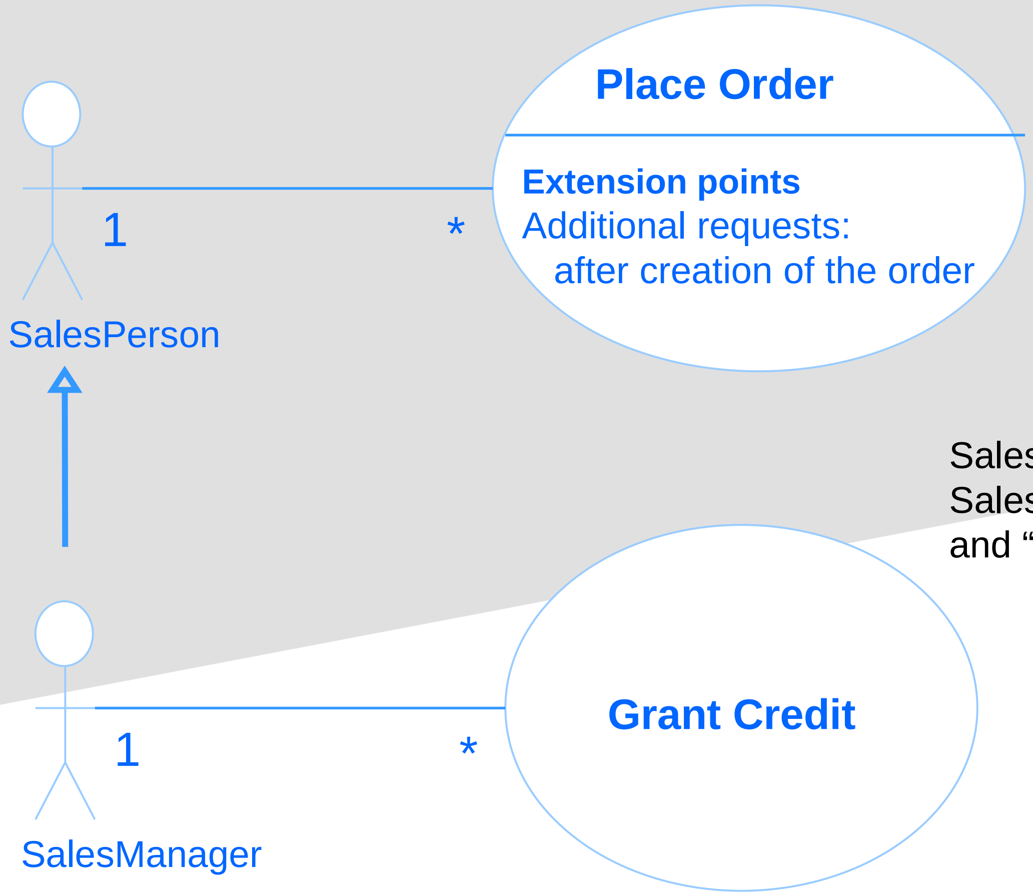
- Four Standard Constraints
  - *complete*: all children in the generalization have been specified; no more children are permitted
  - *incomplete*: not all children have been specified; additional children are permitted
  - *disjoint*: objects of the parent have no more than one of the children as a type
  - *overlapping*: objects of the parent may have more than one of the children as a type

- One Stereotype
  - *implementation*: the child inherits the implementation of the parent but does not make public nor support its interfaces

Generalization – Along Roles



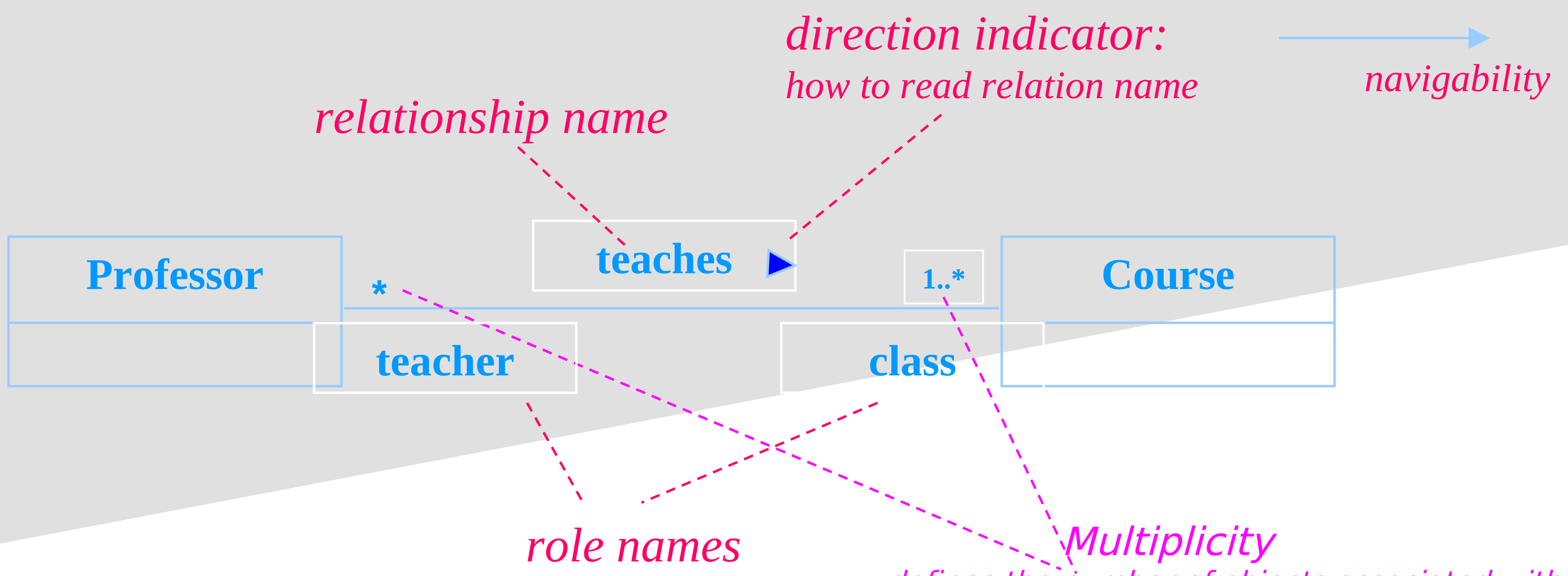
Generalization –Among Actors



Sales person can do only “Place Order”;
Sales manager can do both “Place Order” and “Grant Credit”

Associations

- Represent conceptual relationships between classes
  (cf. dependency with no communication/message passing)



defines the number of objects associated with an instance of the association.
***Default of 1**;*
Zero or more (*);
n..m; range from n to m inclusive

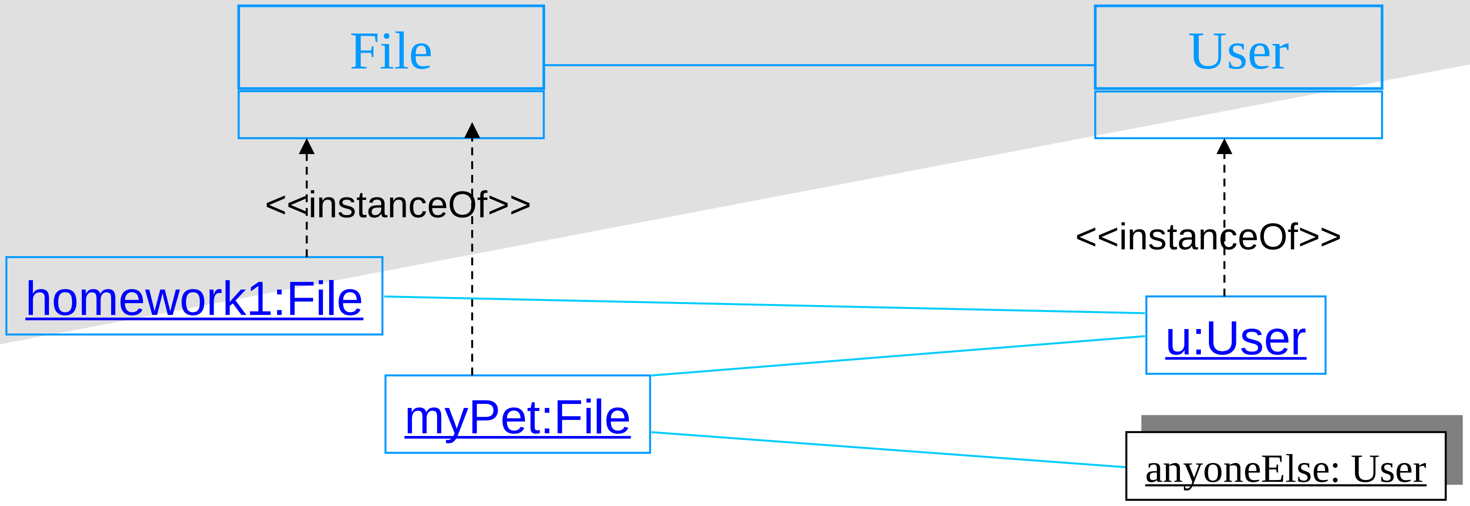
{visibility} {/} role name {: interface name}

Associations – A Question

- How would you model the following situation?

"You have two files, say homework1 and myPet, where homework1 is read-accessible only by you, but myPet is write-accessible by anybody."

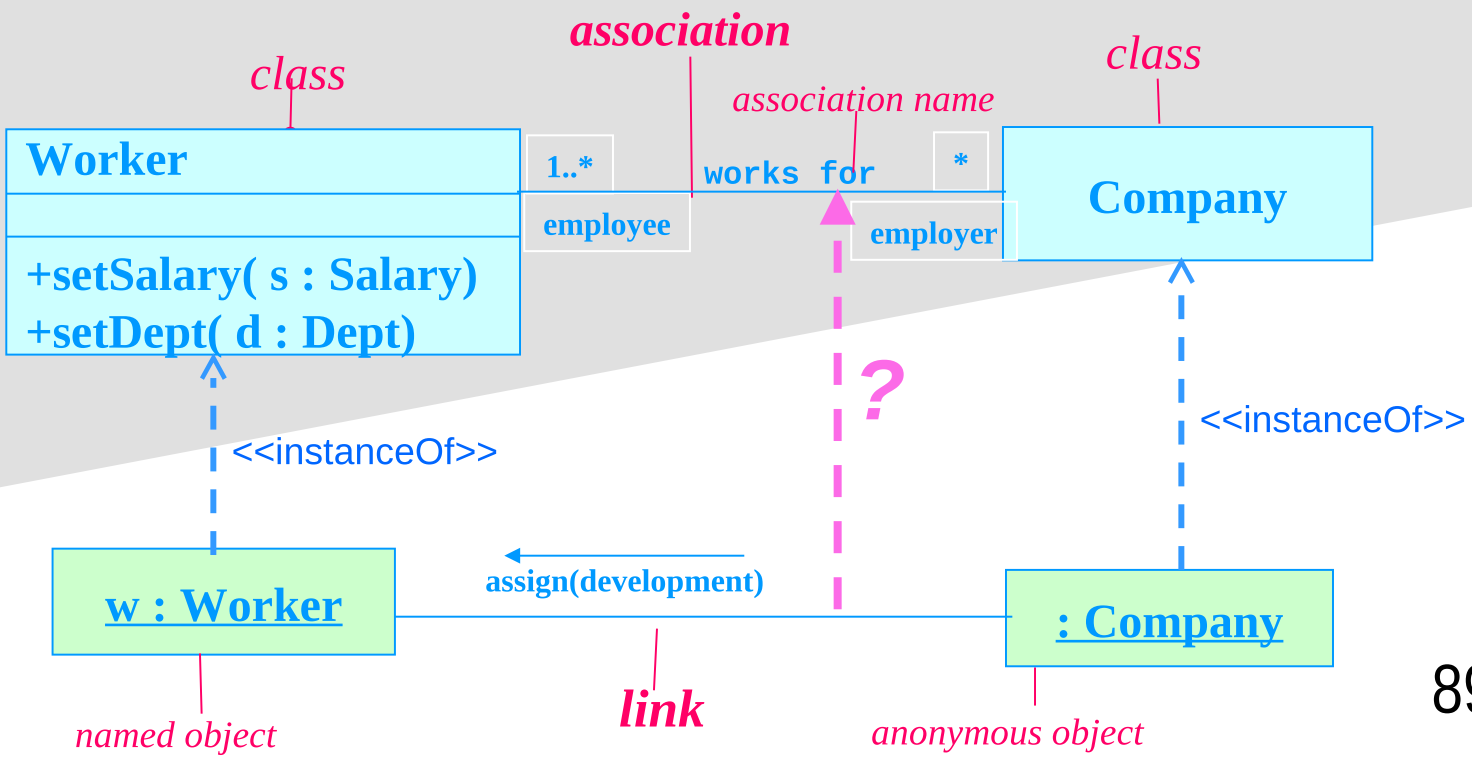
You could create two classes, File and User.
Homework1 and MyPet are files, and you are a user.



Approach 1: Now, would you associate the file access right with File?
Approach 2: Or, would you associate the file access right with User?

Associations – Links

- link is a semantic connection among objects.
- A link is an instance of an association.

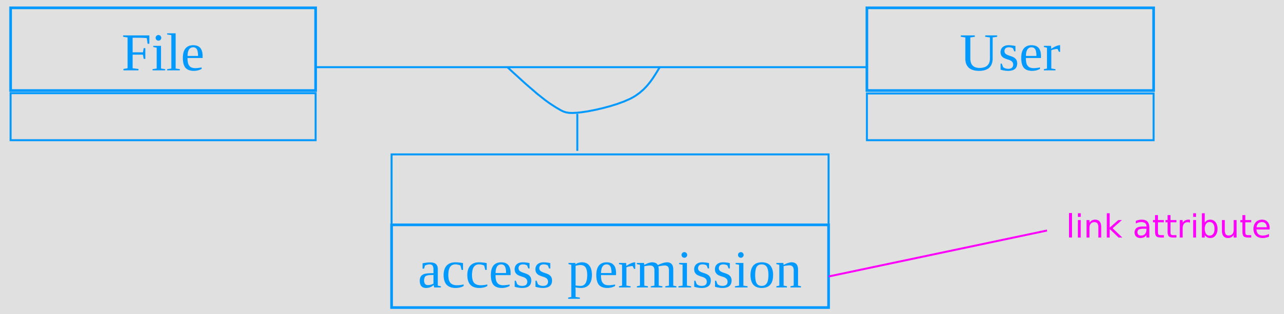


Association generalization is not automatic, but should be explicit in UML

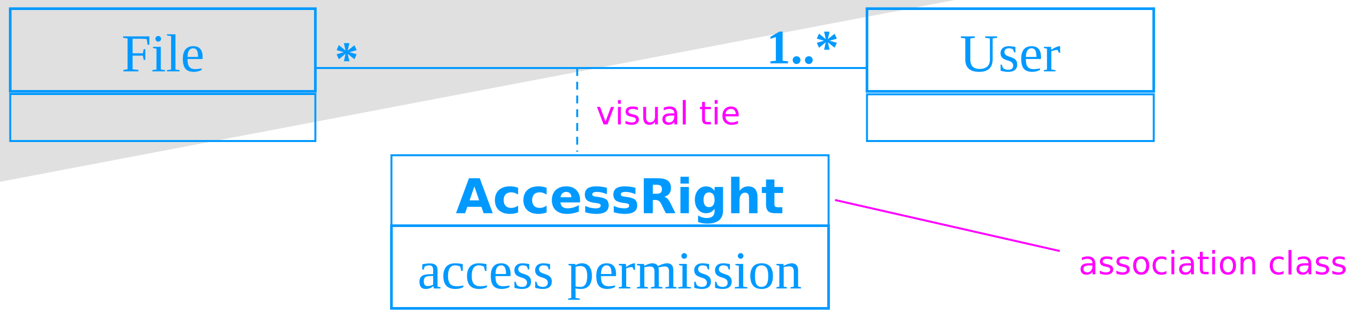
Associations – Link Attributes

- Link Attributes

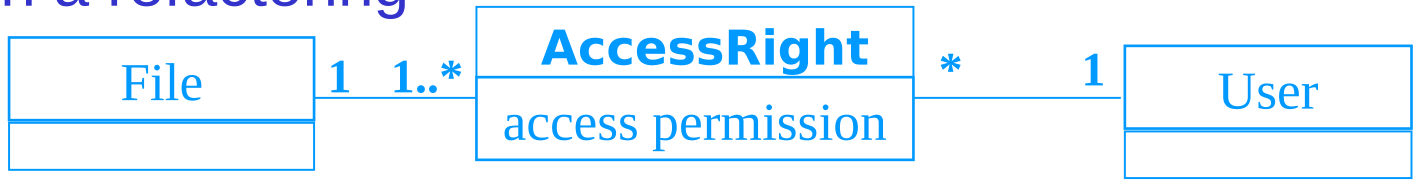
The most compelling reason for having link attributes is for-many-to-many relationships



- Association Class

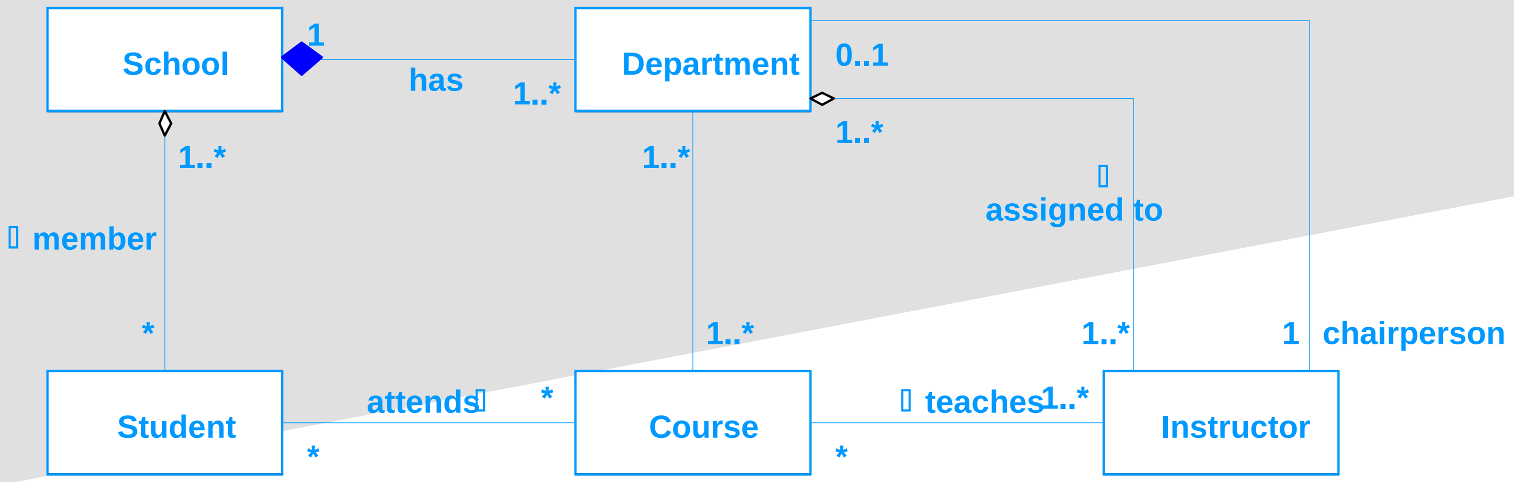


- With a refactoring



Modeling Structural Relationships

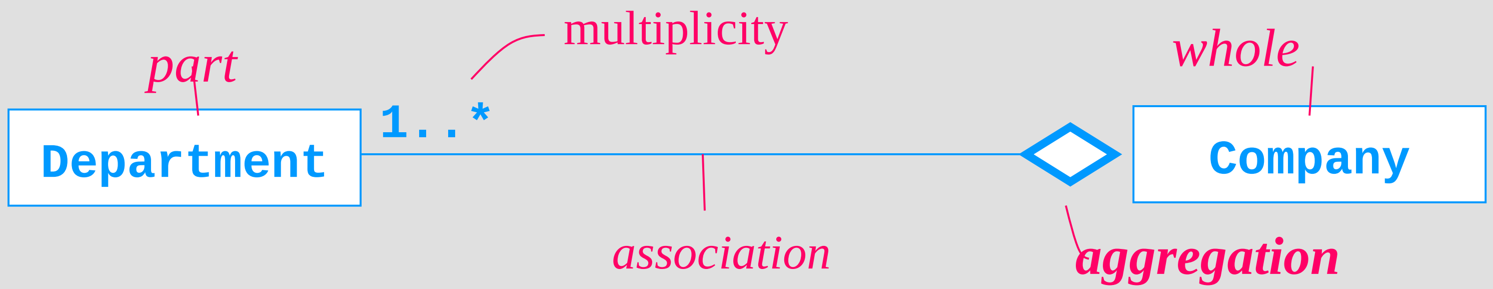
- Considering a bunch of classes and their association relationships



The model above is from Rational Rose. How did the composite symbol (♦)get loaded versus the aggregation? Use the Role Detail and select aggregation and then the "by value" radio button.

Modeling Structural Relationships

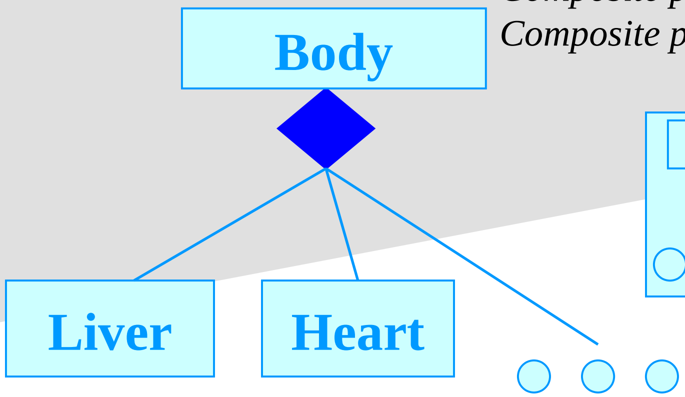
Aggregation - structural association representing "whole/part" relationship.
- "has-a" relationship.

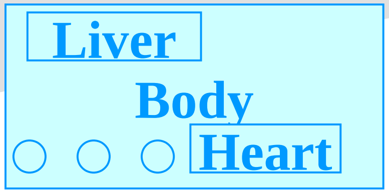


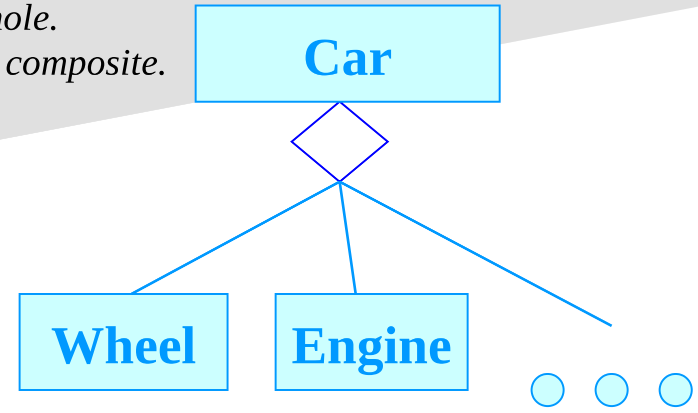
Composite

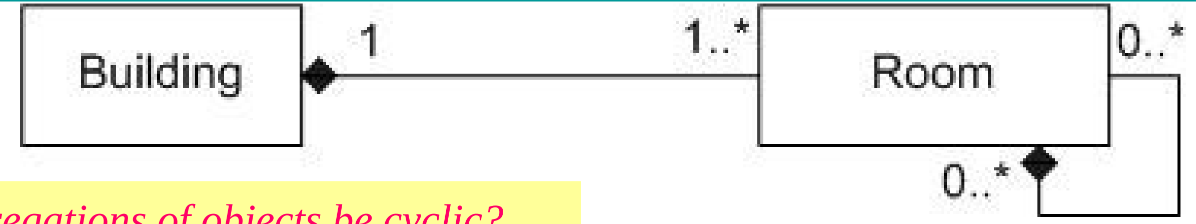
Composite is a stronger form of aggregation.
Composite parts live and die with the whole.
Composite parts may belong to only one composite.

Part -> whole?



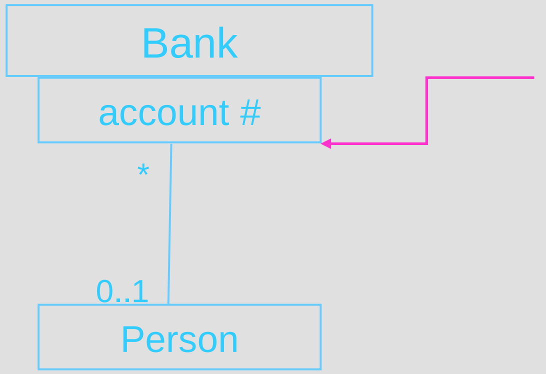




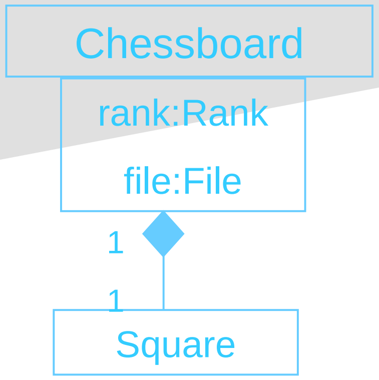


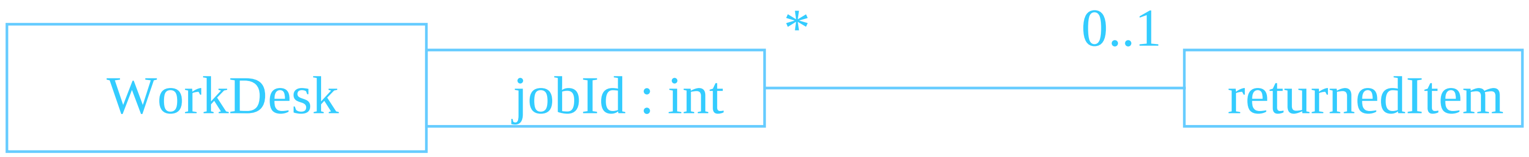
Can aggregations of objects be cyclic?

Association – Qualification

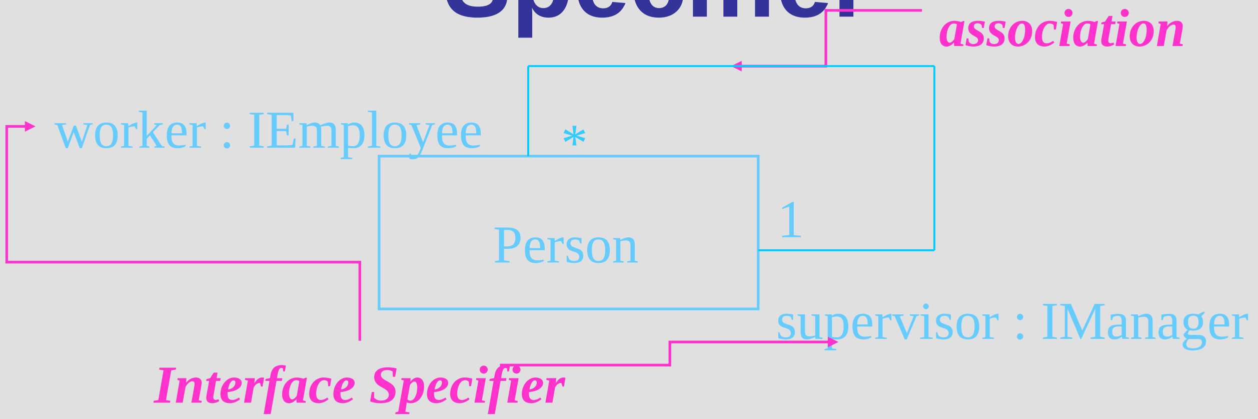


Qualifier,
cannot access person without knowing the account #



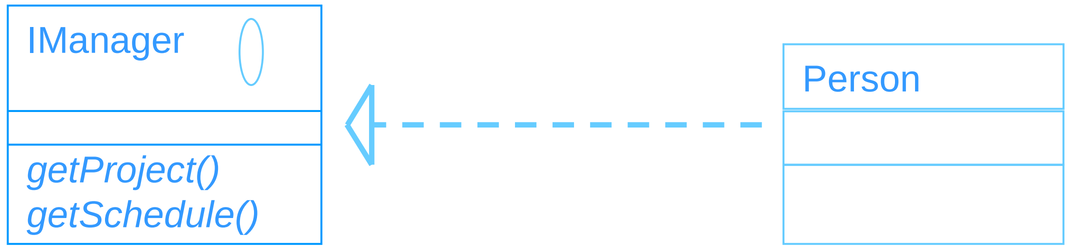


Association – Interface Specifier



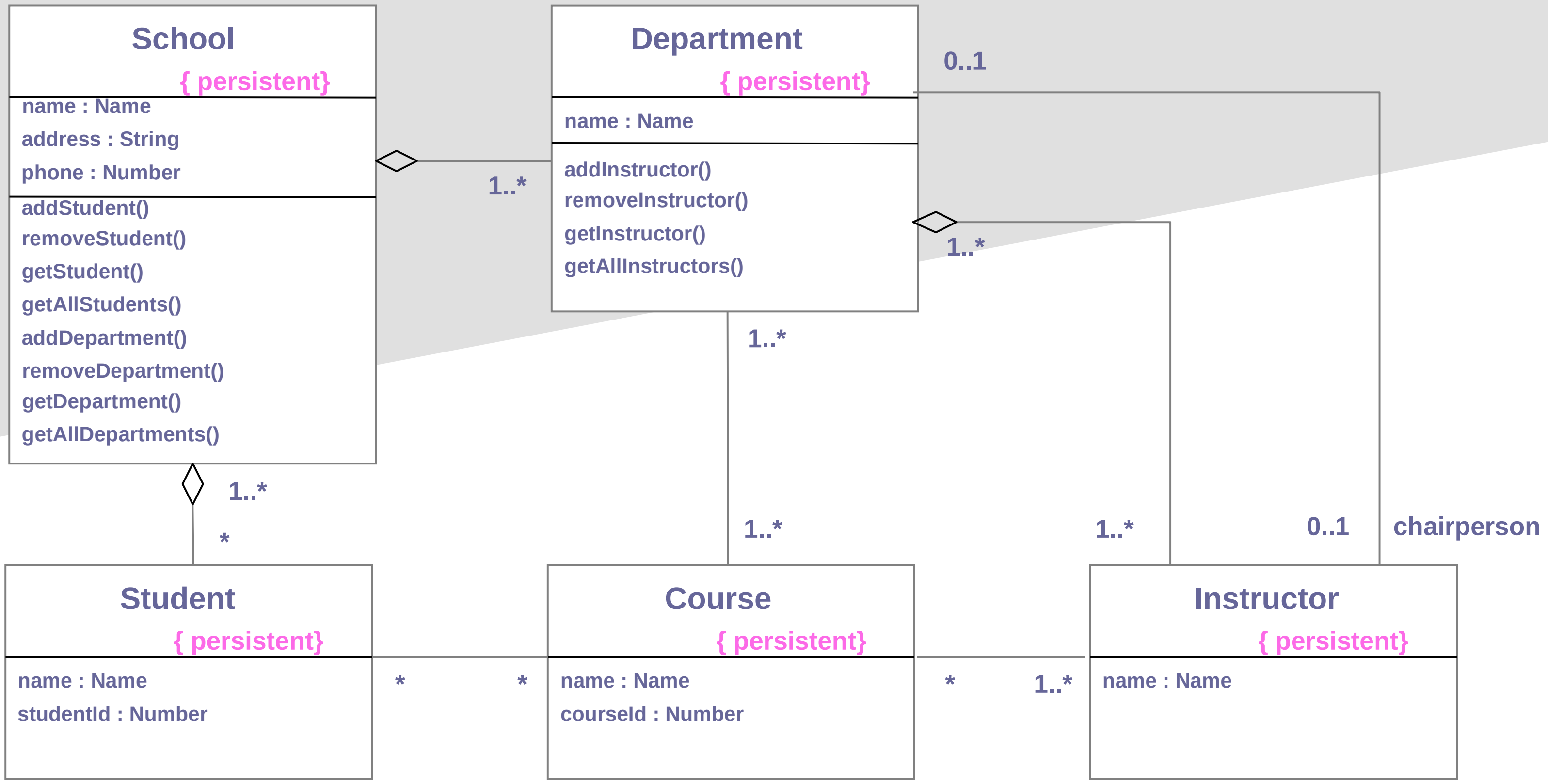
Realization

- A semantic relationship between classifiers in which one classifier specifies a contract that another guarantees to carry out.
- In the context of interfaces and collaborations
- **An interface can be realized by many classes/components**
- **A class/component may realize many interfaces**



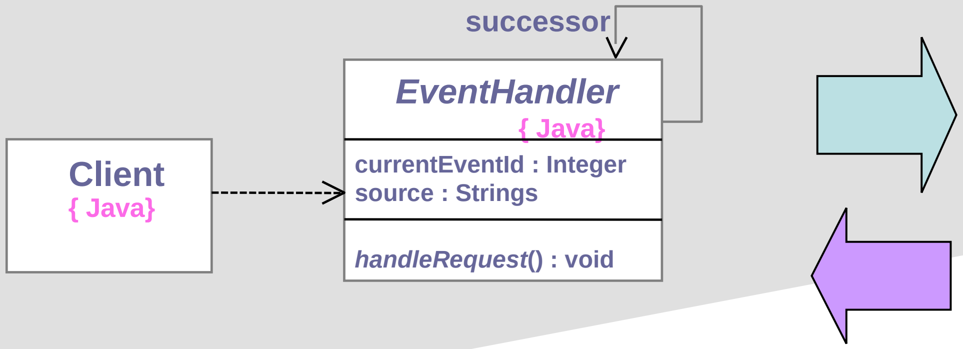
Modeling a Logical Database

- Class diagrams to provide more semantics
- From a general class diagram, first identify classes whose state must be persistent (e.g. after you turn off the computer, the data survives, although the program doesn't).
- Create a class diagram using standard tagged value, (e.g. {persistent}).
- Include attributes and associations.
- Use tools, if available, to transform logical design (e.g., tables and attributes) into physical design (e.g., layout of data on disk and indexing mechanisms for fast access to the data).



Forward/ Reverse Engineering

- translate a collaboration into a logical database schema/operations
- transform a model into code through a mapping to an implementation language.
- Steps
  - Selectively use UML to match language semantics (e.g. mapping multiple inheritance in a collaboration diagram into a programming language with only single inheritance mechanism).
  - Use **tagged values** to identify language..



```
public abstract class EventHandler
{
    private EventHandler
successor;
    private Integer
currentEventId;
    private String source;

    EventHandler() {}
    public void handleRequest() {}
}
```

- translate a logical database schema/operations into a collaboration
- transform code into a model through mapping from a specific implementation language

Object Diagrams

Structural Diagrams

- Class;

  Object
- Component
- Deployment
- **Composite Structure**
- Package

Instances & Object Diagrams

- “instance” and “object” are largely synonymous; used interchangeably.

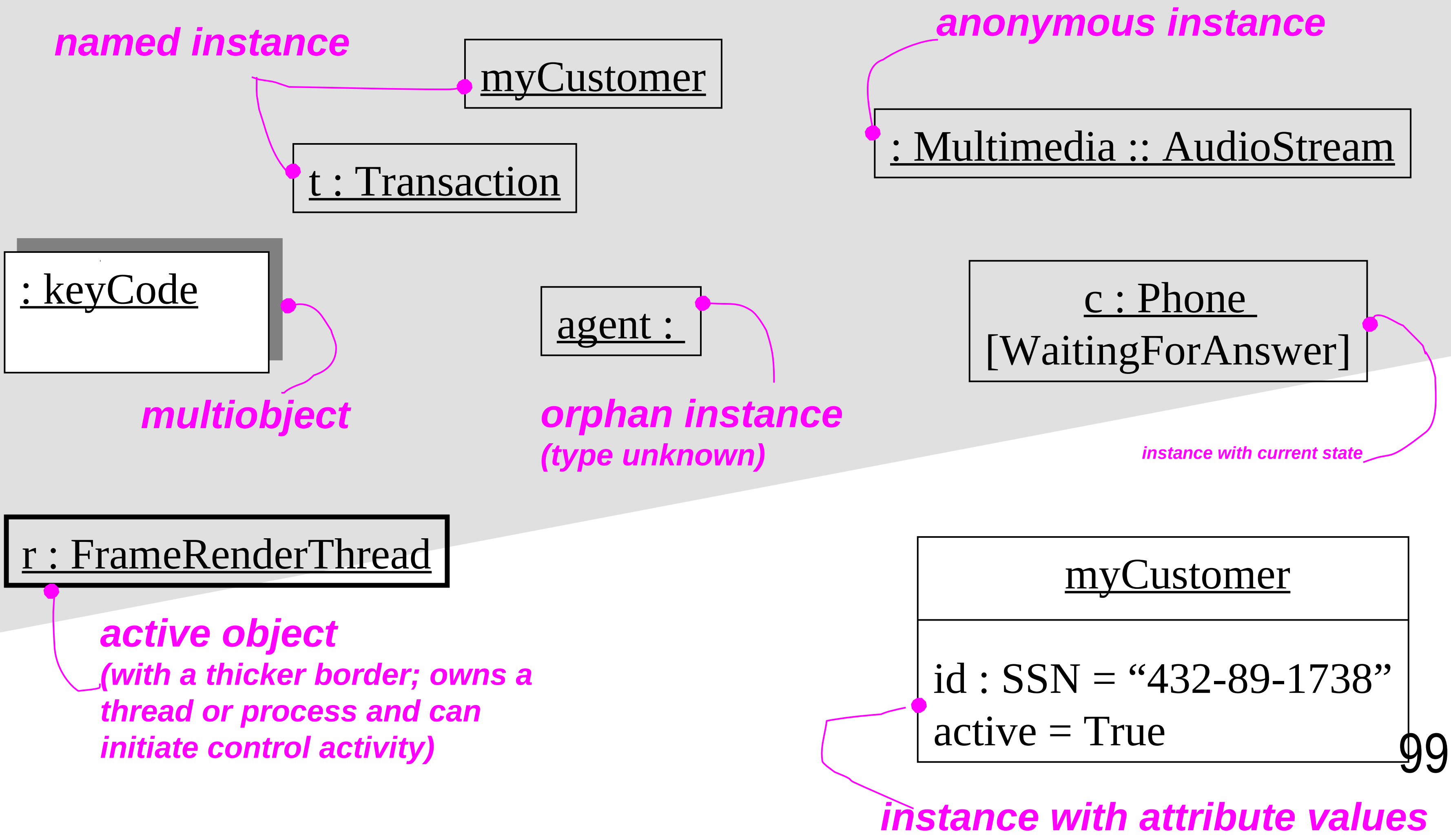
- difference:
  - instances of a **class** are called objects or instances; but
  - instances of other abstractions (components, nodes, use cases, and associations) are not called objects but only instances.

What is an instance of an association called?

Object Diagrams

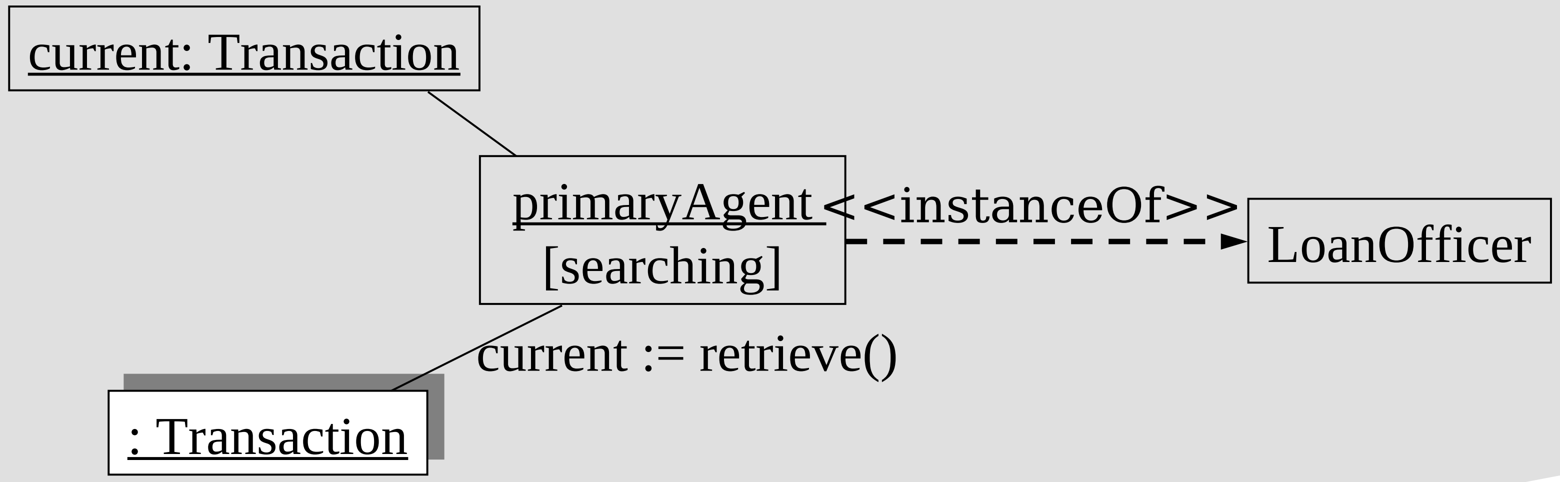
- very useful in debugging process.
  - walk through a scenario (e.g., according to use case flows).
  - Identify the set of objects that collaborate in that scenario (e.g., from use case flows).
  - Expose these object’s states, attribute values and links among these objects.

Instances & Objects - Visual Representation



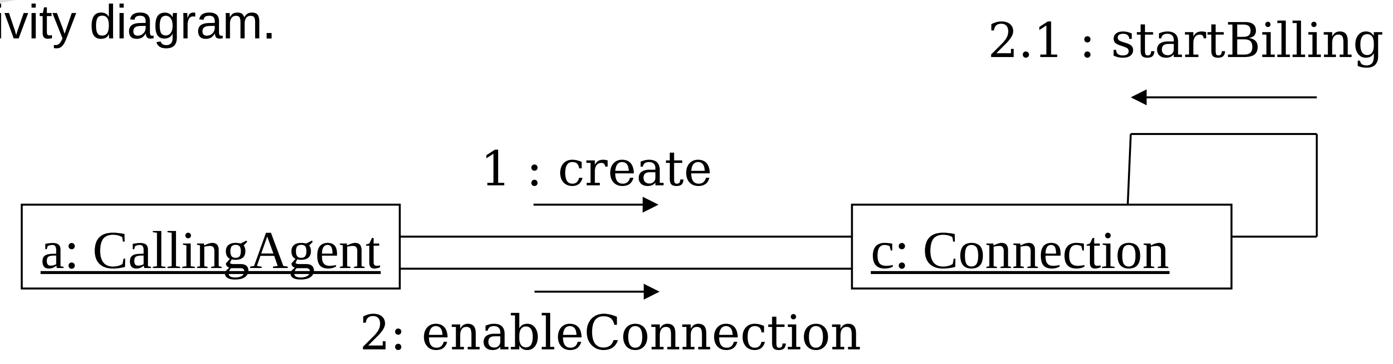
Instances & Objects - Modeling Concrete Instances

- Expose the stereotypes, tagged values, and attributes.
- Show these instances and their relationships in an object diagram.

