Object Modeling with UML: Advanced Modeling

Karin Palmkvist, Bran Selic, and Jos Warmer

March 2000

UNIFIED
MODELING
LANGUAGE

Overview

- Tutorial series
- UML Overview
- Advanced Modeling
  - Part 1: Model Management
    - Karin Palmkvist, Enea Data
  - Part 2: Extension Mechanisms and Profiles
    - Bran Selic, ObjecTime Limited
  - Part 3: Object Constraint Language (OCL)
    - Jos Warmer, Klasse Objecten

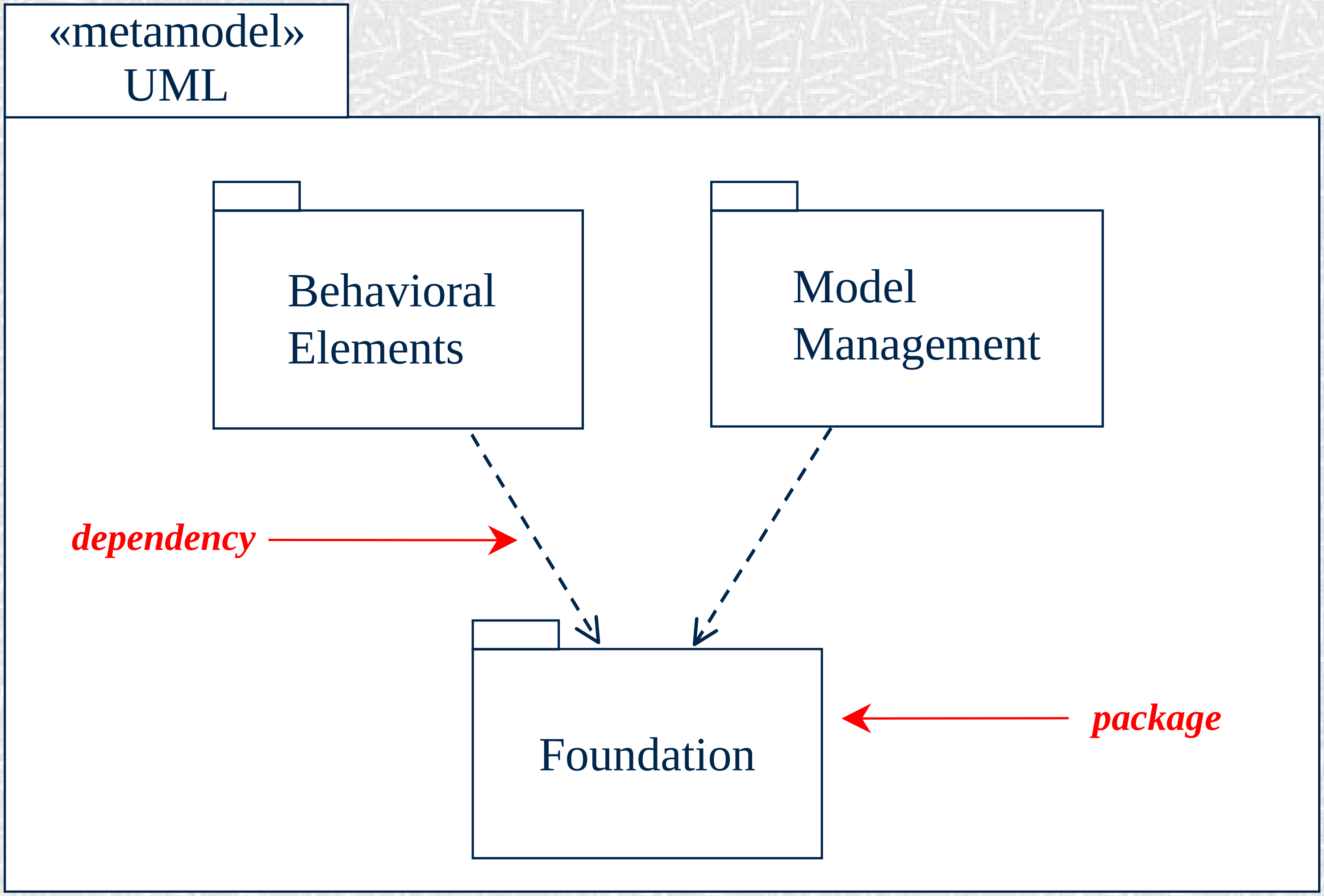
Tutorial Series

- Introduction to UML
  - November 1999, Cambridge, US
- Behavioral Modeling with UML
  - January 2000, Mesa, Arizona, US
- Advanced Modeling with UML
  - March 2000, Denver, US
- Metadata Integration with UML, XMI and MOF
  - June 2000, Oslo, Norway

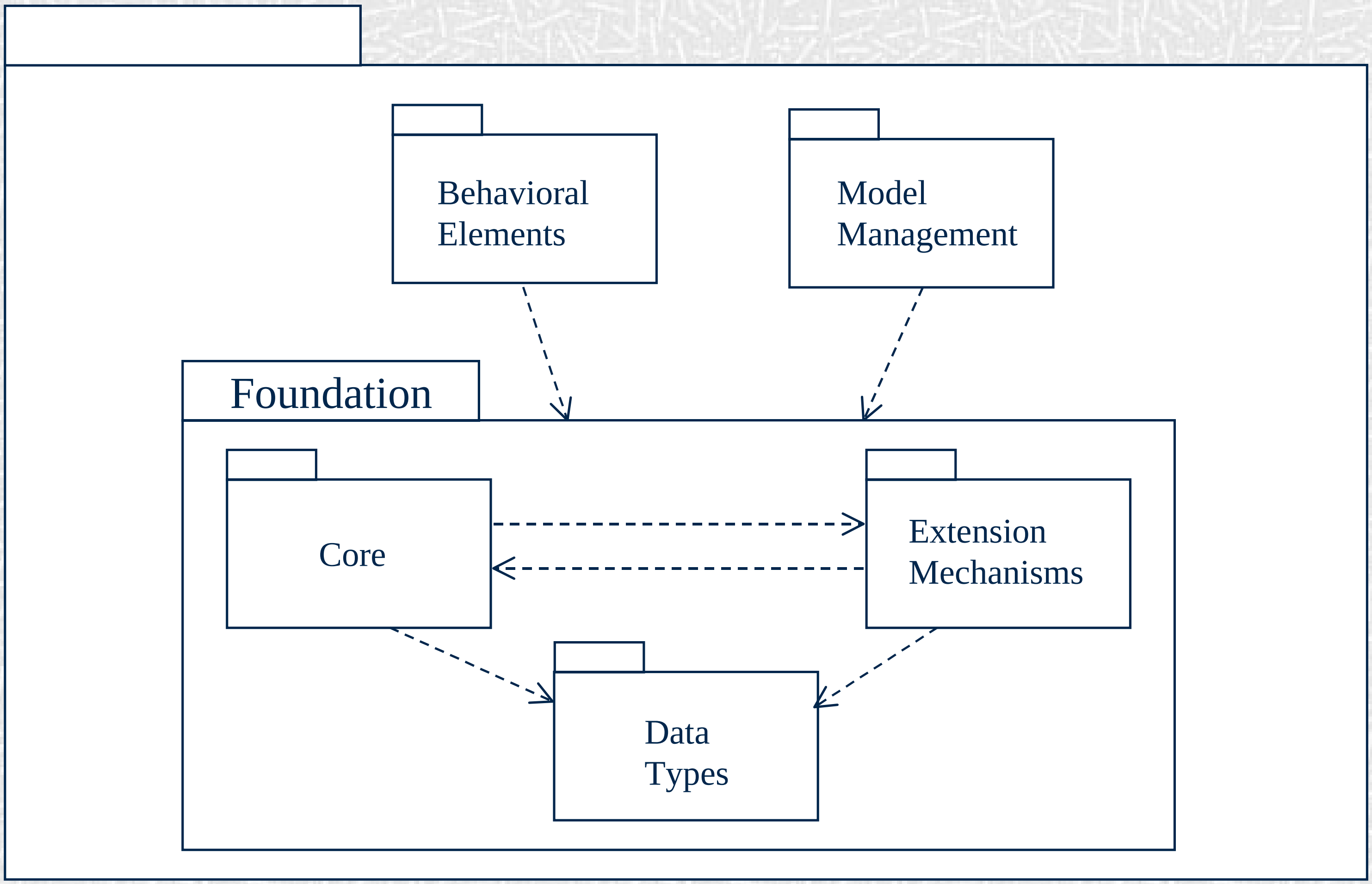
Tutorial Focus: the Language

- language = syntax + semantics
  - syntax = language elements (e.g. words) are assembled into expressions (e.g. phrases, clauses)
  - semantics = the meanings of the syntactic expressions
- *UML Notation Guide* – defines UML’s graphic syntax
- *UML Semantics* – defines UML’s semantics

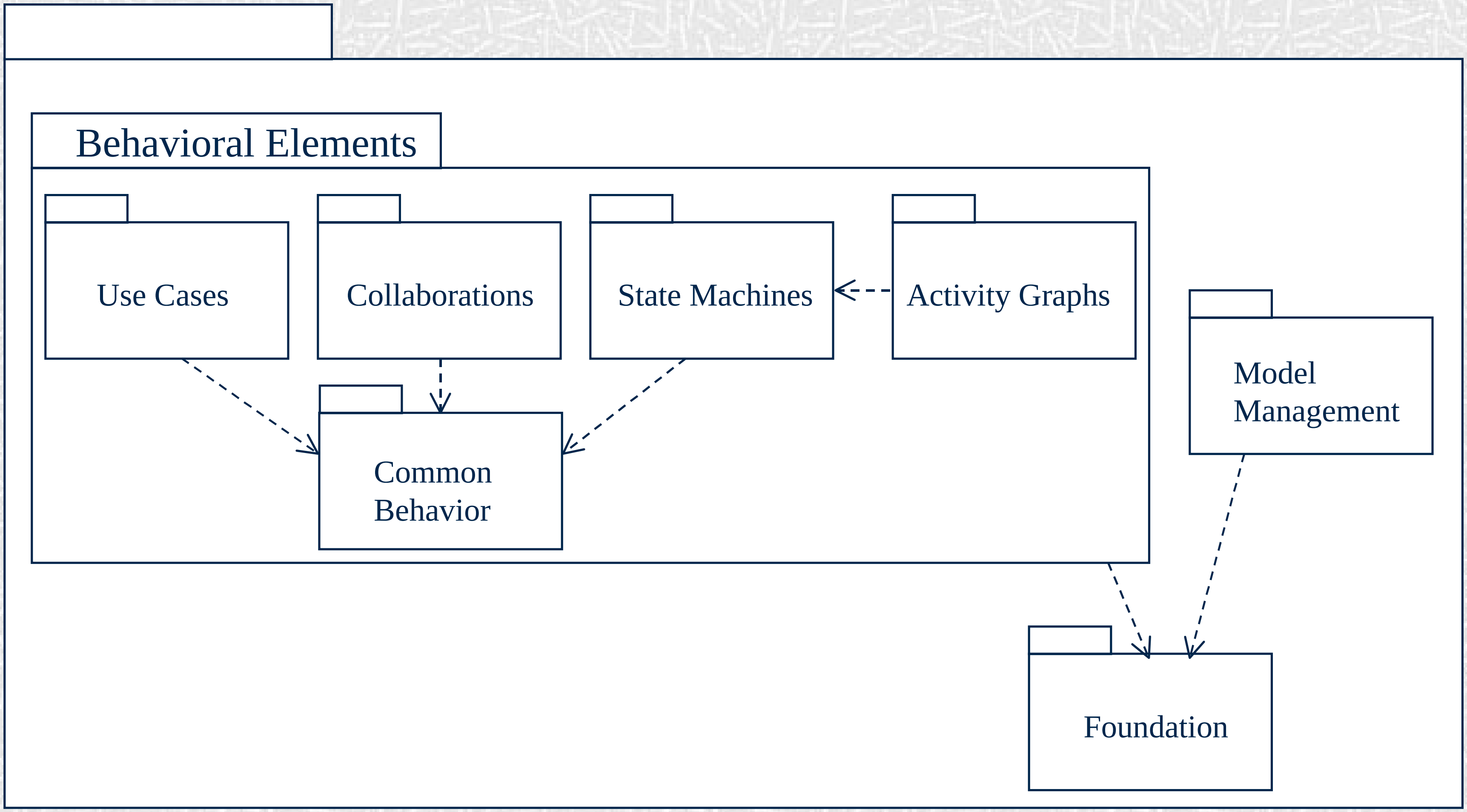
UML Overview



UML Overview



UML Overview



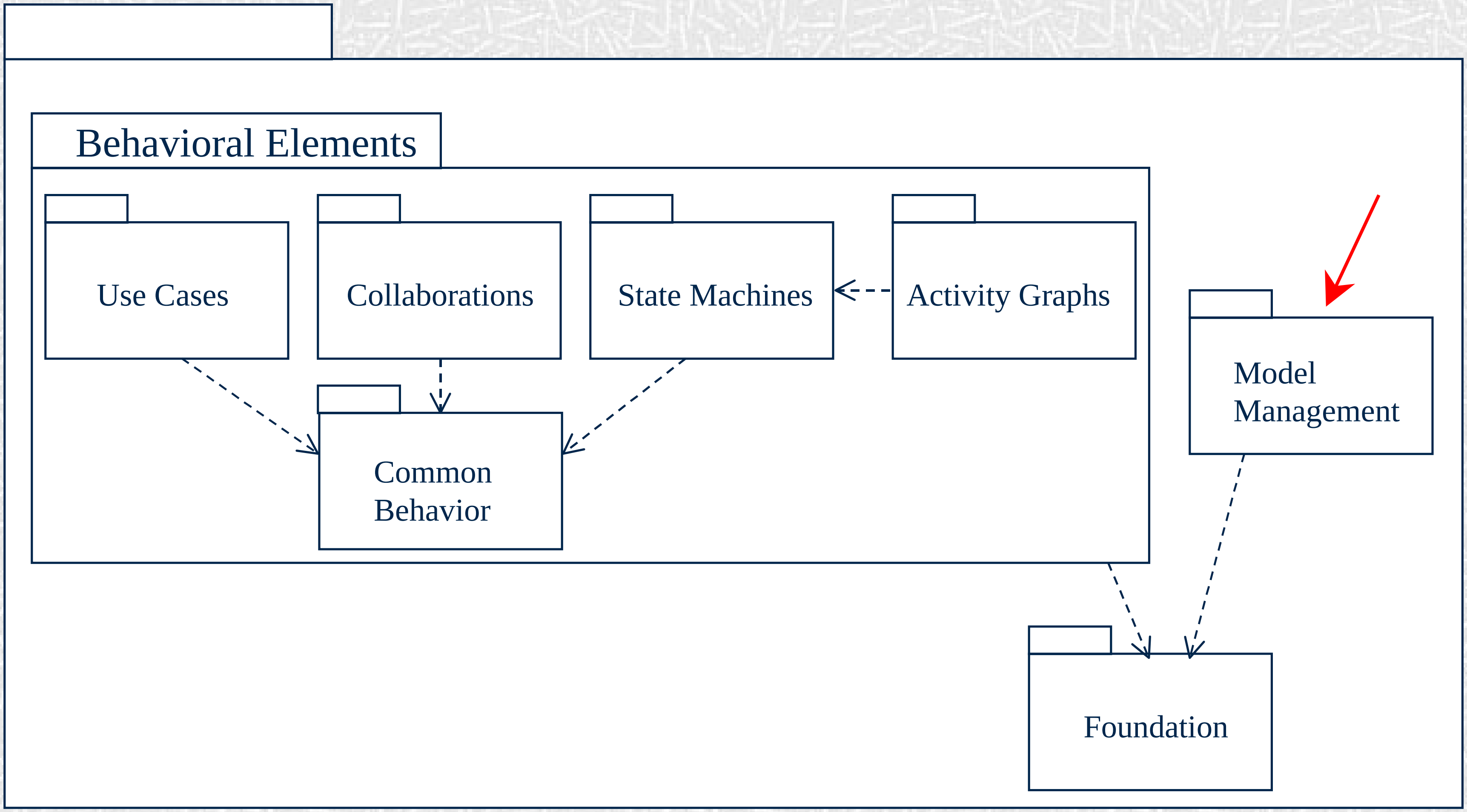
Advanced Modeling with UML

- Part 1: Model Management

Karin Palmkvist, Enea Data
karin.palmkvist@enea.se

- Part 2: Extension Mechanisms and Profiles
- Part 3: Object Constraint Language (OCL)

