zühlke

OO Analysis and Design with UML 2 and UP

Dr. Jim Arlow,

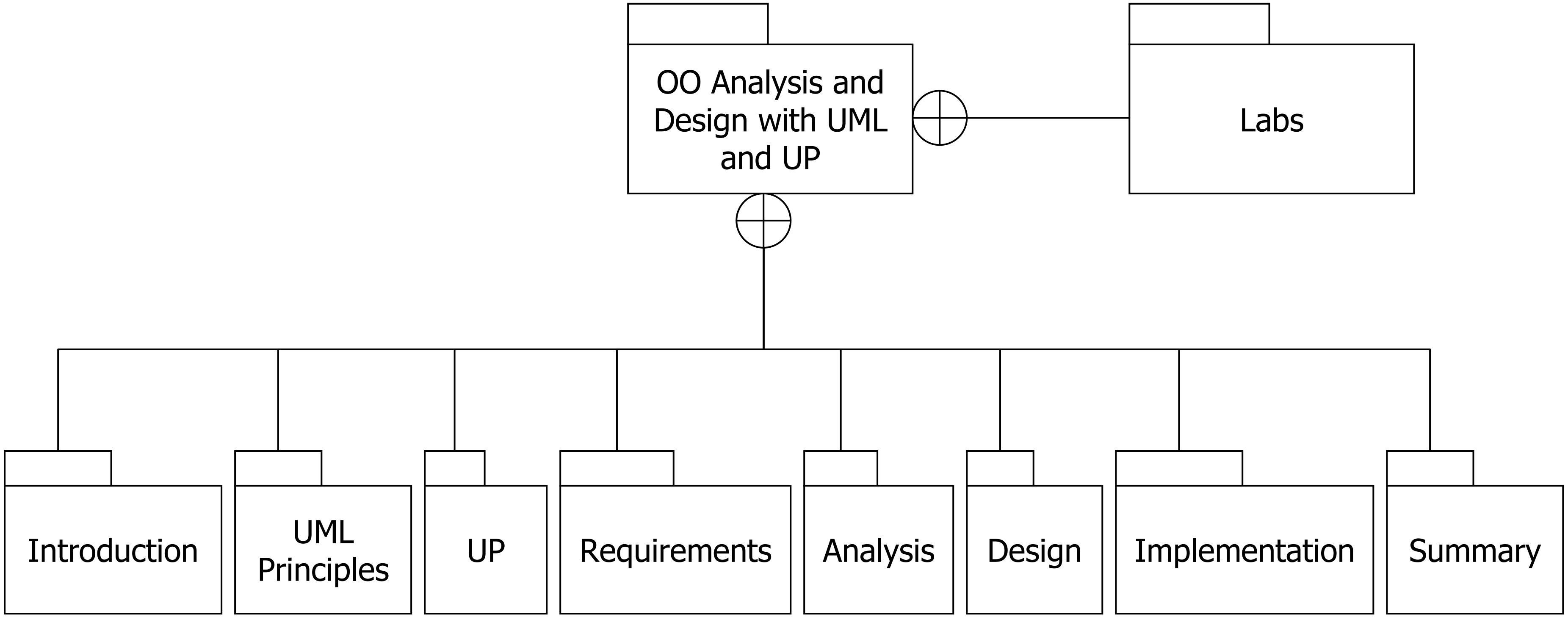
Zuhlke Engineering Limited

Introduction

About you…

- Name?
- Company?
- What are you working on?
- Previous experience of OO?
- Previous experience of modelling?
- One thing you hope to gain from this course?
- Any hobbies or interests?

Structure of this course



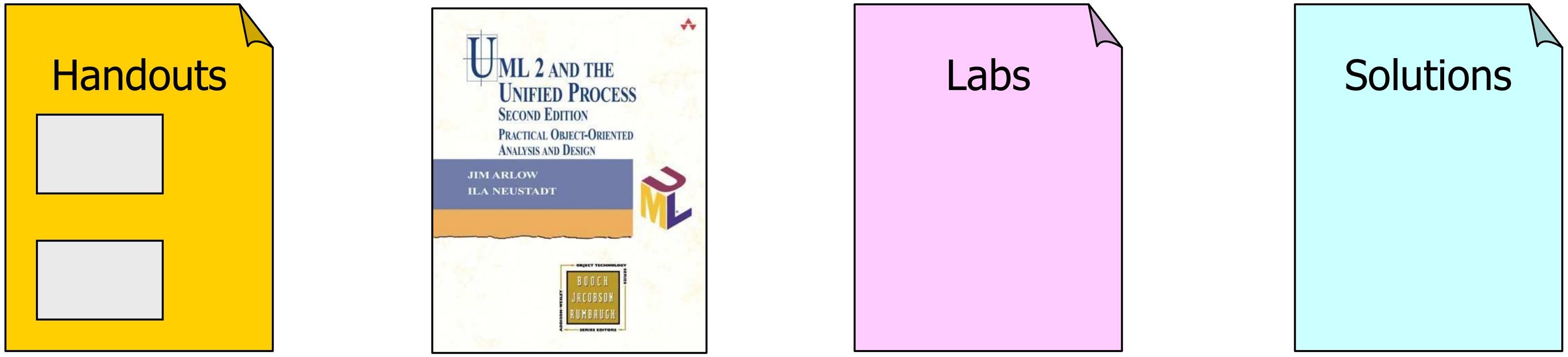
Guiding principles

- This course uses the Unified Software Development Process (UP) to define the activities of OO analysis and design
- The UP is the industry standard software engineering process for the UML

xref

zühlke

Course materials


Handouts
UML 2 and the Unified Process
Second Edition
Practical Object-Oriented Analysis and Design
Jim Arlow
Ila Neustadt
Labs
Solutions

ISBN:0321321278

- For easy reference, all slides in this course are cross referenced to sections in the course book "UML 2 and the Unified Process"
  - There is an example cross reference icon in the top left hand corner of this slide

Labs

- This is a practical course, and there is a lot of laboratory work
- Our approach to this work is cooperative rather than competitive
  - Work together
  - Ask each other for help
  - Share ideas and experience
- Don’t get bogged down!
  - If something brings you to a halt for more than 10 minutes, then ask for help

Goals of the course

- To provide a thorough understanding of OO analysis and design with UML
- To follow the process of OO analysis and design from requirements capture through to implementation using the Unified Software Engineering Process as the framework
- To have fun!

Conditions of satisfaction

- You will know you are succeeding when:
  - You can read and understand UML diagrams
  - You can produce UML models in the laboratory work
  - You apply your knowledge effectively back at your workplace

Questions

- You can ask questions at any time!
- Your participation is always valued

Summary

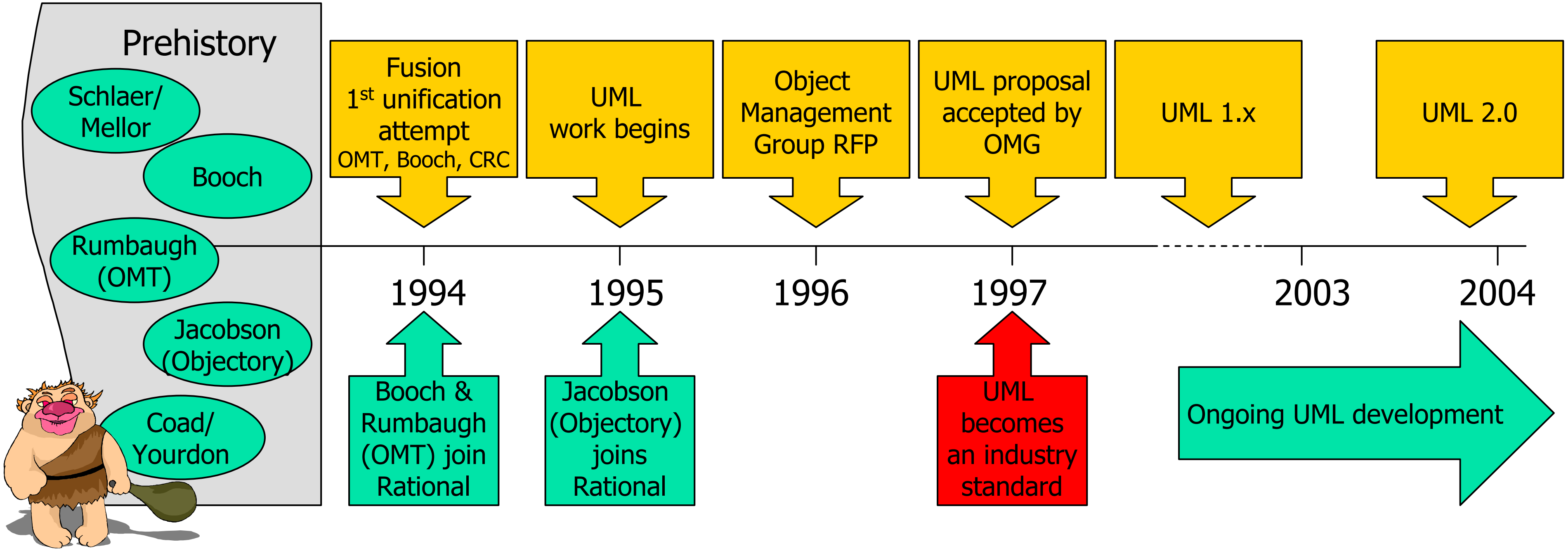
- That’s the end of the introduction so on with the course!

UML principles

What is UML?

- Unified Modelling Language (UML) is a general purpose visual modelling language
  - Can support all existing lifecycles
  - Intended to be supported by CASE tools
- Unifies past modelling techniques and experience
- Incorporates current best practice in software engineering
- UML is *not* a methodology!
  - UML is a visual language
  - UP is a methodology

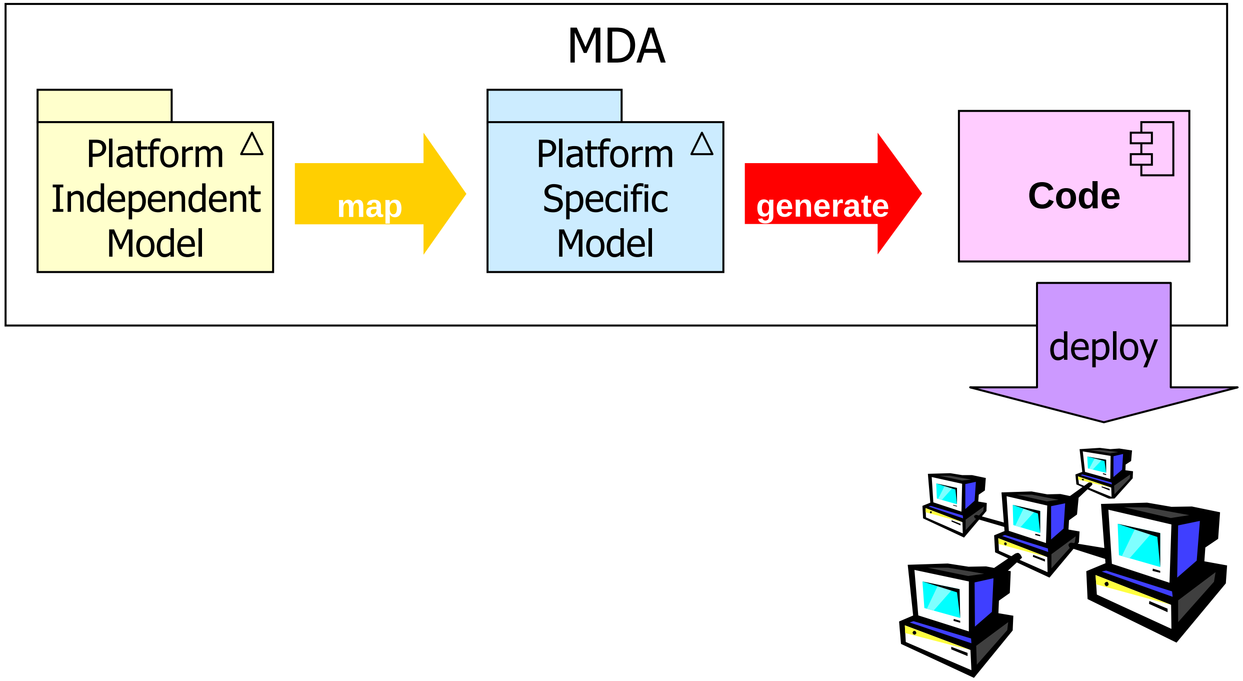
UML history


Prehistory
Schlaer/ Mellor
Booch
Rumbaugh (OMT)
Jacobson (Objectory)
Coad/ Yourdon
Fusion 1st unification attempt OMT, Booch, CRC
UML work begins
Object Management Group RFP
UML proposal accepted by OMG
UML 1.x
UML 2.0
1994
1995
1996
1997
2003
2004
Booch & Rumbaugh (OMT) join Rational
Jacobson (Objectory) joins Rational
UML becomes an industry standard
Ongoing UML development

- A major upgrade to UML at the end of 2003:
  - Greater consistency
  - More precisely defined semantics
  - New diagram types
  - Backwards compatible

UML future?

- The future of development of UML will be increasingly affected by Model Driven Architecture (MDA)



Why "unified"?

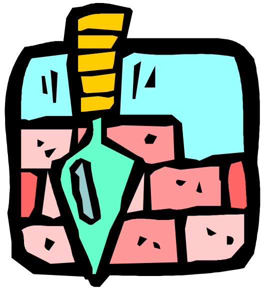
- UML is unified across several domains:
  - Across historical methods and notations
  - Across the development lifecycle
  - Across application domains
  - Across implementation languages and platforms
  - Across development processes
  - Across internal concepts

Objects and the UML

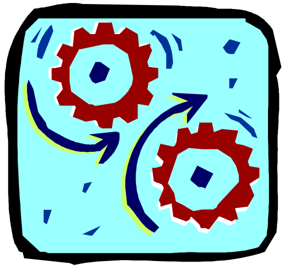
- UML models systems as collections of objects that interact to deliver benefit to outside users
- Static structure
  - What kinds of objects are important
  - What are their relationships
- Dynamic behaviour
  - Lifecycles of objects
  - Object interactions to achieve goals

UML Structure

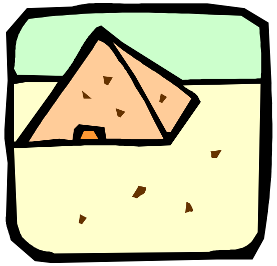
- In this section we present an overview of the structure of the UML
- All the modelling elements mentioned here are discussed later, and in much more detail!



Building blocks



Common mechanisms



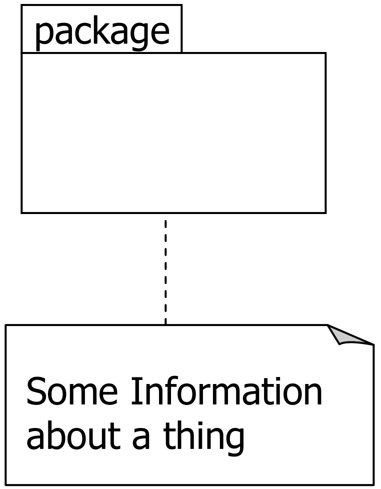
Architecture

UML building blocks

- Things
  - Modelling elements
- Relationships
  - Tie things together
- Diagrams
  - Views showing interesting collections of things
  - Are views of the model

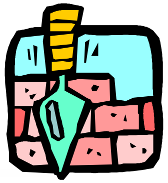
Things

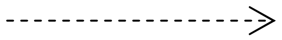
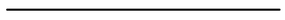
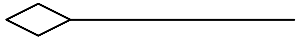
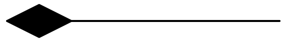
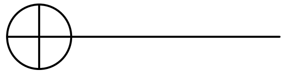
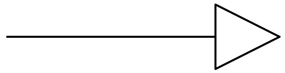
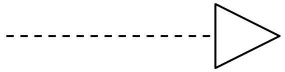
- Structural things – nouns of a UML model
  - Class, interface, collaboration, use case, active class, component, node
- Behavioural things – verbs of a UML model
  - Interactions, state machine
- Grouping things
  - Package
    - Models, frameworks, subsystems
- Annotational things
  - Notes
  - Tagged values



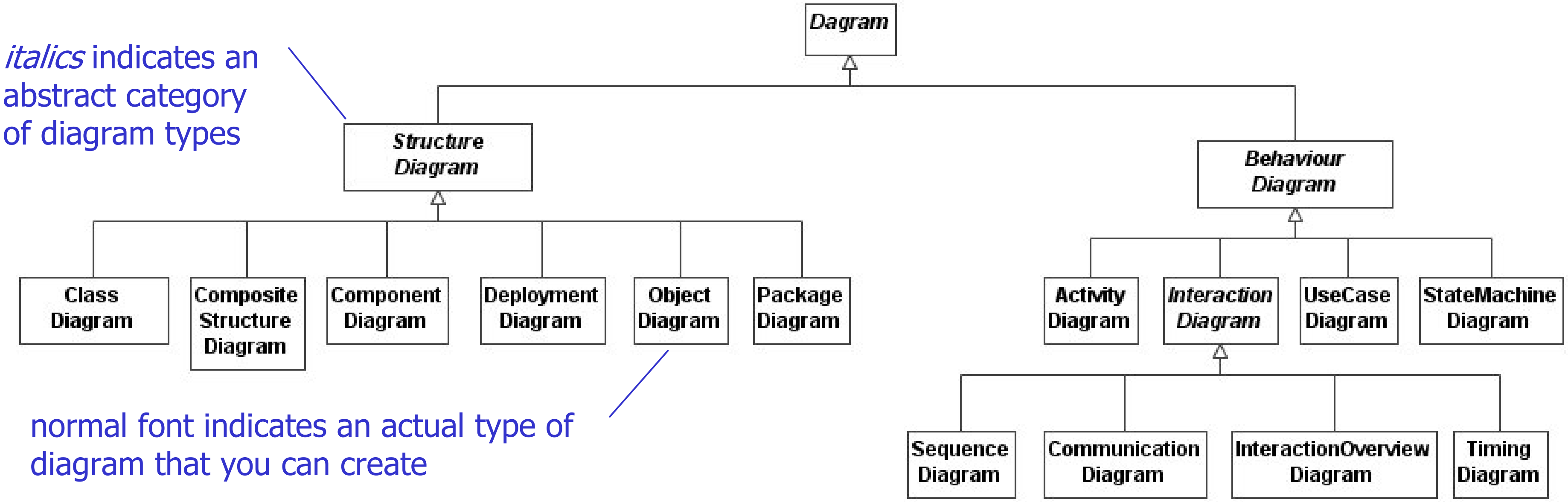
1.8.2

Relationships



relationship	UML syntax	brief semantics
dependency		The source element depends on the target element and may be affected by changes to it.
association		The description of a set of links between objects.
aggregation		The target element is a part of the source element.
composition		A strong (more constrained) form of aggregation.
containment		The source element contains the target element.
generalization		The source element is a specialization of the more general target element and may be substituted for it.
realization		The source element guarantees to carry out the contract specified by the target element

UML has 13 types of diagram

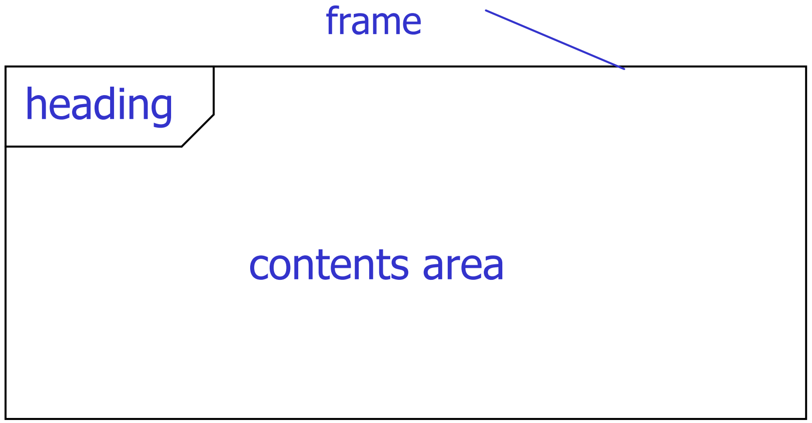


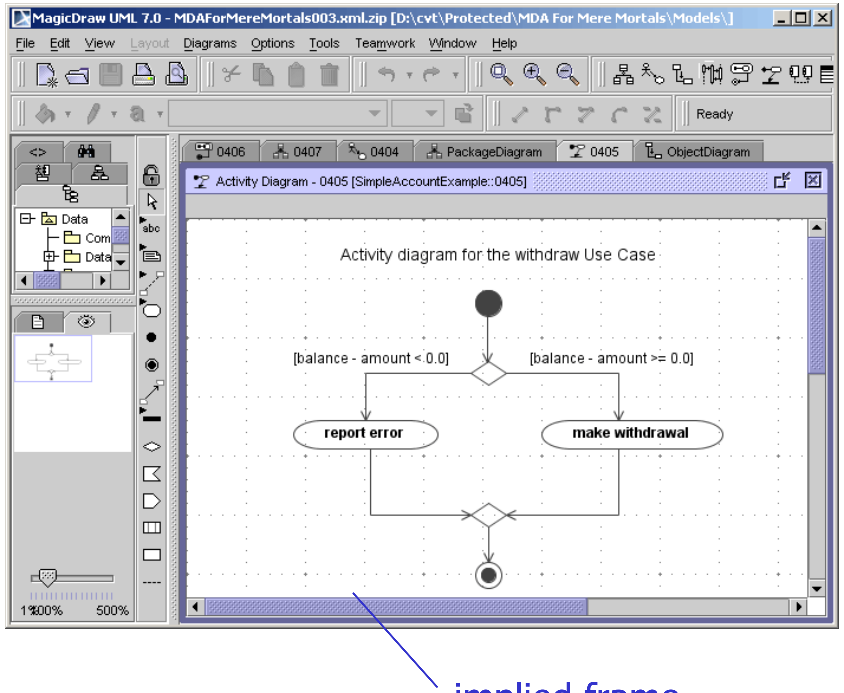
italics indicates an abstract category of diagram types

normal font indicates an actual type of diagram that you can create

- Structure diagrams model the structure of the system (the static model)
- Behavior diagrams model the dynamic behavior of the system (the dynamic model)
- Each type of diagram gives a different type of view of the model

UML 2 diagram syntax




MagicDraw UML 7.0 - MDAForMereMortals003.xml.zip [D:\cvt\Protected\MDA For Mere Mortals\Models\]
File Edit View Layout Diagrams Options Tools Teamwork Window Help
Ready
0406
0407
0404
PackageDiagram
0405
ObjectDiagram
Activity Diagram - 0405 [SimpleAccountExample::0405]
Data
Com
Data
100%
500%
Activity diagram for the withdraw Use Case
[balance - amount < 0.0]
[balance - amount >= 0.0]
report error
make withdrawal

implied frame

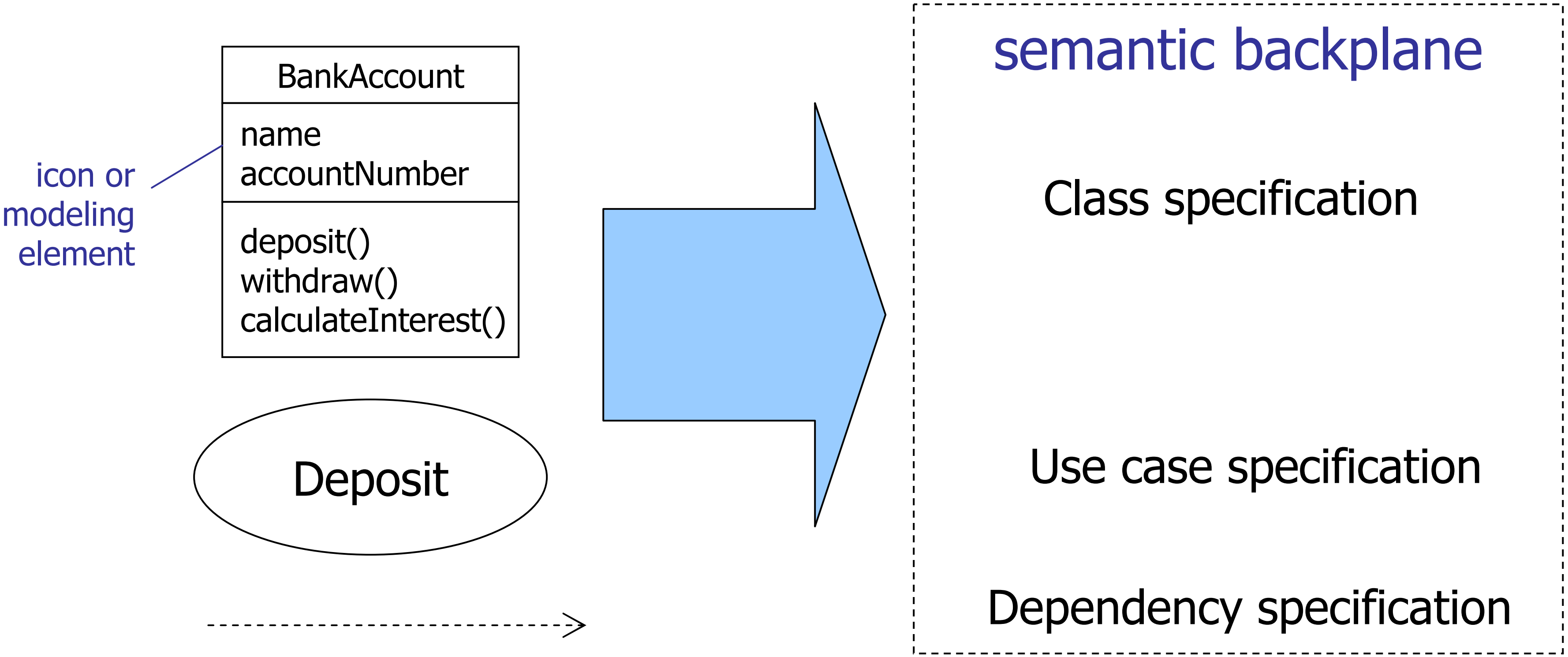
heading syntax: <kind> <name> <parameters>
N.B. <kind> and <parameters> are optional

- The heading specifies the kind of diagram, it's name and any information (parameters) needed by elements in the diagram
- The frame may be implied by a diagram area in the UML tool

UML common mechanisms

- UML has four common mechanisms that apply consistently throughout the language:
  - Specifications
  - Adornments
  - Common divisions
  - Extensibility mechanisms

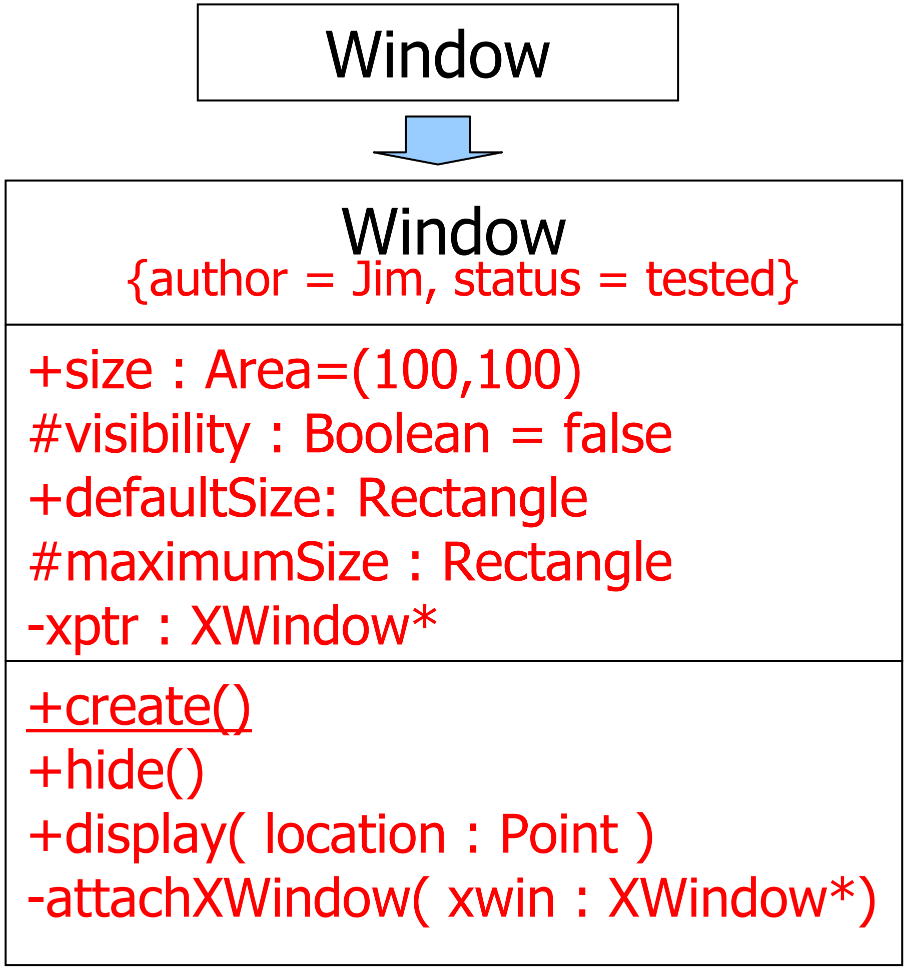
Specifications



- Behind every UML modelling element is a *specification* which provides a textual statement of the syntax and semantics of that element
- These specifications form the *semantic backplane* of the model

Adornments

- Every UML modelling element starts with a basic symbol to which can be added a number of *adornments* specific to that symbol
- We only show adornments to *increase the clarity* of the diagram or to highlight a specific feature of the model



Window

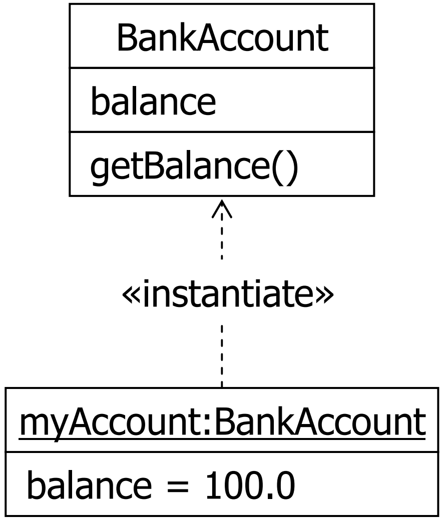
Window
{author = Jim, status = tested}

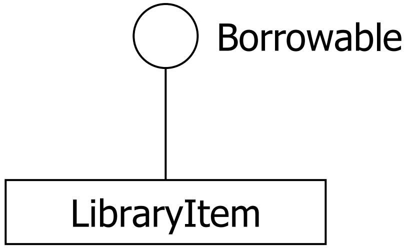
+size : Area=(100,100)
#visibility : Boolean = false
+defaultSize: Rectangle
#maximumSize : Rectangle
-xptr : XWindow*

+create()
+hide()
+display(location : Point)
-attachXWindow(xwin : XWindow*)

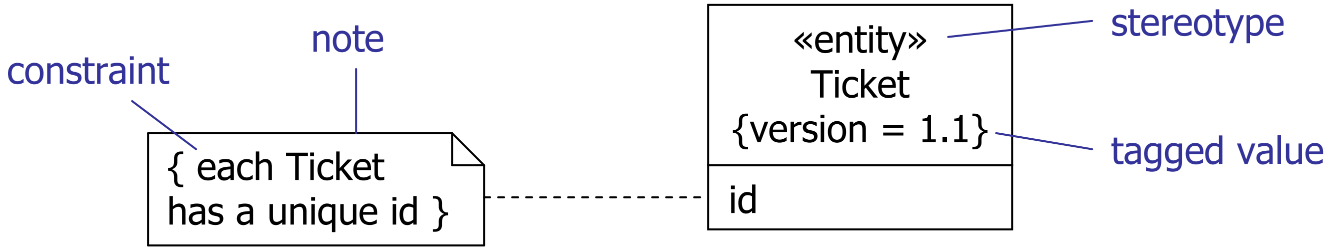
Common divisions

- Classifier and instance
  - A classifier is an abstraction, an instance is a concrete manifestation of that abstraction
  - The most common form is class/object e.g. a classifier might be a BankAccount class, and an instance might be an object representing my bank account
  - Generally instances have the same notation as classes, but the instance name is underlined
- Interface and implementation
  - An interface declares a contract and an implementation represents a concrete realization of that contract





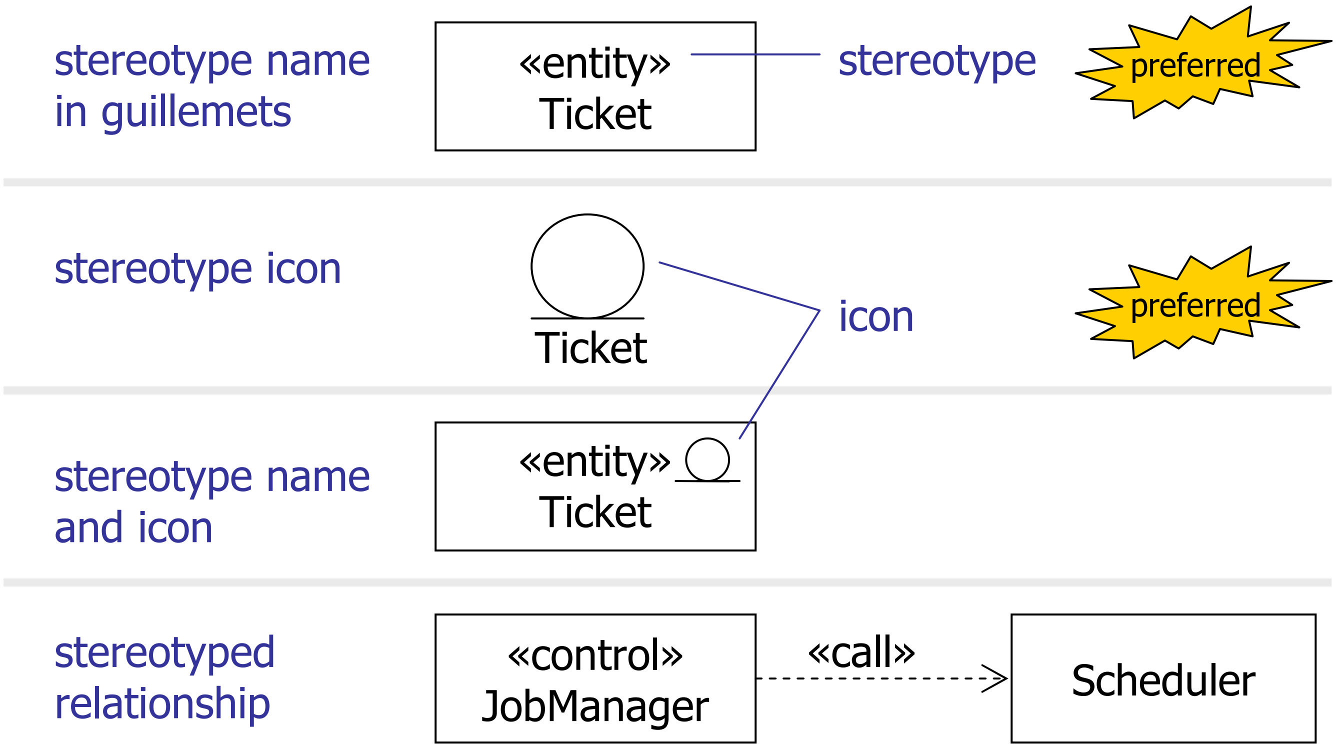
Extensibility mechanisms



- Stereotypes
  - A stereotype allows us to define a new UML modelling element based on an existing one
  - We define the semantics of the stereotype ourselves
  - Stereotypes add new elements to the UML metamodel
  - Written as «stereotypeName»
- Constraints
  - Extends the semantics of an element by allowing us to add new rules about the element
  - Written as { some constraint }
- Tagged values
  - Allows us to add new, ad-hoc information to an element's specification
  - Written as { tag1 = value1, tag2 = value2 … }

are attached to a stereotype

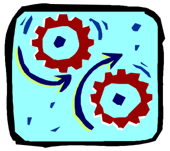
Stereotype syntax options



- A stereotype introduces a *new* modelling element and so we must *always* define semantics for our stereotypes
- Each model element can have many stereotypes

1.9.4.4

UML profiles



- A profile customizes UML for a specific purposes
- A UML profile is a collection of stereotypes, tagged values and constraints
  - The tagged values and constraints are associated with stereotypes
- Stereotypes extend one of the UML meta-model elements (e.g. Class, Association)
  - Any element that gets the stereotype also gets the associated tagged values and constraints

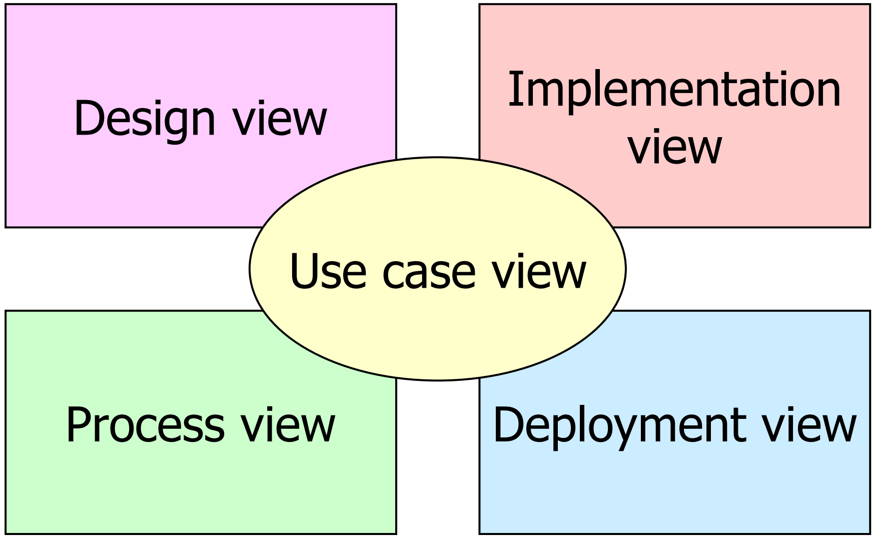
Architecture

- "The organisational structure of a software system"
  - UML specification & IEEE Std. 610.12-1990
  - RUP has a 4+1 view of architecture

vocabulary
functionality

behaviour

performance
scalability
throughput



system assembly
configuration
management

The 4+1 View of Architecture, Philippe Kruchten, IEEE Software, 12(6) Nov. 1995, p. 45-50

system topology
distribution
delivery
installation

Summary

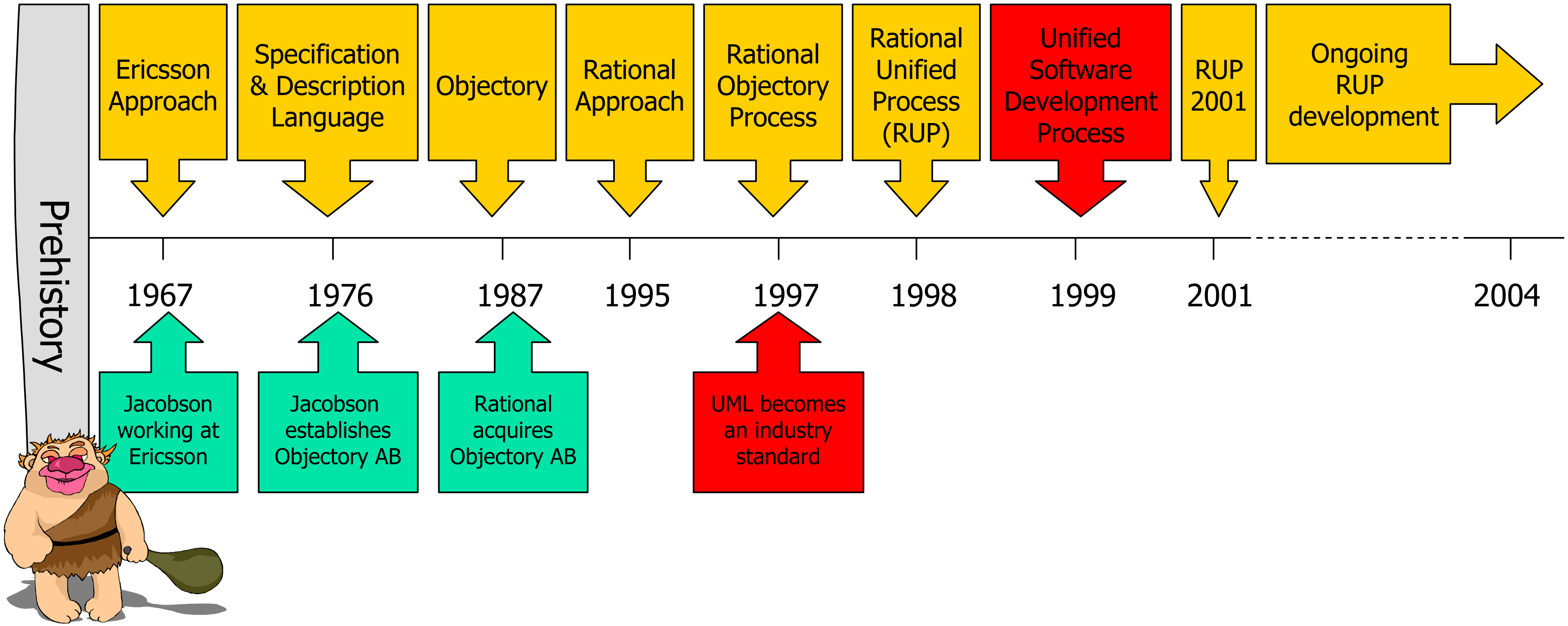
- The UML is composed of *building blocks:*
  - Things
  - Relationships
  - Diagrams
- The UML has four *common mechanisms:*
  - Specifications
  - Adornments
  - Common divisions
  - Extensibility mechanisms
- The UML is based on a 4+1 view of *system architecture*

Introduction to the Unified Process

The Unified Process (UP)

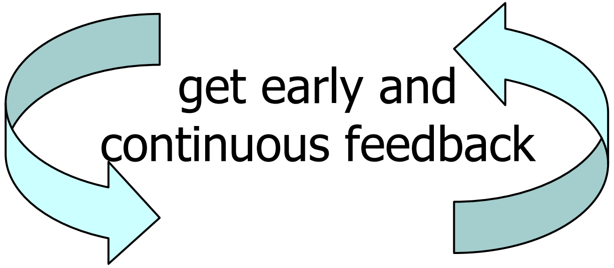
- The Unified Software Development Process is an industry standard software engineering process
  - It is commonly referred to as the "Unified Process" or UP
  - It is the generic process for the UML
  - It is free - described in "The Unified Software Development Process", ISBN:0201571692"
- UP is:
  - Use case (requirements) driven
  - Risk driven
  - Architecture centric
  - Iterative and incremental
- UP is a generic software engineering process. It has to be customised (instantiated) for your project
  - In house standards, document templates, tools, databases, lifecycle modifications, …
- Rational Unified Process (RUP) is an instantiation of UP
  - RUP is a product marketed and owned by Rational Corporation
  - RUP also has to be instantiated for your project

UP history


Prehistory
Ericsson Approach
Specification & Description Language
Objectory
Rational Approach
Rational Objectory Process
Rational Unified Process (RUP)
Unified Software Development Process
RUP 2001
Ongoing RUP development
1967
1976
1987
1995
1997
1998
1999
2001
2004
Jacobson working at Ericsson
Jacobson establishes Objectory AB
Rational acquires Objectory AB
UML becomes an industry standard

Iterations

- Iterations are the key to the UP
- Each iteration is like a mini-project including:
  - Planning
  - Analysis and design
  - Integration and test
  - An internal or external release

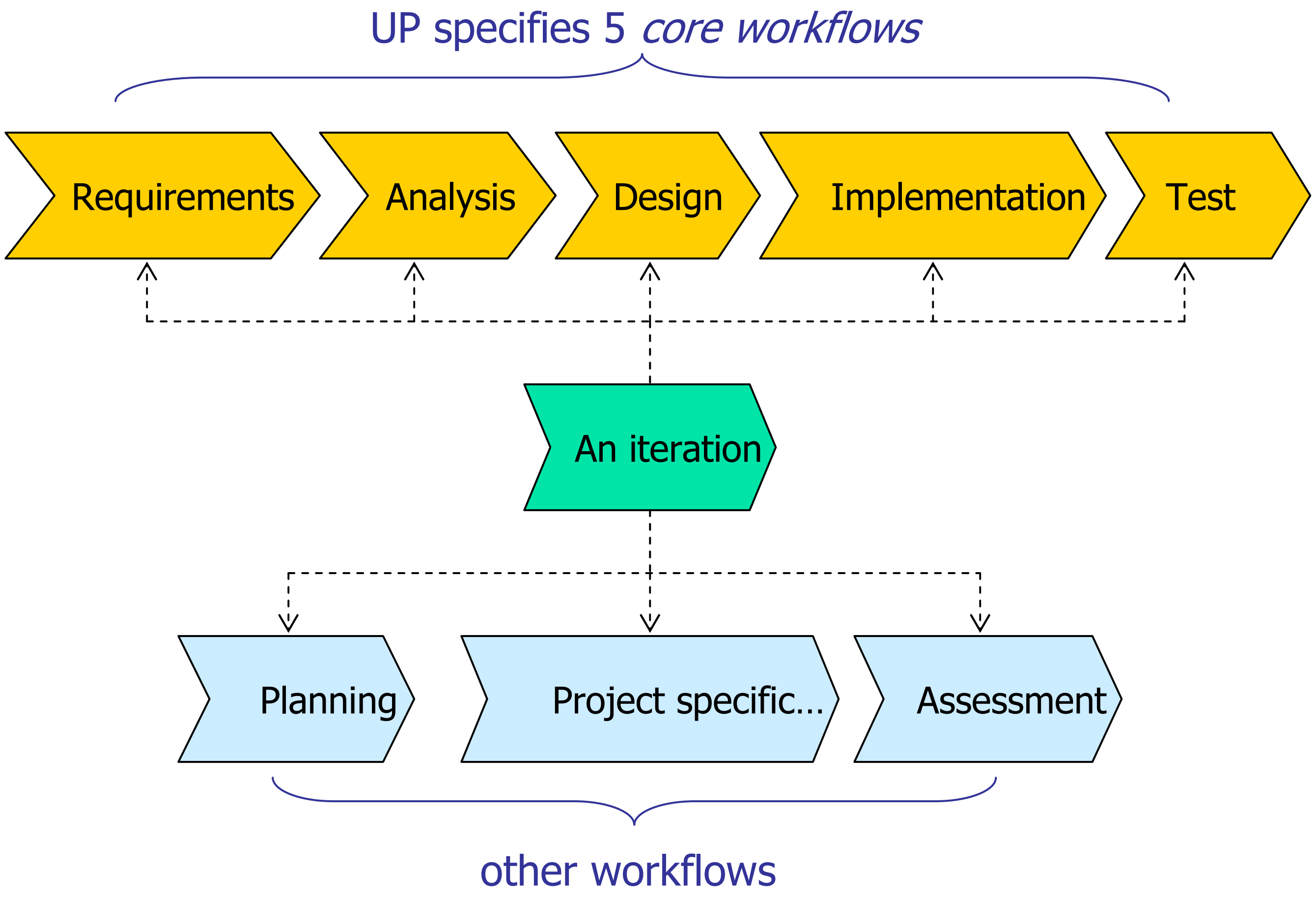


get early and continuous feedback

- We arrive at a final product release through a sequence of iterations
- Iterations can overlap - this allows parallel development and flexible working in large teams
  - Requires careful planning
- Iterations are organised into phases

Iteration workflows

- Each iteration may contain all of the core workflows but with a different emphasis depending on where the iteration is in the lifecycle