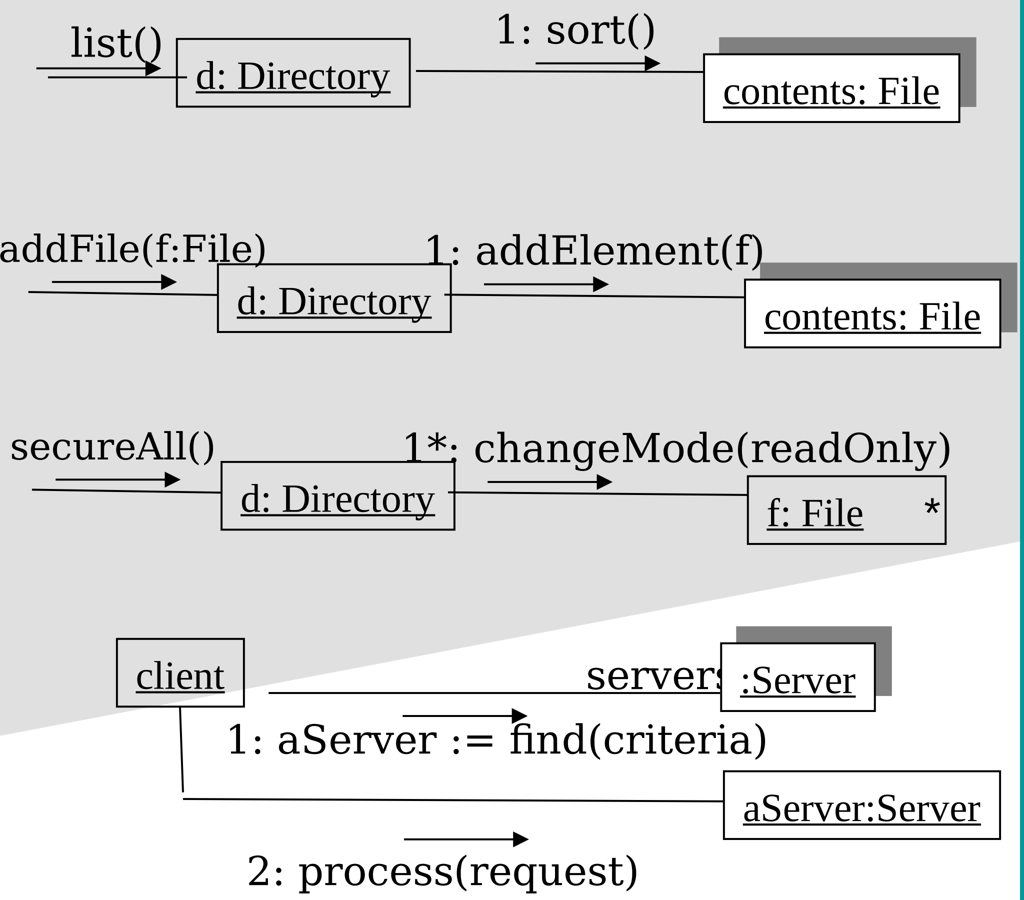


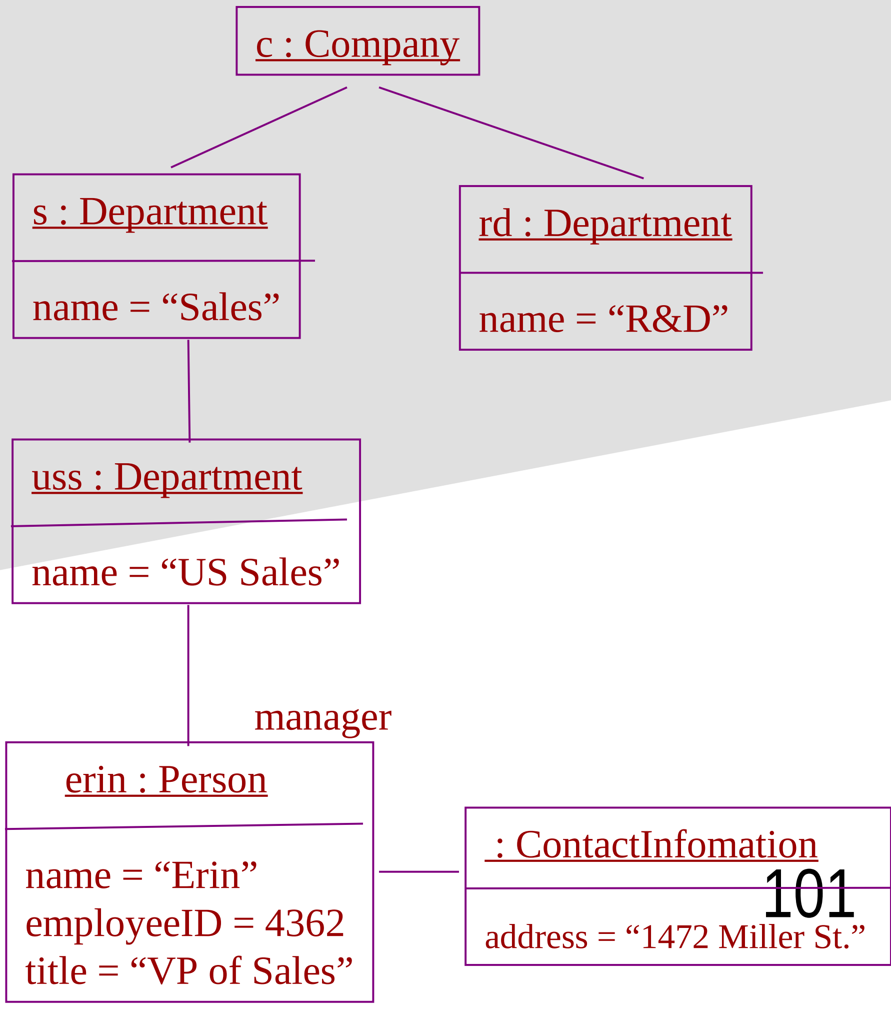
Instances & Objects - Modeling Prototypical Instances

- Show these instances and their relationships in an interaction diagram or an activity diagram.



Instances & Objects – More Examples





call ::= label [guard] ["*"] [return-val-list ":="] msg-name "(" arg-list ")"

Component Diagrams

Structural Diagrams

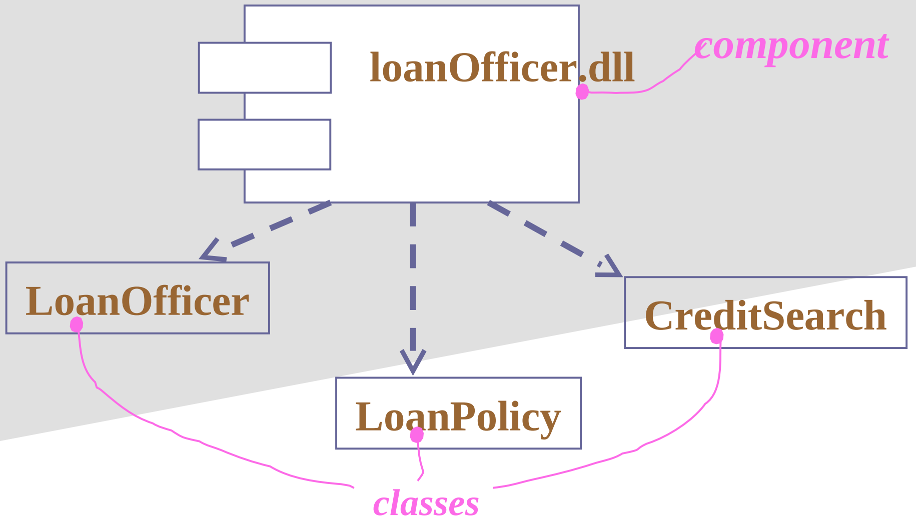
- Class;
  Object
- Component
- Deployment
- **Composite Structure**
- Package

Component Diagram

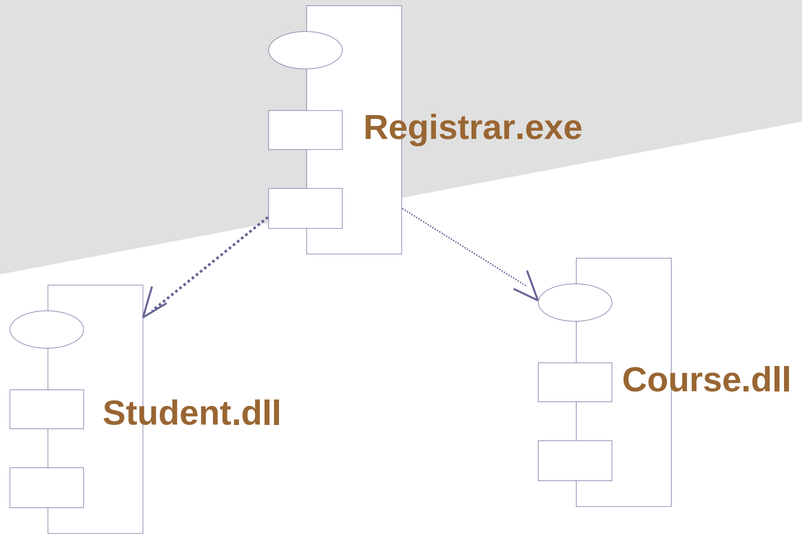
UML1.x – implementation view

- Shows a set of components and their relationships.
- Represents the static implementation view of a system.
- Components map to one or more classes, interfaces, or collaborations.

Mapping of Components into Classes



Components and their Relationships



Component Diagram UML2.0 – architectural view

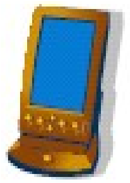
Big demand, hmm…

- Short history behind architecture
- Architecture still an emerging discipline
- Challenges, a bumpy road ahead

- UML and architecture evolving in parallel
- Component diagram in need of better formalization and experimentation

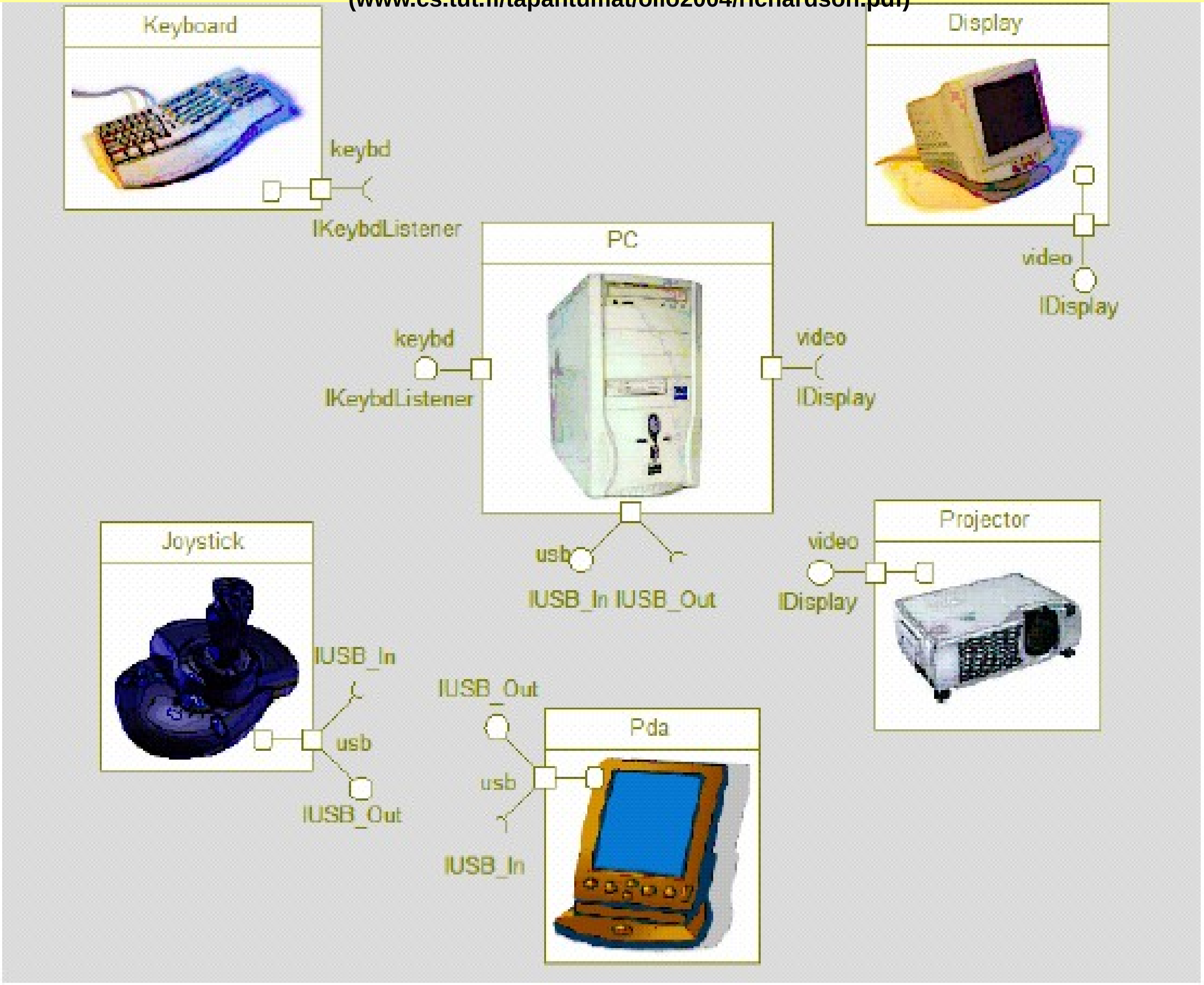
Component Diagram – another example

(www.cs.tut.fi/tapahtumat/olio2004/richardson.pdf)



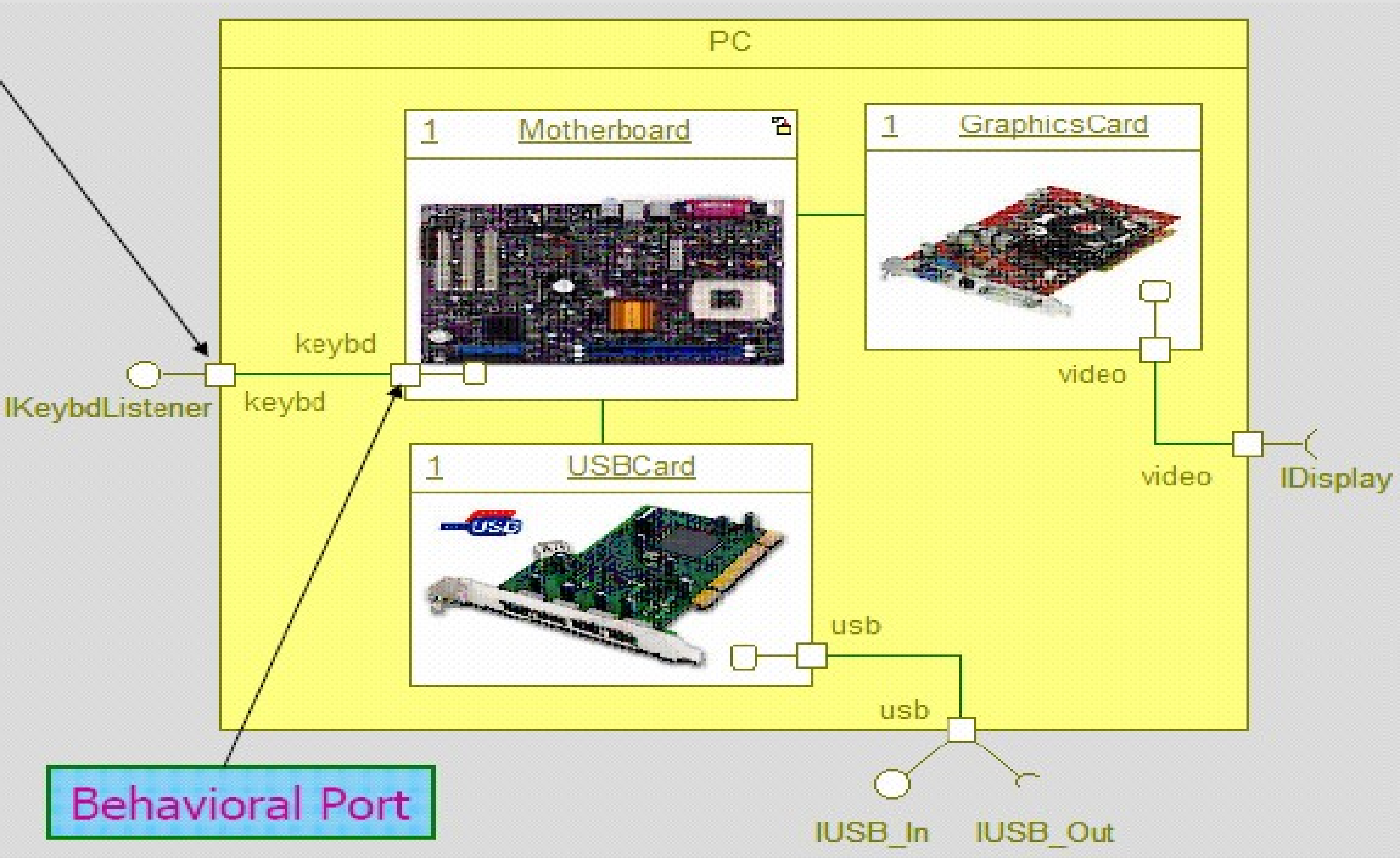
Component Diagram – another example

(www.cs.tut.fi/tapahtumat/olio2004/richardson.pdf)



Component Diagram – another example

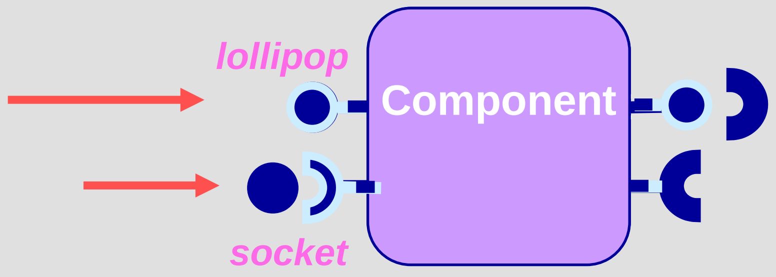
(www.cs.tut.fi/tapahtumat/olio2004/richardson.pdf)



Component Diagram UML2.0 – architectural view

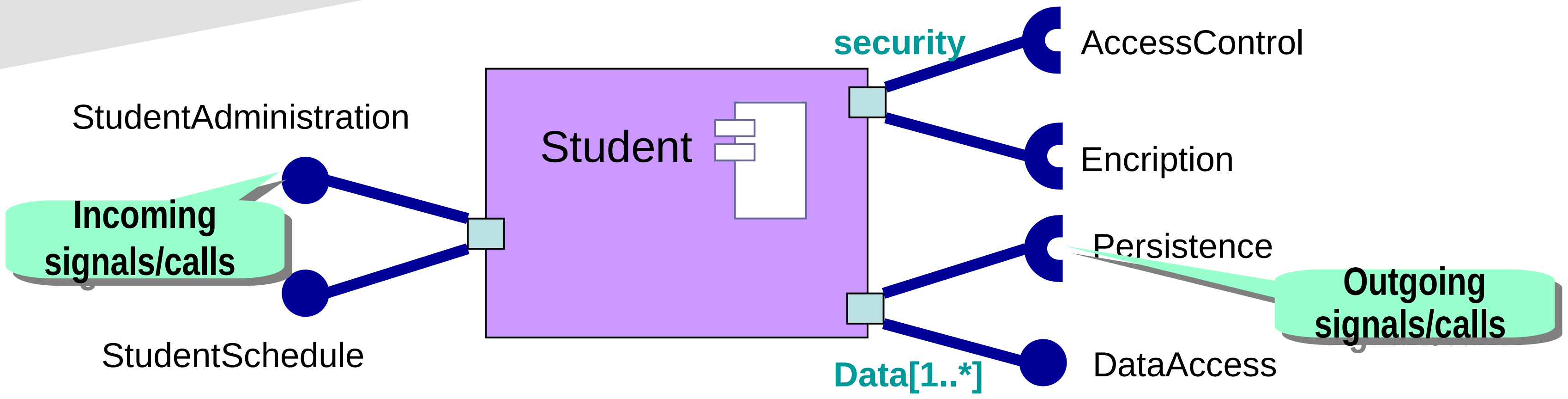
Explicit description of *interfaces*:

- provided services to other components
- requested services from other components



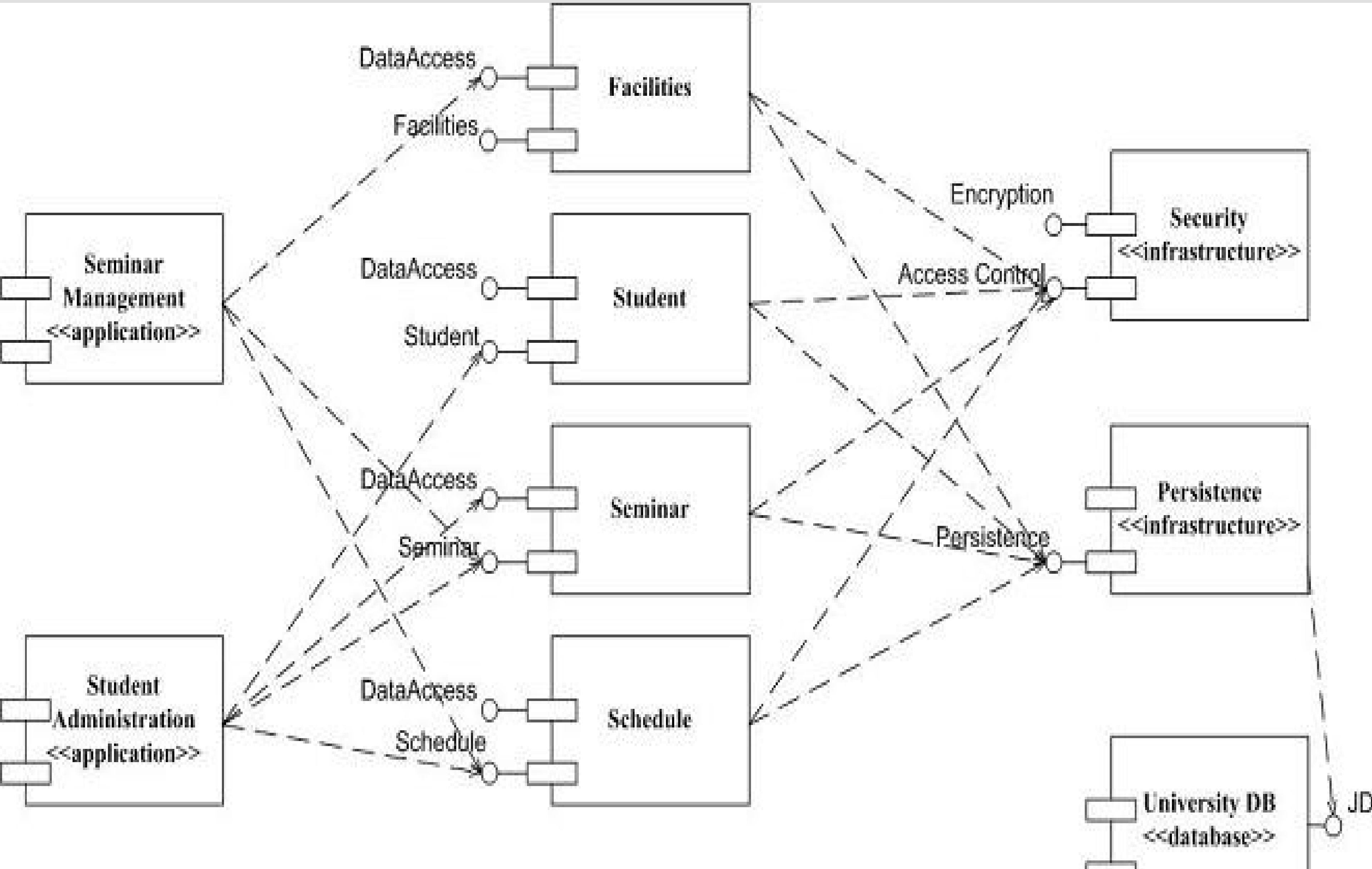
- An interface is a collection of 1..* methods, and 0..* attributes
- Interfaces can consist of synchronous and / or asynchronous operations
- A *port* (*square*) is an interaction point between the component and its environment.
- Can be named; Can support uni-directional (either provide or require) or bi-directional (both provide and require) communication; Can sup multiple interfaces.
- possibly concurrent interactions
- fully isolate an object's internals from its environment

caller or callee?



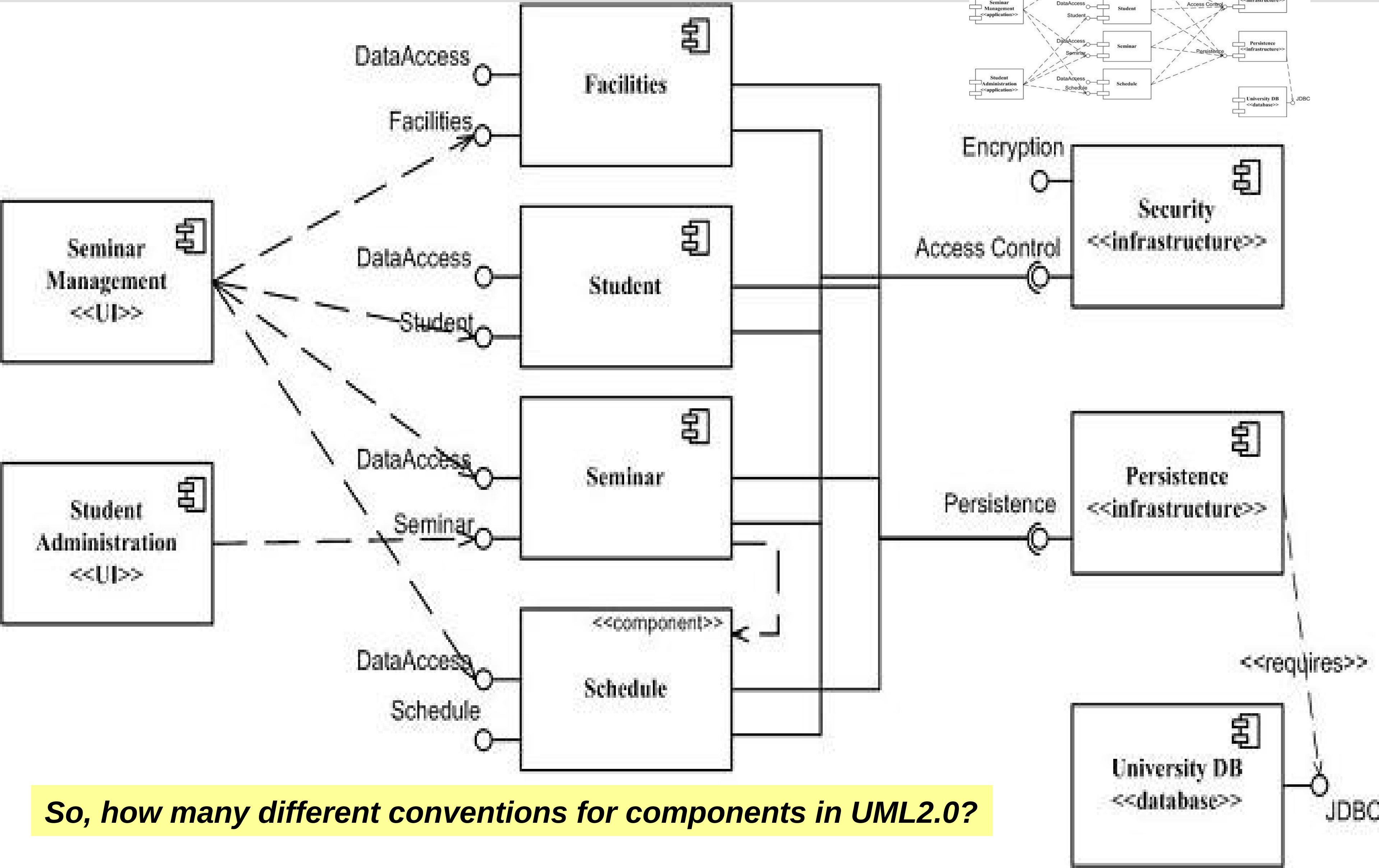
Component Diagram: UML 1.x and UML 2.0

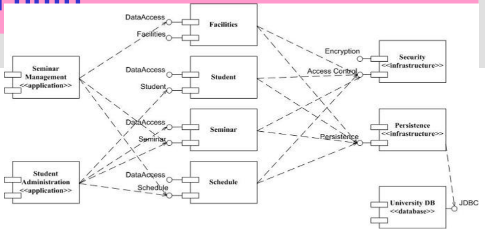
(http://www.agilemodeling.com/artifacts/componentDiagram.htm)


Facilities
DataAccess
Facilities
Seminar Management <<application>>
Student
DataAccess
Student
Seminar
DataAccess
Seminar
Student Administration <<application>>
Schedule
DataAccess
Schedule
Security <<infrastructure>>
Encryption
Access Control
Persistence <<infrastructure>>
Persistence
University DB <<database>>
JD

Component Diagram: UML 1.x and UML 2.0

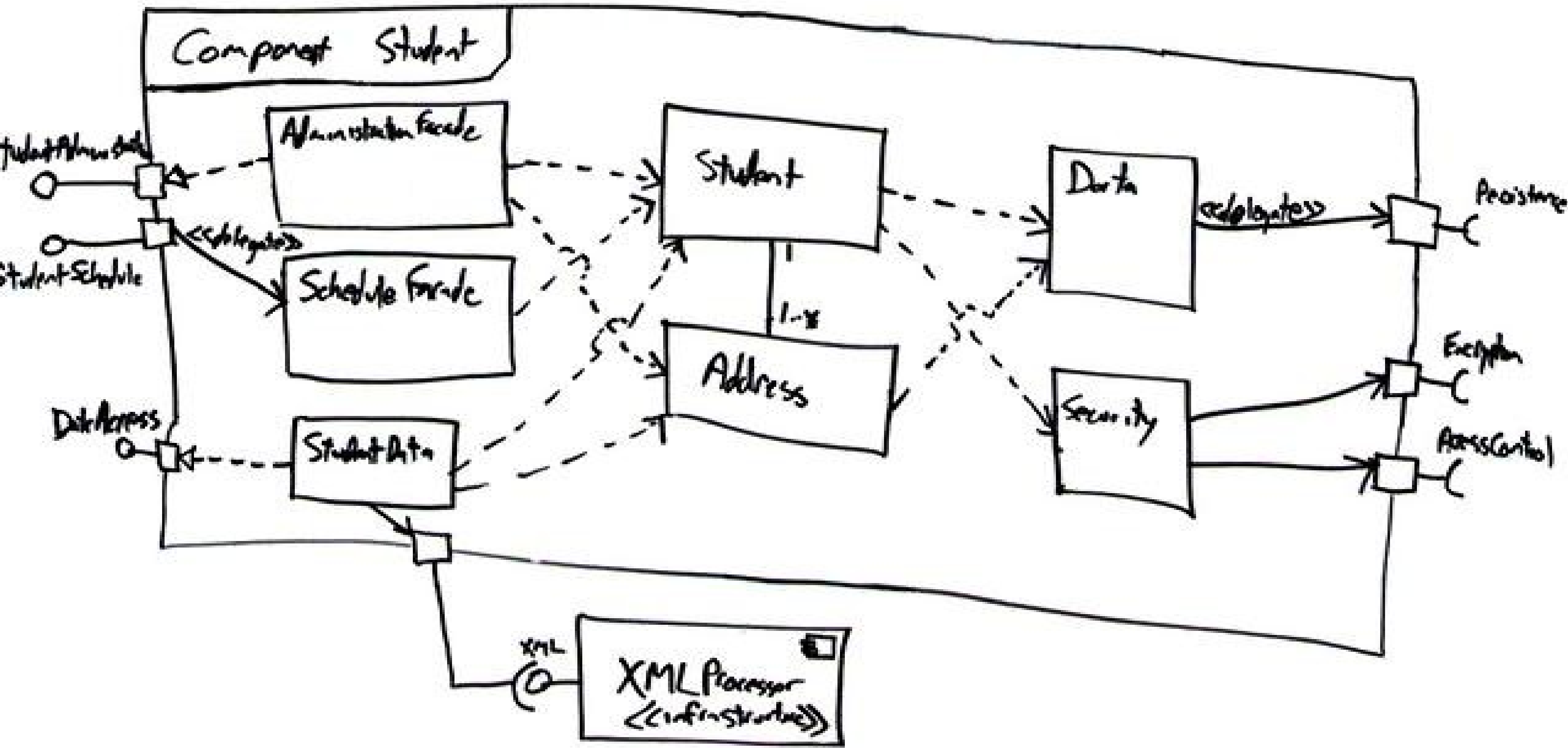
(http://www.agilemodeling.com/artifacts/componentDiagram.htm)




DataAccess
Facilities
Facilities
Seminar Management <<application>>
DataAccess
Student
Student
DataAccess
Seminar
Seminar
Student Administration <<application>>
DataAccess
Schedule
Schedule
Encryption
Access Control
Security <<infrastructure>>
Persistence
Persistence <<infrastructure>>
University DB <<database>>
JDBC

So, how many different conventions for components in UML2.0?

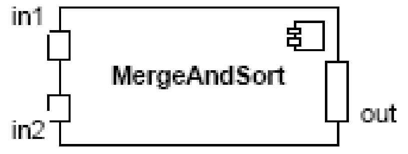
Building a Component



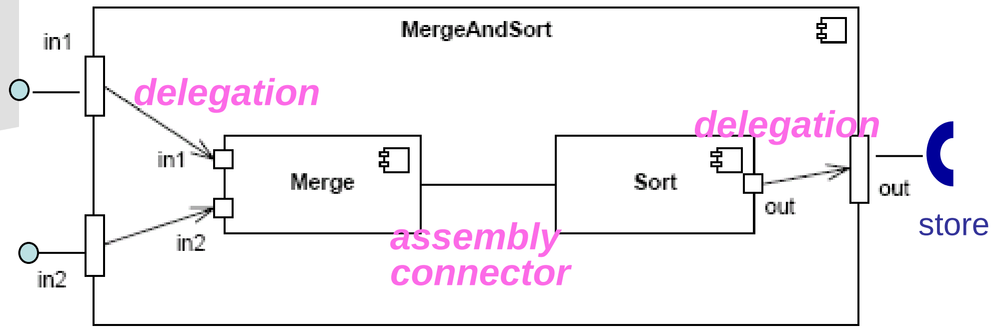
- simplified the ports to either provide or require a single interface
- relationships between ports and internal classes in three different ways:
  i) as ***stereotyped delegates*** (flow), as ***delegates***, and as ***realize***s (logical->physical) relationships
- Cohesive reuse and change of classes; acyclic component dependency ***???***

Component Diagram – Connector & Another Example

- a connector: just a link between two or more connectable elements (e.g., ports or interfaces)
- 2 kinds of connectors: *assembly* and *delegation*. For ***"wiring"***
  - An ***assembly*** connector: a binding between a provided interface and a required interface (or ports) that indicates that one component provides the services required by another; ***simple line/ball-and-socket/lollipop-socket notation***
  - A ***delegation*** connector binds a component's external behavior (as specified at a port) to an internal realization of that behavior by one of its parts *(provide-provide, request-request).*



External View of a Component with Ports



Internal View of a Component with Ports

Left delegation: direction of arrowhead indicates "provides"

So, what levels of abstractions for connections?

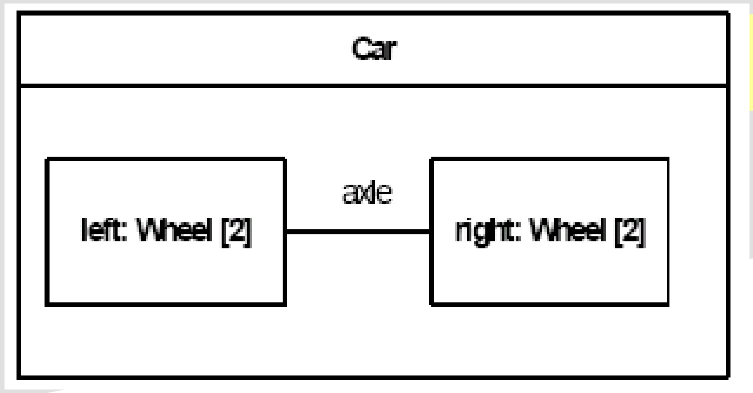
Right delegation: direction of arrowhead indicates "requests"

Structured Class

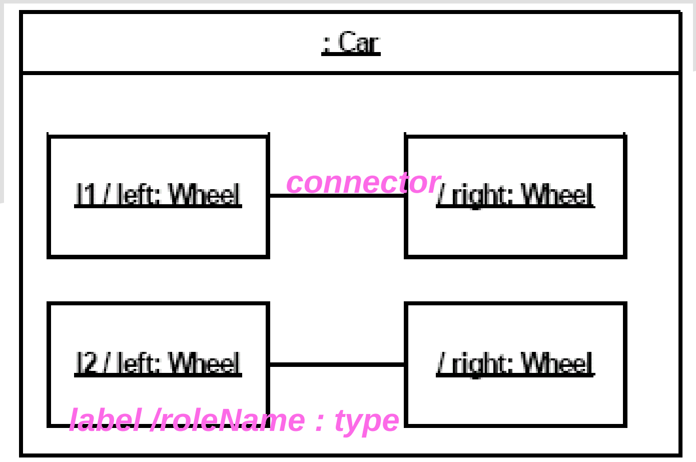
- A *structured class(ifier)* is defined, in whole or in part, in terms of a number of *parts* - contained instances owned or referenced by the structured class(ifier).
  - With a similar meaning to a **composit**

Any difference?

- A structured classifier's parts are created within the containing classifier (either when the structured classifier is created or later) and are destroyed when the containing classifier is destroyed.
- Like classes and components, combine the descriptive capabilities of structured classifiers with ***ports and interfaces***



component or class?

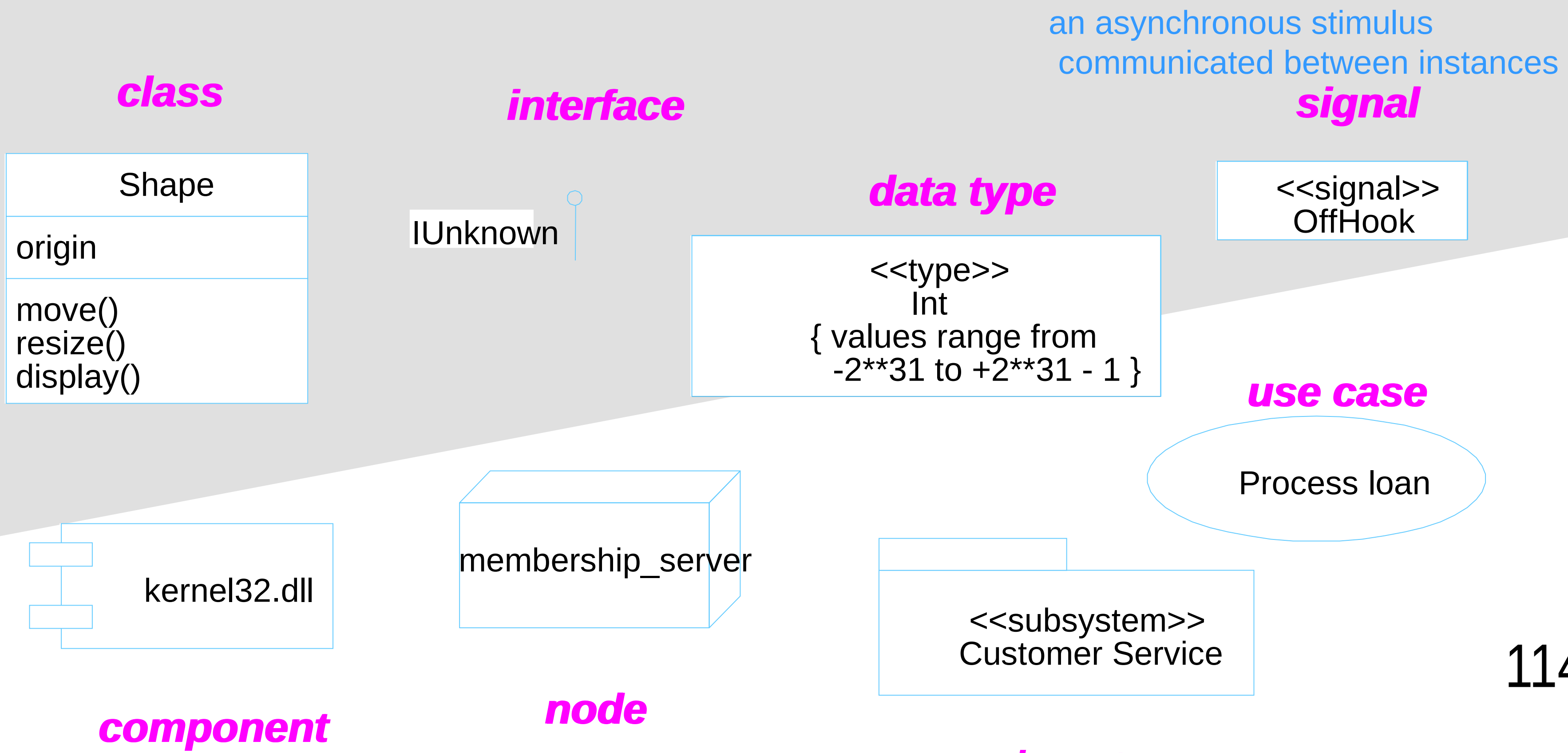


Components extend classes with additional features such as

- the ability to ***own more types of elements*** than classes can; e.g., packages, constraints, use cases, and artifacts
- deployment specifications that define the execution parameters of a component deployed to a node

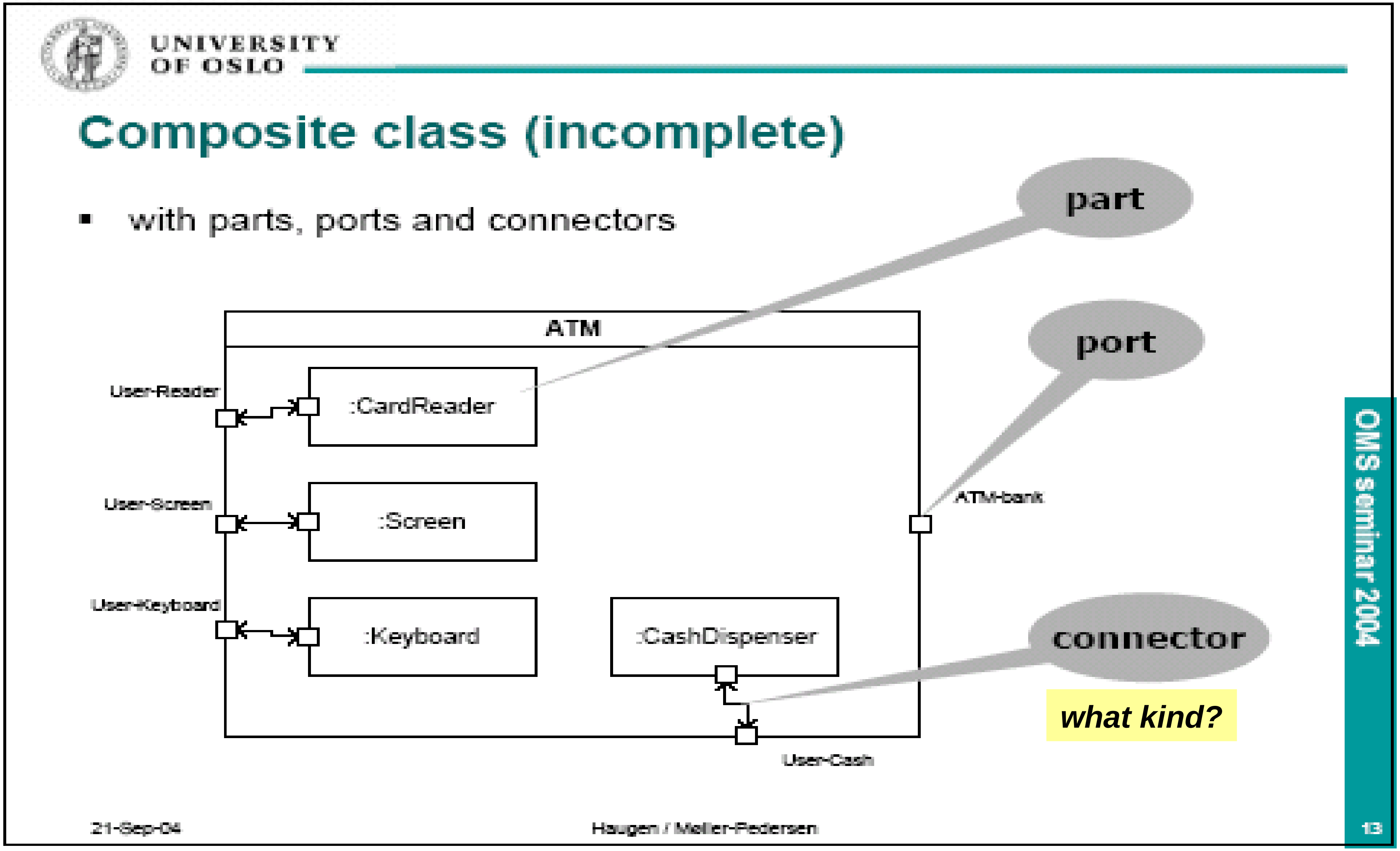
Classifiers

- Classifier—mechanism that describes structural (e.g. class attributes) and behavioral (e.g. class operations) features. In general, those *modeling elements* that can have *instances* are called classifiers.
  - *cf. Packages and generalization relationships do not have instances.*



Generalizable Element, Classifier, Class, Component?