UML Overview



Model Management Overview

- Package
- Subsystem
- Model

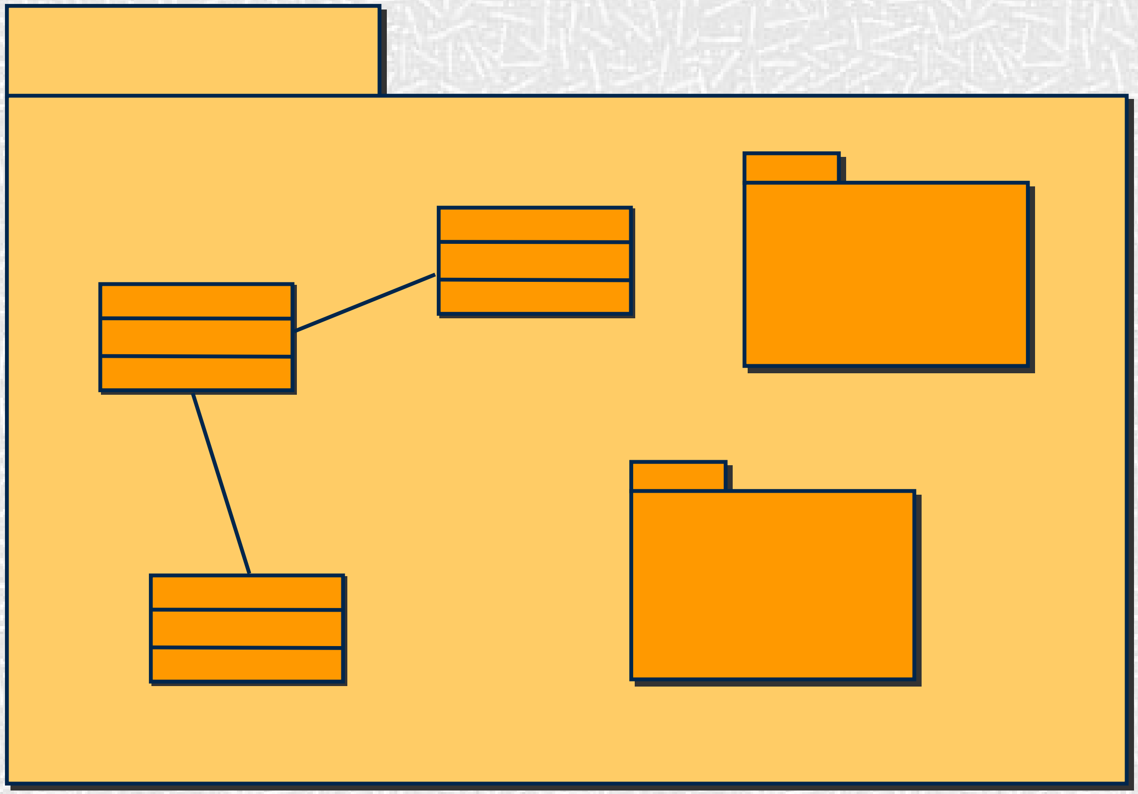
Unifying Concepts

- Grouping - Packages, Subsystems, and Models all group other model elements, although with very differing purposes
- Other grouping elements include Classes and Components

Package

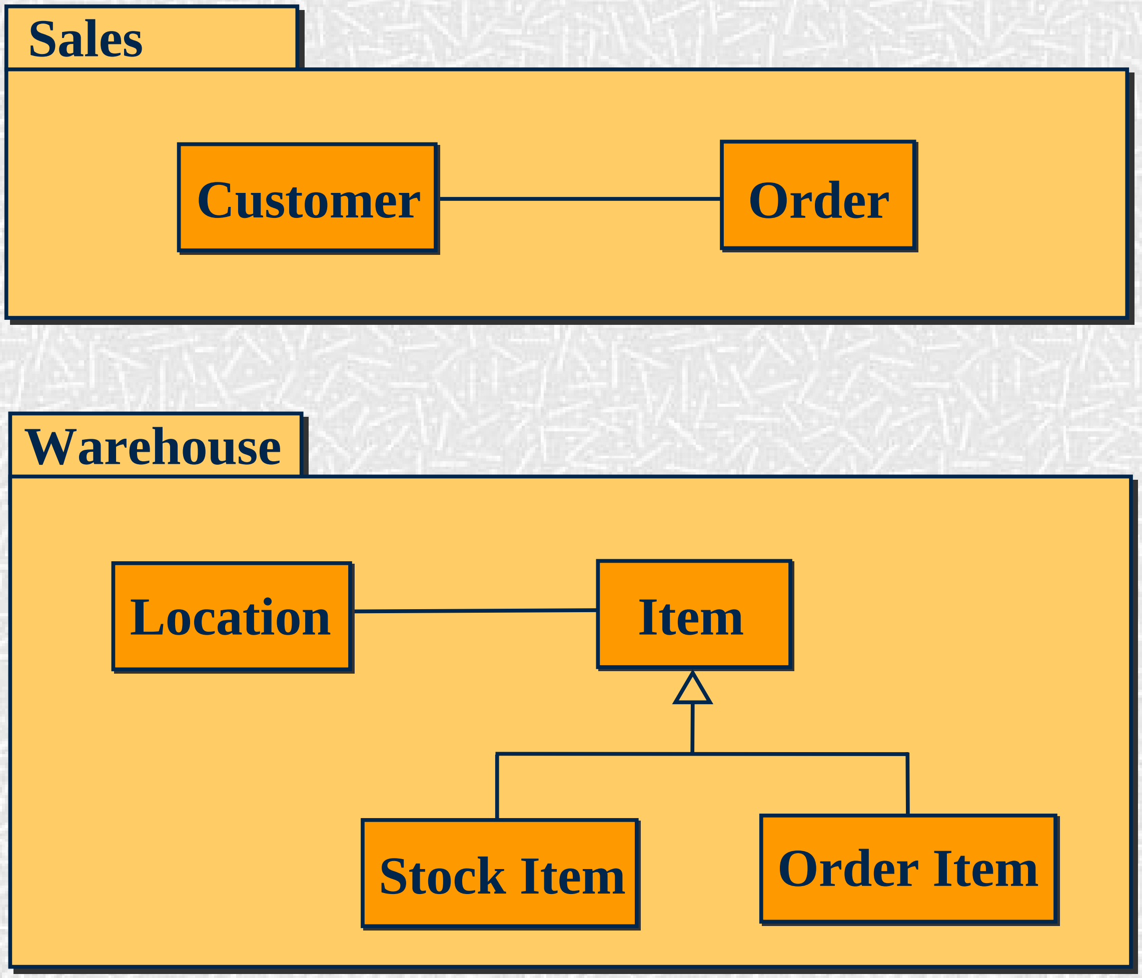
- What are Packages
- Core Concepts
- Diagram Tour
- When to Use Packages
- Modeling Tips

Package



A package is a grouping of model elements

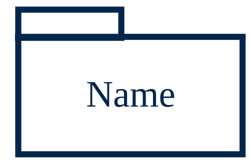
Package – Example



Package

- A package can contain model elements of different kinds
- In particular, there can be a containment hierarchy of nested packages
- A package defines a namespace for its contents
- Packages can be used for different purposes

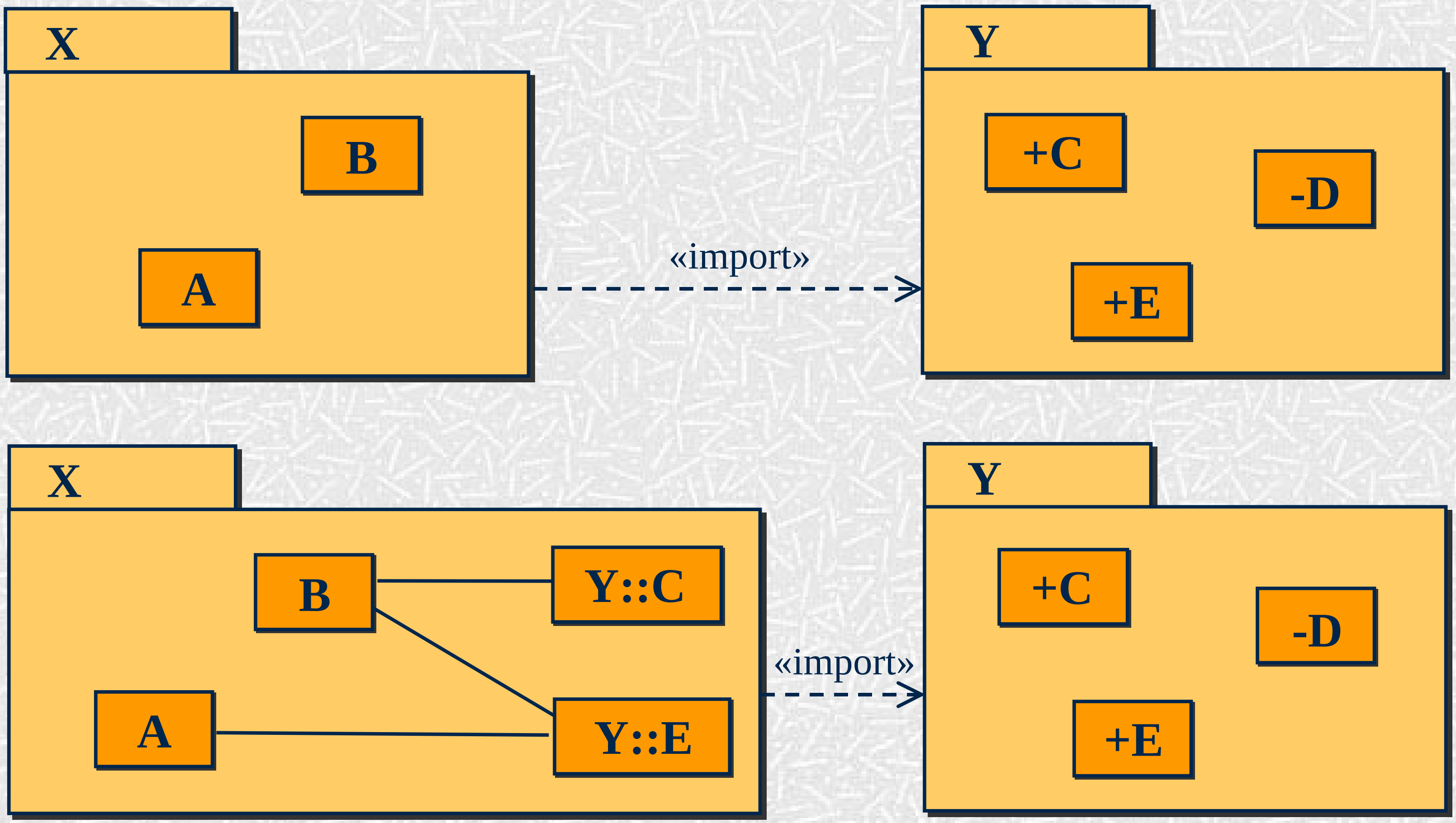
Core Concepts

Construct	Description	Syntax
Package	A grouping of model elements.	
Import	A dependency indicating that the public contents of the target package are added to the namespace of the source package.	
Access	A dependency indicating that the public contents of the target package are available in the namespace of the source package.	

Visibility

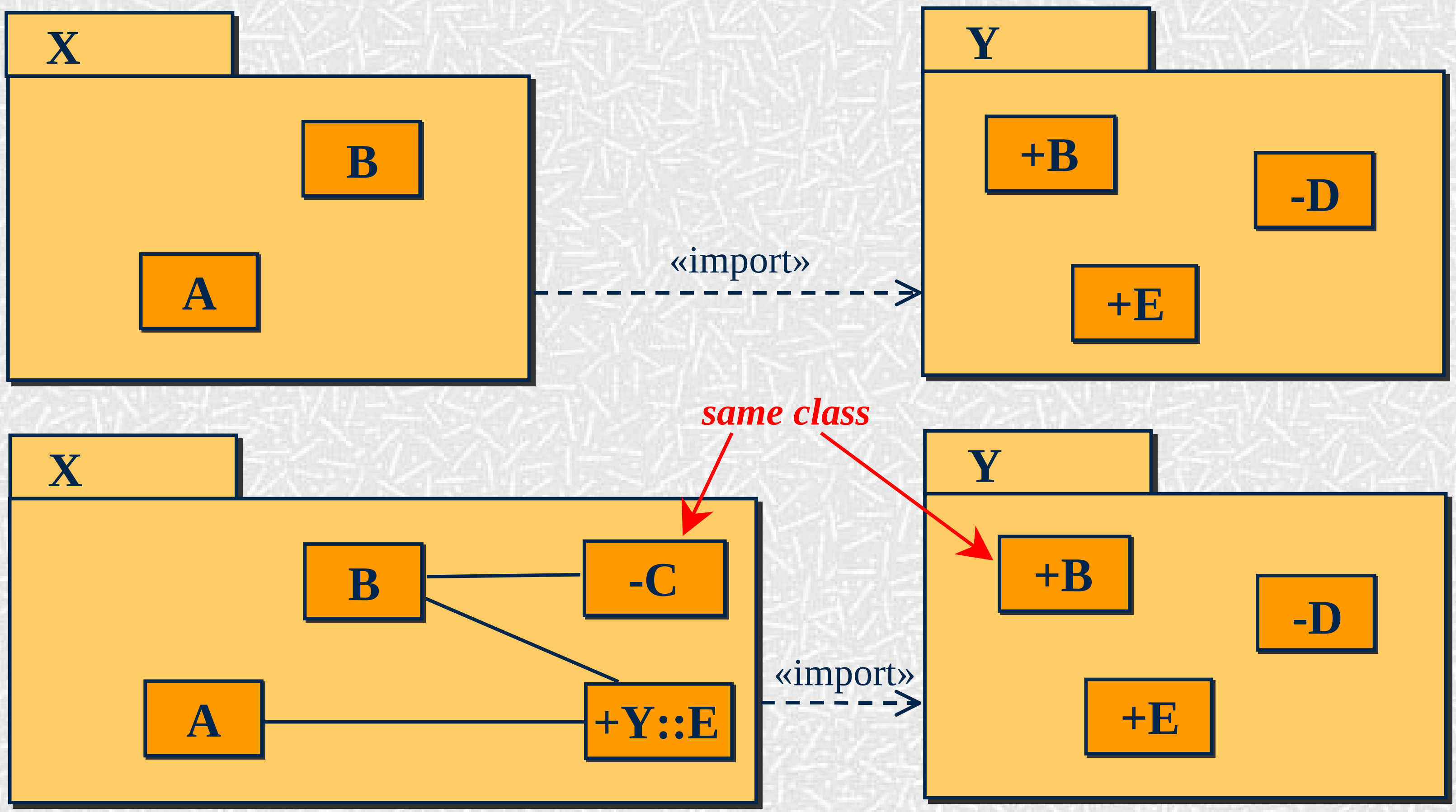
- A *public* element is visible to elements outside the package, denoted by ‘+’
- A *protected* element is visible only to elements within inheriting packages, denoted by ‘#’
- A *private* element is not visible at all to elements outside the package, denoted by ‘-’