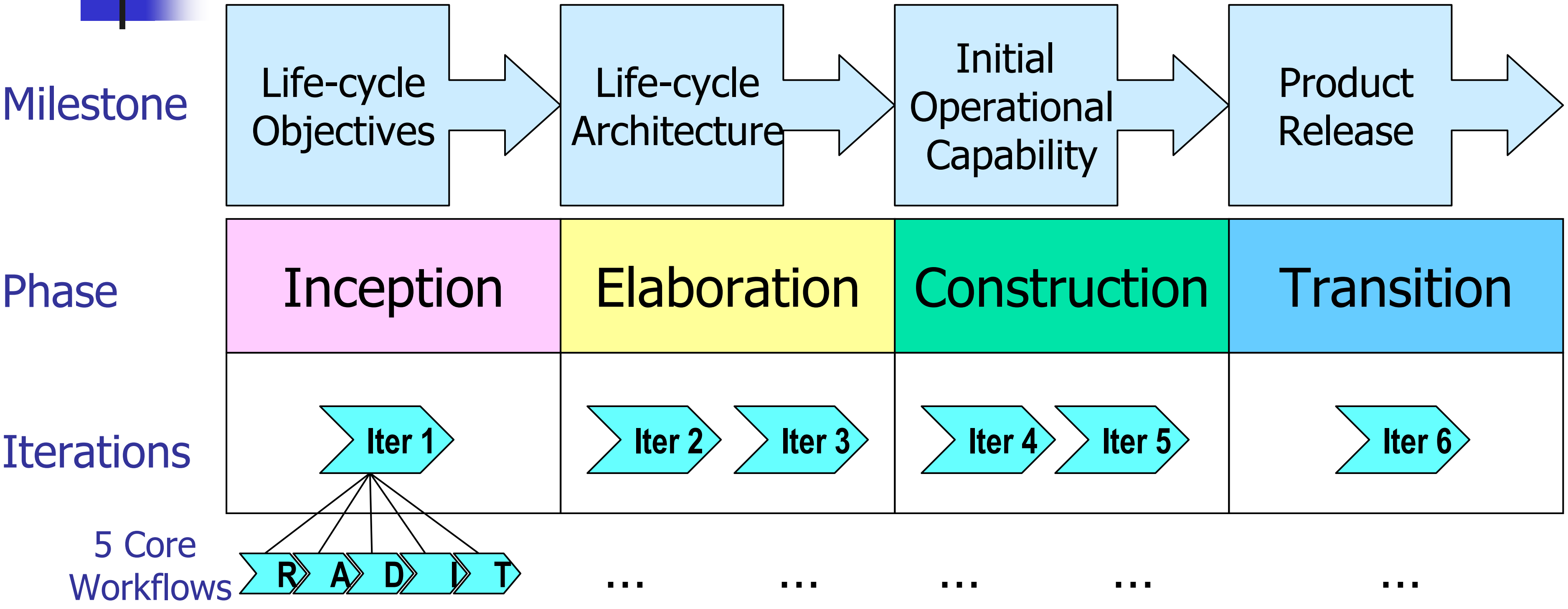


Baselines and increments

- Each iteration generates a baseline
- A baseline is a set of reviewed and approved artefacts that:
  - Provide an agreed basis for further review and development
  - Can be changed only through formal procedures such as configuration and change management
- An *increment* is the difference between the baseline generated by one iteration and the baseline generated by the next iteration
  - This is why the UP is called "iterative and incremental"

zühlke

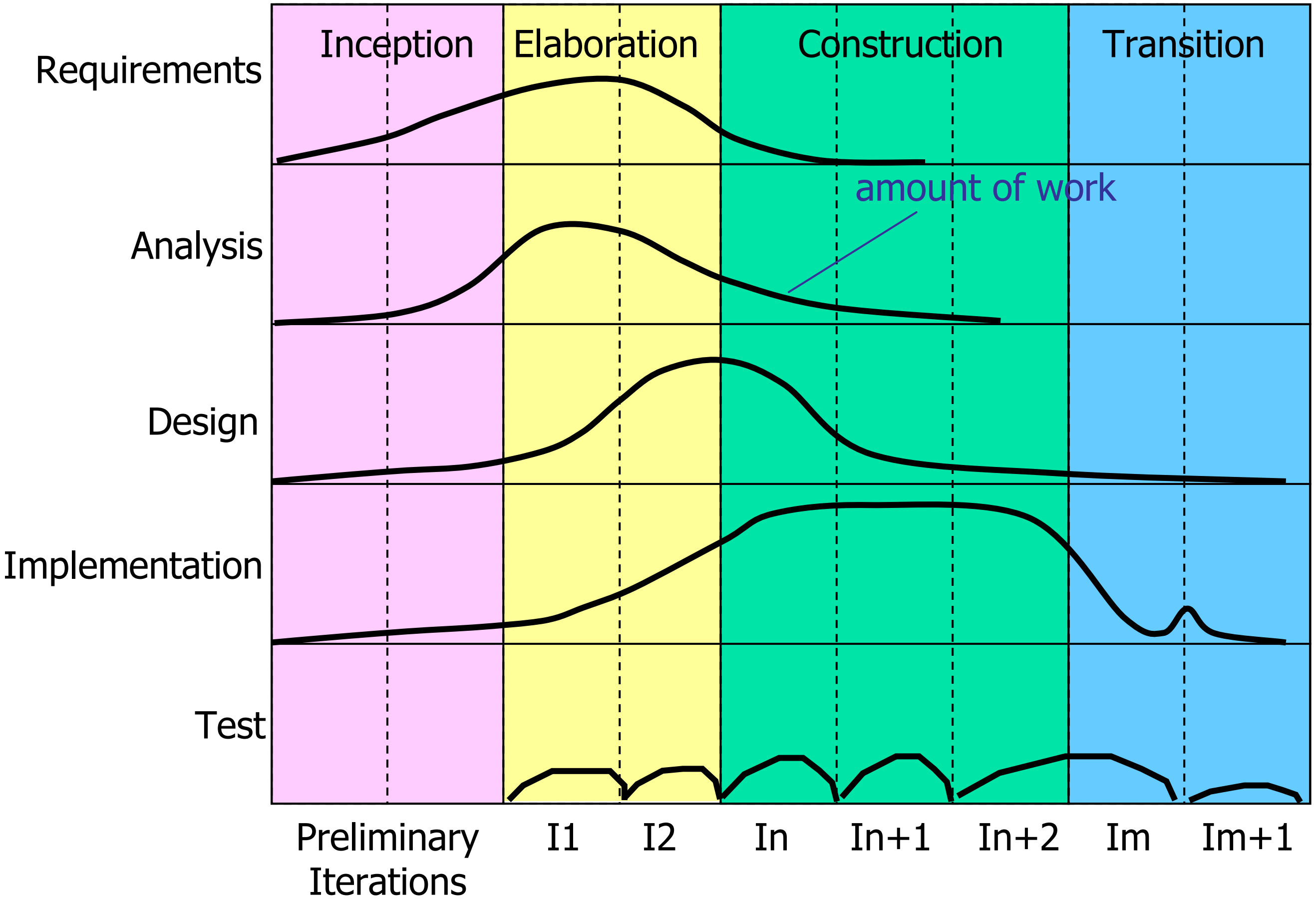
UP Structure



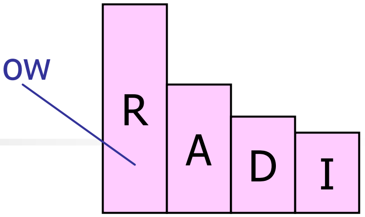
- Each phase can include several iterations
  - The exact number of iterations per phase depends on the size of the project! e.g. one iteration per phase for small projects
- Each phase concludes with a major milestone

Phases and Workflows

- This figure is the key to understanding UP!
- For each phase we will consider:
  - The focus in terms of the core workflows
  - The goal for the phase
  - The milestone at the end of the phase

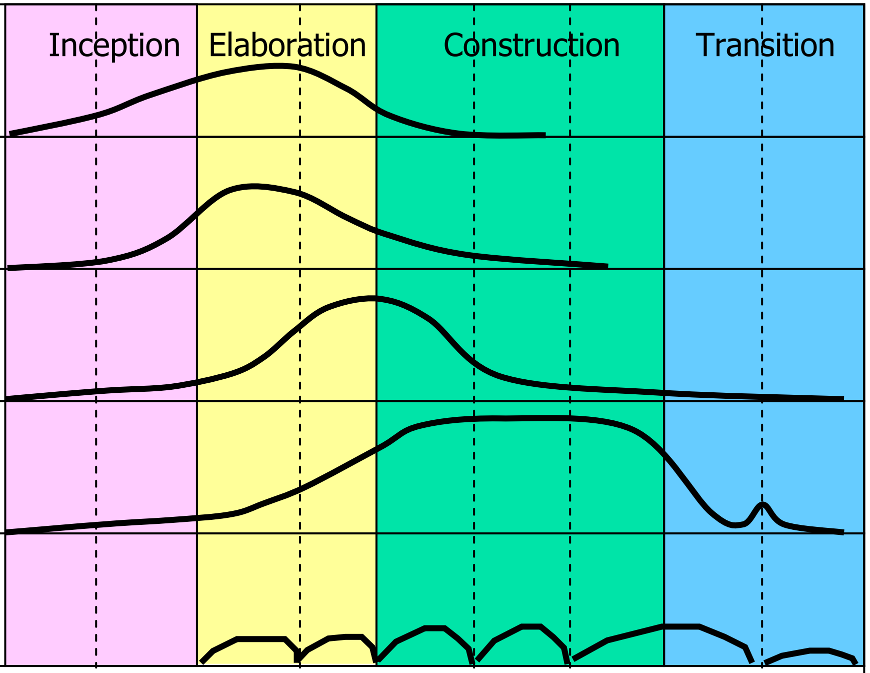


Inception



amount of work in each core workflow

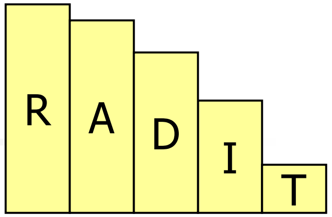
Focus	**Requirements** – establish business case and scope. Capture core requirements	
	Analysis – establish feasibility	
	Design – design proof of concept or technical prototypes	
	Implementation – build proof of concept or technical prototype	
	Test – not generally applicable	
Goals	Establish feasibility of the project - create proof of concept/technical prototypes Create a business case Scope the system - capture key requirements Identify critical risks	



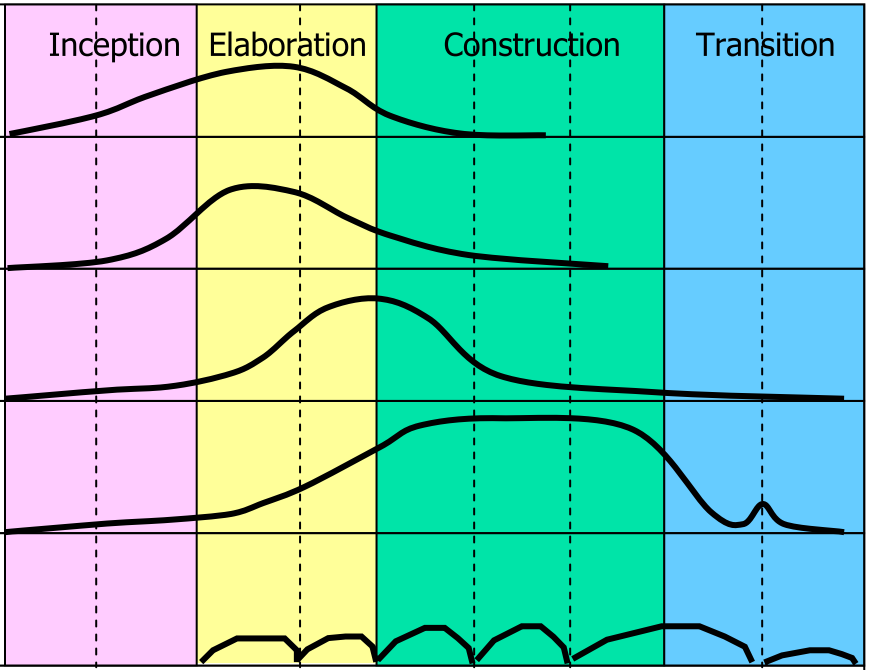
Inception - milestone

- Life Cycle Objectives - conditions of satisfaction:
  - System scope has been defined
  - Key requirements for the system have been captured. These have been defined and agreed with the stakeholders
  - An architectural vision exists. This is just a sketch at this stage
  - A Risk Assessment
  - A Business Case
  - Project feasibility is confirmed
  - The stakeholders agree on the objectives of the project

Elaboration



Focus	**Requirements** – refine system scope and requirements
	Analysis – establish what to build
	Design – create a stable architectural baseline
	Implementation – build the architectural baseline
	Test – test the architectural baseline
Goals	Create an executable architectural baseline Refine Risk Assessment and define quality attributes (defect rates etc.) Capture use cases to 80% of the functional requirements Create a plan with sufficient detail for the construction phase Formulate a bid which includes resources, time, equipment, staff, cost

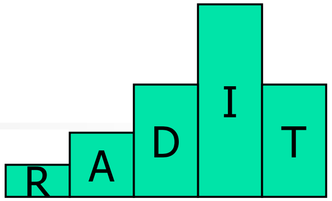


Elaboration - milestone

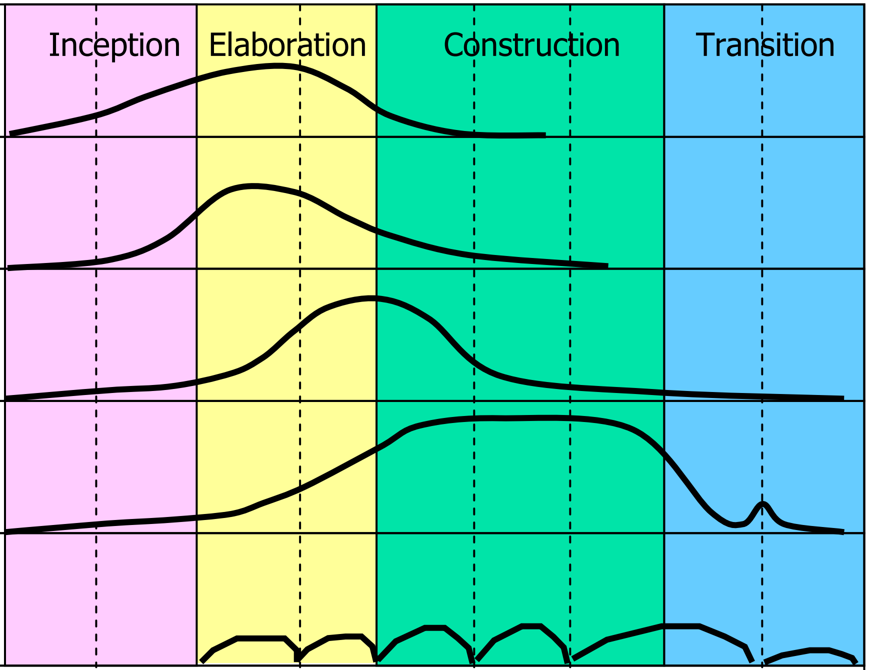
- Lifecycle Architecture - conditions of satisfaction:
    - A resilient, robust executable architectural baseline has been created
    - The Risk Assessment has been updated
    - A project plan has been created to enable a realistic bid to be formulated
    - The business case has been verified against the plan
    - The stakeholders agree to continue

2.9.7

Construction



Focus	**Requirements** – uncover any requirements that had been missed
	Analysis – finish the analysis model
	Design – finish the design model
	Implementation – build the Initial Operational Capability
	Test – test the Initial Operational Capability
Goals	Complete use case identification, description and realization Finish analysis, design, implementation and test Maintain the integrity of the system architecture Revise the Risk Assessment



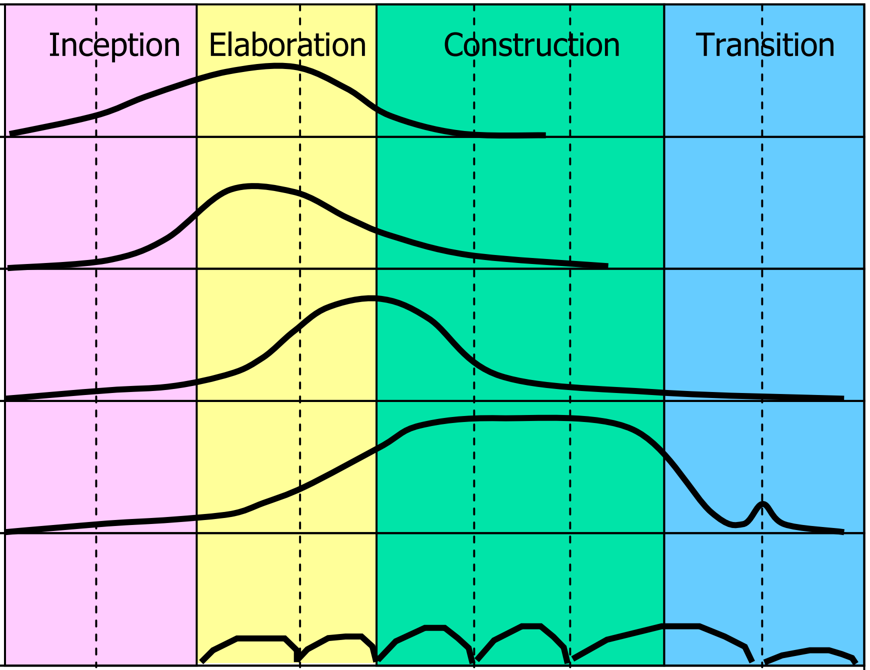
Construction - milestone

- Initial Operational Capability - conditions of satisfaction:
  - The product is ready for beta testing in the user environment

Transition

D I T

Focus	**Requirements** – not applicable
	Analysis – not applicable
	Design – modify the design if problems emerge in beta testing
	Implementation – tailor the software for the user site. Fix bugs uncovered in beta testing
	Test – perform beta testing and acceptance testing at the user site
Goals	Correct defects Prepare the user site for the new software and tailor the software to operate at the user site Modify software if unforeseen problems arise Create user manuals and other documentation Provide customer consultancy Conduct post project review



Transition – milestone

- Product Release - conditions of satisfaction:
  - Beta testing, acceptance testing and defect repair are finished
  - The product is released into the user community

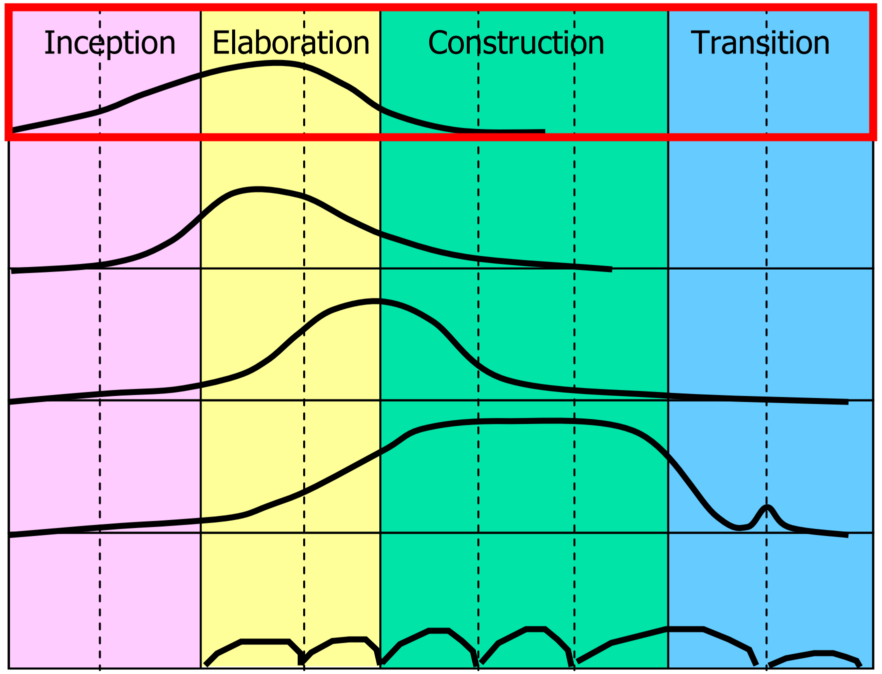
Summary

- UP is a risk and use case driven, architecture centric, iterative and incremental software development process
- UP has four phases:
  - Inception
  - Construction
  - Elaboration
  - Transition
- Each iteration has five core workflows:
  - Requirements
  - Analysis
  - Design
  - Implementation
  - Test

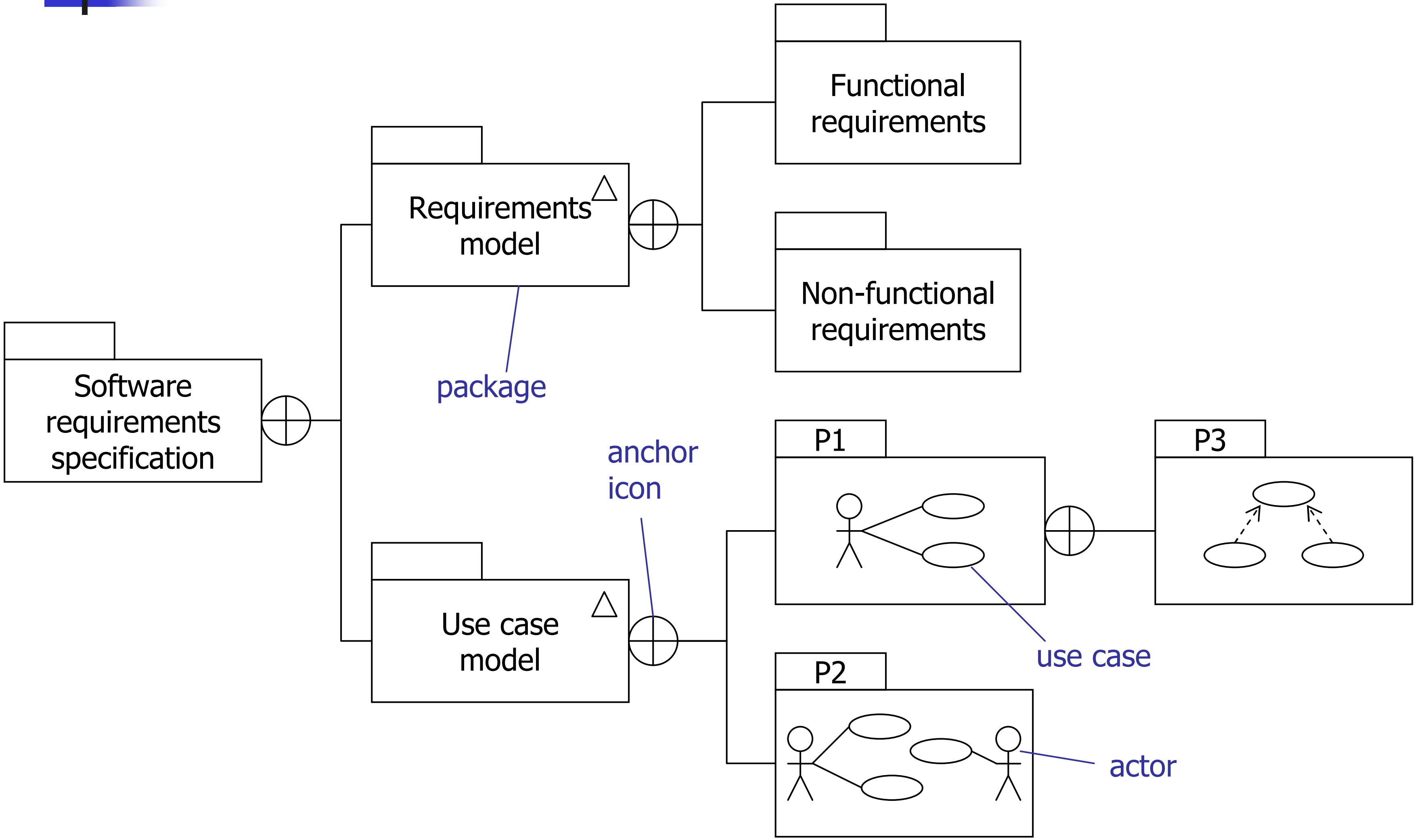
Requirements - introduction

Requirements - purpose

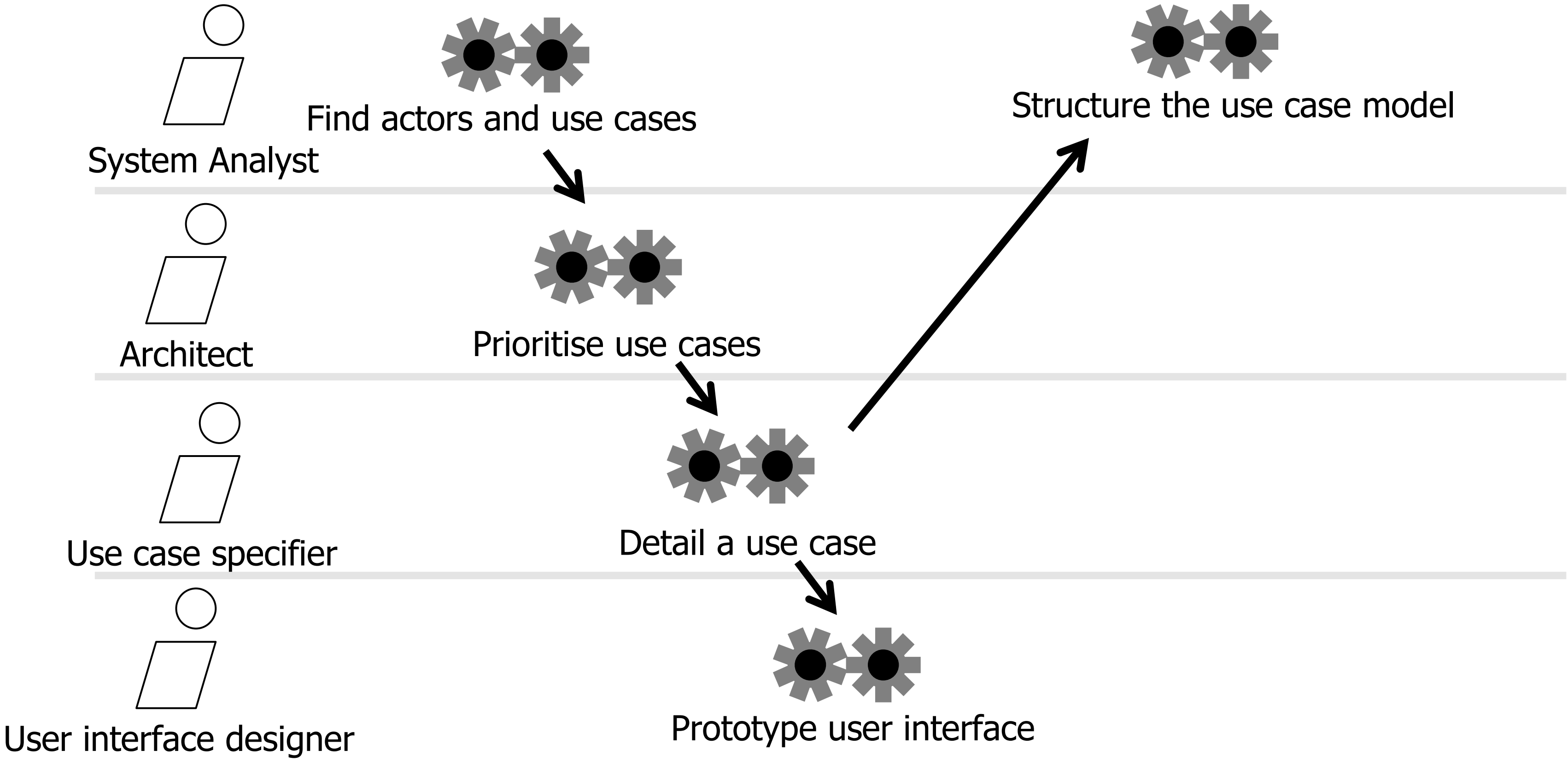
- The purpose of the requirements workflow is to create a high-level specification of what should be implemented
- We interview the stakeholders to find out what they need the system to do for them – their requirements



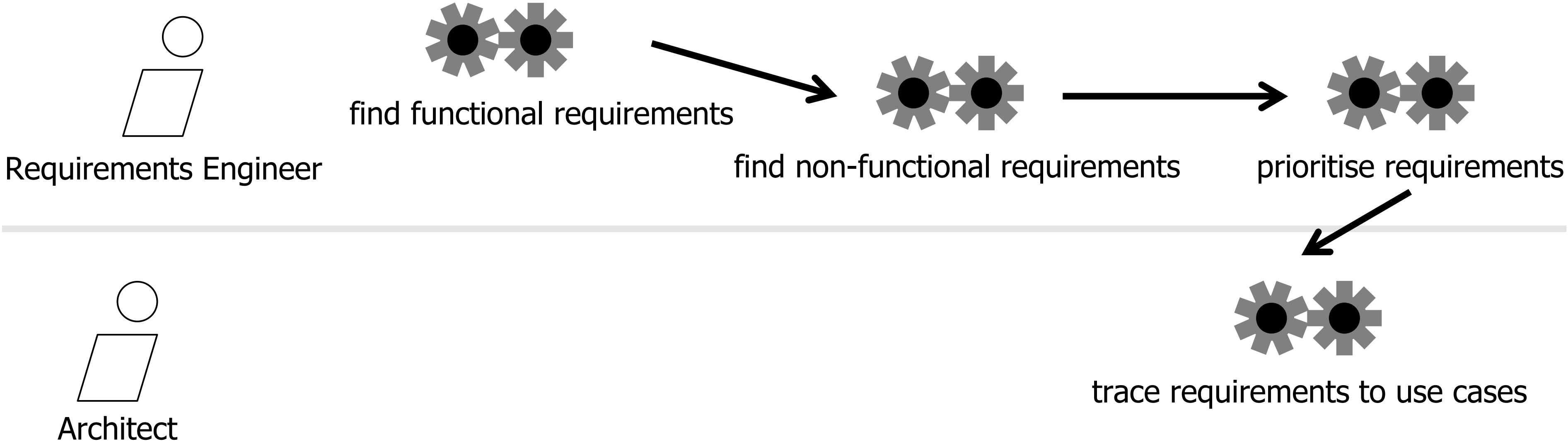
Requirements - metamodel



Requirements - workflow

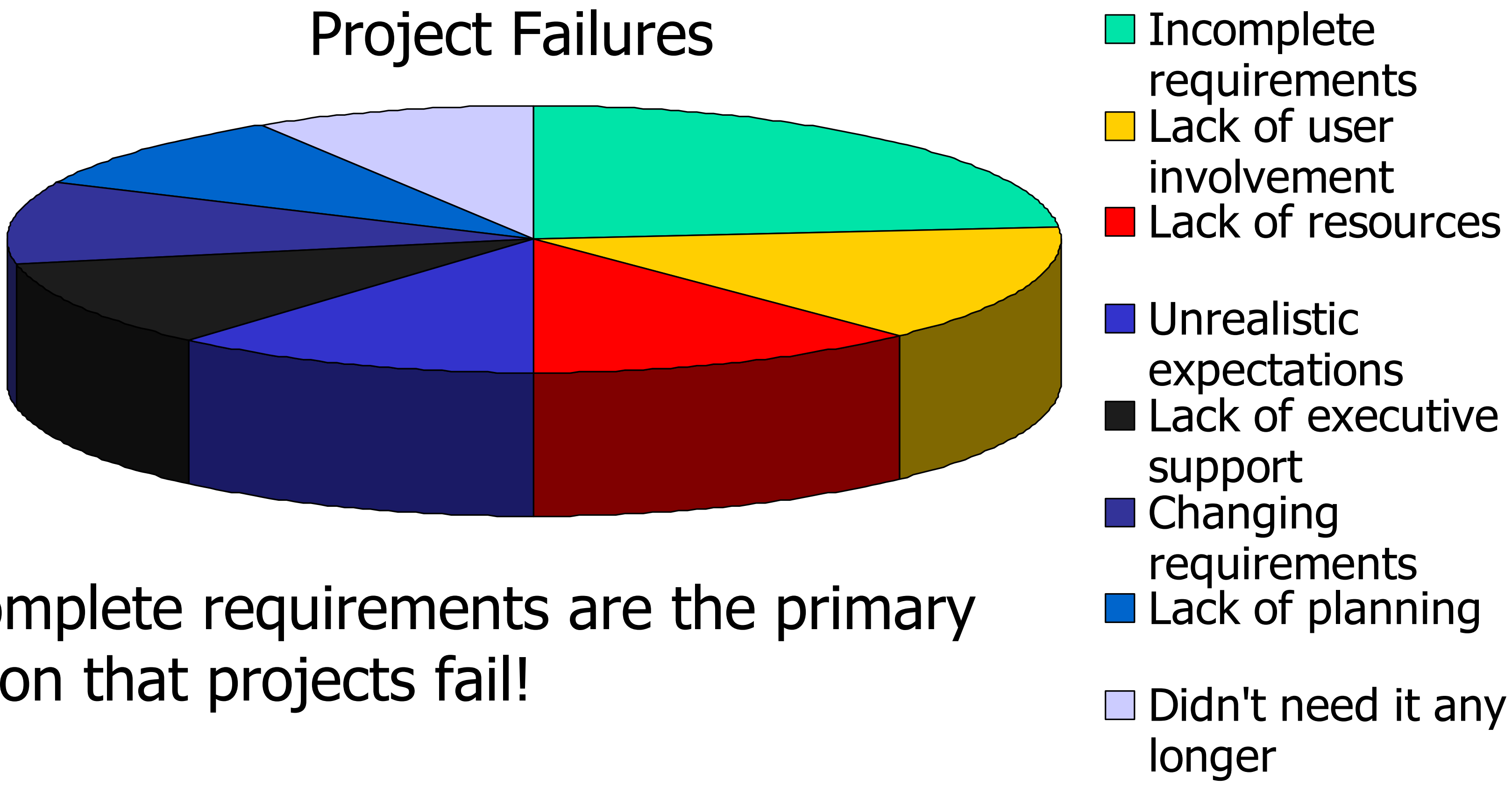


We add…


Requirements Engineer
find functional requirements
find non-functional requirements
prioritise requirements
Architect
trace requirements to use cases

- In order to adopt a rigorous approach to requirements we need to extend the basic UP workflow with functional and non-functional requirements elicitation and requirements traceability

The importance of requirements



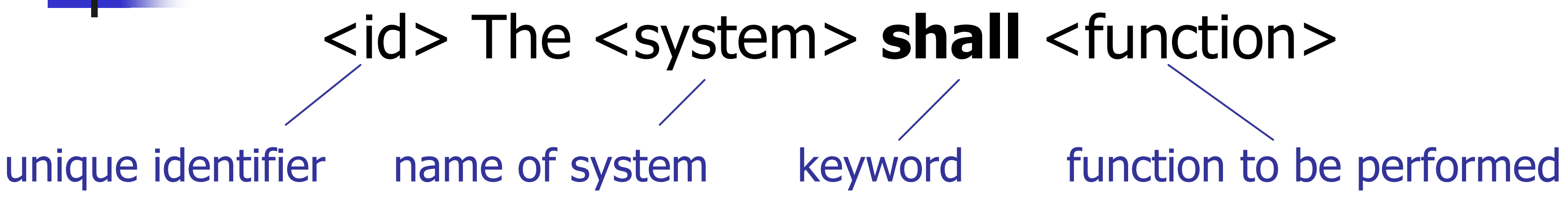
Incomplete requirements are the primary reason that projects fail!

The Standish Group, "The CHAOS Report (1994) "

What are requirements?

- Requirements - "A specification of what should be implemented":
  - What behaviour the system should offer
  - A specific property of the system
  - A constraint on the system
- In UP we create a Software Requirements Specification (SRS)
  - The beginning of the OO software construction process it is a statement of the system requirements for all stakeholders
  - Organises related requirements into sections
- The SRS consists of:
  - Requirements model comprising functional and non-functional requirements
  - Use case model comprising actors and use cases

Writing requirements



<id> The <system> **shall** <function>

unique identifier name of system keyword function to be performed

e.g. "32 The ATM system **shall** validate the PIN number."

- There is no UML standard way of writing requirements!
  - We recommend the uniform sentence structure above
- Functional Requirements - what the system should do
  - "The ATM system shall provide a facility for authenticating the identity of a system user"
- Non-functional Requirements - a constraint on how the functional requirements are implemented
  - "The ATM system shall authenticate a customer in four seconds or less"

The map is not the territory

- Everyone filters information to create their own particular model of the world. Noam Chomsky described this as three processes:
  - Deletion – information is filtered out
  - Distortion – information is modified by the related mechanisms of creation and hallucination
  - Generalisation –the creation of rules, beliefs and principles about truth and falsehood
- These filters shape natural language and so we may need to work to recover filtered information

these filters are applied automatically and unconsciously

Summary

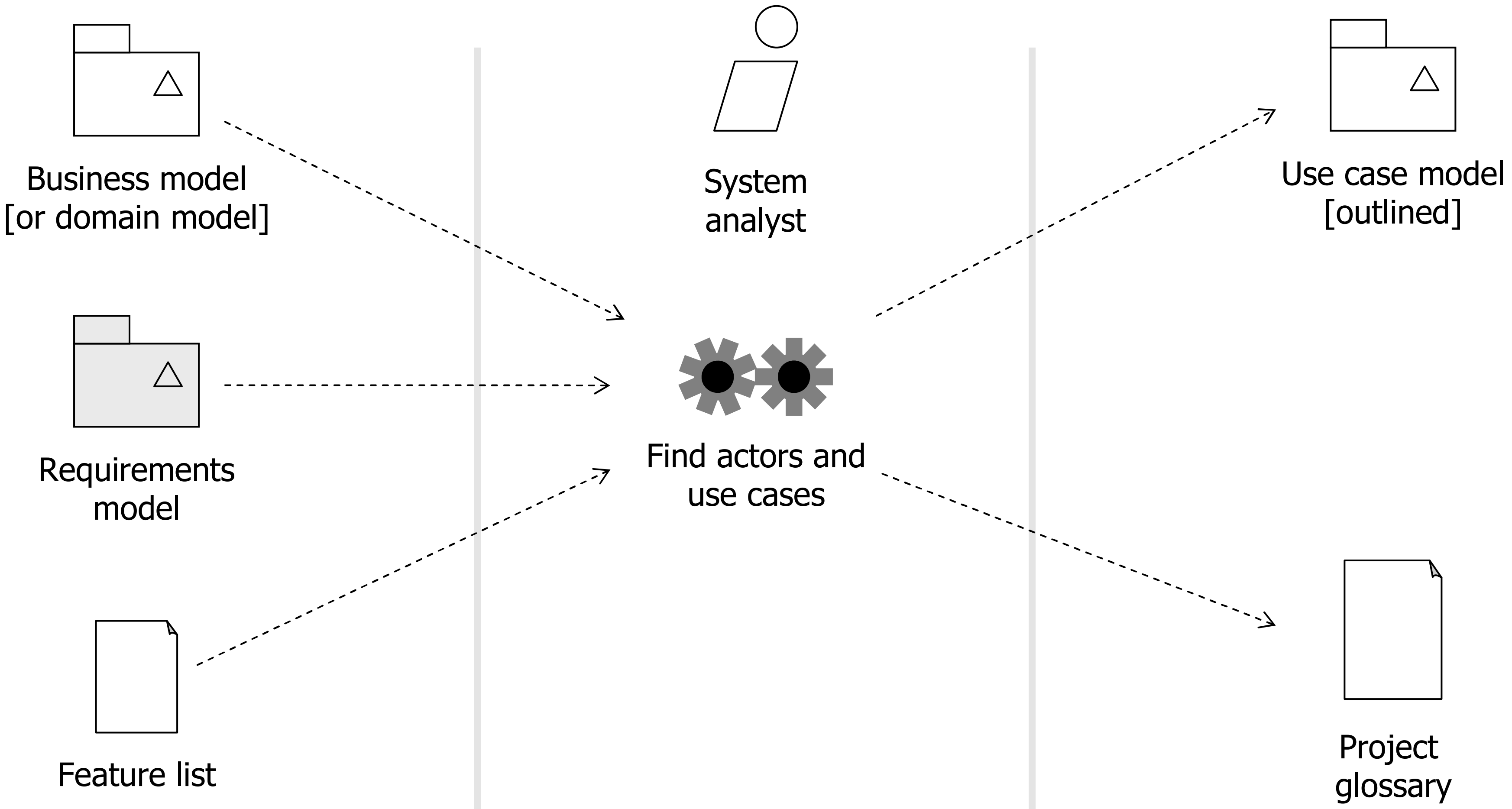
- We have seen how to capture:
  - Functional requirements
  - Non-functional requirements
- We have had a brief overview of the three filters which people use to construct their model of the world

Requirements – use case modelling

Use case modelling

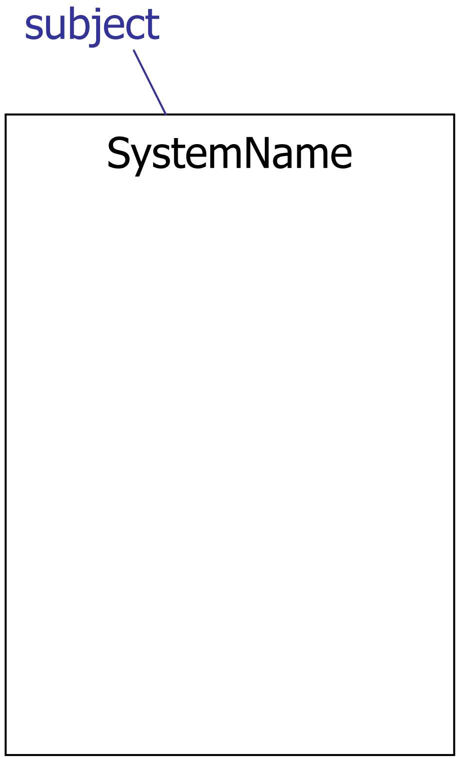
- Use case modelling is a form of requirements engineering
- Use case modelling proceeds as follows:
  - Find the system boundary
  - Find actors
  - Find use cases
    - Use case specification
    - Scenarios
- It lets us identify the system boundary, who or what uses the system, and what functions the system should offer

Find actors and use cases



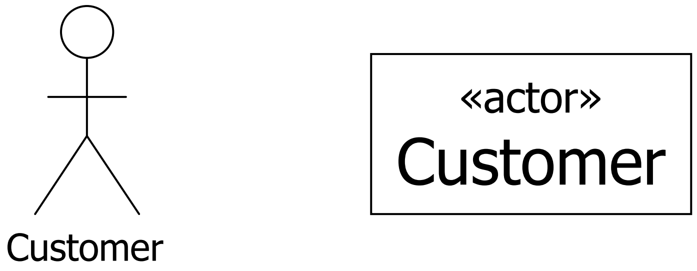
The subject

- Before we can build anything, we need to know:
  - Where the boundary of the system lies
  - Who or what uses the system
  - What functions the system should offer to its users
- We create a Use Case model containing:
  - Subject – the edge of the system
    - also known as the system boundary
  - Actors – who or what uses the system
  - Use Cases – things actors do with the system
  - Relationships - between actors and use cases



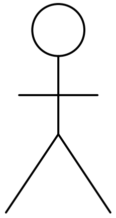
What are actors?

- An actor is anything that interacts *directly* with the system
  - Actors identify who or what uses the system and so indicate where the system boundary lies
- Actors are *external* to the system
- An Actor specifies a *role* that some external entity adopts when interacting with the system



Identifying Actors

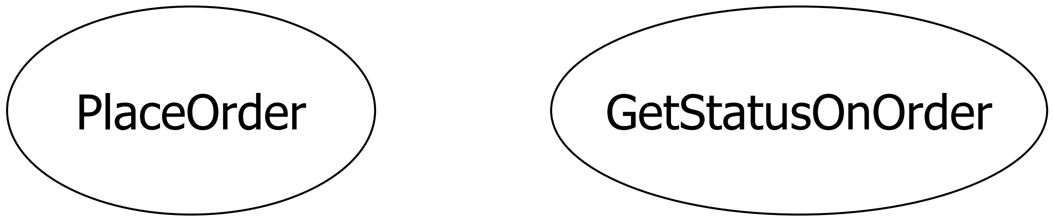
- When identifying actors ask:
  - Who or what uses the system?
  - What roles do they play in the interaction?
  - Who installs the system?
  - Who starts and shuts down the system?
  - Who maintains the system?
  - What other systems use this system?
  - Who gets and provides information to the system?
  - Does anything happen at a fixed time?



Time

What are use cases?

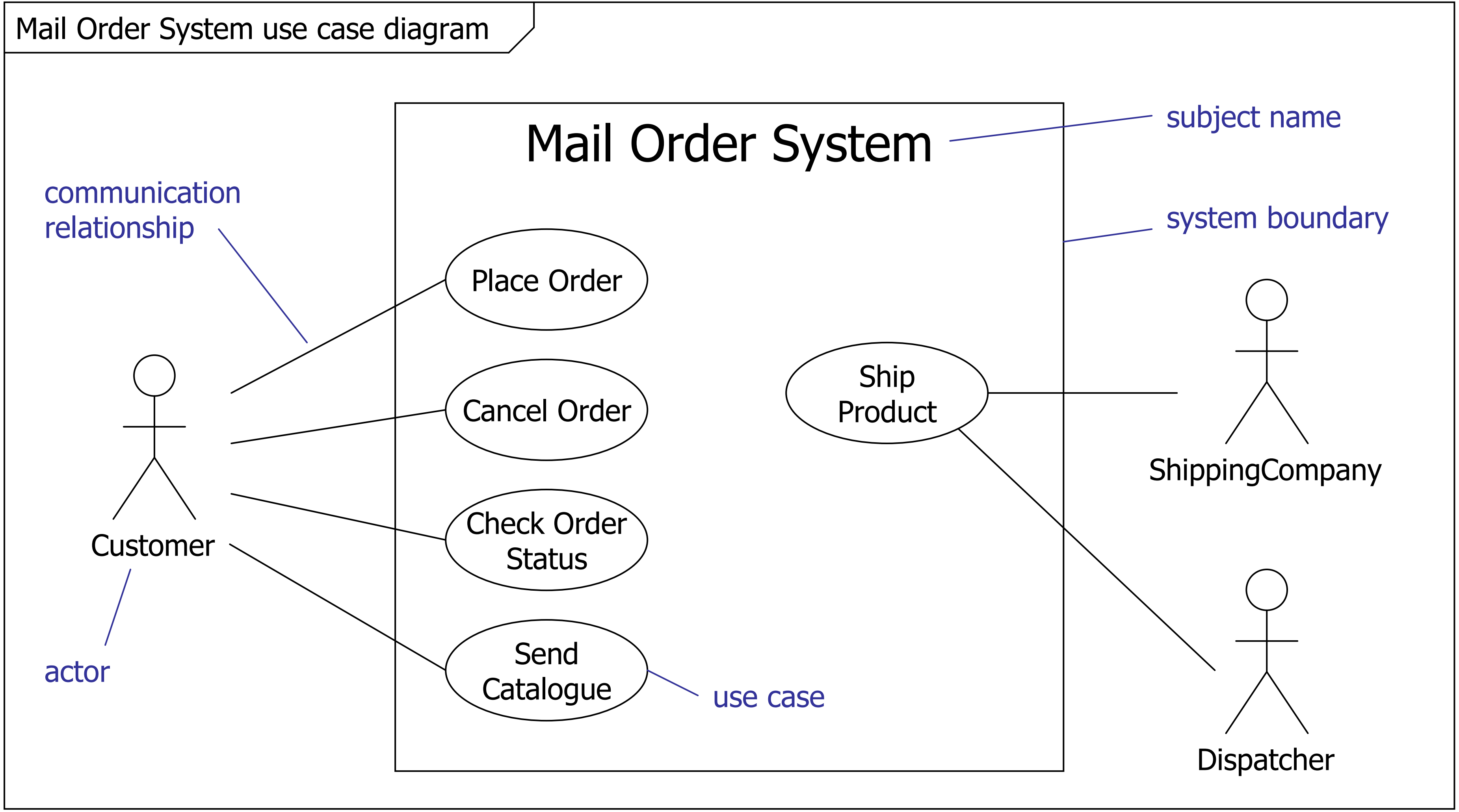
- A use case is something an actor needs the system to do. It is a “case of use” of the system by a specific actor
- Use cases are *always* started by an actor
  - The *primary actor* triggers the use case
  - Zero or more *secondary actors* interact with the use case in some way
- Use cases are *always* written from the point of view of the actors



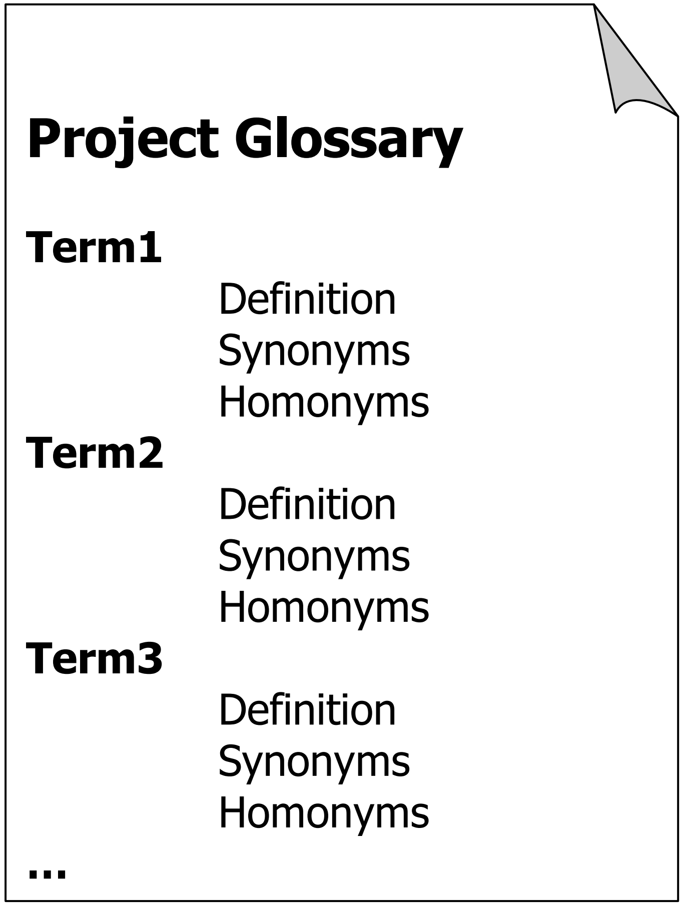
Identifying use cases

- Start with the list of actors that interact with the system
- When identifying use cases ask:
    - What functions will a specific actor want from the system?
    - Does the system store and retrieve information? If so, which actors trigger this behaviour?
    - What happens when the system changes state (e.g. system start and stop)? Are any actors notified?
    - Are there any external events that affect the system? What notifies the system about those events?
    - Does the system interact with any external system?
    - Does the system generate any reports?

The use case diagram



The Project Glossary



Project Glossary

Term1

Definition
Synonyms
Homonyms

Term2

Definition
Synonyms
Homonyms

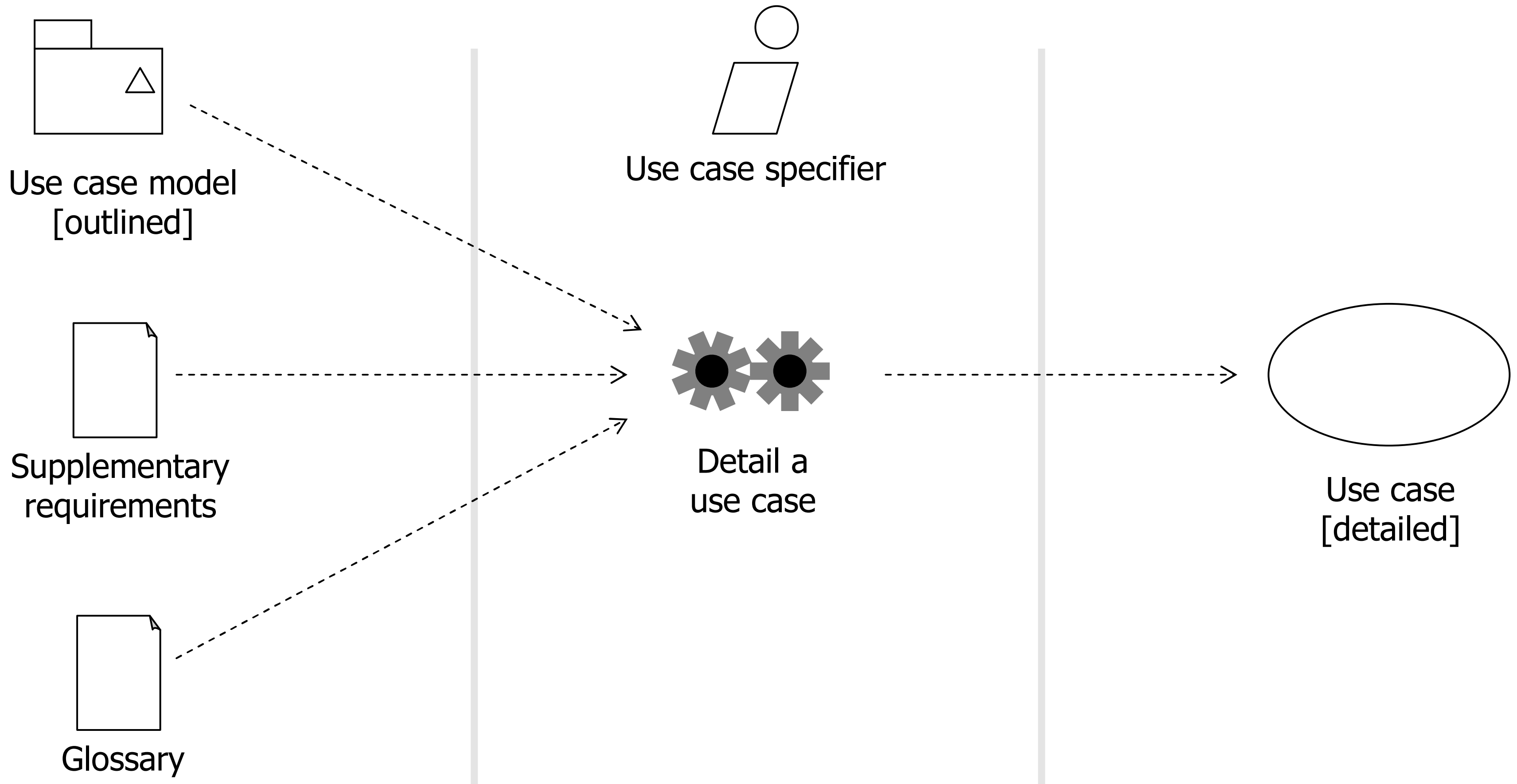
Term3

Definition
Synonyms
Homonyms

…

- In any business domain there is always a certain amount of jargon. It's important to capture the language of the domain in a project glossary
- The aim of the glossary is to define key terms and to resolve synonyms and homonyms
- You are building a vocabulary that you can use to discuss the system with the stakeholders

Detail a use case

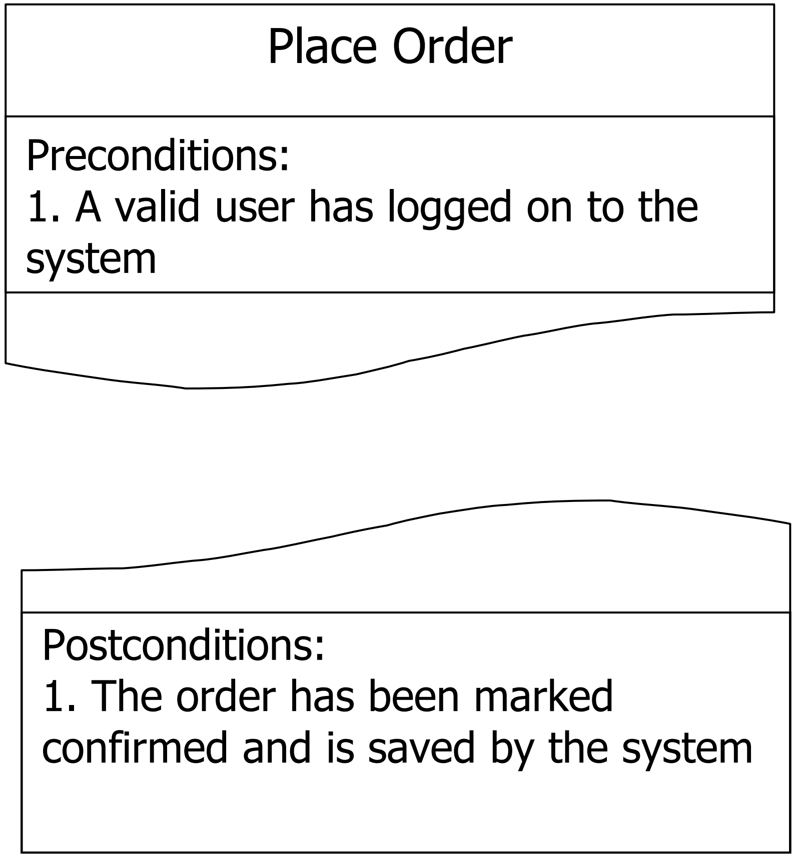


Use case specification

use case name	Use case: PaySalesTax
use case identifier	ID: 1
brief description	Brief description: Pay Sales Tax to the Tax Authority at the end of the business quarter.
the actors involved in the use case	Primary actors: Time
	Secondary actors: TaxAuthority
the system state before the use case can begin	Preconditions: 1. It is the end of the business quarter.
the actual steps of the use case	Main flow: implicit time actor 1. The use case starts when it is the end of the business quarter. 2. The system determines the amount of Sales Tax owed to the Tax Authority. 3. The system sends an electronic payment to the Tax Authority.
the system state when the use case has finished	Postconditions: 1. The Tax Authority receives the correct amount of Sales Tax.
alternative flows	Alternative flows: None.