Structured Class – Another Example



UNIVERSITY OF OSLO

Composite class (incomplete)

- with parts, ports and connectors

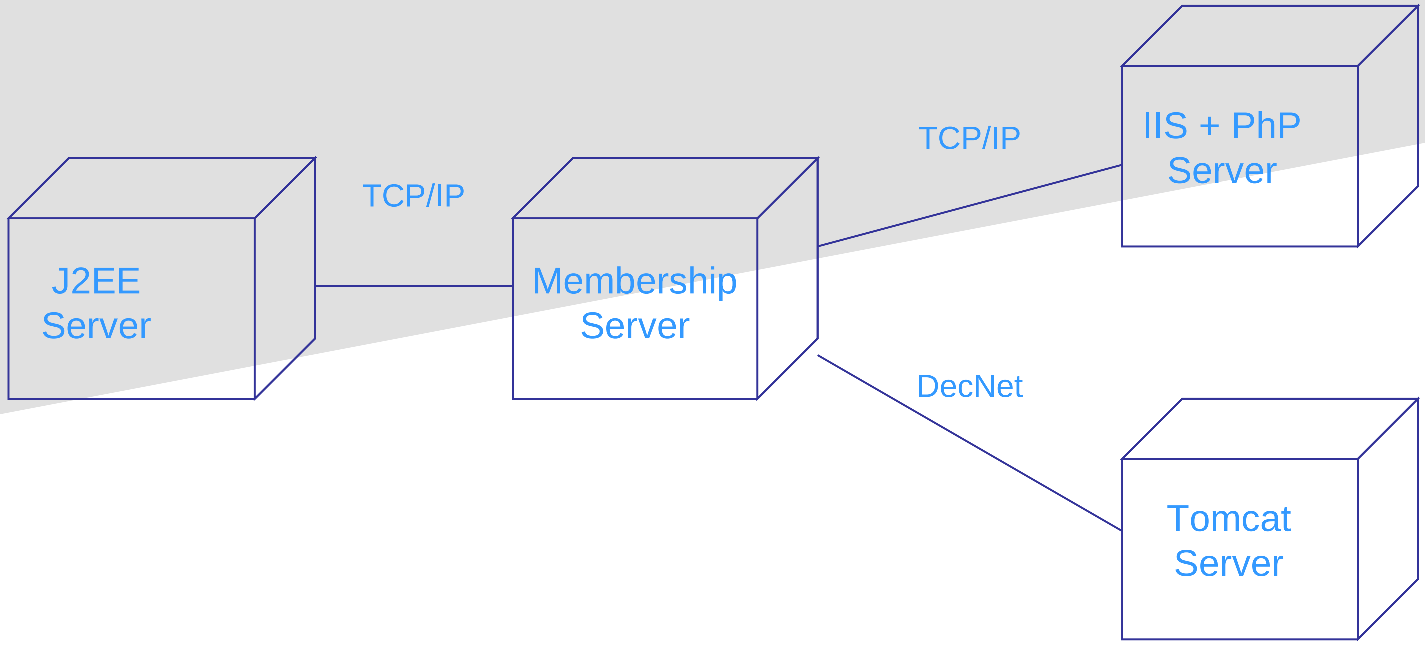
Deployment Diagrams

Structural Diagrams

- Class;
  Object
- Component
- Deployment
- **Composite Structure**
- Package

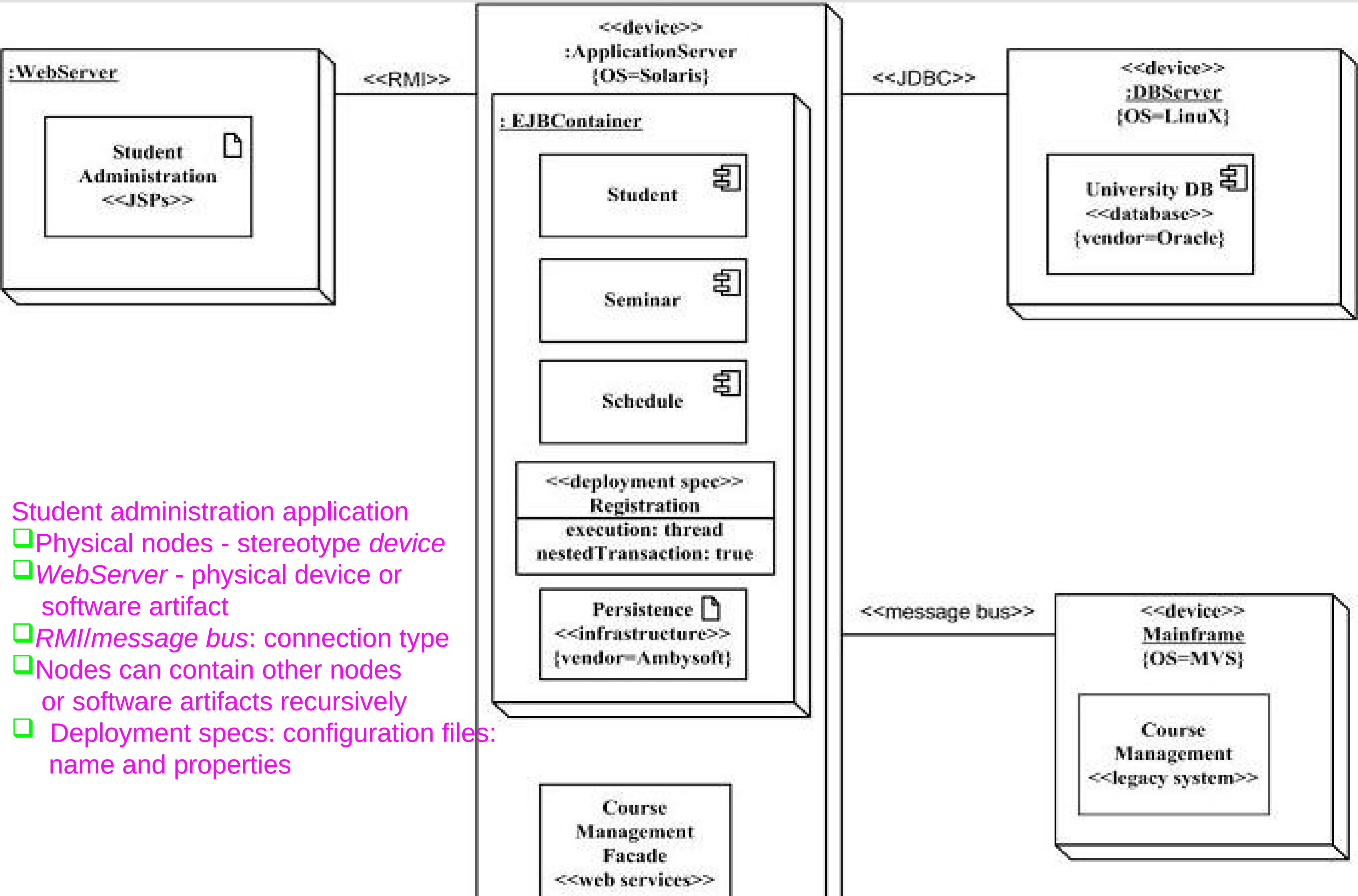
Deployment Diagram

- Shows a set of *processing* nodes and their relationships.
- Represents the static deployment view of an *architecture*.
- Nodes typically enclose one or more components.



Structural Diagrams - Deployment Diagram

(http://www.agilemodeling.com/artifacts/deploymentDiagram.htm)

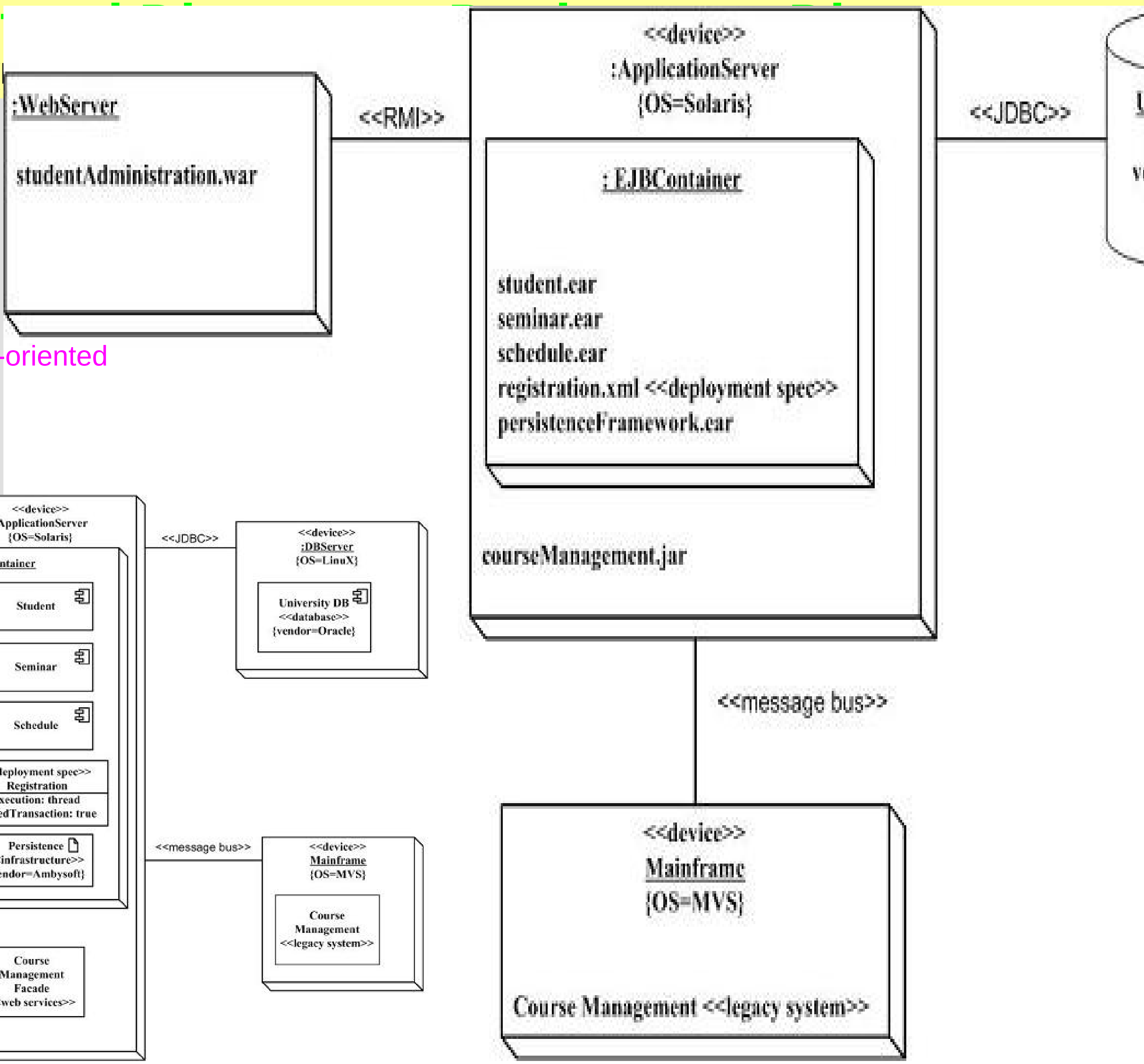


Student administration application

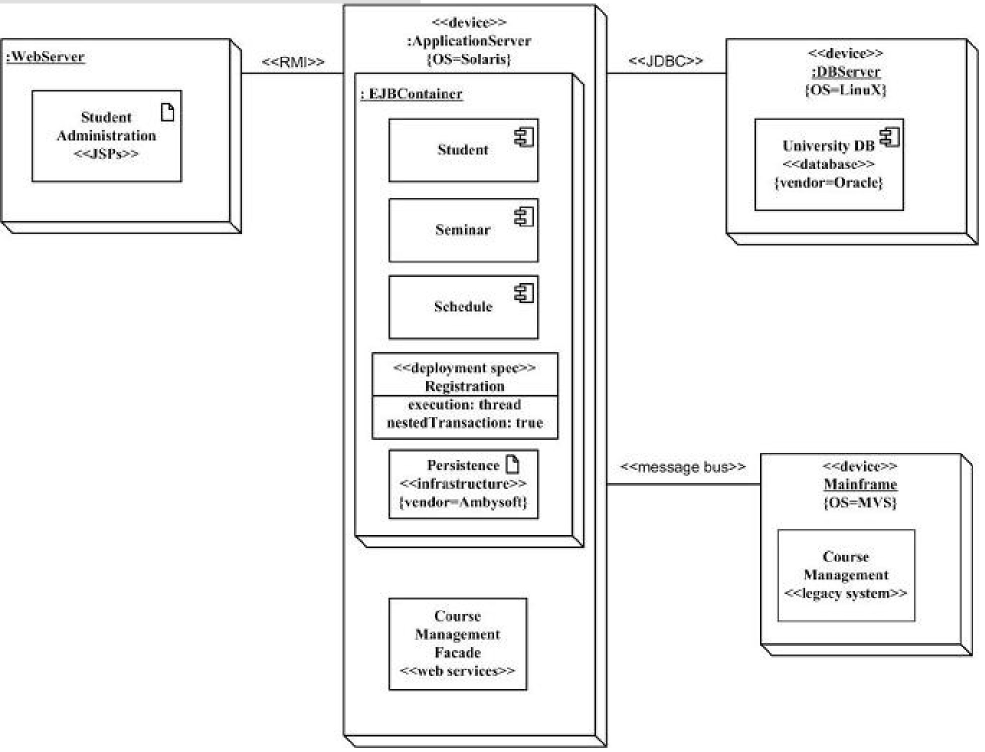
- Physical nodes - stereotype *device*
- *WebServer* - physical device or software artifact
- *RMI/message bus*: connection type
- Nodes can contain other nodes or software artifacts recursively
- Deployment specs: configuration files: name and properties

Struc

(



Is this better?
- ❑More concrete
- ❑Implementation-oriented



Composite Structure Diagrams

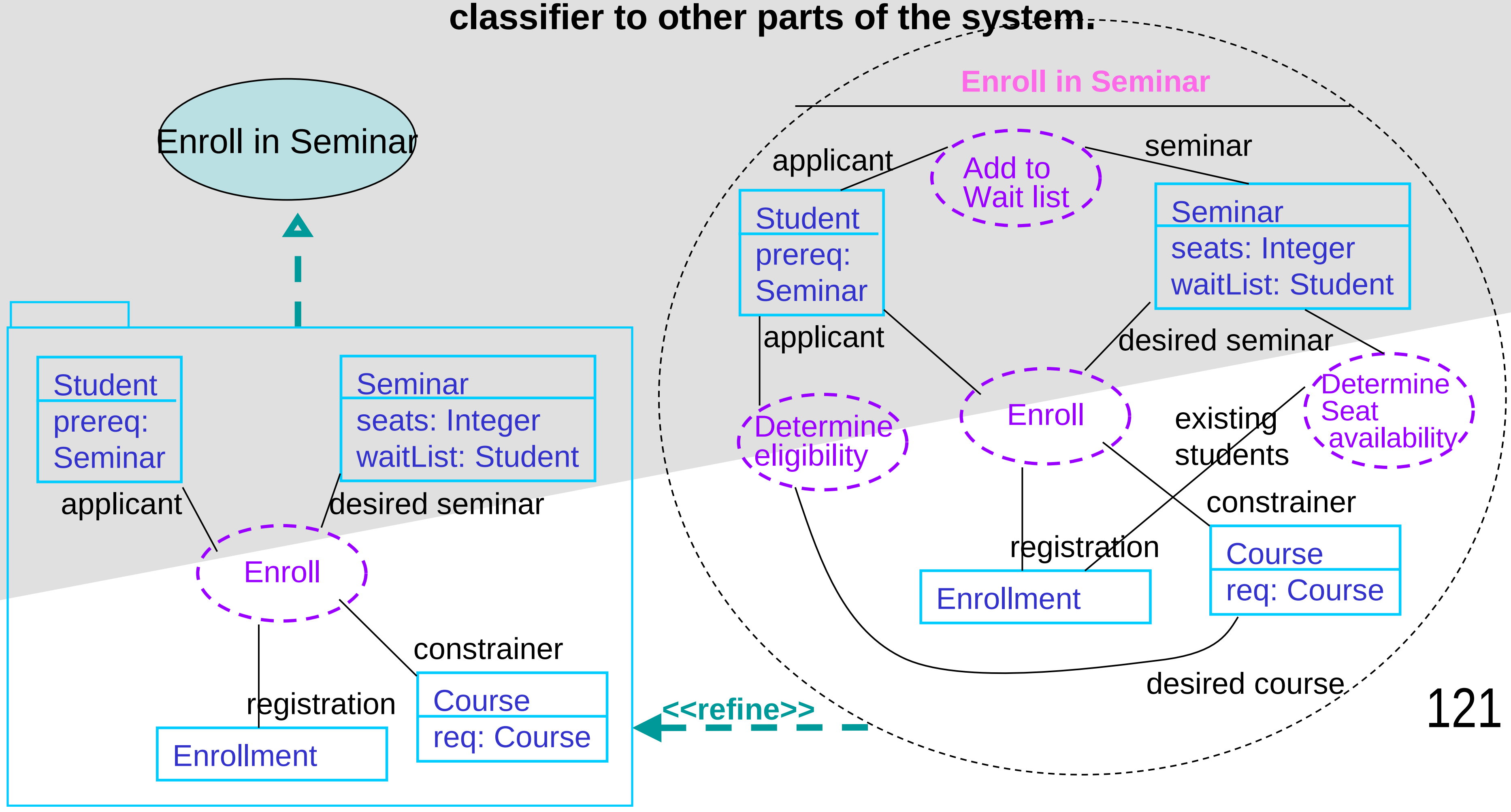
Structural Diagrams

- Class;
  Object
- Component
- Deployment
- **Composite Structure**
- Package

Composite Structure Diagrams

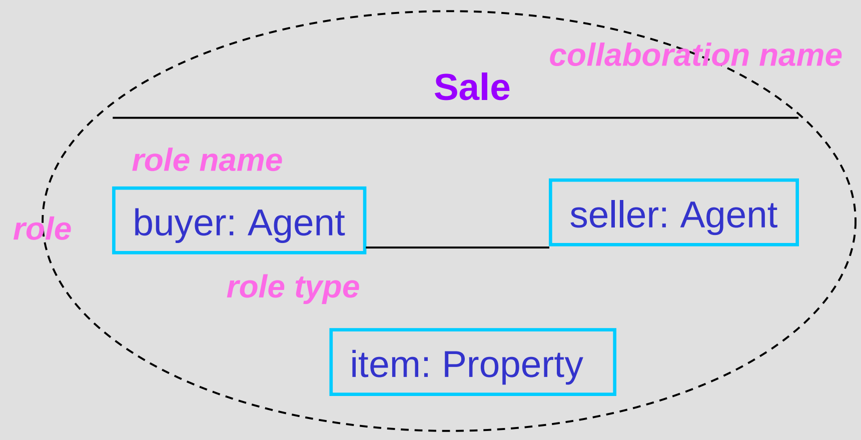
(http://www.agilemodeling.com/artifacts/compositeStructureDiagram.htm)

- Depicts the internal structure of a classifier (such as a class, component, or collaboration), including the interaction points of the classifier to other parts of the system.

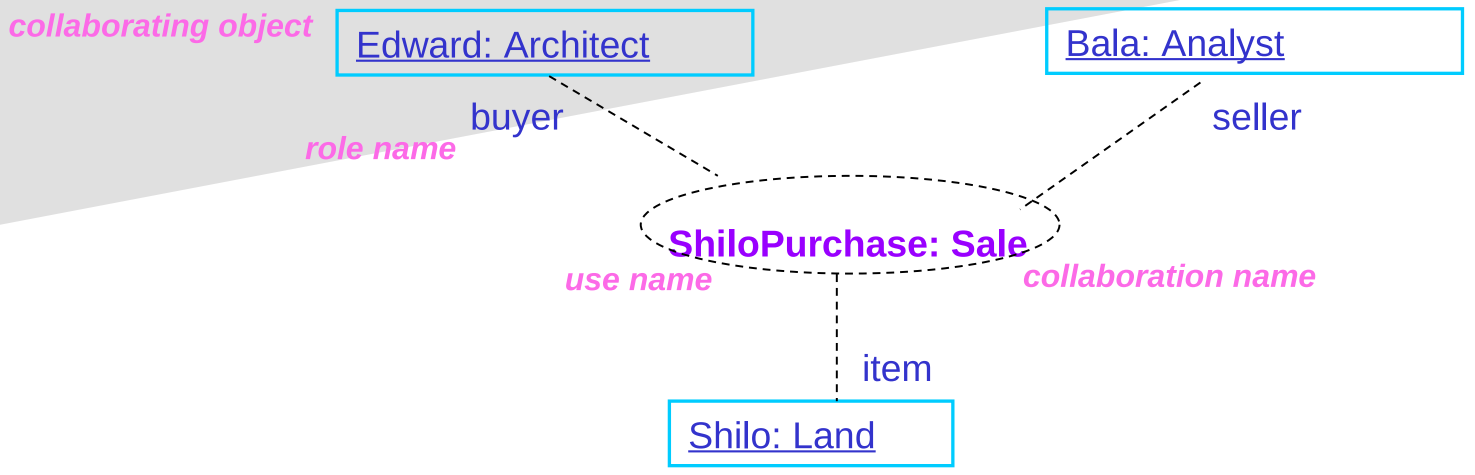


structured class, structured component, structured use case, structured node, structured interface, …

Variations [Rumbaugh – UML 2.0 Reference: p234]



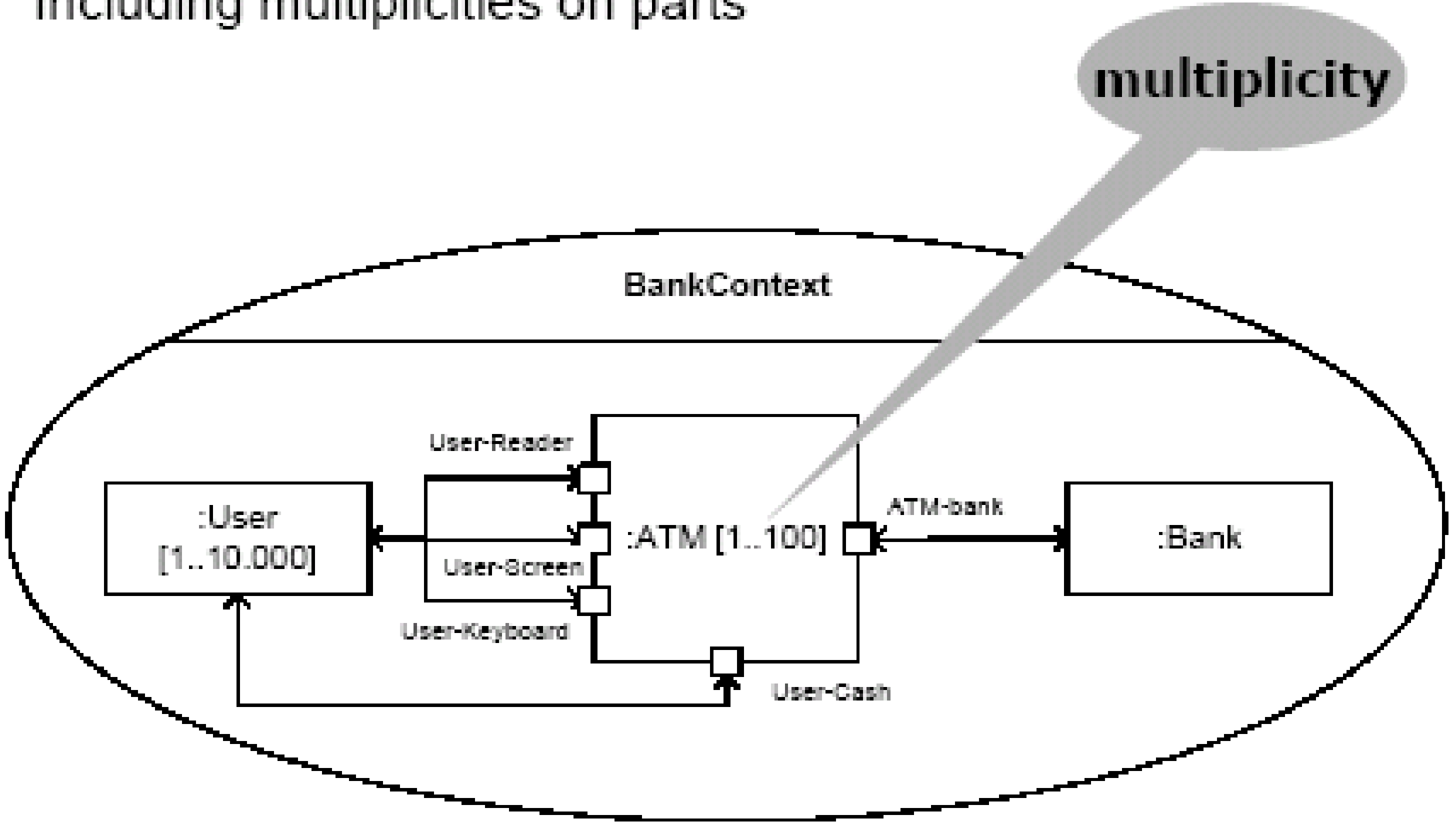
Collaboration definition



Collaboration use in object diagram

Context Model in UML2.0 - II

- Including multiplicities on parts

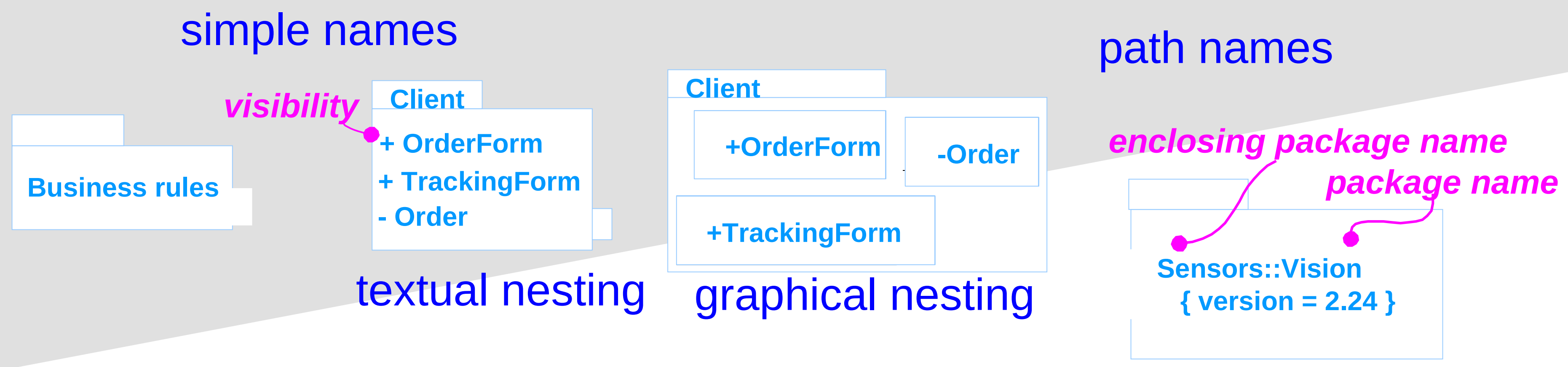


Structural Diagrams

- Class;
  Object
- Component
- Deployment
- Composite Structure
- **Package**

Packages

- Package — general-purpose mechanism for organizing elements into groups.
- Nested Elements: Composite relationship (When the whole dies, its parts die as well, but not necessarily vice versa)
- (C++ namespace; specialization means "derived")

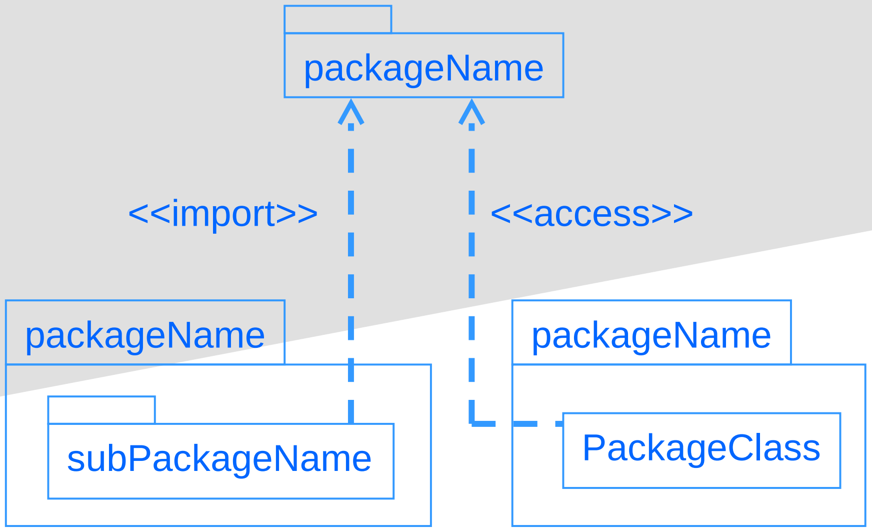


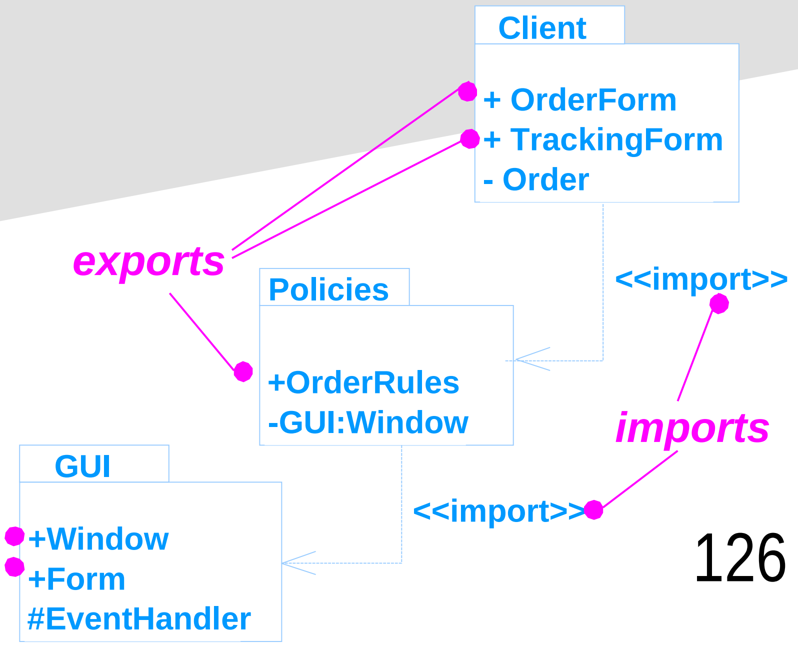
Visibility

- Packages that are friends to another may see all the elements of that package, no matter what their visibility.
- If an element is visible within a package, it is visible within all packages nested inside the package.

Dependency –Among Packages

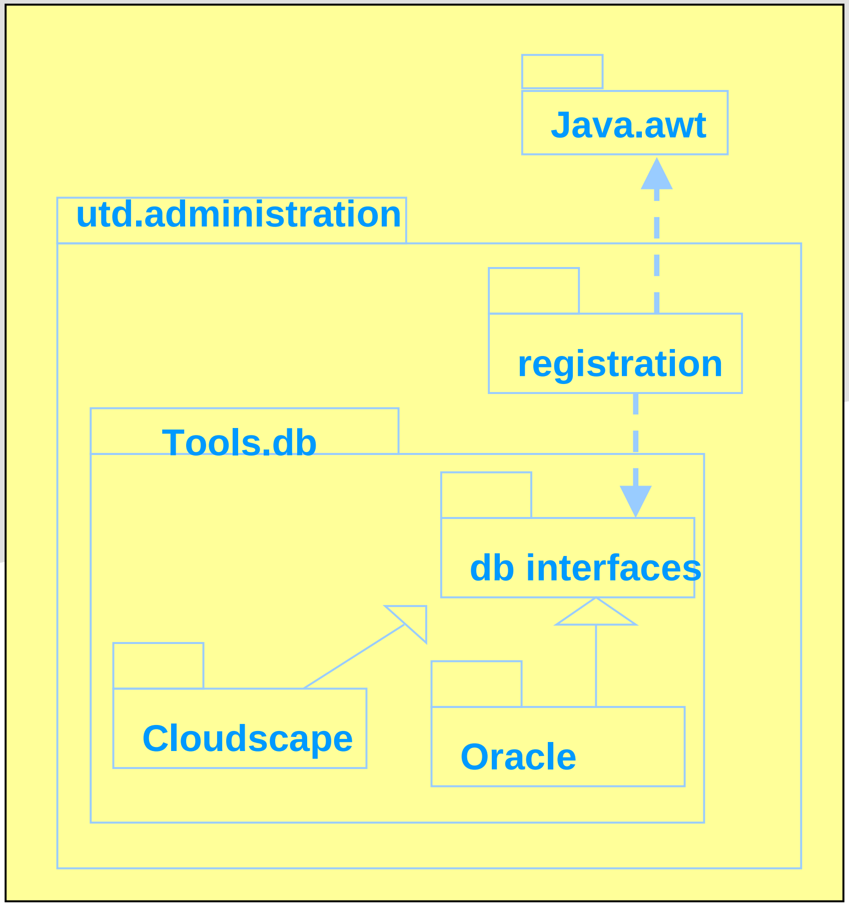
- Two Stereotypes of Dependency Among Packages:
  - *access*: the source package is granted the right to reference the elements of the target package *(:: convention)*
  - *import*: a kind of access; the ***public*** contents of the target package enter the flat namespace of the source as if they had been declared in the source





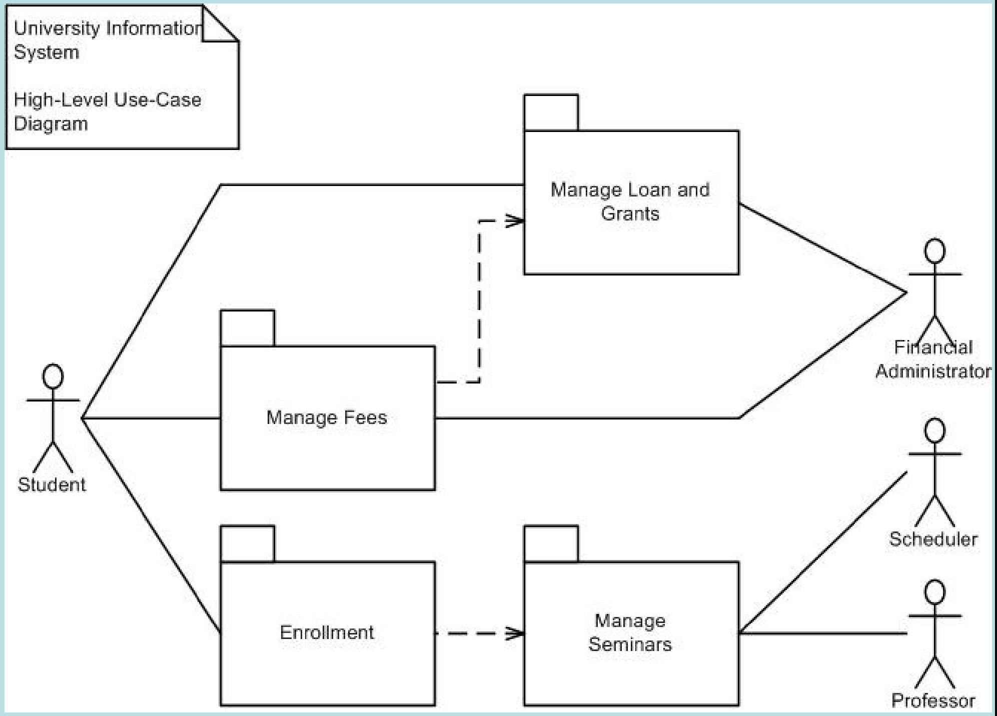
Modeling Groups of Elements

- Look for “clumps” of elements that are semantically close to one another.
- Surround “clumps” with a package.
- Identify public elements of each package.
- Identify import dependencies.



Use Case package Diagram

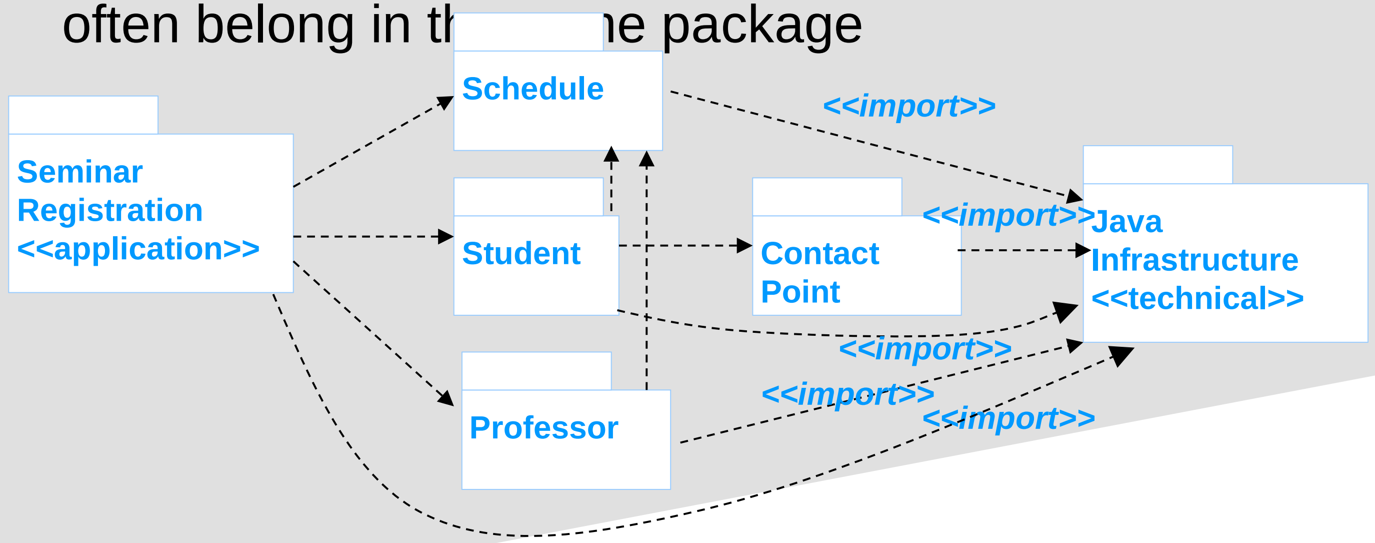
- Included and extending use cases belong in the same package as the parent/base use case
- Cohesive, and goal-oriented packaging
- Actors could be inside or outside each package



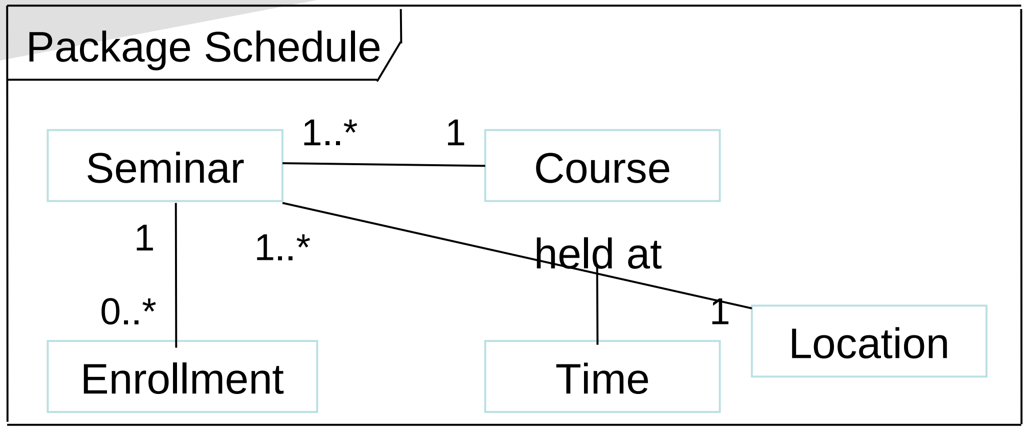
Class Package Diagrams

(http://www.agilemodeling.com/artifacts/packageDiagram.htm)

- Classes related through inheritance, composition or communication often belong in the same package



- A ***frame*** depicts the contents of a package (or components, classes, operations, etc.)
- Heading: rectangle with a cut-off bottom-right corner, [kind] name [parameter]



A frame encapsulates
a collection of collaborating instances or
refers to another representation of such

Common Mechanisms

- Adornments

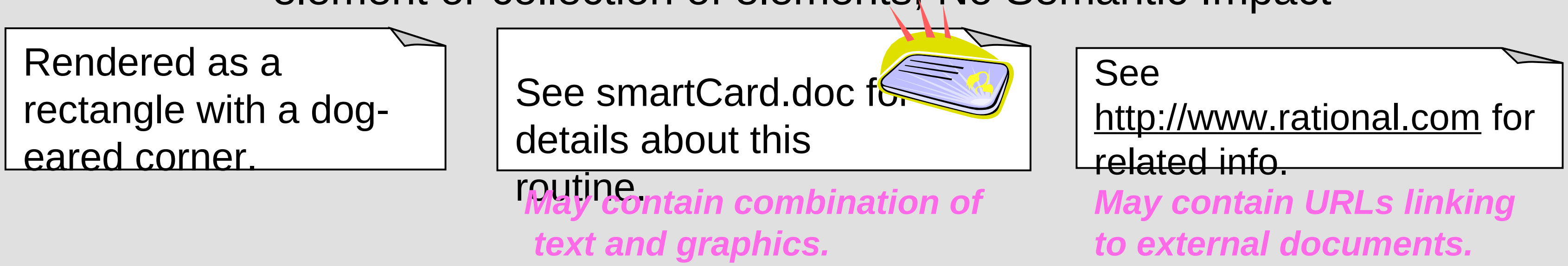
  Notes & Compartments

- Extensibility Mechanisms
  - Stereotypes - Extension of the UML metaclasses.
  - Tagged Values - Extension of the properties of a UML element.
  - Constraints - Extension of the semantics of a UML element.