Import



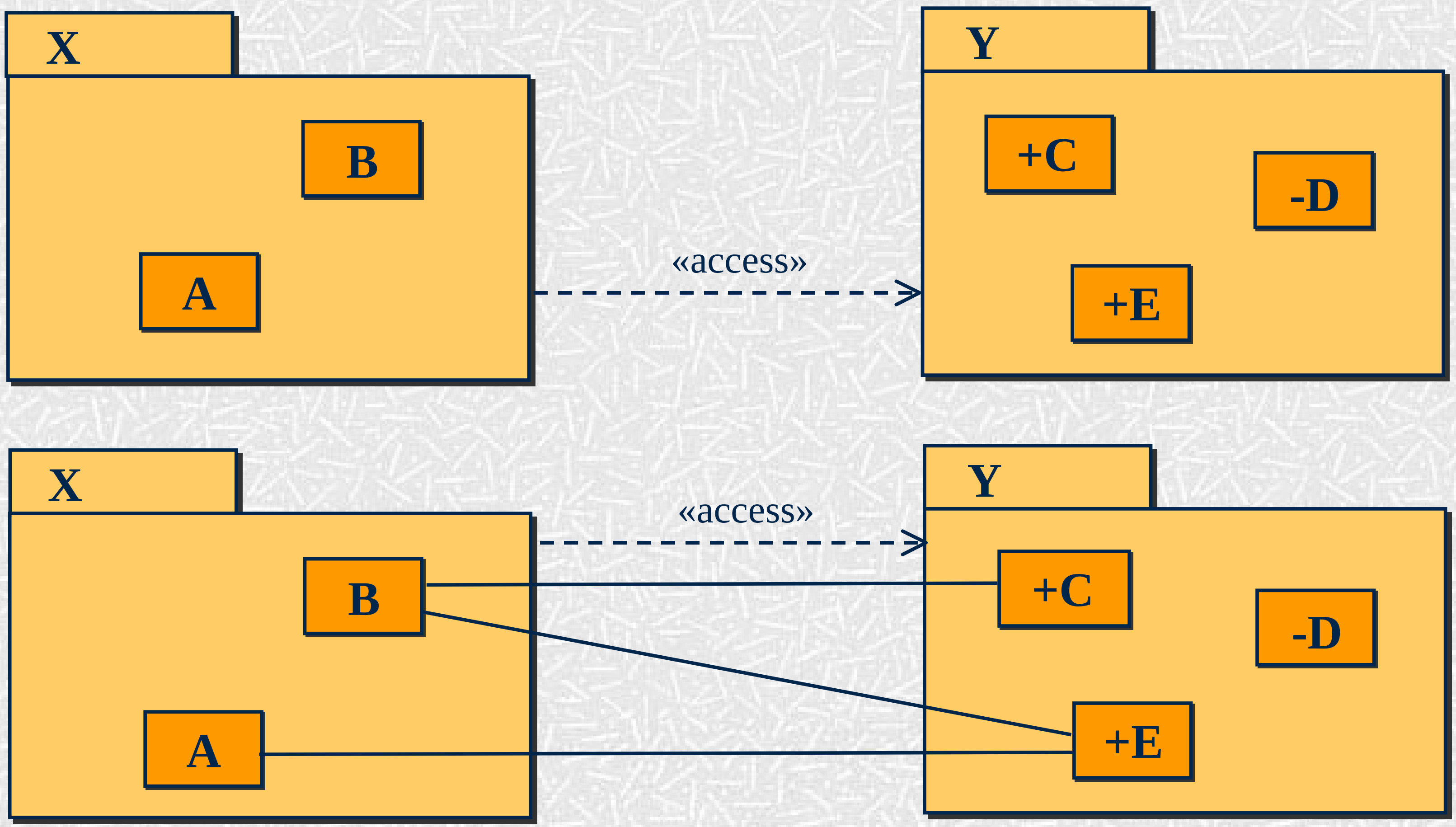
The associations are owned by package X

Import – Alias



An imported element can be given a local alias and a local visibility

Access



The associations are owned by package X

Import vs. Access

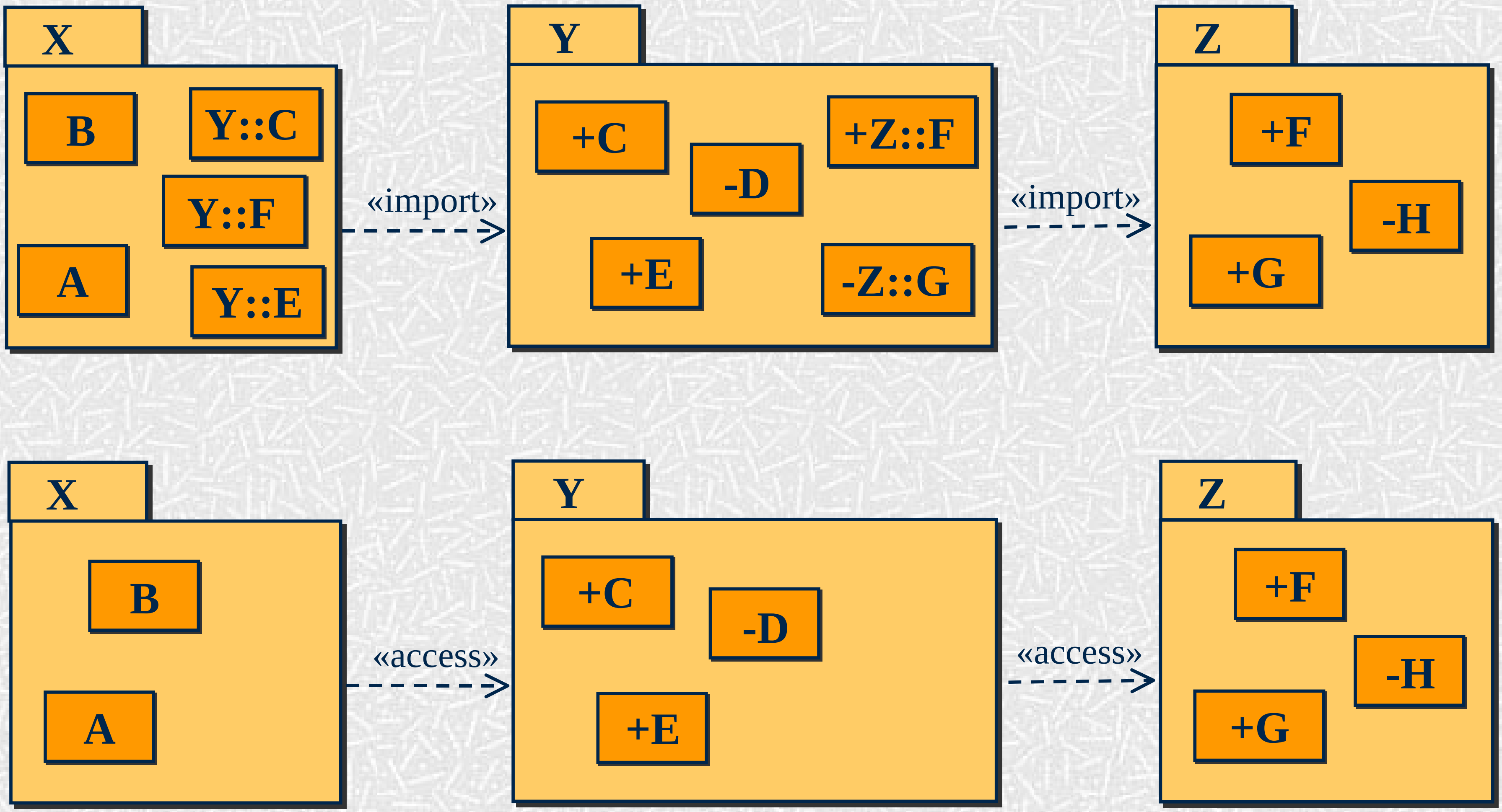
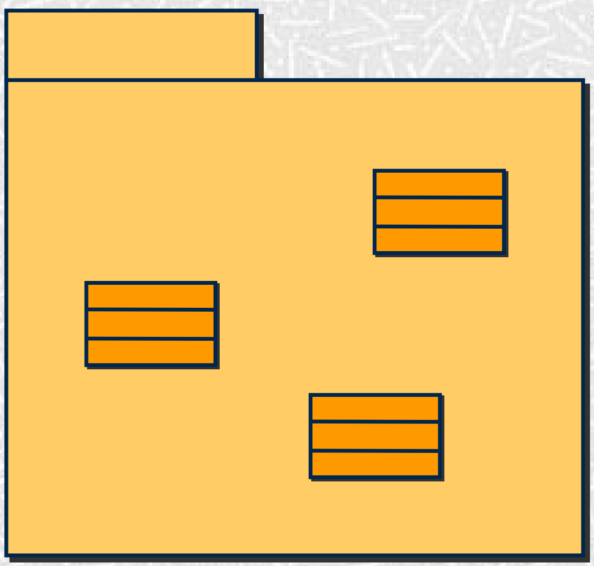
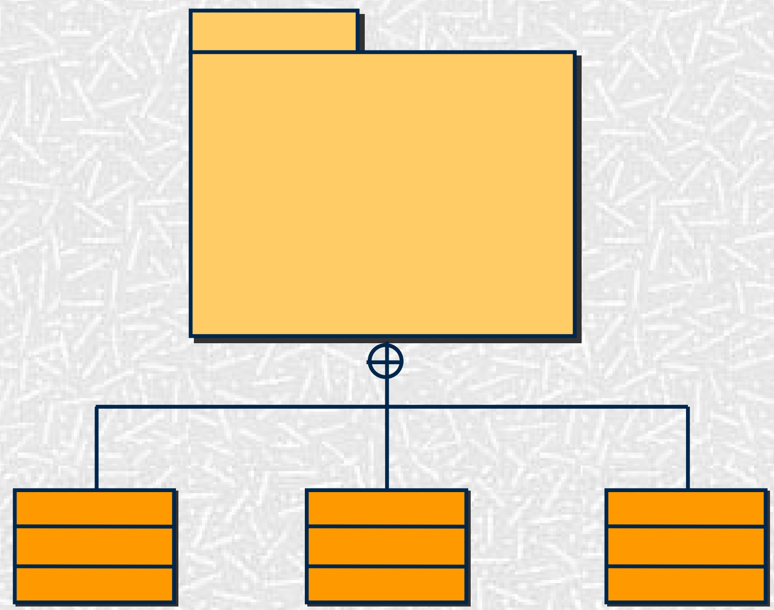


Diagram Tour

- Packages are shown in static diagrams
- Two equivalent ways to show containment:





When to Use Packages

- To create an overview of a large set of model elements
- To organize a large model
- To group related elements
- To separate namespaces

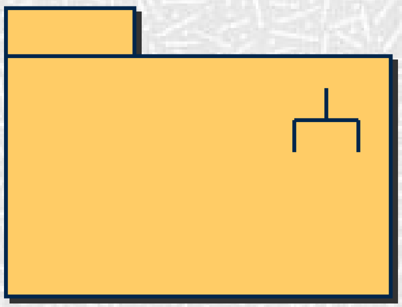
Modeling Tips – Package

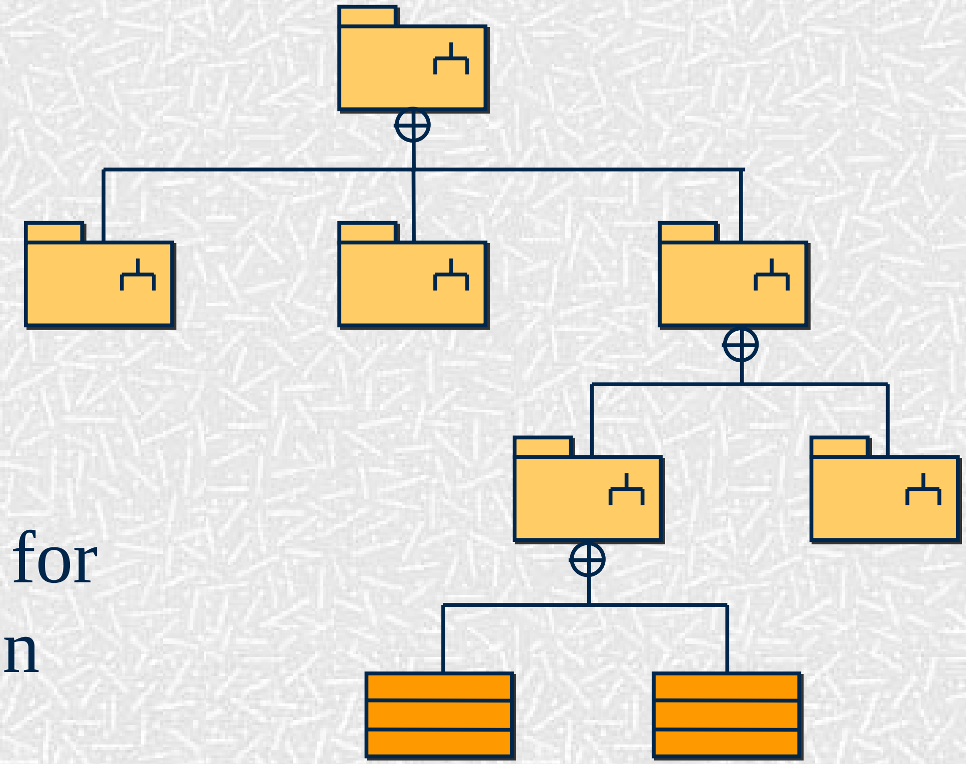
- Gather model elements with strong cohesion in one package
- Keep model elements with low coupling in different packages
- Minimize relationships, especially associations, between model elements in different packages
- Namespace implication: an element imported into a package does not “know” what is done to it in the imported package

Subsystem

- What are Subsystems
- Core Concepts
- Diagram Tour
- When to Use Subsystems
- Modeling Tips

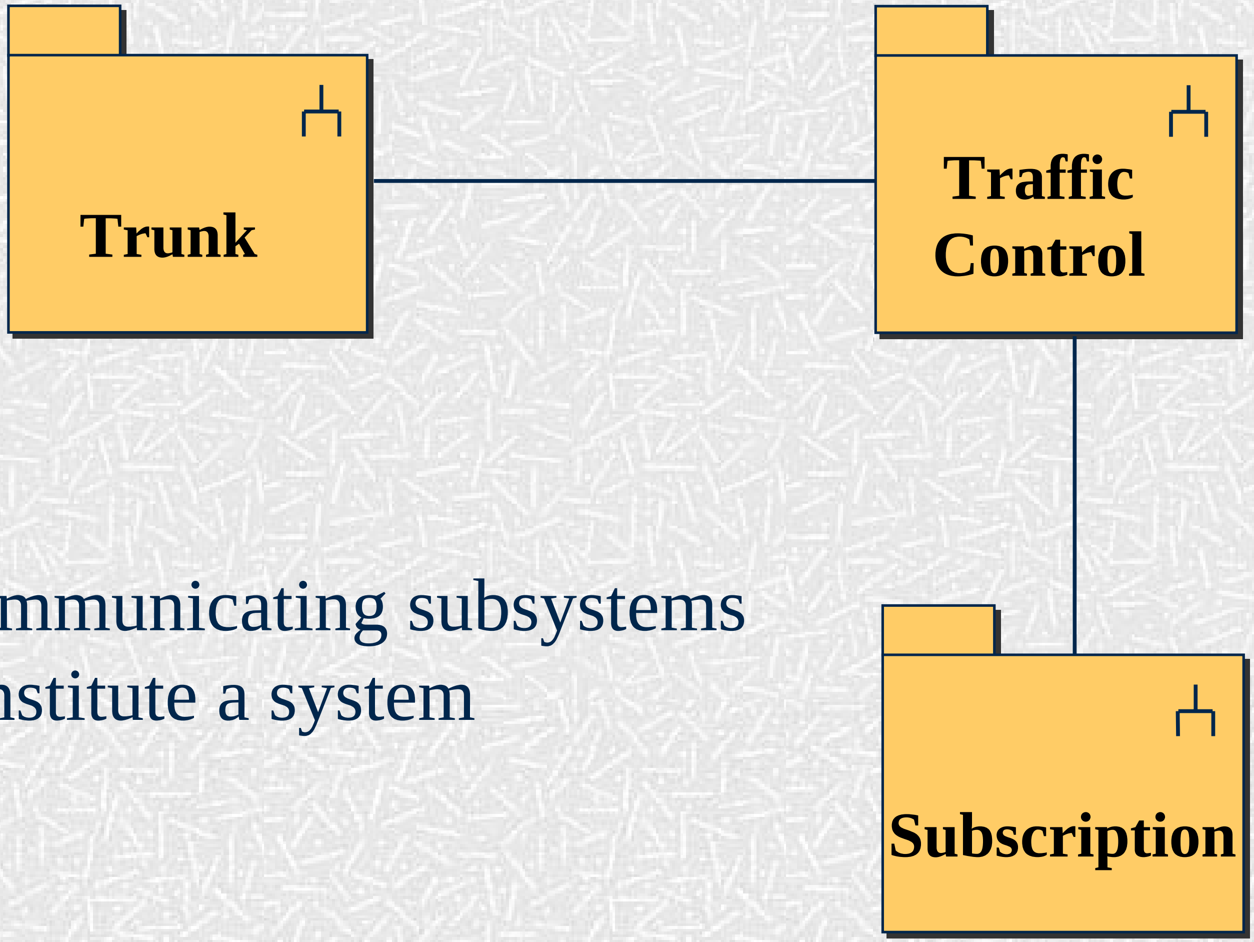
Subsystem





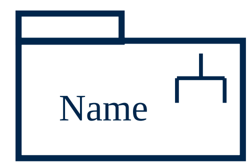
Subsystems are used for system decomposition