Pre and postconditions

- Preconditions and postconditions are *constraints*
- Preconditions constrain the state of the system *before* the use case can start
- Postconditions constrain the state of the system *after* the use case has executed
- If there are no preconditions or postconditions write "None" under the heading



Place Order

Preconditions:
1. A valid user has logged on to the system

Postconditions:
1. The order has been marked confirmed and is saved by the system

Main flow

<number> The <something> <some action>

- The flow of events lists the steps in a use case
- It *always* begins by an actor doing something
  - A good way to start a flow of events is:
    1) The use case starts when an <actor> <function>
- The flow of events should be a sequence of short steps that are:
  - Declarative
  - Numbered,
  - Time ordered
- The main flow is always the *happy day* or *perfect world* scenario
  - Everything goes as expected and desired, and there are no errors, deviations, interrupts, or branches
  - Alternatives can be shown by branching or by listing under Alternative flows (see later)

Branching within a flow: If

- Use the keyword if to indicate alternatives within the flow of events
  - There must be a Boolean expression immediately after if
- Use indentation and numbering to indicate the conditional part of the flow
- Use else to indicate what happens if the condition is false (see next slide)

Use case: ManageBasket
ID: 2
Brief description: The Customer changes the quantity of an item in the basket.
Primary actors: Customer
Secondary actors: None.
Preconditions: 1. The shopping basket contents are visible.
Main flow: 1. The use case starts when the Customer selects an item in the basket. 2. If the Customer selects "delete item" 2.1 The system removes the item from the basket. 3. If the Customer types in a new quantity 3.1 The system updates the quantity of the item in the basket.
Postconditions: None.
Alternative flows: None.

Repetition within a flow: For

- We can use the keyword For to indicate the start of a repetition within the flow of events
- The iteration expression immediately after For statement indicates the number of repetitions of the indented text beneath the For statement.

Use case: FindProduct
ID: 3
Brief description: The system finds some products based on Customer search criteria and displays them to the Customer.
Actors: Customer
Preconditions: None.
Main flow: 1. The use case starts when the Customer selects "find product". 2. The system asks the Customer for search criteria. 3. The Customer enters the requested criteria. 4. The system searches for products that match the Customer's criteria. 5. If the system finds some matching products then   5.1 For each product found   5.1.1. The system displays a thumbnail sketch of the product.   5.1.2. The system displays a summary of the product details.   5.1.3. The system displays the product price. 6. Else   6.1. The system tells the Customer that no matching products could be found.
Postconditions: None.
Alternative flows: None.

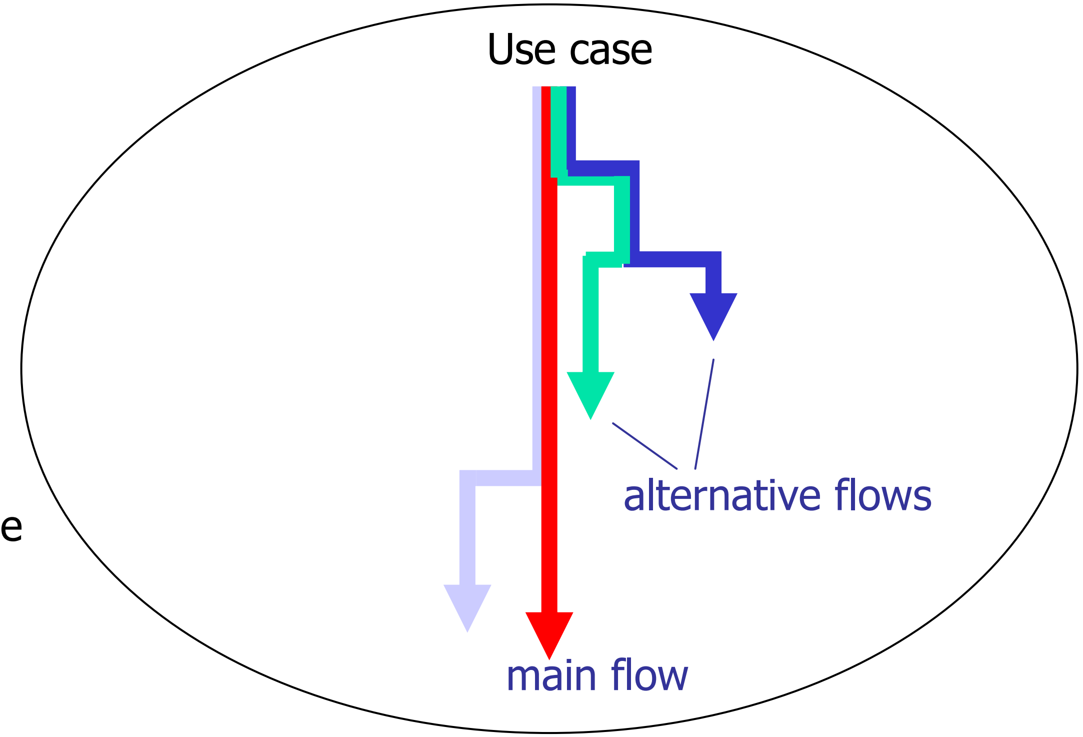
Repetition within a flow: While

- We can use the keyword while to indicate that something repeats while some Boolean condition is true

Use case: FindProduct
ID: 3
Brief description: The system finds some products based on Customer search criteria and displays them to the Customer.
Primary actors: Customer
Secondary actors: None
Preconditions: None.
Main flow: 1. The use case starts when the Customer selects "find product". 2. The system asks the Customer for search criteria. 3. The Customer enters the requested criteria. 4. The system searches for products that match the Customer's criteria. 5. If the system finds some matching products then 5.1 For each product found 5.1.1. The system displays a thumbnail sketch of the product. 5.1.2. The system displays a summary of the product details. 5.1.3. The system displays the product price. 6. Else 6.1. The system tells the Customer that no matching products could be found.
Postconditions: None.
Alternative flows: None.

Branching: Alternative flows

- We may specify one or more *alternative flows* through the flow of events:
  - Alternative flows capture errors, branches, and interrupts
  - Alternative flows *never* return to the main flow
- Potentially very many alternative flows! You need to manage this:
  - Pick the most important alternative flows and document those.
  - If there are groups of similar alternative flows - document one member of the group as an exemplar and (if necessary) add notes to this explaining how the others differ from it.



Only document enough alternative flows to clarify the requirements!

Referencing alternative flows

- List the names of the alternative flows at the end of the use case
- Find alternative flows by examining each step in the main flow and looking for:
  - Alternatives
  - Exceptions
  - Interrupts

Use case: CreateNewCustomerAccount
ID: 5
Brief description: The system creates a new account for the Customer.
Primary actors: Customer
Secondary actors: None.
Preconditions: None.
Main flow: 1. The use case begins when the Customer selects "create new customer account". 2. While the Customer details are invalid 2.1. The system asks the Customer to enter his or her details comprising email address, password and password again for confirmation. 2.2 The system validates the Customer details. 3. The system creates a new account for the Customer.
Postconditions: 1. A new account has been created for the Customer.
Alternative flows: InvalidEmailAddress InvalidPassword Cancel

alternative flows

An alternative flow example

notice how we name and number alternative flows

Alternative flow: CreateNewCustomerAccount:InvalidEmailAddress
ID: 5.1
Brief description: The system informs the Customer that they have entered an invalid email address.
Primary actors: Customer
Secondary actors: None.
Preconditions: 1. The Customer has entered an invalid email address
Alternative flow: 1. The alternative flow begins after step 2.2. of the main flow. 2. The system informs the Customer that he or she entered an invalid email address.
Postconditions: None.

always indicate how the alternative flow begins. In this case it starts after step 2.2 in the main flow

- The alternative flow may be triggered *instead* of the main flow - started by an actor
- The alternative flow may be triggered *after a particular step* in the main flow - after
- The alternative flow may be triggered *at any time* during the main flow - at any time

Requirements tracing

- Given that we can capture functional requirements in a requirements model *and* in a use case model we need some way of relating the two
- There is a many-to-many relationship between requirements and use cases:
  - One use case covers many individual functional requirements
  - One functional requirement may be realised by many use cases
- Hopefully we have CASE support for requirements tracing:
  - With UML tagged values, we can assign numbered requirements to use cases
  - We can capture use case names in our Requirements Database
- If there is no CASE support, we can create a Requirements Traceability matrix

		Use cases			
		U1	U2	U3	U4
Requirements	R1	🟥			
	R2		🟥	🟥	
	R3			🟥	
	R4				🟥
	R5	🟥			

Requirements Traceability Matrix

When to use use case analysis

- Use cases describe system behaviour from the point of view of one or more actors. They are the *best* choice when:
  - The system is dominated by functional requirements
  - The system has many types of user to which it delivers different functionality
  - The system has many interfaces
- Use cases are designed to capture *functional* requirements. They are a *poor* choice when:
  - The system is dominated by non-functional requirements
  - The system has few users
  - The system has few interfaces

Summary

- We have seen how to capture functional requirements with use cases
- We have looked at:
  - Use cases
  - Actors
  - Branching with if
  - Repetition with for and while
  - Alternative flows
  - Requirements tracing

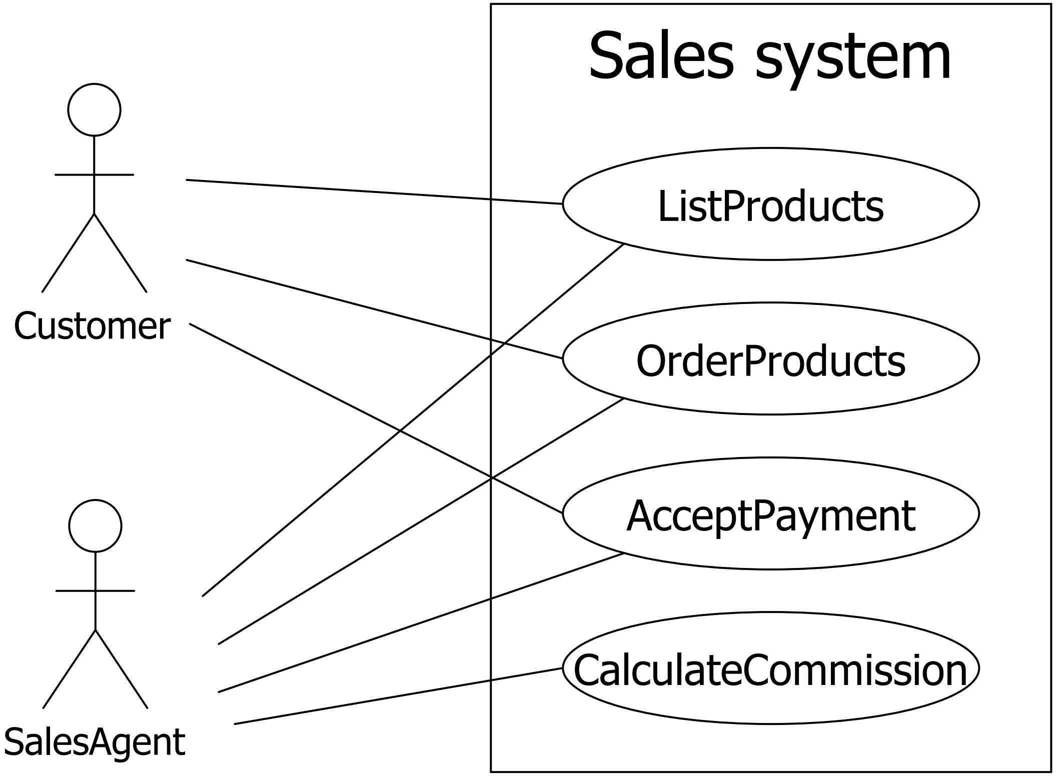
Requirements – advanced use case modelling

More relationships…

- We have studied basic use case analysis, but there are relationships that we have still to explore:
  - Actor generalisation
  - Use case generalisation
  - «include» – between use cases
  - «extend» – between use cases

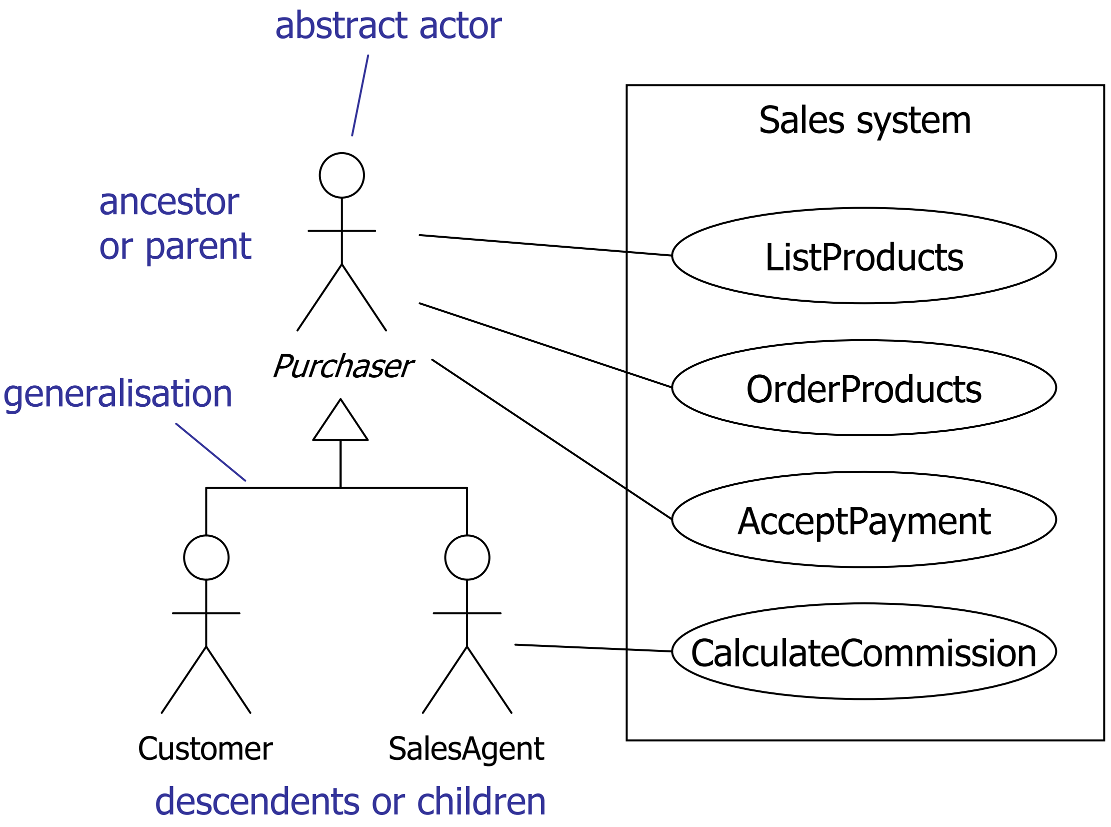
Actor generalization - example

- The Customer and the Sales Agent actors are *very* similar
- They both interact with List products, Order products, Accept payment
- Additionally, the Sales Agent interacts with Calculate commission
- Our diagram is a *mess* – can we simplify it?



Actor generalisation

- If two actors communicate with the same set of use cases in the same way, then we can express this as a generalisation to another (possibly abstract) actor
- The descendent actors inherit the roles and relationships to use cases held by the ancestor actor
- We can substitute a descendent actor anywhere the ancestor actor is expected. This is the *substitutability principle*

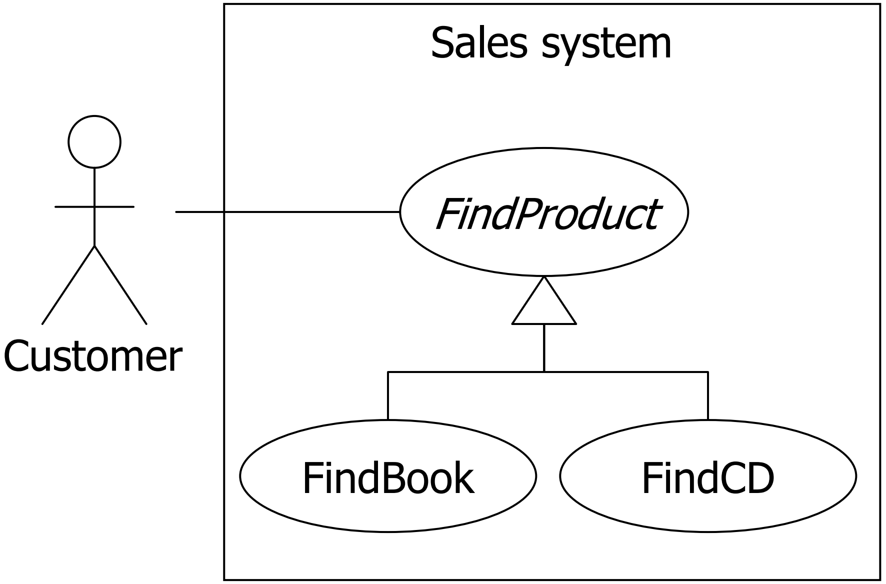


Use actor generalization when it simplifies the model

Use case generalisation

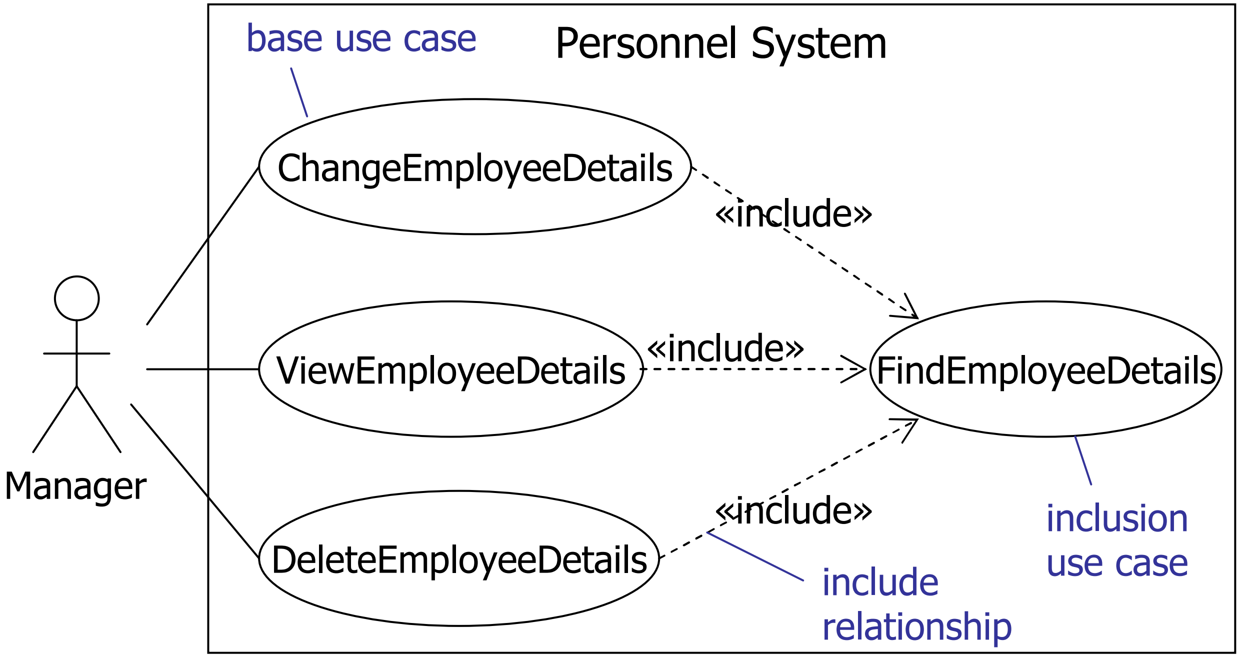
- The ancestor use case must be a more general case of one or more descendant use cases
- Child use cases are more specific forms of their parent
- They can inherit, add and override features of their parent

Use case generalization semantics			
Use case element	Inherit	Add	Override
Relationship	Yes	Yes	No
Extension point	Yes	Yes	No
Precondition	Yes	Yes	Yes
Postcondition	Yes	Yes	Yes
Step in main flow	Yes	Yes	Yes
Alternative flow	Yes	Yes	Yes



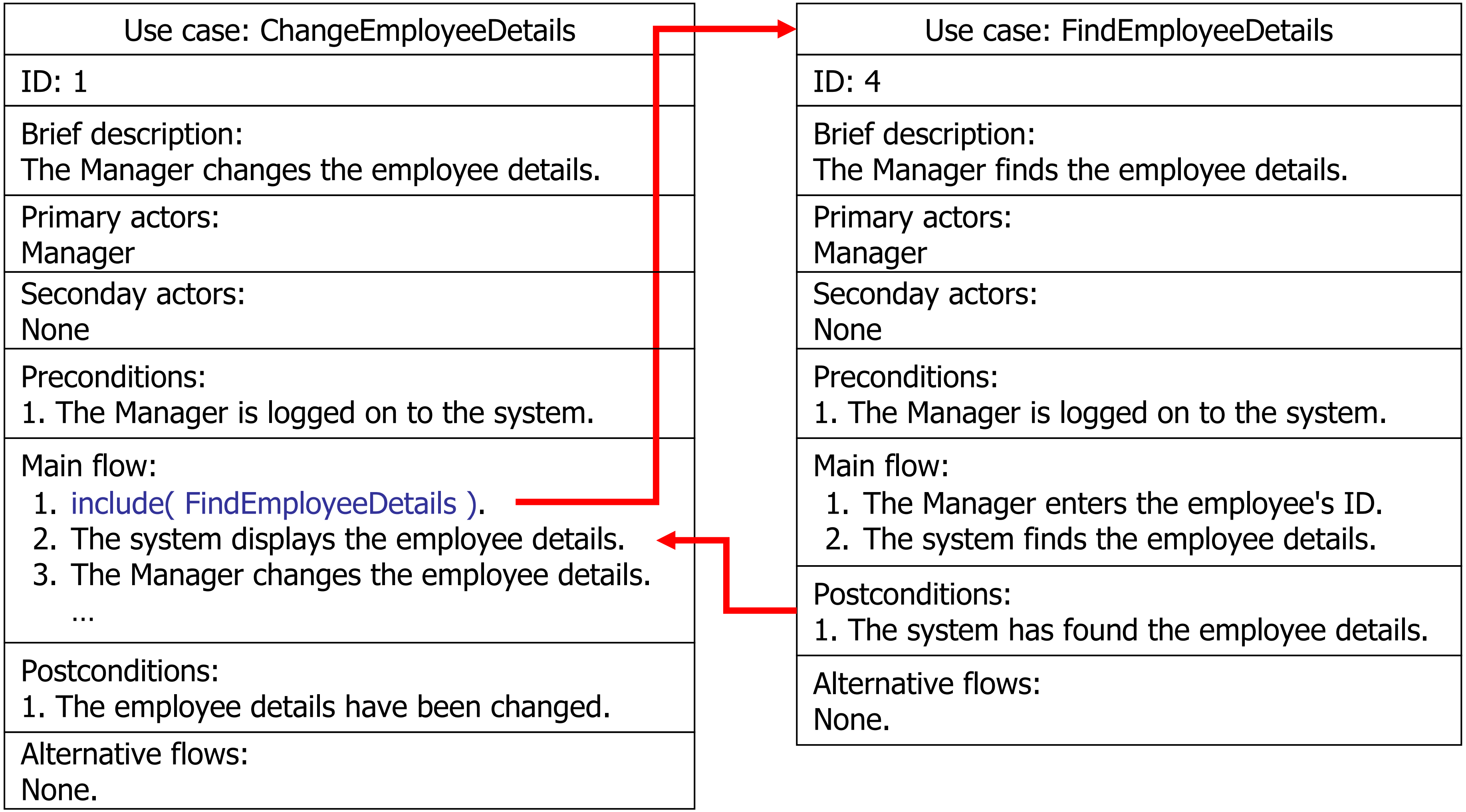
«include»

- The base use case executes until the point of inclusion: include(InclusionUseCase)
  - Control passes to the inclusion use case which executes
  - When the inclusion use case is finished, control passes back to the base use case which finishes execution
- Note:
  - Base use cases are *not complete* without the included use cases
  - Inclusion use cases may be complete use cases, or they may just specify a fragment of behaviour for inclusion elsewhere



When use cases share common behaviour we can factor this out into a separate inclusion use case and «include» it in base use cases

«include» example

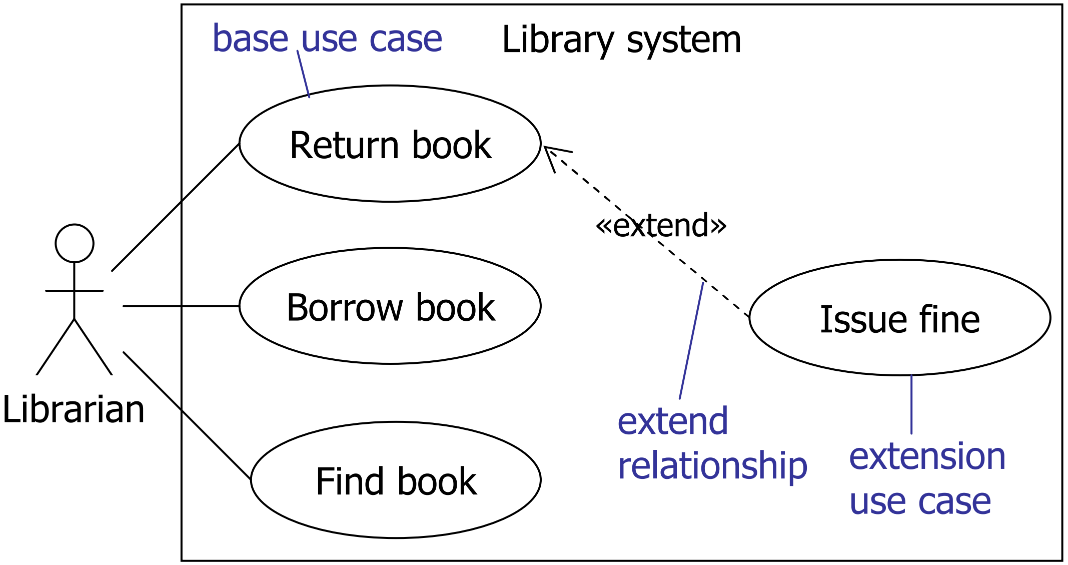


Use case: ChangeEmployeeDetails
ID: 1
Brief description: The Manager changes the employee details.
Primary actors: Manager
Seconday actors: None
Preconditions: 1. The Manager is logged on to the system.
Main flow: 1. include(FindEmployeeDetails). 2. The system displays the employee details. 3. The Manager changes the employee details. …
Postconditions: 1. The employee details have been changed.
Alternative flows: None.

Use case: FindEmployeeDetails
ID: 4
Brief description: The Manager finds the employee details.
Primary actors: Manager
Seconday actors: None
Preconditions: 1. The Manager is logged on to the system.
Main flow: 1. The Manager enters the employee's ID. 2. The system finds the employee details.
Postconditions: 1. The system has found the employee details.
Alternative flows: None.

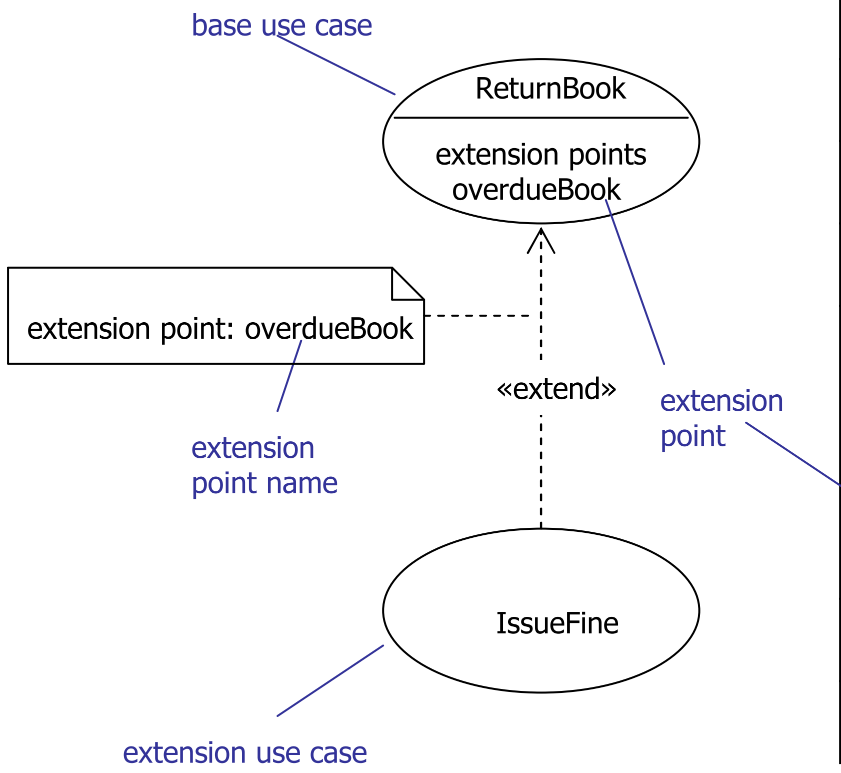
«extend»

- «extend» is a way of adding new behaviour into the base use case by inserting behaviour from one or more extension use cases
    - The base use case specifies one or more extension points in its flow of events
- The extension use case may contain several insertion segments
- The «extend» relationship may specify *which* of the base use case extension points it is extending



The extension use case inserts behaviour into the base use case.
The base use case provides extension points, but *does not know* about the extensions.

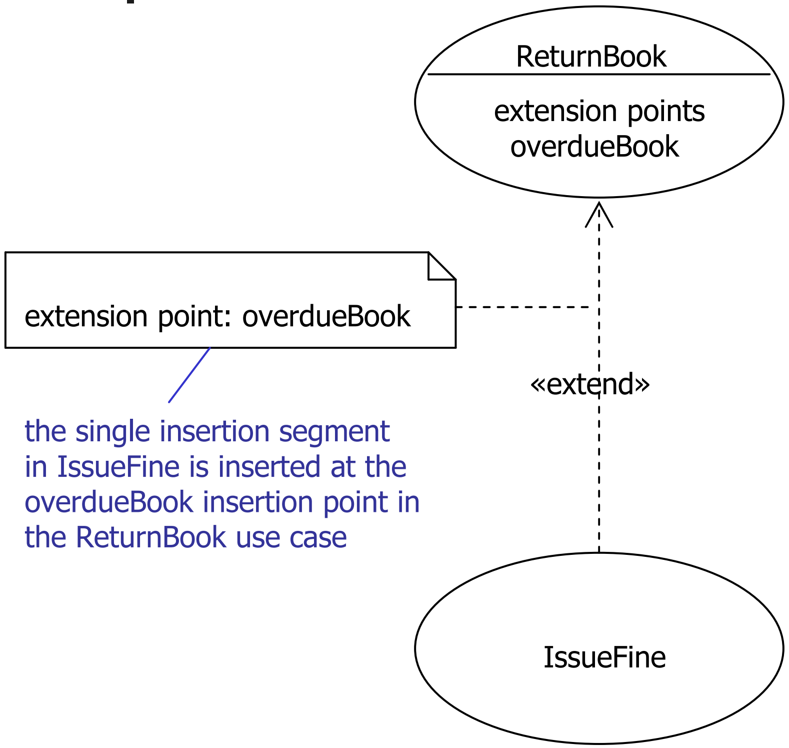
Base use case



Use case: ReturnBook
ID: 9
Brief description: The Librarian returns a borrowed book.
Primary actors: Librarian
Secondary actors: None.
Preconditions: 1. The Librarian is logged on to the system.
Main flow: 1. The Librarian enters the borrower's ID number. 2. The system displays the borrower's details including the list of borrowed books. 3. The Librarian finds the book to be returned in the list of books. extension point: overdueBook 4. The Librarian returns the book. …
Postconditions: 1. The book has been returned.
Alternative flows: None.

- There is an extension point overdueBook just before step 4 of the flow of events
- Extension points are *not* numbered, as they are *not* part of the flow

Extension use case

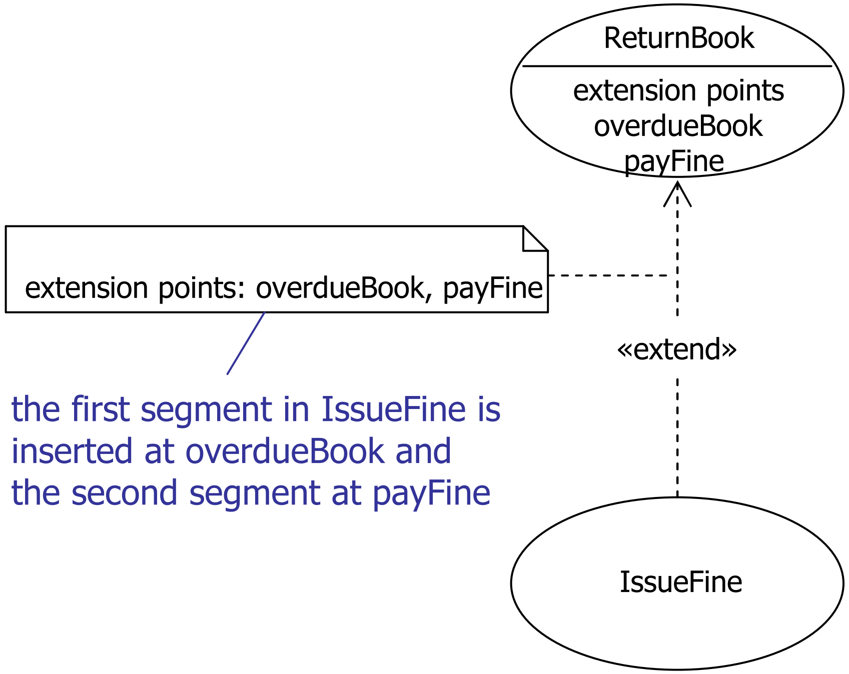


the single insertion segment in IssueFine is inserted at the overdueBook insertion point in the ReturnBook use case

Extension Use case: IssueFine
ID: 10
Brief description: Segment 1: The Librarian records and prints out a fine.
Primary actors: Librarian
Secondary actors: None.
Segment 1 preconditions: 1. The returned book is overdue.
Segment 1 flow: 1. The Librarian enters details of the fine into the system. 2. The system prints out the fine.
Segment 1 postconditions: 1. The fine has been recorded in the system. 2. The system has printed out the fine.

- Extension use cases have one or more *insertion segments* which are behaviour fragments that will be inserted at the specified extension points in the base use case

Multiple insertion points

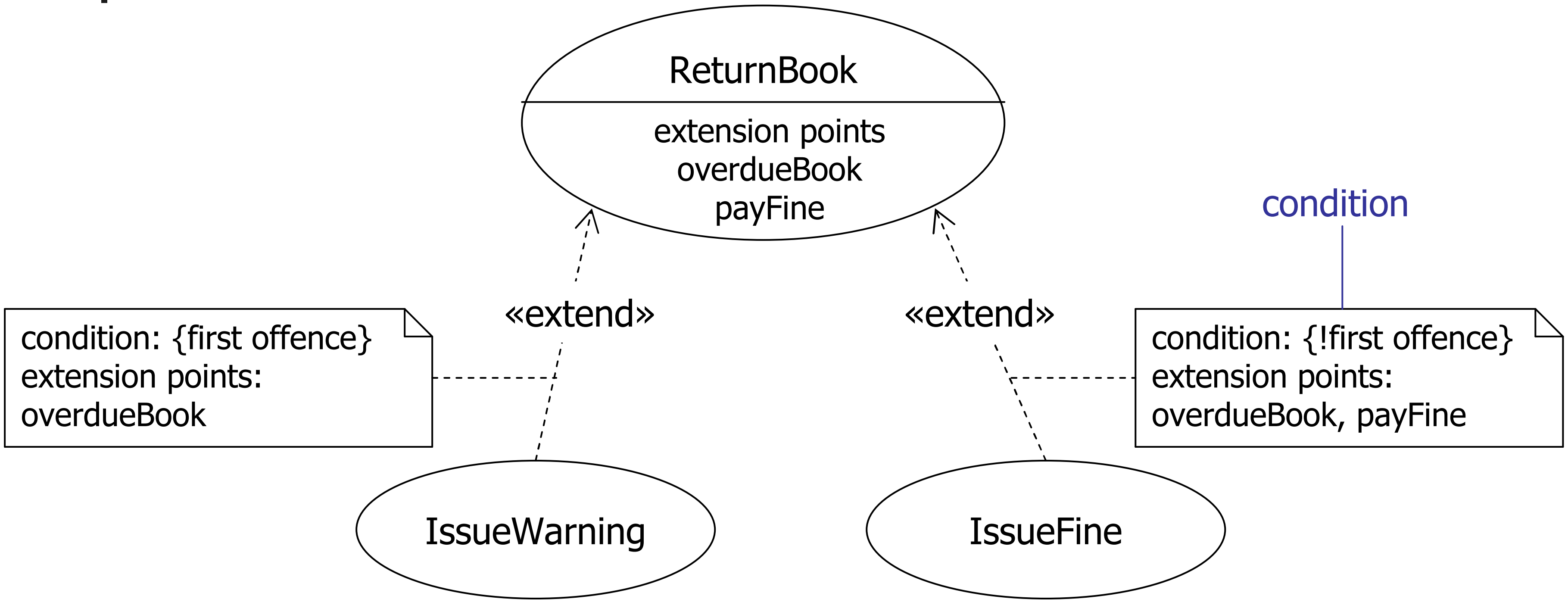


the first segment in IssueFine is inserted at overdueBook and the second segment at payFine

Extension Use case: IssueFine
ID: 10
Brief description: Segment 1: The Librarian records and prints out a fine. Segment 2: The Librarian accepts payment for a fine.
Primary actors: Librarian
Secondary actors: None.
Segment 1 preconditions: 1. The returned book is overdue.
Segment 1 flow: 1. The Librarian enters details of the fine into the system. 2. The system prints out the fine.
Segment 1 postconditions: 1. The fine has been recorded in the system. 2. The system has printed out the fine.
Segment 2 preconditions: 1. A fine is due from the borrower.
Segment 2 flow: 1. The Librarian accepts payment for the fine from the borrower. 2. The Librarian enters the paid fine in the system. 3. The system prints out a receipt for the paid fine.
Segment 2 postconditions: 1. The fine is recorded as paid. 2. The system has printed a receipt for the fine.

- If more than one extension point is specified in the «extend» relationship then the extension use case must have the *same number* of insertion segments

Conditional extensions



- We can specify conditions on «extend» relationships
  - Conditions are Boolean expressions
  - The insertion is made if and only if the condition evaluates to true

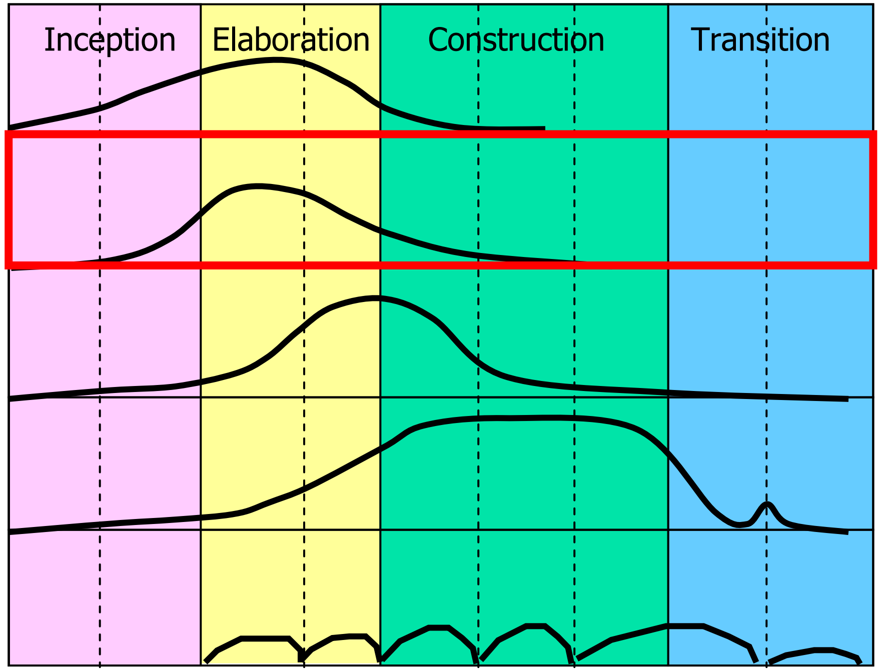
Summary

- We have learned about techniques for advanced use case modelling:
  - Actor generalisation
  - Use case generalisation
  - «include»
  - «extend»
- Use advanced features with discretion only where they simplify the model!

Analysis - introduction

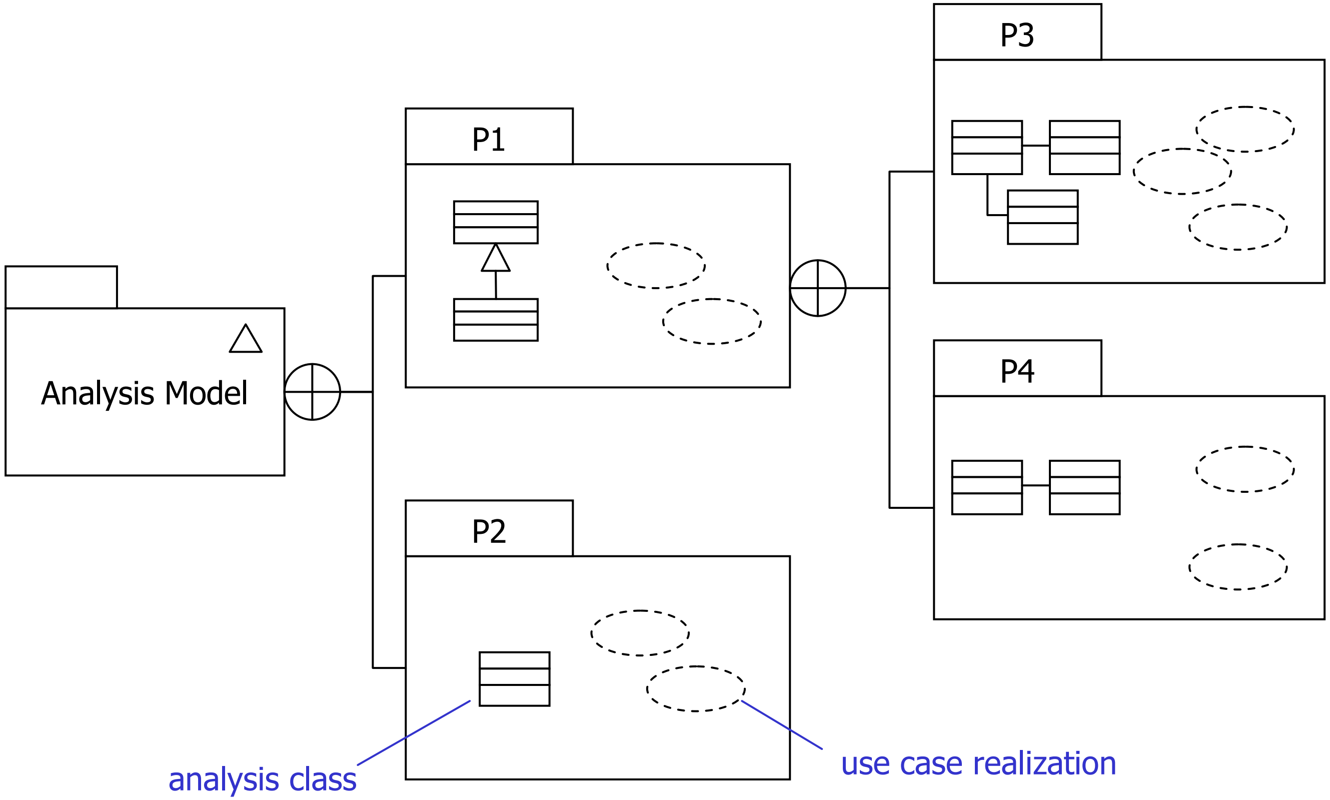
Analysis - purpose

- Produce an Analysis Model of the system's desired behaviour:
  - This model should be a statement of what the system does not how it does it
  - We can think of the analysis model as a "first-cut" or "high level" design model
  - It is in the language of the business
- In the Analysis Model we identify:
  - Analysis classes
  - Use-case realizations



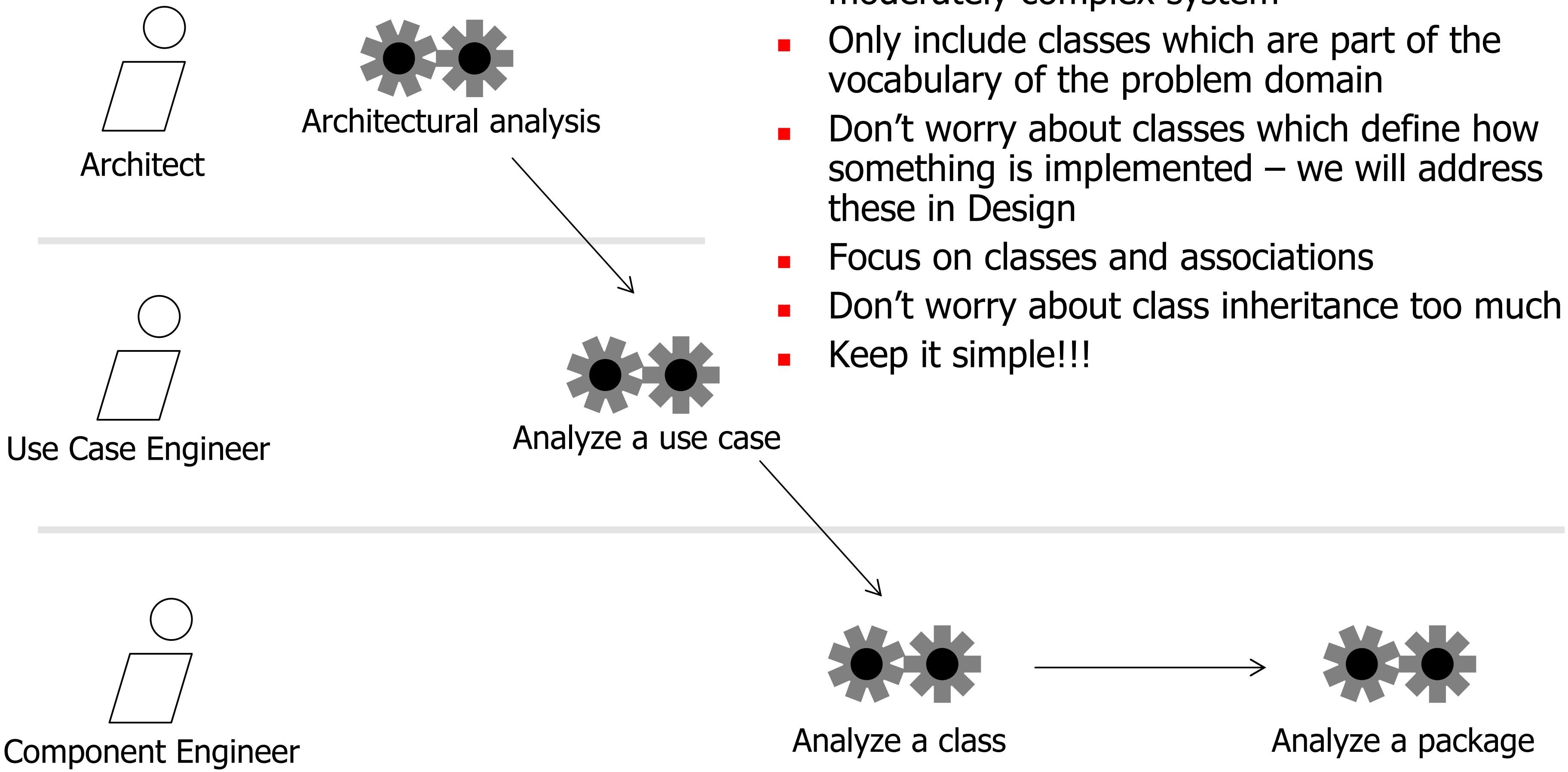
Analysis - metamodel

- Packages contain UML modelling elements and diagrams (we only show the elements here)
- Each element or diagram is owned by exactly one package



Workflow - Analysis

- Analysis guidelines: 6.5
  - 50 to 100 classes in the analysis model of a moderately complex system
  - Only include classes which are part of the vocabulary of the problem domain
  - Don’t worry about classes which define how something is implemented – we will address these in Design
  - Focus on classes and associations
  - Don’t worry about class inheritance too much
  - Keep it simple!!!