Adornments

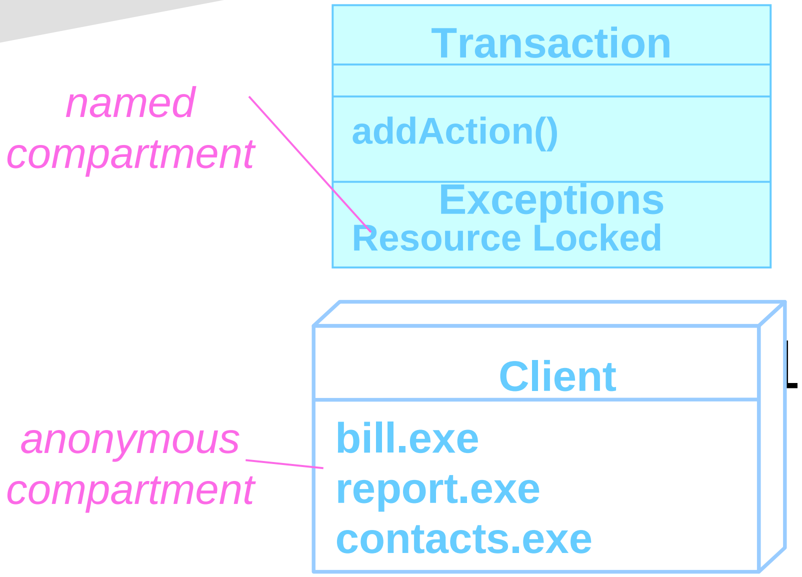
- Textual or graphical items added to an element’s basic notation.

- Notes - Graphical symbol for rendering constraints or comments attached to an element or collection of elements; No Semantic Impact


Rendered as a rectangle with a dog-eared corner.
See smartCard.doc for details about this routine.
May contain combination of text and graphics.
See http://www.rational.com for related info.
May contain URLs linking to external documents.

Additional Adornments

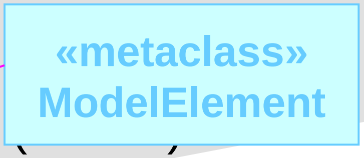
- Placed near the element as
  - Text
  - Graphic
- Special compartments for adornments in
  - Classes
  - Components
  - Nodes



Stereotypes

- Mechanisms for extending the UML vocabulary.
- Allows for new modeling building blocks or parts.

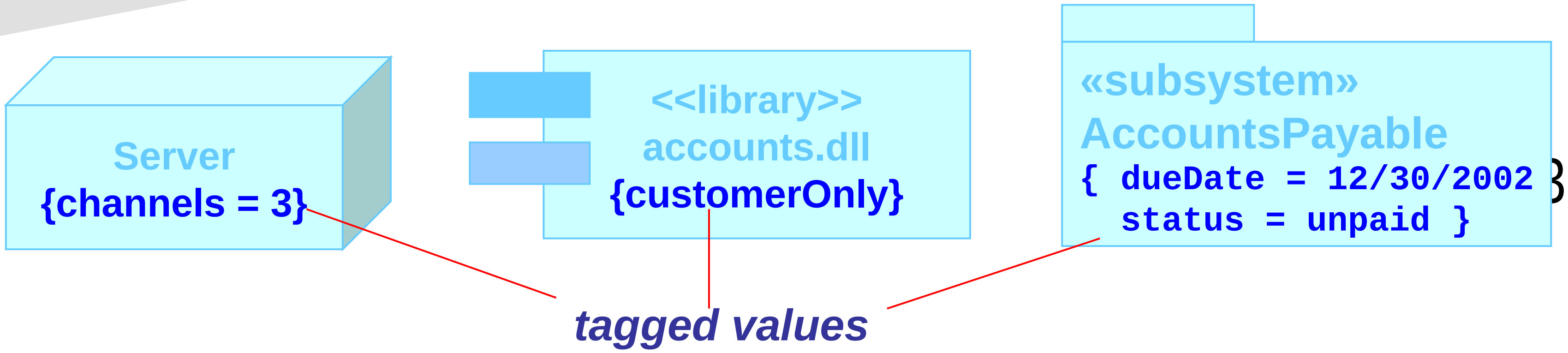
  - Allow controlled extension of metamodel classes. [UML11_Metamodel_Diagrams.pdf]
    - Graphically rendered
      - Name enclosed in guillemets (   )
        - <<stereotype>>
      - New icon



- The new building block can have
  - its own special properties through a set of tagged values
  - its own semantics through constraints

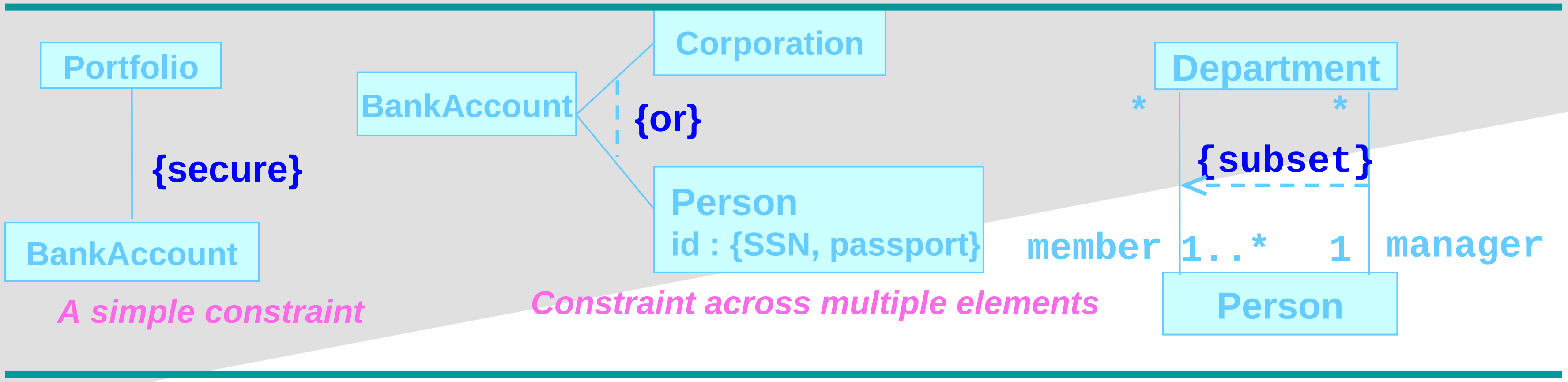
Tagged Values

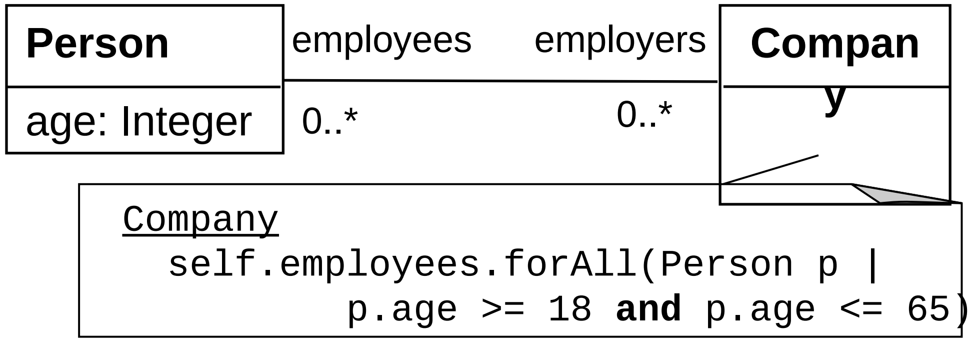
- a (name, value) pair describes a property of a model element.
- Properties allow the extension of "***metamodel***" element attributes.
- modifies the semantics of the element to which it relates.
- Rendered as a text string enclosed in braces { }
- Placed below the name of another element.



Constraints

- Extension of the semantics of a UML element.
- Allows new or modified rules
- Rendered in braces {}.
  - Informally as free-form text, or
  - Formally in UML's Object Constraint Language (OCL):
    E.g., {self.wife.gender = female and self.husband.gender = male}





```
Company
  self.employees.forAll(Person p |
          p.age >= 18 and p.age <= 65)
```

Appendix
Some Additional Material

Classes: Notation and Semantics

Class - Name
attribute-name-1 : data-type-1 = default-value-1 **attribute-name-2 : data-type-2 = default-value-2**
operation-name-1 (argument-list-1) : result-type-1 **operation-name-2 (argument-list-2) : result-type-2**
responsibilities

To model the <<semantics>> (meaning) of a class:

- Specify the body of each method (pre-/post-conditions and invariants)
- Specify the state machine for the class
- Specify the collaboration for the class
- Specify the responsibilities (contract)

Attributes

- Syntax

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

- Visibility

+ public; - private; # protected; {default = +}

- type
  - There are several defined in Rational Rose.
  - You can define your own.

Or you can define your own: e.g. {leaf}

- property-string

Built-in property-strings:

- *changeable*—no restrictions (default)
- *addOnly*—values may not be removed or altered, but may be added
- *frozen*—may not be changed after initialization

origin	Name only
+ origin	Visibility and name
origin : Point	Name and type
head : *Item	Name and complex type
name [0..1] : String	Name, multiplicity, and type
origin : Point = { 0, 0 }	Name, type, and initial value
id : Integer { frozen }	Name and property

Operations

- Syntax

[visibility] name [(parameter-list)] [: return-type] [(property-string)]

- Visibility

+ public; - private; # protected; {default = +}

- parameter-list syntax

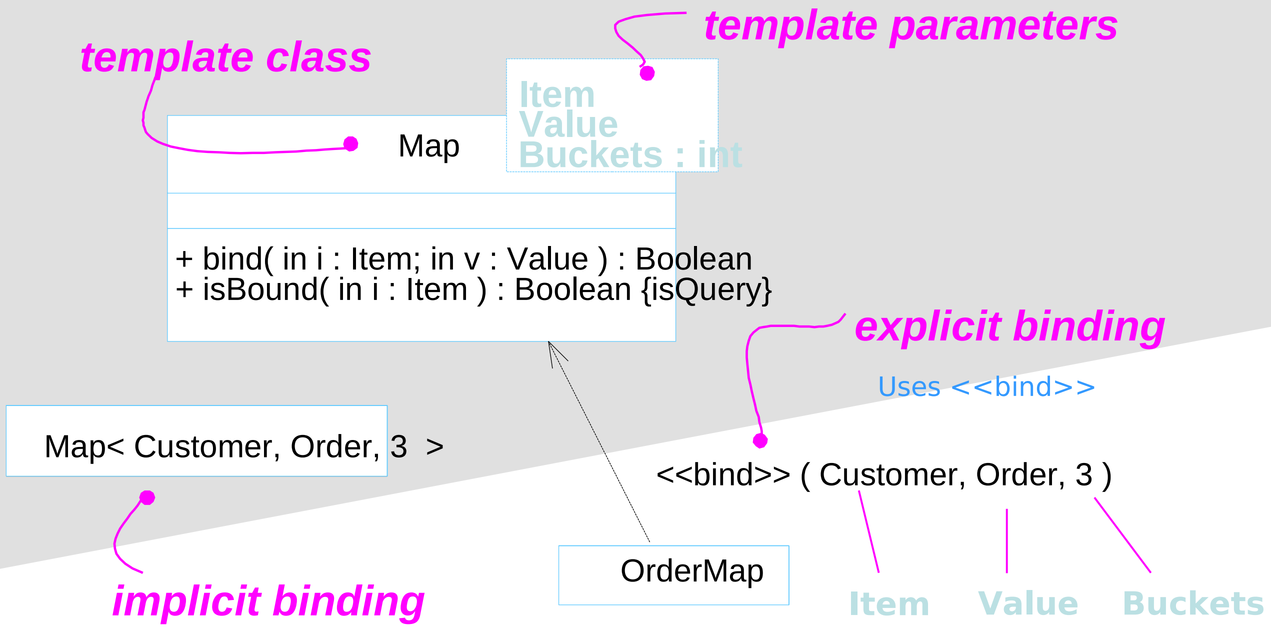
[direction] name : type [= default-value]

- direction
  - *in*—input parameter; may not be modified
  - *out*—output parameter; may be modified
  - *inout*—input parameter; may be modified

- property-string
  - *leaf*
  - *isQuery*—state is not affected
  - *sequential*—not thread safe
  - *guarded*—thread safe (Java synchronized)
  - *concurrent*—typically atomic; safe for multiple flows of control

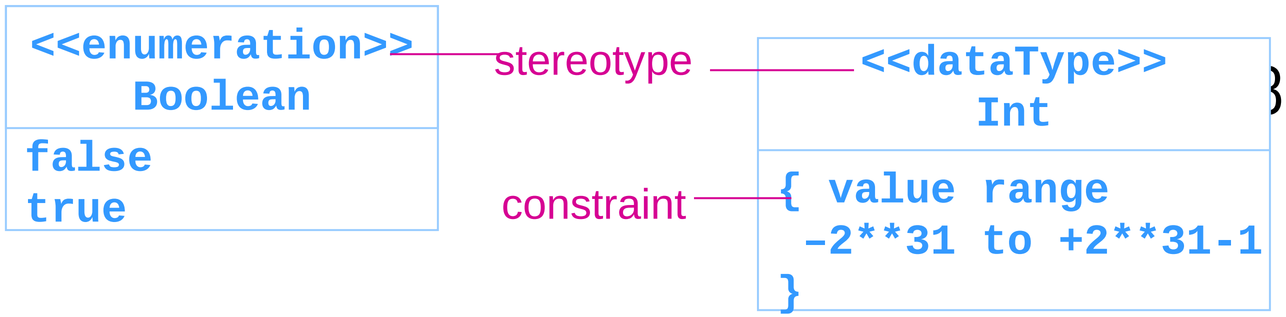
Template Classes; Primitive Types

- A template class is a parameterized element and defines a family of classes
- In order to use a template class, it has to be instantiated
- Instantiation involves binding formal template parameters to actual ones, resulting in a concrete class



template class
template parameters
Item
Value
Buckets : int
Map
+ bind(in i : Item; in v : Value) : Boolean
+ isBound(in i : Item) : Boolean {isQuery}
explicit binding
Uses <<bind>>
Map< Customer, Order, 3 >
<<bind>> (Customer, Order, 3)
OrderMap
implicit binding
Item
Value
Buckets

Primitive Types using a class notation



Interface: A Java Example

```
public interface SoundFromSpaceListener extends EventListener {
 void handleSoundFromSpace(SoundFromSpaceEventObject sfseo);
}

public class SpaceObservatory implements SoundFromSpaceListener
public void handleSoundFromSpace(SoundFromSpaceEventObject sfseo) {
        soundDetected = true;
        callForPressConference();
}
```

Can you draw a UML diagram corresponding to this?

Package Diagrams: Standard Elements

- *Façade* — only a view on some other package.
- *Framework* — package consisting mainly of patterns.
- *Stub* — a package that serves as a proxy for the public contents of another package.
- *Subsystem* — a package representing an independent part of the system being modeled.
- *System* — a package representing the entire system being modeled.

Is <<import>> transitive?
Is visibility transitive?
Does <<friend>> apply to all types of visibility: +, -, #?

Dependency –Among Objects

- 3 Stereotypes of Dependency in Interactions among Objects:
  - *become*: the target is the same object as the source but at a later point in time and with possibly different values, state, or roles
    - *call*: the source operation invokes the target operation
  - *copy*: the target object is an exact, but independent, copy of the source

Module 3: Advanced Features – Part II: Behavioral Diagrams

3 basic building blocks of UML - Diagrams

Graphical representation of a set of elements.
Represented by a connected graph: Vertices are things; Arcs are relationships/behaviors.
5 most common views built from

UML 1.x: 9 diagram types.

UML 2.0: 12 diagram types

UML 1.x

Structural Diagrams

Represent the static aspects of a system.

- Class;
  Object
- Component
- Deployment

Behavioral Diagrams

Represent the dynamic aspects.

- Use case
- Sequence;
  Collaboration
- Statechart
- Activity

UML 2.0

Structural Diagrams

- Class;
  Object
- Component
- Deployment
- Composite Structure
- Package

Behavioral Diagrams

- Use case
- Statechart
- Activity

Interaction Diagrams

- Sequence;
  Communication
- Interaction Overview
- Timing

Use Case Diagrams

Behavioral Diagrams

- **Use case**
- Statechart
- Activity

Interaction Diagrams

- Sequence; *Communication*
- Interaction Overview
- Timing

Use Cases

- When to Use Use Cases
  - Fowler's View: do use cases first before object modeling
    - Capture the **simple, normal use-case** first
    - For every step ask "***What could go wrong***?" and how it might work out differently
    - Plot all variations as ***extension***s of the given use case
  - Another view: do object modeling first, then use cases
  - Another: iterate model - use case - model - use case ...

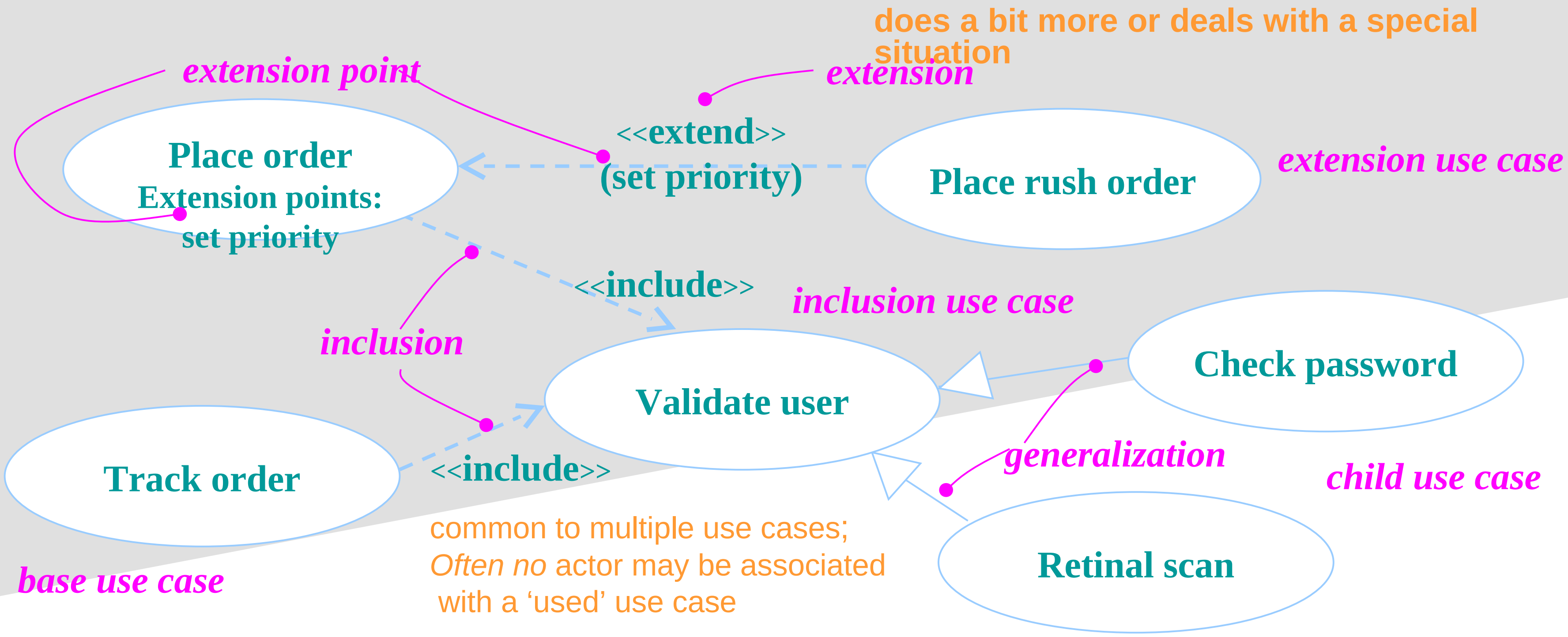
What did we do?

- Scenarios describe a single path, or a particular sequence
  - E.g., Use Case: Order Goods
    - Scenario 1: all goes well
    - Scenario 2: insufficient funds
    - Scenario 3: out of stock

- System test cases: Generate a test script for each scenario (flow of events).
  - Obtain initial state from preconditions.
  - Test success against post conditions.

Organizing Use Cases

- Generalization, Extend, Include/Use, packages



- **Track Order - Obtain and verify the order number; For each part in the order, query its status, then report back to the user.**
- **Place Order - Collect the user's order items. (set priority). Submit the order for processing.**

UML 1.3: Replaces <<uses>> relationship with Generalization and <<include>> dependency

A Use Case Template

(http://www.bredemeyer.com/pdf_files/use_case.pdf)

Use Case	Identifier: e.g., “Withdraw money”; ref # = wm3; mod history = …
Actors	List of actors involved in use case
Brief description	*Goal*: E.g., “This use case lets a bank account owner withdraw money from an ATM machine”; *Source*: Bank doc 2.3
Preconditions	What should be true before the use case can start.
Postconditions	What should hold after the use case successfully completes.
Basic flow of events	The happy/sunny day flow. The most common successful case.
Alt. flow of events /subflows	Difference for the specific subflow
Exception flows	Subflows may be divided into 1) normal, 2) successful alternate actions, and 3) exception/error flows.
Non-Functional (optional)	List of NFRs that the use case must meet
Issues	List of issues that remain to be resolved

A Use Case Template

(http://www.bredemeyer.com/pdf_files/use_case.pdf)

Use Case (id, ref#, mod history)	2. Reparing_Cellular_Network History created 1/5/98 Derek Coleman, modified 5/5/98
Description (goal, source)	Operator rectifies a report by changing parameters of a cell
Actors	Operator (primary, Cellular network, Field maintenance engineer)
Assumptions (successful use case termination condition)	Changes to network are always successful when applied to a network
Steps	1. Operator notified of network problem 2. Operator starts repair session 3. REPEAT 3.1 Operator runs network diagnosis application 3.2 Operator identifies cells to be changes and their new parameter values 3.3 IN PARALLEL 3.3.1 Maintenance engineer tests network cells || 3.3.2 Maintenance engineer sends fault reports UNTIL no more reports of problem 4. Operator closes repair session
Variations (optional)	#1. System may detect fault and notify operator or Field maintenance engineer may report fault to Operator
Non-Functional (optional)	Performance Mean: time to repair network fault must be less than 3 hours
Issues (that remain to be resolved)	What are the modes of communication between field maintenance engineer and operator

Use Case Extension	Repair_may_fail extends 2. Reparing_Cellular_Network
Description	Deals with assumption that network changes can never fail
Steps	#3.3. if the changes to network fail then the network is rolled back to its previous state
Issues	How are failures detected? Are roll backs automatic or is Operator intervention required?

Sequence Diagrams

Behavioral Diagrams

- Use case
- Statechart
- Activity

Interaction Diagrams

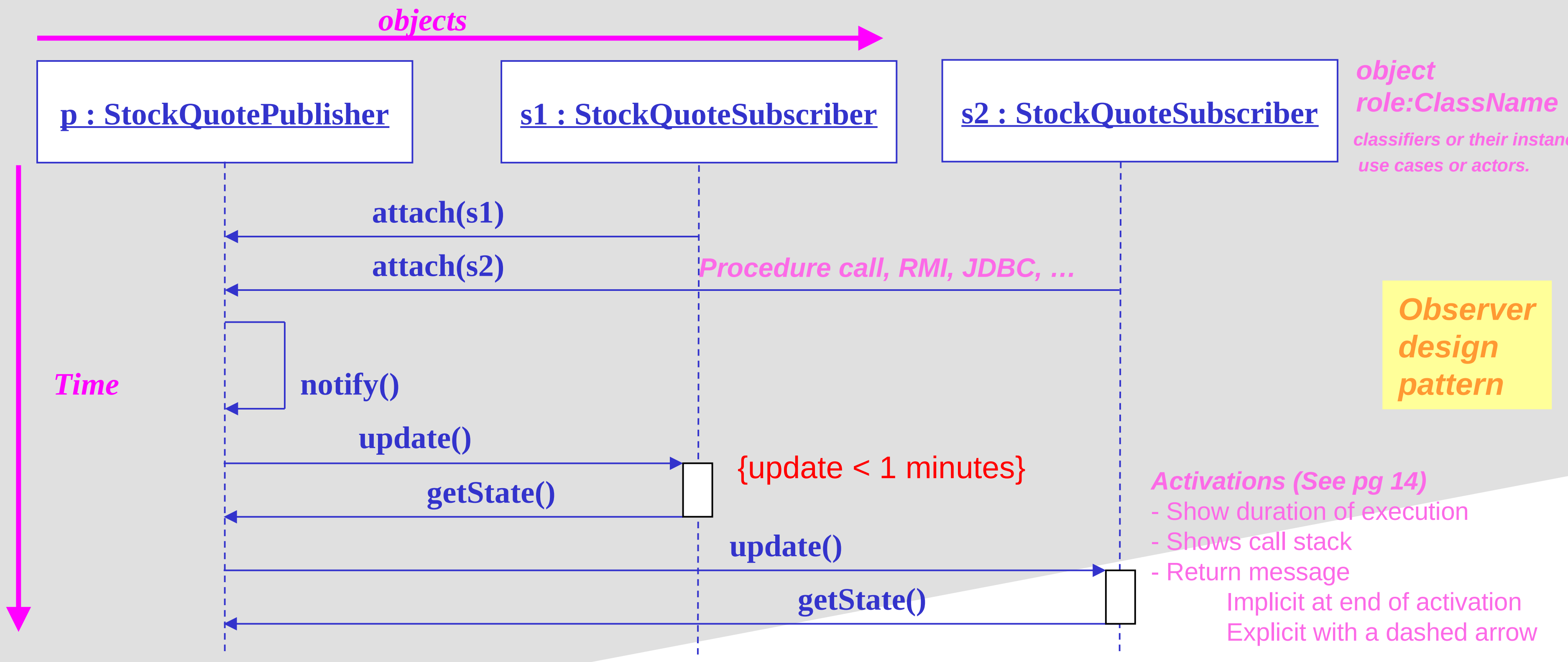
- **Sequence**;
  Communication
- Interaction Overview
- Timing

Interaction Diagrams (sd and cd)

- show the interaction of *any kind of instance (classes, interfaces, components, nodes and subsystems);*
- messages sent/received by those objects/instances (invocation, construction/destruction, of an operation)
- realizes use cases to model a scenario
- Interaction types (these are isomorphic, when no loops or branching)
  - Sequence diagram —emphasizes the time ordering of messages.
  - Communication (Collaboration) diagram — emphasizes the structural organization of objects that directly send and receive messages.

- Objects within an interaction can be:
  - Concrete: something from the real world. (e.g., **John: Person**)
  - Prototypical: representative instance of something from the real world (e.g., **p: Person**)
    - Communication diagrams use (strictly) prototypical things.
    - Prototypical instances of interfaces and abstract types are valid.

Interaction Diagram: sequence vs communication

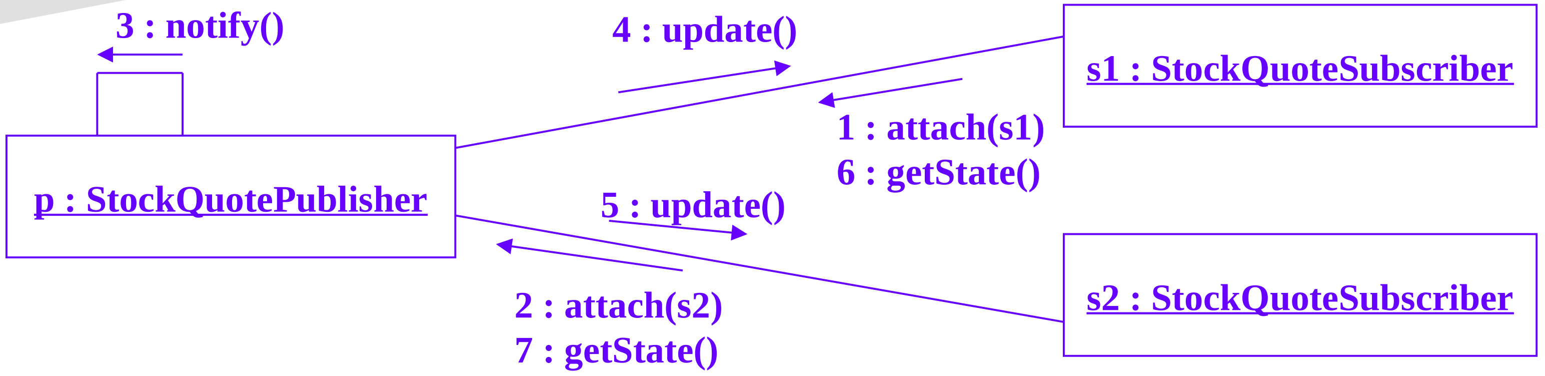


object role:ClassName

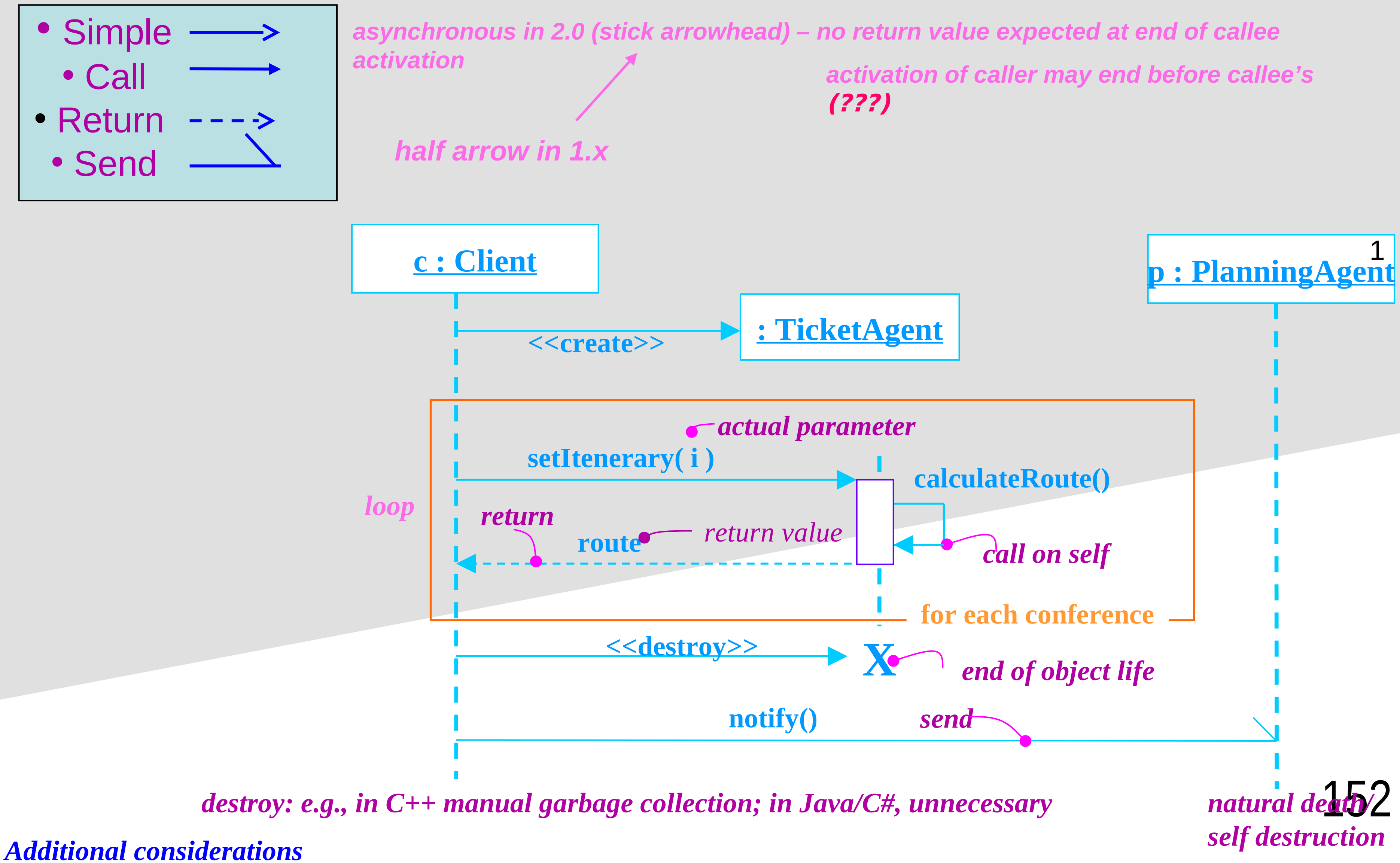
classifiers or their instances, use cases or actors.

Observer design pattern

Activations (See pg 14)
- Show duration of execution
- Shows call stack
- Return message
  - Implicit at end of activation
  - Explicit with a dashed arrow



Interactions - Modeling Actions

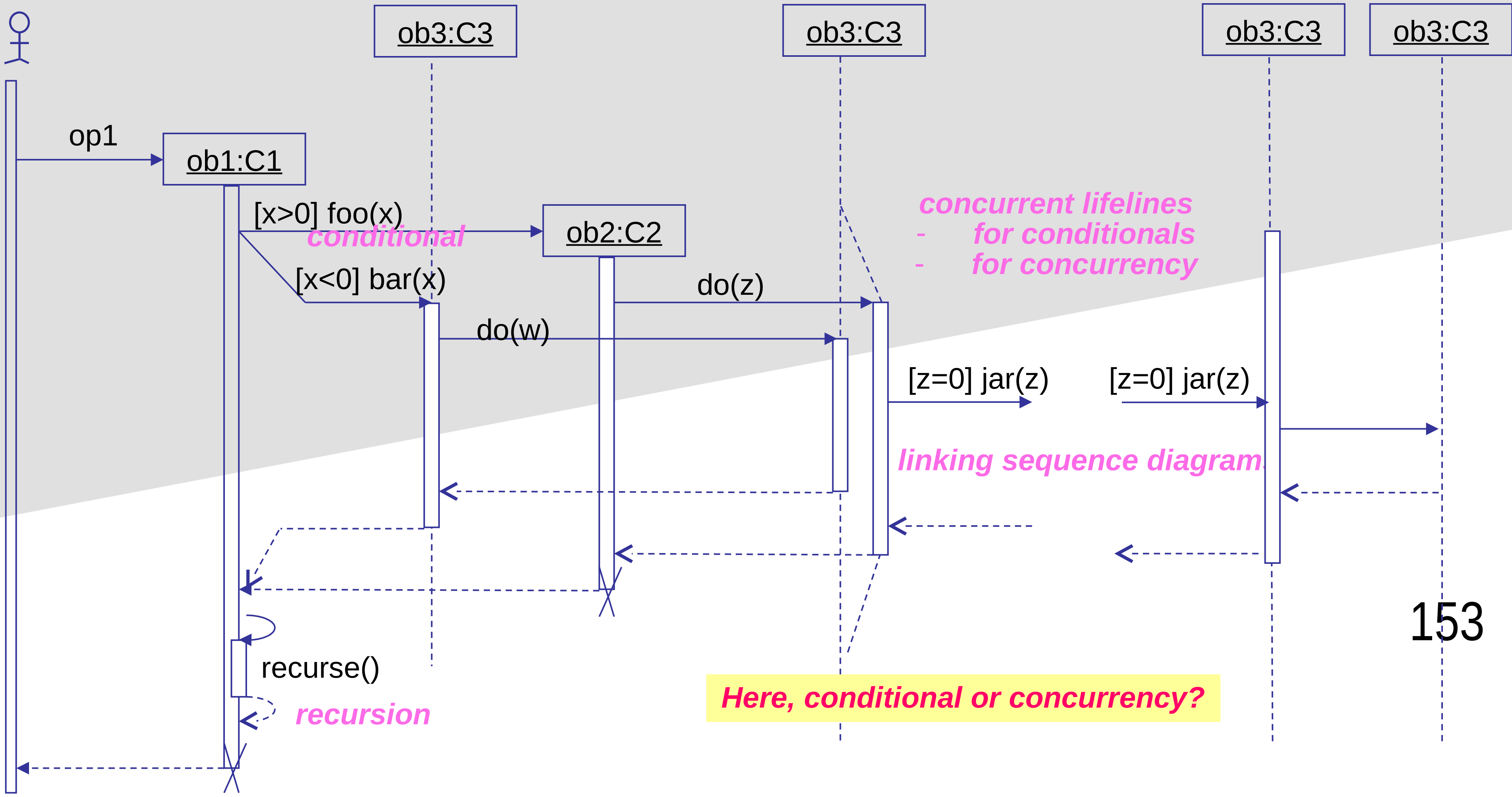


Additional considerations

- To show nested messages, use **?**.
- To show constraints like time and space, use **?**.
- For formal flow of control, attach pre and post conditions to each message (**?**)

Sequence Diagrams – Generic vs. Instance

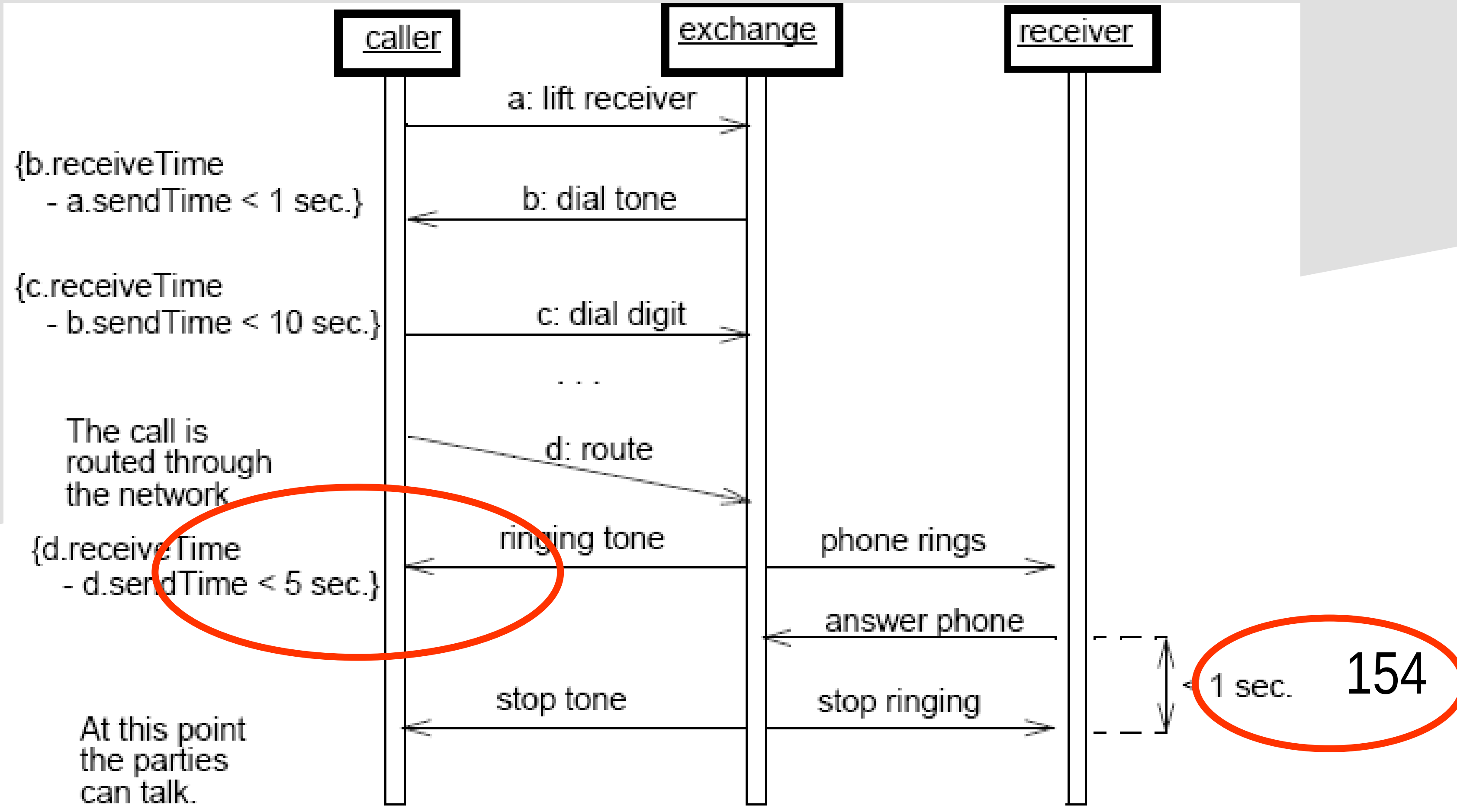
- 2 forms of sd:
  - **Instance** sd: describes a specific scenario in detail; no conditions, branches or loops.
  - **Generic** sd: a use case description with alternative courses.



Timing constraints

- Useful in real-time applications
- useful to specify race condition behaviour
- Two ways to specify (in 1.x)

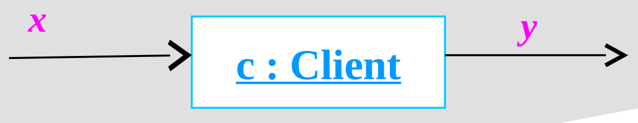
any example?

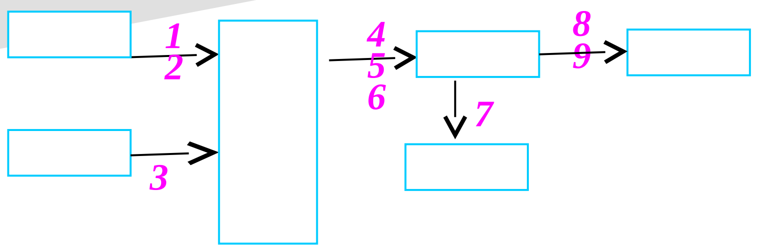

caller
exchange
receiver
a: lift receiver
{b.receiveTime - a.sendTime < 1 sec.}
b: dial tone
{c.receiveTime - b.sendTime < 10 sec.}
c: dial digit
. . .
The call is routed through the network
d: route
{d.receiveTime - d.sendTime < 5 sec.}
ringing tone
phone rings
answer phone
< 1 sec.
stop tone
stop ringing
At this point the parties can talk.

Interactions - Procedural Sequencing vs. Flat Sequencing

Flat Sequencing

- Infrequent: *Not recommended for most situations.*
- Each message is numbered sequentially in order of timing.
  - Rendered with stick arrowhead.