Subsystem – Example



Communicating subsystems constitute a system

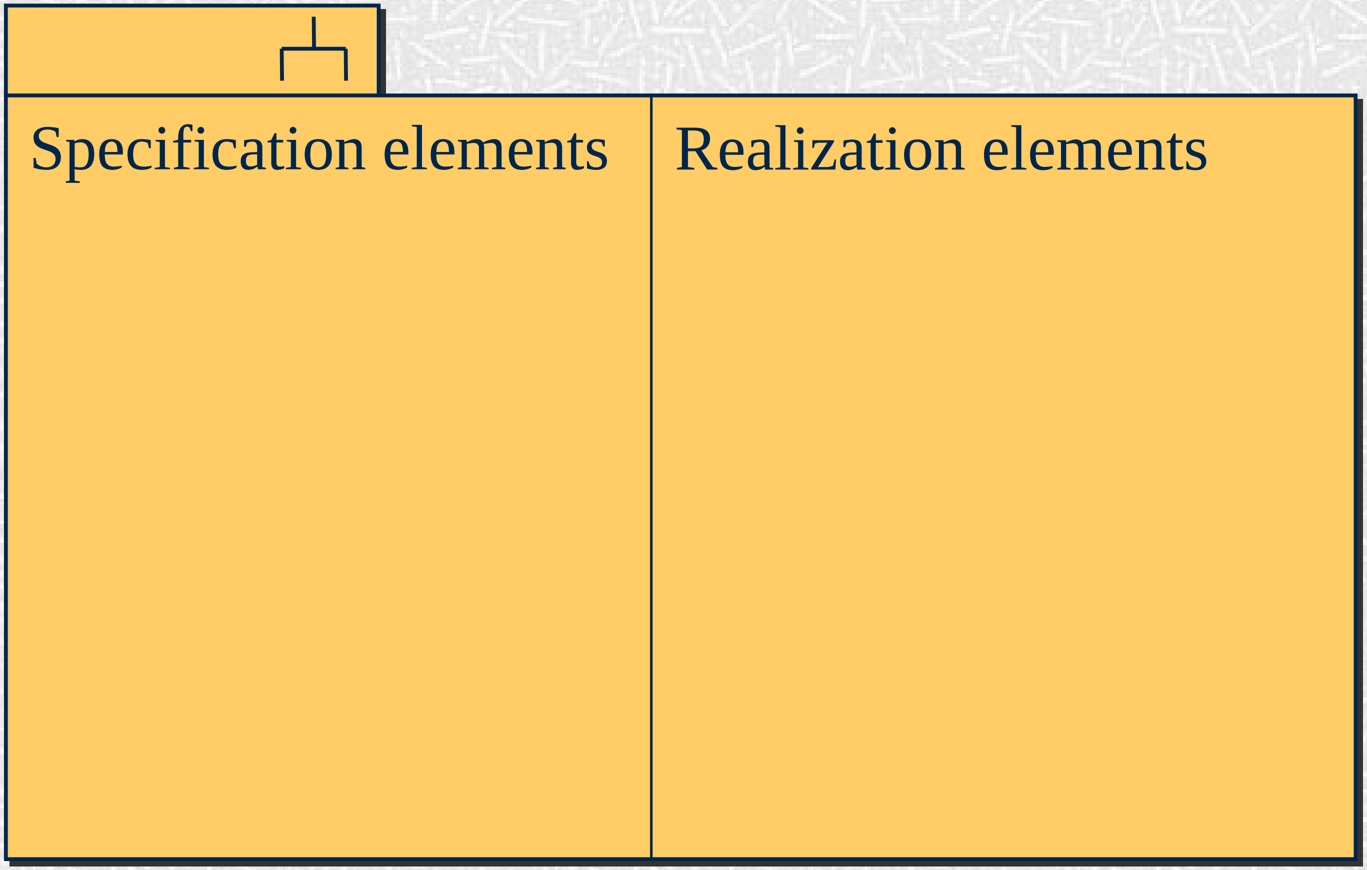
Core Concepts

Construct	Description	Syntax
Subsystem	A grouping of model elements that represents a behavioral unit in a physical system.	

Subsystem Aspects

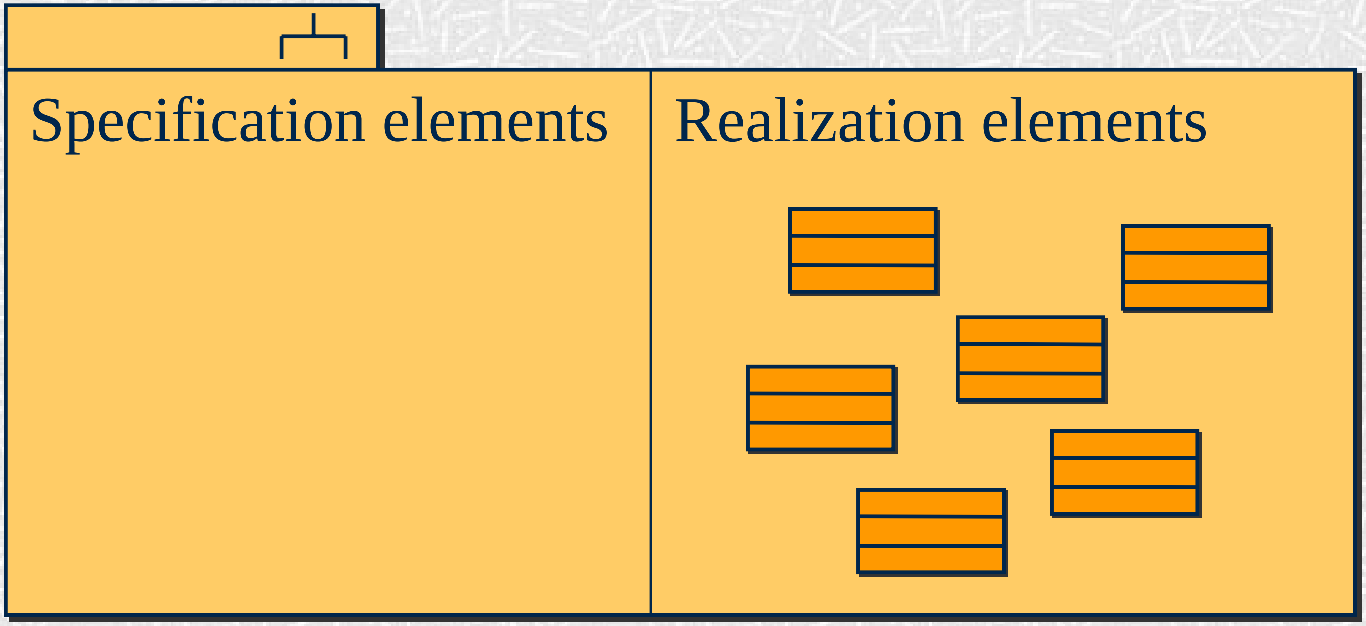
- A subsystem has two aspects:
  - An *external* view, showing the services provided by the subsystem
  - An *internal* view, showing the realization of the subsystem
- There is a mapping between the two aspects

Subsystem Aspects



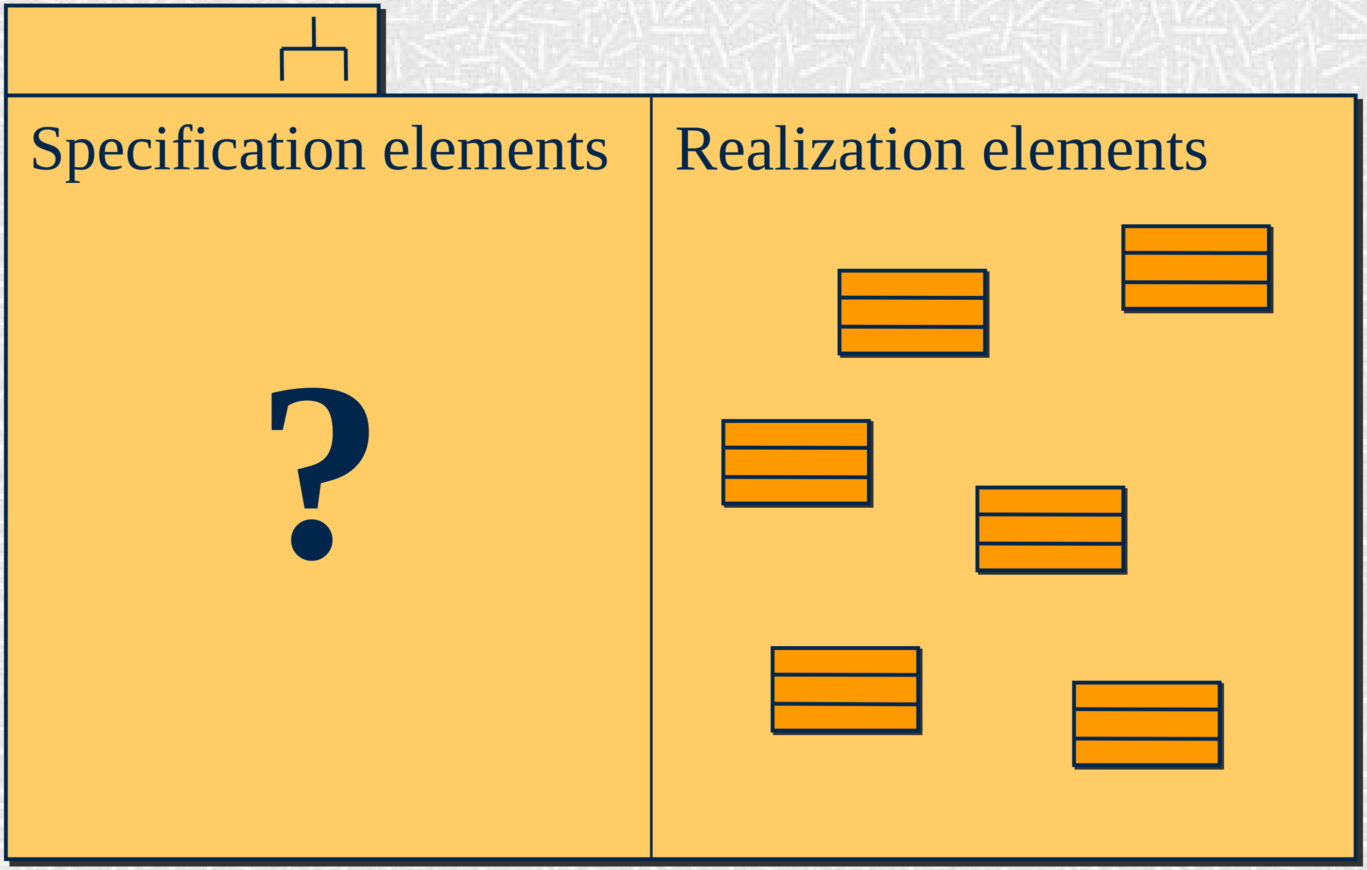
A subsystem has a specification and a realization

Subsystem Realization



- The subsystem realization defines the actual contents of the subsystem
- The subsystem realization typically consists of classes and their relationships, or a contained hierarchy of subsystems with classes as leaves

Subsystem Specification



The subsystem specification defines the external view of the subsystem

Subsystem Specification

The subsystem specification:

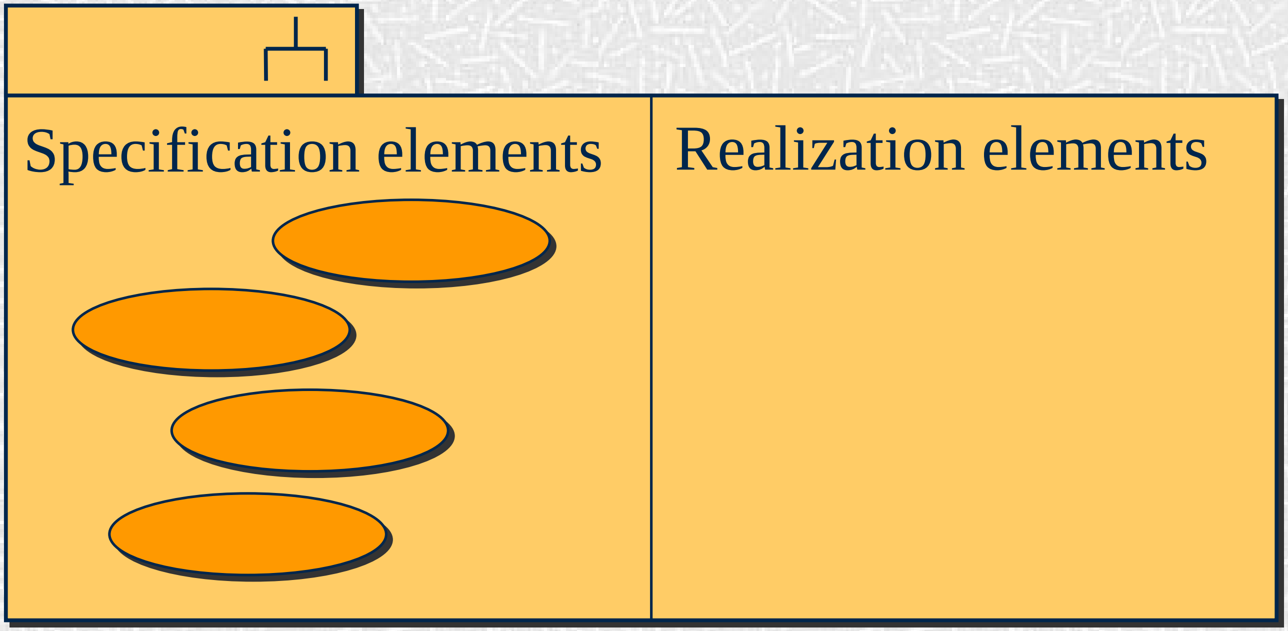
- describes the services offered by the subsystem
- describes the externally experienced behavior of the subsystem
- does not reveal the internal structure of the subsystem
- describes the interface of the subsystem

Specification Techniques

- The Use Case approach
- The State Machine approach
- The Logical Class approach
- The Operation approach

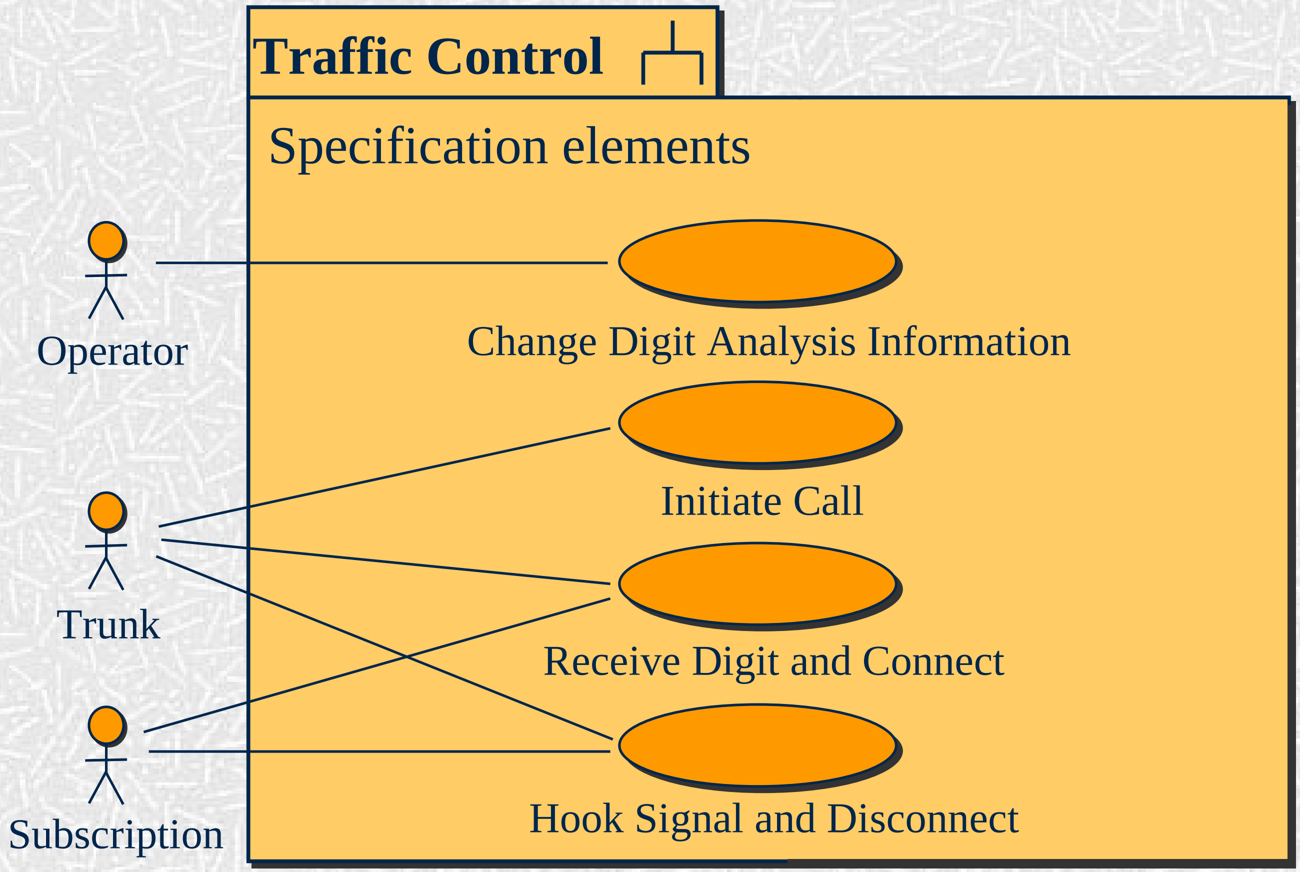
…and combinations of these.