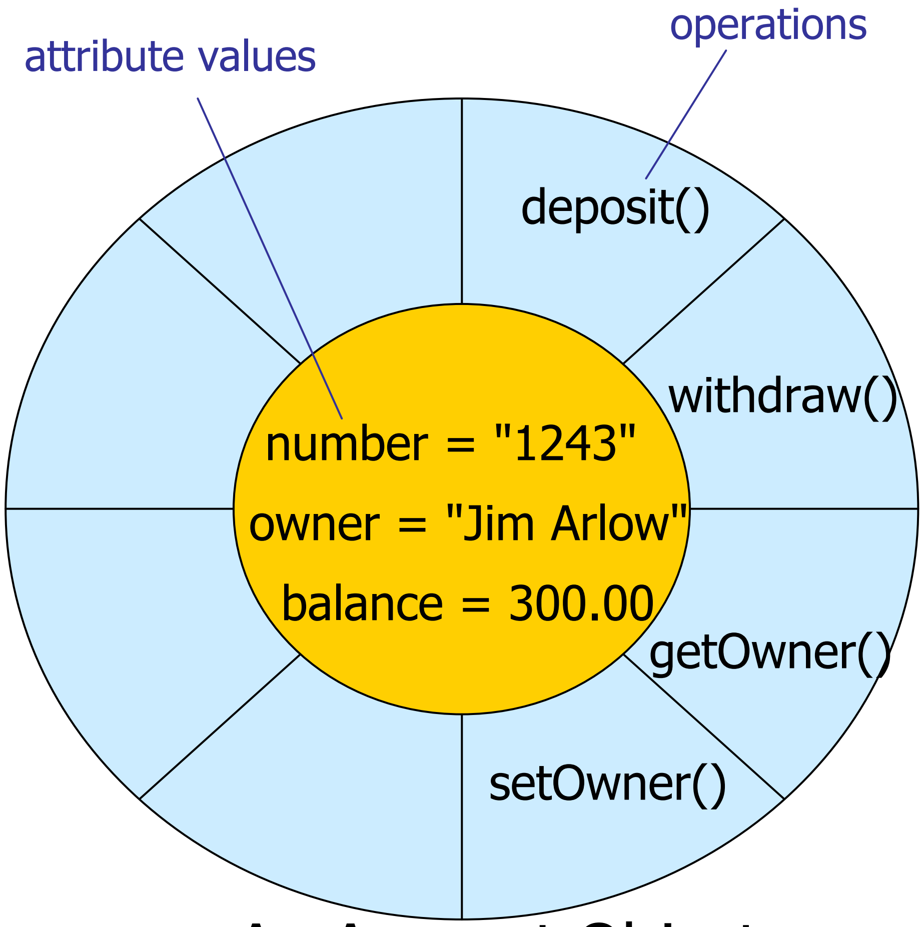
Analysis - objects and classes

What are objects?

- Objects consist of data and function packaged together in a reusable unit. Objects *encapsulate* data
- Every object is an instance of some *class* which defines the common set of *features* (attributes and operations) shared by all of its instances. Objects have:
  - Attribute values – the data part
  - Operations – the behaviour part
- All objects have:
  - *Identity*: Each object has its own unique identity and can be accessed by a unique handle
  - *State*: This is the actual data values stored in an object at any point in time
  - *Behaviour*: The set of operations that an object can perform

Encapsulation

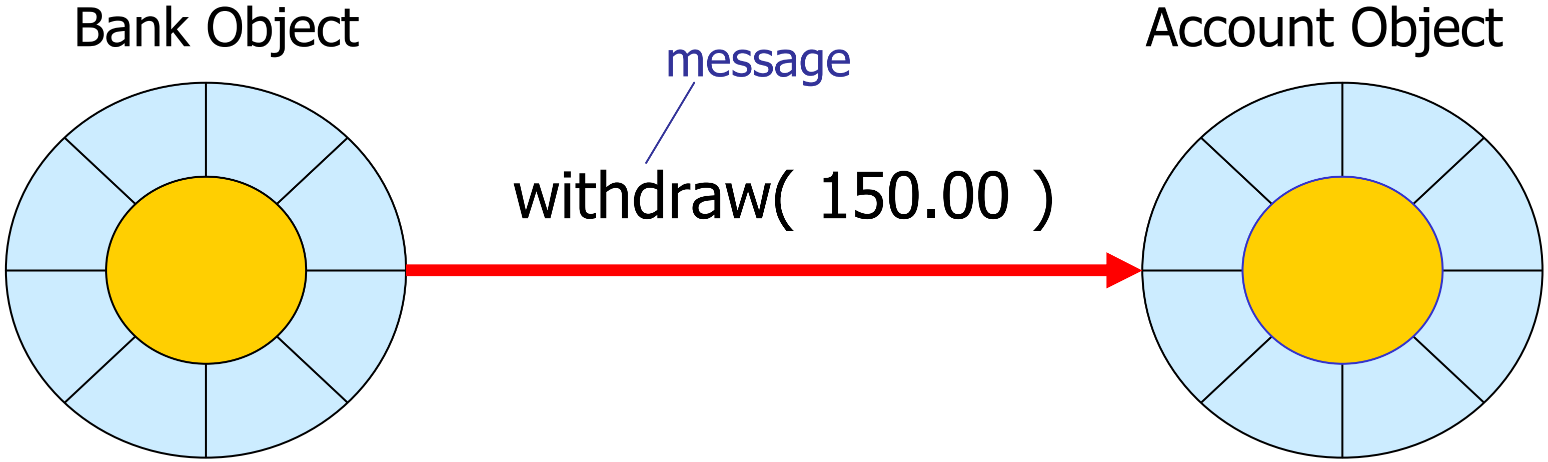
- Data is hidden inside the object. The only way to access the data is via one of the operations
- This is *encapsulation* or *data hiding* and it is a very powerful idea. It leads to more robust software and reusable code.



An Account Object

Messaging

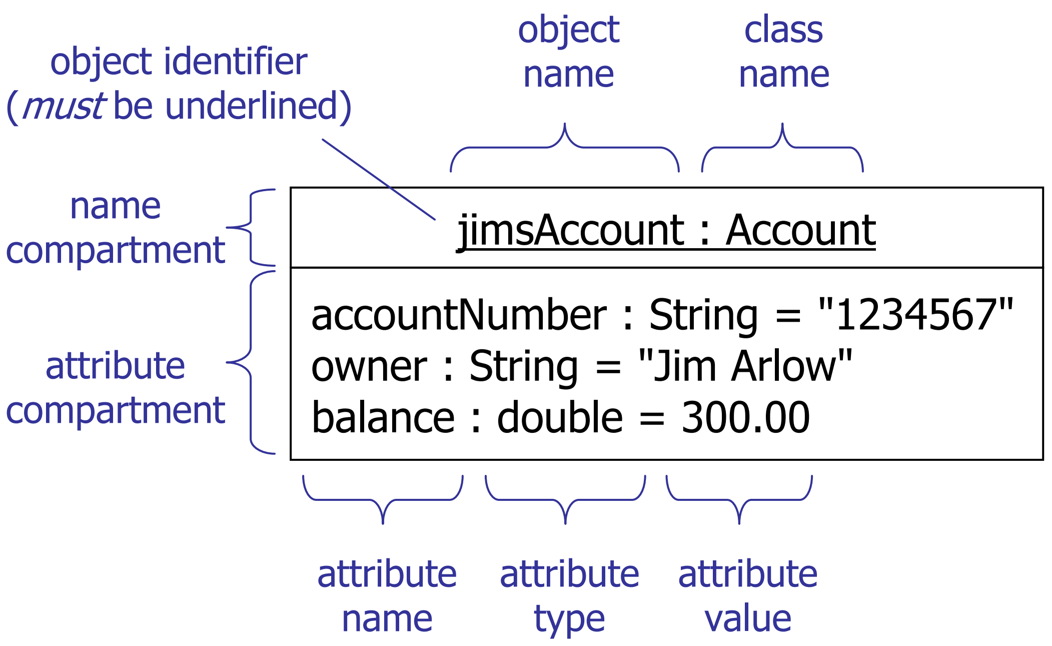
- In OO systems, objects send messages to each other over links
- These messages cause an object to invoke an operation



the Bank object sends the message "withdraw 150.00" to an Account object.

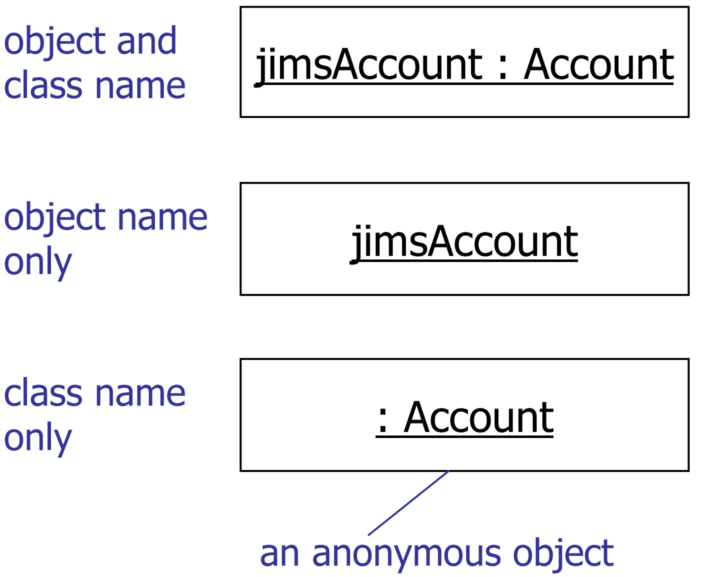
the Account object responds by invoking its withdraw operation. This operation decrements the account balance by 150.00.

UML Object Syntax



variants

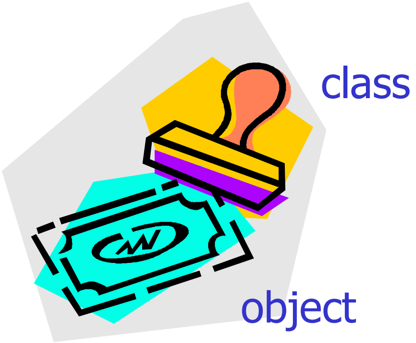
(N.B. we've omitted the attribute compartment)



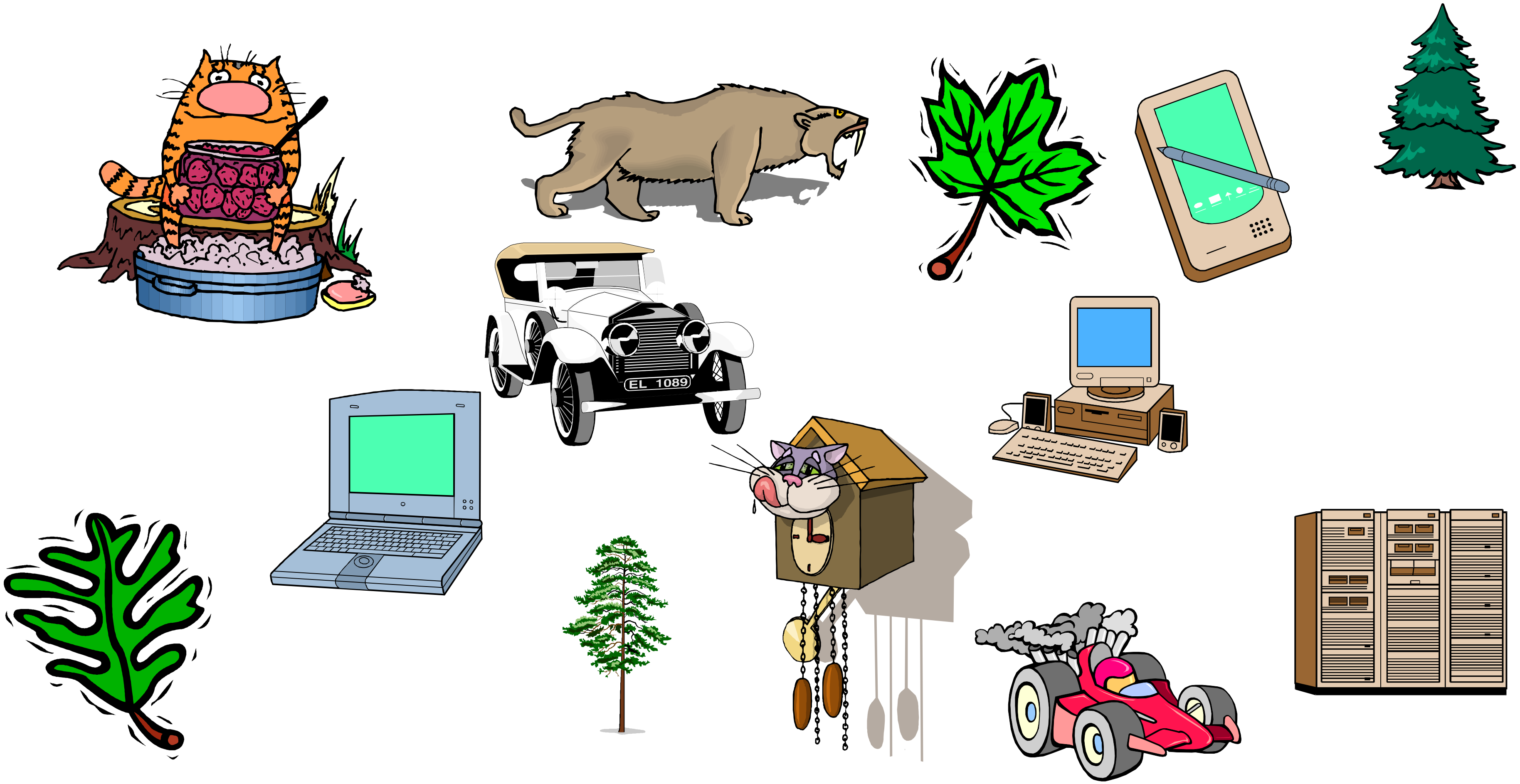
- All objects of a particular class have the same set of operations. They are not shown on the object diagram, they are shown on the class diagram (see later)
- Attribute types are often omitted to simplify the diagram
- Naming:
  - object and attribute names in lowerCamelCase
  - class names in UpperCamelCase

What are classes?

- Every object is an instance of one class - the class describes the "type" of the object
- Classes allow us to model sets of objects that have the *same* set of features - a class acts as a template for objects:
  - The class determines the structure (set of features) of all objects of that class
  - All objects of a class *must* have the same set of operations, *must* have the same attributes, but *may* have different attribute values
- Classification is one of the most important ways we have of organising our view of the world
- Think of classes as being like:
  - Rubber stamps
  - Cookie cutters

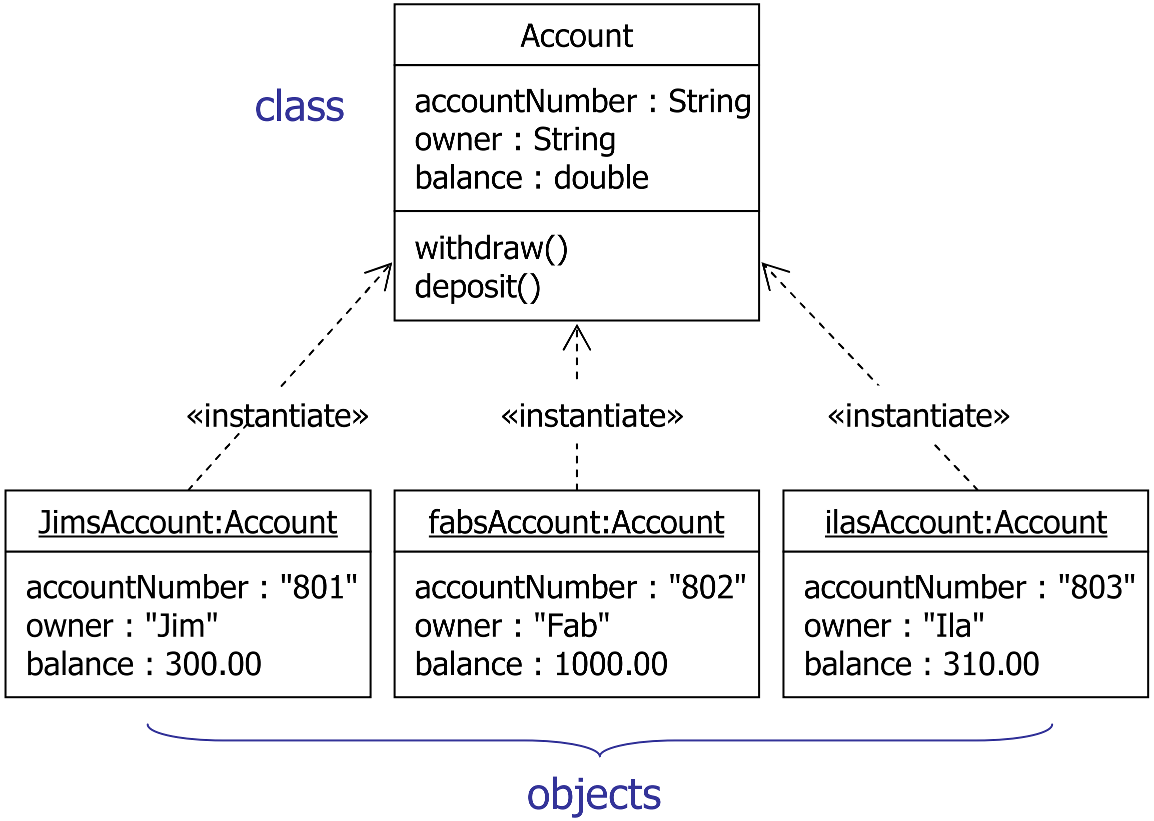


Exercise - how many classes?



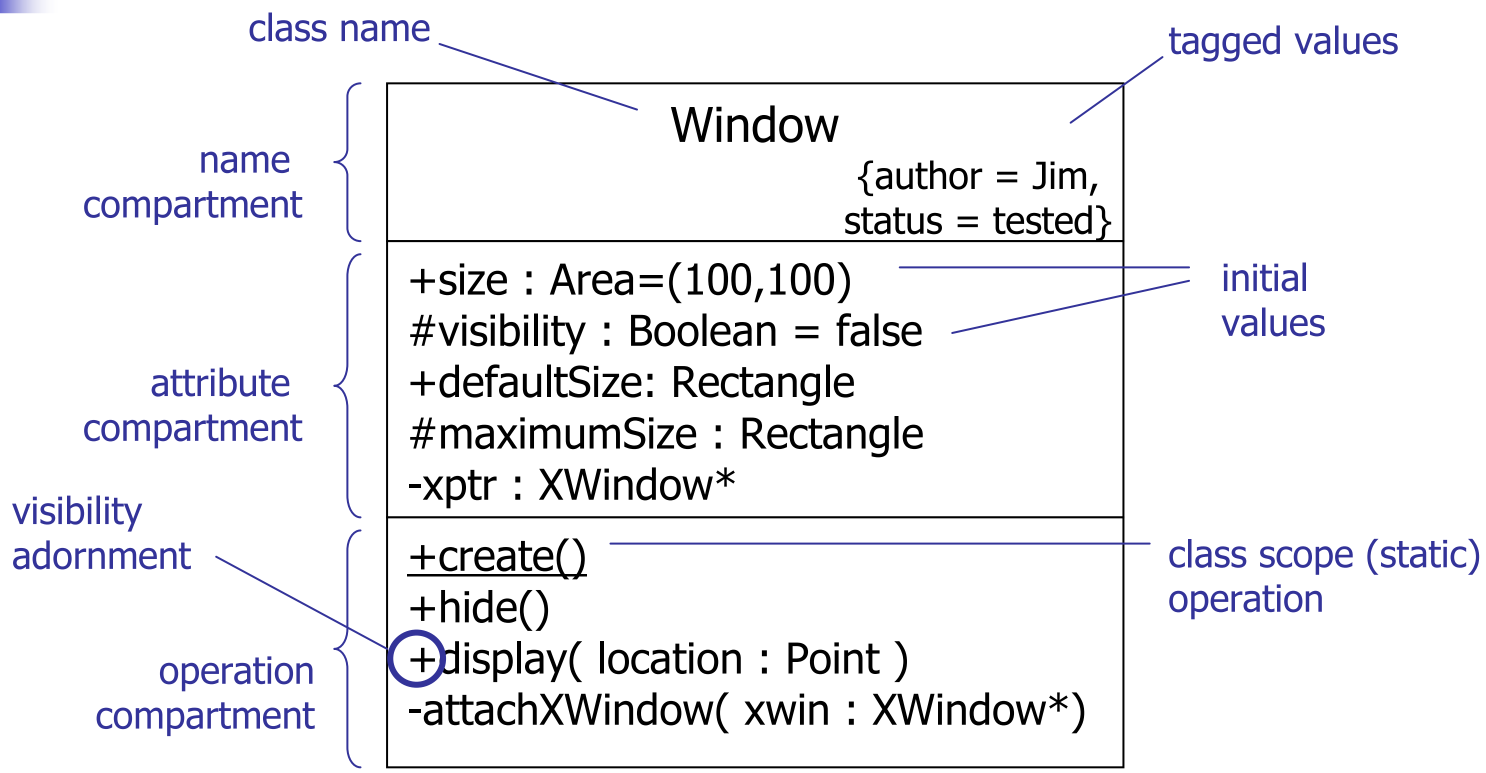
Classes and objects

- Objects are instances of classes
- Instantiation is the creation of new instances of model elements
- Most classes provide special operations called *constructors* to create instances of that class. These operations have class-scope i.e. they belong to the class itself rather than to objects of the class
- We will see instantiation used with other modelling elements later on



objects are instances of classes

UML class notation



class name

tagged values

name compartment

Window

{author = Jim, status = tested}

attribute compartment

+size : Area=(100,100)
#visibility : Boolean = false
+defaultSize: Rectangle
#maximumSize : Rectangle
-xptr : XWindow*

initial values

visibility adornment

operation compartment

+create()
+hide()
+display(location : Point)
-attachXWindow(xwin : XWindow*)

class scope (static) operation

- Classes are named in UpperCamelCase
- Use descriptive names that are nouns or noun phrases
- Avoid abbreviations!

Attribute compartment

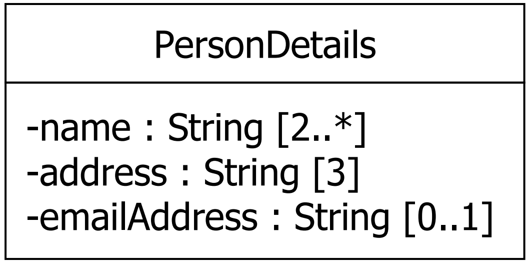
visibility name : type multiplicity = initialValue

mandatory

- Everything is optional except name
- initialValue is the value the attribute gets when objects of the class are instantiated
- Attributes are named in lowerCamelCase
  - Use descriptive names that are nouns or noun phrases
  - Avoid abbreviations
- Attributes may be prefixed with a stereotype and postfixed with a list of tagged values

Visibility

Symbol	Name	Semantics
+	public	Any element that can access the class can access any of its features with public visibility
-	private	Only operations within the class can access features with private visibility
#	protected	Only operations within the class, or within children of the class, can access features with protected visibility
~	package	Any element that is in the same package as the class, or in a nested subpackage, can access any of its features with package visibility



PersonDetails

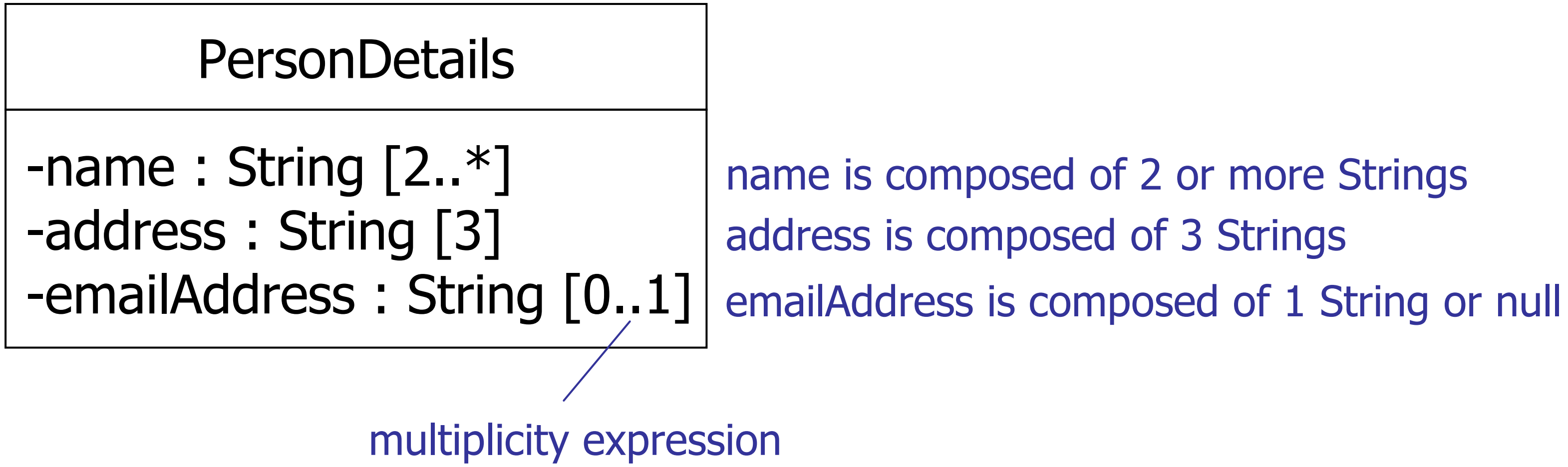
-name : String [2..*]
-address : String [3]
-emailAddress : String [0..1]

- You may ignore visibility in analysis
- In design, attributes usually have private visibility (encapsulation)

7.5.2.3

Multiplicity

- Multiplicity allows you to model collections of things
  - [0..1] means an that the attribute may have the value null



PersonDetails

-name : String [2..*]
-address : String [3]
-emailAddress : String [0..1]

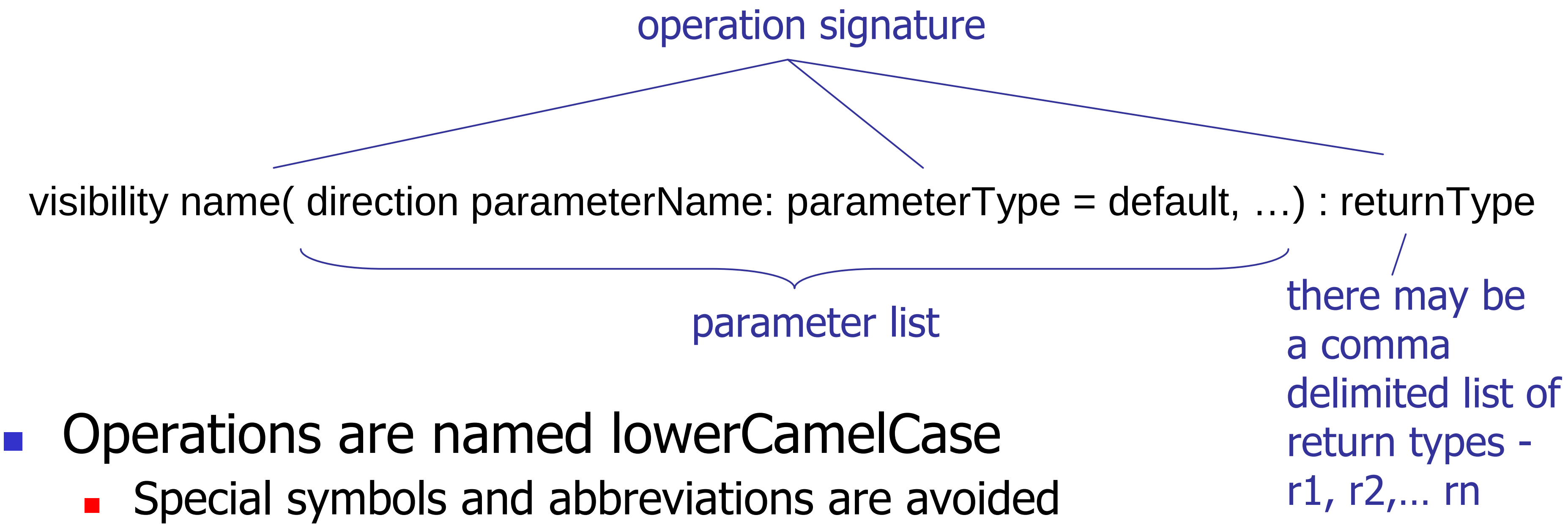
name is composed of 2 or more Strings

address is composed of 3 Strings

emailAddress is composed of 1 String or null

multiplicity expression

Operation compartment



operation signature

visibility name(direction parameterName: parameterType = default, …) : returnType

parameter list

there may be a comma delimited list of return types - r1, r2,… rn

- Operations are named lowerCamelCase
  - Special symbols and abbreviations are avoided
  - Operation names are usually a verb or verb phrase
- Operations may have more than one returnType
  - They can return multiple objects (see next slide)
- Operations may be prefixed with a stereotype and postfixed with a list of tagged values

Parameter direction

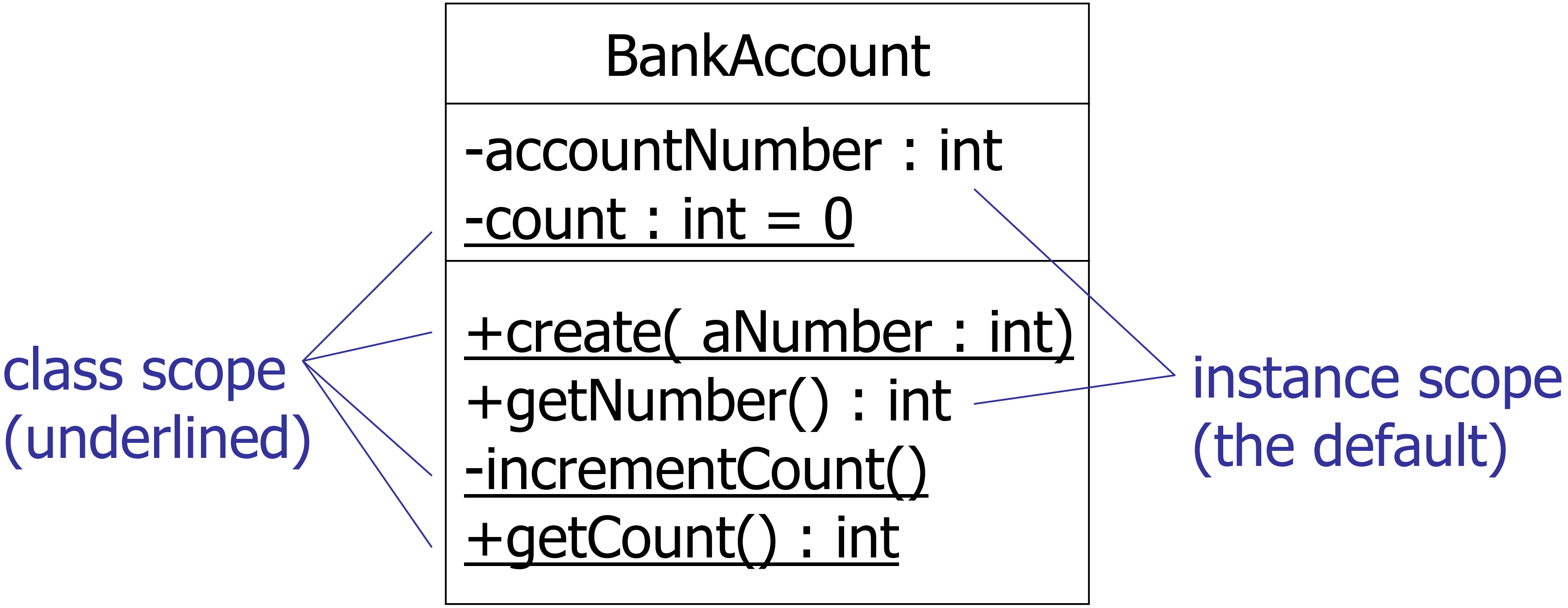
parameter direction	semantics
in	the parameter is an input to the operation. It is not changed by the operation. This is the default
out	the parameter serves as a repository for output from the operation
inout	the parameter is an input to the operation and it may be changed by the operation
return	the parameter is one of the return values of the operation. An alternative way of specifying return values

example of multiple return values:

```
maxMin( in a: int, in b:int, return maxValue:int return minValue:int )
…
max, min = maxMin( 5, 10 )
```

Scope

- There are two kinds of scope for attributes and operations:



BankAccount

-accountNumber : int
-count : int = 0

+create(aNumber : int)
+getNumber() : int
-incrementCount()
+getCount() : int

class scope (underlined)

instance scope (the default)

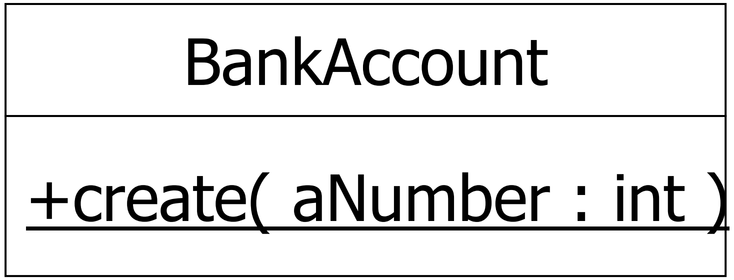
Instance scope vs. class scope

	instance scope	class scope
attributes	By default, attributes have instance scope	Attributes may be defined as class scope
	Every object of the class gets its own copy of the instance scope attributes	Every object of the class shares the same, single copy of the class scope attributes
	Each object may therefore have different instance scope attribute values	Each object will therefore have the same class scope attribute values
operations	By default, operations have instance scope	Operations may be defined as class scope
	Every invocation of an instance scope operation applies to a specific instance of the class	Invocation of a class scope operation does not apply to any specific instance of the class – instead, you can think of class scope operations as applying to the class itself
	You can't invoke an instance scope operation unless you have an instance of the class available. You can't use an instance scope operation of a class to create objects of that class, as you could never create the first object	You can invoke a class scope operation even if there is no instance of the class available – this is ideal for object creation operations

scope determines access

Object construction

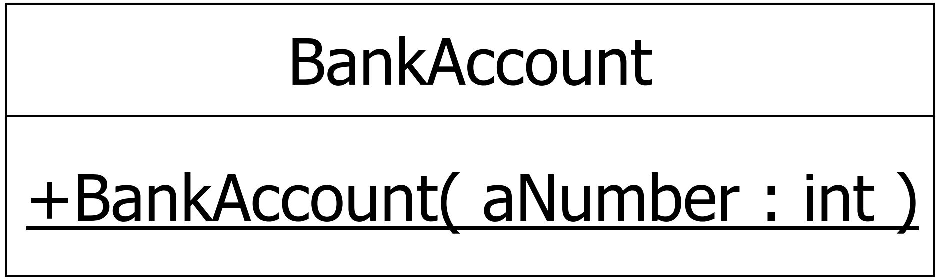
- How do we create instances of classes?
- Each class defines one or more class scope operations which are *constructors*. These operations create new instances of the class



BankAccount

+create(aNumber : int)

generic constructor name



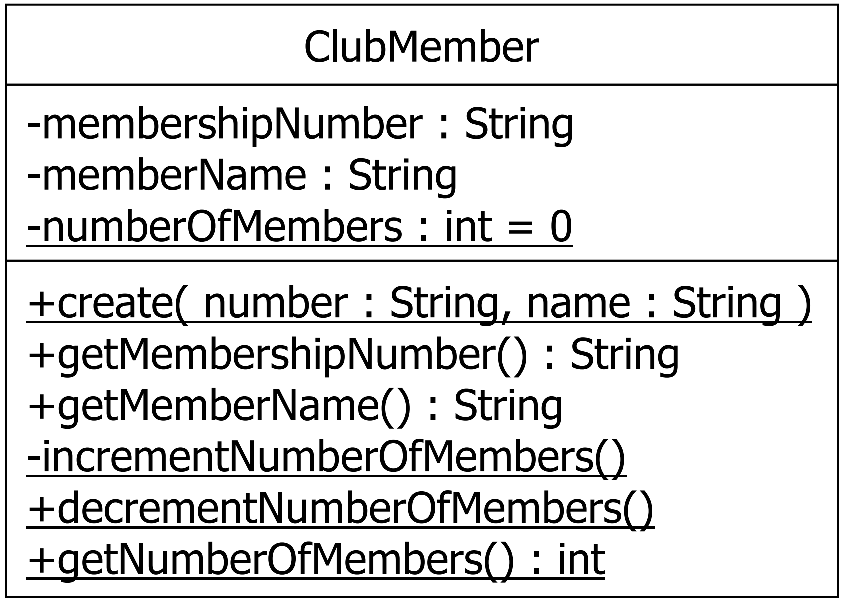
BankAccount

+BankAccount(aNumber : int)

Java/C++ standard

ClubMember class example

- Each ClubMember object has its own copy of the attribute membershipNumber
- The numberOfMembers attribute exists only once and is shared by all instances of the ClubMember class
- Suppose that in the create operation we increment numberOfMembers:
  - What is the value of count when we have created 3 account objects?



ClubMember

-membershipNumber : String
-memberName : String
-numberOfMembers : int = 0

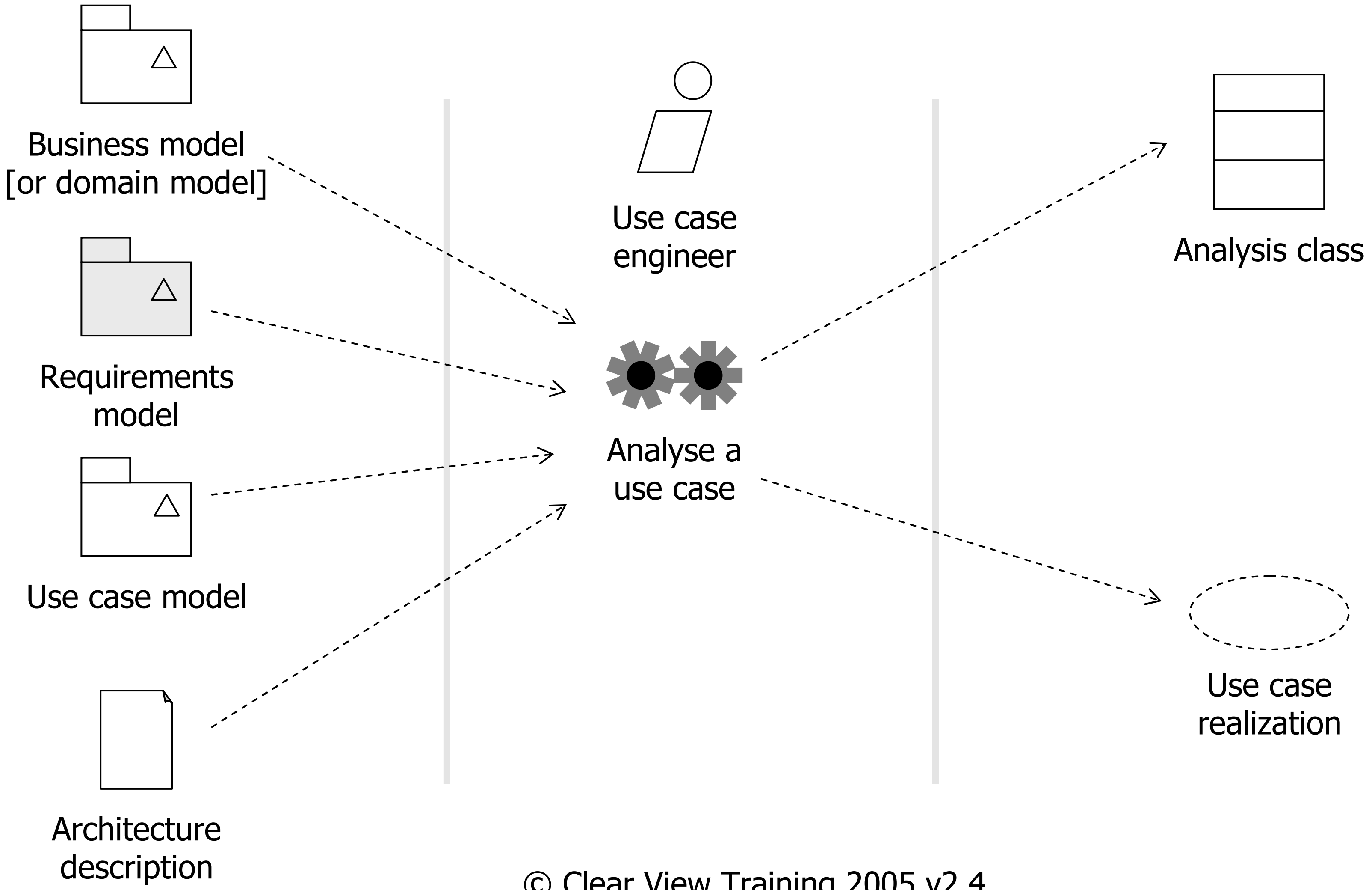
+create(number : String, name : String)
+getMembershipNumber() : String
+getMemberName() : String
-incrementNumberOfMembers()
+decrementNumberOfMembers()
+getNumberOfMembers() : int

Summary

- We have looked at objects and classes and examined the relationship between them
- We have explored the UML syntax for modelling classes including:
  - Attributes
  - Operations
- We have seen that scope controls access
  - Attributes and operations are normally instance scope
  - We can use class scope operations for constructor and destructors
  - Class scope attributes are shared by all objects of the class and are useful as counters

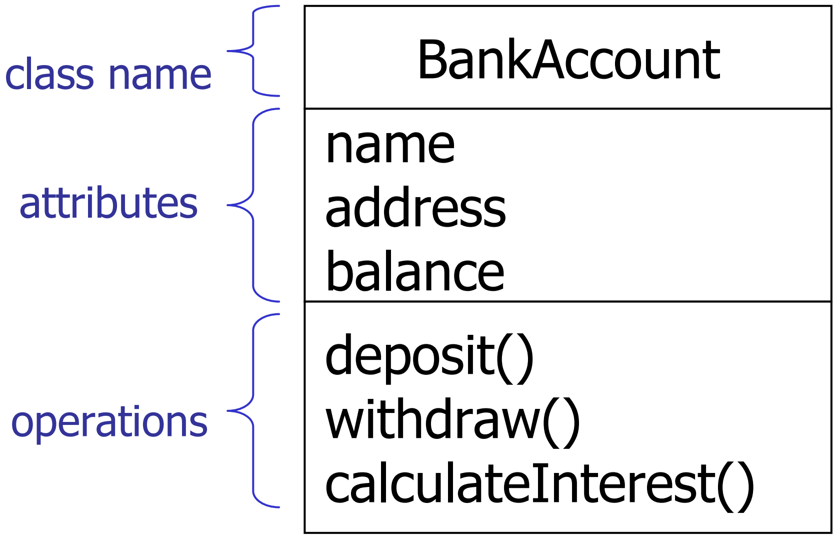
Analysis - finding analysis classes

Analyse a use case



What are Analysis classes?

- Analysis classes represent a crisp abstraction in the problem domain
  - They may ultimately be refined into one or more design classes
- All classes in the Analysis model should be Analysis classes
- Analysis classes have:
  - A very "high level" set of attributes. They *indicate* the attributes that the design classes *might* have.
  - Operations that specify at a high level the key services that the class must offer. In Design, they will become actual, implementable, operations.
- Analysis classes must map onto real-world business concepts



What makes a good analysis class?

- Its name reflects its intent
- It is a *crisp abstraction* that models one specific element of the problem domain
  - It maps onto a clearly identifiable feature of the problem domain
- It has *high cohesion*
  - Cohesion is the degree to which a class models a single abstraction
  - Cohesion is the degree to which the *responsibilities* of the class are semantically related
- It has *low coupling*
  - Coupling is the degree to which one class depends on others
- Rules of thumb:
  - 3 to 5 responsibilities per class
  - Each class collaborates with others
  - Beware many very small classes
  - Beware few but very large classes
  - Beware of "functoids"
  - Beware of "omnipotent" classes
  - Avoid deep inheritance trees

A *responsibility* is a contract or obligation of a class - it resolves into operations and attributes

Finding classes

- Perform noun/verb analysis on documents:
  - Nouns are candidate classes
  - Verbs are candidate *responsibilities*
- Perform CRC card analysis
  - A brainstorming technique using sticky notes
  - Useful for brainstorming, Joint Application Development (JAD) and Rapid Application development (RAD)
- With both techniques, beware of spurious classes:
  - Look for *synonyms* - different words that mean the same
  - Look for *homonyms* - the same word meaning different things
- Look for "hidden" classes!
  - Classes that don't appear as nouns or as cards

Noun/verb analysis procedure

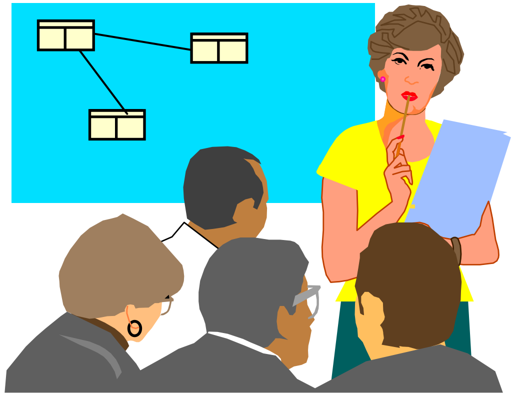
- Collect all of the relevant documentation
  - Requirements document
  - Use cases
  - Project Glossary
  - Anything else!
- Make a list of nouns and noun phrases
  - These are candidate classes or attributes
- Make a list of verbs and verb phrases
  - These are candidate responsibilities
- Tentatively assign attributes and responsibilities to classes

CRC card procedure

Class Name: BankAccount	
Responsibilities:	Collaborators:
Maintain balance	Bank

things the class does

things the class works with



- Class, Responsibilities and Collaborators
- Separate information collection from information analysis
  - Part 1: Brainstorm
    - *All* ideas are good ideas in CRC analysis
    - Never argue about something – write it down and analyse it later!
  - Part 2: Analyse information - consolidate with noun/verb

Other sources of classes

- Physical objects
- Paperwork, forms etc.
  - Be careful with this one – if the existing business process is very poor, then the paperwork that supports it might be irrelevant
- Known interfaces to the outside world
- Conceptual entities that form a cohesive abstraction e.g. LoyaltyProgramme

Summary

- We've looked at what constitutes a well-formed analysis class
- We have looked at two analysis techniques for finding analysis classes:
    - Noun verb analysis of use cases, requirements, glossary and other relevant documentation
    - CRC analysis

Analysis - relationships

What is a relationship?

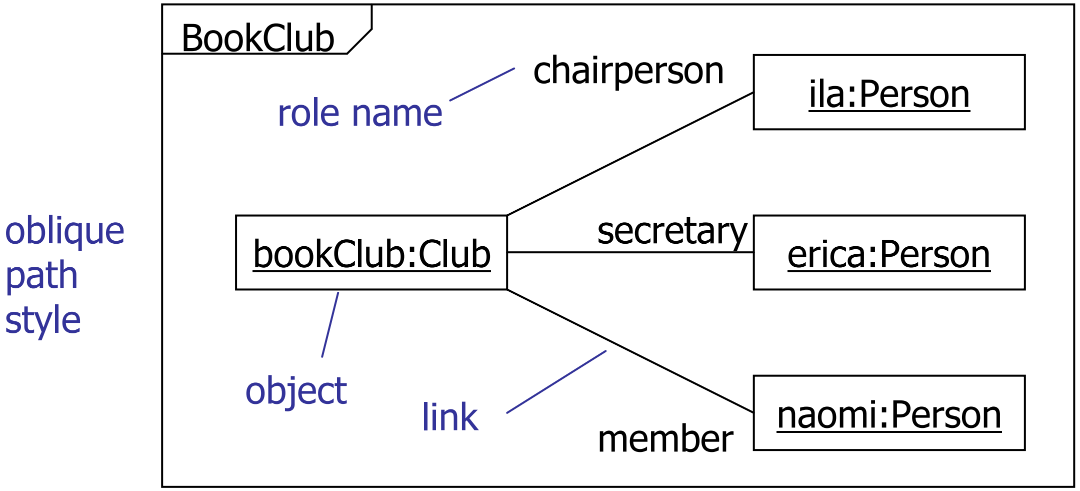
- A *relationship* is a connection between modelling elements
- In this section we'll look at:
  - *Links* between objects
  - *Associations* between classes
    - aggregation
    - composition
    - association classes

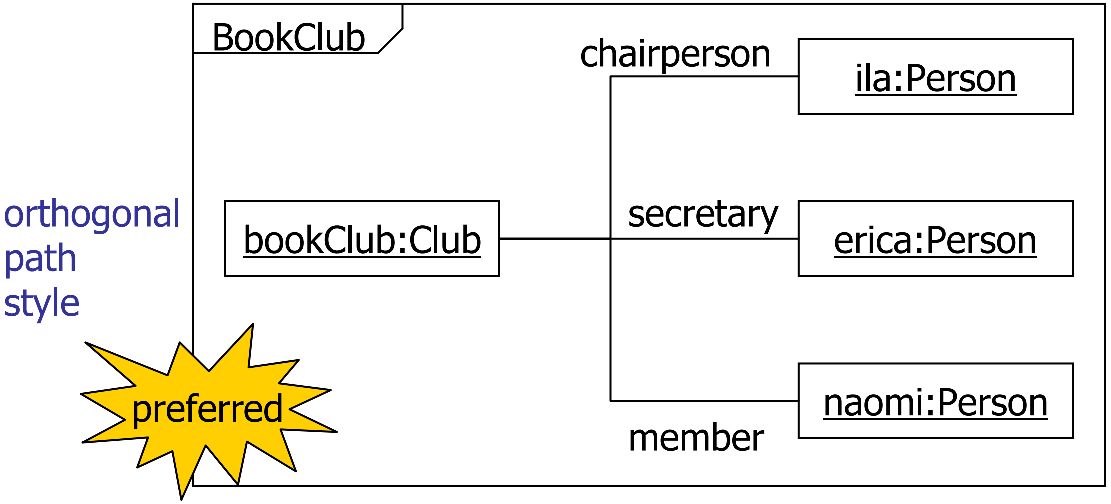
What is a link?