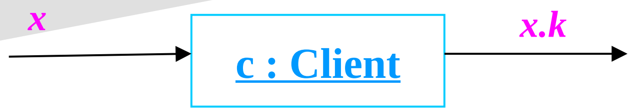


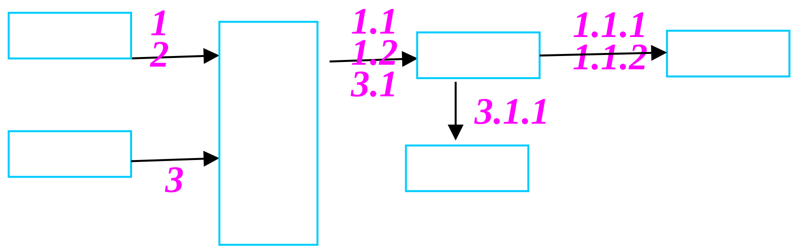
CAN'T TELL RELATIONSHIPS

Procedural Sequencing

(Dewey decimal system)

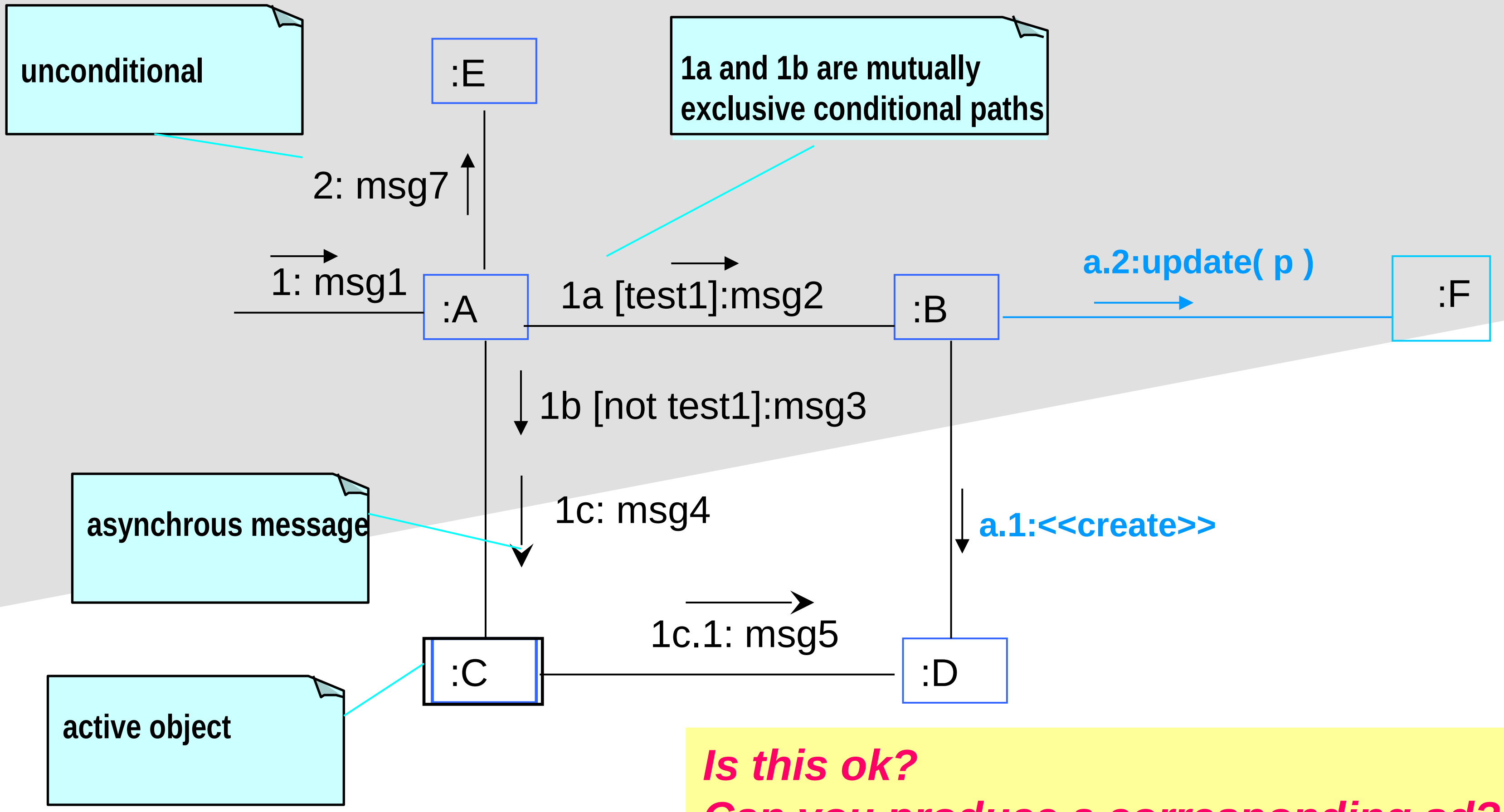
- Most common.
- Each message within the same operation is numbered sequentially.
- Nested messages are prefixed with the sequence number of the invoking operation.
- Rendered with filled solid arrow.





Interactions – conditional paths, asynchronous message

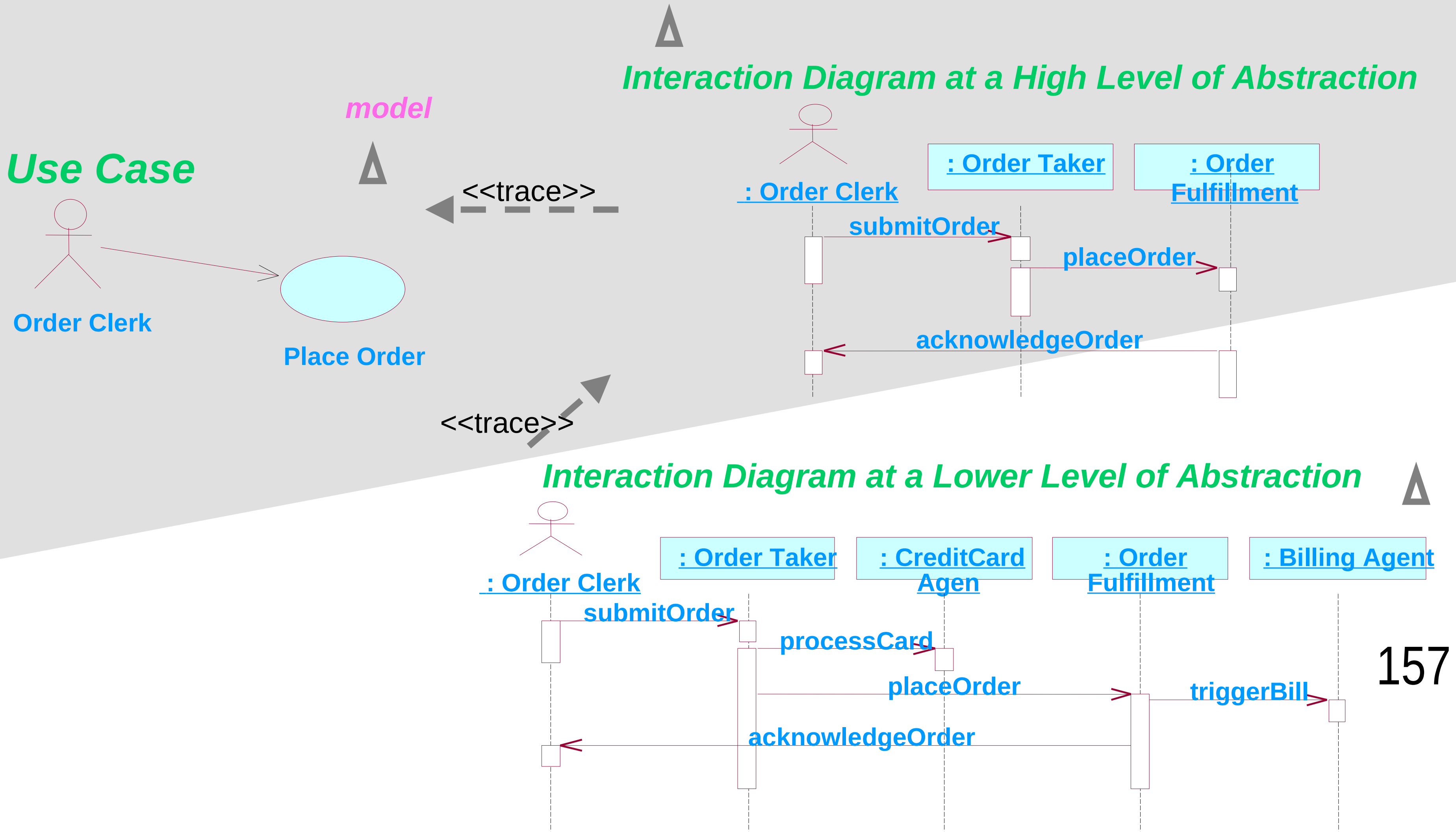
[Craig Larman] [http://www.phptr.com/articles/article.asp?p=360441&seqNum=6&rl=1]



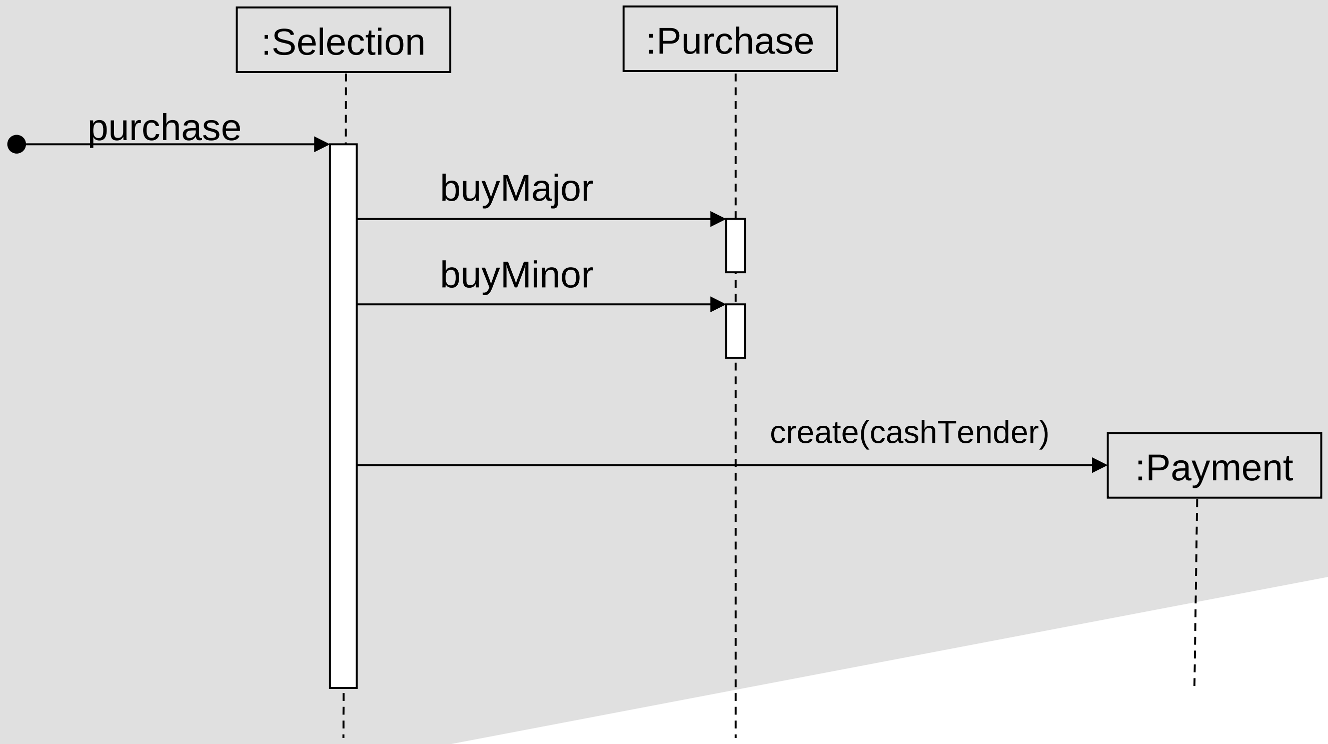
Is this ok?
Can you produce a corresponding sd?
Is there a unique sequence of paths?

Modeling Different Levels of Abstraction

- Establish trace dependencies between high and low levels of abstraction
- Lo*osely couple different levels of abstraction*
  - ***Use Cases*** trace to ***Collaborations*** in the ***Design*** Model, to a society of ***classes***
  - ***Components*** trace to the elements in the design model, then to ***Nodes***



Sequence Diagrams & Some Programming

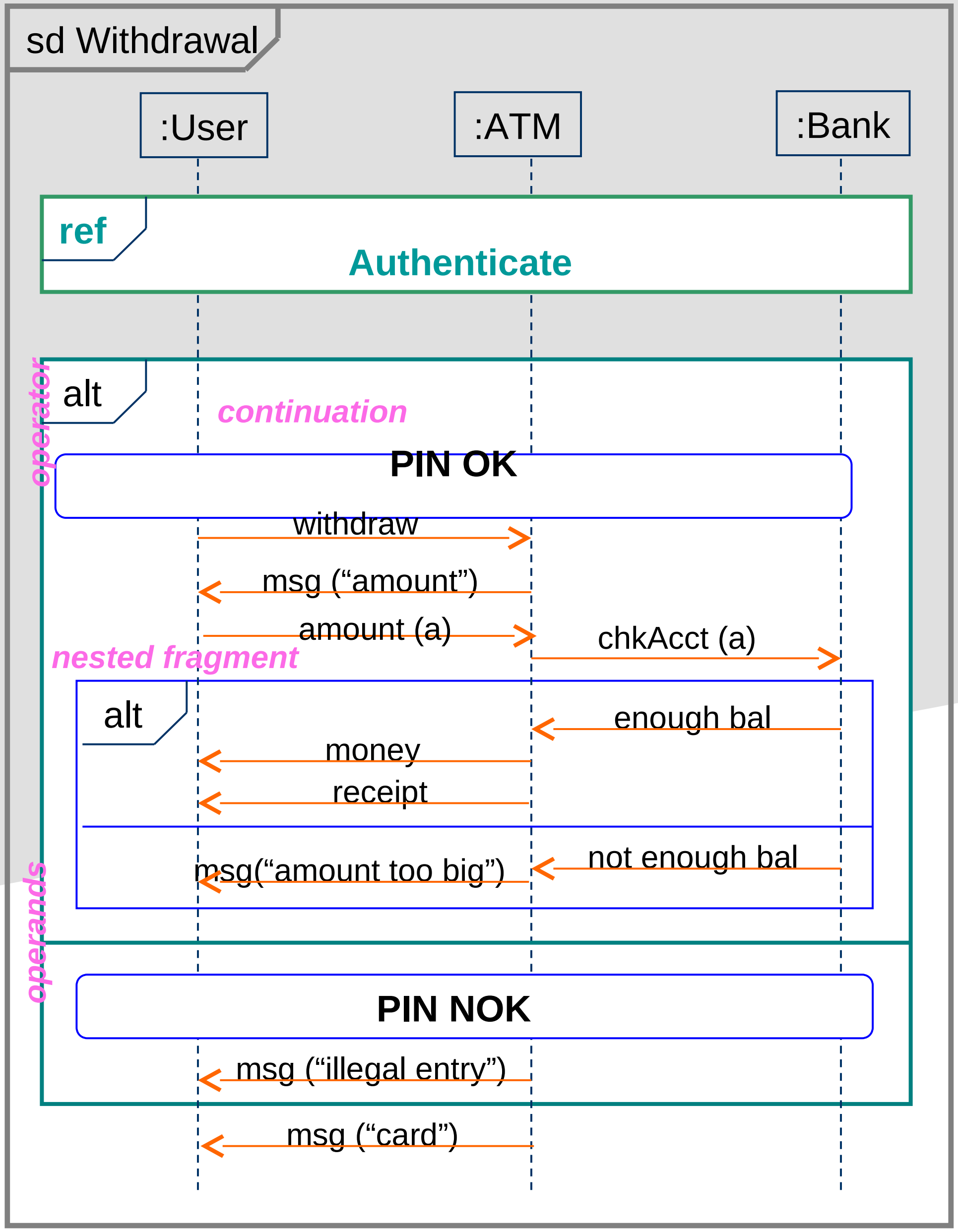


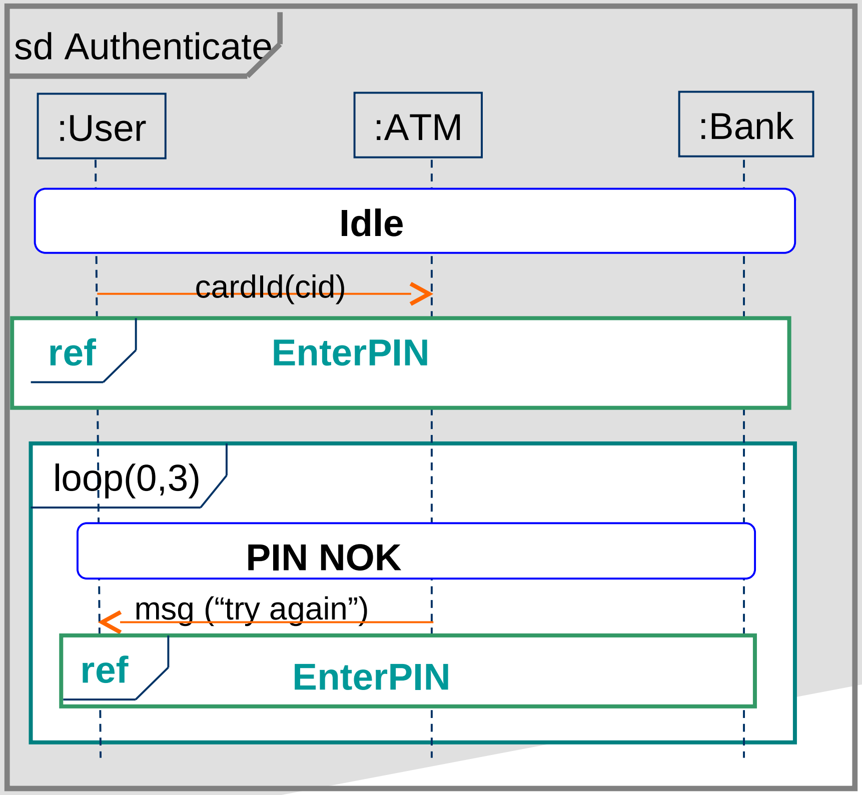
```
public Class Selection
        { private Purchase myPurchase = new Purchase();
          private Payment myPayment;
           public void purchase()
                 { myPurchase.buyMajor();
                   myPurchase.buyMinor():
                   myPayment = new Payment( cashTender );
                   //. .
                 }
          // . .
        }
```

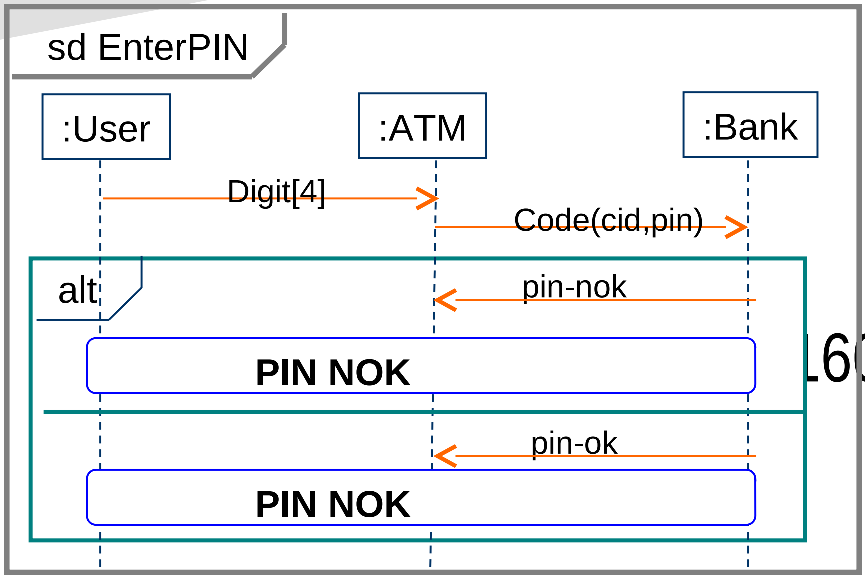
© Original Artist
Reproduction rights obtainable from
www.CartoonStock.com
CASH
TM
Silligan
Internet Conmen trick cash from customers.

- ❖ **Why do people call it an ATM machine, but they know it's really saying Automated Teller Machine Machine?**
- ➢ **Why do people say PIN number when that truly means Personal Identification Number Number?**

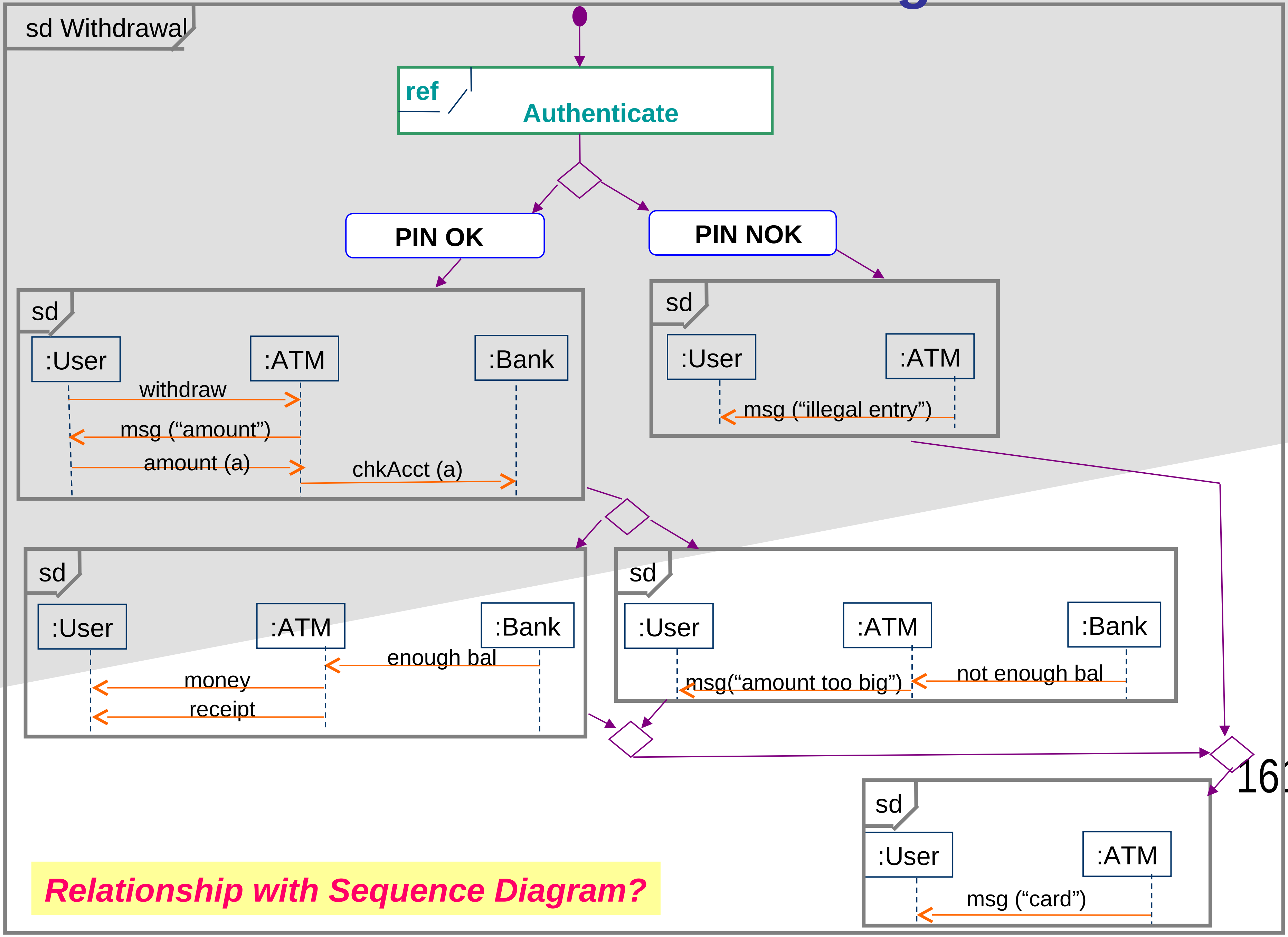
Frames: References







Interaction Overview Diagram



Relationship with Sequence Diagram?

Frames & Interaction Fragment Operators

Frame: as the graphical boundary, and a labeling mechanism

Frame name: “frame type name[(param type: param name)] [: return type: return name]”

Flow of Control

- sd: named sequence diagram ref: reference to “interaction fragment”

Naming

- loop: repeat interaction fragment
- opt: optional “exemplar” (cf. break)
- alt: selectior[illegible]
- par: concurrent (parallel) regions (e.g., mo[illegible] par[illegible])

[guard condition] can appear as the first item underneath

Ordering

- seq: partial ordering (default) (aka “weak”)
- strict: strict ordering
- criticalRegion: identifies “atomic” fragments

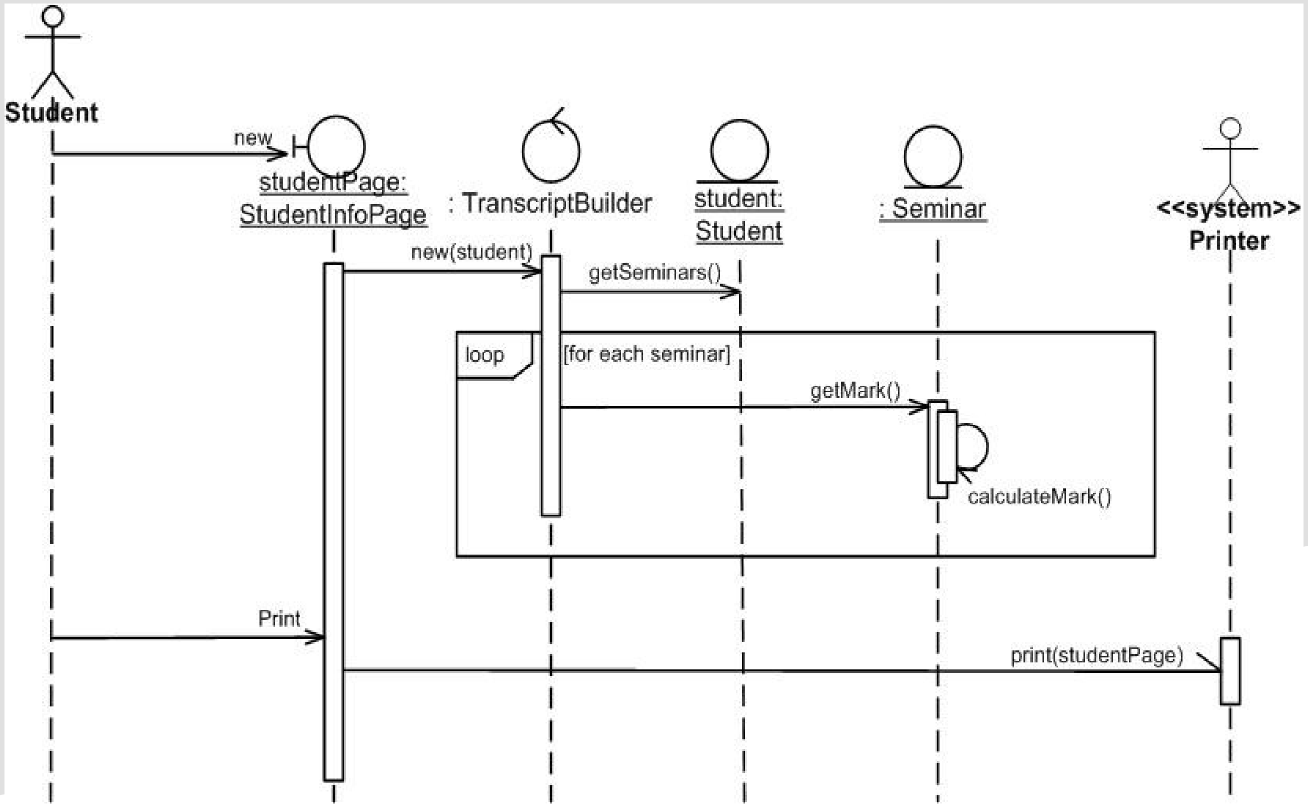
Causality

- assert: required (i.e. causal)
- neg: “can’t happen” or a negative specification
- Ignore/consider: messages outside/inside causal stream

What can be in the top boxes?

(http://www.agilemodeling.com/artifacts/sequenceDiagram.htm)

Outputting transcripts



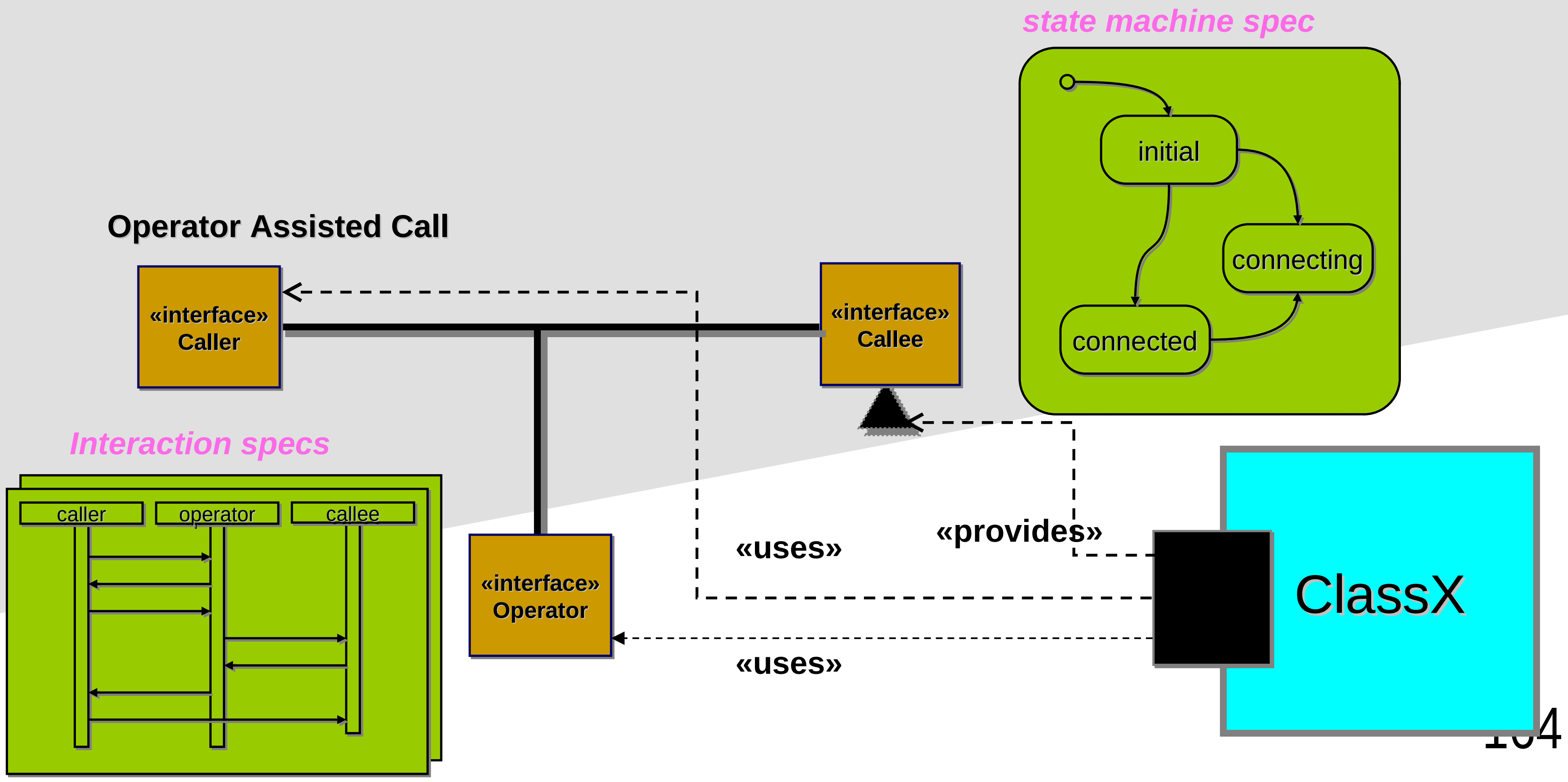
***Boundary/interface* elements**: software elements such as screens, reports, HTML pages, or system interfaces that actors interact with.

***Control/process* elements (*controllers*)**: These serve as the glue between boundary elements and entity elements, implementing the logic required to manage the various elements and their interactions. Often implemented using objects, but simple ones using methods of an entity or boundary class.

***Entity* elements**

Modeling Protocols - Associating Protocols with Ports

- By a set of ***interconnected interfaces,*** invoked according to a formal ***behavioral*** specification.

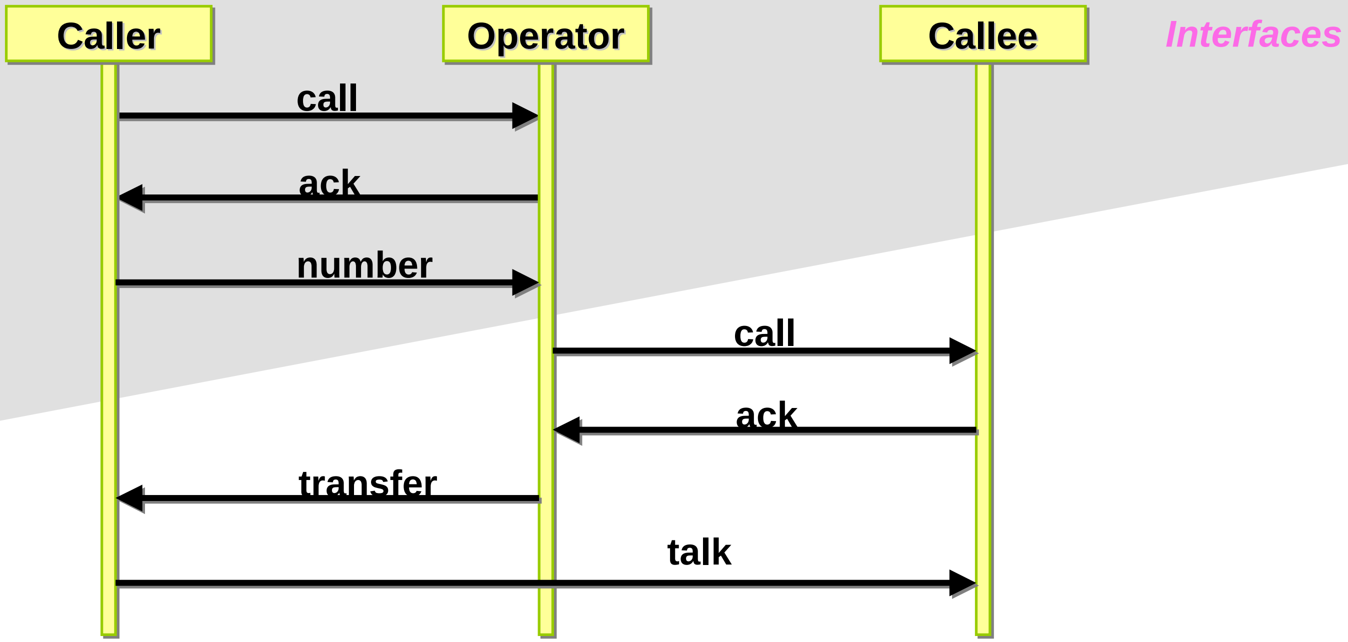


Can you depict this using balls & sockets?

Protocols: Reusable Interaction Sequences

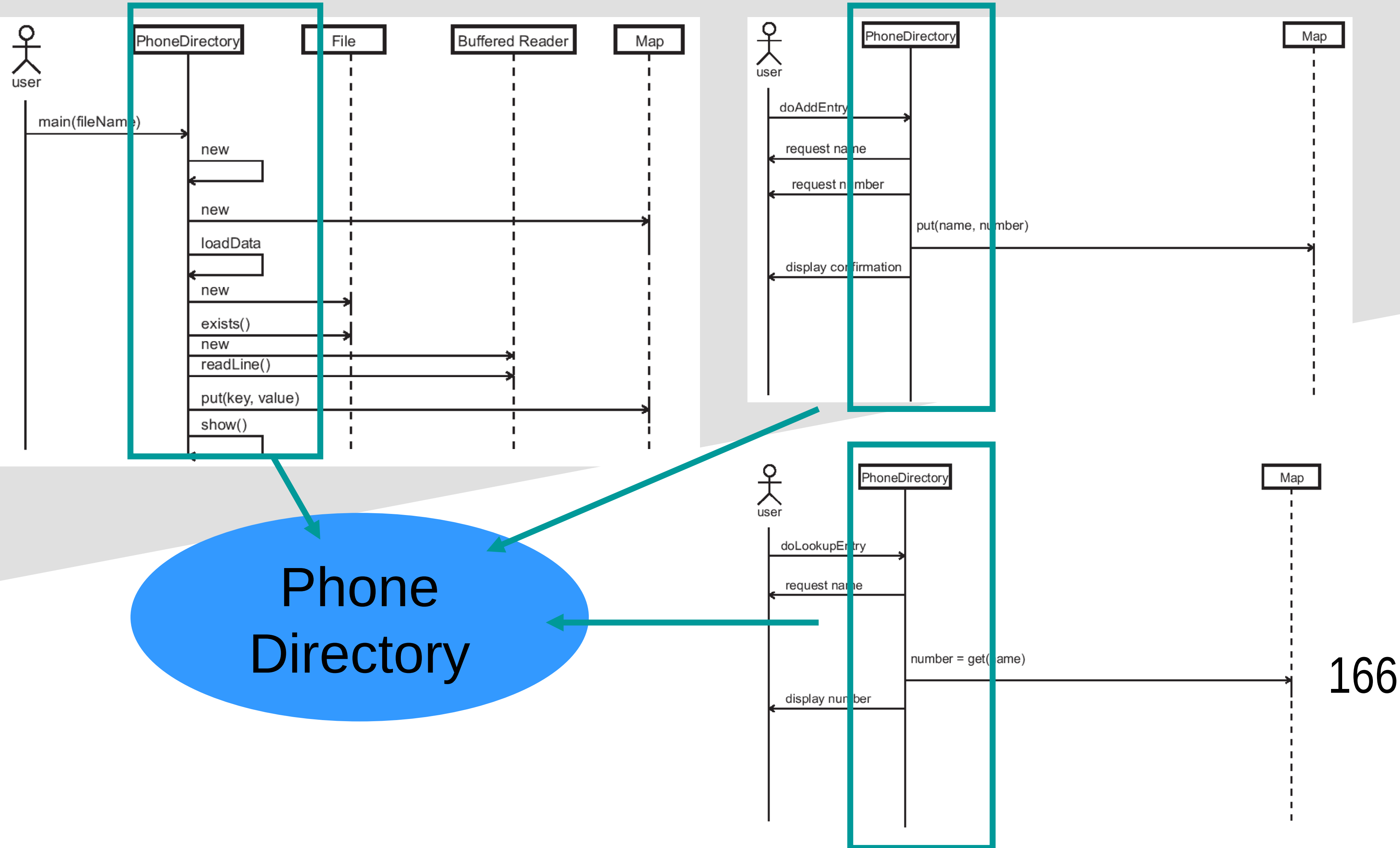
(http://cot.uni-mb.si/ots2003/ppt/Selic-UML2.0-tutorial.030504.pdf)

- Communication sequences that
- always follow a pre-defined dynamic order
- occur in different contexts with different specific participants (i.e., instances)



From Diagrams to Objects

Collect all messages to define object’s methods and state transitions !



State Transition Diagrams

Behavioral Diagrams

- Use case
- ***Statechart***
- Activity

Interaction Diagrams

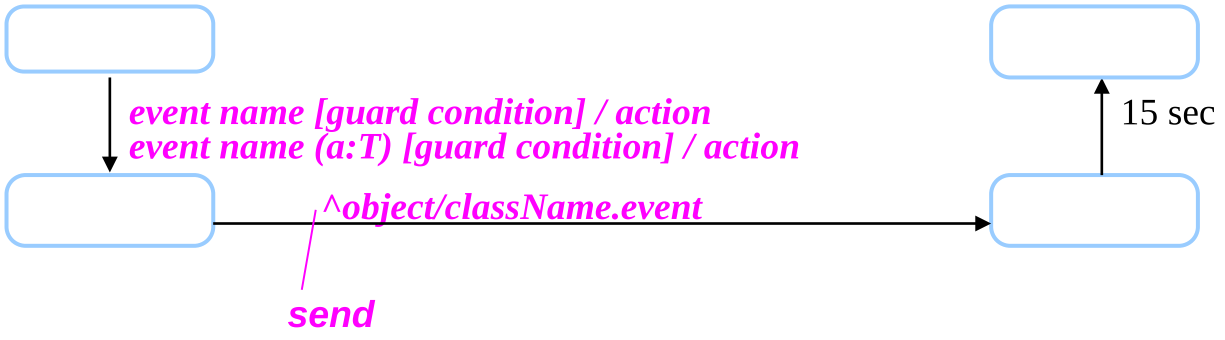
- Sequence; Communication
- Interaction Overview
- Timing

State Transitions

- ***State machine*** - event-ordered behavior that specifies the sequences of ***states*** an object/instance (of class/interface/collaboration/…/system) goes through during its lifetime; ***events*** trigger ***transitions*** and cause ***responses***.
  (StateChart is one particular kind of state machine by David Harel)

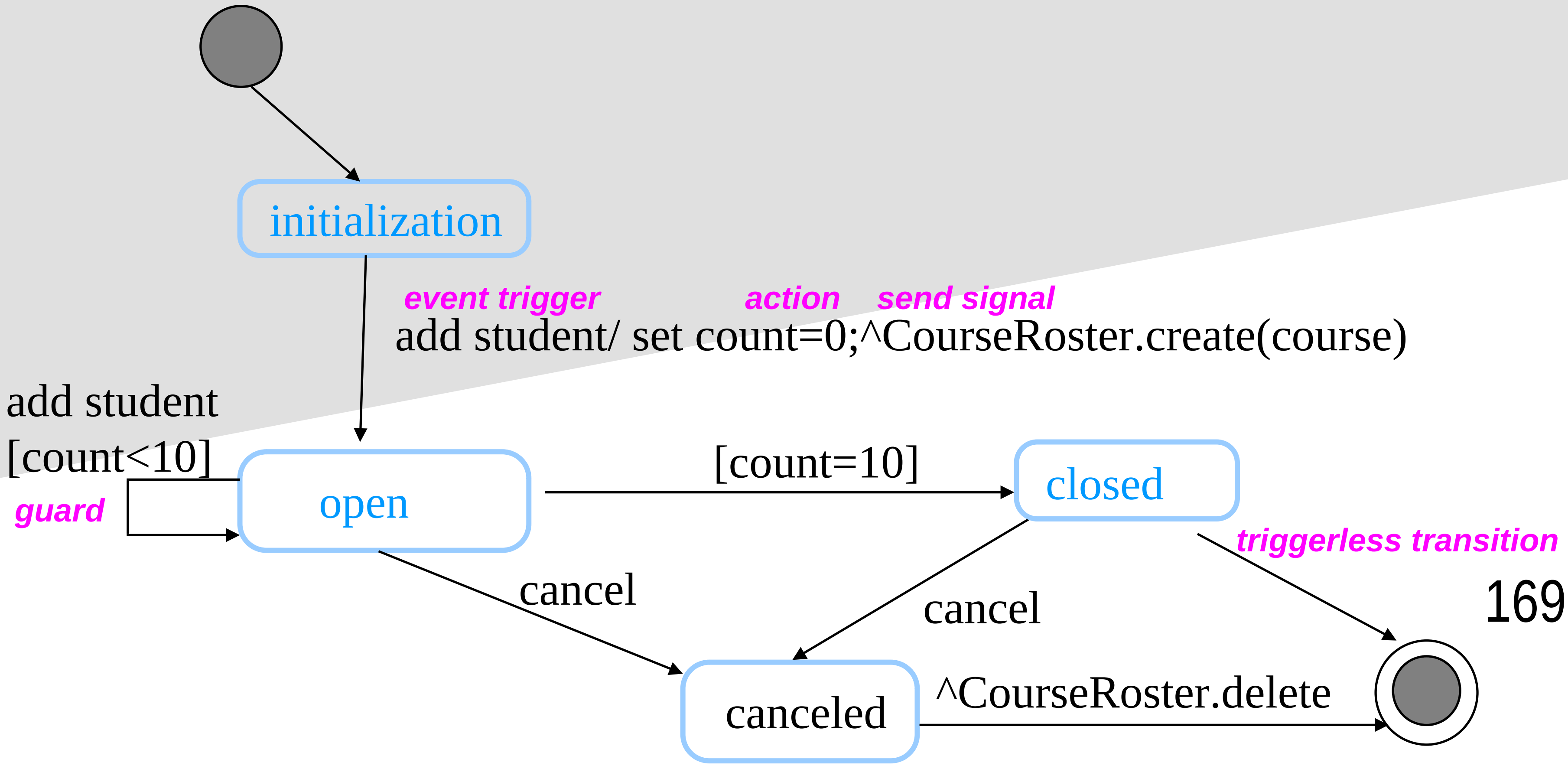
- ***State*** - condition or situation during which an object/instance may perform some activity; The state of an object is characterized by the value of one or more of its attributes.