Use Case Approach

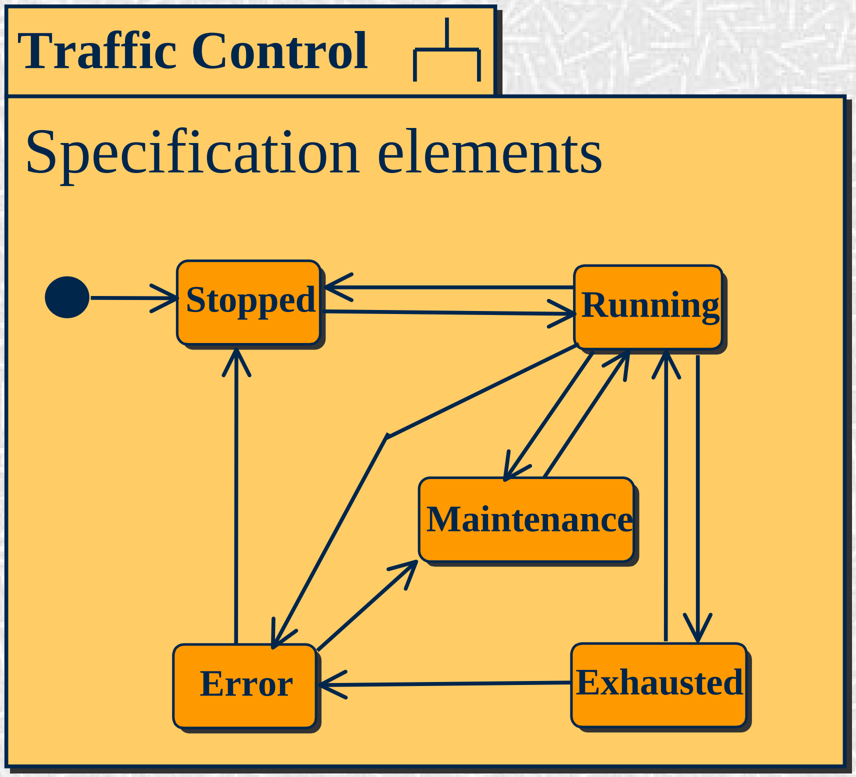


- For subsystem services used in certain sequences
- When the specification is to be understood by non-technical people
- For complex behavior

Use Case Approach – Example

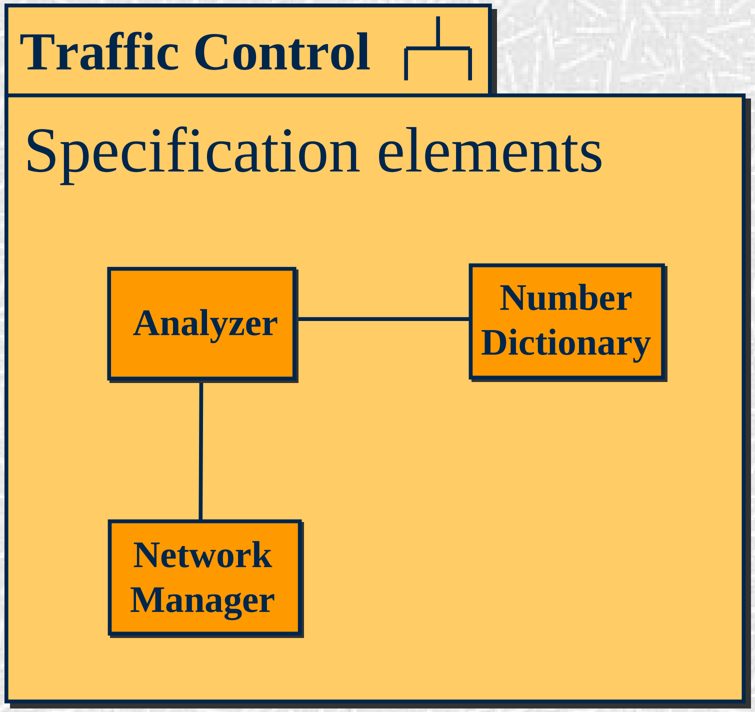


State Machine Approach



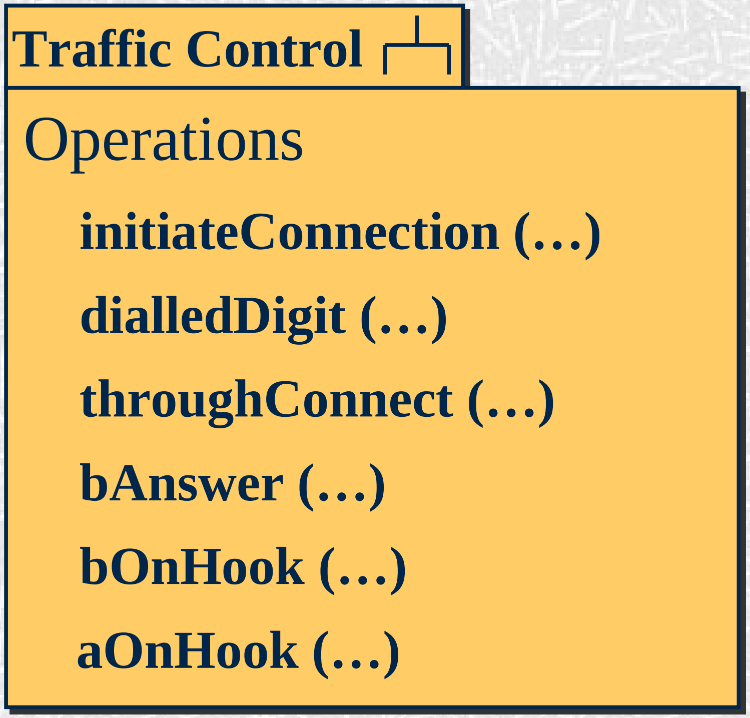
- For subsystems with state dependent behavior
- Focuses on the states of the subsystem and the transitions between them

Logical Class Approach



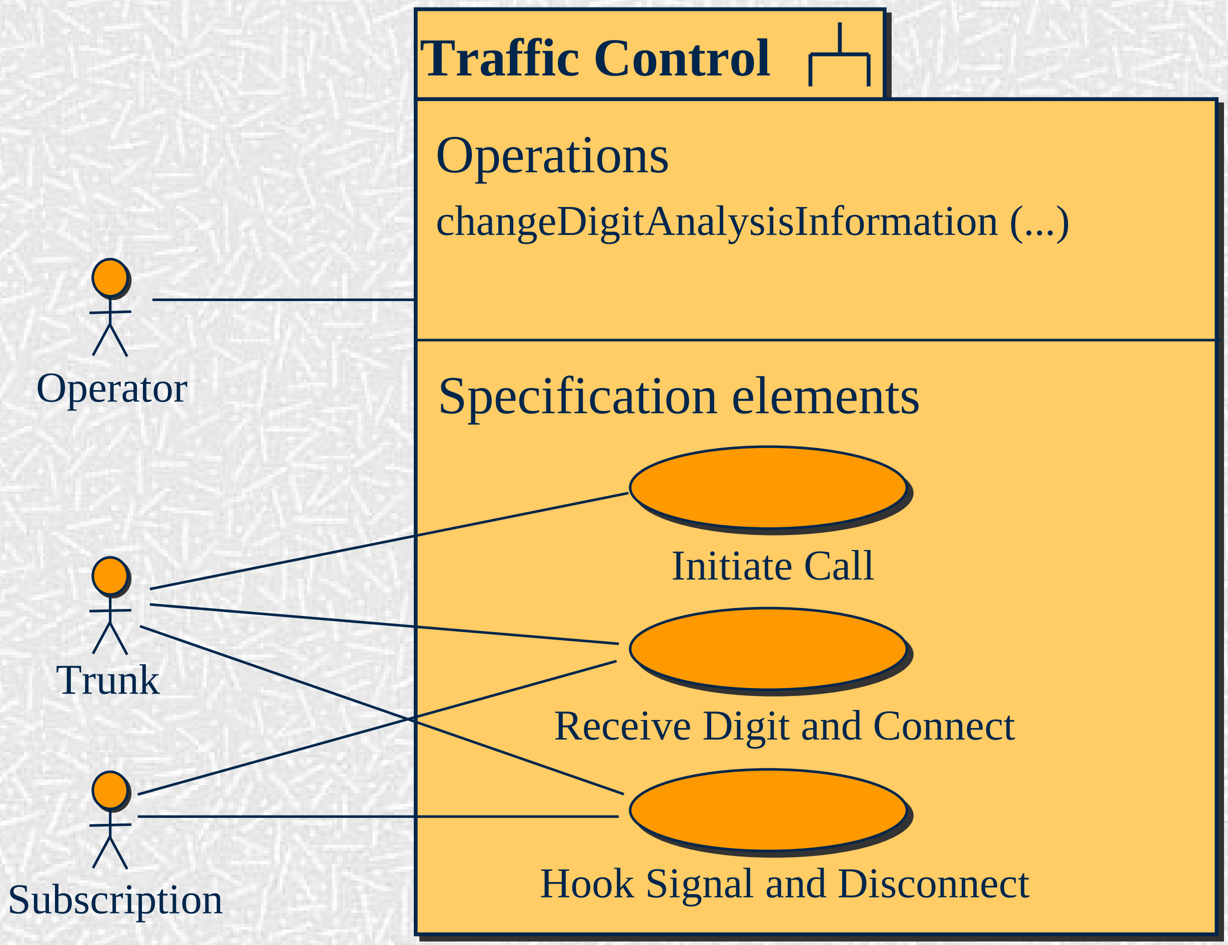
- When usage of the subsystem is perceived as manipulation of objects
- When the requirements are guided by a particular standard

Operation Approach

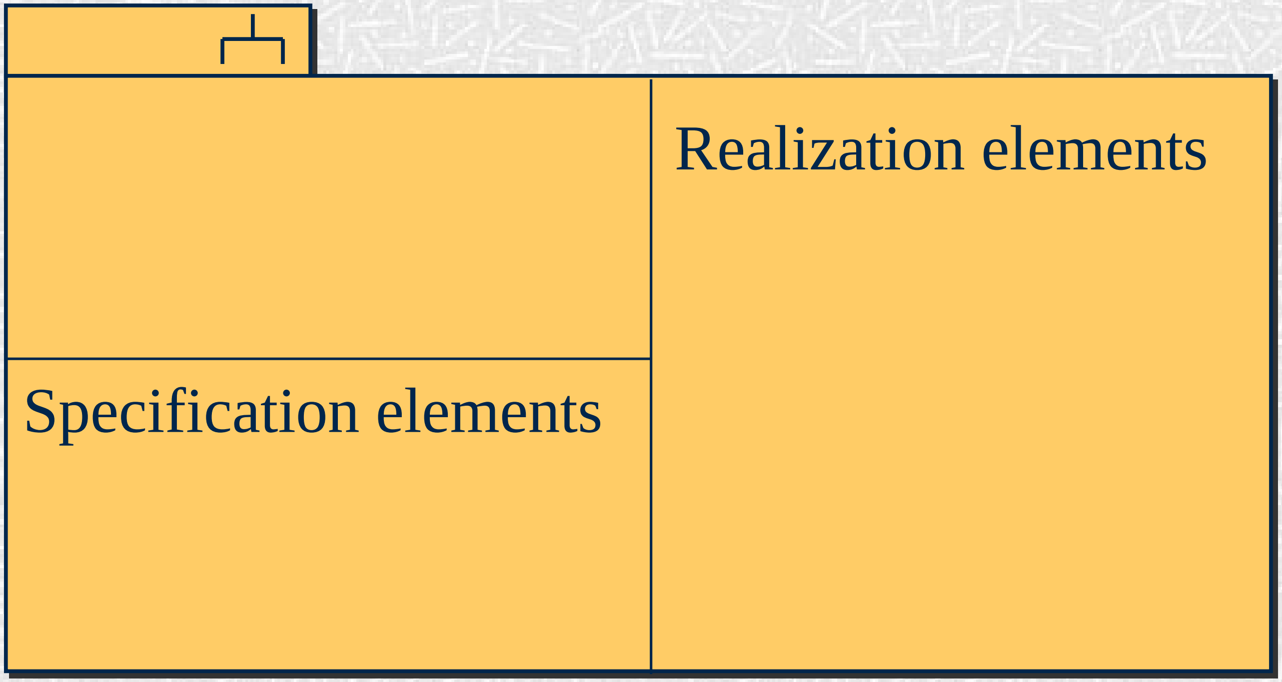


- For subsystems providing simple, “atomic” services
- When the operations are invoked independently