- Links are connections between objects
  - Think of a link as a telephone line connecting you and a friend. You can send messages back and forth using this link
- Links are the way that objects communicate
  - Objects send messages to each other via links
  - Messages invoke operations
- OO programming languages implement links as object references or pointers. These are unique handles that refer to specific objects
  - When an object has a reference to another object, we say that there is a *link* between the objects

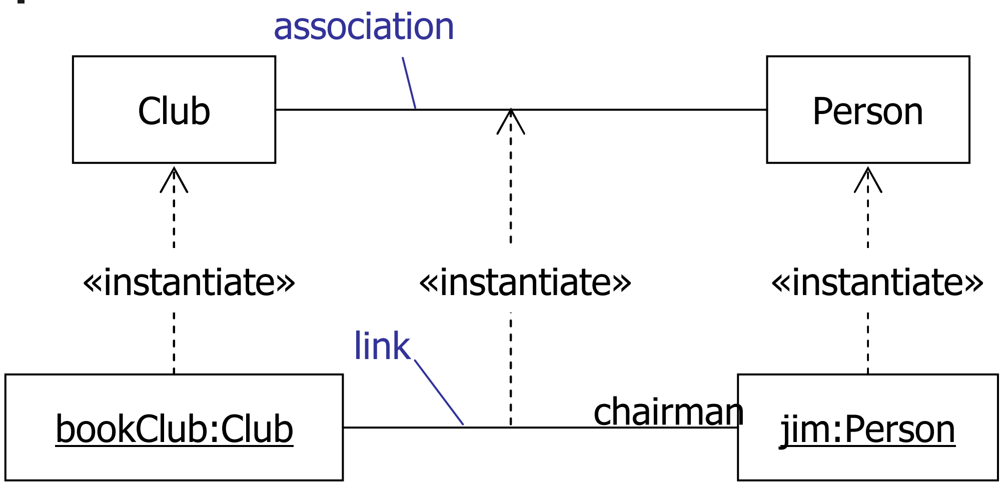
Object diagrams

- Paths in UML diagrams (lines to you and me!) can be drawn as orthogonal, oblique or curved lines
- We can combine paths into a tree *if* each path has the same properties





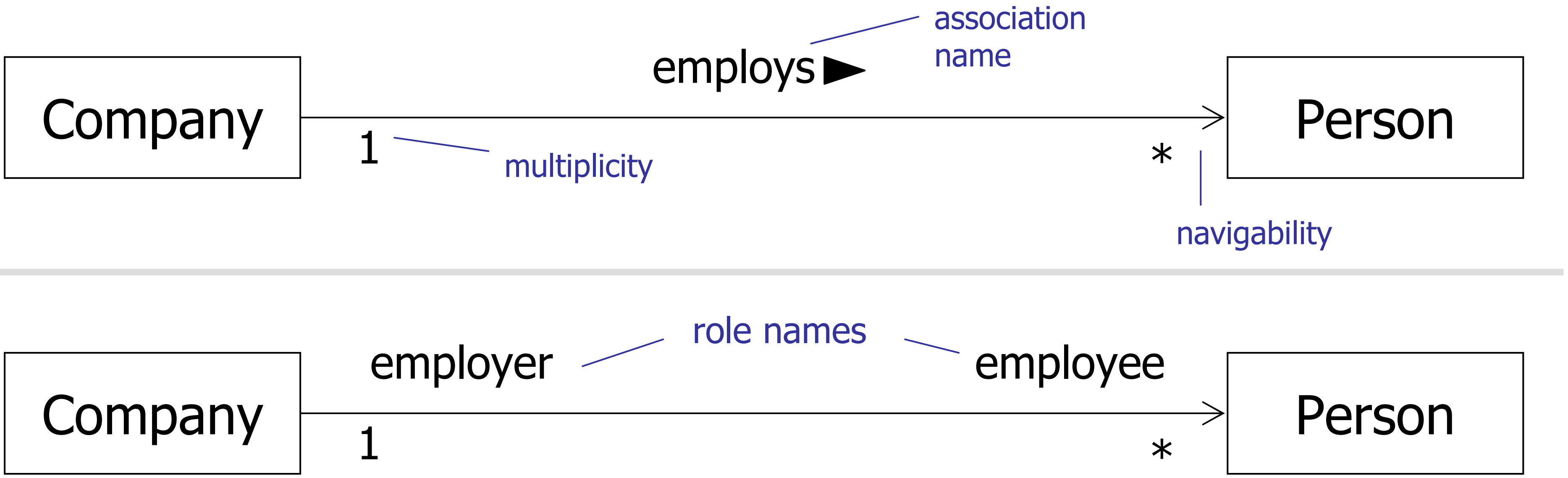
What is an association?



links instantiate associations

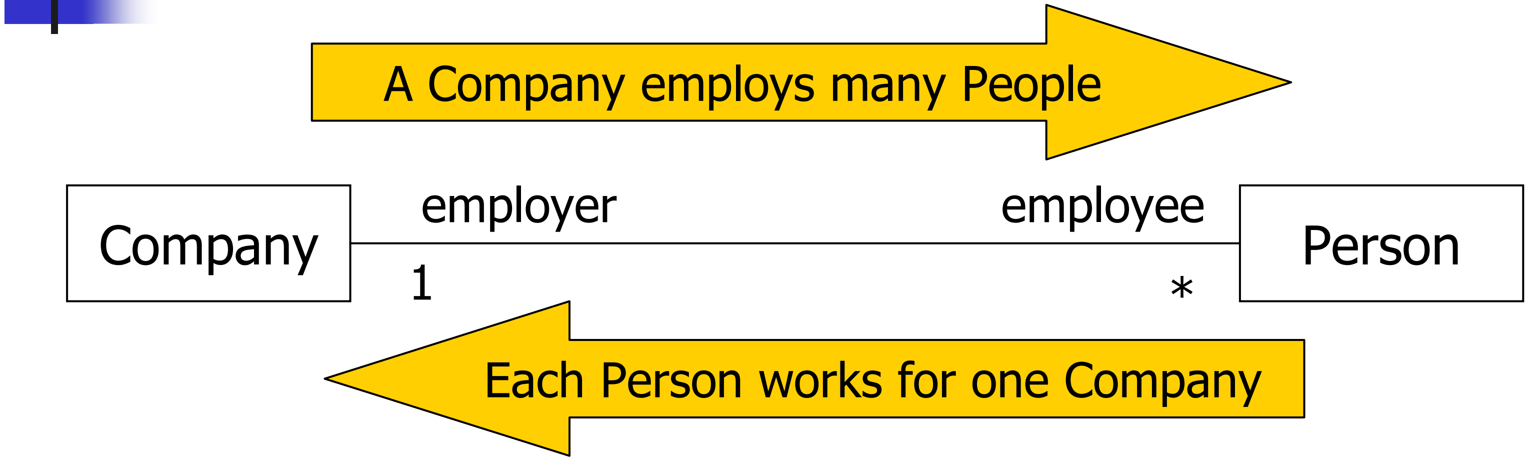
- Associations are relationships between classes
- Associations between classes indicate that there are links between objects of those classes
- A link is an instantiation of an association just as an object is an instantiation of a class

Association syntax



- An association can have role names *or* an association name. It's bad style to have both. The black triangle indicates the direction in which the
- association name is read: "Company employs many Person(s)"

Multiplicity



A Company employs many People

Company

employer

1

employee

*

Person

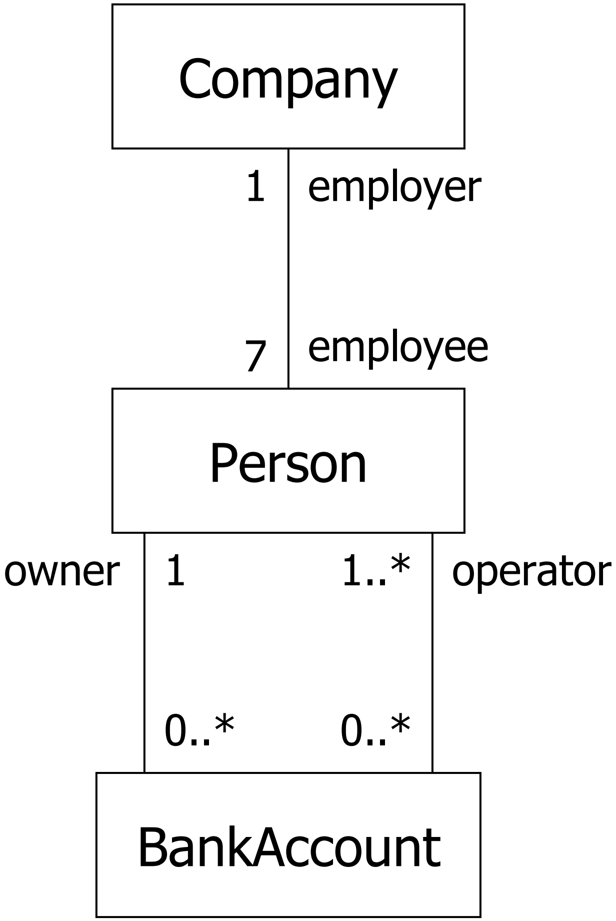
Each Person works for one Company

- Multiplicity is a constraint that specifies the number of objects that can participate in a relationship at *any point in time*
- If multiplicity is not explicitly stated in the model then it is undecided – *there is no default multiplicity*

multiplicity syntax: minimum..maximum	
0..1	zero or 1
1	exactly 1
0..*	zero or more
*	zero or more
1..*	1 or more
1..6	1 to 6

Multiplicity exercise

- How many
  - Employees can a Company have?
  - Employers can a Person have?
  - Owners can a BankAccount have?
  - Operators can a BankAccount have?
  - BankAccounts can a Person have?
  - BankAccounts can a Person operate?

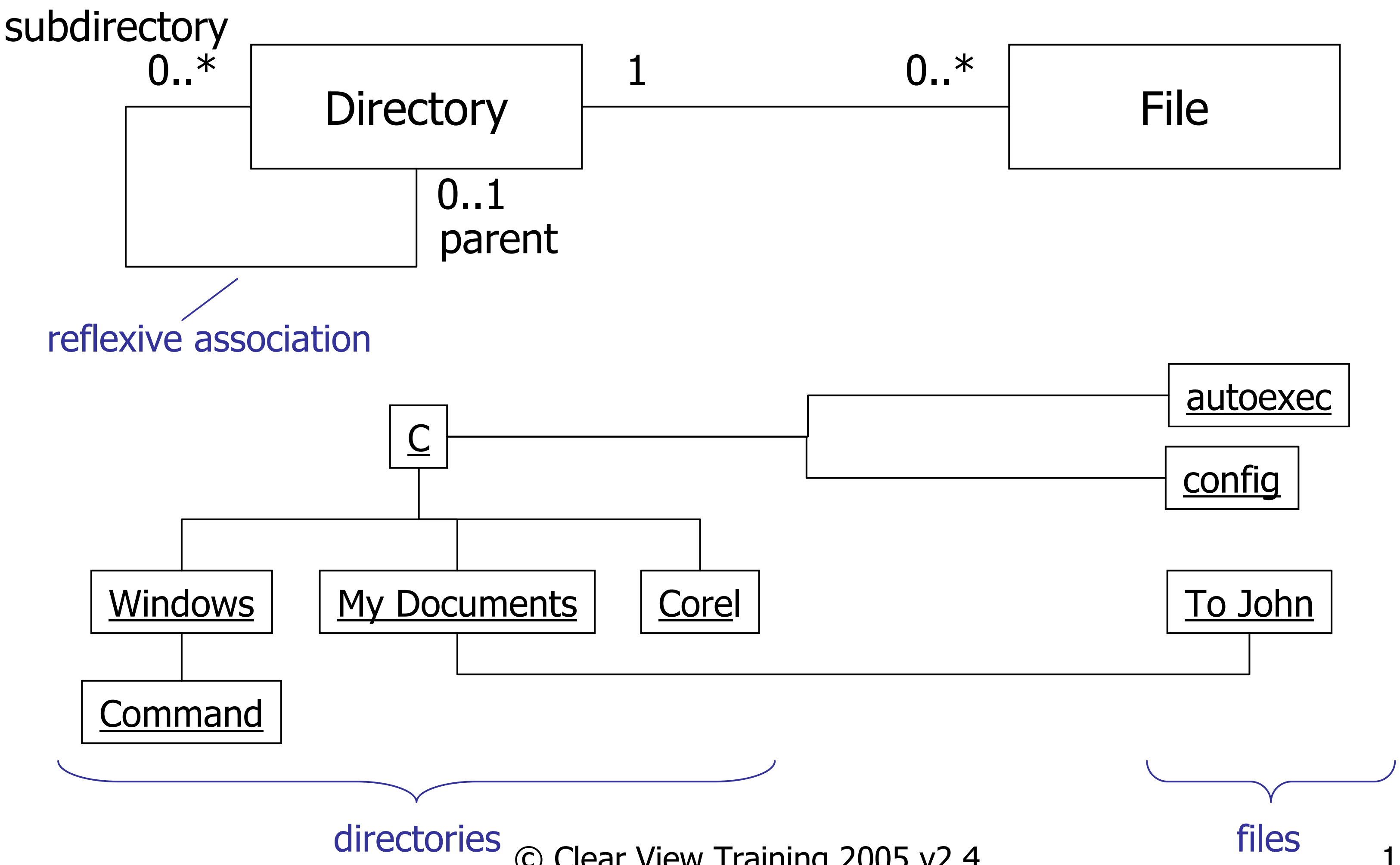


Exercise

- Model a computer file system. Here are the minimal facts you need:
  - The basic unit of storage is the file
  - Files live in directories
  - Directories can contain other directories
- Use your own knowledge of a specific file system (e.g. Windows 95 or UNIX) to build a model

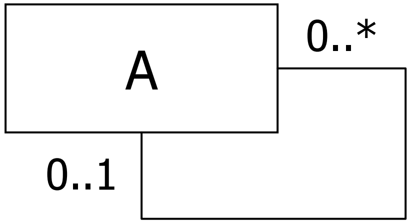
Hint: a class can have an association to itself!

Reflexive associations

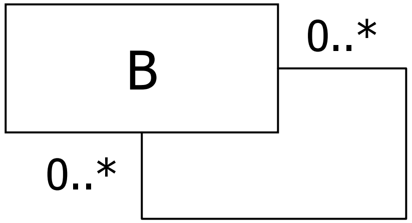


Hierarchies and networks

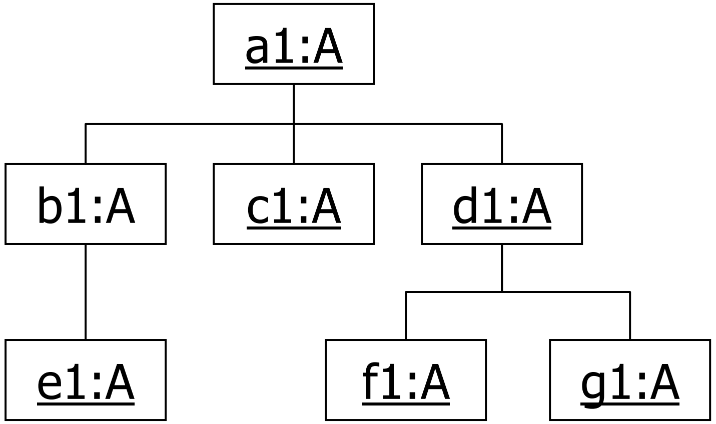
<table>
<tr><th>hierarchy</th><th>network</th></tr>
<tr><td>



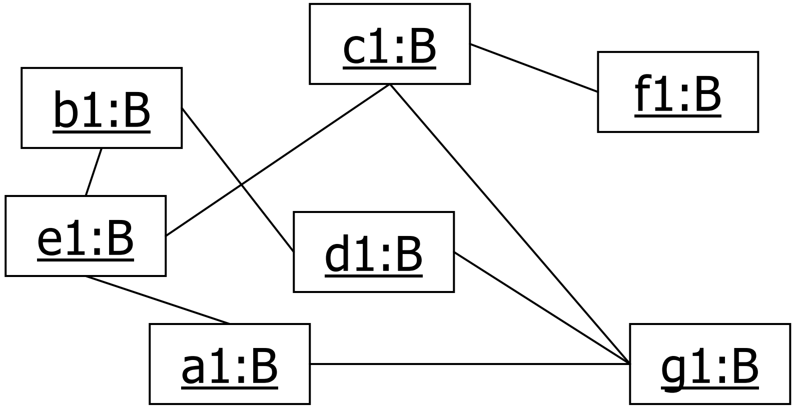
</td><td>



</td></tr>
<tr><td>



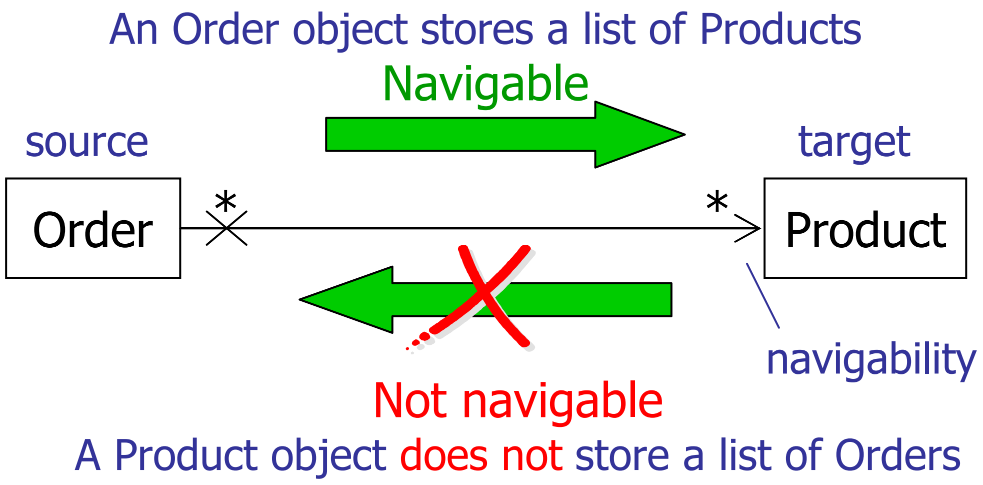
</td><td>



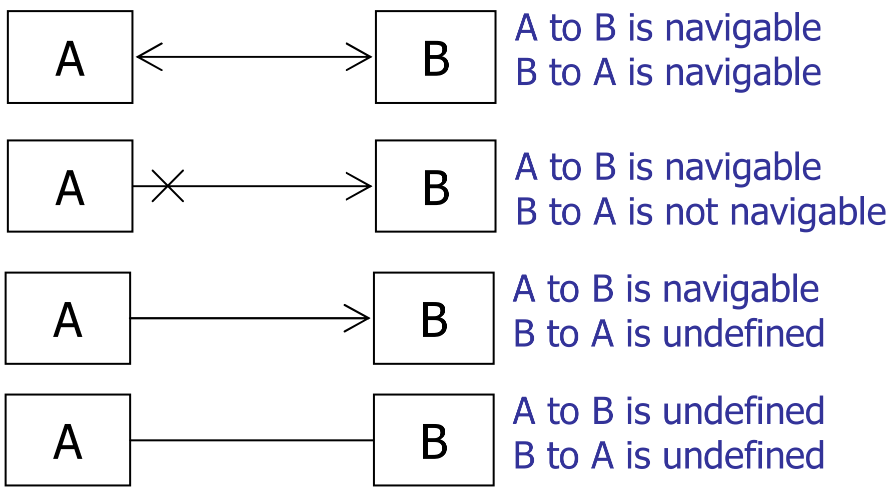
</td></tr>
<tr><td>an an association hierarchy, each object has <em>zero or one</em> object directly above it</td><td>in an association network, each object has zero or many objects directly above it</td></tr>
</table>

Navigability

- Navigability indicates that it is possible to traverse from an object of the *source* class to objects of the *target* class
  - Objects of the source class may reference objects of the target class using the role name
- Even if there is *no* navigability it might still be possible to traverse the relationship via some indirect means. However the computational cost of the traversal might be very high



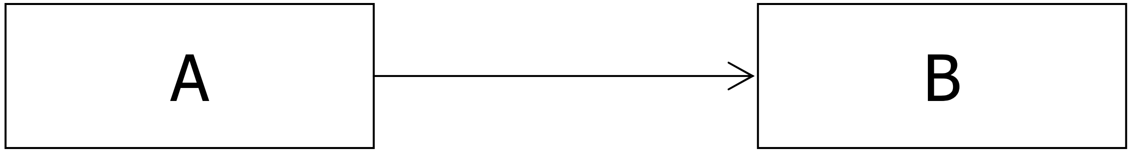
An Order object stores a list of Products
Navigable
source
target
Order
*
*
Product
navigability
Not navigable
A Product object does not store a list of Orders



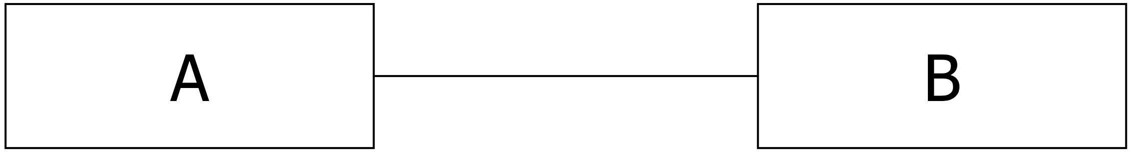
Navigability - standard practice

- Strict UML 2 navigability can clutter diagrams so the UML standard suggests three possible modeling idioms:
  - Show navigability explicitly on diagrams
  - Omit all navigability from diagrams
  - Omit crosses from diagrams
    - bi-directional associations have no arrows
    - unidirectional associations have a single arrow
    - you can't show associations that are not navigable in either direction (not useful anyway!)

standard practice

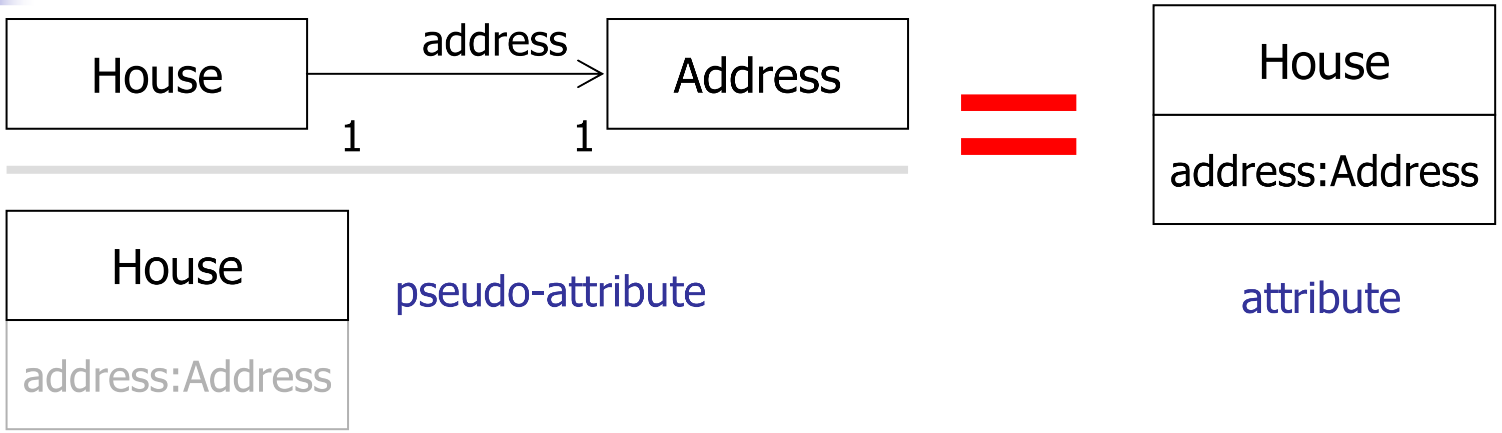


A to B is navigable
B to A is not navigable



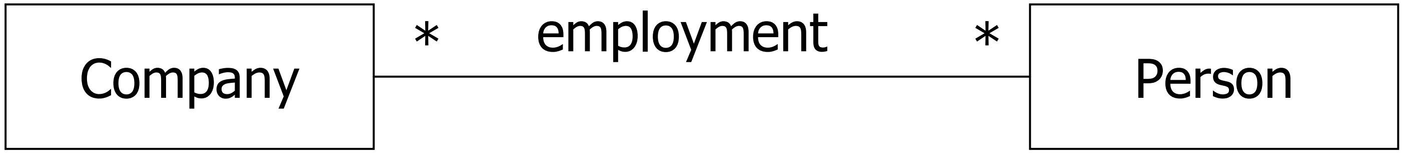
A to B is navigable
B to A is navigable

Associations and attributes



- If a navigable relationship has a role name, it is as though the source class has a pseudo-attribute whose attribute name is the role name and whose attribute type is the target class
- Objects of the source class can refer to objects of the target class using this pseudo-attribute
- Use associations when:
  - The target class is an important part of the model
  - The target class is a class that you have designed yourself and which must be shown on the model
- Use attributes when:
  - The target class is *not* an important part of the model e.g. a primitive type such as number, string etc.
  - The target class is just an implementation detail such as a bought-in component or a library component e.g. Java.util.Vector (from the Java standard libraries)

Association classes

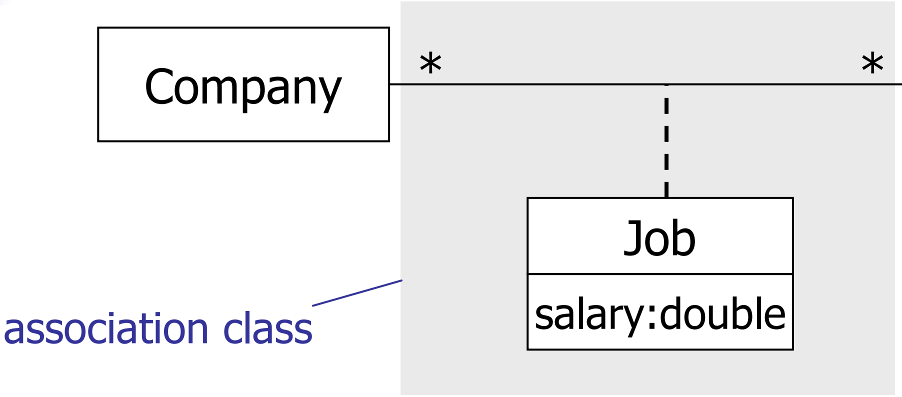


Each Person object can work for many Company objects.
Each Company object can employ many Person objects.
When a Person object is employed by a Company object, the Person has a salary.

But where do we record the Person's salary?

- Not on the Person class - there is a different salary for each employment
- Not on the Company class - different Person objects have different salaries
- The salary is a property of the employment relationship itself
  - every time a Person object is employed by a Company object, there is a salary

Association class syntax

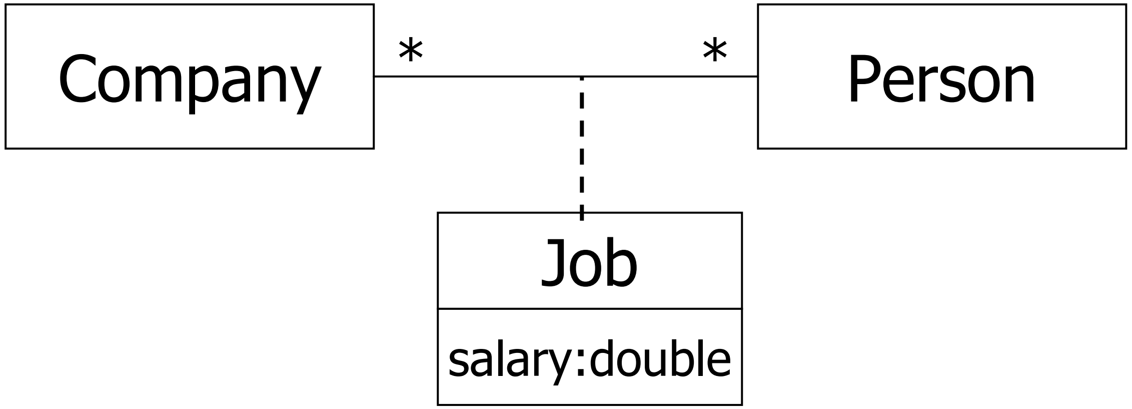


the association class consists of the class, the association and the dashed line

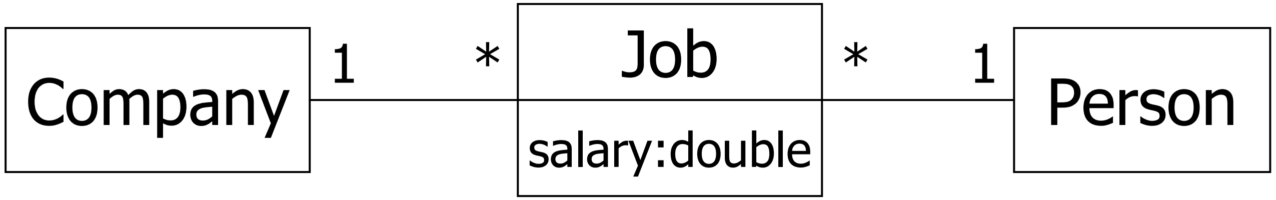
- We model the association itself as an association class. One instance of this class exists for each link between a Person object and a Company object
  - Instances of the association class are links that have attributes and operations
  - Can only use association classes when there is *one unique link* between two specific objects. This is because the identity of links is determined exclusively by the identities of the objects on the ends of the link
- We can place the salary and any other attributes or operations which are really features of the association into this class

Using association classes

If we use an association class, then a particular Person can have only *one* Job with a particular Company

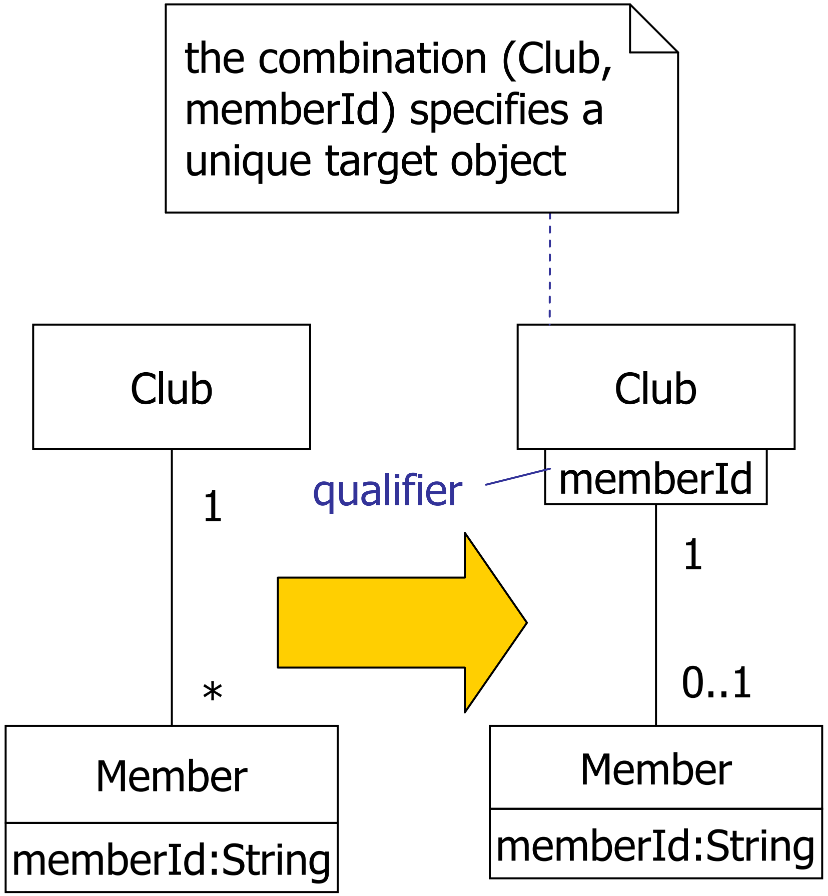


If, however a particular Person can have *multiple* jobs with the same Company, then we must use a *reified association*



Qualified associations

- Qualified associations reduce an n to many association to an n to 1 association by specifying a unique object (or group of objects) from the set
- They are useful to show how we can look up or navigate to specific objects
- Qualifiers usually refer to an attribute on the target class



Summary

- In this section we have looked at:
  - Links – relationships between objects
  - Associations – relationships between classes
    - role names
    - multiplicity
    - navigability
    - association classes
    - qualified associations

Analysis - dependencies

What is a dependency?

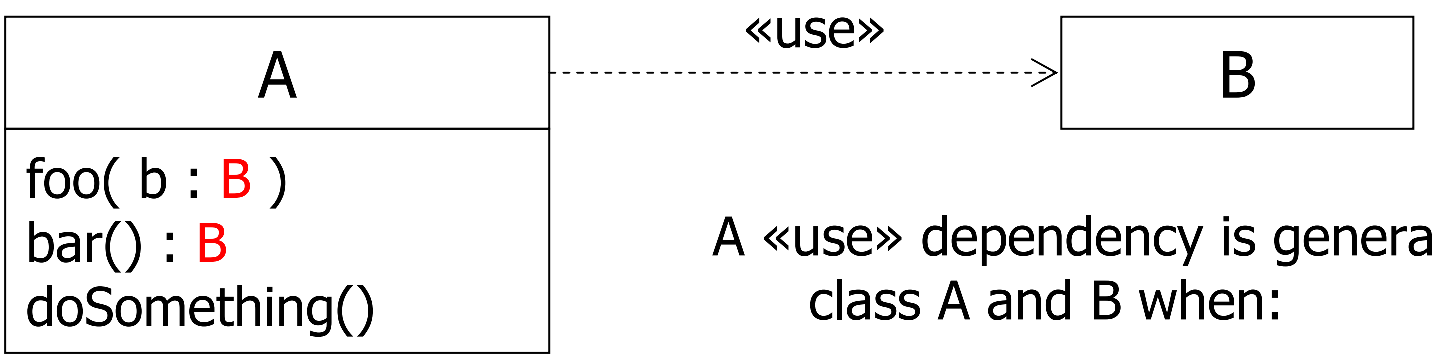
- "A dependency is a relationship between two elements where a change to one element (the supplier) may affect or supply information needed by the other element (the client)". In other words, the client *depends* in some way on the supplier
  - Dependency is really a catch-all that is used to model several different types of relationship. We've already seen one type of dependency, the «instantiate» relationship
- Three types of dependency:
  - Usage - the client uses some of the services made available by the supplier to implement its own behavior – this is the most commonly used type of dependency
  - Abstraction - a shift in the level of abstraction. The supplier is more abstract than the client
  - Permission - the supplier grants some sort of permission for the client to access its contents – this is a way for the supplier to control and limit access to its contents

Usage dependencies

- «use» - the client makes use of the supplier to implement its behaviour
- «call» - the client operation invokes the supplier operation
- «parameter» - the supplier is a parameter of the client operation
- «send» - the client (an operation) sends the supplier (a signal) to some unspecified target
- «instantiate» - the client is an instance of the supplier

«use» - example

the stereotype is often omitted



```
A :: doSomething()
{
   B myB = new B();
   …
}
```

A «use» dependency is generated between class A and B when:

1) An operation of class A needs a parameter of class B

2) An operation of class A returns a value of class B

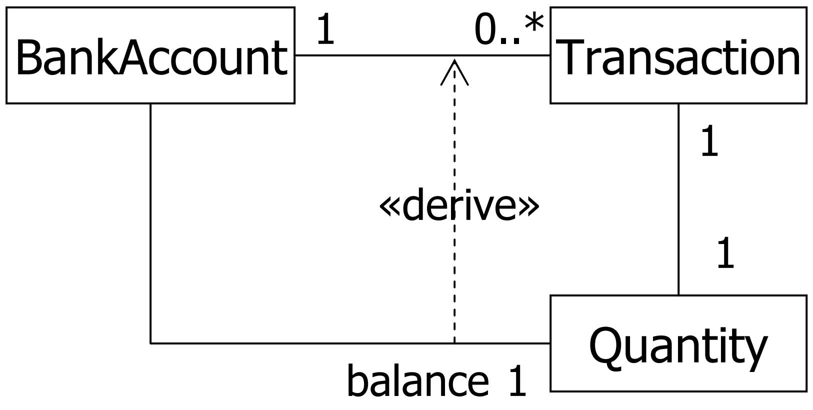
3) An operation of class A uses an object of class B somewhere in its implementation

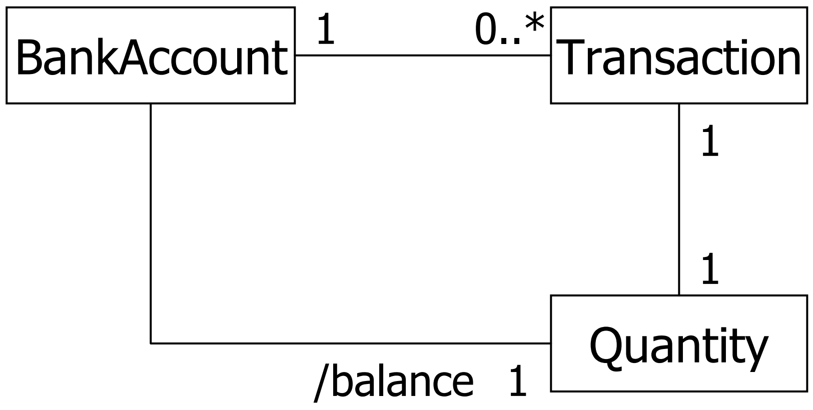
Abstraction dependencies

- «trace» - the client and the supplier represent the same concept but at different points in development
- «substitute» - the client may be substituted for the supplier at runtime. The client and supplier must realize a common contract. Use in environments that *don't* support specialization/generalization
- «refine» - the client represents a fuller specification of the supplier
- «derive» - the client may be derived from the supplier. The client is logically redundant, but may appear for implementation reasons

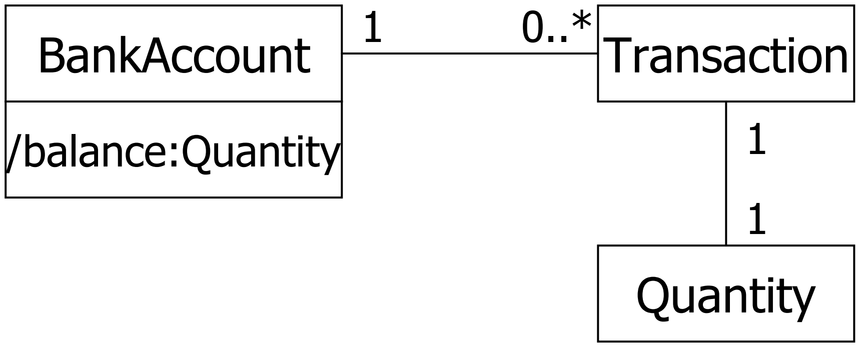
9.5.2.4

«derive» - example





This example shows three possible ways to express a «derive» dependency



Permission dependencies

- «access»
  - The public contents of the supplier package are added as private elements to the namespace of the client package
- «import»
  - The public contents of the supplier package are added as public elements to the namespace of the client package
- «permit»
  - The client element has access to the supplier element despite the declared visibility of the supplier

Summary

- Dependency
  - The weakest type of association
  - A catch-all
- There are three types of dependency:
  - Usage
  - Abstraction
  - Permission

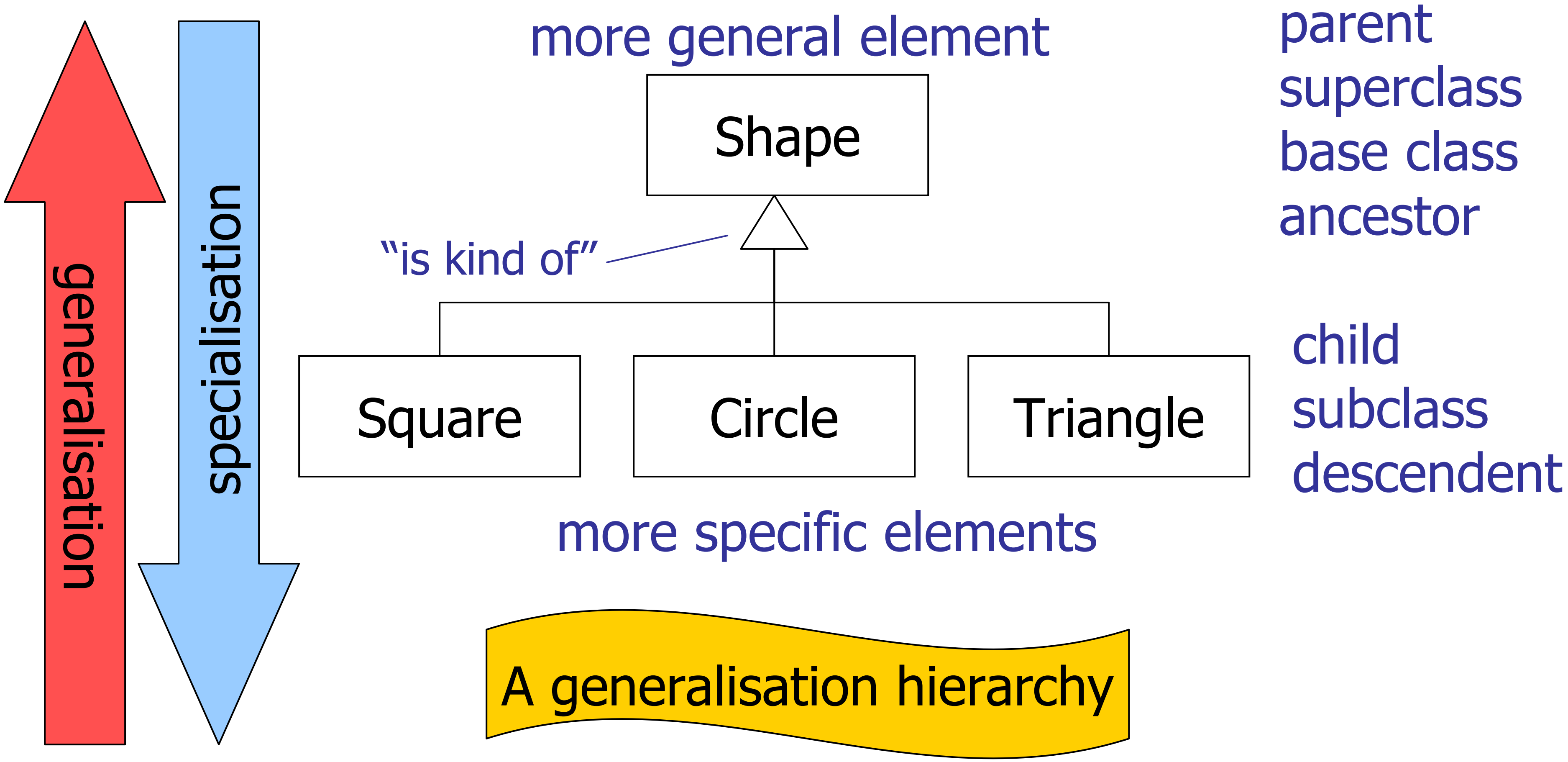
Analysis – inheritance and polymorphism

Generalisation

- A relationship between a more general element and a more specific element
- The more specific element is entirely consistent with the more general element but contains more information
- An instance of the more specific element may be used where an instance of the more general element is expected

Substitutability Principle

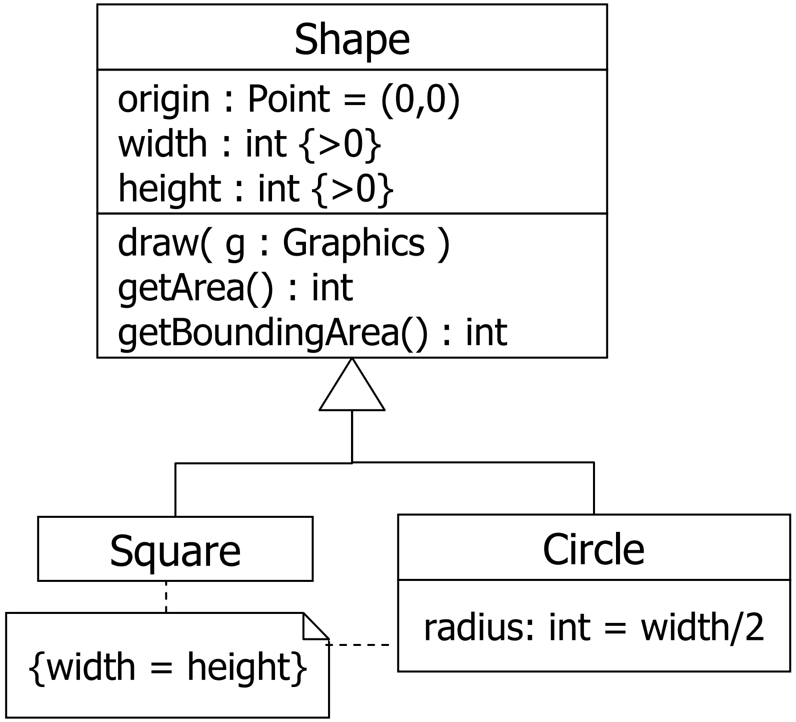
Example: class generalisation



Class inheritance

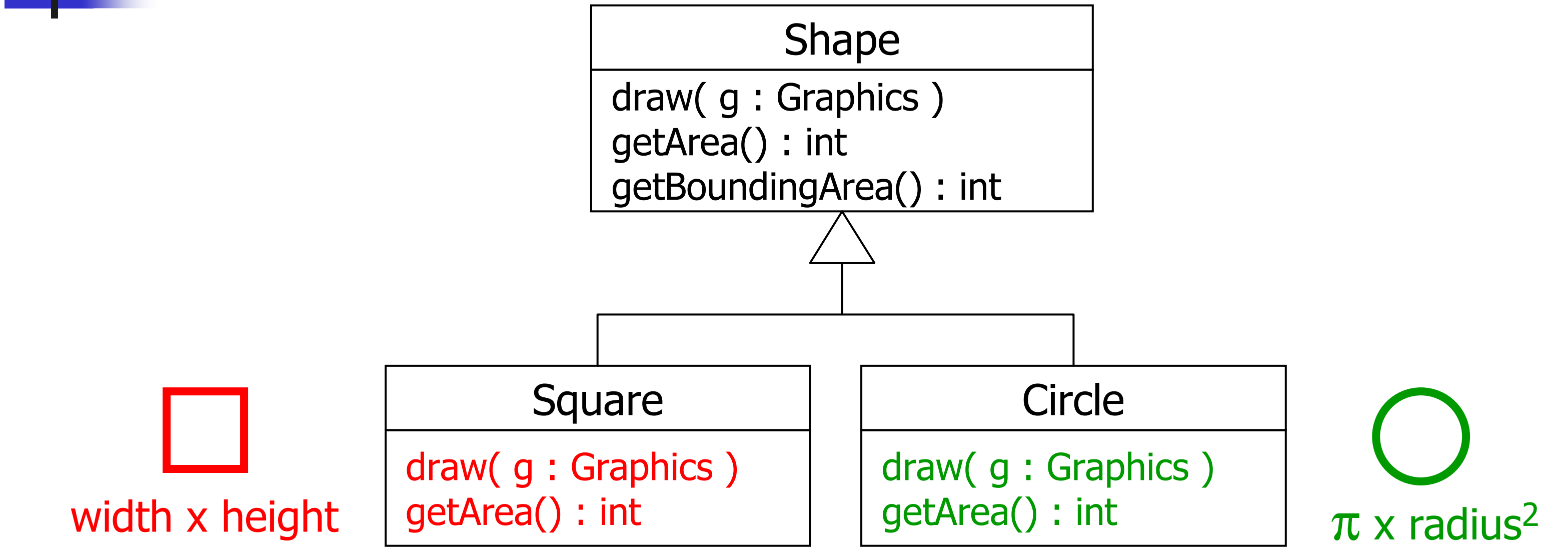
- Subclasses inherit *all* features of their superclasses:
  - attributes
  - operations
  - relationships
  - stereotypes, tags, constraints
- Subclasses can add new features
- Subclasses can override superclass operations
- We can use a subclass instance *anywhere* a superclass instance is expected

Substitutability Principle



But what's wrong with these subclasses

Overriding



Shape

draw(g : Graphics)
getArea() : int
getBoundingArea() : int

Square

draw(g : Graphics)
getArea() : int

Circle

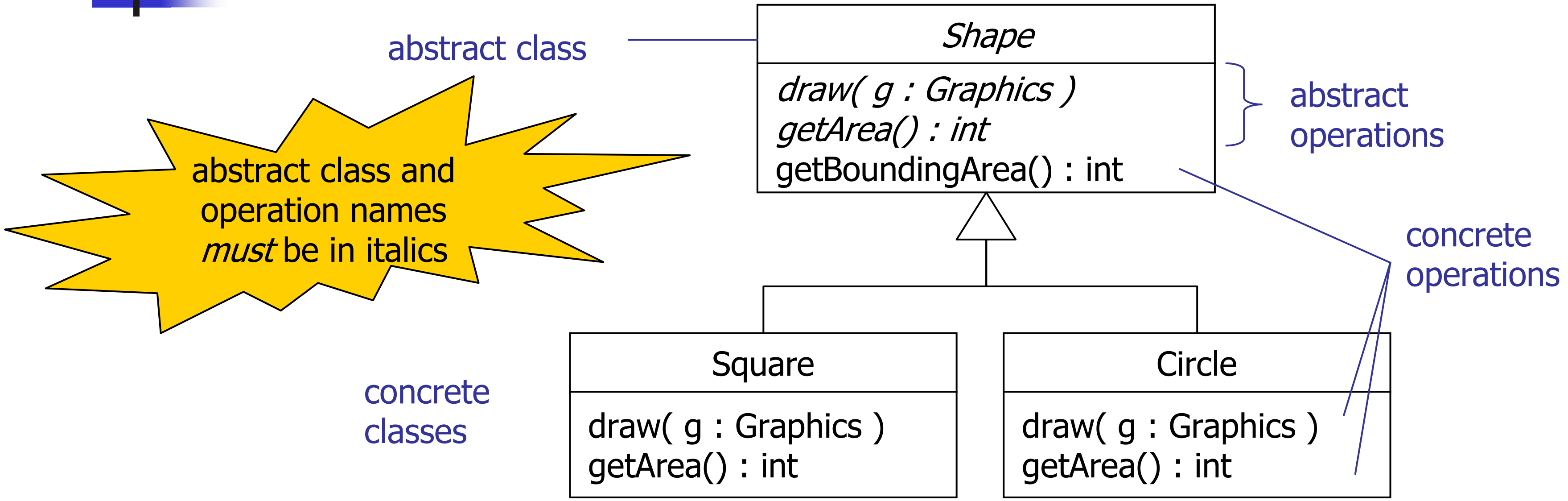
draw(g : Graphics)
getArea() : int

width x height

π x radius^2

- Subclasses often need to *override* superclass behaviour
- To override a superclass operation, a subclass must provide an operation with the same signature
  - The operation signature is the operation name, return type and types of all the parameters
  - The names of the parameters don’t count as part of the signature

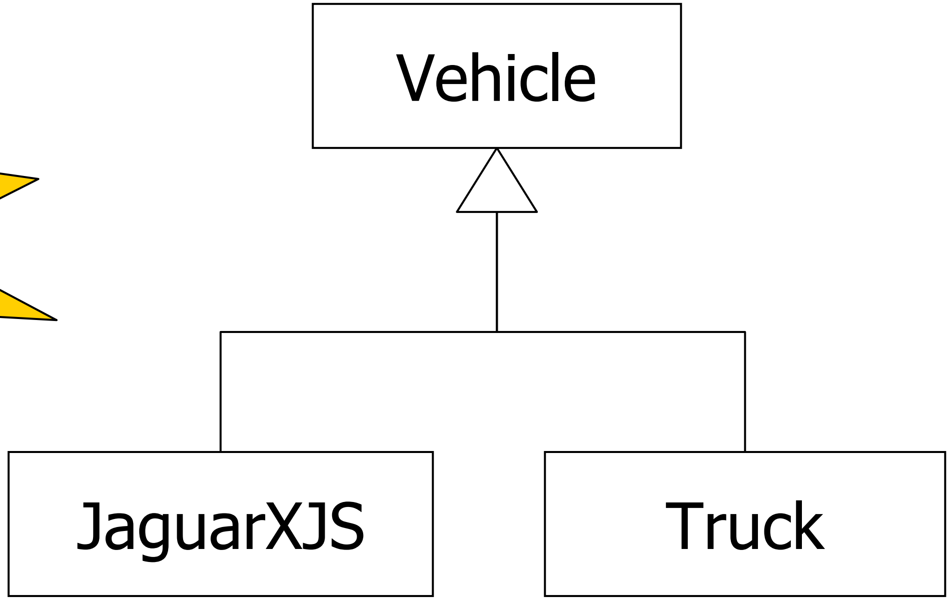
Abstract operations & classes



- We can't provide an implementation for *Shape :: draw(g : Graphics)* or for *Shape :: getArea() : int* because we don't know how to draw or calculate the area for a "shape"!
- Operations that lack an implementation are *abstract operations*
- A class with any abstract operations *can't* be instantiated and is therefore an *abstract class*

Exercise

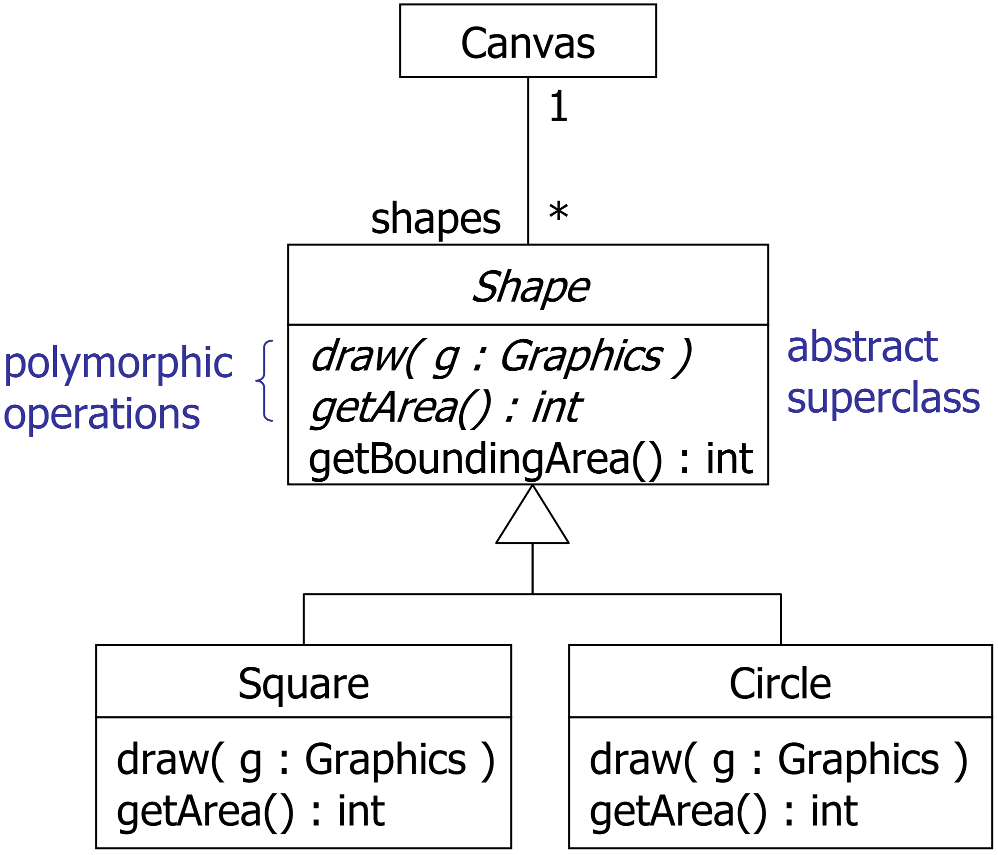
what's wrong with this model?