- ***Activity*** - ongoing *non-atomic* execution within a state machine.
- ***Action*** - executable *atomic* computation that results in a change in state of the model or the return of a value.

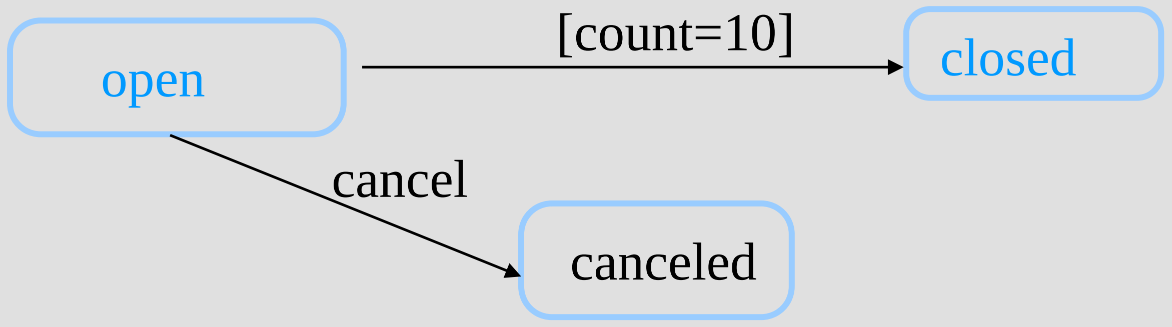

event name [guard condition] / action
event name (a:T) [guard condition] / action
^object/className.event
send
15 sec

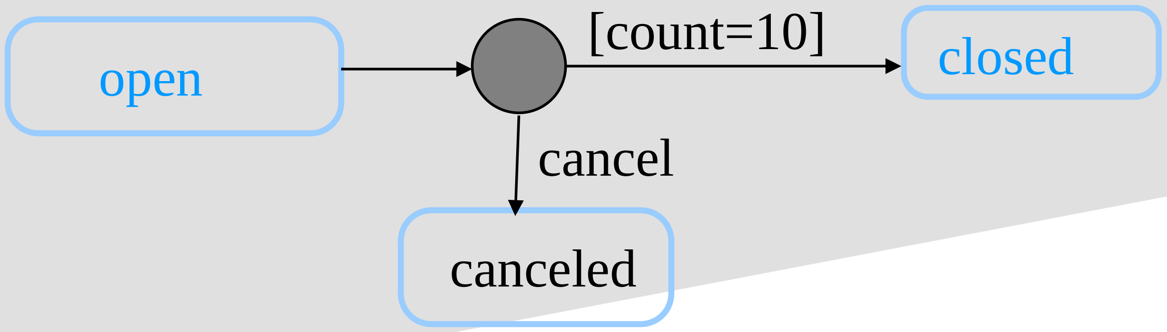
State Transitions

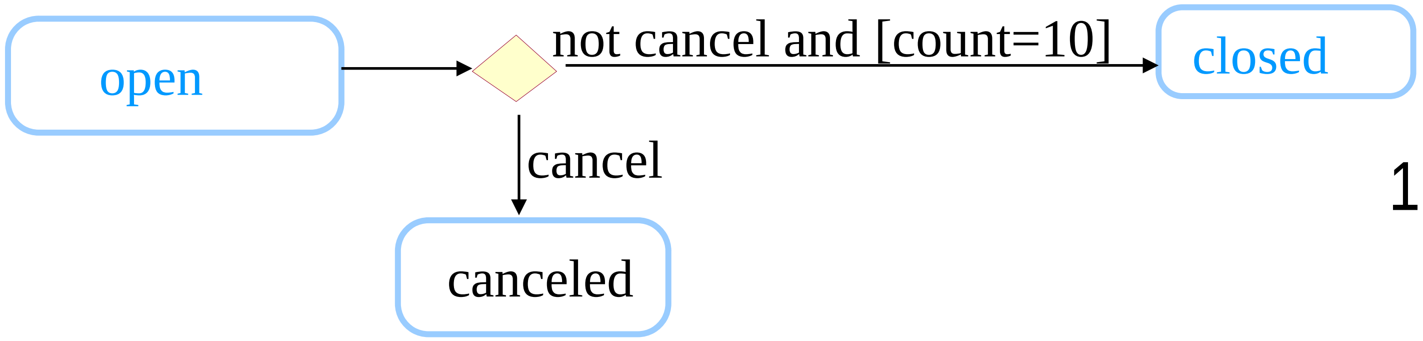
- Each diagram must have one and only one start state
- A diagram may have one or more stop states
- Automatic transition - occurs when the activity of one state completes
- Non-automatic transition - caused by a named event



State Transitions – notational variation

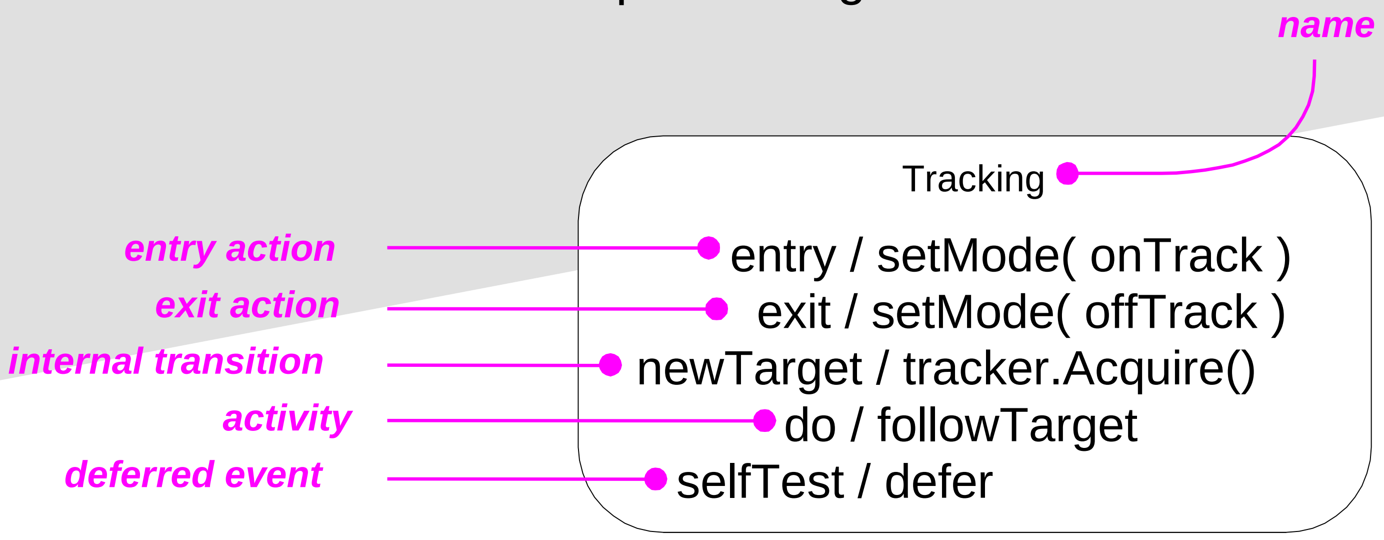






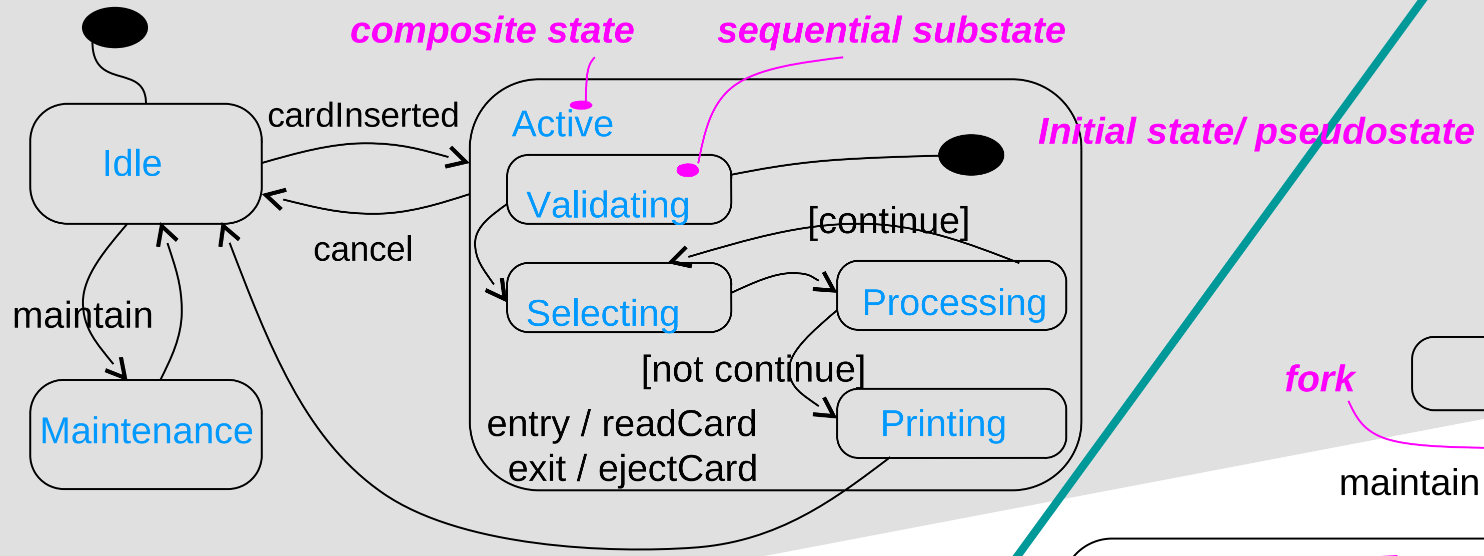
Advanced States

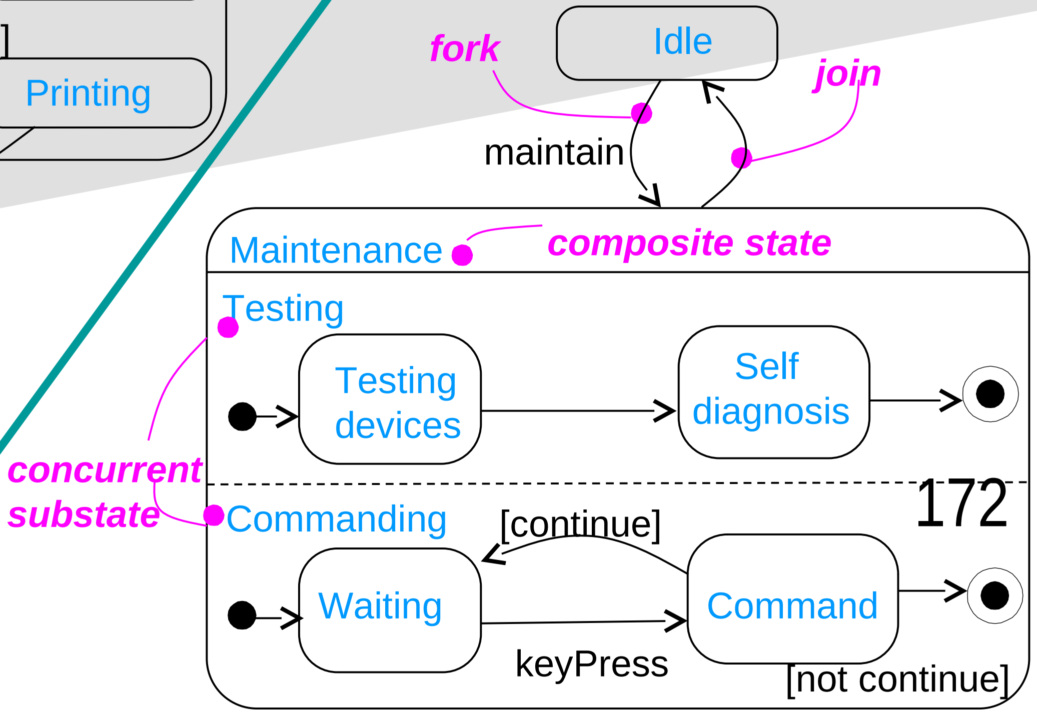
- **Entry & exit actions** - actions that always occur upon entry into or exit away from a state regardless of transition.
- **Internal Transitions** - triggered by events but don't change state.
- **Activities** - ongoing behavior which continues until interrupted.
- **Deferred events** - events ignored by the current state, but postponed for later processing.



Substates

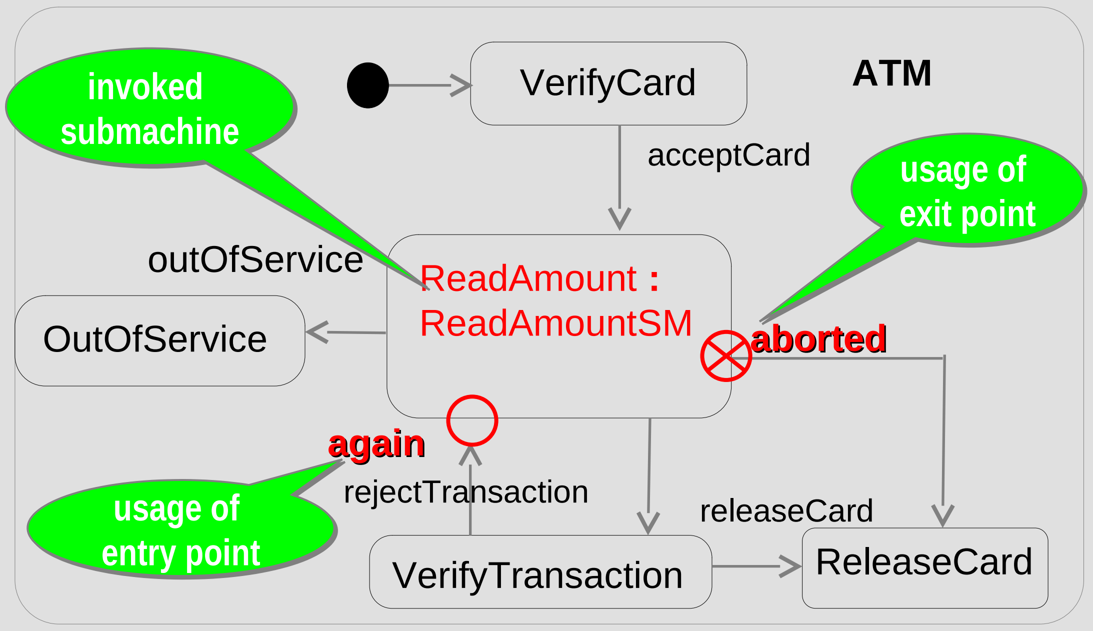
- Substate -- state nested inside of another state.
- Sequential substates (then a nonorthogonal state)
- Concurrent substates (then an orthogonal state)

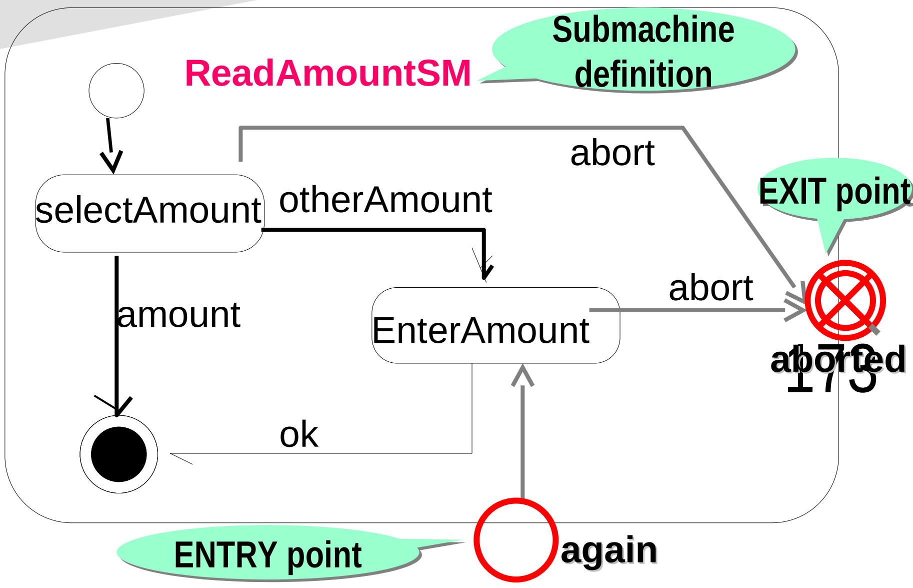




Modular Submachines

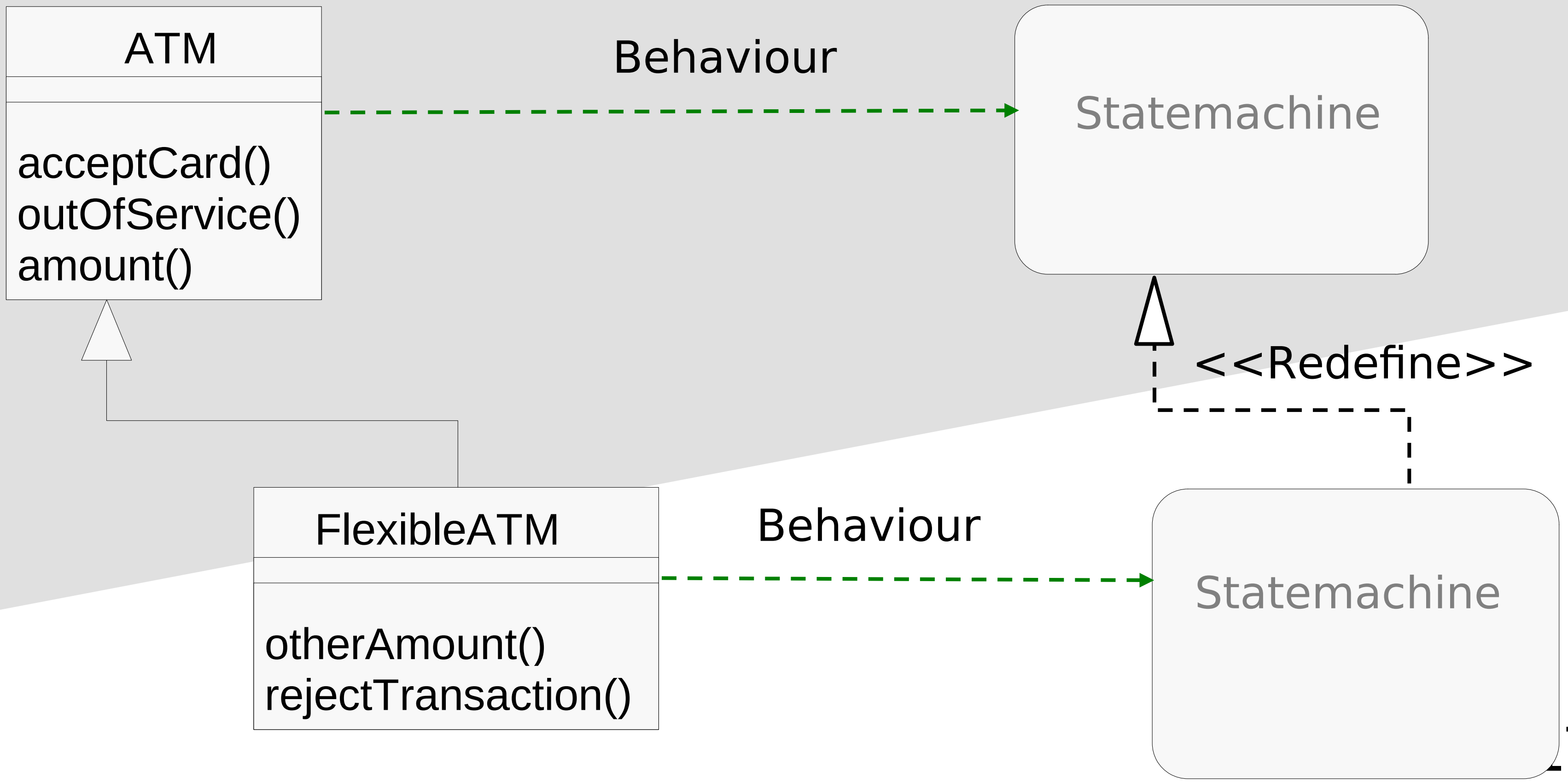
http://www.xpdian.com/UML2.0changes.html





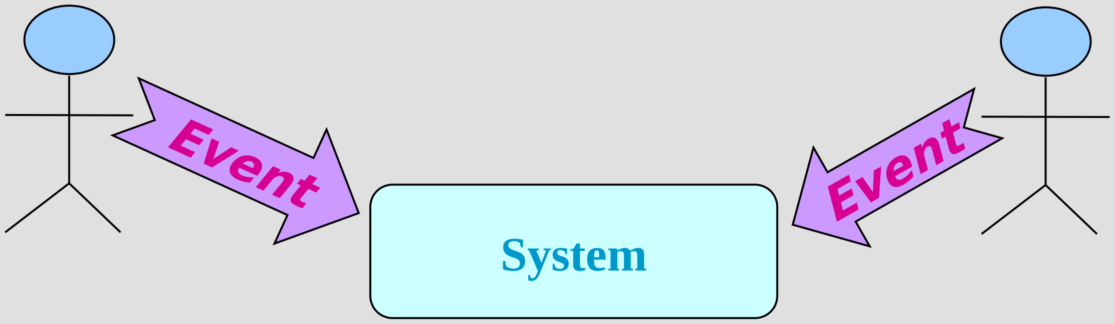
Specialization

- Redefinition as part of standard class specialization

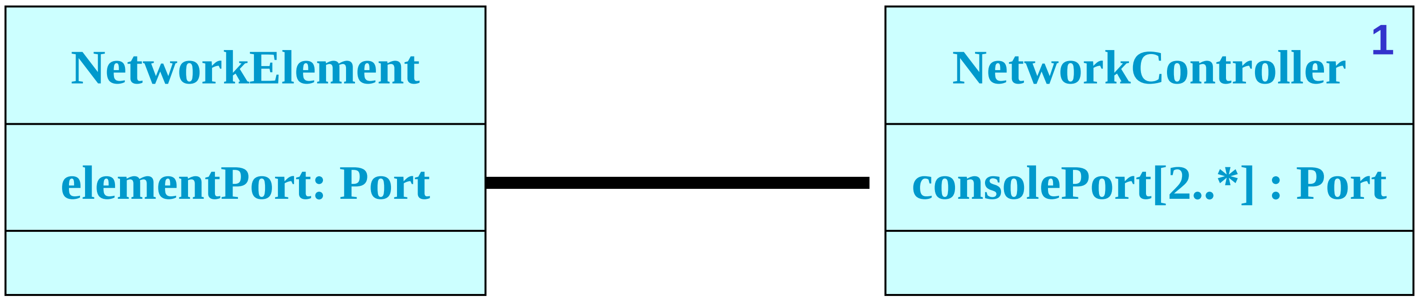


Events – External vs. Internal Events

- Events can be categorized into external or internal events.
- ***External*** events are those that pass between the Actors and the system.



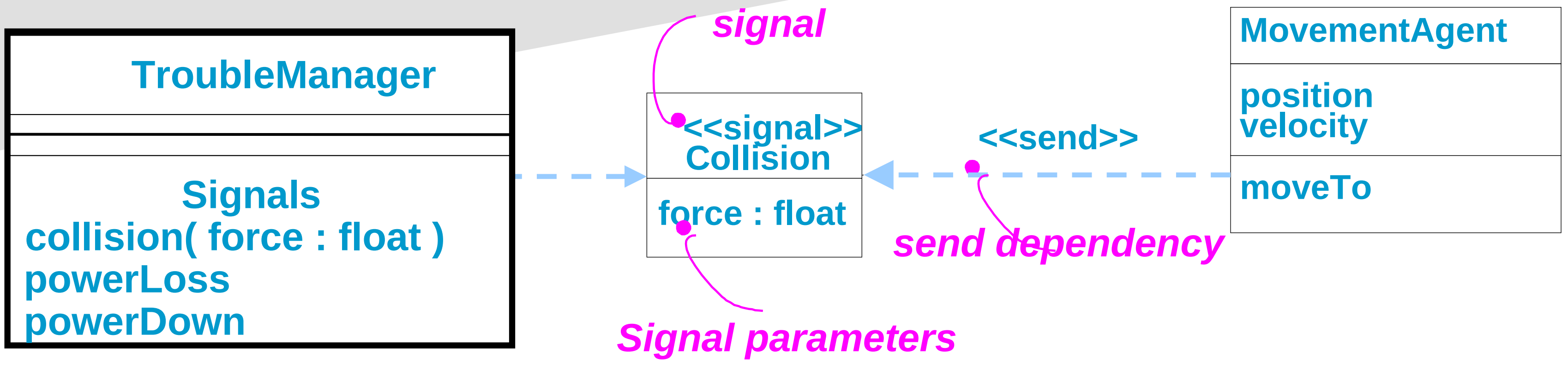
- ***Internal*** events pass between objects residing within the application system.



Events – 4 Kinds: *Signals*; Calls; Passing of Time; Change in State

- *Signal* - kind of event that represents the specification of an ***asynchronous*** stimulus communicated between instances.
  - Modeled as a class
- Dispatched (thrown) by one object and continues flow of execution
- Received (caught) by another object at some future point in time.

- Can be sent as:
  - Action of a state transition
  - Message in an object interaction
  - Dependencies <<send>> show signals sent by a class

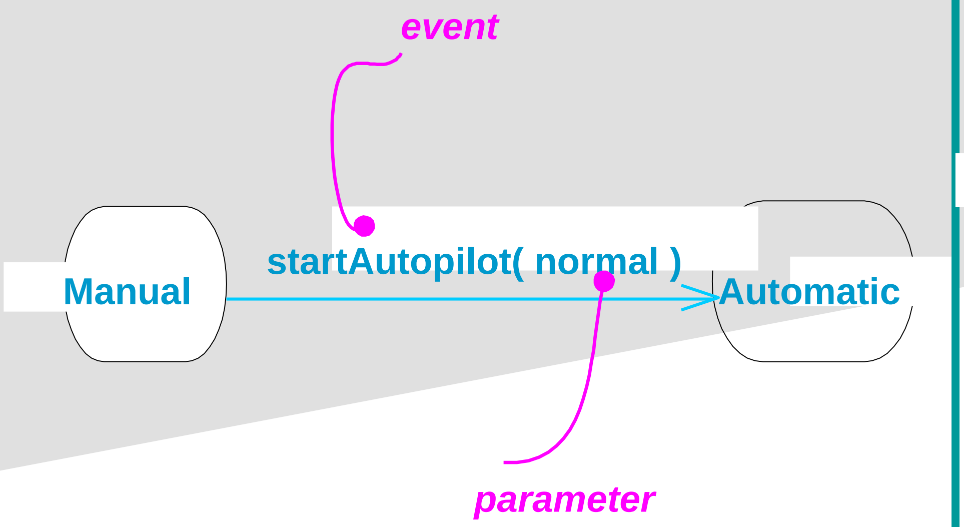


- Modeling Signal Receiver: as an active class; Consider 4th compartment for signals.

Events – 4 Kinds: Signals; *Calls; Passing of Time; Change in State*

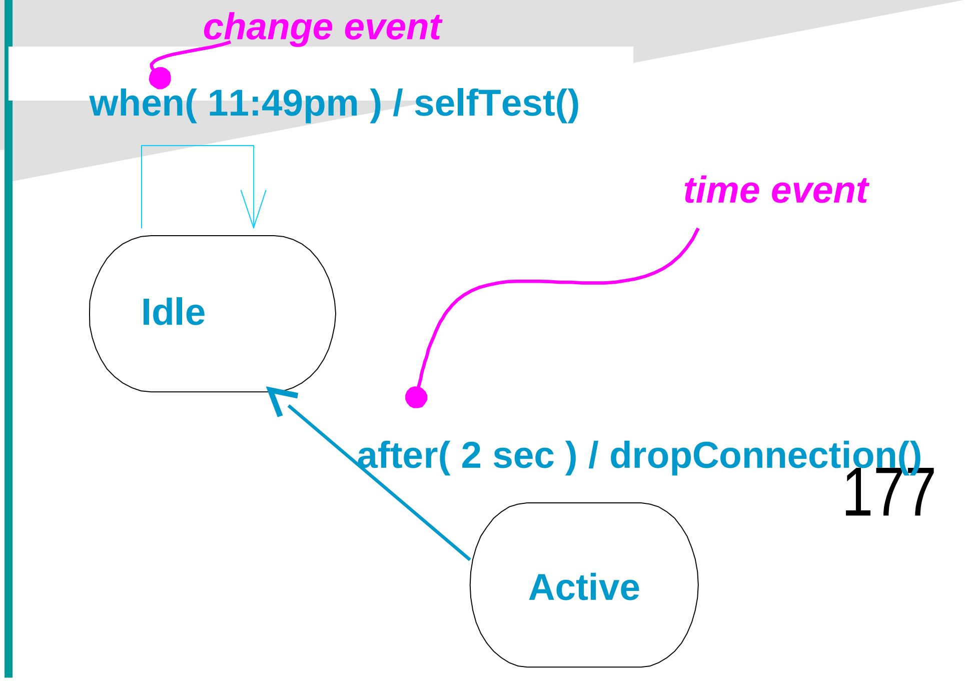
Call Events

- Represents the dispatch of an operation
    - ***Synchronous***



Time & Change Event

- Time event - represents the passage of time: *after(periodOfTime)*
- Change event - represents a change in state or the satisfaction of some condition: *when(booleanExpr)*

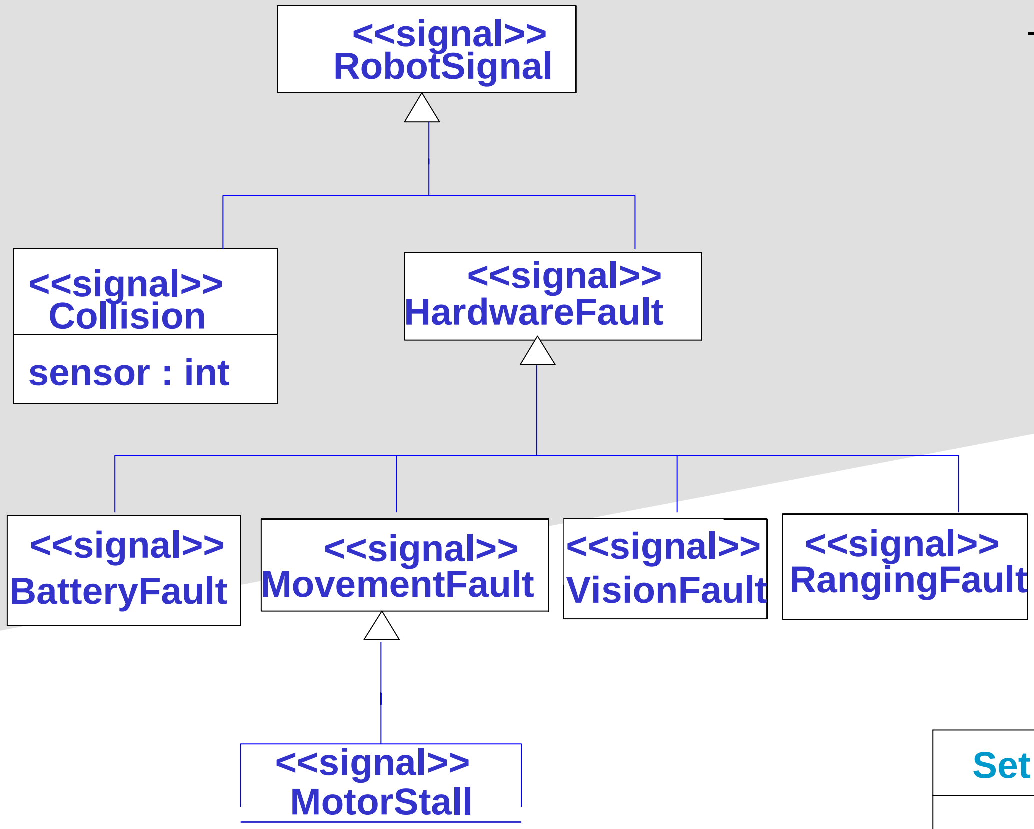


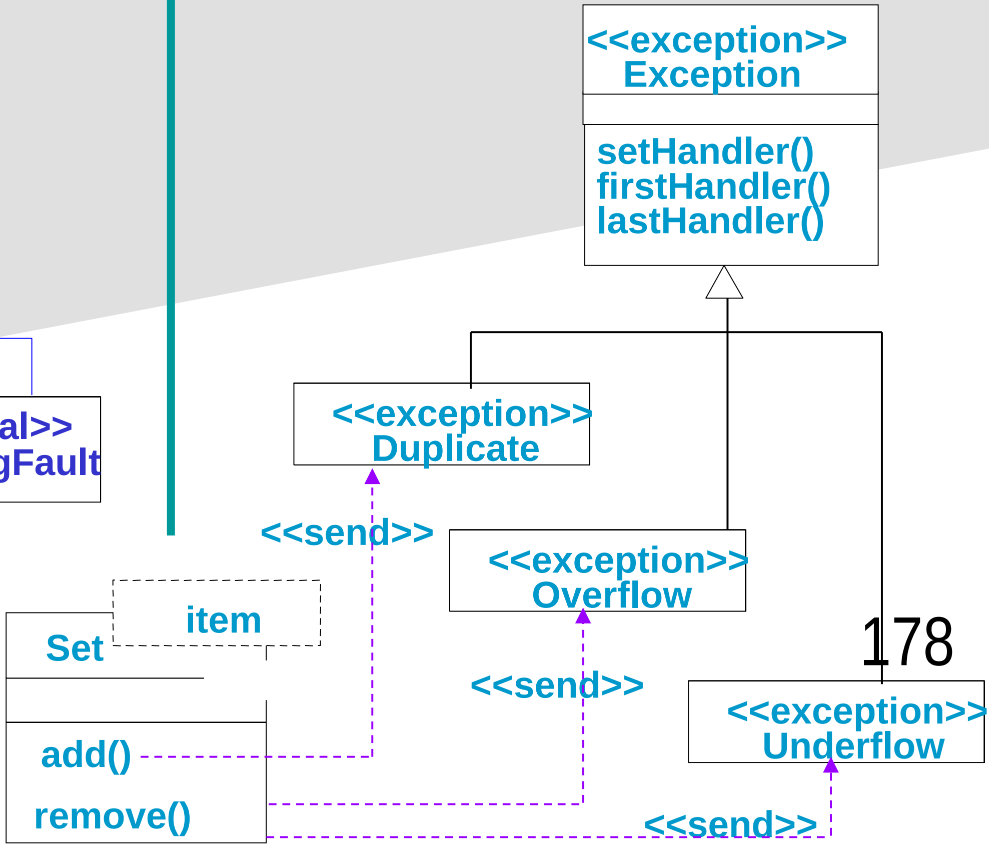
change event
when(11:49pm) / selfTest()
Idle
time event
after(2 sec) / dropConnection()
Active

Modeling Family of Signals and Exceptions

- Signal events are typically hierarchical.
  - Look for common generalizations.
  - Look for polymorphic opportunities.

- Consider the exceptional conditions of each clas
- Arrange exceptions in generalization hierarchy.
- Specify the exceptions that each operation may
  - Use send dependencies
  - Show in operation specification





Activity Diagrams

Behavioral Diagrams

- Use case
- Statechart
- ***Activity***

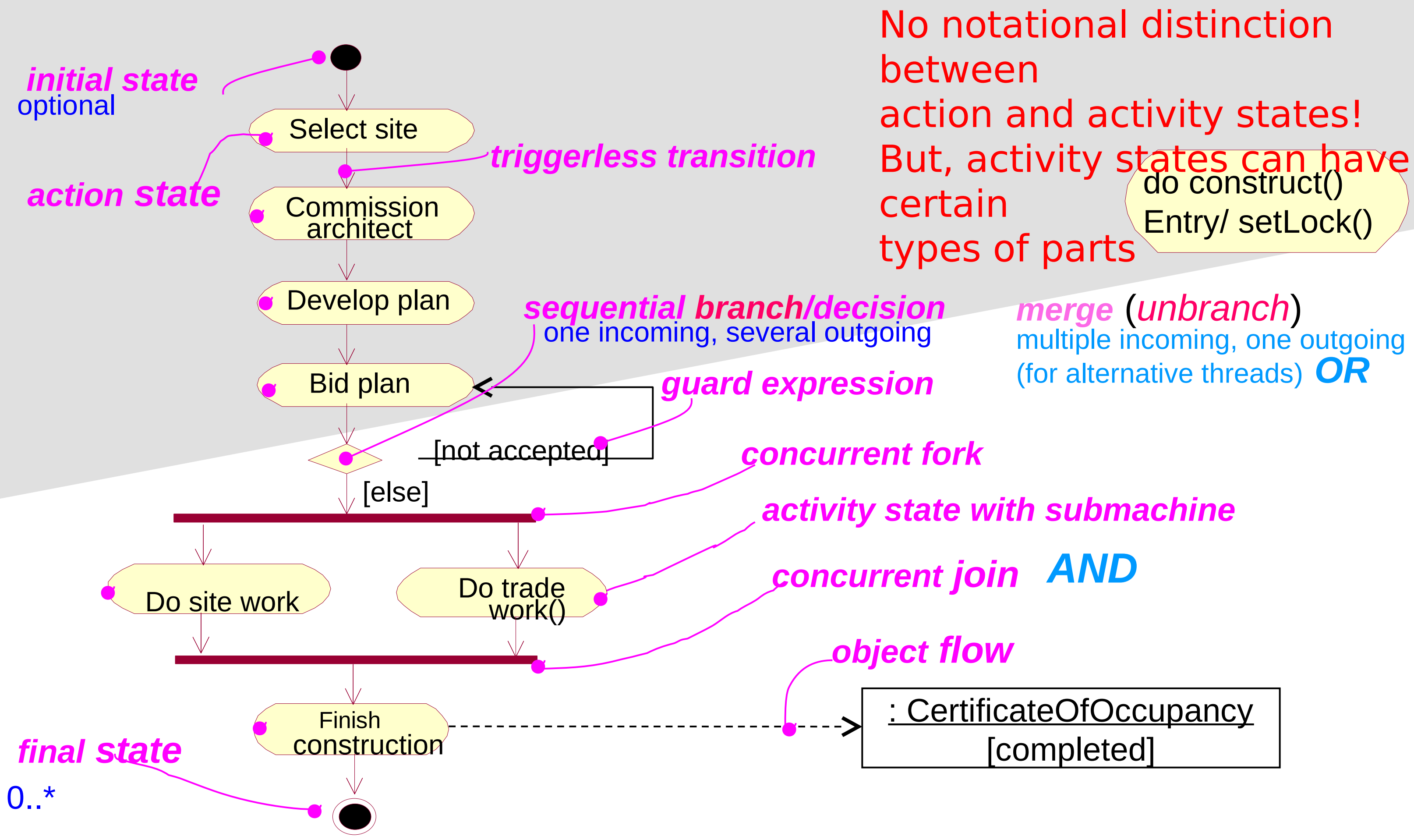
Interaction Diagrams

- Sequence; Communication
- Interaction Overview
- Timing

Activity Diagram Basics

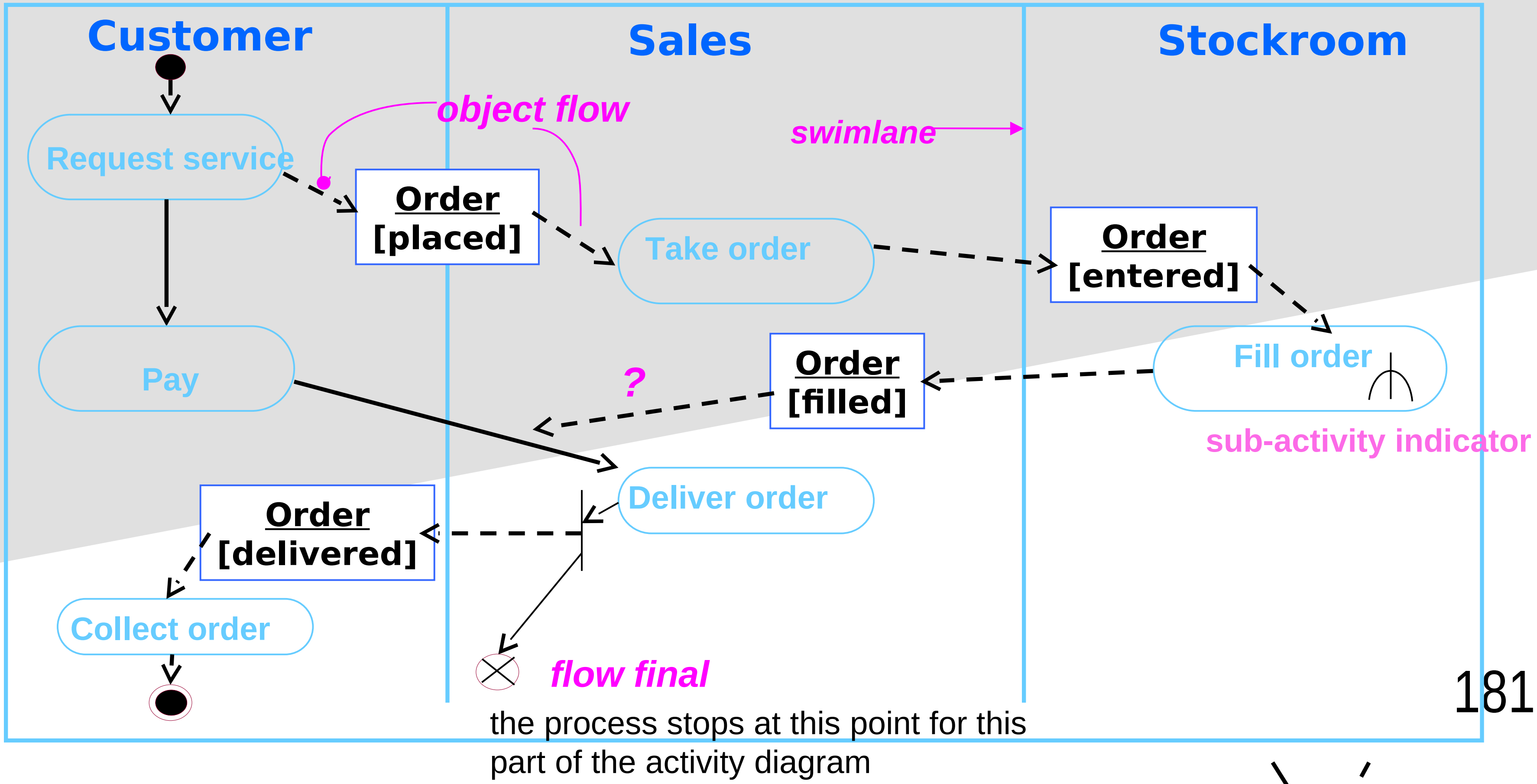
- *Activity Diagram* – a special kind of Statechart diagram, but showing the flow from activity to activity (not from state to state).
- *Activity state* –*non-atomic* execution, ultimately result in some action; a composite made up of other activity/action states; can be represented by other activity diagrams
- *Action state* –*atomic* execution, results in a change in state of the system or the return of a value (i.e., calling another operation, sending a signal, creating or destroying an object, or some computation); non-decomposable

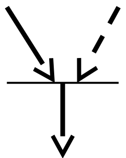
No notational distinction between action and activity states!
But, activity states can have certain types of parts



Swimlanes & Object Flow

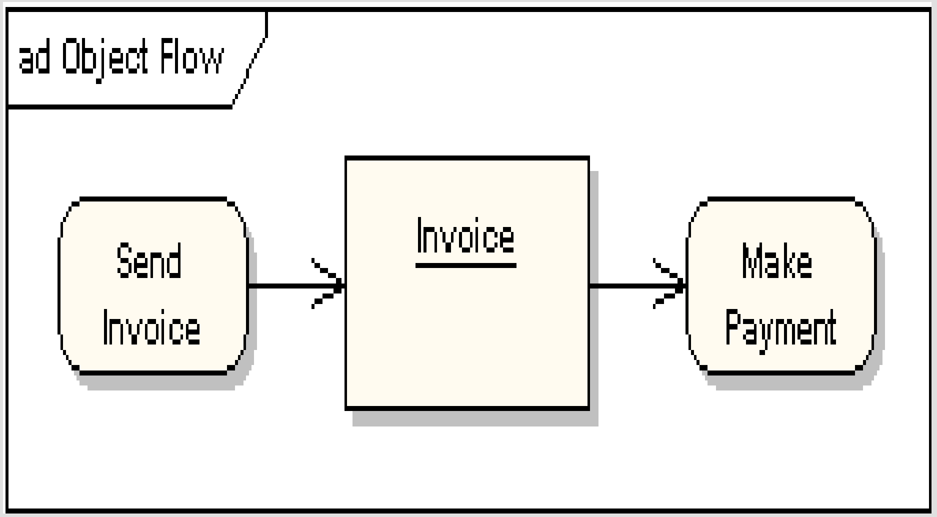
- A *swimlane* is a kind of package.
- Every activity belongs to exactly one swimlane, but transitions may cross lanes.
- *Object flow* – objects connected using a dependency to the activity or transition that creates, destroys, or modifies them





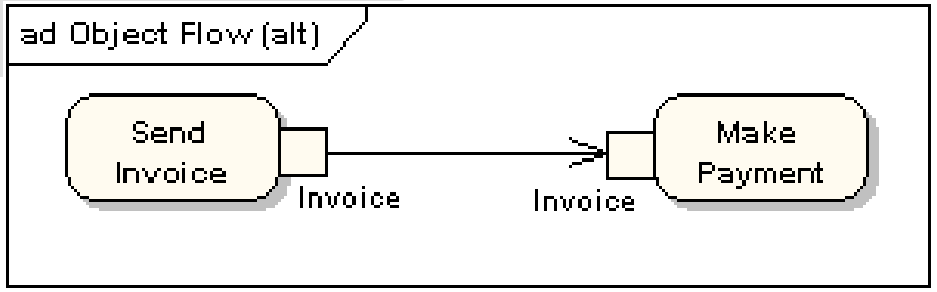
What if using pins?