Mixing Techniques

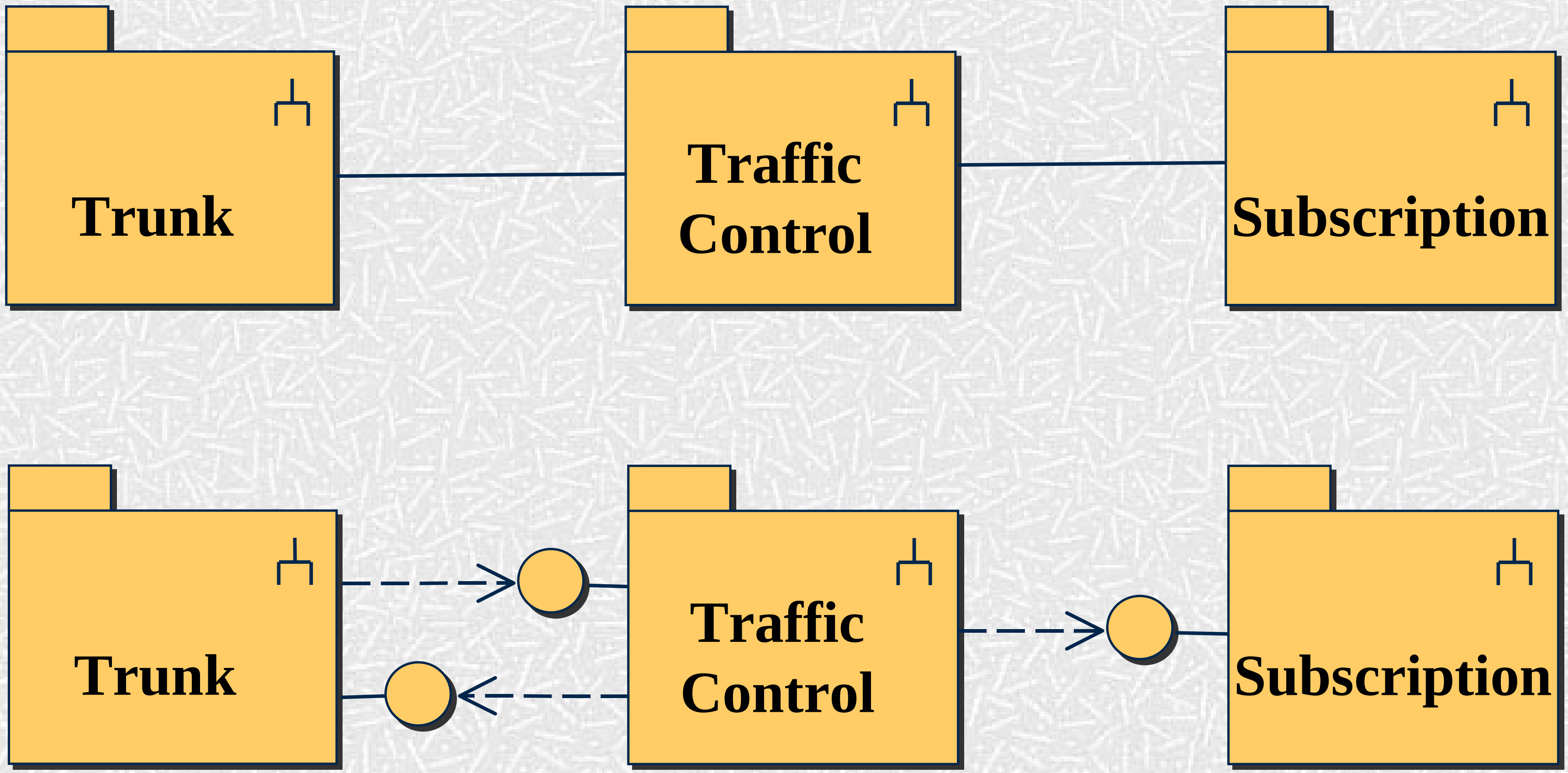


Complete Subsystem Notation

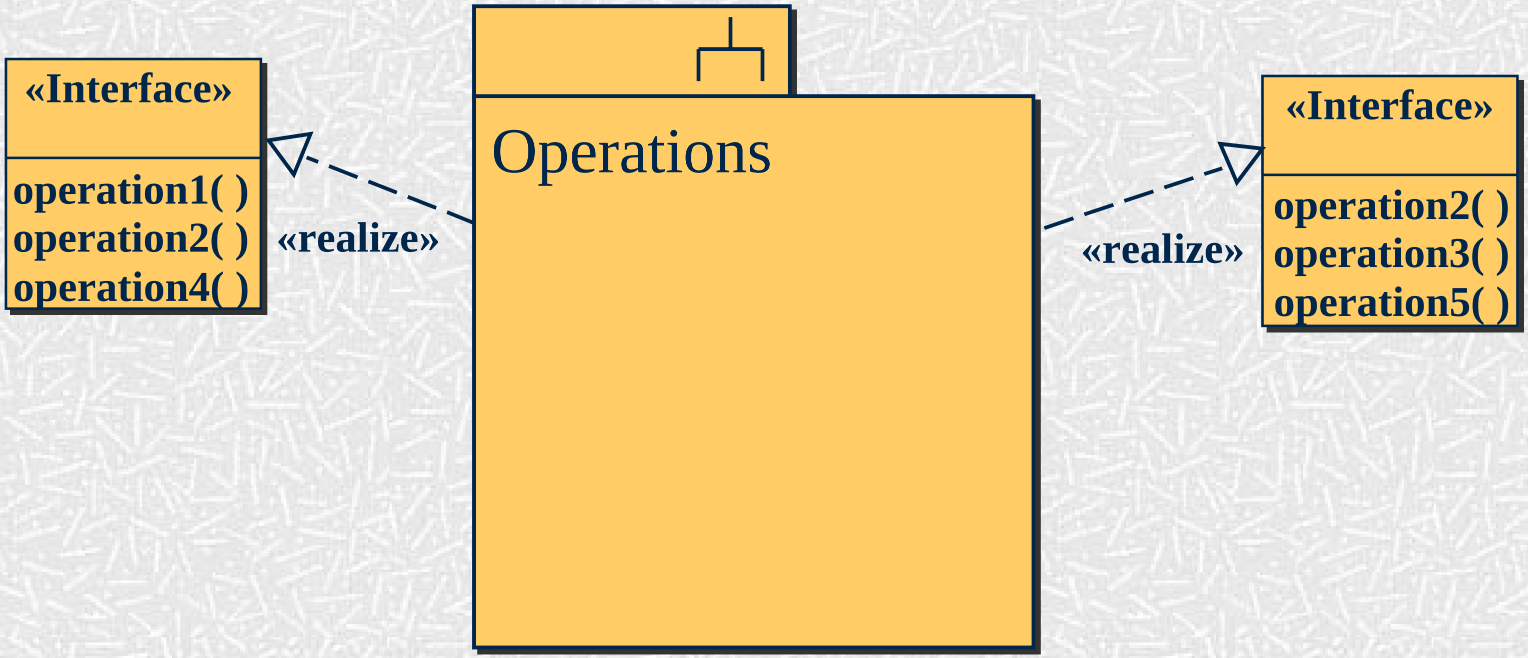


- The complete subsystem symbol has three pre-defined compartments
- Each of the compartments may optionally be omitted from the diagram

Subsystem Interfaces

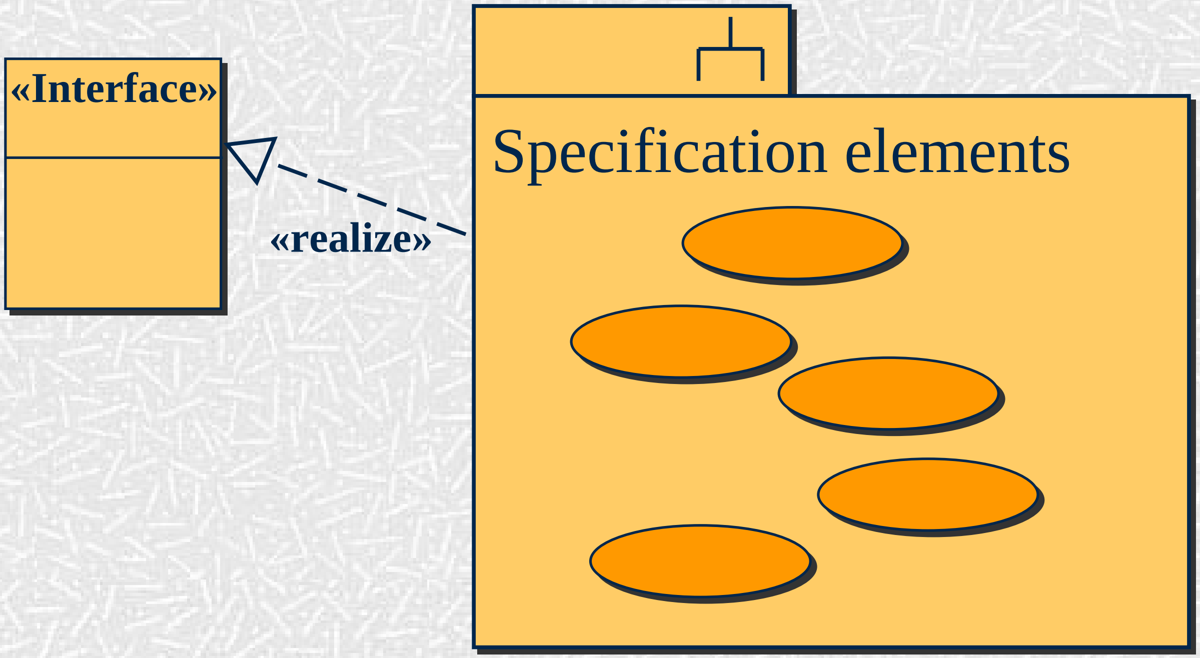


Operations and Interfaces



The subsystem must support all operations in the offered interfaces

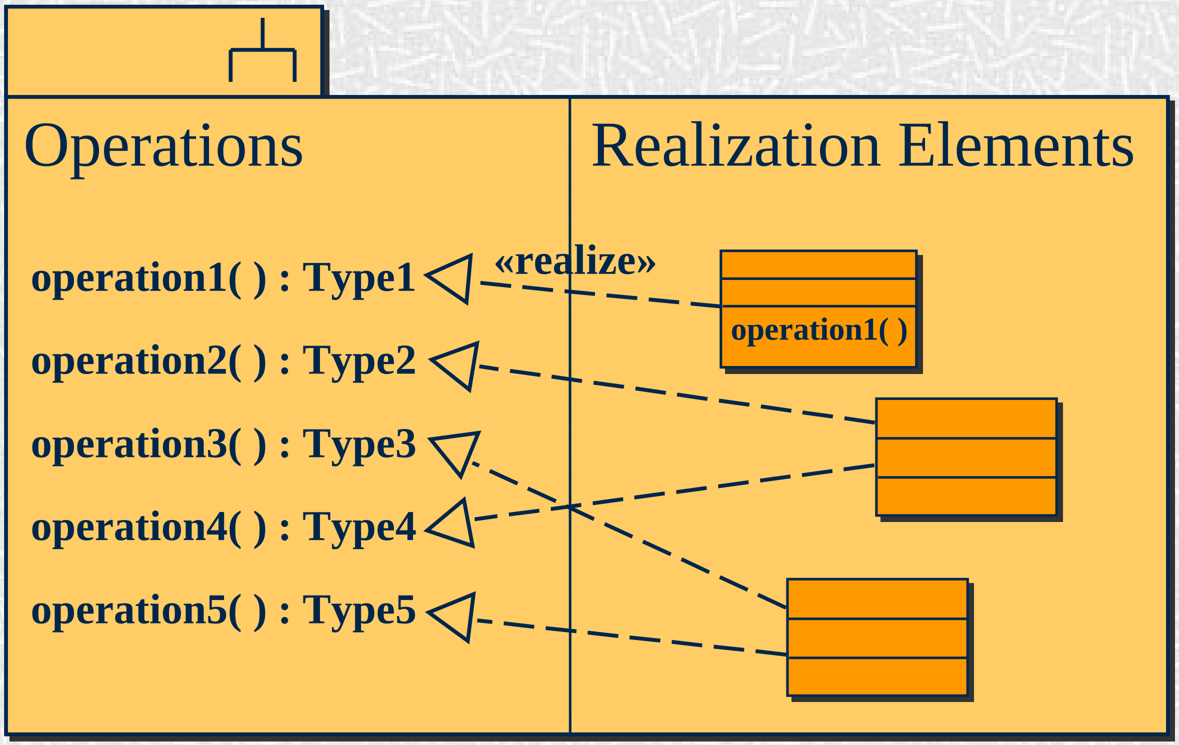
Subsystem Interfaces



Specification – Realization

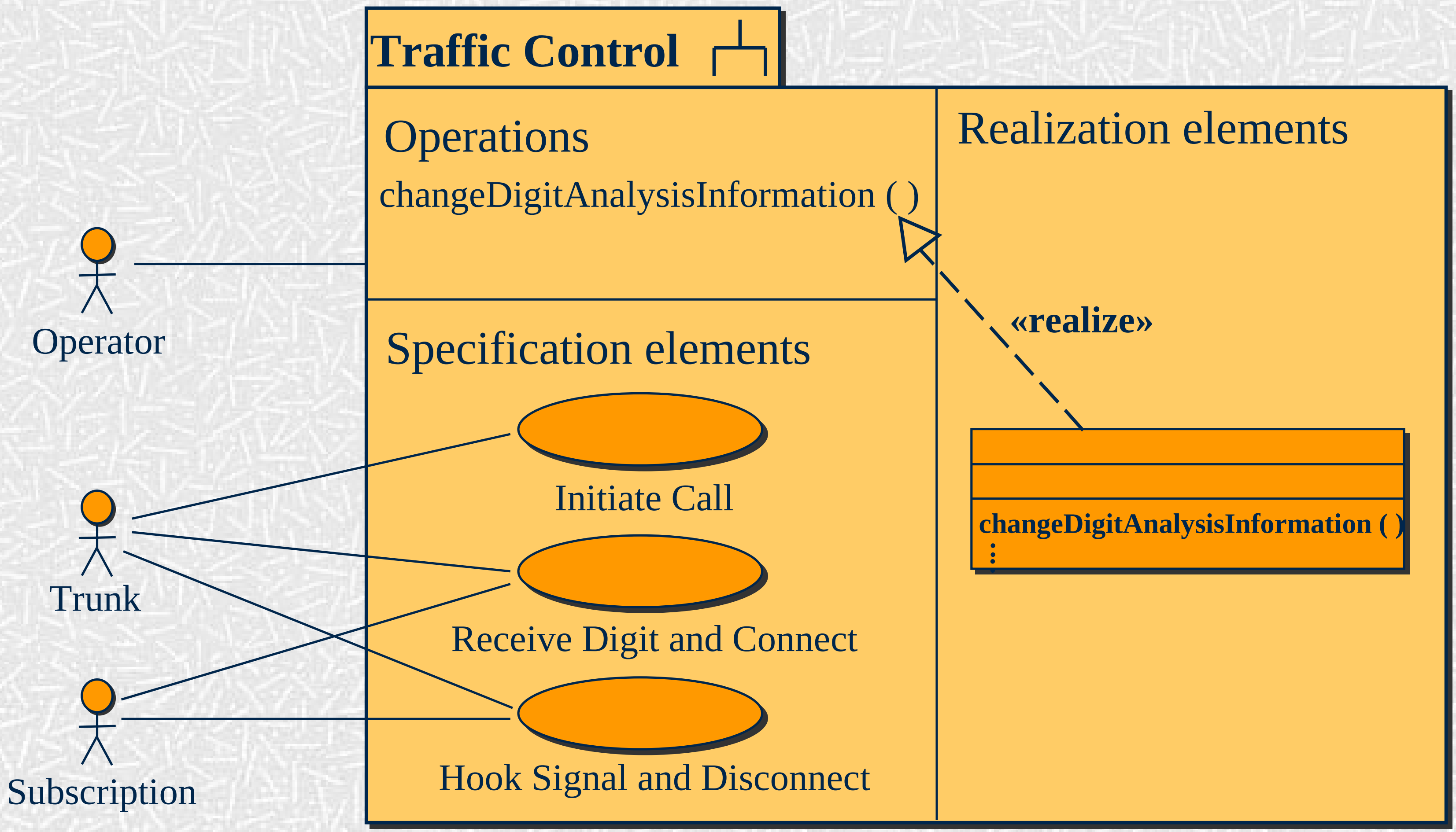
- The specification and the realization must be consistent
- The mapping between the specification and the realization can be expressed by:
  - «realize» relationships
  - collaborations

Realize Relationship



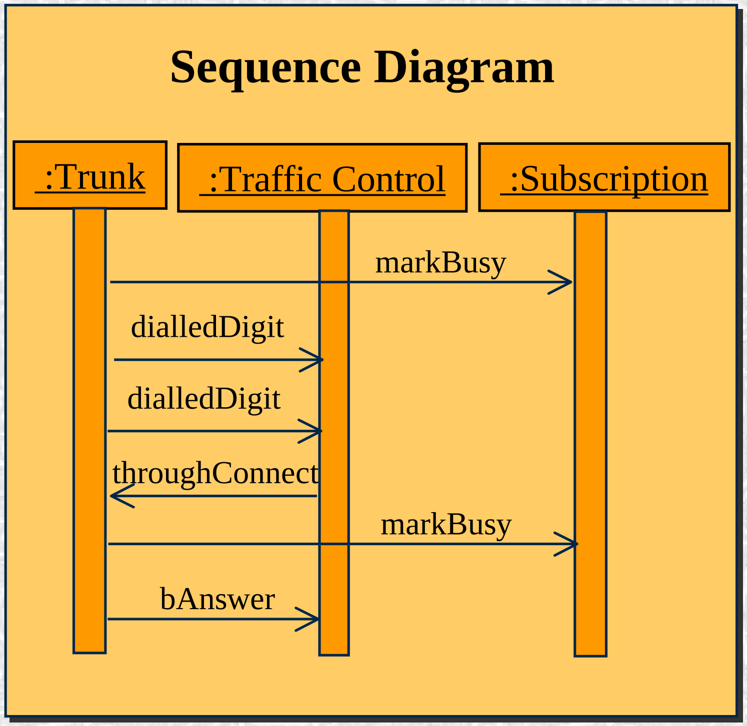
«realize» is particularly useful in simple mappings