Polymorphism

- Polymorphism = "many forms"
  - A polymorphic operation has many implementations
  - Square and Circle provide implementations for the polymorphic operations *Shape::draw()* and *Shape::getArea()*
- All concrete subclasses of Shape *must* provide concrete draw() and getArea() operations because they are abstract in the superclass
  - For draw() and getArea() we can treat all subclasses of Shape in a similar way - we have defined a contract for Shape subclasses

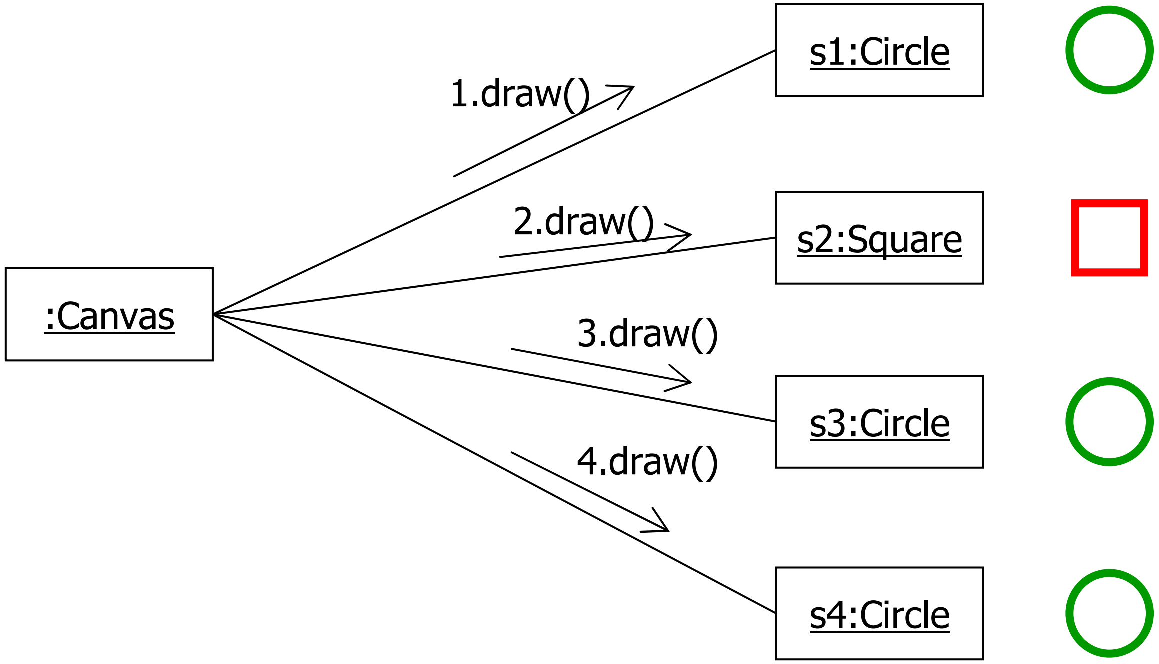
A Canvas object has a collection of *Shape* objects where each *Shape* may be a Square or a Circle



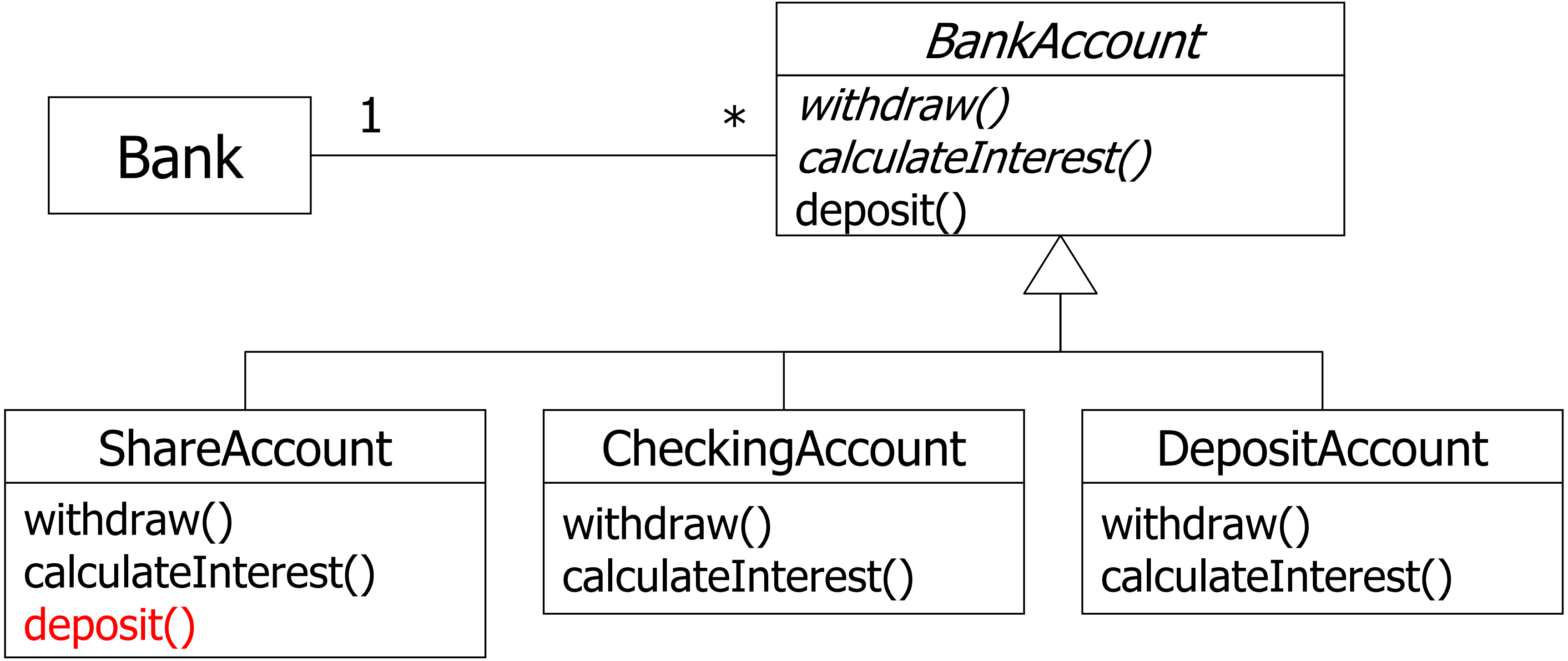
concrete subclasses

What happens?

- Each class of object has its own implementation of the draw() operation
- On receipt of the draw() message, each object invokes the draw() operation specified by its class
- We can say that each object "decides" how to interpret the draw() message based on its class



BankAccount example



- We have overridden the deposit() operation even though it is *not* abstract. This is perfectly legal, and quite common, although it is generally considered to be bad style and should be avoided if possible

Summary

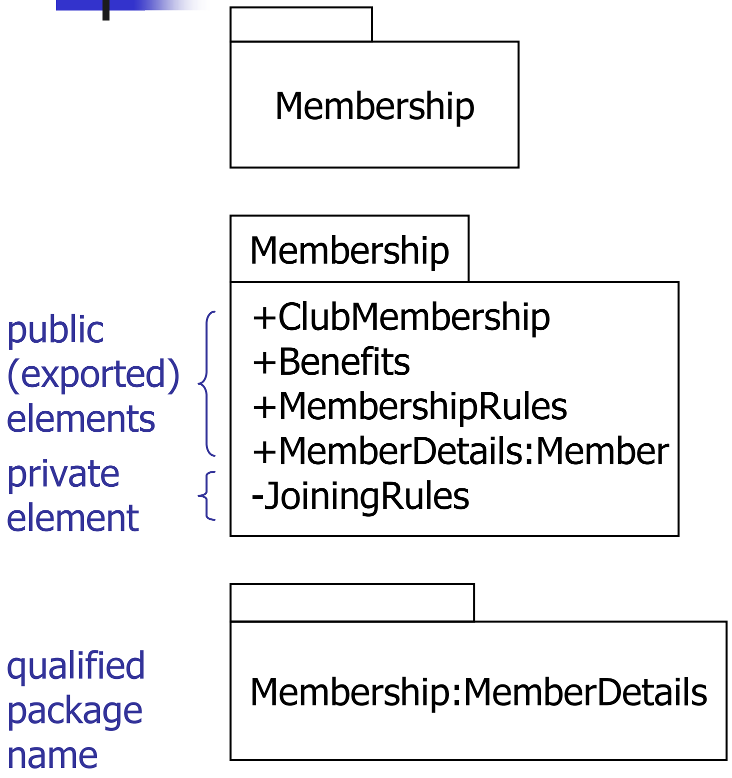
- Subclasses:
  - inherit *all* features from their parents including constraints and relationships
  - may add *new* features, constraints and relationships
  - may *override* superclass operations
- A class that can't be instantiated is an abstract class

Analysis - packages

Analysis packages

- A package is a *general purpose* mechanism for organising model elements into groups
  - Group semantically related elements
  - Define a “semantic boundary” in the model
  - Provide units for parallel working and configuration management
  - Each package defines an *encapsulated namespace* i.e. all names must be unique within the package
- In UML 2 a package is a purely logical grouping mechanism
  - Use components for physical grouping
- *Every* model element is owned by exactly one package
  - A hierarchy rooted in a top level package that can be stereotyped «topLevel»
- Analysis packages contain:
  - Use cases, analysis classes, use case realizations, analysis packages

Package syntax



Membership

Membership

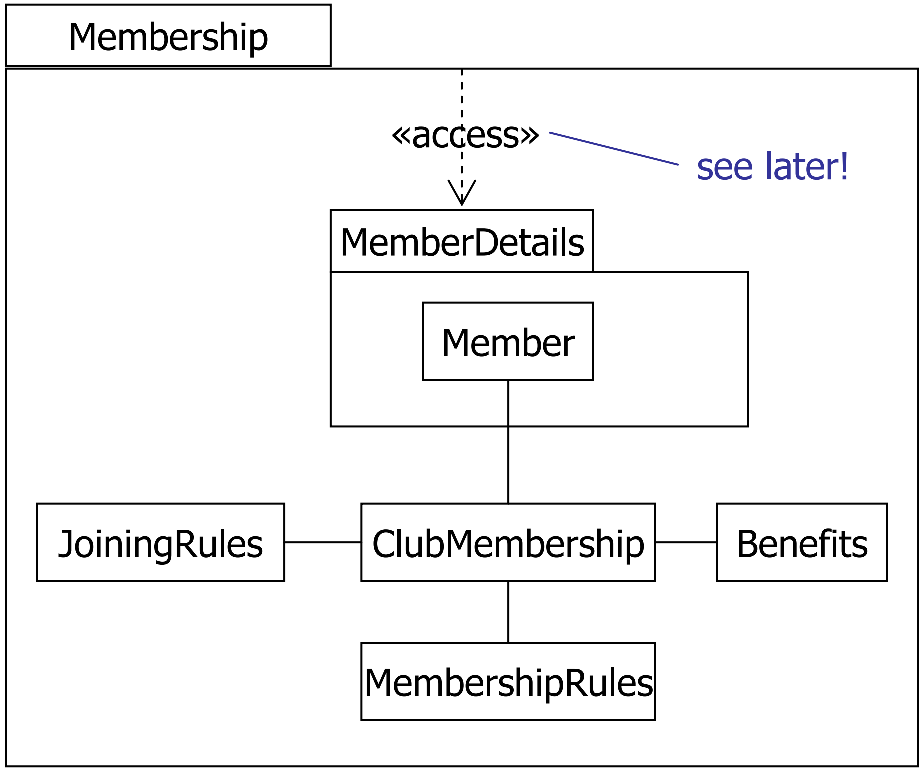
+ClubMembership
+Benefits
+MembershipRules
+MemberDetails:Member
-JoiningRules

public (exported) elements

private element

Membership:MemberDetails

qualified package name



Membership

«access»

see later!

MemberDetails

Member

JoiningRules

ClubMembership

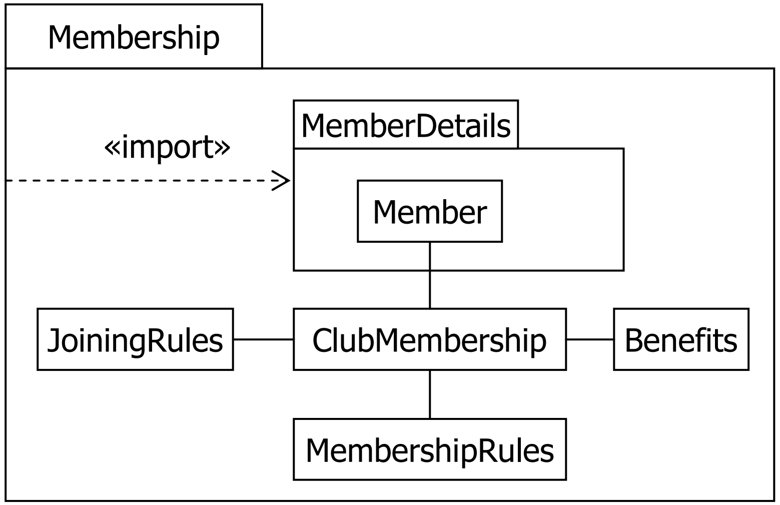
Benefits

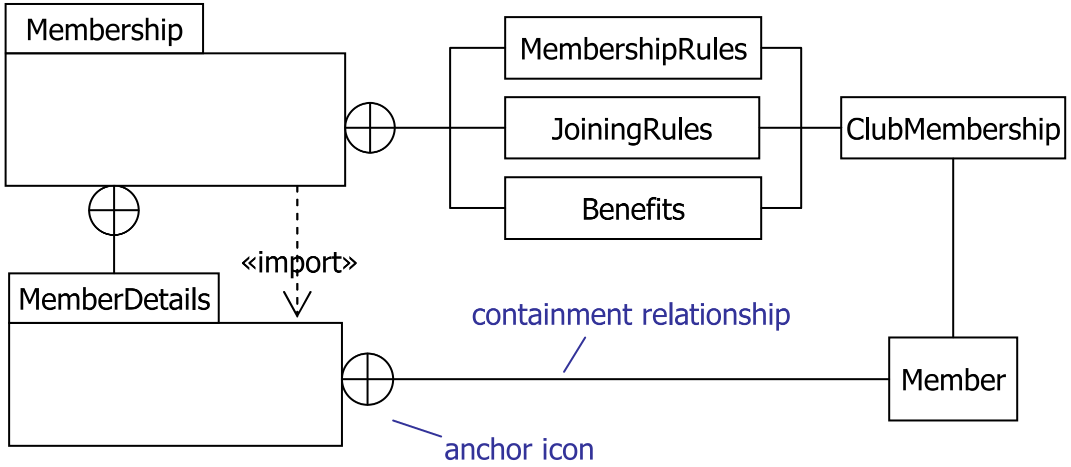
MembershipRules

standard UML 2 package stereotypes	
«framework»	A package that contains model elements that specify a reusable architecture
«modelLibrary»	A package that contains elements that are intended to be reused by other packages Analogous to a class library in Java, C# etc.

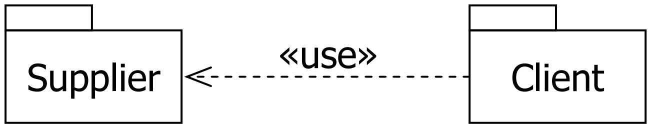
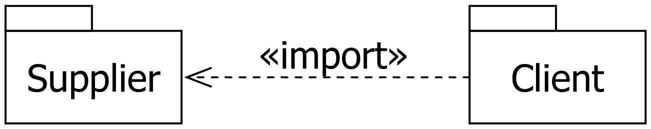
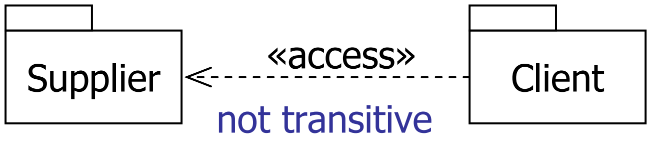
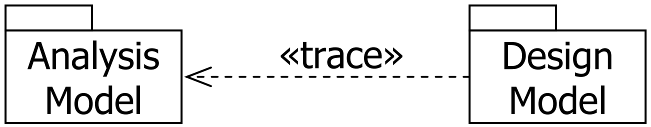
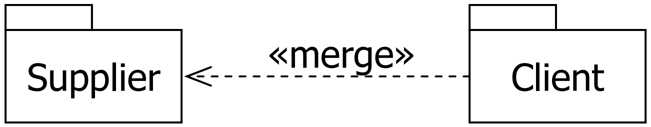
Nested packages

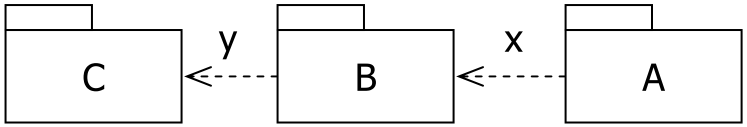
- If an element is visible within a package then it is visible within all nested packages
  - e.g. Benefits is visible within MemberDetails
- Show containment using nesting or the containment relationship
- Use «access» or «import» to merge the namespace of nested packages with the parent namespace





Package dependencies

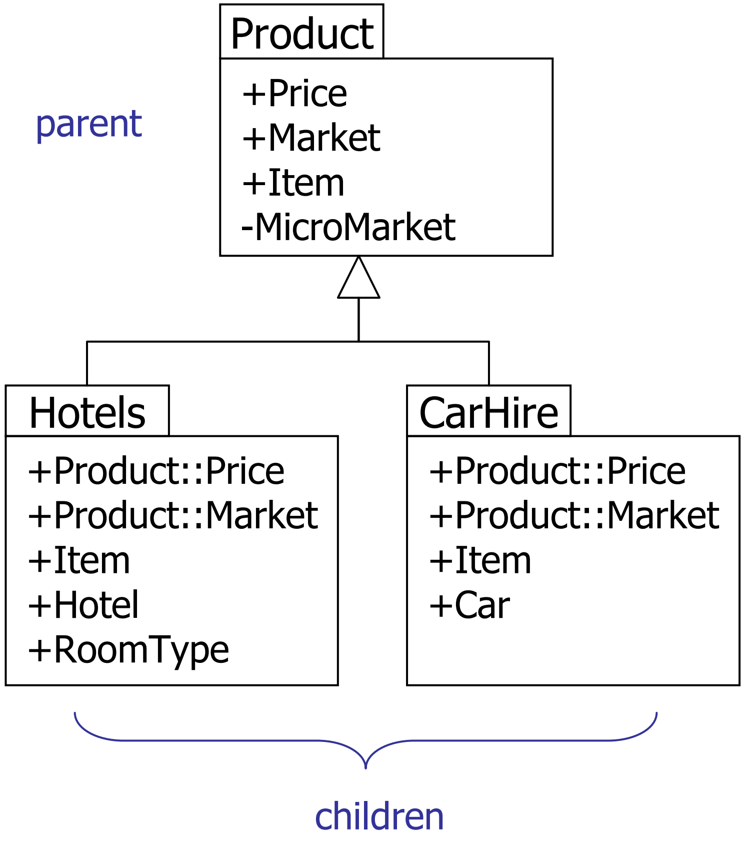
dependency	semantics
 Supplier «use» Client	An element in the client uses an element in the supplier in some way. The client depends on the supplier. Transitive.
 Supplier «import» Client	Public elements of the supplier namespace are added as public elements to the client namespace. Transitive.
 Supplier «access» Client not transitive	Public elements of the supplier namespace are added as private elements to the client namespace. Not transitive.
 Analysis Model «trace» Design Model	«trace» usually represents an historical development of one element into another more refined version. It is an extra-model relationship. Transitive.
 Supplier «merge» Client	The client package merges the public contents of its supplier packages. This is a complex relationship only used for metamodeling - you can ignore it.



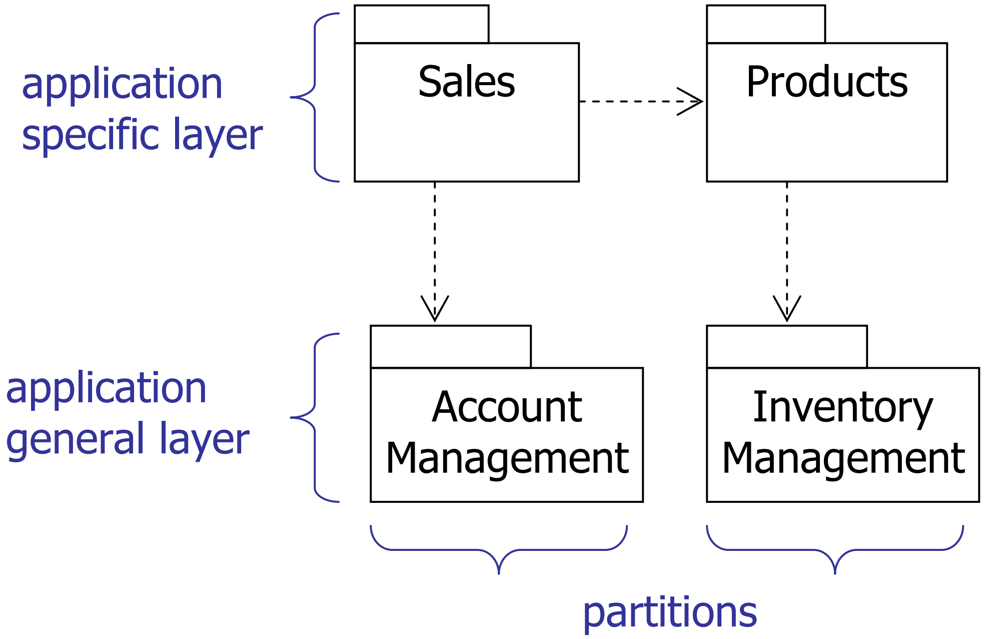
transitivity - if dependencies x and y are transitive, there is an *implicit* dependency between A and C

Package generalisation

- The more specialised child packages *inherit* the public and protected elements in their parent package
- Child packages may *override* elements in the parent package. Both Hotels and CarHire packages override Product::Item
- Child packages may *add* new elements. Hotels adds Hotel and RoomType, CarHire adds Car



Architectural analysis



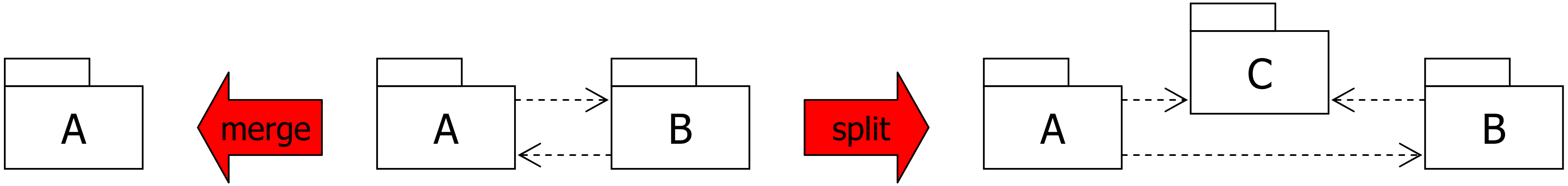
- This involves organising the analysis classes into a set of cohesive packages
- The architecture should be *layered* and *partitioned* to separate concerns
  - It's useful to layer analysis models into application specific and application general layers
- Coupling between packages should be minimised
- Each package should have the minimum number of public or protected elements

Finding analysis packages

- These are often discovered as the model matures
- We can use the natural groupings in the use case model to help identify analysis packages:
  - One or more use cases that support a particular business process or actor
  - Related use cases
- Analysis classes that realise these groupings will often be part of the same analysis package
- Be careful, as it is common for use cases to *cut across* analysis packages!
  - One class may realise several use cases that are allocated to different packages

Analysis packages: guidelines

- A cohesive group of closely related classes or a class hierarchy and supporting classes
- Minimise dependencies between packages
- Localise business processes in packages where possible
- Minimise nesting of packages
- Don’t worry about dependency stereotypes
- Don’t worry about package generalisation
- Refine package structure as analysis progresses
- 4 to 10 classes per package
- Avoid cyclic dependencies!

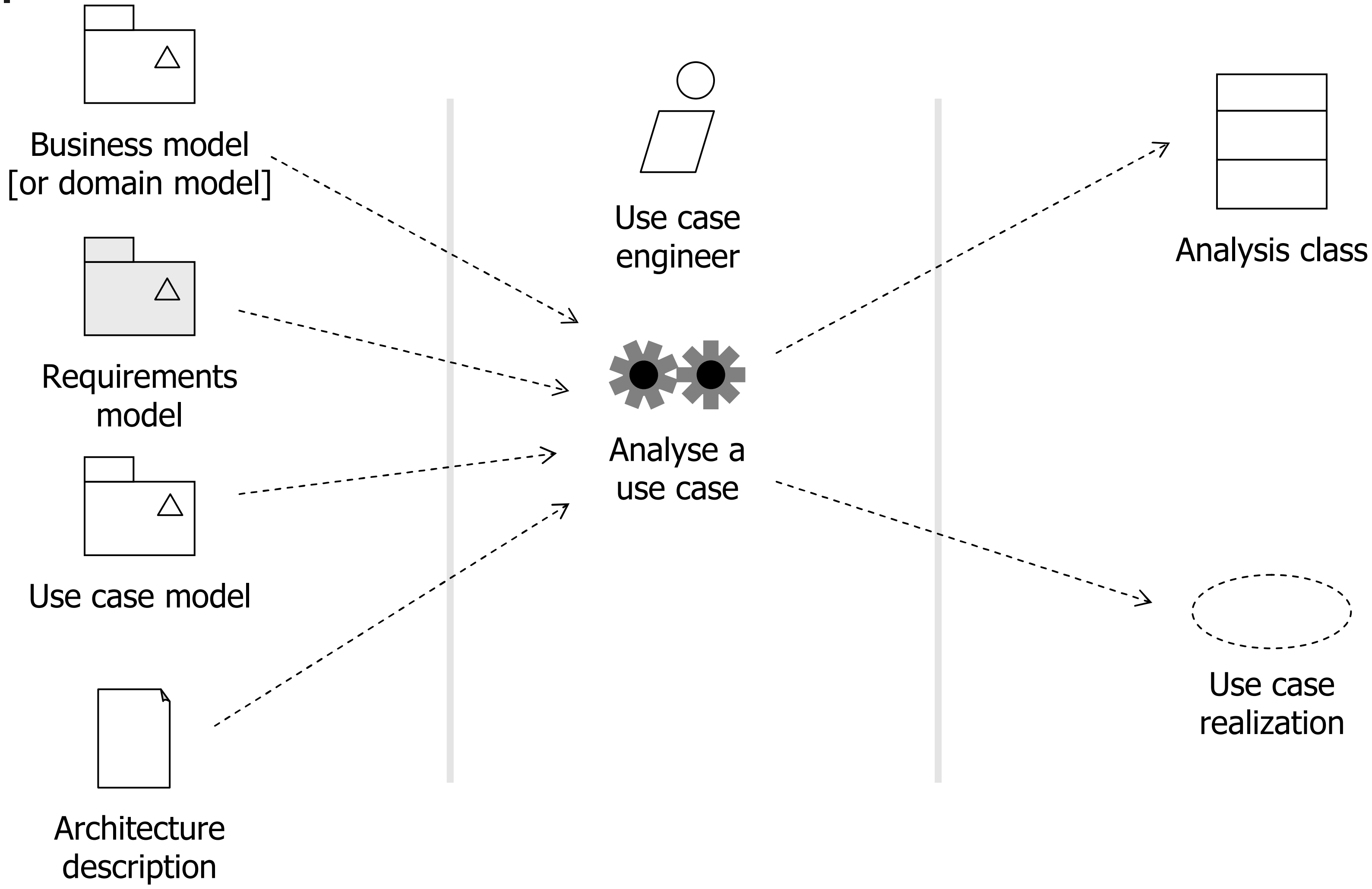


Summary

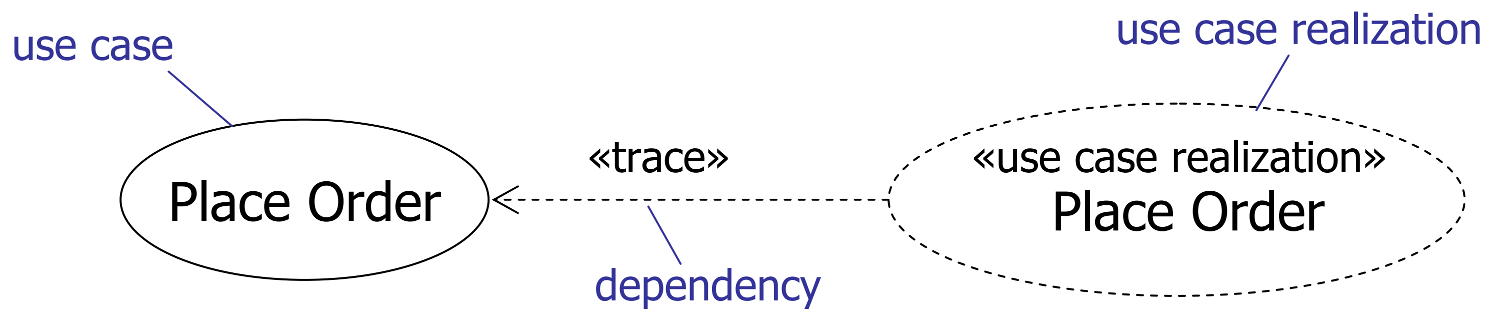
- Packages are the UML way of grouping modeling elements
- There are dependency and generalisation relationships between packages
- The package structure of the analysis model defines the logical system architecture

Analysis - use case realization

Analyse a use case



What are use case realizations?



- Each use case has exactly one *use case realization*
  - parts of the model that show how analysis classes collaborate together to realise the behaviour specified by the use case
  - they model *how* the use case is realised by the analysis classes we have identified
- They are rarely modelled explicitly
  - they form an implicit part of the backplane of the model
  - they can be drawn as a stereotyped collaboration

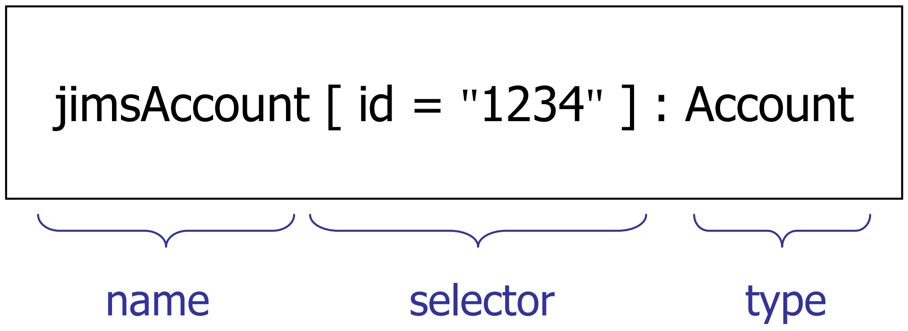
UC realization - elements

- Use case realizations consist of the following elements:
  - Analysis class diagrams
    - These show relationships between the analysis classes that interact to realise the UC
  - Interaction diagrams
    - These show collaborations between specific objects that realise the UC. They are “snapshots” of the running system
  - Special requirements
    - UC realization may well uncover new requirements specific to the use case. These must be captured
  - Use case refinement
    - We may discover new information during realization that means that we have to update the original UC

Interactions

- Interactions are units of behavior of a context classifier
- In use case realization, the context classifier is a use case
  - The interaction shows how the behavior specified by the use case is realized by instances of classifiers
- Interaction diagrams capture an interaction as:
  - Lifelines – participants in the interaction
  - Messages – communications between lifelines

Lifelines

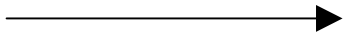
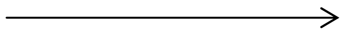
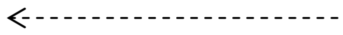
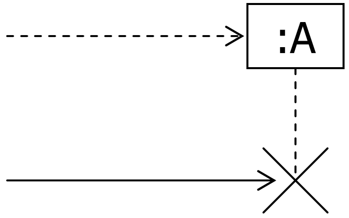
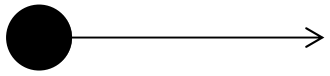
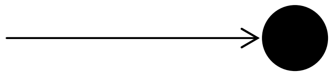


jimsAccount [id = "1234"] : Account

- A lifeline represents a single participant in an interaction
  - Shows how a classifier instance may participate in the interaction
- Lifelines have:
  - name - the name used to refer to the lifeline in the interaction
  - selector - a boolean condition that selects a specific instance
  - type - the classifier that the lifeline represents an instance of
- They *must* be uniquely identifiable within an interaction by name, type or both
- The lifeline has the same icon as the classifier that it represents
  - The lifeline jimsAccount represents an instance of the Account class
  - The selector [id = "1234"] selects a specific Account instance with the id "1234"

Messages

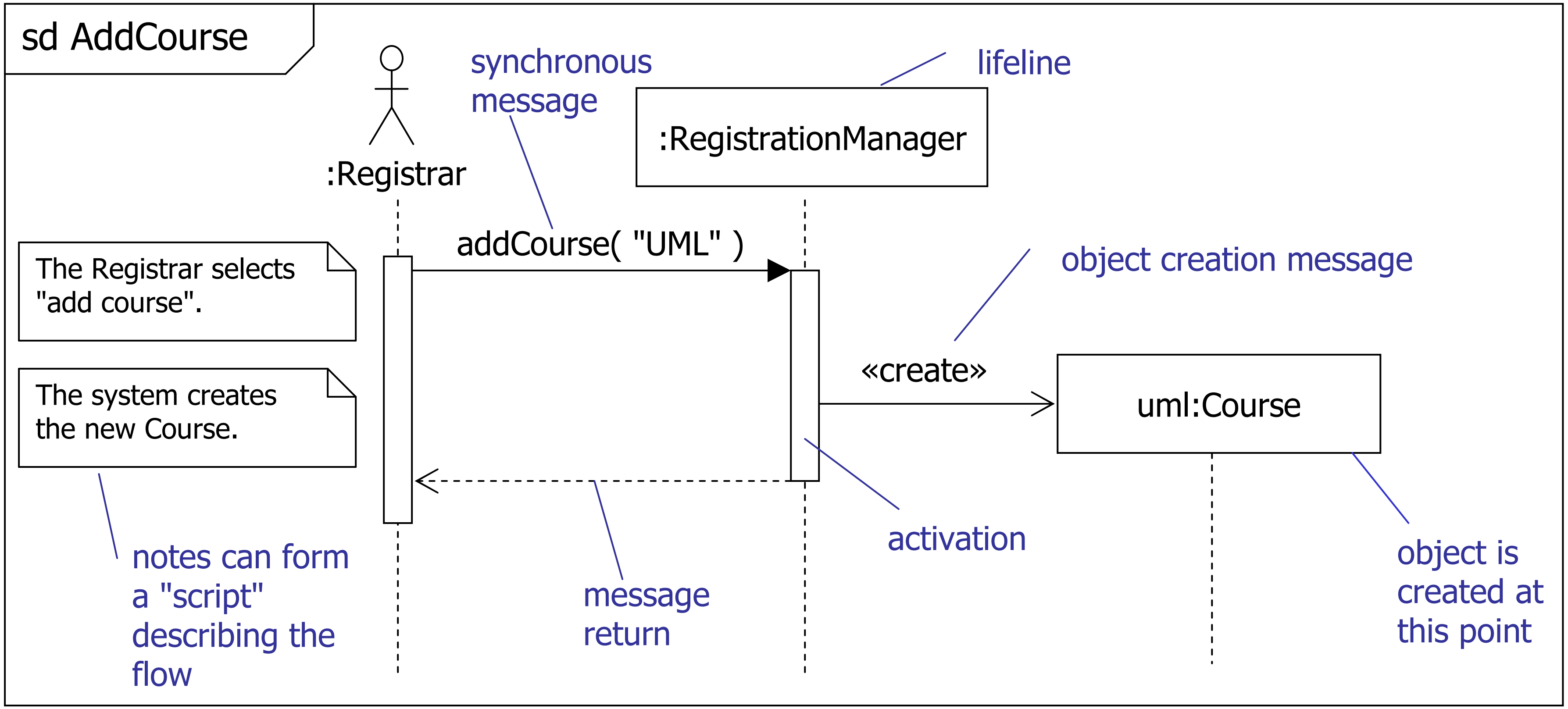
- A message represents a communication between two lifelines

sender → receiver/ target	type of message	semantics
	synchronous message	calling an operation synchronously the sender waits for the receiver to complete
	asynchronous send	calling an operation asynchronously, sending a signal the sender *does not* wait for the receiver to complete
	message return	returning from a synchronous operation call the receiver returns focus of control to the sender
	creation destruction	the sender creates the target the sender destroys the receiver
	found message	the message is sent from outside the scope of the interaction
	lost message	the message fails to reach its destination

Interaction diagrams

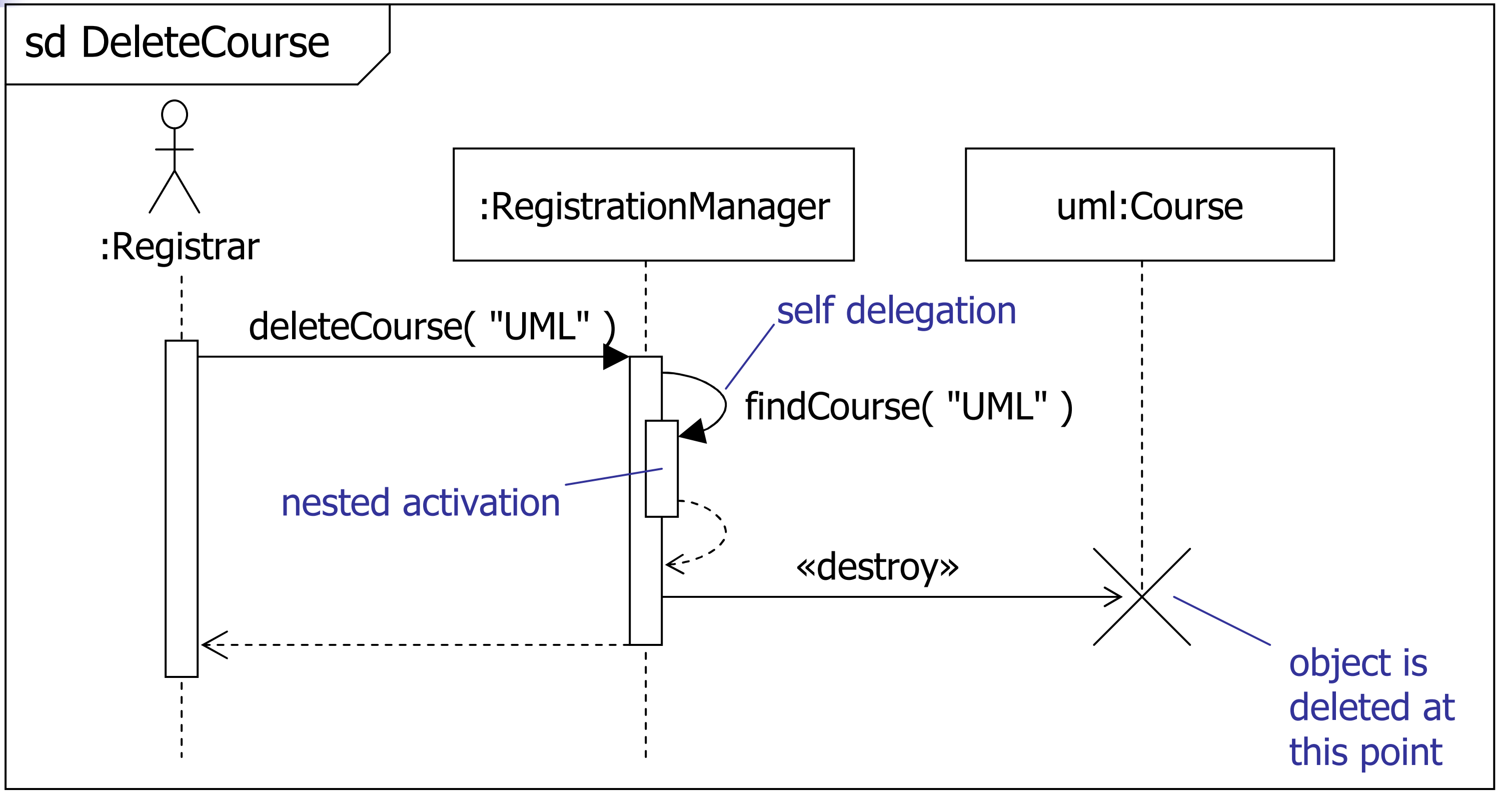
- Sequence diagrams
  - Emphasize time-ordered sequence of message sends
  - Show interactions arranged in a time sequence
  - Are the richest and most expressive interaction diagram
  - Do not show object relationships explicitly - these can be inferred from message sends
- Communication diagrams
  - Emphasize the structural relationships between lifelines
  - Use communication diagrams to make object relationships explicit
- Interaction overview diagrams
  - Show how complex behavior is realized by a set of simpler interactions
- Timing diagrams
  - Emphasize the real-time aspects of an interaction

Sequence diagram syntax



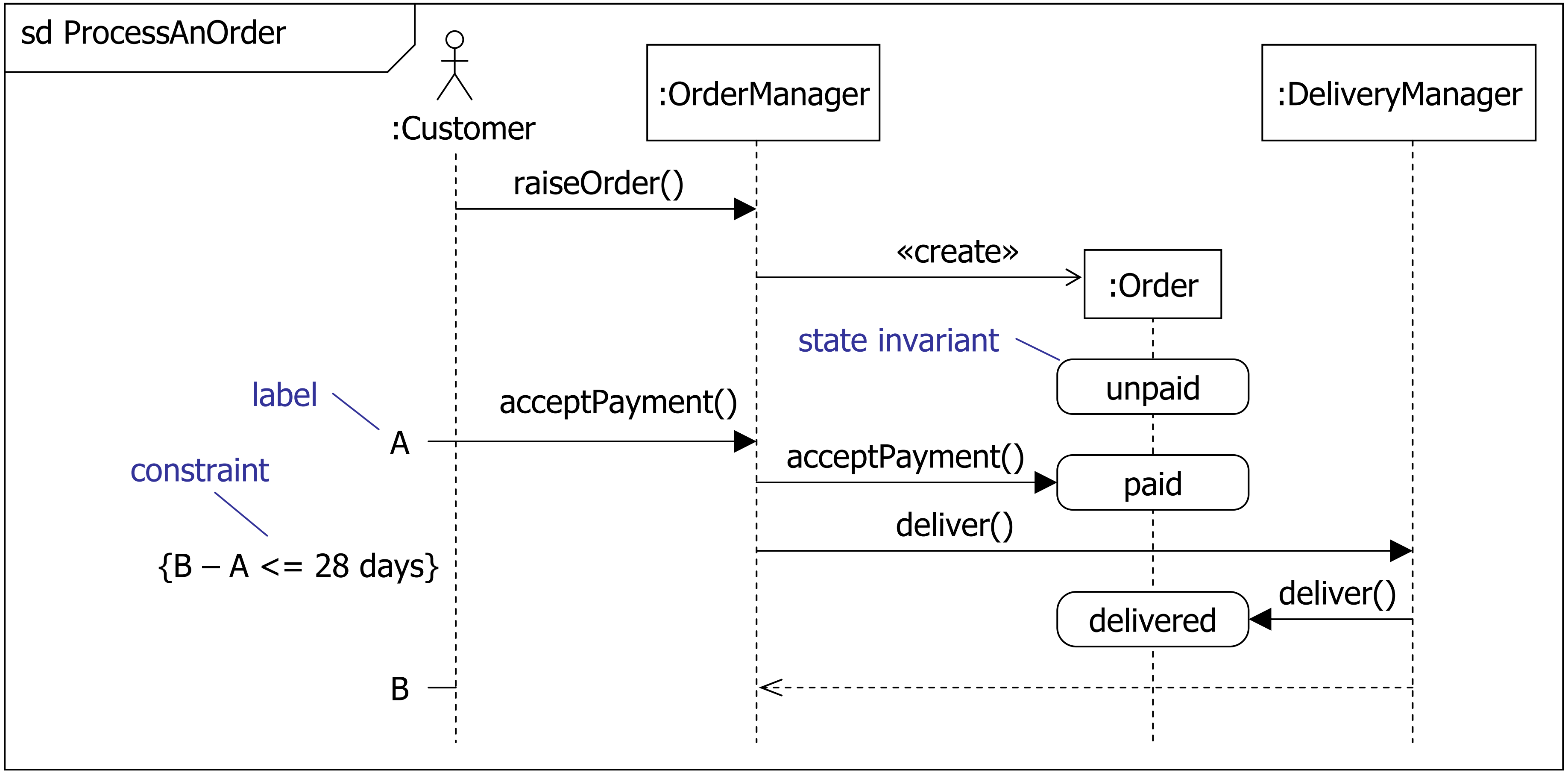
- *All* interaction diagrams may be prefixed sd to indicate their type
  - You can generally infer diagram types from diagram syntax
- Activations indicate when a lifeline has focus of control - they are often omitted from sequence diagrams

Deletion and self-delegation

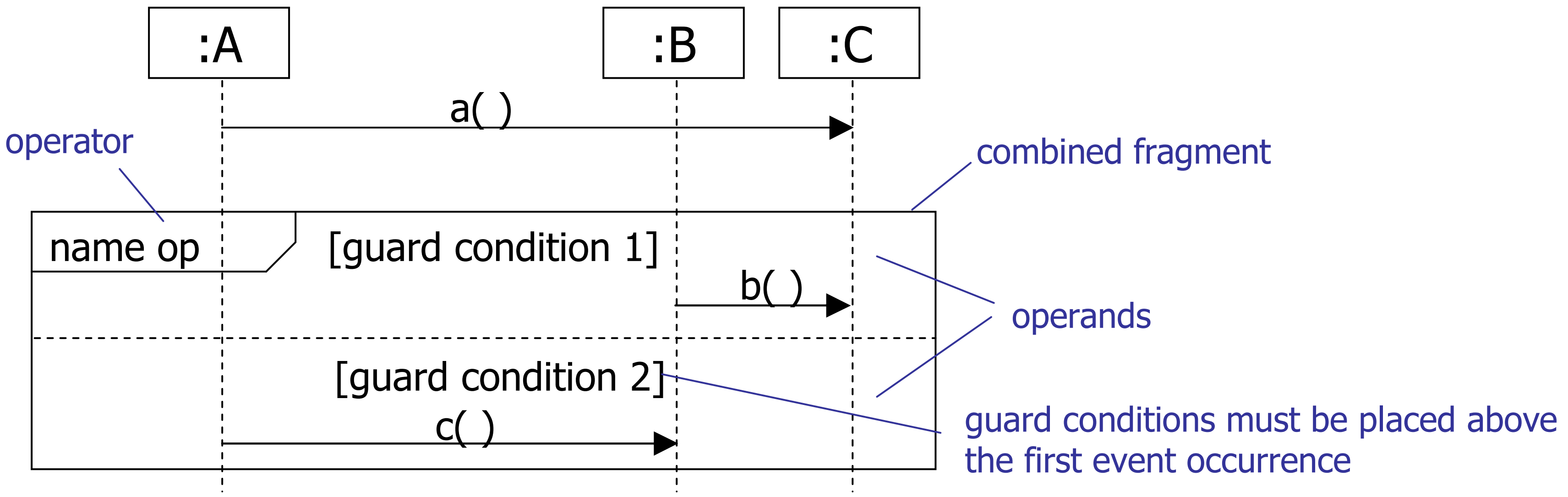


- Self delegation is when a lifeline sends a message to itself
  - Generates a nested activation
- Object deletion is shown by terminating the lifeline's tail at the point of deletion by a large X

State invariants and constraints



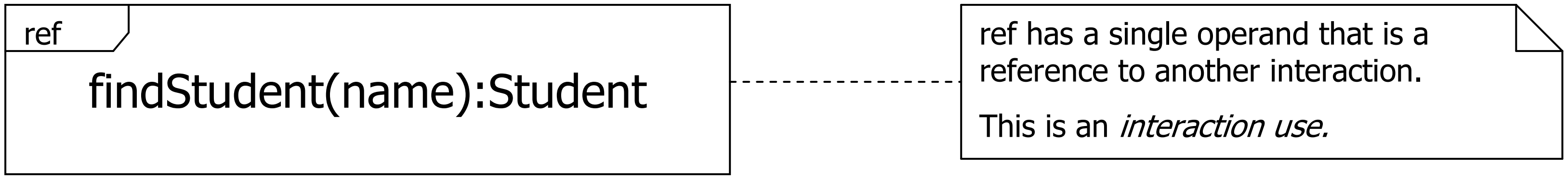
Combined fragments



- Sequence diagrams may be divided into areas called *combined fragments*
- Combined fragments have one or more *operands*
- *Operators* determine how the operands are executed
- *Guard conditions* determine whether operands execute. Execution occurs if the guard condition evaluates to true
  - A single condition may apply to all operands OR
  - Each operand may be protected by its own condition

Common operators

operator	long name	semantics
opt	Option	There is a single operand that executes if the condition is true (like if … then)
alt	Alternatives	The operand whose condition is true is executed. The keyword else may be used in place of a Boolean expression (like select… case)
loop	Loop	This has a special syntax: loop min, max [condition] Iterate min times and then up to max times while condition is true
break	Break	The combined fragment is executed rather than the rest of the enclosing interaction
ref	Reference	The combined fragment refers to another interaction



ref

findStudent(name):Student

ref has a single operand that is a reference to another interaction.