Object Flows and Pins

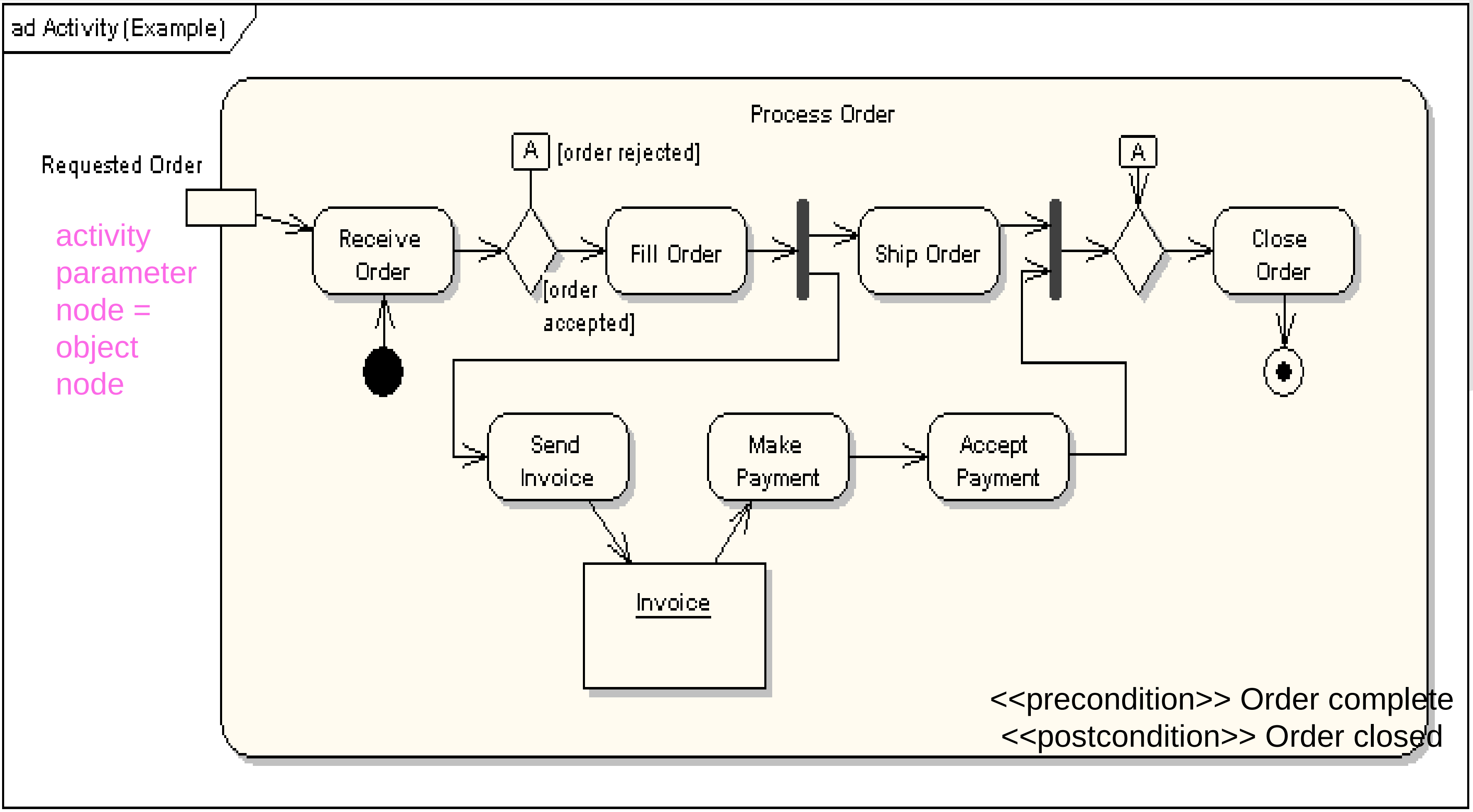


Invoice inv;
inv = new Invoice;
FillOrder(inv);

A shorthand notation: use input pins and output pins (parameters).



A Simple Example – Order Processing

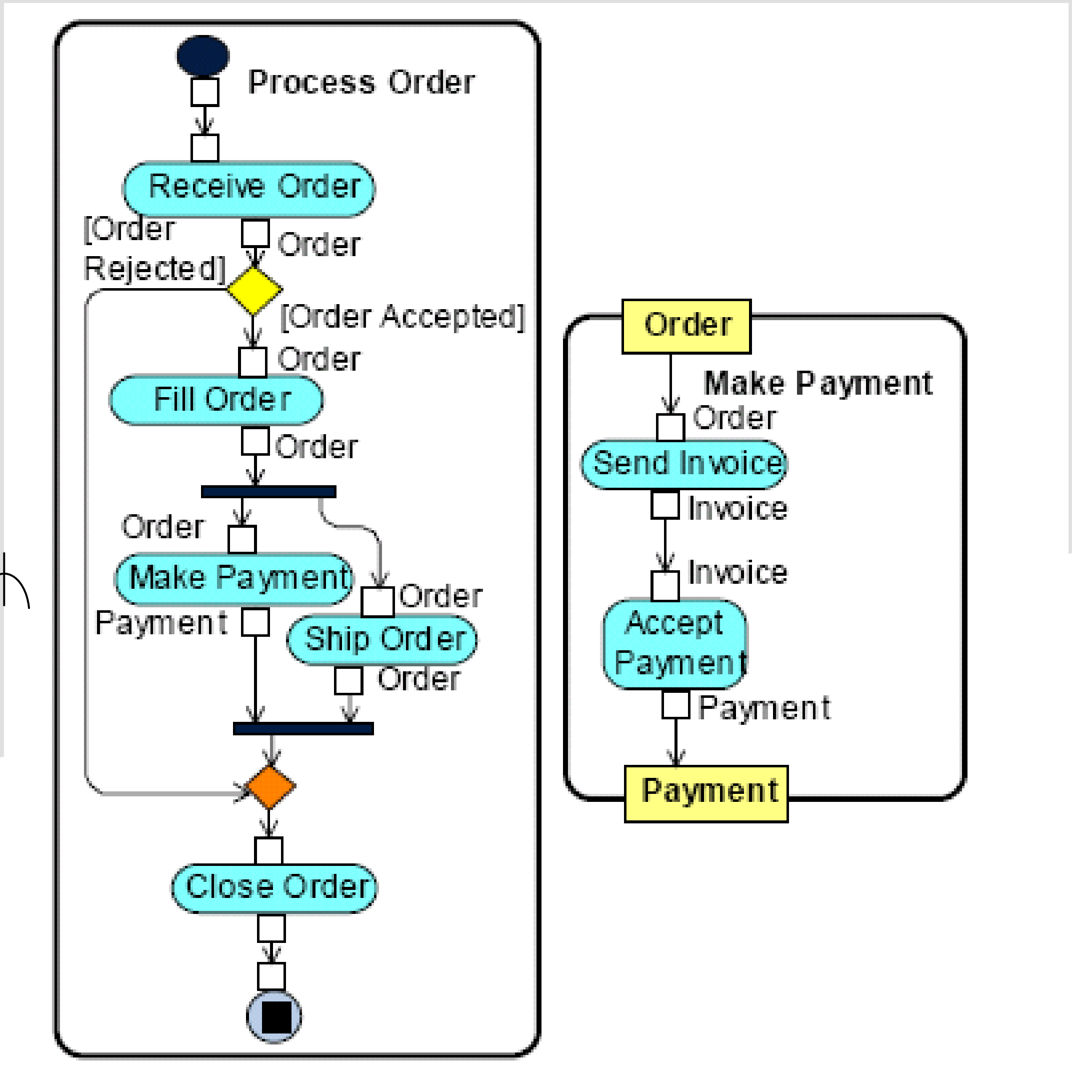


activity parameter node = object node

<<precondition>> Order complete
<<postcondition>> Order closed

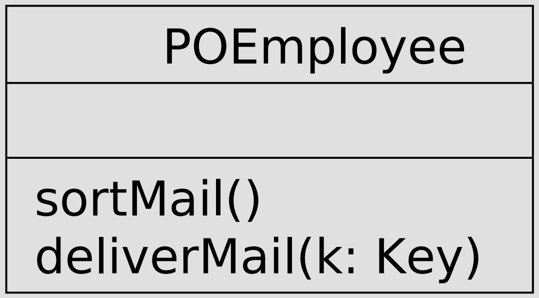
What if using swimlanes?

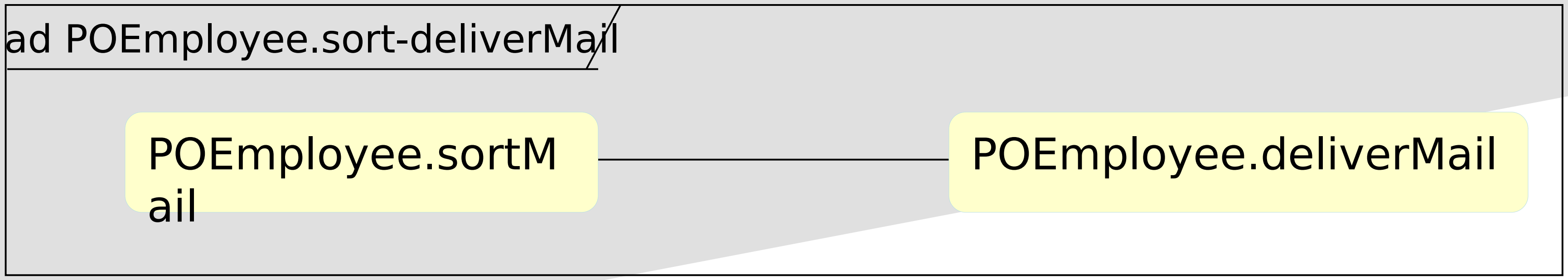
A Simple Example – Order Processing *Using sub-activity*

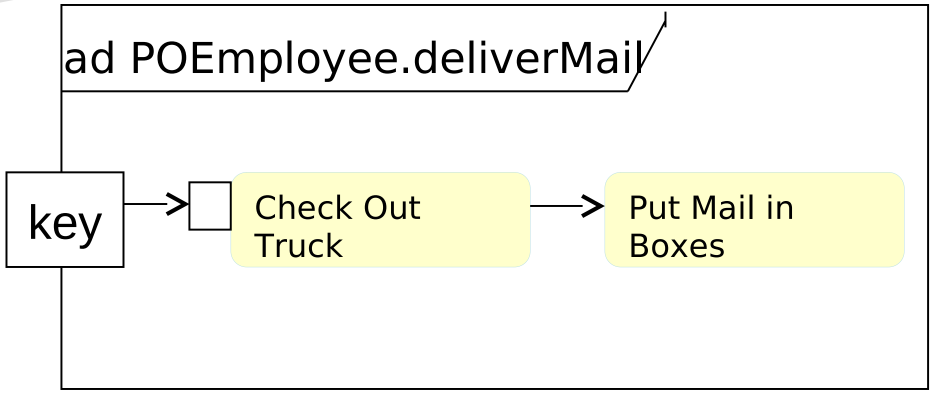


Is this the same as the previous one?

Activity Diagram even as Method

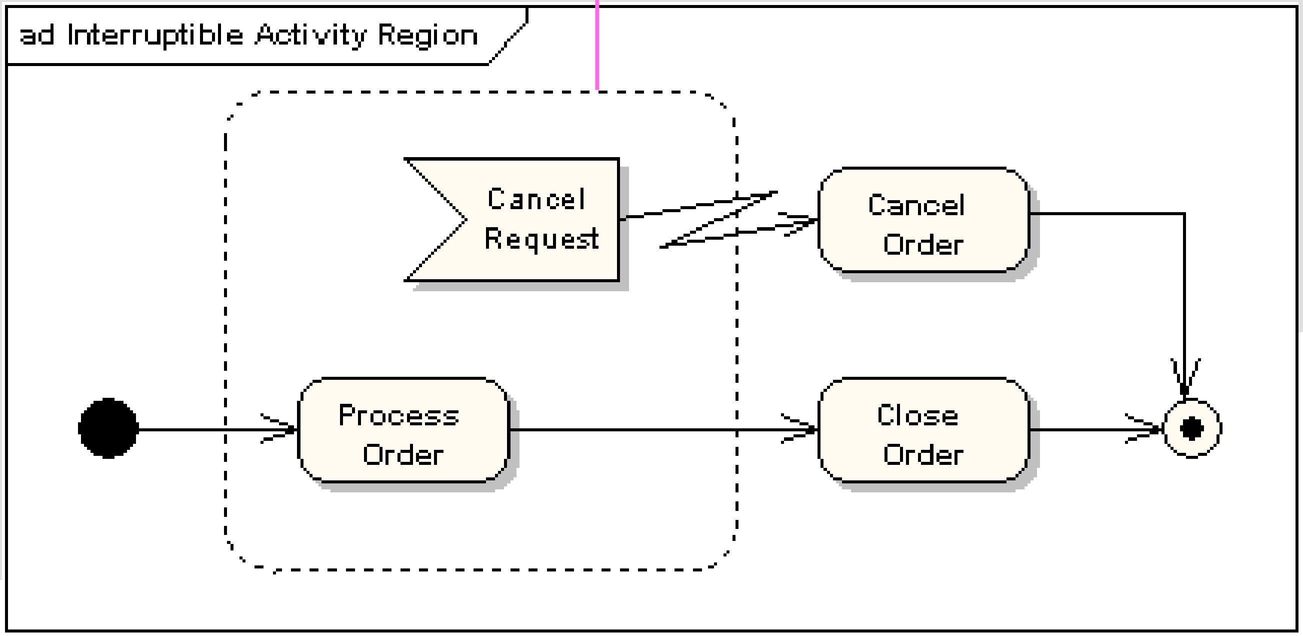






Interruptible Activity Region

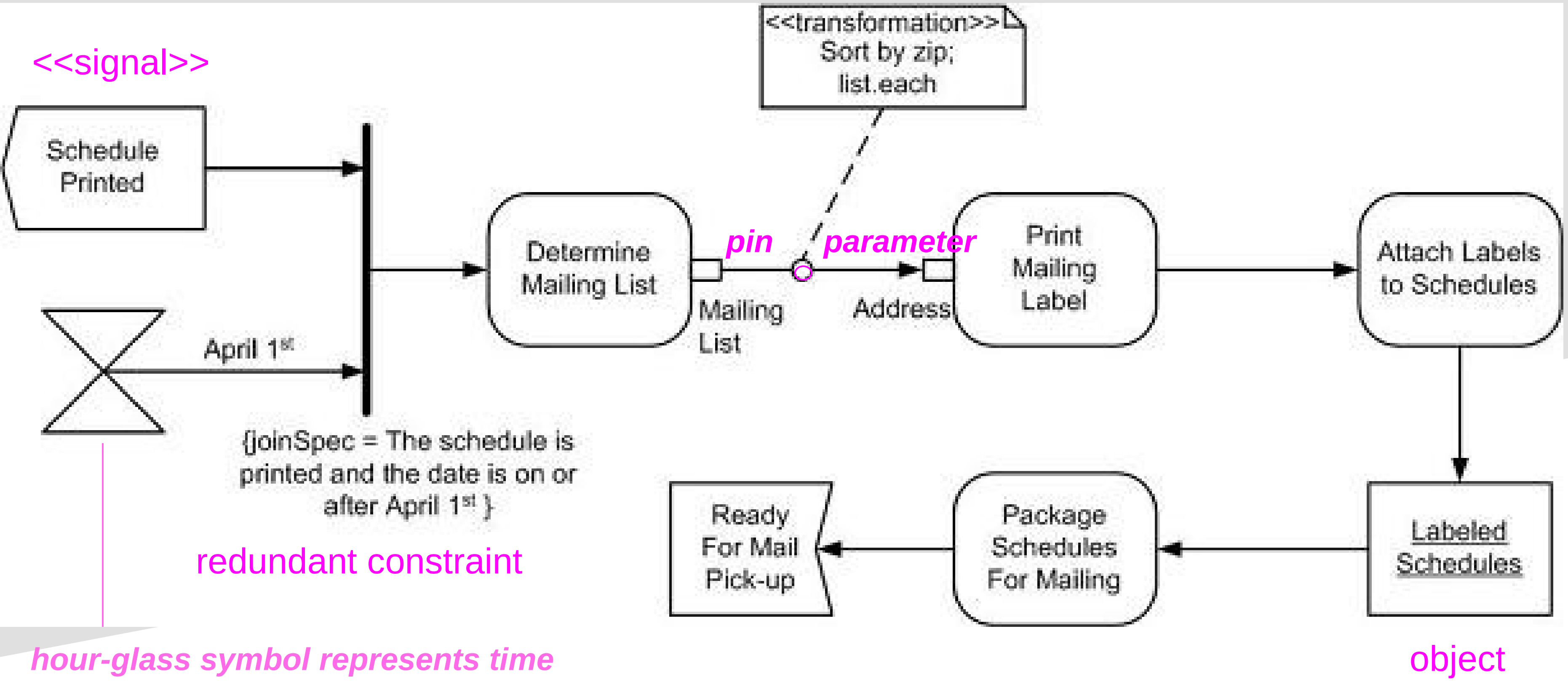
- An ***interruptible activity region*** surrounds a group of actions that can be interrupted.



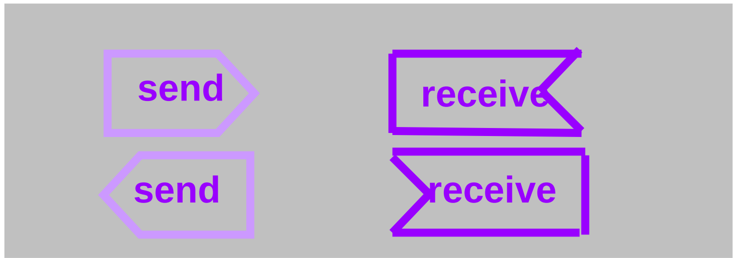
- the Process Order action will execute until completion, then pass control to the Close Order action, unless a Cancel Request interrupt is ***receive***d which will pass control to the Cancel Order action.

An Activity Diagram – Distributing schedules

http://www.agilemodeling.com/artifacts/activityDiagram.htm



<<signal>>
Schedule Printed
April 1st
{joinSpec = The schedule is printed and the date is on or after April 1st }
redundant constraint
hour-glass symbol represents time
<<transformation>> Sort by zip; list.each
Determine Mailing List
pin
Mailing List
parameter
Address
Print Mailing Label
Attach Labels to Schedules
Labeled Schedules
object
Package Schedules For Mailing
Ready For Mail Pick-up



Pins, Parameters, Effects

(www.jot.fm/issues/issue_2004_01/column3.pdf)

- *effect* that their actions have on objects that move through the pin: one of the four values *create, read, update, or delete*.
- Take Order creates an instance of Order and Fill Order reads it.
- The ***create*** effect is only possible on ***outputs***; and the ***delete*** effect is only possible on ***inputs***.

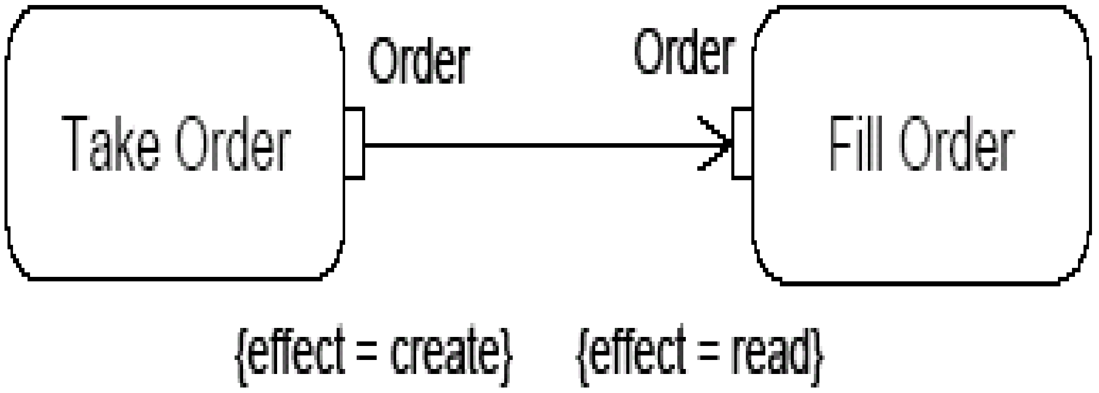


Figure 4: Effect

Multiple Tokens

- Object nodes can hold more than one value at a time, and some of these values can be the same.

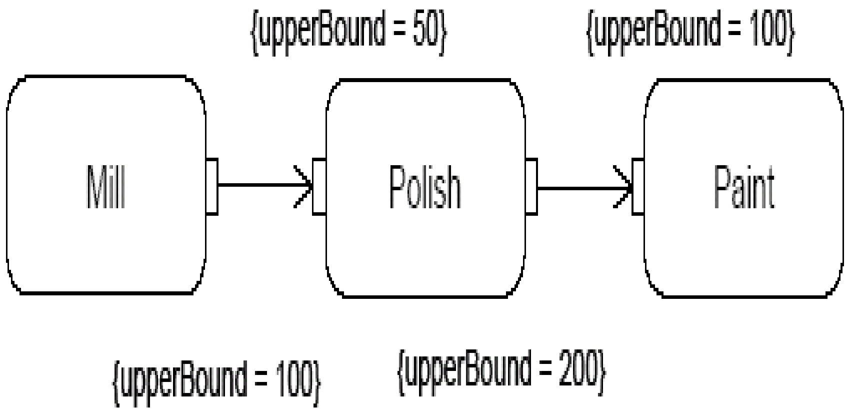
- *Upper bound:* the maximum number of tokens an object node can hold, including any duplicate values.

- At runtime, when the number of values in an object node reaches its upper bound, it cannot accept any more.

- If painting is delayed too much for some reason, the input pin will reach its upper bound, and parts from polishing will not be able to move downstream;

  If painting is delayed further, the output pin of polishing will fill up and the polishing behavior will not be able to transfer out polished parts;

  Unless the polishing behavior has an object node internal to it that buffers output parts, it will not be able to take parts from its input pin, which will likewise fill up and propagate the backup; Only when the input pin to PAINT goes below its upper bound will parts be able to flow again.



Multiple Tokens - Ordering

- Object nodes holding multiple values can specify the *order* in which values move downstream.
- The default is FIFO (a pipe); users can change this to LIFO (a queue), or specify their own behavior to select which value is passed out first.

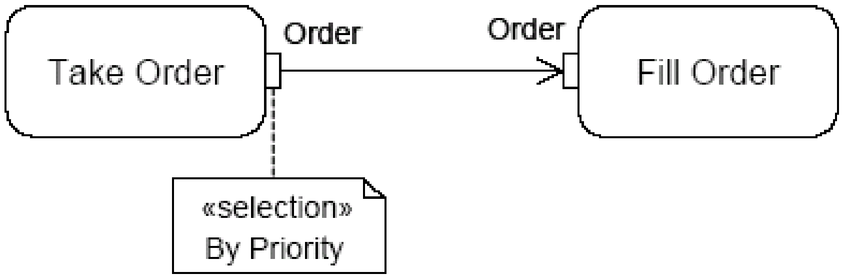


Figure 6: Selection Behavior

Non-Determinism

(http://www.jot.fm/issues/issue_2004_01/column3.pdf)

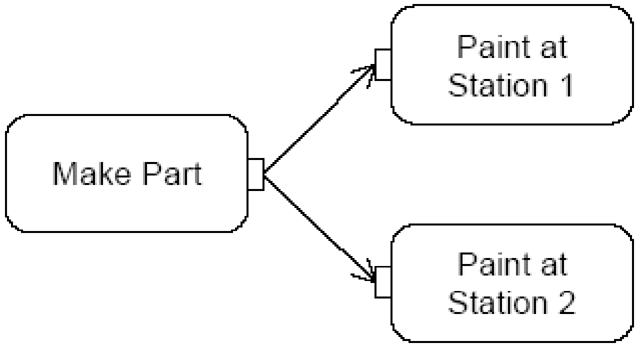


Figure 10: Token Competition

Parameter Multiplicity & Object Flow Weight

- *Minimum multiplicity* on an *input* parameter means a behavior or operation cannot be invoked by an action until the number of values available at each of its input pins reaches the minimum for the corresponding parameter, which might be zero

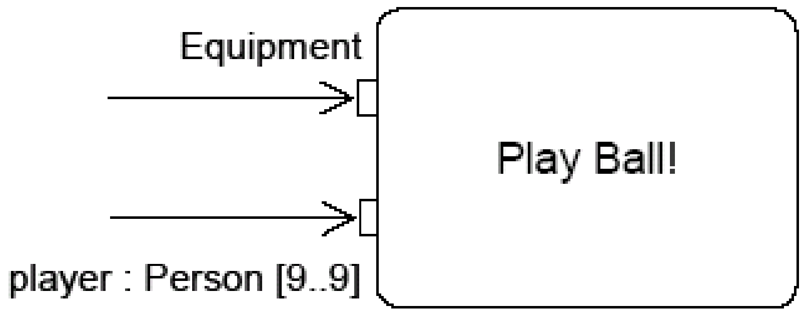


Figure 8: Parameter Multiplicity

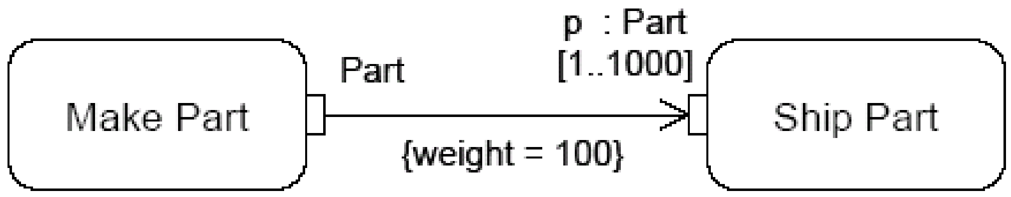


Figure 9: Object Flow Weight

- Weight on object flow edges specifies the minimum number of values that can traverse an object flow edge at one time.

Interaction Overview Diagrams

Behavioral Diagrams

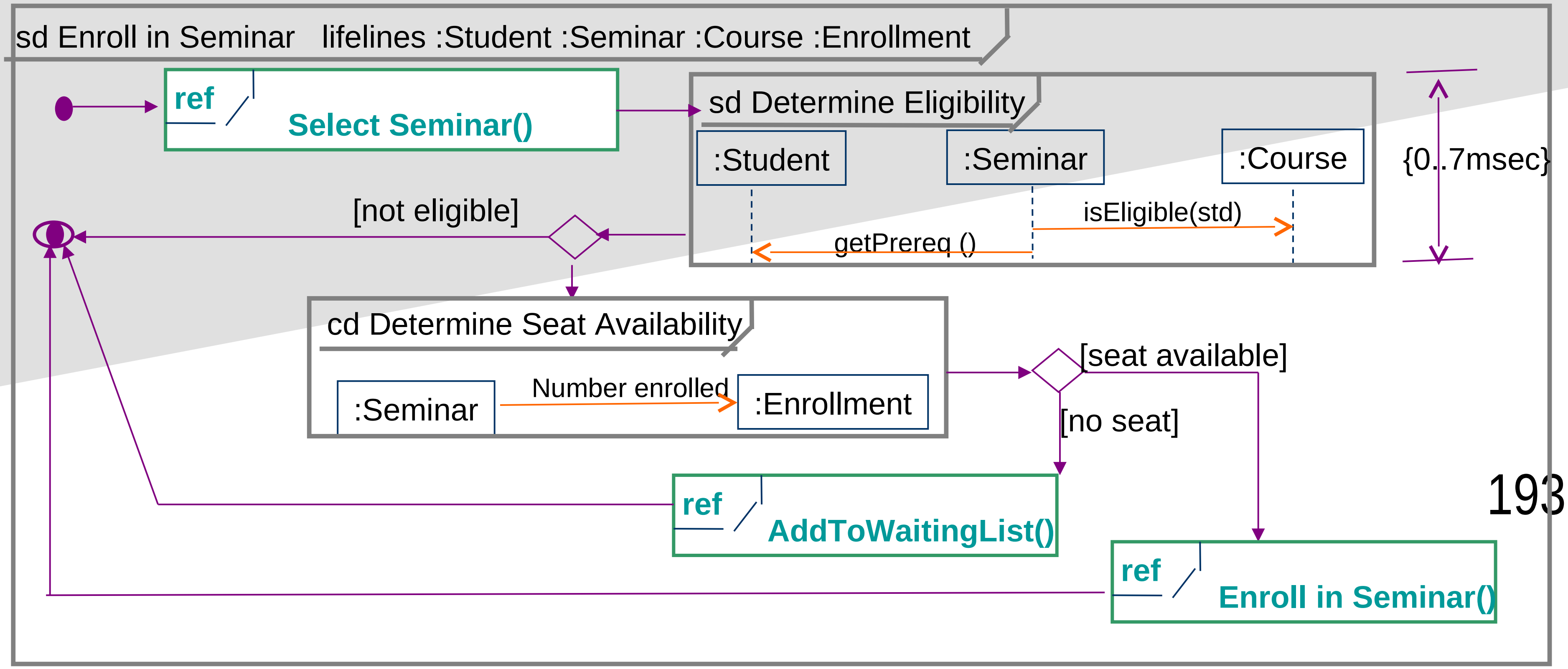
- Use case
- Statechart
- Activity

Interaction Diagrams

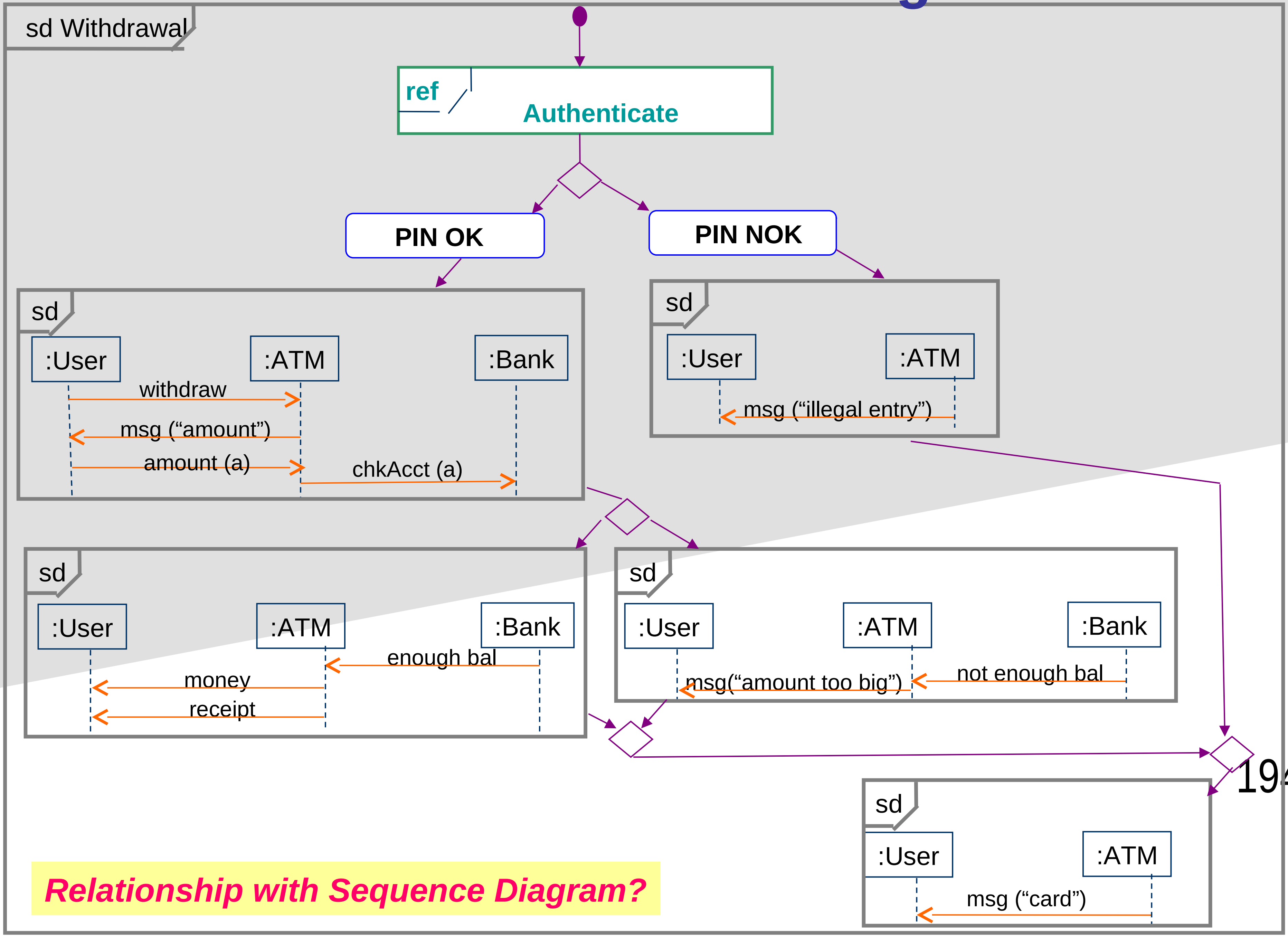
- Sequence; Communication
- ***Interaction Overview***
- Timing

Interaction Overview Diagram (http://www.agilemodeling.com/artifacts/interactionOverviewDiagram.htm)

- variants on UML activity diagrams which overview control flow.
- The nodes within the diagram are ***frames***, not activities
- Two types of frame shown:
  - interaction frames depicting any type of UML interaction diagram (sequence diagram: *sd*, communication diagram: *cd*, timing diagram: *td* , interaction overview diagram: *iod*)
  - interaction occurrence frames (*ref;* typically anonymous) which indicate an activity or operation to invoke.



Interaction Overview Diagram



Relationship with Sequence Diagram?

Timing Diagrams

Behavioral Diagrams

- Use case
- Statechart
- Activity

Interaction Diagrams

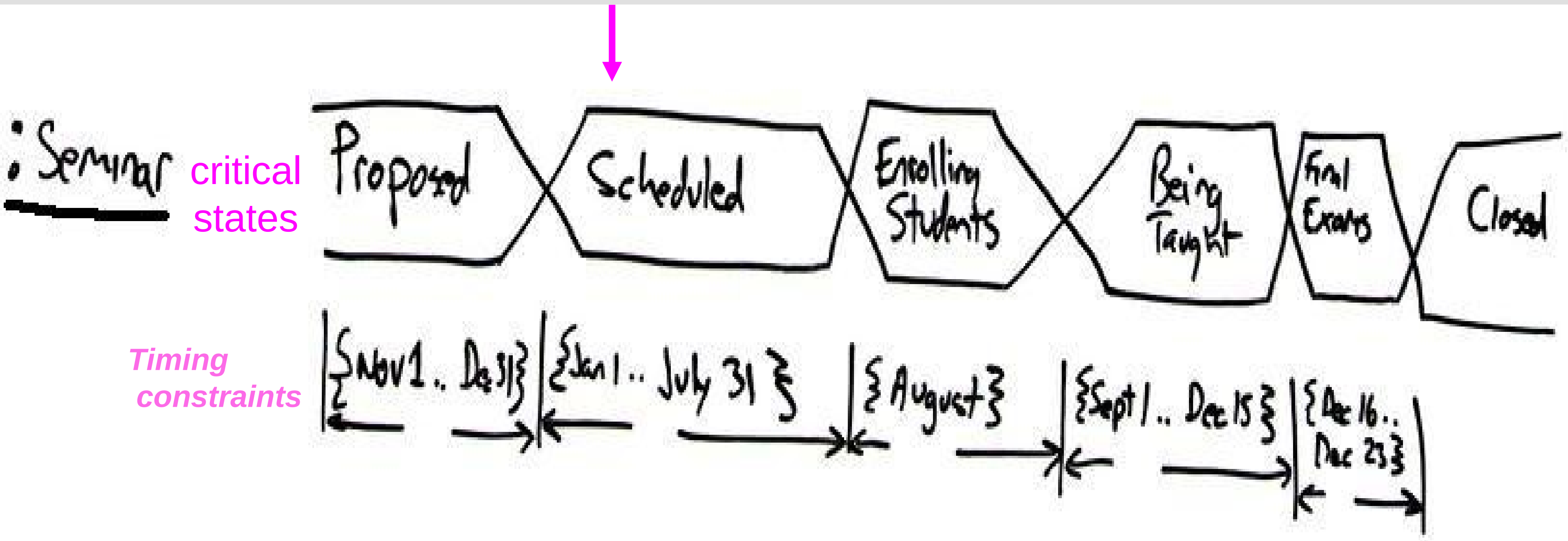
- Sequence; Communication
- Interaction Overview
- ***Timing***

Interaction Diagram: Timing Diagram

- To explore the behaviors of 0..* objects throughout a given period of time.
- Two basic flavors: ***concise*** notation and robust notation

critical states

Timing constraints

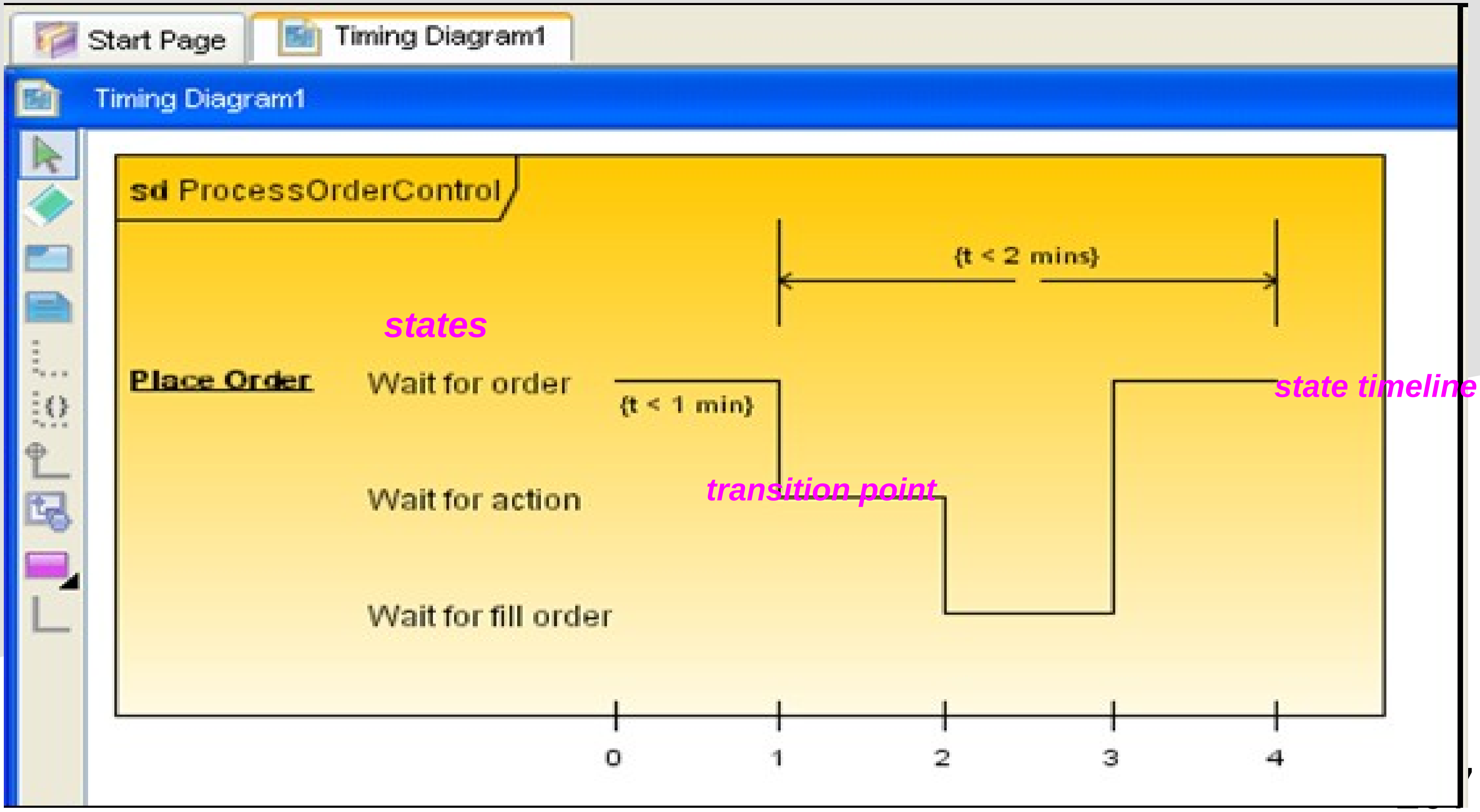


The lifecycle of a single seminar

- The critical states – *Proposed*, *Scheduled*, *Enrolling Students*, *Being Taught*, *Final Exams*, *Closed*
- The two lines surrounding the states are called a general value lifeline.
- When the two lines cross one another it indicates a ***transition point*** between states.
- ***Timing constraints*** along the bottom of the diagram,
  indicating the period of time during which the seminar is in each state.