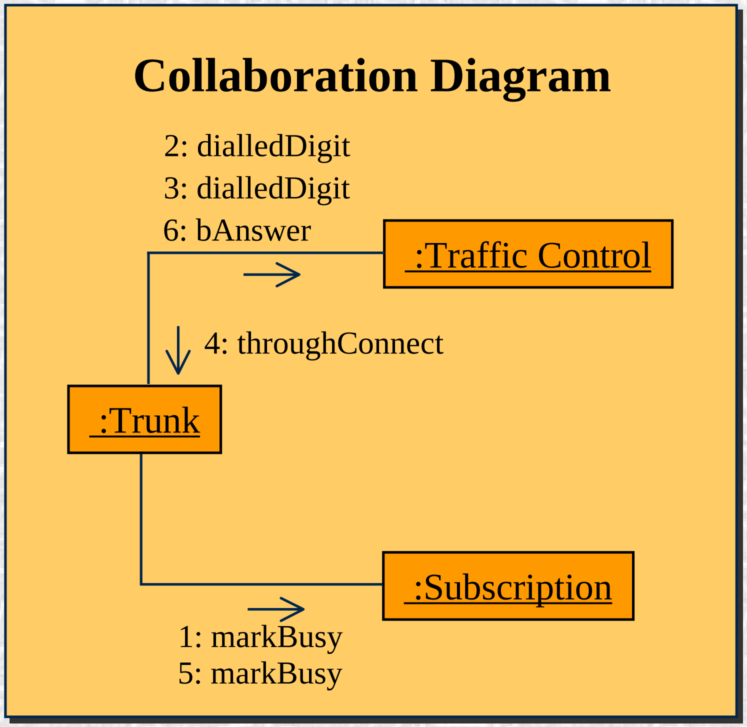
Realize – Example



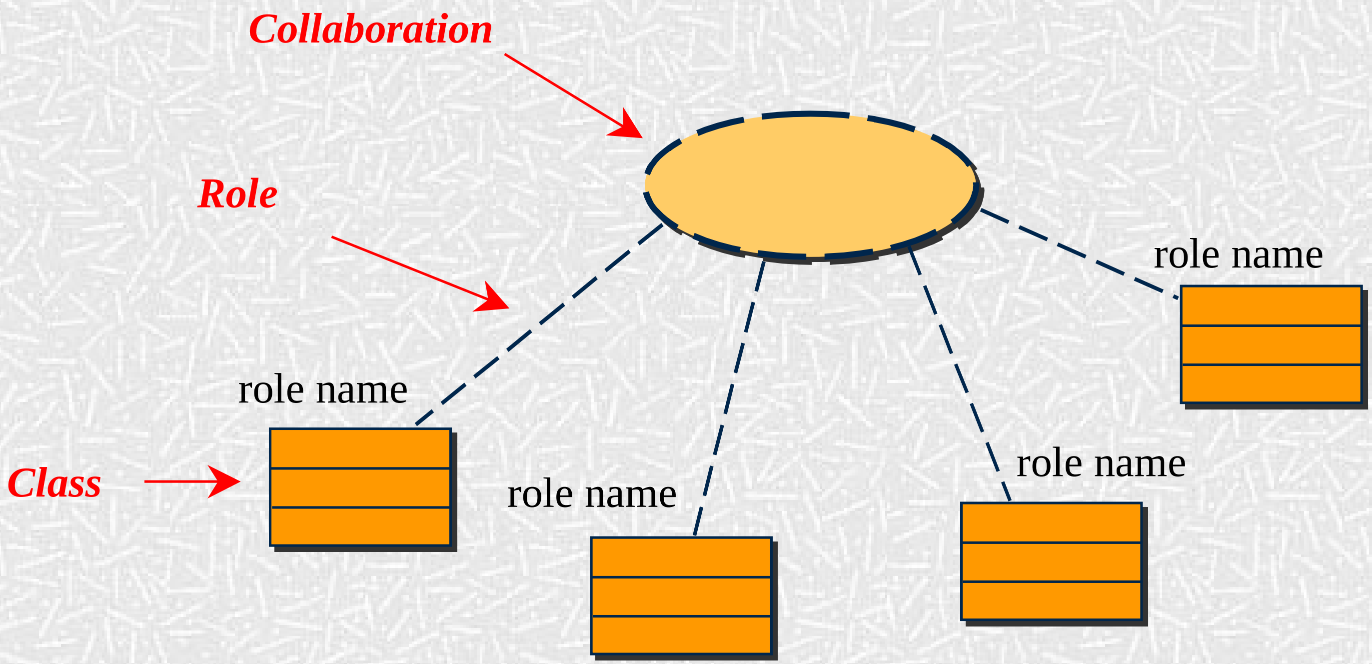
Collaboration

- A collaboration defines the roles to be played when a task is performed
- The roles are played by interacting instances



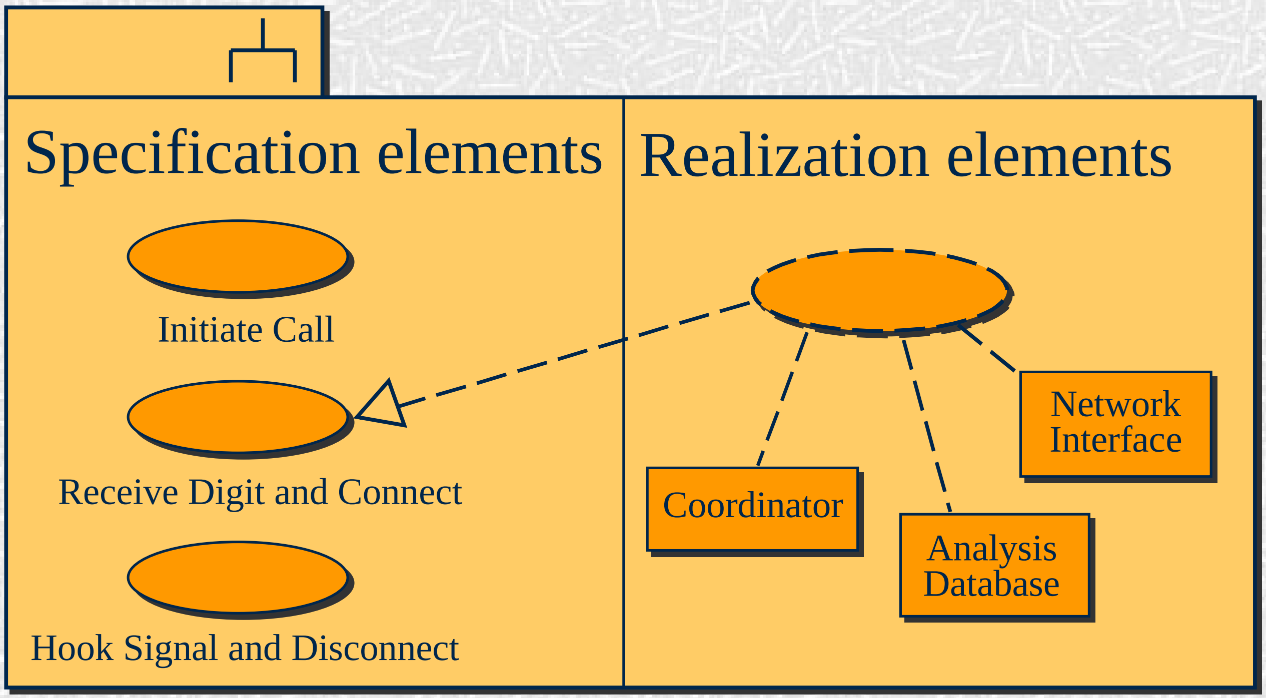


Collaboration – Notation



A collaboration and its participants

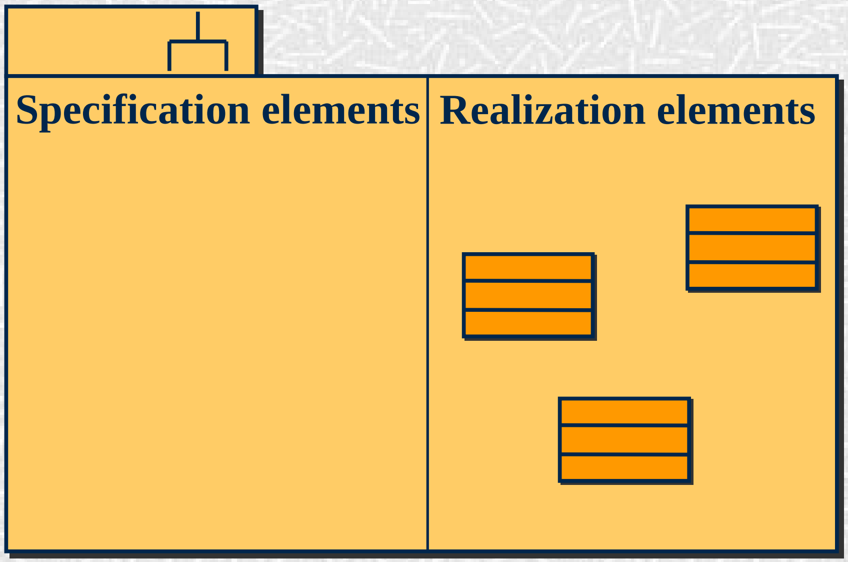
Collaboration – Example



Collaborations are useful in more complex situations

Diagram Tour

- Subsystems can be shown in static diagrams and interaction diagrams
- “Fork” notation alternative for showing contents:



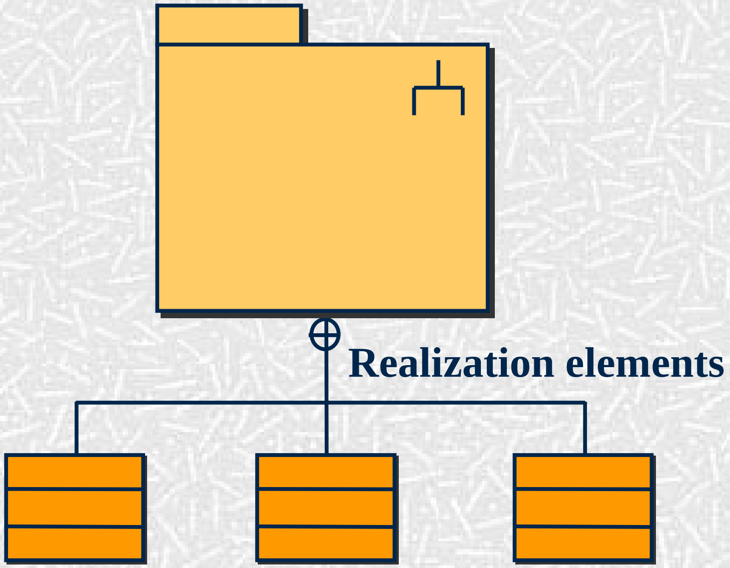
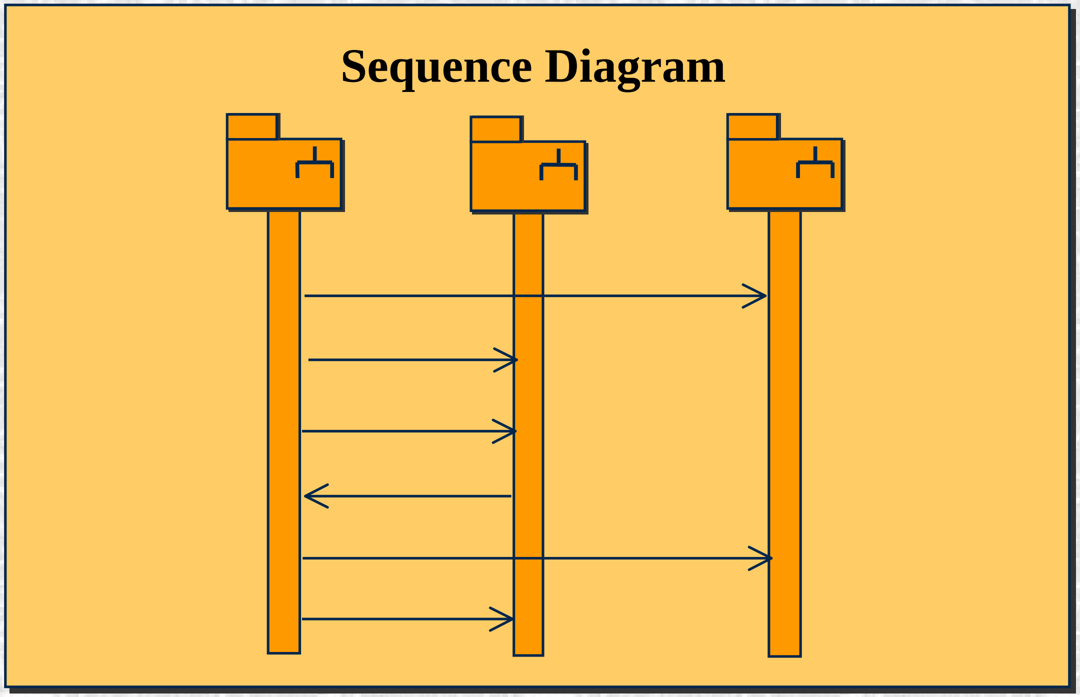


Diagram Tour – continued

- Subsystems can be shown in interaction diagrams
  - collaboration diagrams
  - sequence diagrams



When to Use Subsystems

- To express how a large system is decomposed into smaller parts
- To express how a set of modules are composed into a large system
- To trace requirements between the system and its parts

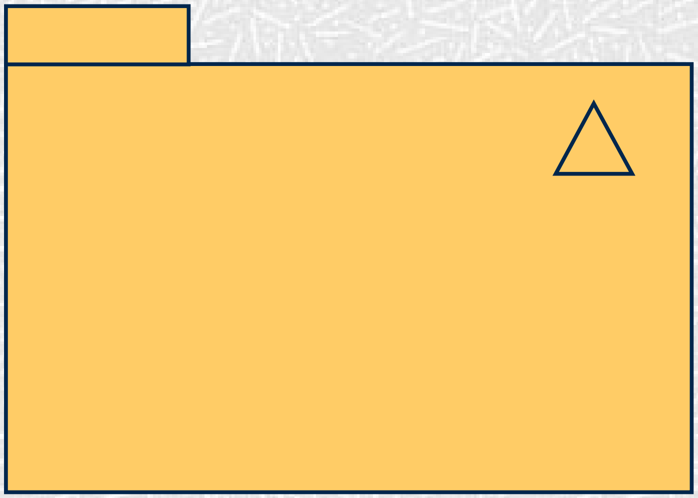
Modeling Tips – Subsystem

- Define a subsystem for each separate part of a large system
- Choose specification technique depending on factors like kind of system and kind of subsystem
- Realize each subsystem independently, using the specification as a requirements specification

Model

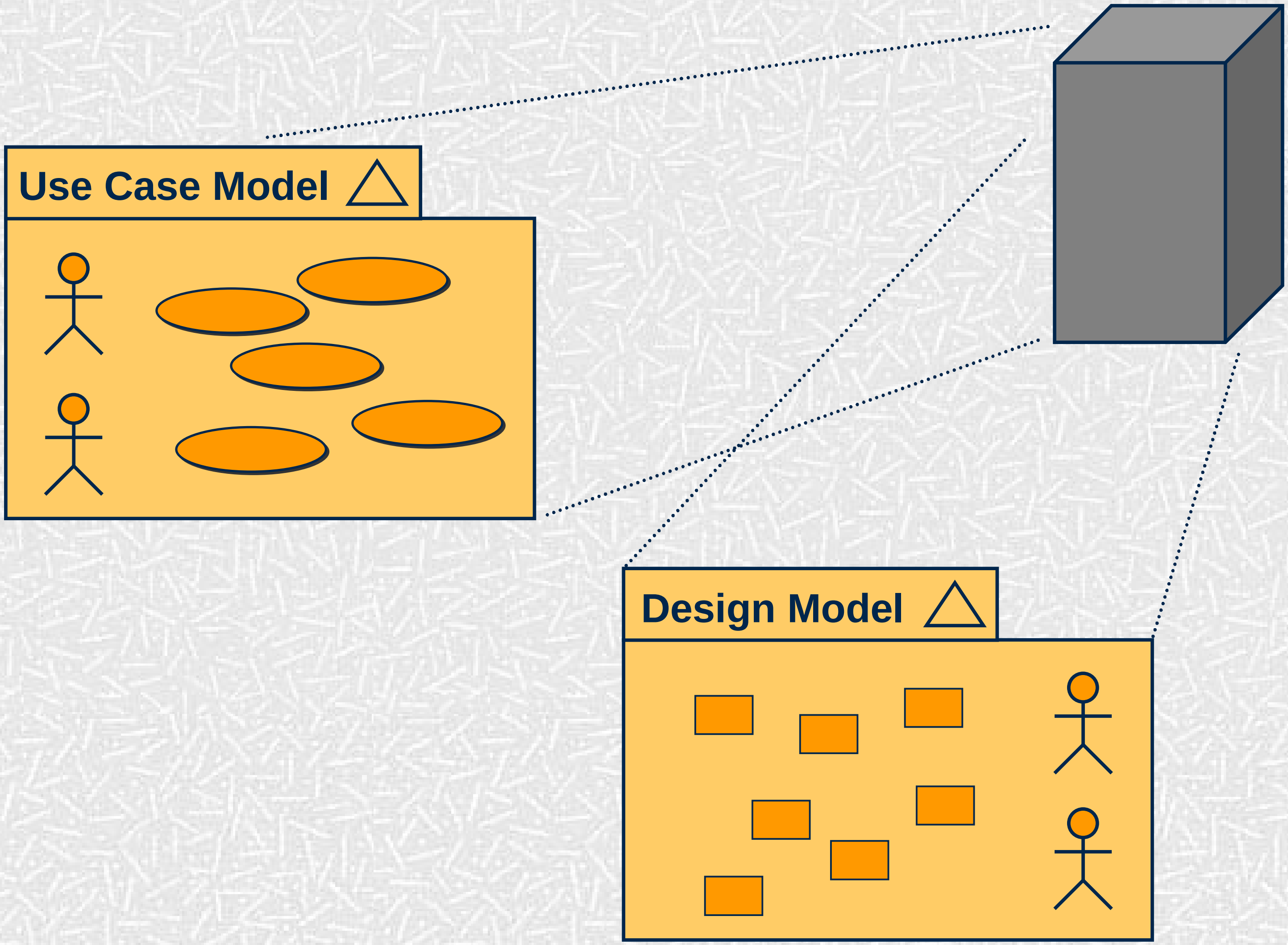
- What are Models
- Core Concepts
- Diagram Tour
- When to Use Models
- Modeling Tips

Model

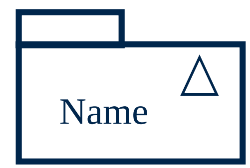


A model is an abstraction of a system, specifying the system from a certain viewpoint and at a certain level of abstraction

Model – Example



Core Concepts

Construct	Description	Syntax
Model	An abstraction of a system, as seen from a specific viewpoint and at a certain level of abstraction and detail.	
Trace	A dependency connecting model elements that represent the same concept within different models. Traces are usually non-directed.	