This is an *interaction use.*

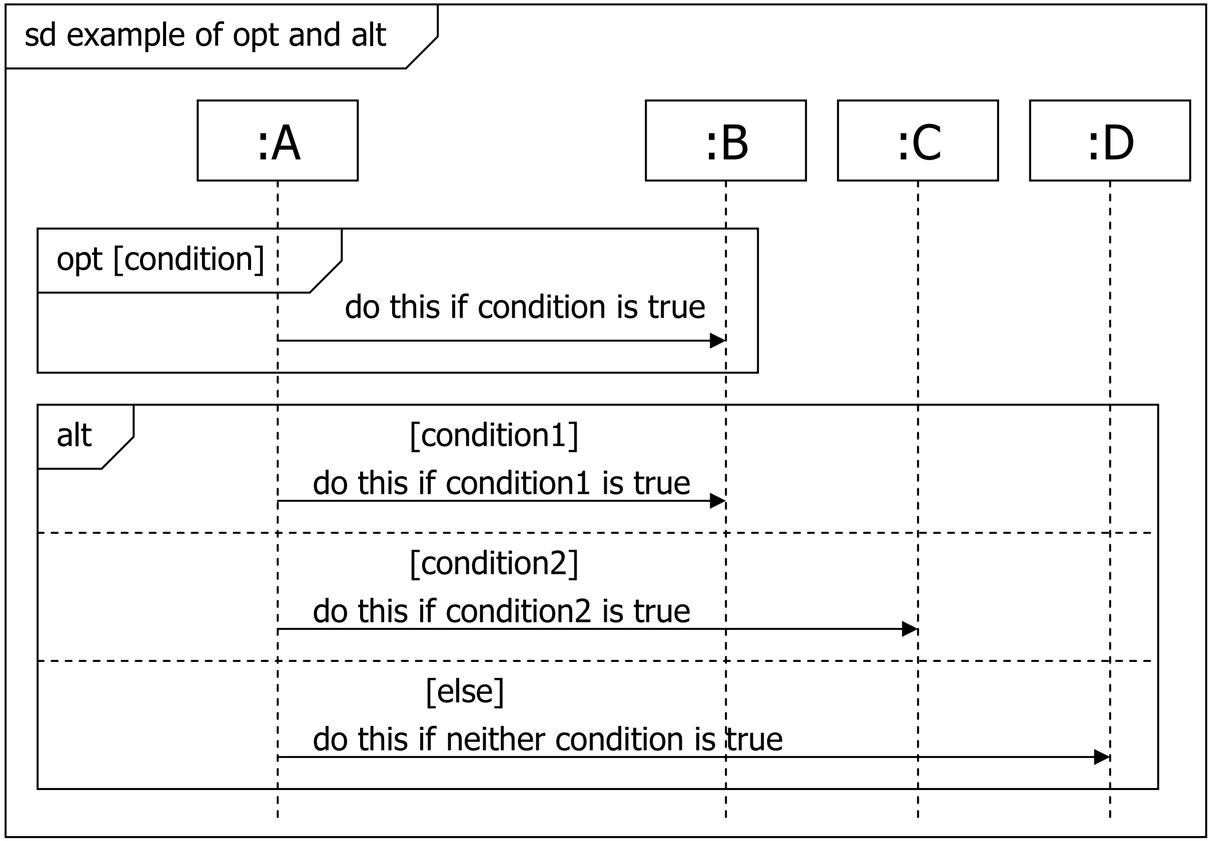
The rest of the operators

- These operators are less common

operator	long name	semantics
par	parallel	Both operands execute in parallel
seq	weak sequencing	The operands execute in parallel subject to the constraint that event occurrences on the *same* lifeline from *different* operands must happen in the same sequence as the operands
strict	strict sequencing	The operands execute in strict sequence
neg	negative	The combined fragment represents interactions that are invalid
critical	critical region	The interaction must execute atomically without interruption
ignore	ignore	Specifies that some message types are intentionally ignored in the interaction
consider	consider	Lists the message types that are considered in the interaction
assert	assertion	The operands of the combined fragments are the only valid continuations of the interaction

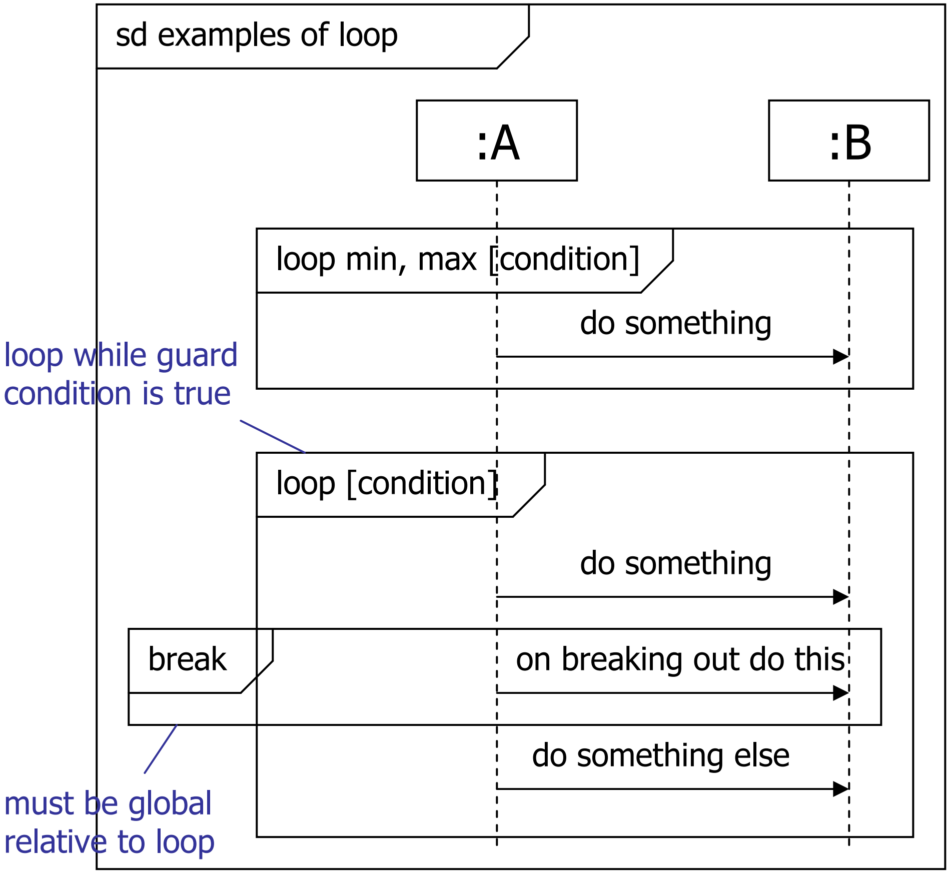
branching with opt and alt

- opt semantics:
  - single operand that executes if the condition is true
- alt semantics:
  - two or more operands each protected by its own condition
  - an operand executes if its condition is true
  - use else to indicate the operand that executes if *none* of the conditions are true


sd example of opt and alt
:A
:B
:C
:D
opt [condition]
do this if condition is true
alt
[condition1]
do this if condition1 is true
[condition2]
do this if condition2 is true
[else]
do this if neither condition is true

Iteration with loop and break

- loop semantics:
  - Loop min times, then loop (max – min) times while condition is true
- loop syntax
  - A loop without min, max or condition is an infinite loop
  - If only min is specified then max = min
  - condition can be
    - Boolean expression
    - Plain text expression *provided* it is clear!
- Break specifies what happens when the loop is broken out of:
  - The break fragment executes
  - The rest of the loop after the break does *not* execute
- The break fragment is *outside* the loop and so should overlap it as shown



loop while guard condition is true

must be global relative to loop

loop idioms

type of loop	semantics	loop expression
infinite loop	keep looping forever	loop *

for i = 1 to n
{body}
	repeat (n) times	loop n

while(booleanExpression)
{body}
	repeat while booleanExpression is true	loop [booleanExpression]

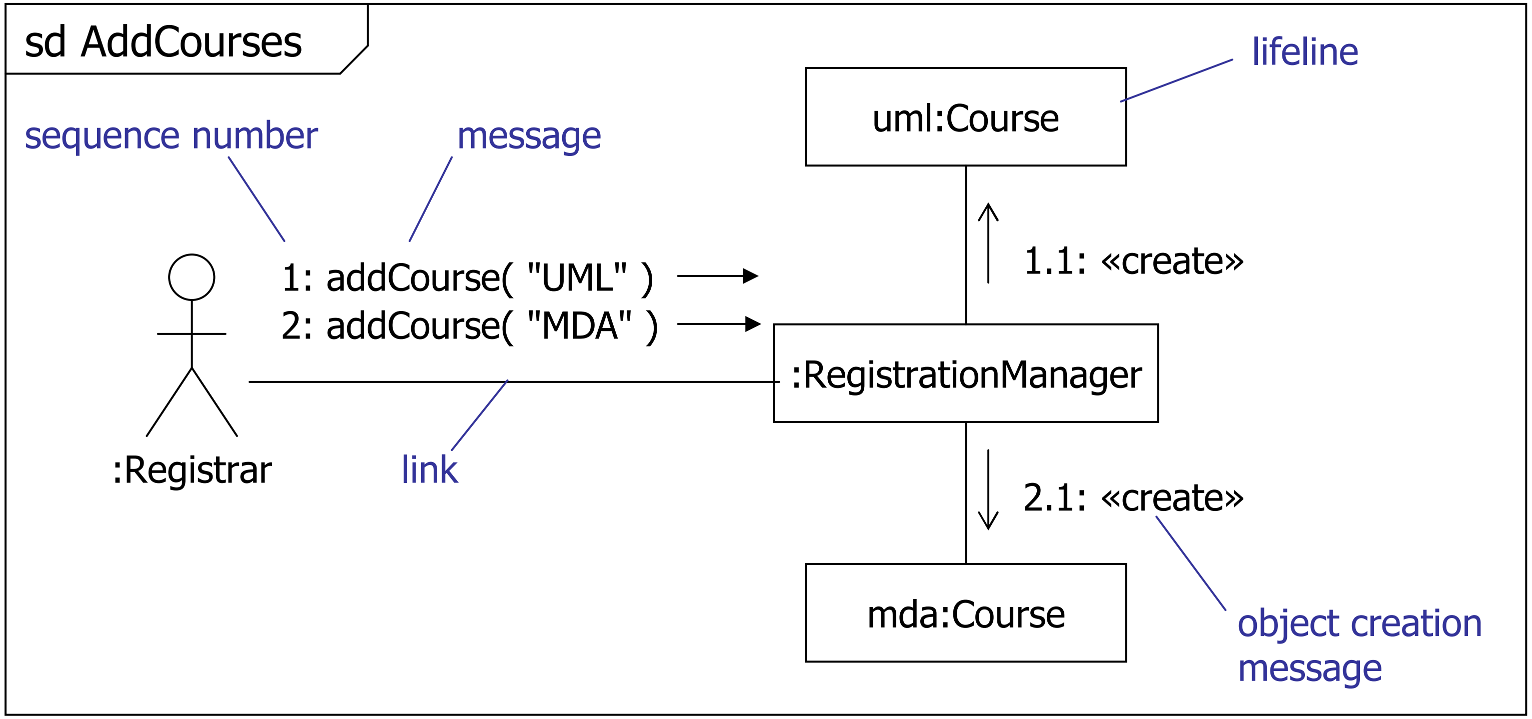
repeat
{body}
while(booleanExpression)
	execute once then repeat while booleanExpression is true	loop 1, * [booleanExpression]

forEach object in set {body}	Execute the loop once for each object in a set	loop [for each object in objectType]

- To specify a forEach loop over a set of objects:
  - use a for loop with an index (see later)
  - use the idiom [for each object in ObjectType] (e.g. [for each student in :Student])

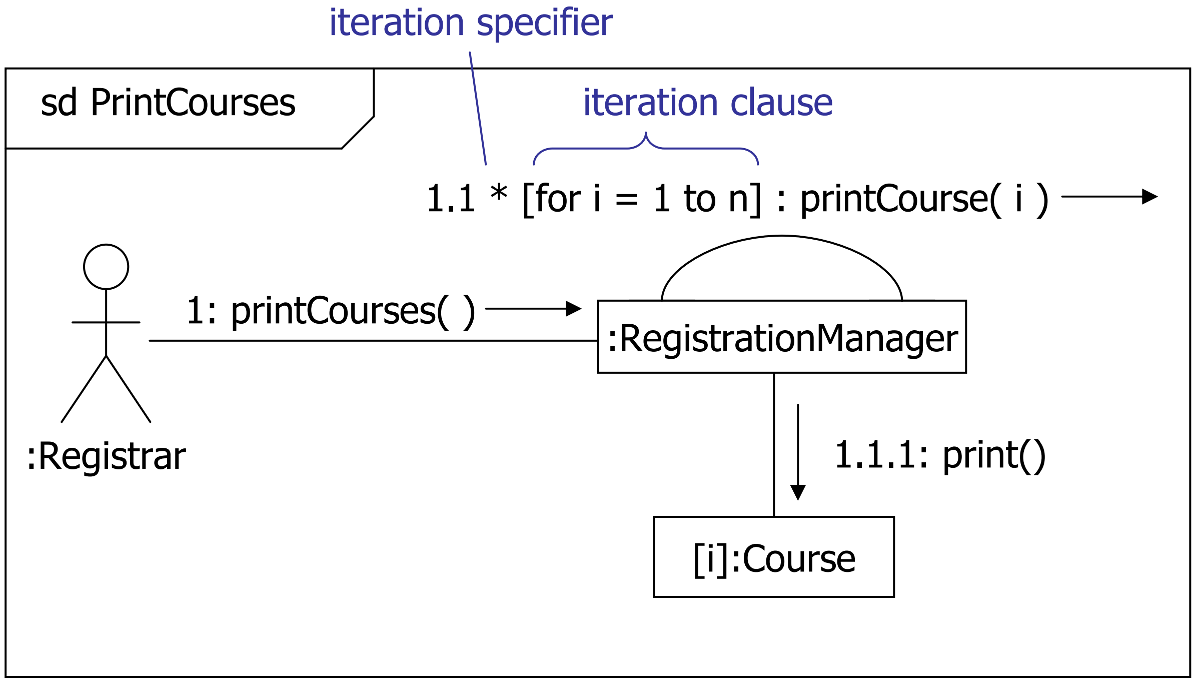
Communication diagram syntax

- Communication diagrams emphasize the structural aspects of an interaction - how lifelines connect together
  - Compared to sequence diagrams they are semantically weak
  - Object diagrams are a special case of communication diagrams

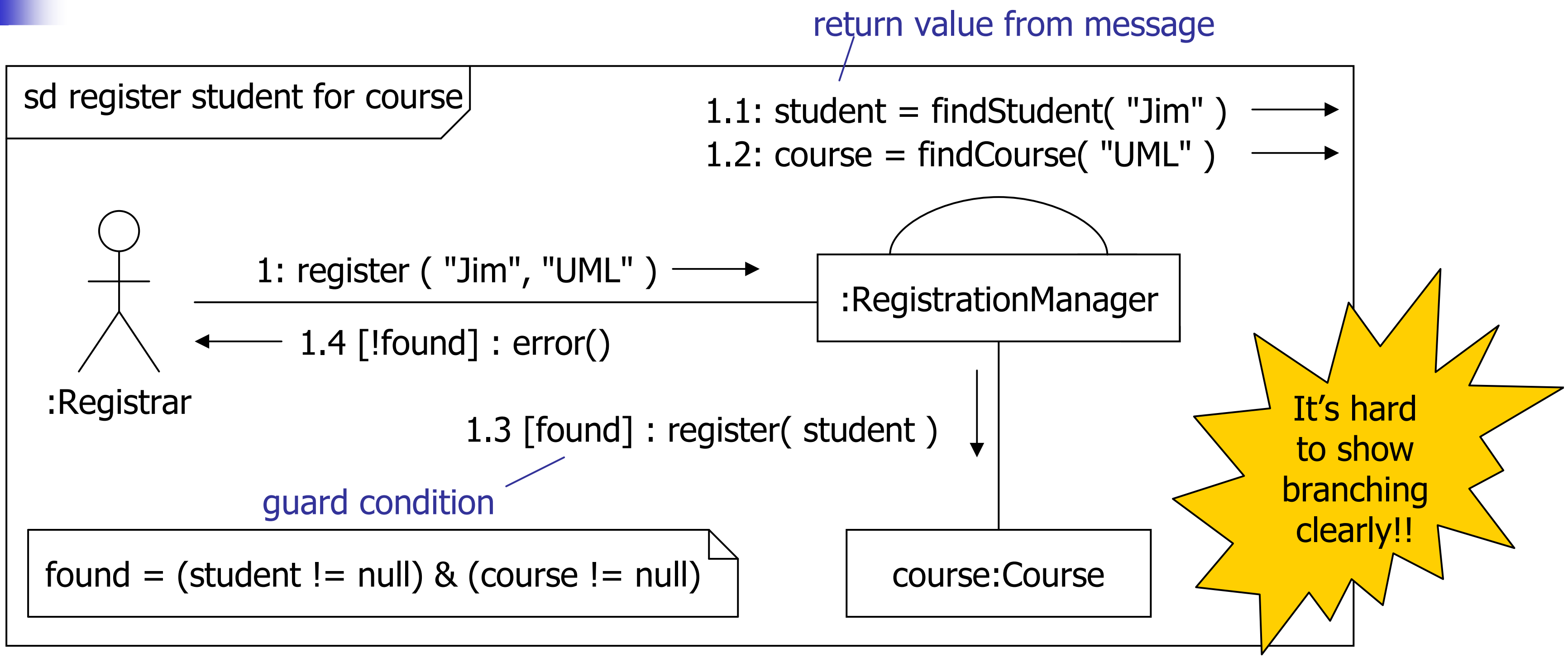


Iteration

- Iteration is shown by using the *iteration specifier* (*), and an optional *iteration clause*
  - There is no prescribed UML syntax for iteration clauses
  - Use code or pseudo code
- To show that messages are sent in parallel use the parallel iteration specifier, *//



Branching


return value from message
sd register student for course
1.1: student = findStudent("Jim")
1.2: course = findCourse("UML")
1: register ("Jim", "UML")
:RegistrationManager
1.4 [!found] : error()
:Registrar
1.3 [found] : register(student)
guard condition
found = (student != null) & (course != null)
course:Course
It's hard to show branching clearly!!

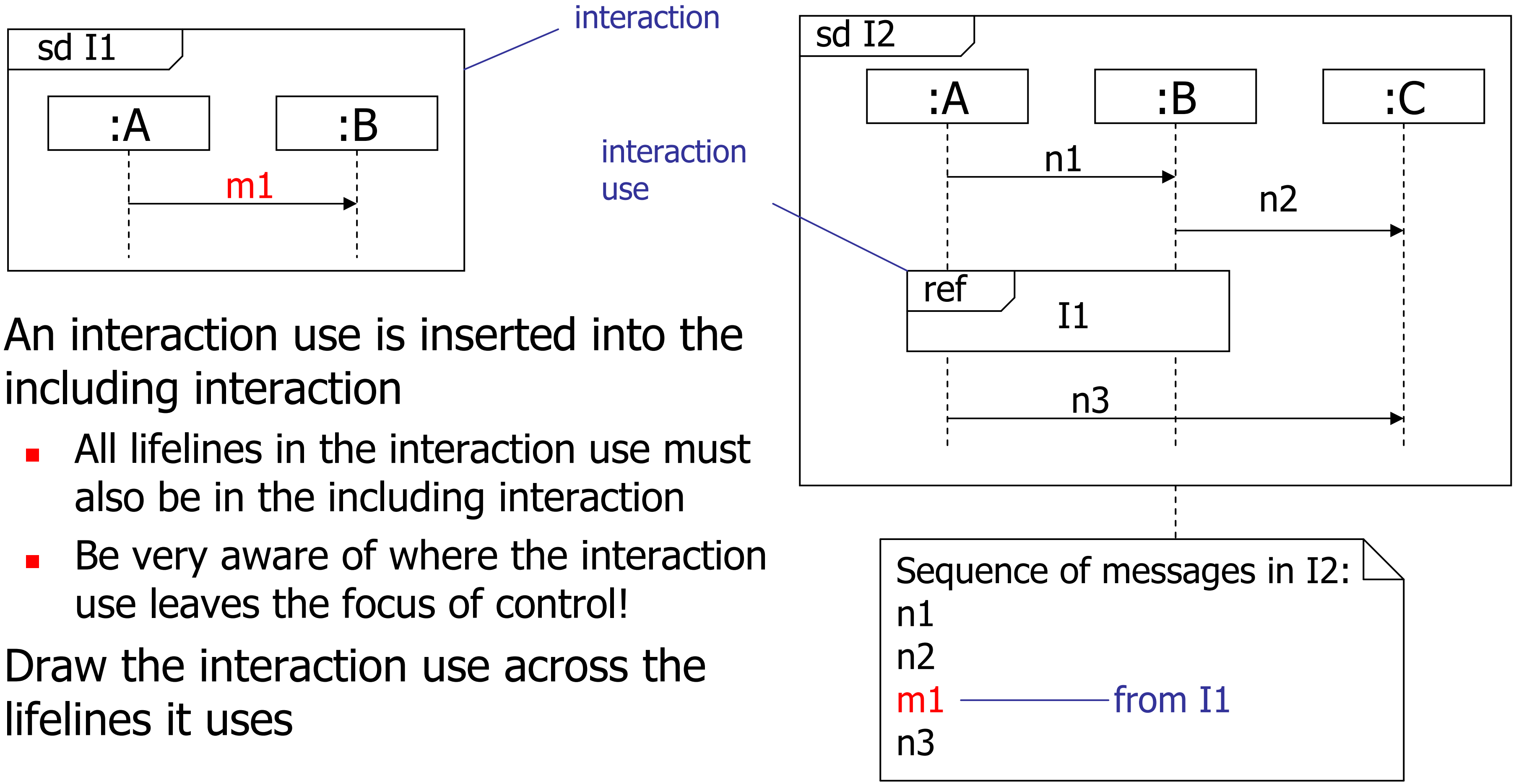
- Branching is modelled by prefixing the sequence number with a *guard condition*
  - There is no prescribed UML syntax for guard conditions!
  - In the example above, we use the variable found. This is true if both the student and the course are found, otherwise it is false

Summary

- In this section we have looked at use case realization using interaction diagrams
- There are four types of interaction diagram:
  - Sequence diagrams – emphasize time-ordered sequence of message sends
  - Communication diagrams – emphasize the structural relationships between lifelines
  - Interaction overview diagrams – show how complex behavior is realized by a set of simpler interactions
  - Timing diagrams – emphasize the real-time aspects of an interaction
- We have looked at sequence diagrams and communication diagrams in this section - we will look at the other types of diagram later

Analysis - advanced use case realization

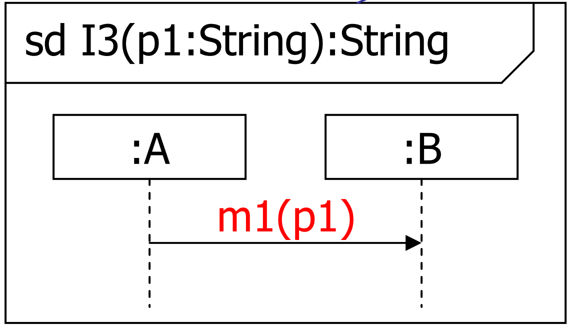
Interaction occurrences



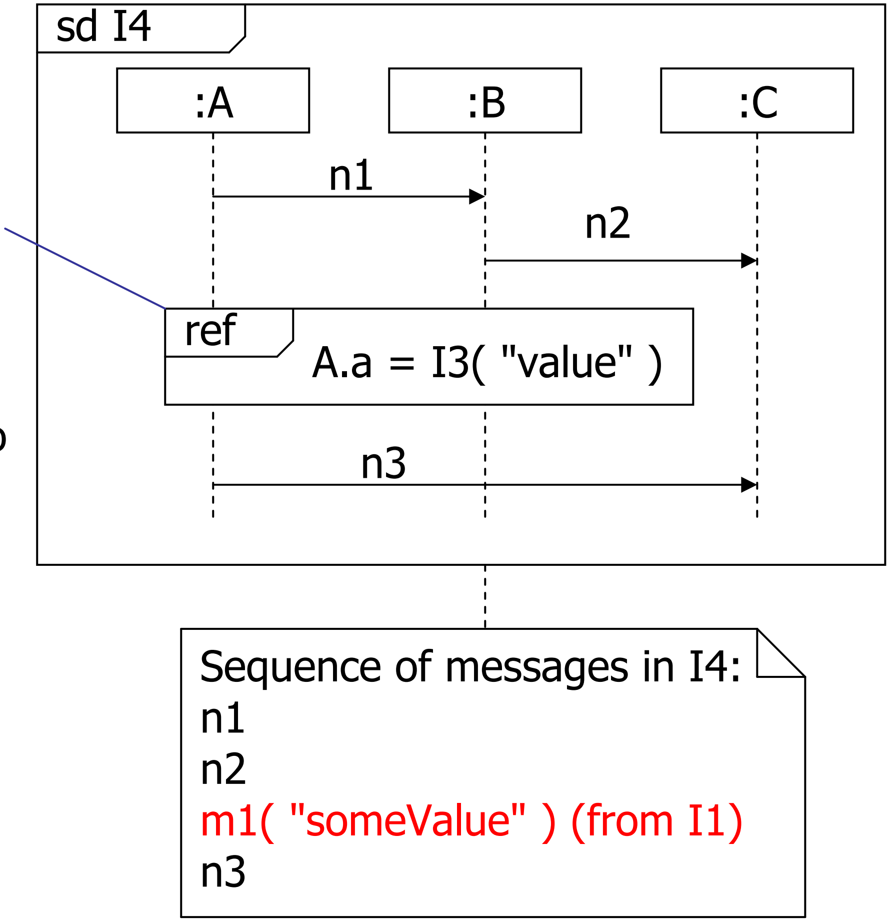
- An interaction use is inserted into the including interaction
  - All lifelines in the interaction use must also be in the including interaction
  - Be very aware of where the interaction use leaves the focus of control!
- Draw the interaction use across the lifelines it uses

Parameters

interaction parameters



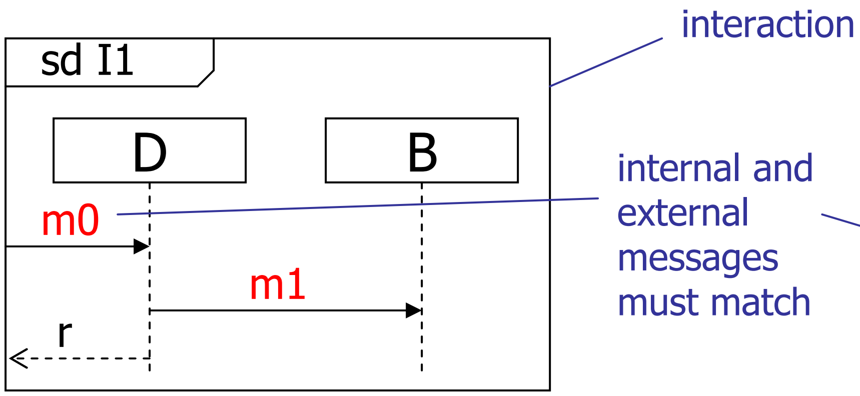
attribute a of class A gets the return value of I3

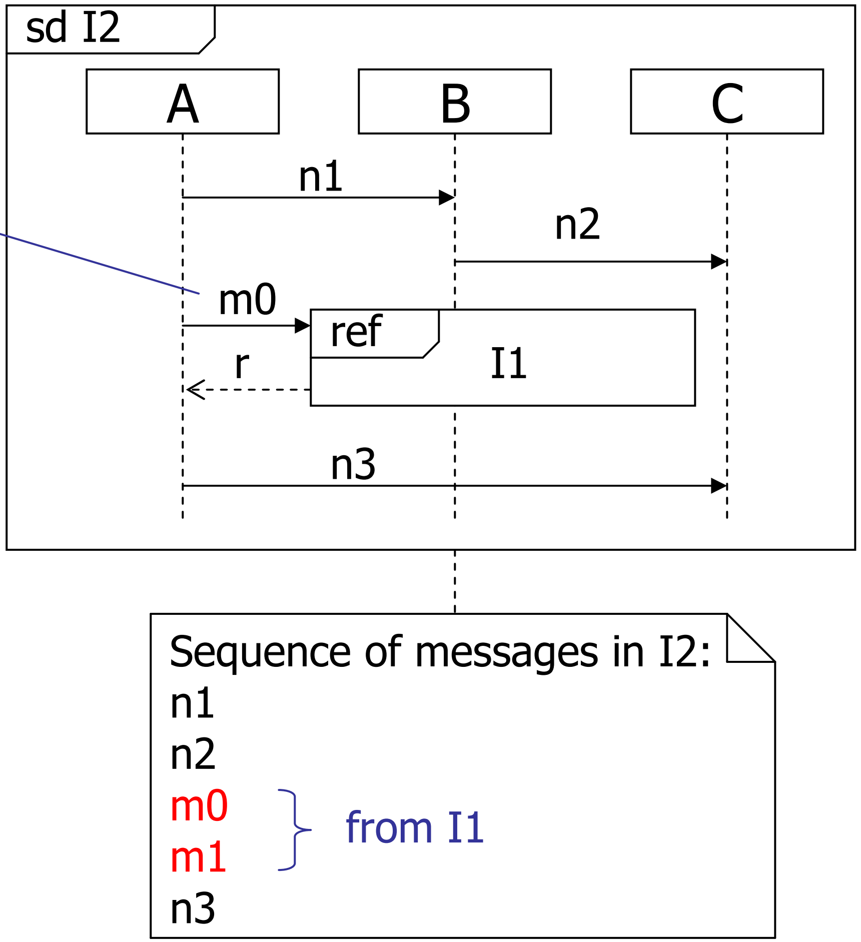


Sequence of messages in I4:
n1
n2
m1("someValue") (from I1)
n3

- Interactions may be parameterized
  - This allows specific values to be supplied to the interaction in each of its occurrences
  - Specify parameters using operation syntax
  - Values for the parameters are supplied in the interaction occurrences
- Interactions may return values
  - You can show a specific return value as a *value return* e.g.
    A:a = I3("value"):"ret"

Gates

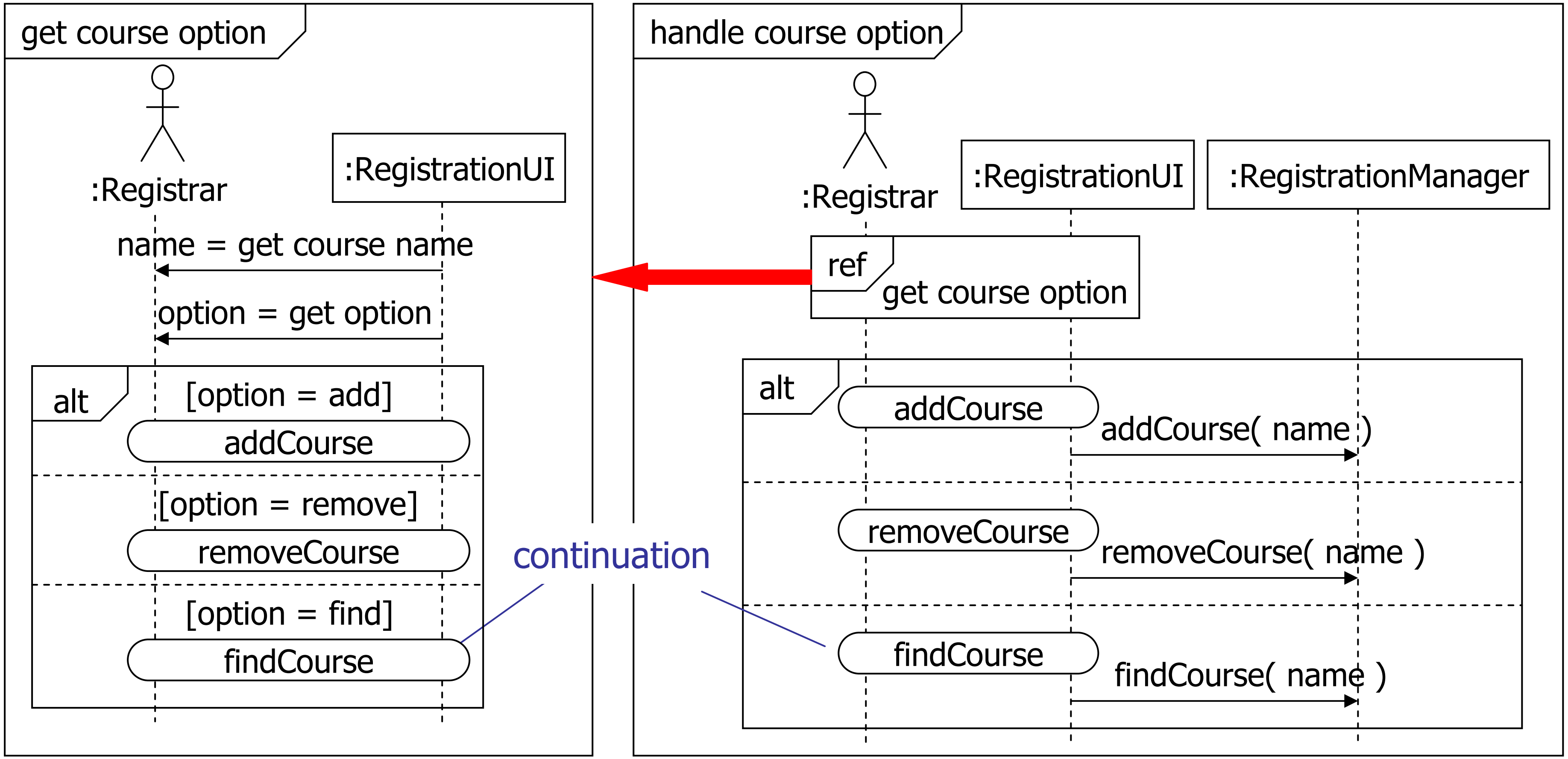




- Gates are inputs and outputs of interactions (and combined fragments – see next slide)
  - Provide connection points that relate messages inside an occurrence or fragment to messages outside it

Continuations

- Continuations allow an interaction fragment to terminate in such a way that it can be continued by another fragment



Summary

- In this section we have looked at:
  - Interaction occurrences
  - Parameters
  - Gates
  - Continuations

Analysis - activity diagrams

What are activity diagrams?

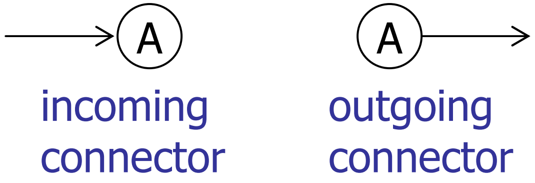
- Activity diagrams are "OO flowcharts"!
- They allow us to model a process as a collection of nodes and edges between those nodes
- Use activity diagrams to model the behavior of:
  - use cases
  - classes
  - interfaces
  - components
  - collaborations
  - operations and methods
  - business processes

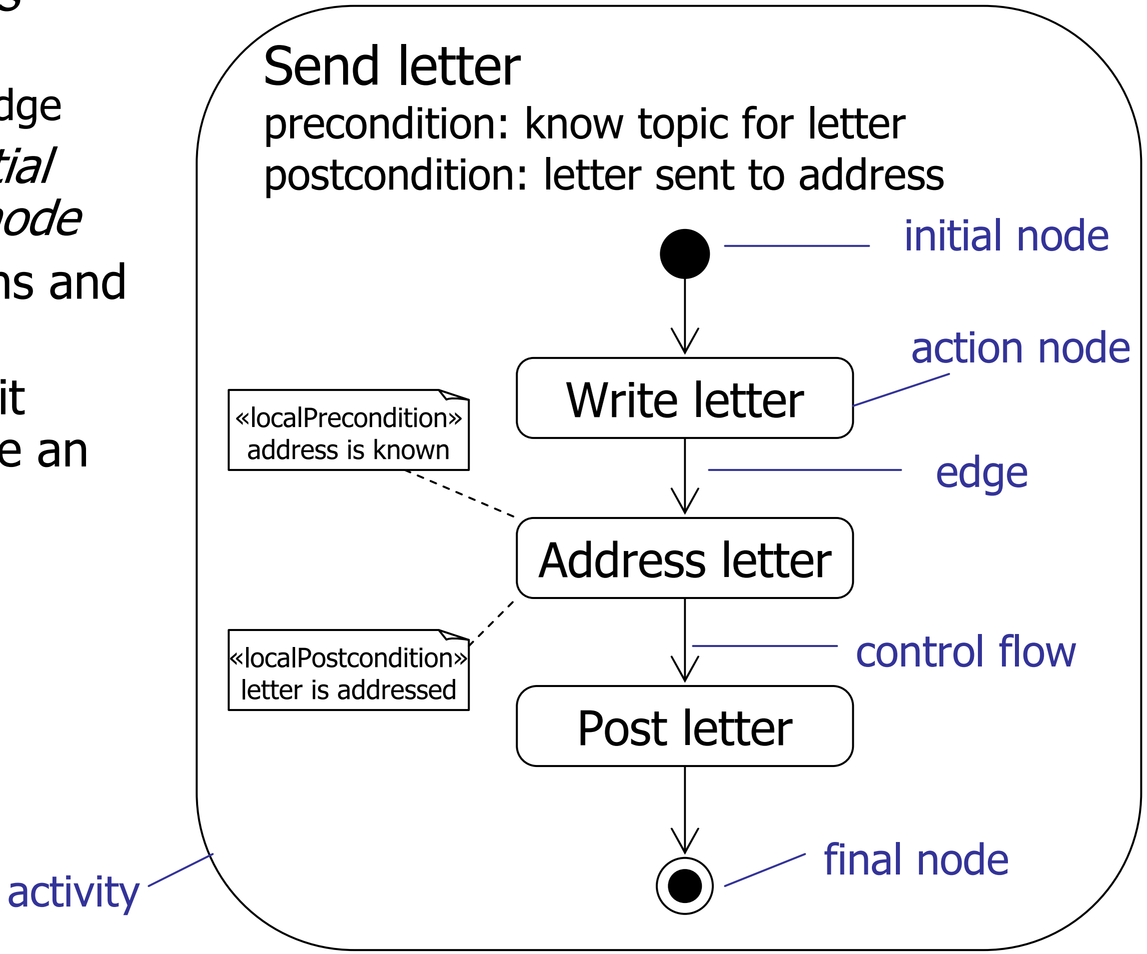
Activities

- Activities are networks of nodes connected by edges
- There are three categories of node:
  - Action nodes - represent discrete units of work that are atomic within the activity
  - Control nodes - control the flow through the activity
  - Object nodes - represent the flow of objects around the activity
- Edges represent flow through the activity
- There are two categories of edge:
  - Control flows - represent the flow of control through the activity
  - Object flows - represent the flow of objects through the activity

Activity diagram syntax

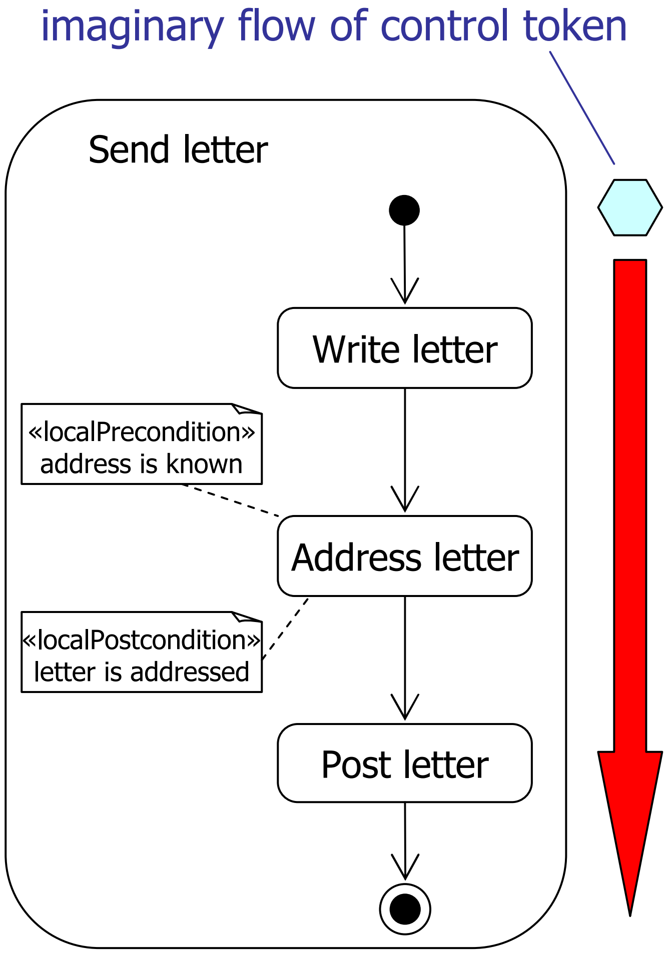
- Activities are networks of *nodes* connected by *edges*
  - The control flow is a type of edge
- Activities usually start in an *initial node* and terminate in a *final node*
- Activities can have preconditions and postconditions
- When an action node finishes, it emits a token that may traverse an edge to trigger the next action
  - This is sometimes known as a *transition*
- You can break an edge using connectors:





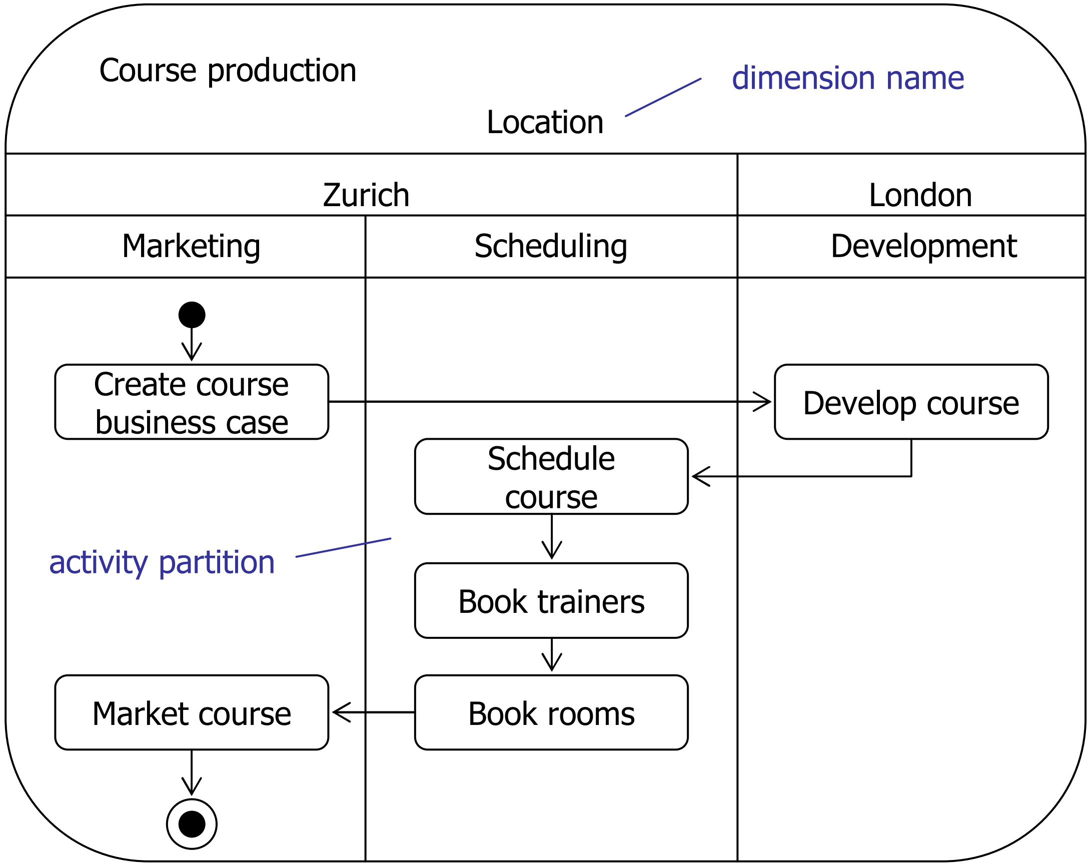
Activity diagram semantics

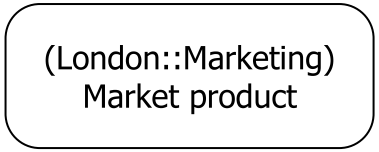
- The *token game*
  - Token – an object, some data or a focus of control
  - Imagine tokens flowing around the activity diagram
- Tokens traverse from a source node to a target node via an edge
  - The source node, edge and target node may all have constraints controlling the movement of tokens
  - All constraints *must* be satisfied before the token can make the traversal
- A node executes when:
  - It has tokens on all of its input edges AND these tokens satisfy predefined conditions (see later)
- When a node starts to execute it takes tokens off its input edges
- When a node has finished executing it offers tokens on its output edges



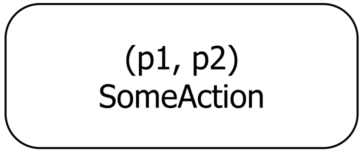
Activity partitions

- Each activity partition represents a high-level grouping of a set of related actions
  - Partitions can be hierarchical
  - Partitions can be vertical, horizontal or both
- Partitions can refer to many different things e.g. business organisations, classes, components and so on
- If partitions can't be shown clearly using parallel lines, put their name in brackets directly above the name of the activities





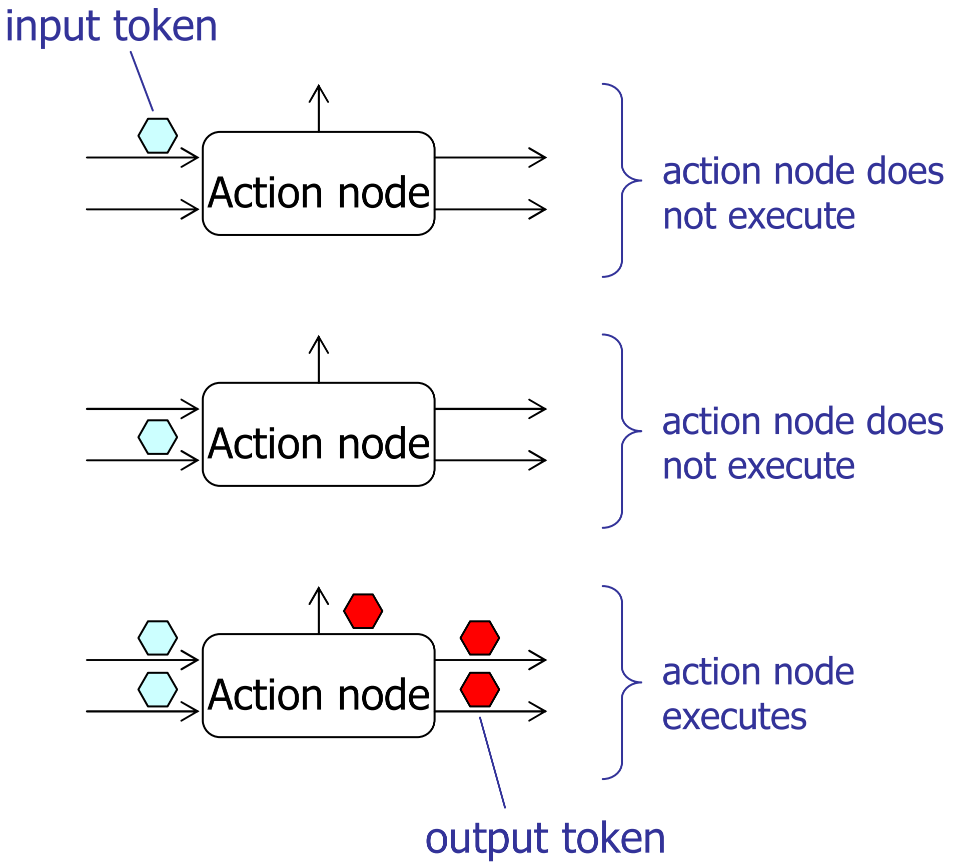
nested partitions



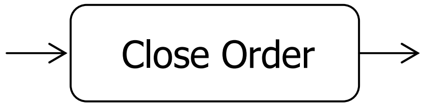
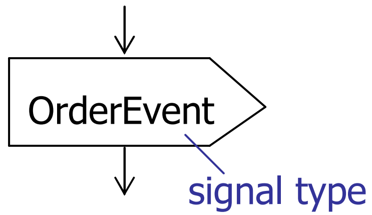
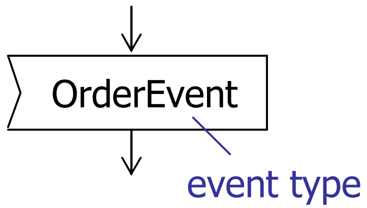
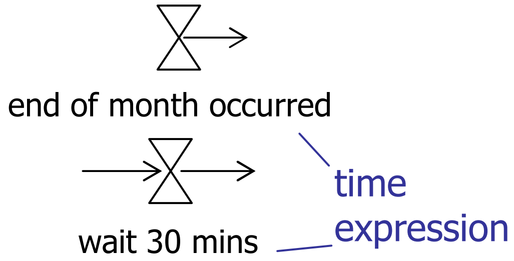
multiple partitions

Action nodes

- Action nodes offer a token on *all* of their output edges when:
  - There is a token *simultaneously* on each input edge
  - The input tokens satisfy all preconditions specified by the node
- Action nodes:
  - Perform a logical AND on their input edges when they begin to execute
  - Perform an implicit fork on their output edges when they have finished executing

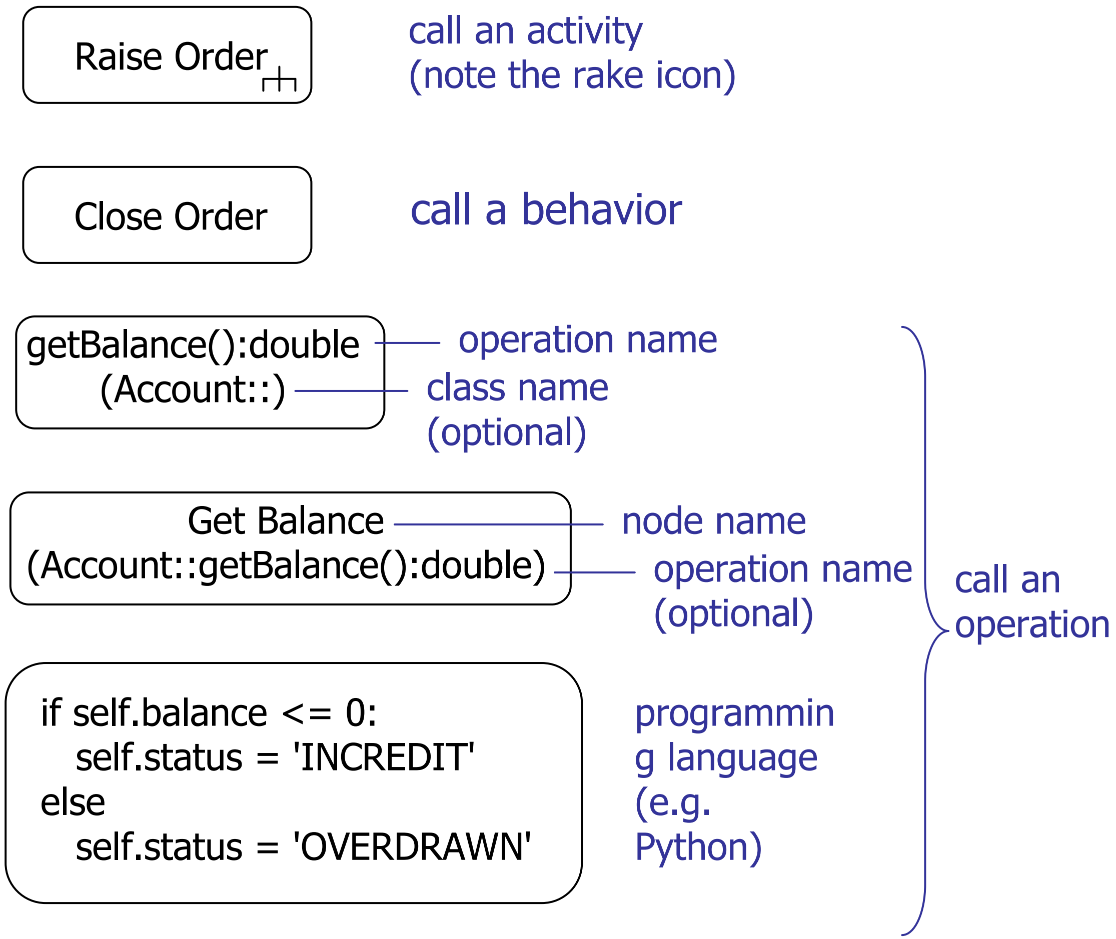


Types of action node

action node syntax	action node semantics
	Call action - invokes an activity, a behavior or an operation. The most common type of action node. See next slide for details.
	Send signal action - sends a signal asynchronously. The sender *does not* wait for confirmation of signal receipt. It may accept input parameters to create the signal
	Accept event action - waits for events detected by its owning object and offers the event on its output edge. Is enabled when it gets a token on its input edge. If there is *no* input edge it starts when its containing activity starts and is *always* enabled.
	Accept time event action - waits for a set amount of time. Generates time events according to it's time expression.