Interaction Diagram: Timing Diagram (robust notation)

http://www.visual-paradigm.com/highlight/highlightuml2support.jsp

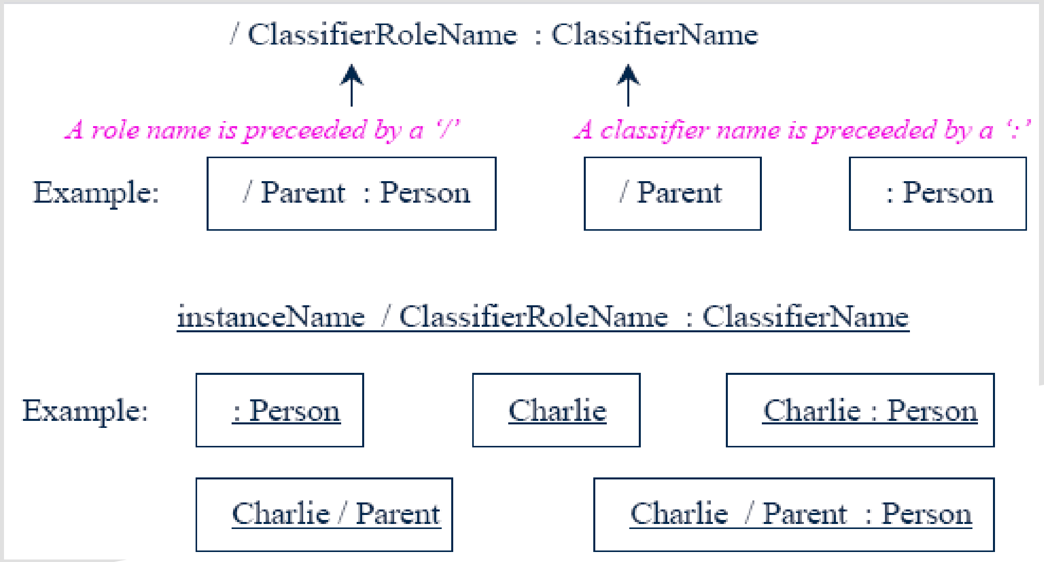


timing ruler w. tick marks

Can you transform this into a concise notation?

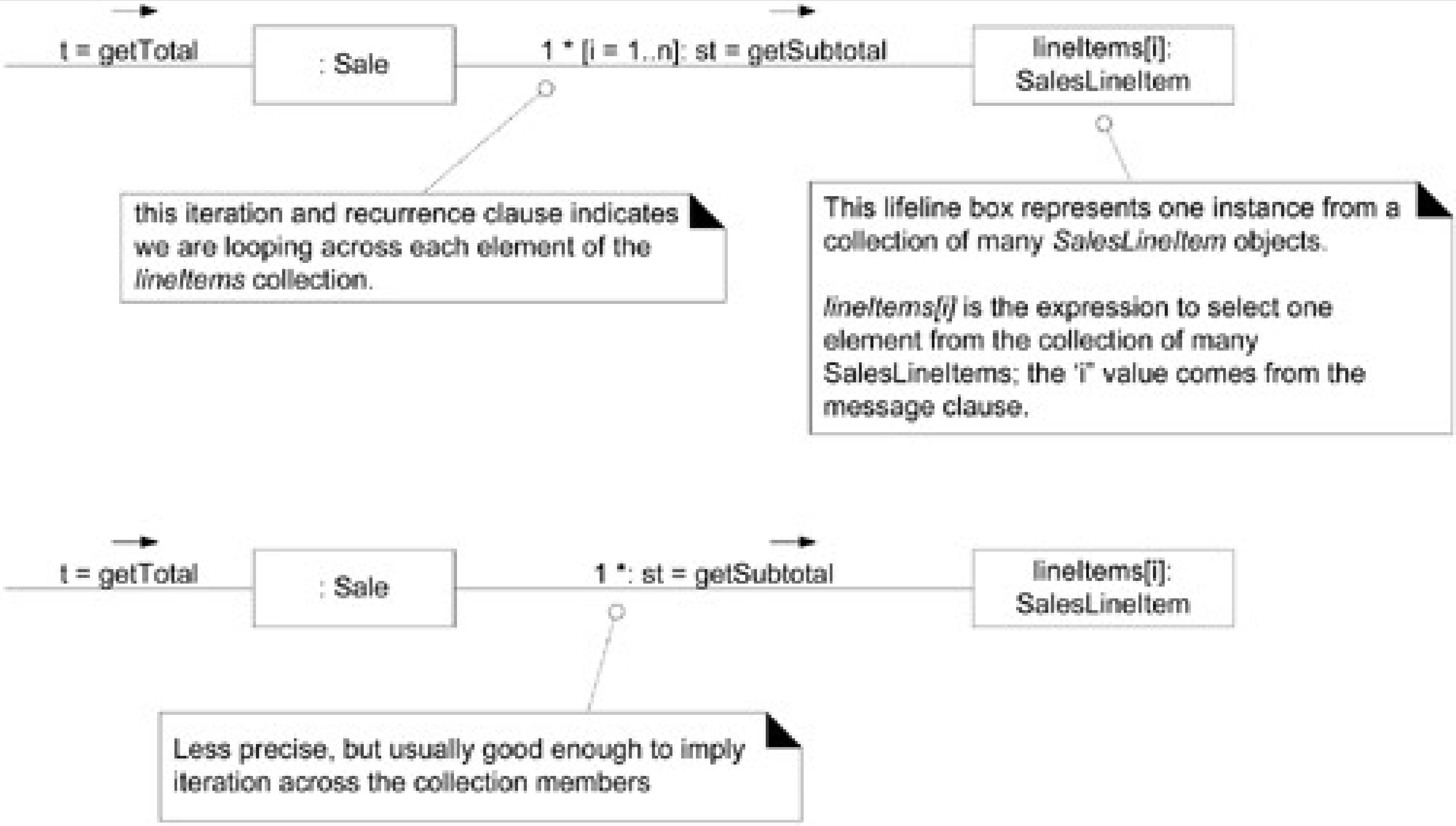
Appendix: Miscellaneous

Role Names


/ ClassifierRoleName : ClassifierName
A role name is preceeded by a '/'
A classifier name is preceeded by a ':'
Example:
/ Parent : Person
/ Parent
: Person
instanceName / ClassifierRoleName : ClassifierName
Example:
: Person
Charlie
Charlie : Person
Charlie / Parent
Charlie / Parent : Person

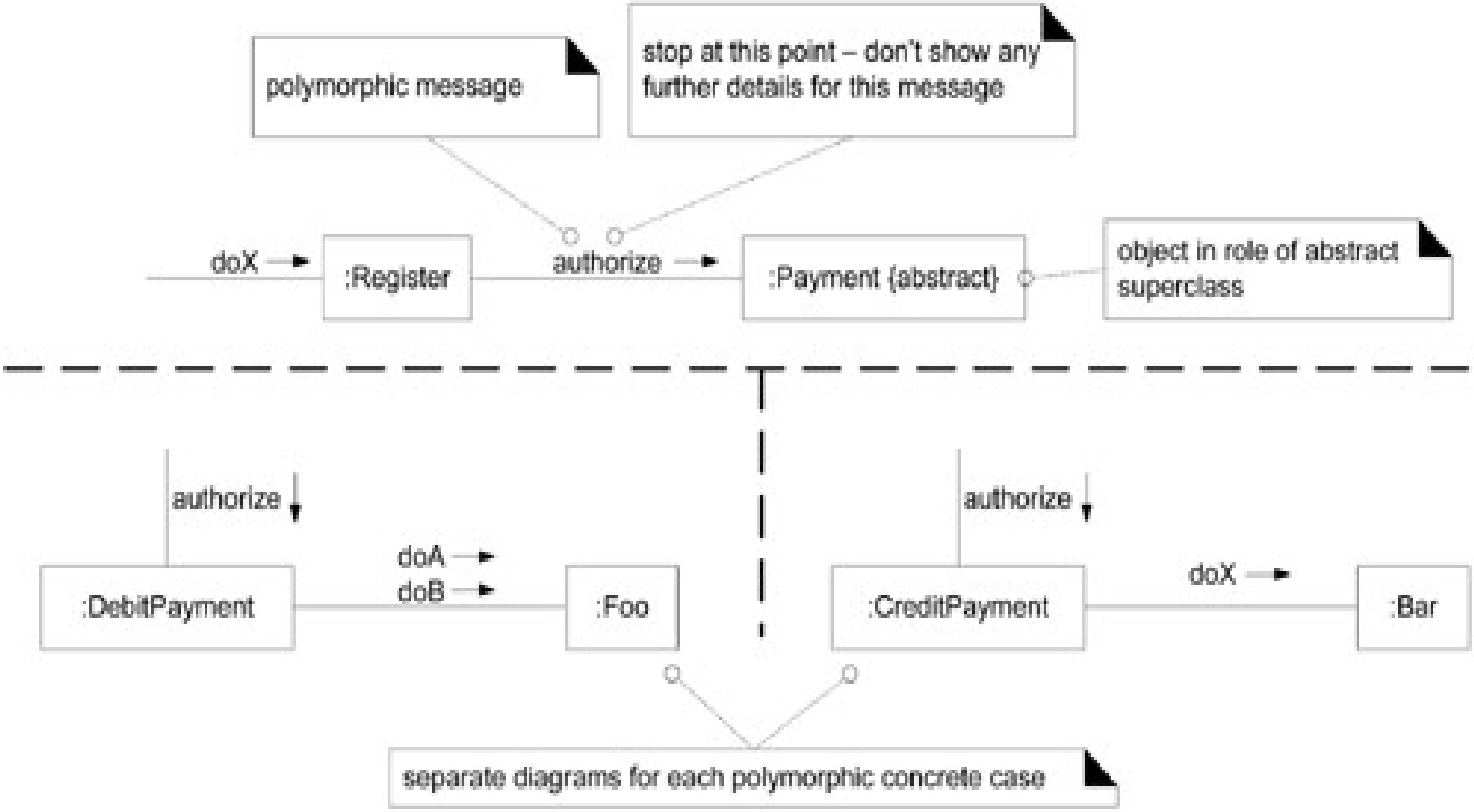
Iterative Messages

[Craig Larman] [http://www.phptr.com/articles/article.asp?p=360441&seqNum=6&rl=1]


t = getTotal
: Sale
1 * [i = 1..n]: st = getSubtotal
lineItems[i]:
SalesLineItem
this iteration and recurrence clause indicates we are looping across each element of the *lineItems* collection.
This lifeline box represents one instance from a collection of many *SalesLineItem* objects.
lineItems[i] is the expression to select one element from the collection of many SalesLineItems; the 'i" value comes from the message clause.
t = getTotal
: Sale
1 *: st = getSubtotal
lineItems[i]:
SalesLineItem
Less precise, but usually good enough to imply iteration across the collection members

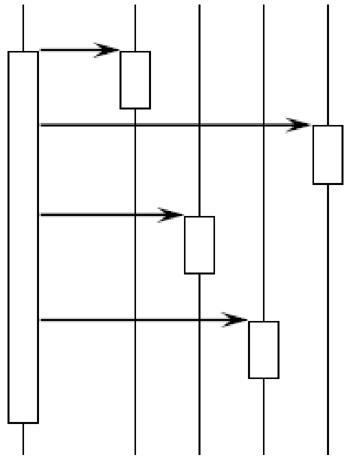
Polymorphic Message

[Craig Larman] [http://www.phptr.com/articles/article.asp?p=360441&seqNum=6&rl=1]


polymorphic message
stop at this point – don't show any further details for this message
doX
:Register
authorize
:Payment {abstract}
object in role of abstract superclass
authorize
:DebitPayment
doA
doB
:Foo
authorize
:CreditPayment
doX
:Bar
separate diagrams for each polymorphic concrete case

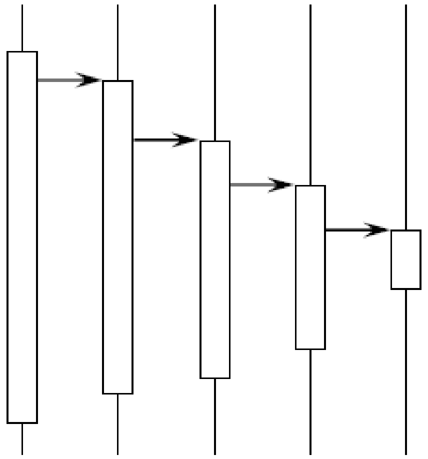
Sequence Diagram Shapes

Fork - centralised



- operations can change order
- new operations may be added

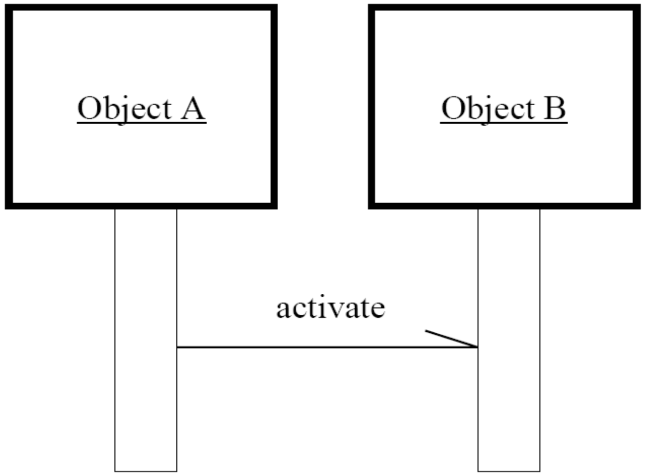
Stair - decentralised



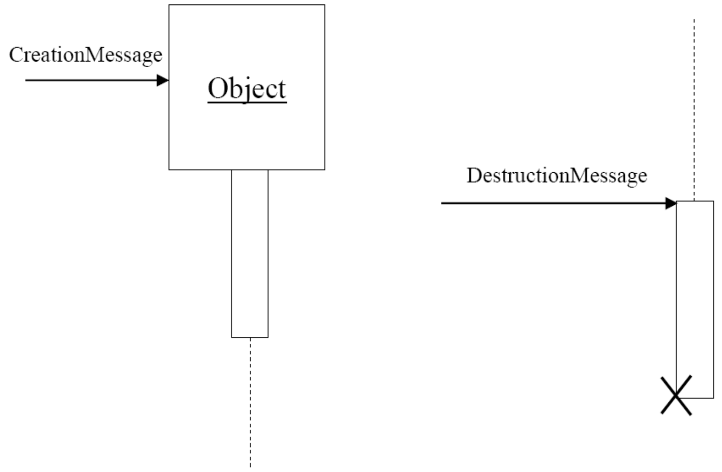
- Operations have strong connections
- performed in same order
- behaviour is encapsulated

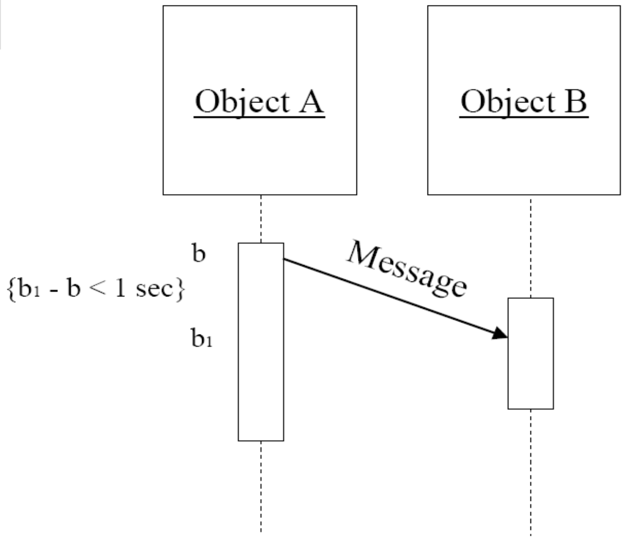
Concurrency

In some systems, objects run concurrently, each with its own thread of control. If the system uses concurrent objects, it is shown by activation, by asynchronous messages and by active objects.



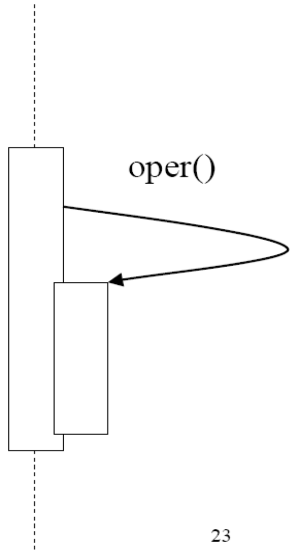
Creation and Destruction





Message with Transmission Time

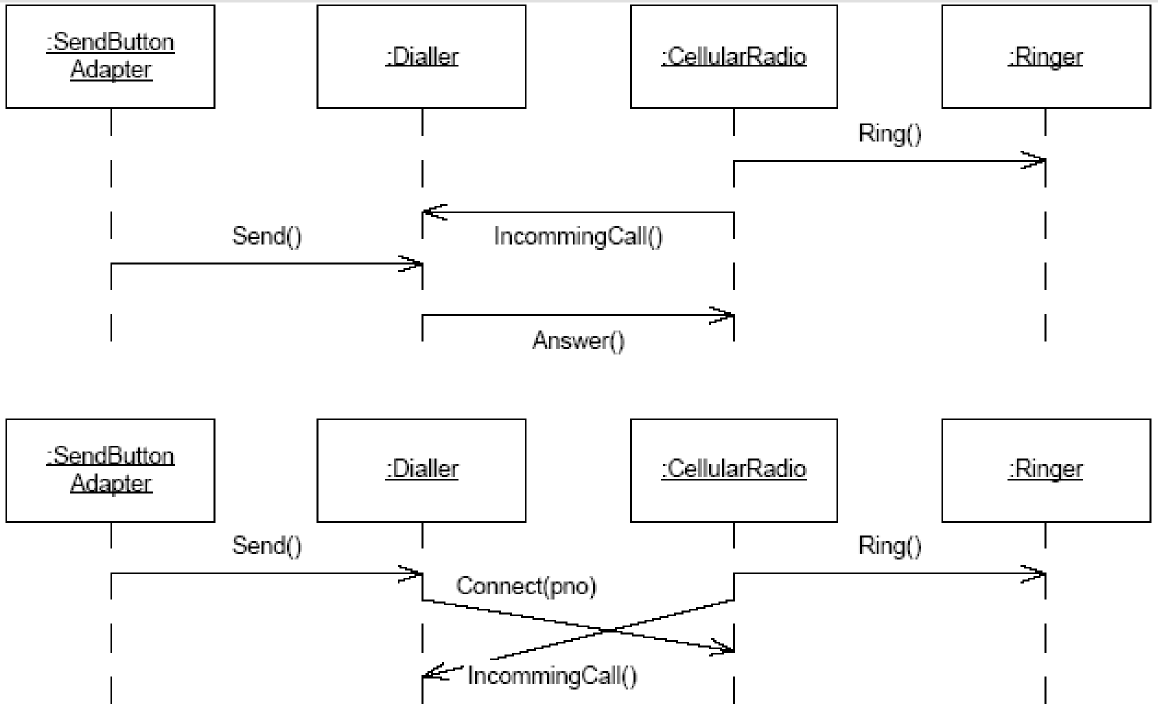
Recursion:



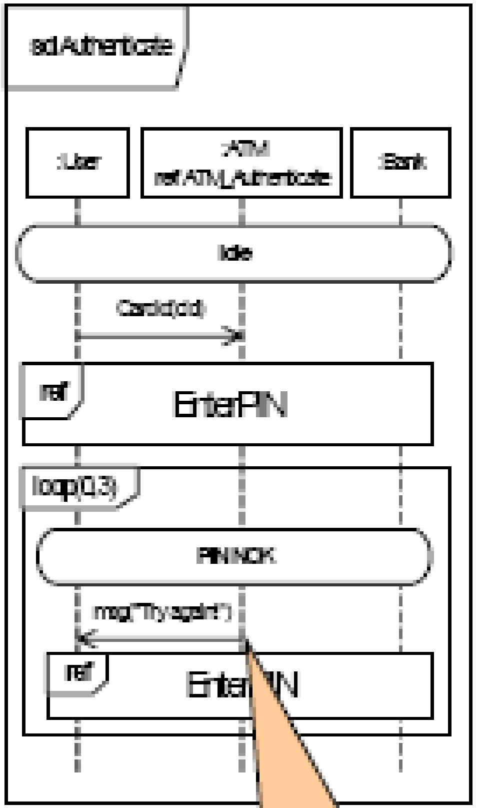
Race conditions

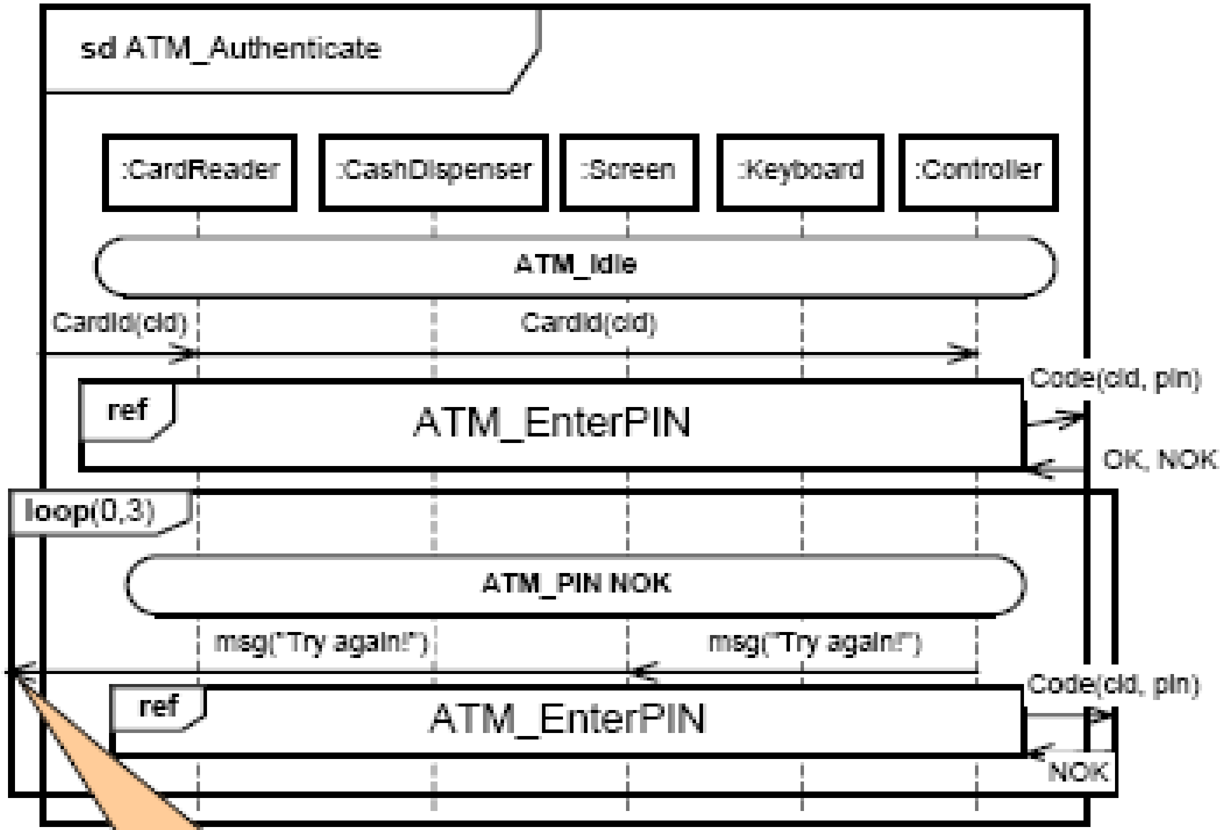
- E.g. an object receives two messages
- Order of arrival changes behaviour
- Only one order leads to correct behaviour
- CellularRadio expects Answer() not Connect(pno)
- Two states when Send can be pressed
- To make outgoing call (after dialling digits)
- To answer incoming call
- State diagram can be useful here
- To help realize there is a race condition
- To specify what should happen
- Angled arrows- Show message delay

- Dialer: gather digits, emit tones
- Cellular Radio: communicate with cellular network
- Button, Speaker, Microphone, Display hardware



Decomposition



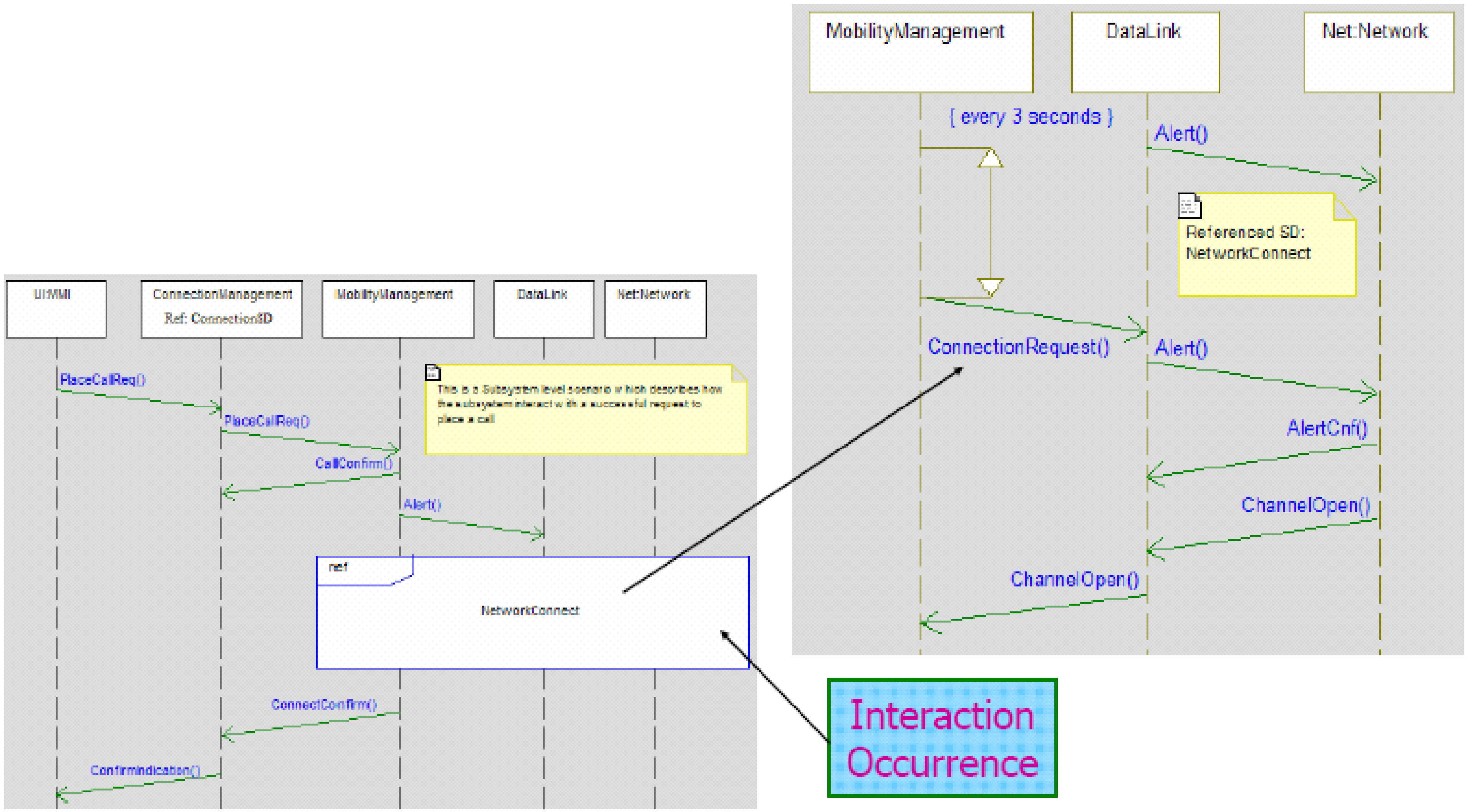


notice the correspondance!

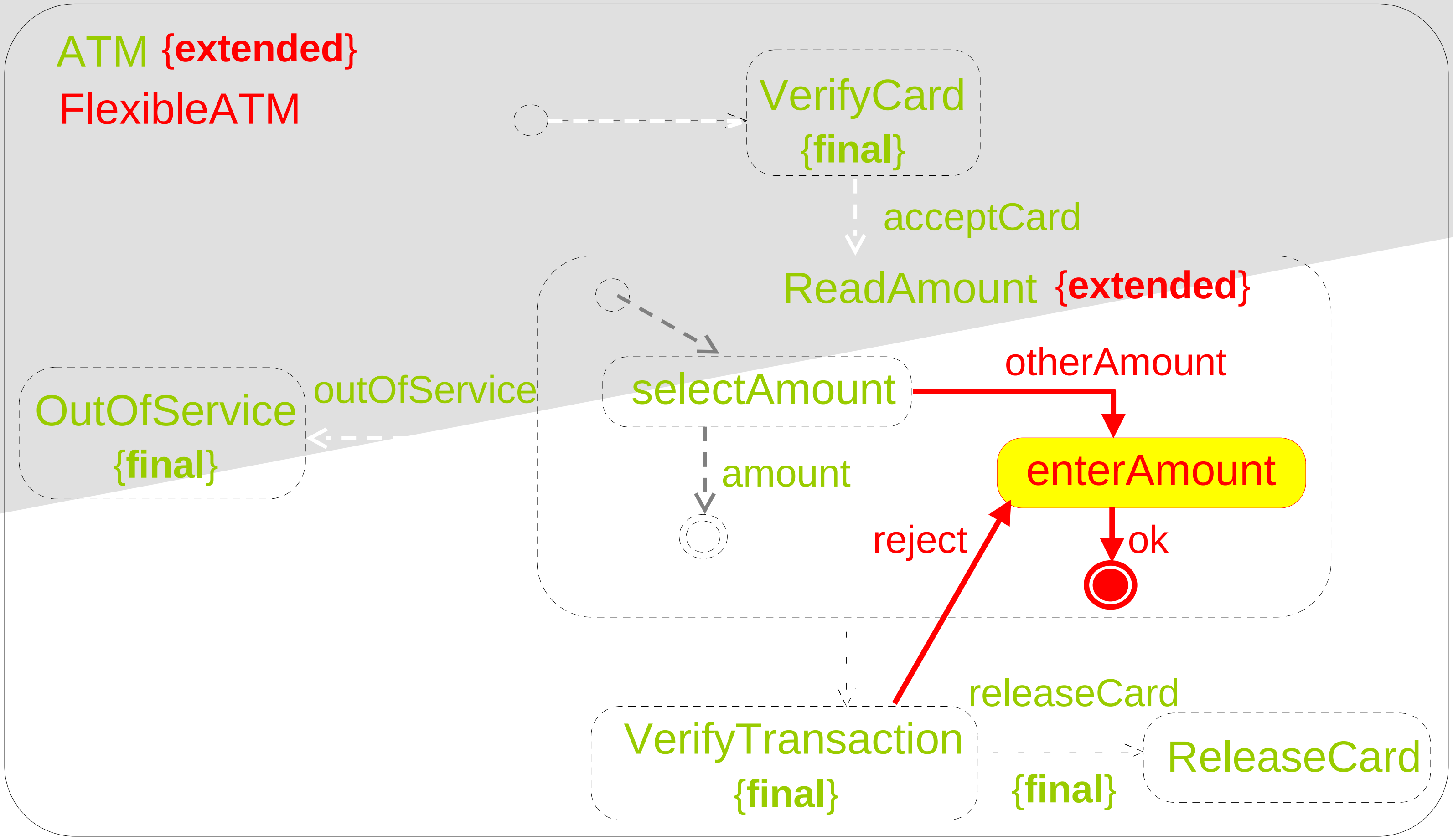
notice the correspondance!

Sequence Diagram - Reference

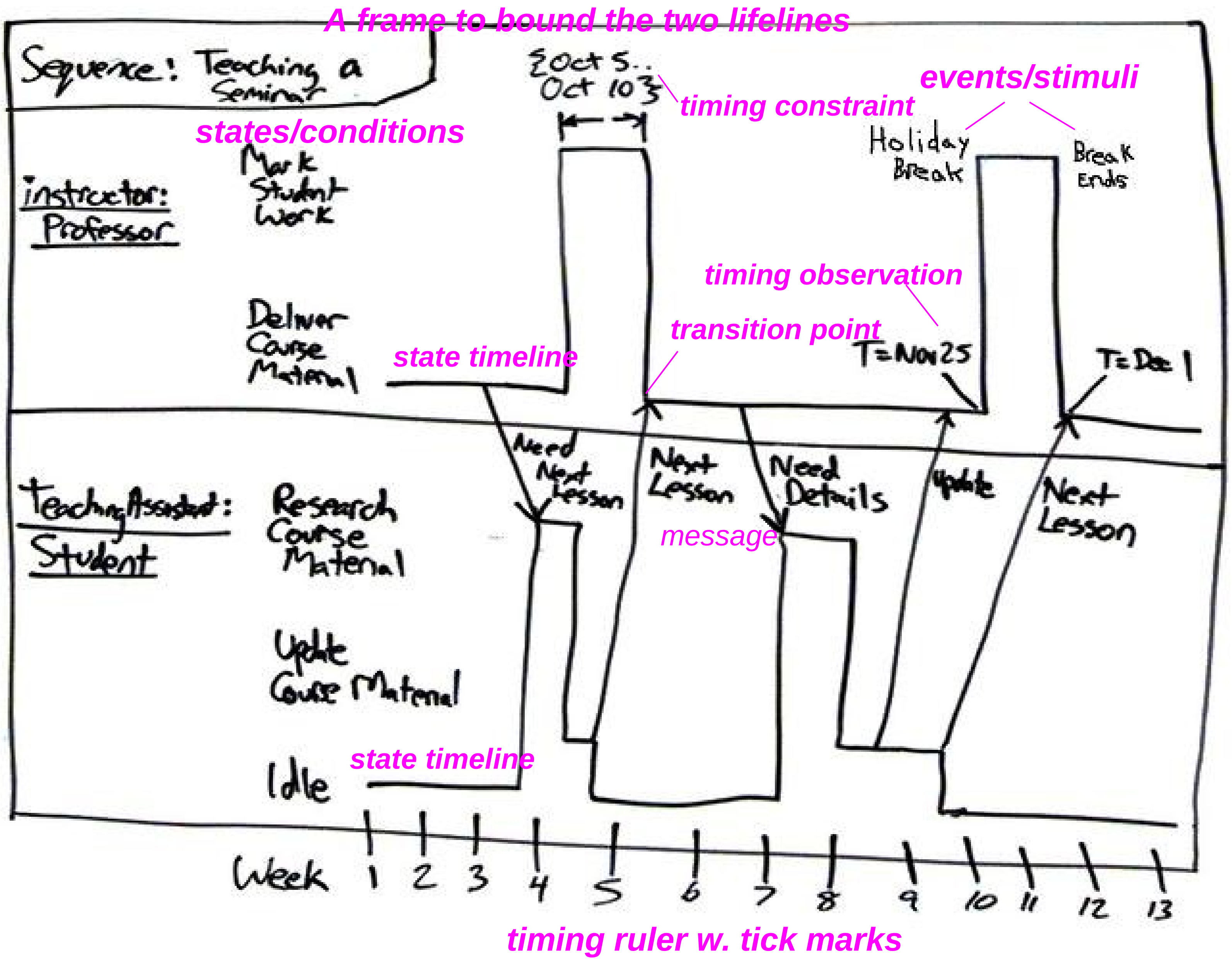
(www.cs.tut.fi/tapahtumat/olio2004/richardson.pdf)



State Machine Redefinition

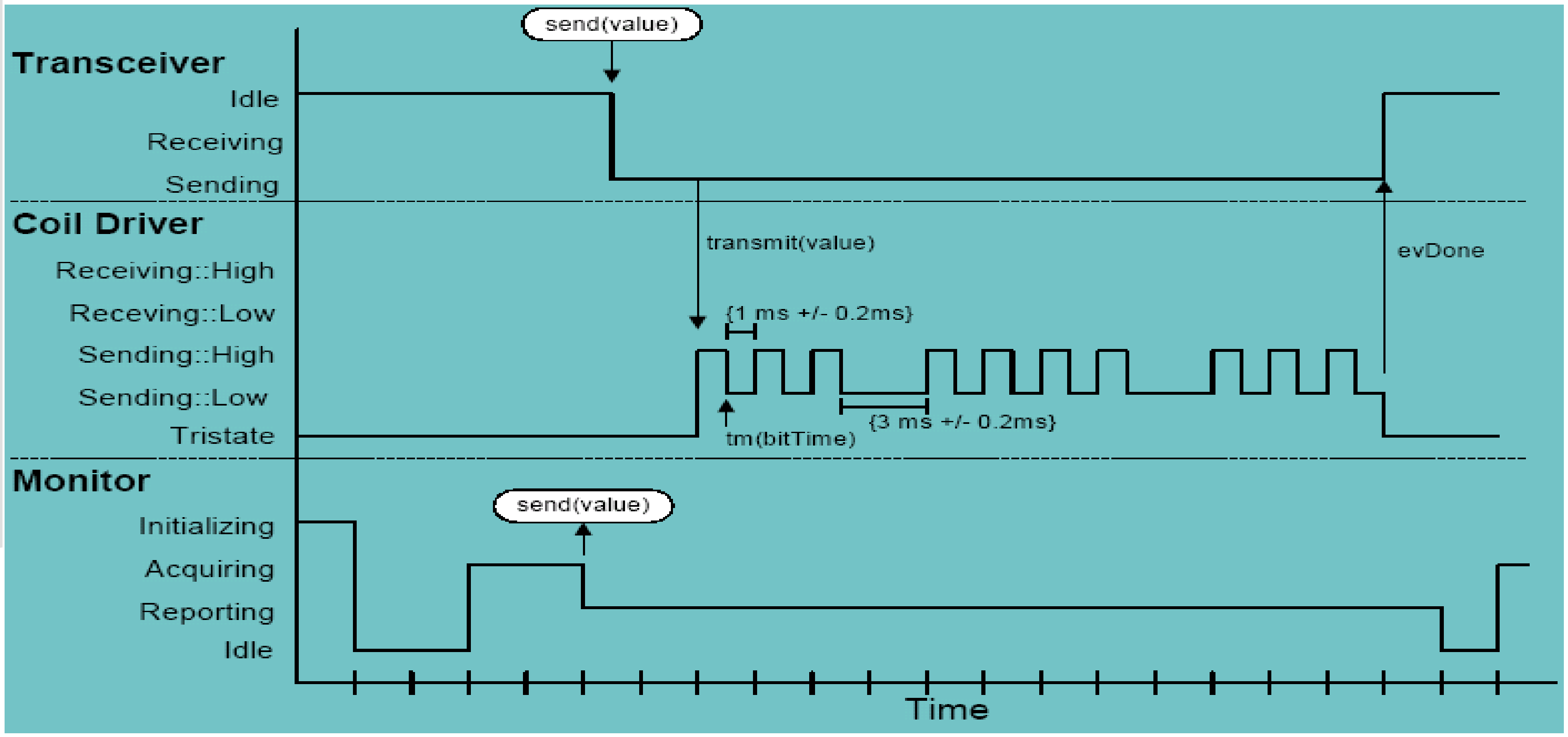


Interaction Diagram: Timing Diagram (robust notation)

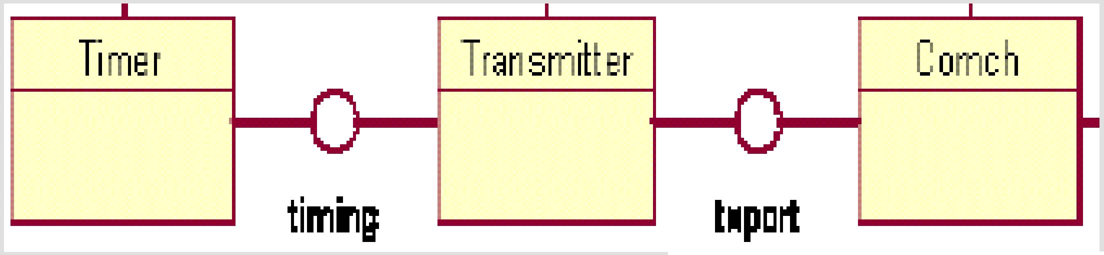


Timing Diagram – another example

(www.cs.tut.fi/tapahtumat/olio2004/richardson.pdf)



Real-Time Extensions: Using CCS



_timing?timeout ^_txport!data0

_timing: interface/connection
?: receive
timeout: event
^: send
_txport: interface/connection
!: send
data0: event
+: multiple receive
":": multiple send

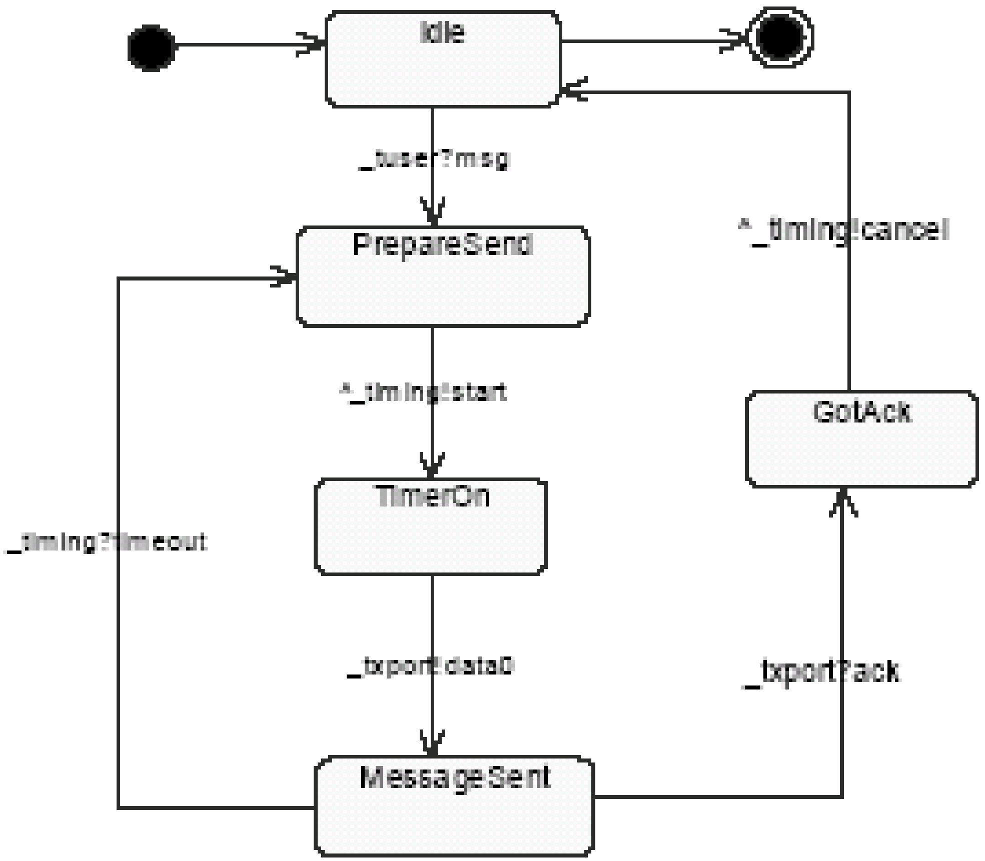


Figure 4: *Normalized* Transmitter Component