Call action node syntax

- The most common type of node
- Call action nodes may invoke:
  - an activity
  - a behavior
  - an operation
- They may contain code fragments in a specific programming language
  - The keyword 'self' refers to the context of the activity that owns the action



Raise Order

call an activity
(note the rake icon)

Close Order

call a behavior

getBalance():double
(Account::)

operation name

class name
(optional)

Get Balance
(Account::getBalance():double)

node name

operation name
(optional)

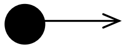
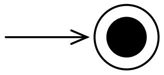
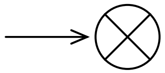
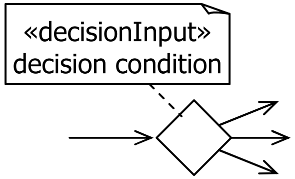
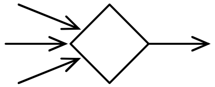
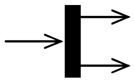
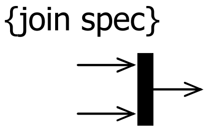
```
if self.balance <= 0:
   self.status = 'INCREDIT'
else
   self.status = 'OVERDRAWN'
```

programmin
g language
(e.g.
Python)

call an
operation

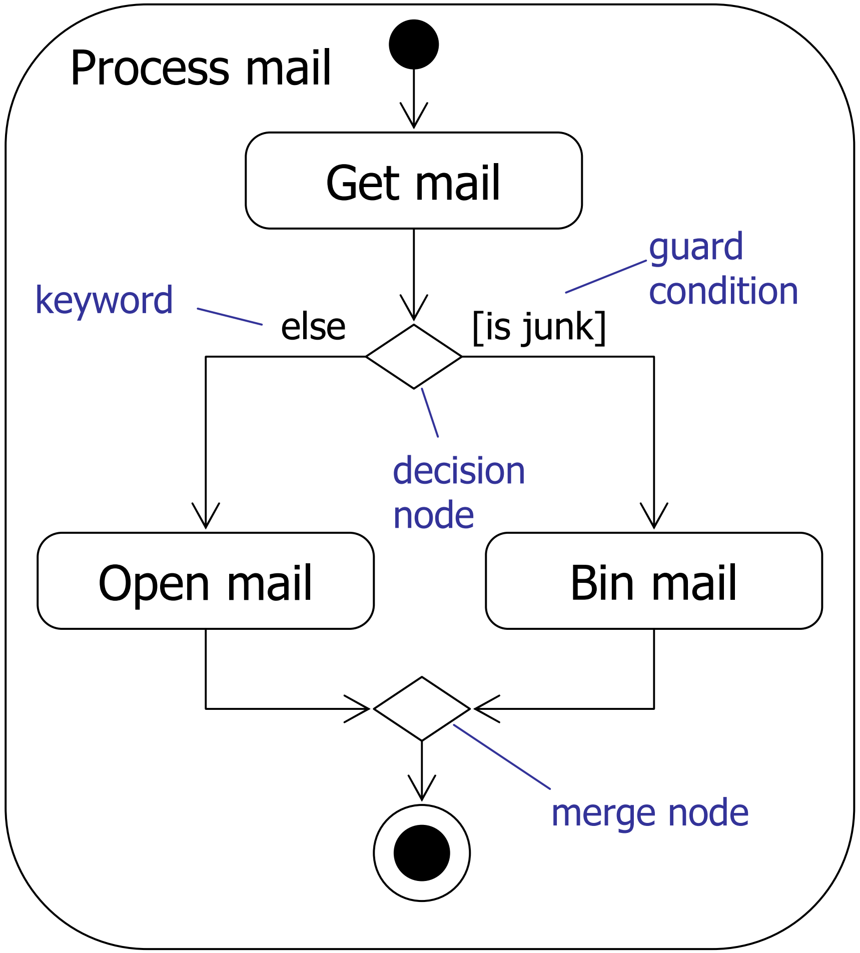
14.8

Control nodes

control node syntax	control node semantics	
	Initial node – indicates where the flow starts when an activity is invoked	
	Activity final node – terminates an activity	Final nodes
	Flow final node – terminates a specific flow within an activity. The other flows are unaffected	
	Decision node– guard conditions on the output edges select one of them for traversal May optionally have inputs defined by a «decisionInput»	See examples on next two slides
	Merge node – selects *one* of its input edges	
	Fork node – splits the flow into multiple concurrent flows	
	Join node – synchronizes multiple concurrent flows May optionally have a join specification to modify its semantics	

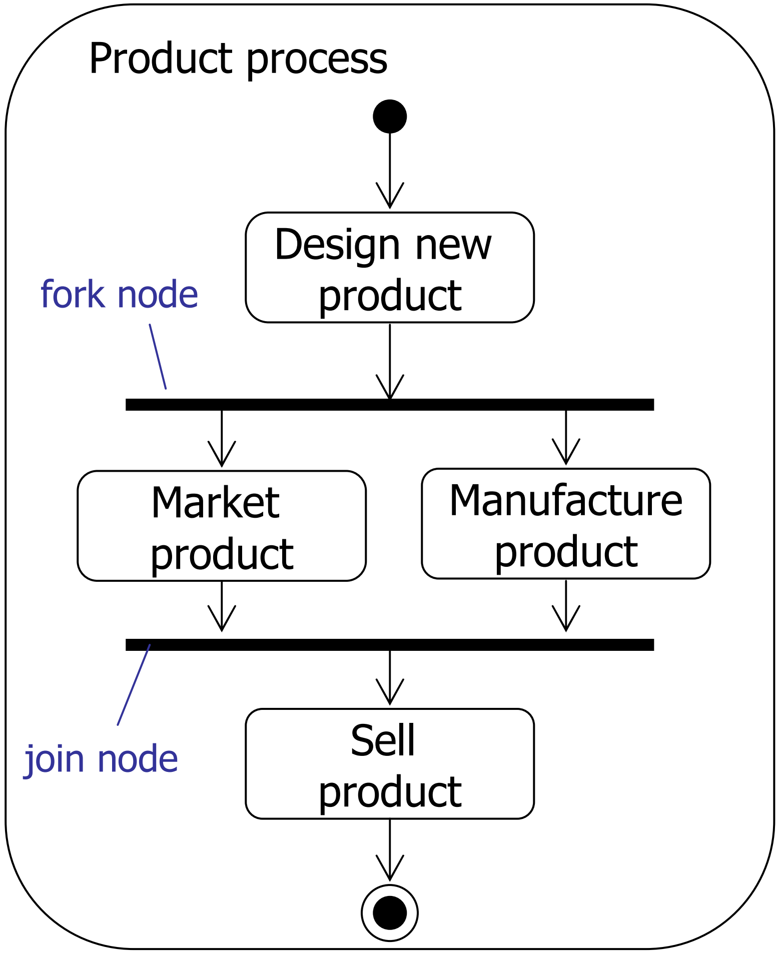
Decision and merge nodes

- A decision node is a control node that has one input edge and two or more alternate output edges
  - Each edge out of the decision is protected by a *guard condition*
  - guard conditions must be mutually exclusive
  - The edge can be taken if and only if the guard condition evaluates to true
  - The keyword *else* specifies the path that is taken if *none* of the guard conditions are true
- A merge node accepts one of several alternate flows
  - It has two or more input edges and exactly one output edge



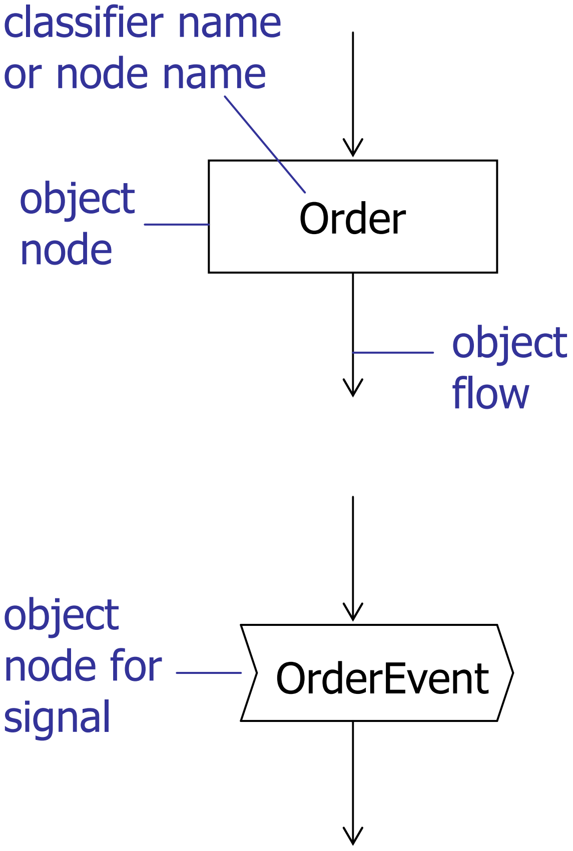
Fork and join nodes - concurrency

- Forks nodes model concurrent flows of work
  - Tokens on the single input edge are replicated at the multiple output edges
- Join nodes synchronize two or more concurrent flows
  - Joins have two or more incoming edges and exactly one outgoing edge
  - A token is offered on the outgoing edge when there are tokens on *all* the incoming edges i.e. when the concurrent flows of work have all finished



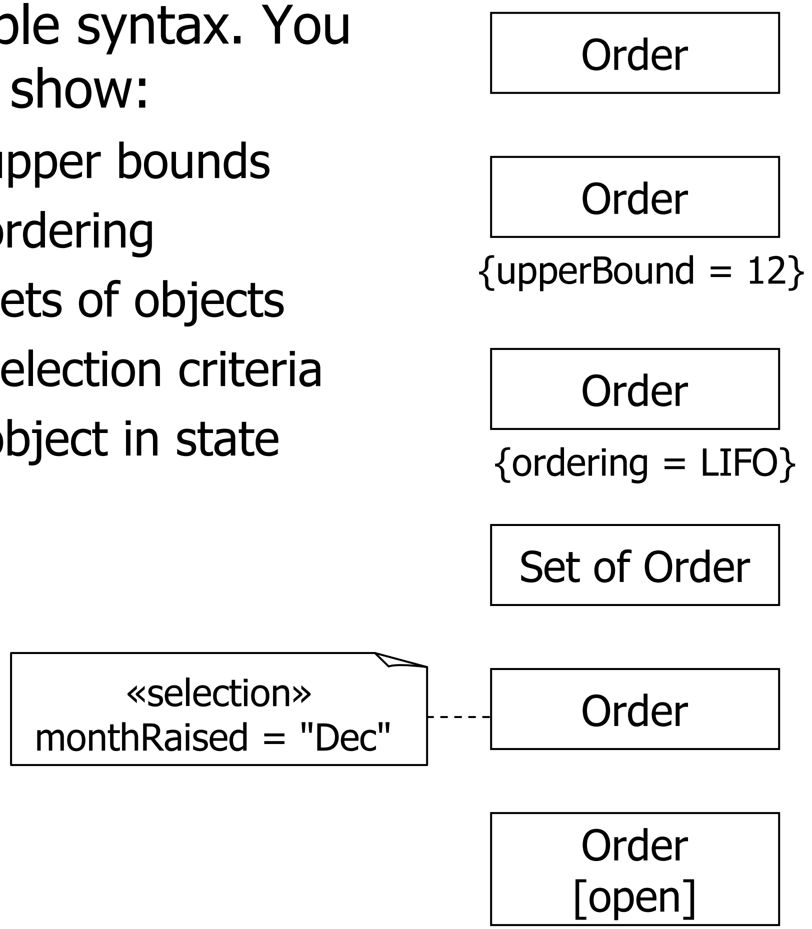
Object nodes

- Object nodes indicate that instances of a particular classifier may be available
  - If no classifier is specified, then the object node can hold any type of instance
- Multiple tokens can reside in an object node *at the same time*
  - The upper bound defines the maximum number of tokens (infinity is the default)
- Tokens are presented to the single output edge according to an ordering:
  - FIFO – first in, first out (the default)
  - LIFI – last in, first out
  - Modeler defined – a selection criterion is specified for the object node



Object node syntax

- Object nodes have a flexible syntax. You may show:
    - upper bounds
    - ordering
    - sets of objects
    - selection criteria
    - object in state



Order

order objects may be available

Order
{upperBound = 12}

zero to 12 Order objects may be available

Order
{ordering = LIFO}

last Order object in is the first out
(FIFO is the default)

Set of Order

sets of Order objects may be available

«selection»
monthRaised = "Dec"

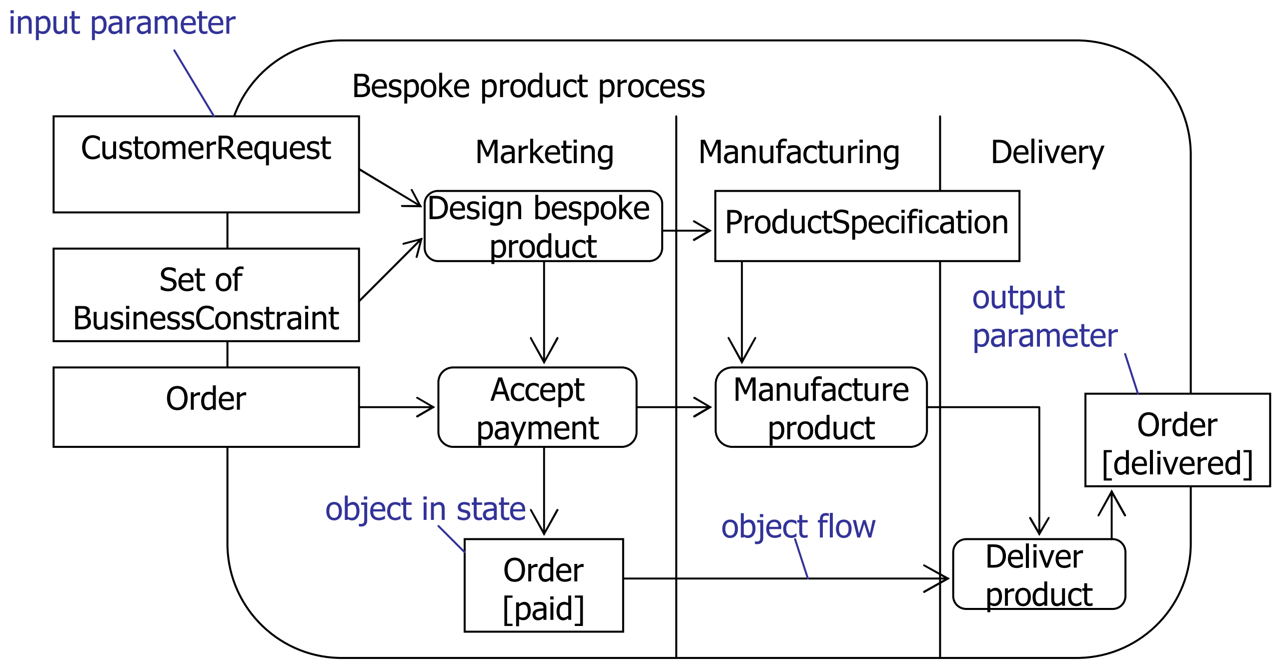
Order

Order objects raised in December may be available

Order
[open]

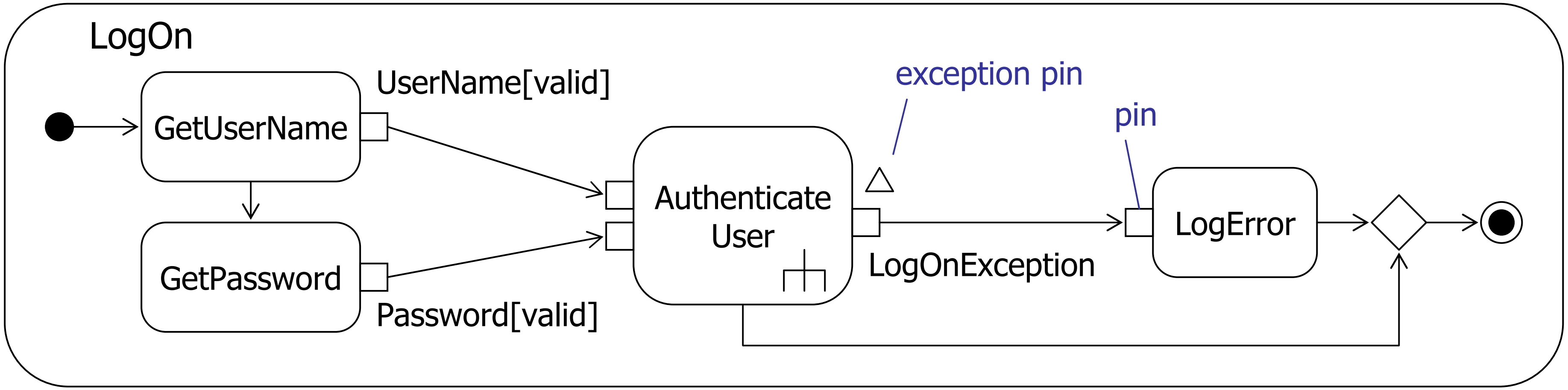
select Order objects in the open state

Activity parameters



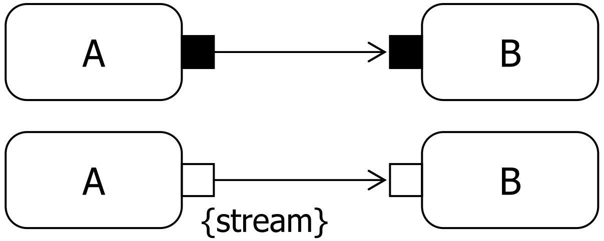
- Object nodes can provide input and output parameters to activities
  - Input parameters have one or more output object flows into the activity
  - Output parameters have one or more input object flows out of the activity
- Draw the object node overlapping the activity boundary

Pins



- Pins are object nodes for inputs to, and outputs from, actions
  - Same syntax as object nodes
  - Input pins have exactly one input edge
  - Output pins have exactly one output edge
  - Exception pins are marked with an equilateral triangle
  - Streaming pins are filled in black or marked with {stream}

streaming – see notes

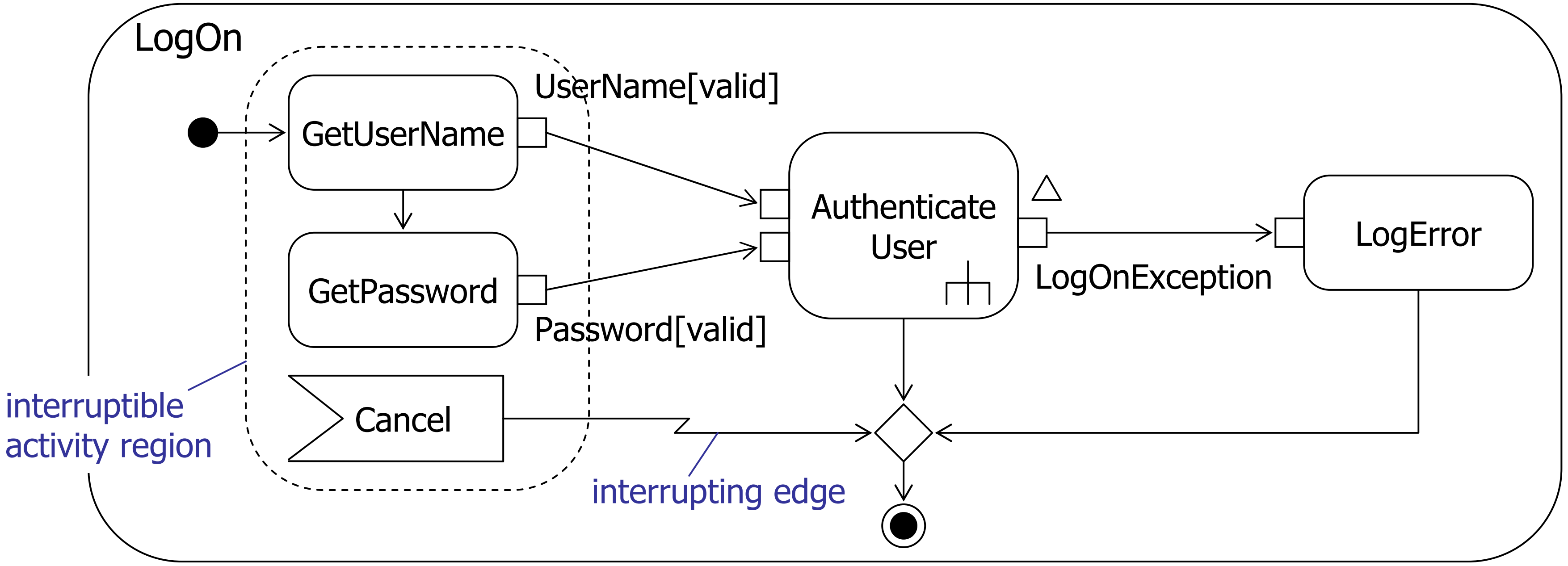


Summary

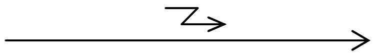
- We have seen how we can use activity diagrams to model flows of activities using:
  - Activities
    - Connectors
  - Activity partitions
  - Action nodes
    - Call action node
    - Send signal/accept event action node
    - Accept time event action node
  - Control nodes
    - decision and merge
    - fork and join
  - Object nodes
    - input and output parameters
    - pins

Analysis - advanced activity diagrams

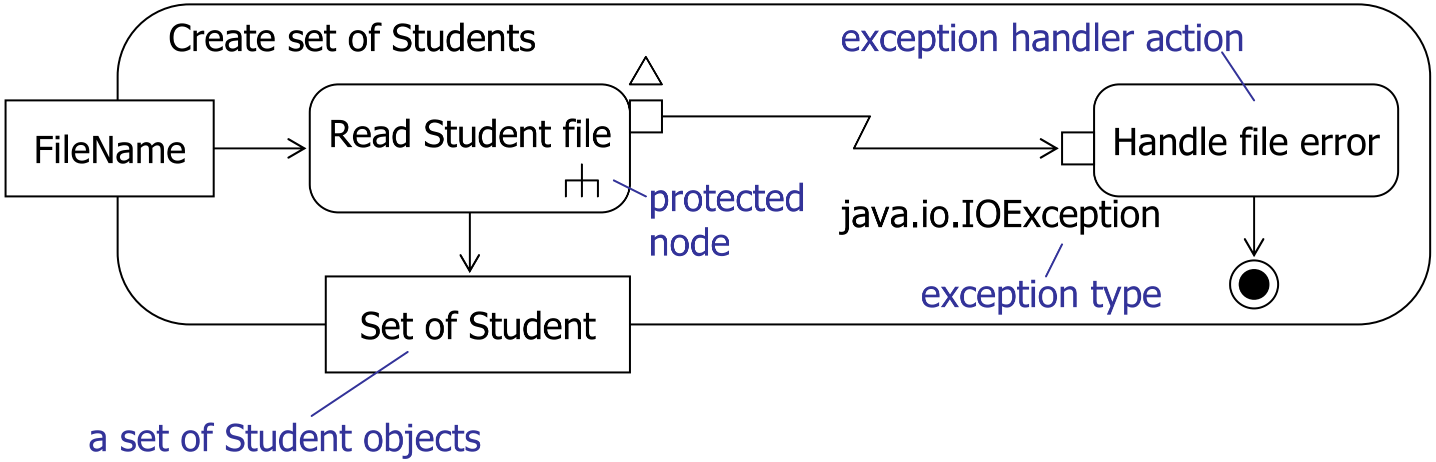
Interruptible activity regions



- Interruptible activity regions may be interrupted when a token traverses an interrupting edge
  - All flows in the region are aborted
- Interrupting edges *must* cross the region boundary


alternative notation

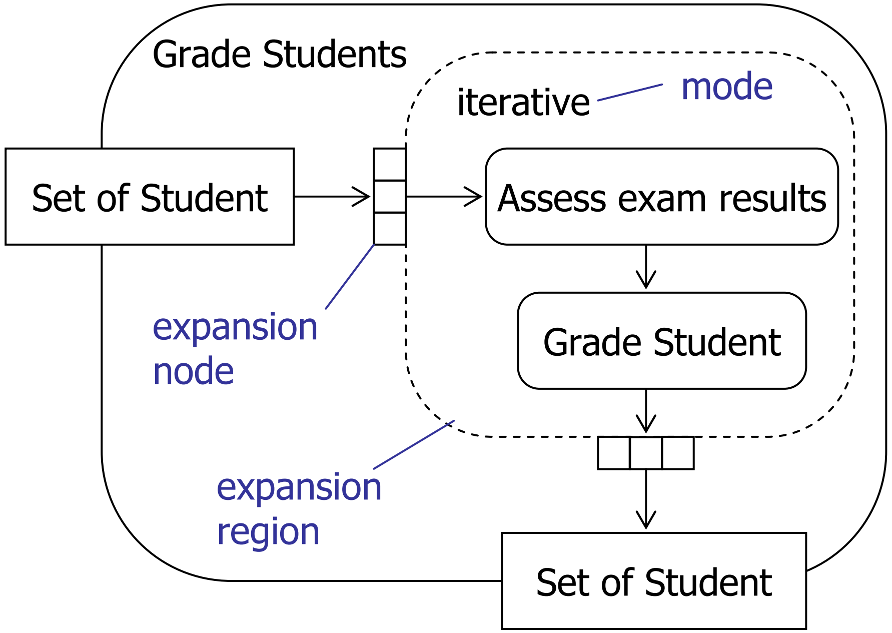
Exception handling



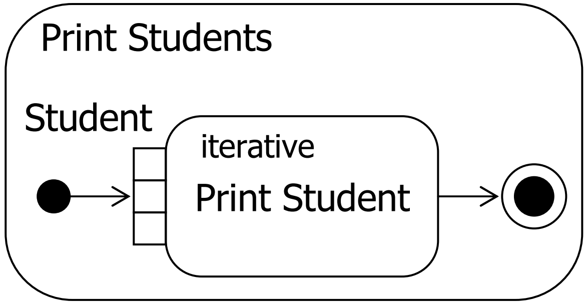
- Protected nodes have exception handlers:
  - When the exception object is raised in the protected node, flow is directed along an interrupting edge to the exception handler body

Expansion nodes

- Expansion node – an object node that represents a collection of objects flowing into or out of an *expansion region*
  - Output collections *must* correspond to input collections in collection type and object type!
- The expansion region is executed once per input element according to the keyword:
  - iterative – process sequentially
  - parallel – process in parallel
  - stream – process a stream of input objects

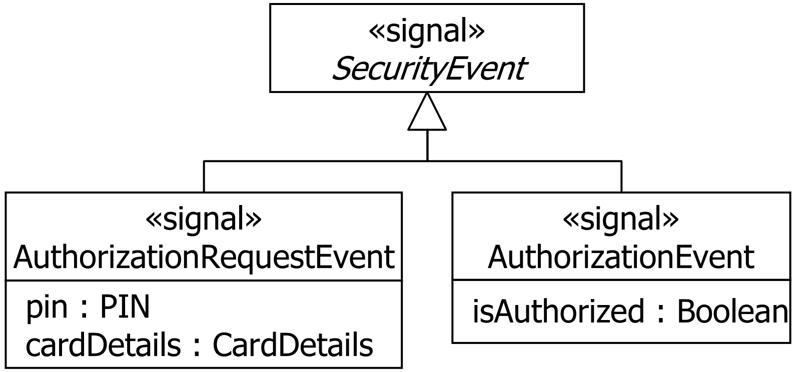


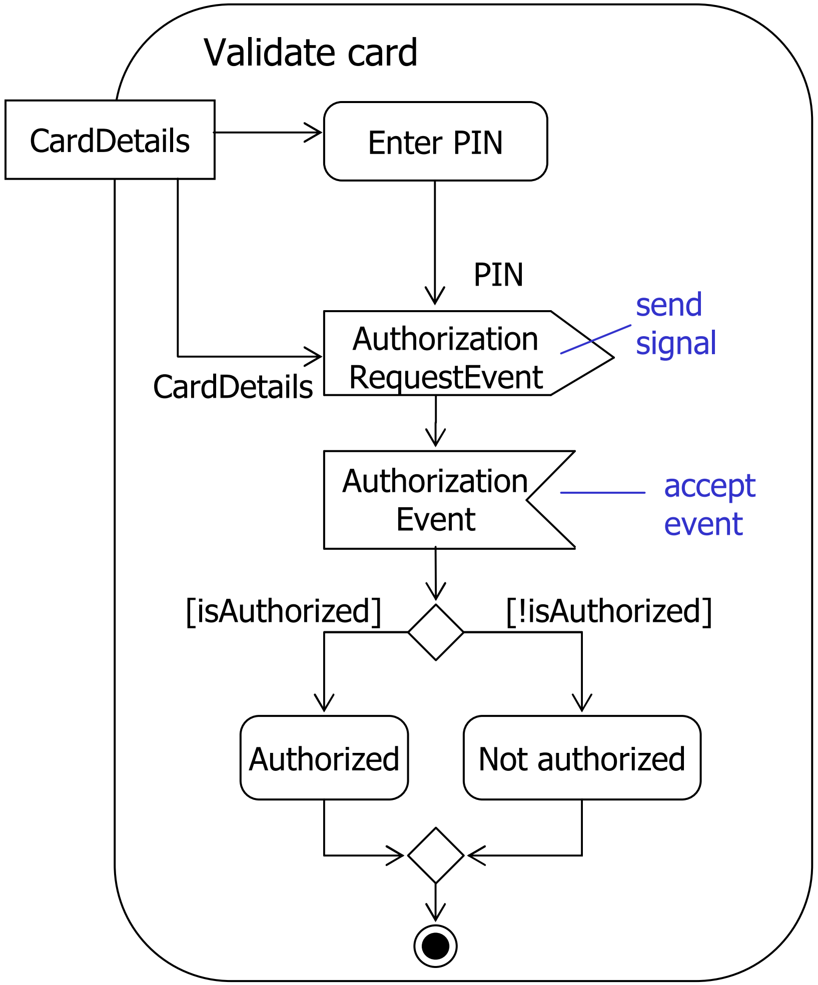
Expansion regions containing a single action - place the expansion node directly on the action



Sending signals and accepting events

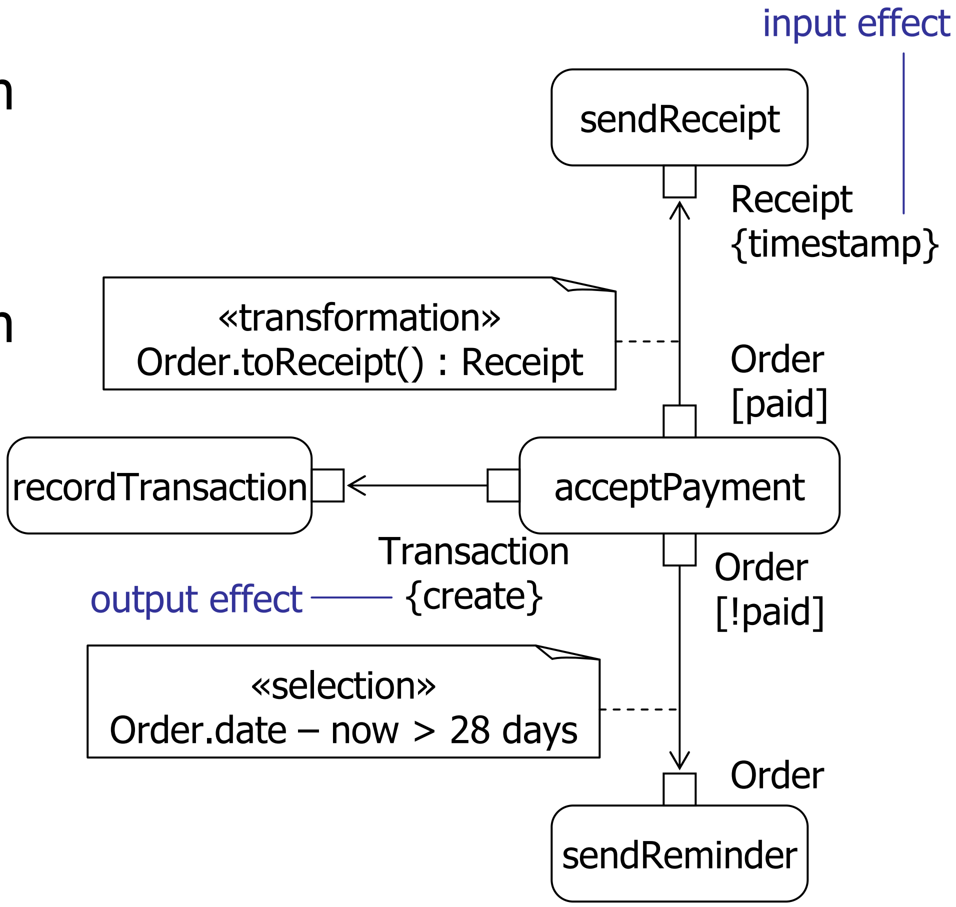
- Signals represent information passed asynchronously between objects
  - This information is modelled as attributes of a signal
  - A signal is a classifier stereotyped «signal»
- The accept event action asynchronously accepts event triggers which may be signals or other objects





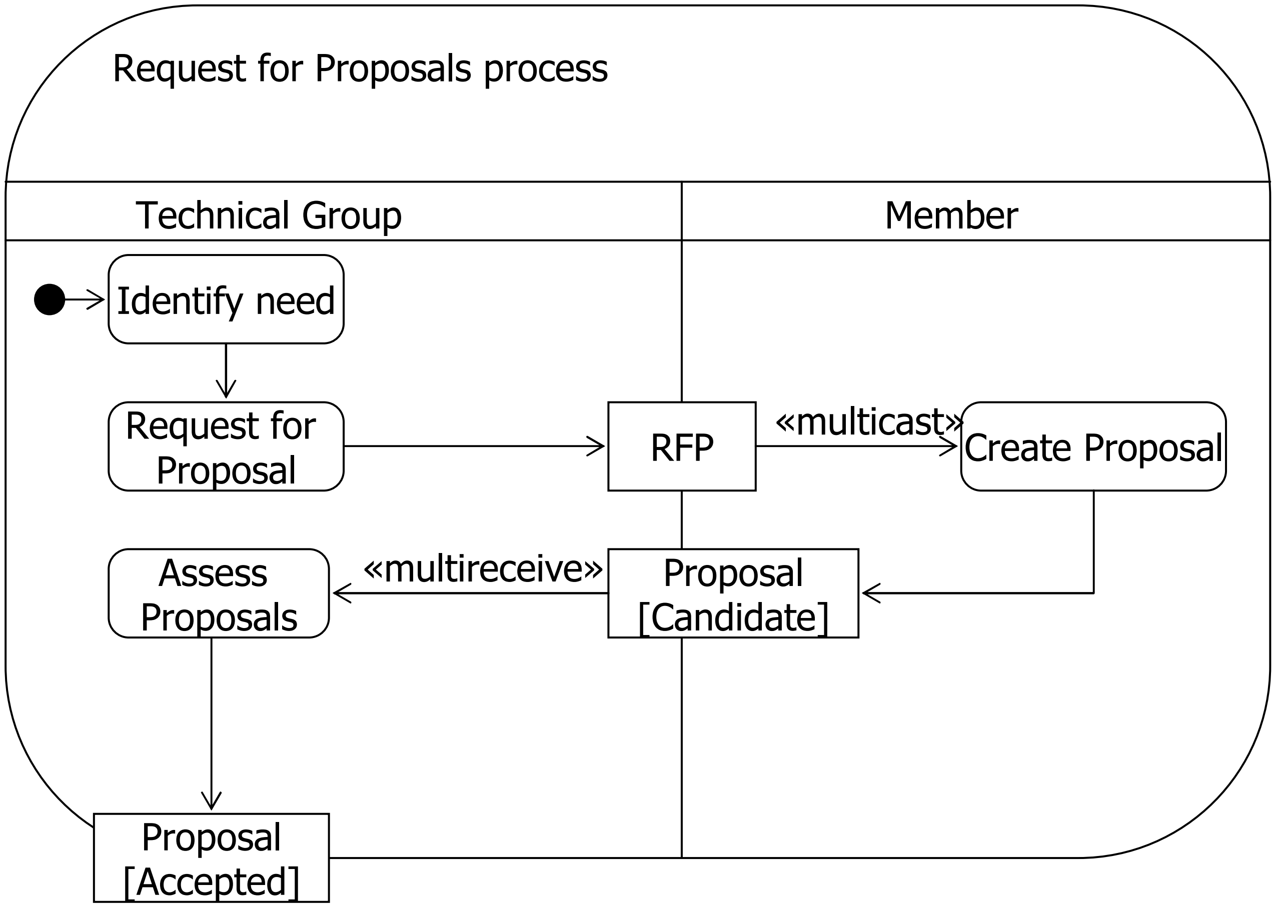
Advanced object flow

- Input effect
  - Specifies the effect of the action on objects flowing into it
- Output effect
  - Specifies the effect of the action on objects flowing out of it
- «selection»
  - the flow to selects objects that meet a specific criterion
- «transformation»
  - An object is transformed by the object flow

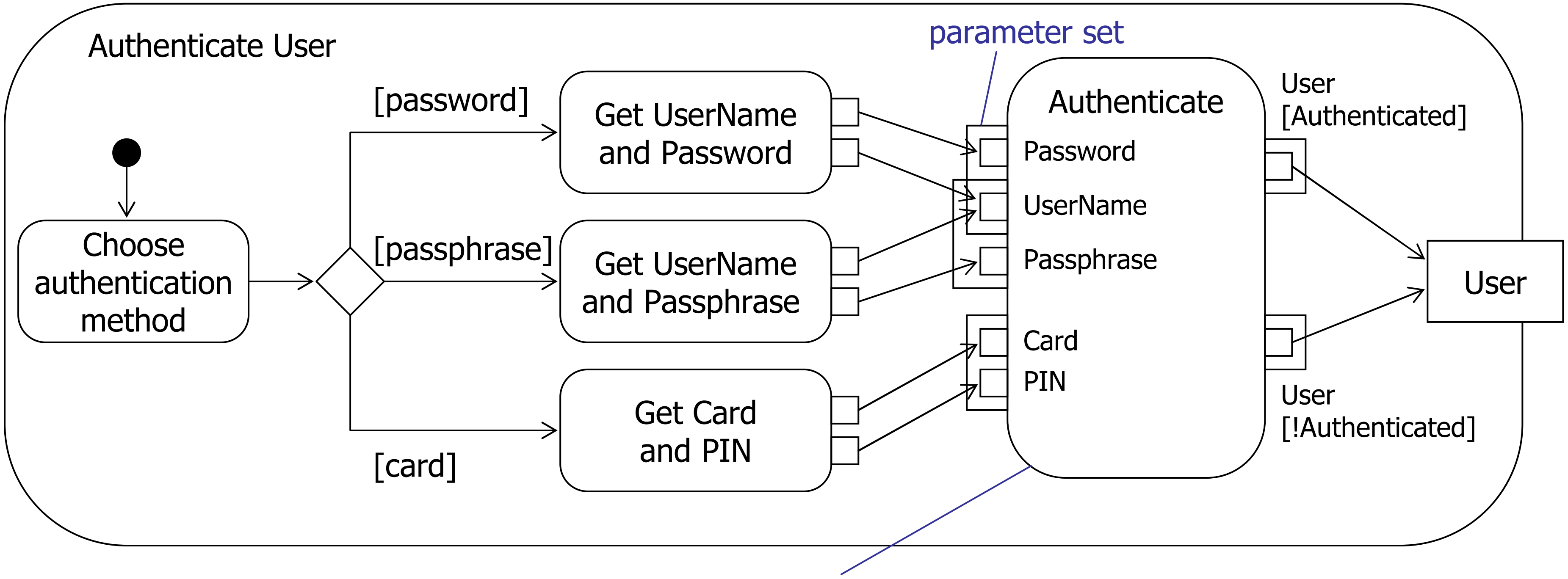


Multicast and multireceive

- A «multicast» object flow sends an object to multiple receivers
- A «multireceive» object flow receives an object from multiple receivers



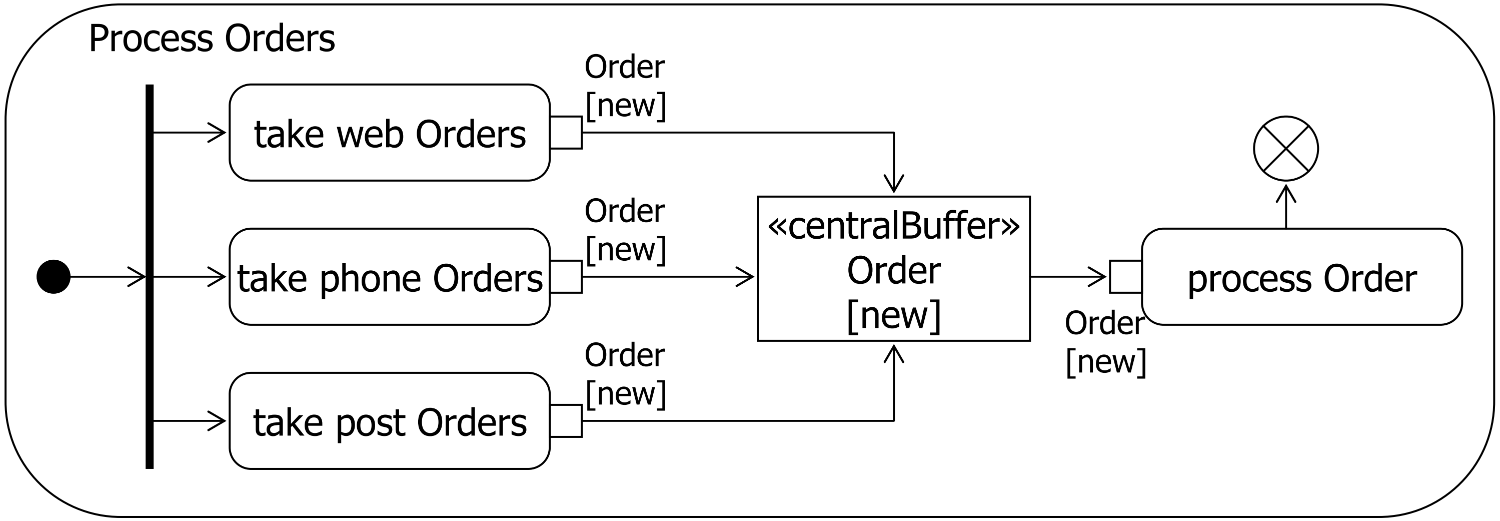
Parameter sets



input condition: (UserName AND Password) XOR (UserName AND Passphrase) XOR (Card AND PIN)
output: (User [Authenticated]) XOR (User [!Authenticated])

- Parameter sets provide alternative sets of input pins and output pins to an action
  - Only one input set and one output set may be chosen (XOR)

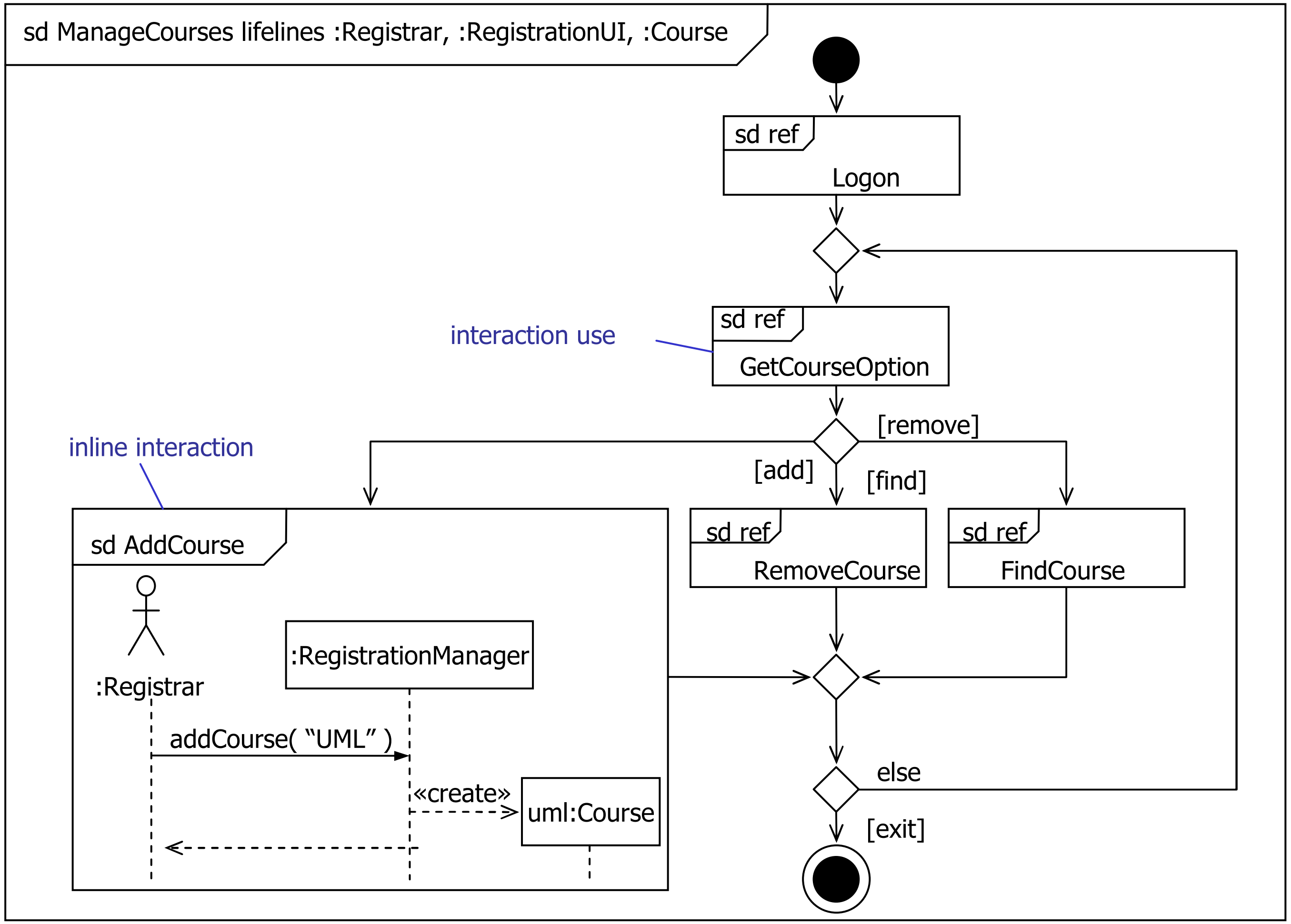
«centralBuffer» node



- Central buffer nodes accept multiple upstream object flows
- They hold the objects until downstream nodes are ready for them

Interaction overview diagrams

- Model the high level flow of control between interactions
- Show interactions and interaction occurrences
- Have activity diagram syntax



Summary

- In this section we have looked at some of the more advanced features of activity diagrams:
  - Interruptible activity regions
  - Exception handlers
  - Expansion nodes
  - Advanced object flow
  - Multicast and multireceive
  - Parameter sets
  - Central buffer nodes
  - Interaction overview diagrams