Trace

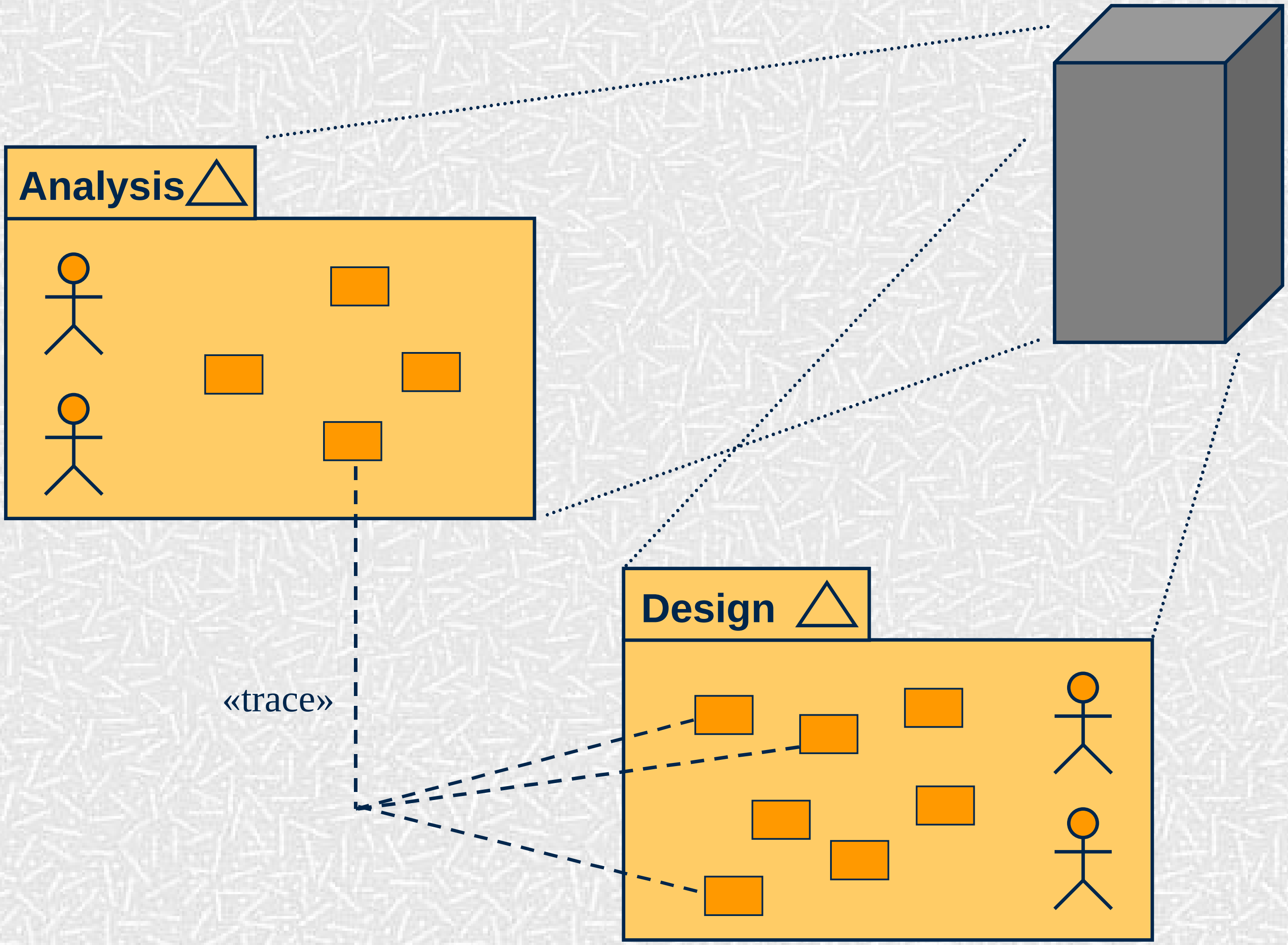
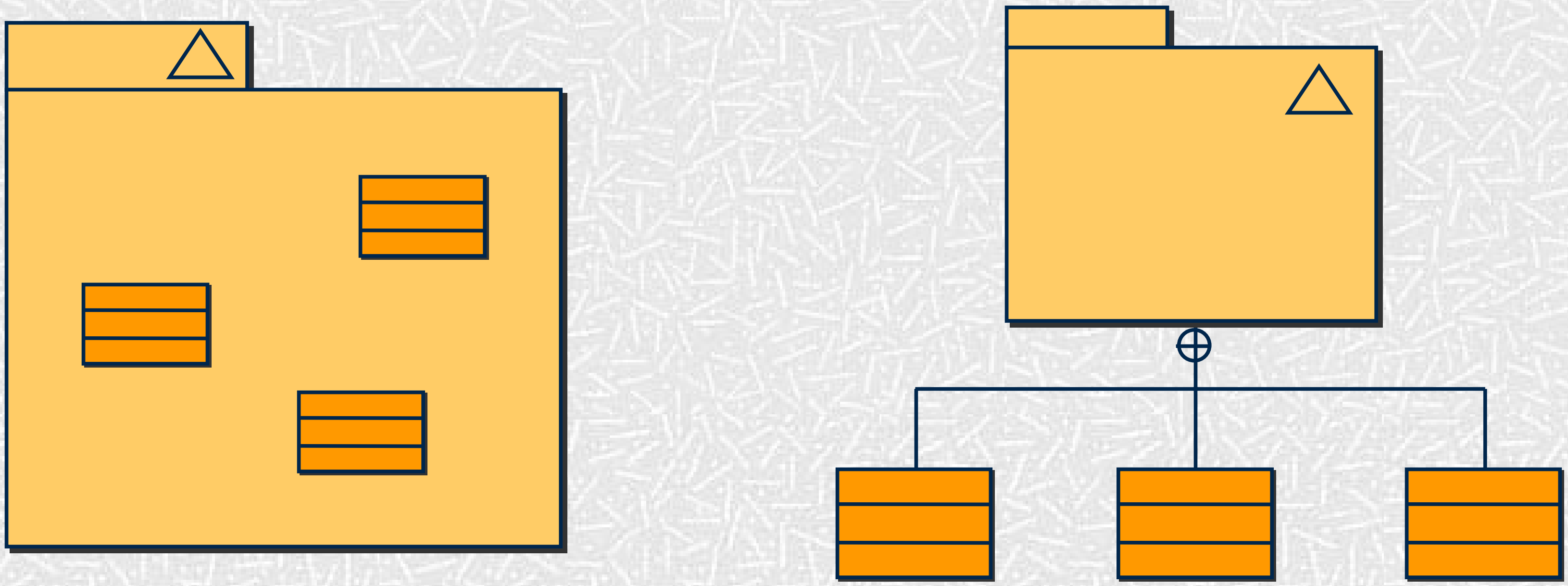
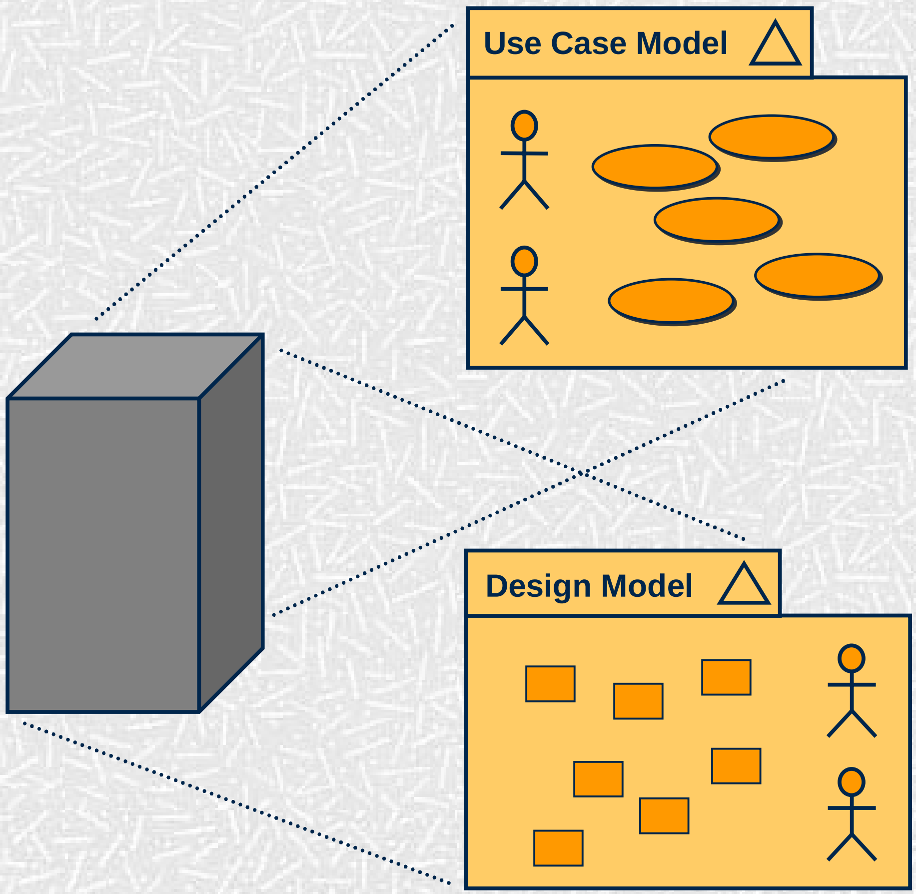


Diagram Tour

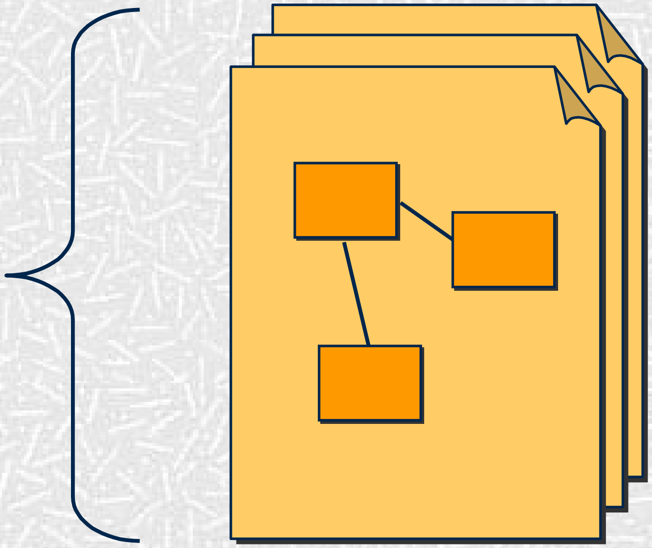
- Models as such are seldom shown in diagrams
- Two equivalent ways to show containment:



Model vs. Diagram



Diagrams make up the documentation of a model



When to Use Models

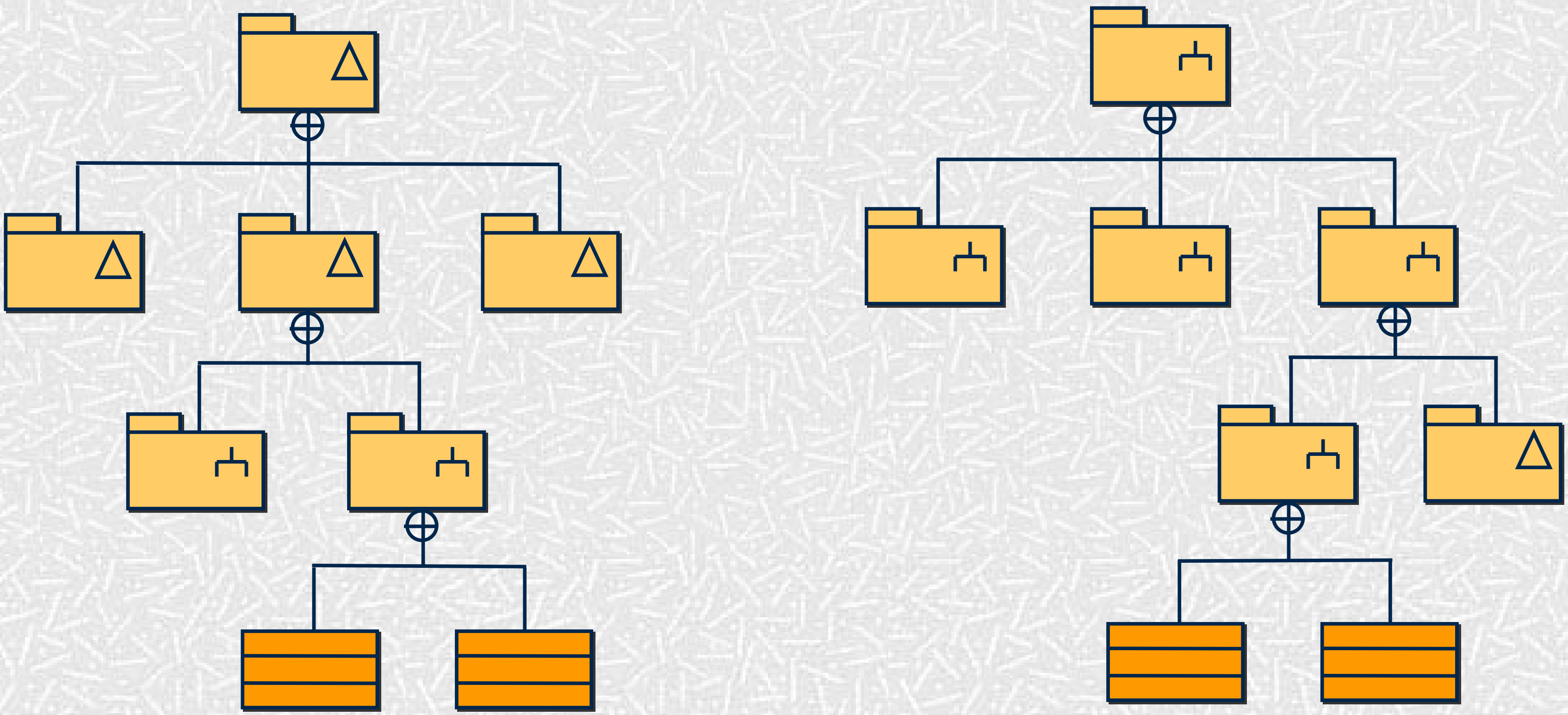
- To give different views of a system to different stakeholders
- To focus on a certain aspect of a system at a time
- To express the results of different stages in a software development process

Modeling Tips – Model

- Define the purpose for each model
- A model must give a complete picture of the system, within its viewpoint and level of abstraction
- Focus on the purpose of the model; omit irrelevant information

Models and Subsystems

Models and subsystems can be combined in a hierarchy:



Wrap Up Model Management

- Packages are used to organize a large set of model elements
  - Visibility
  - Import
  - Access
- Subsystems are used to structure a large system
  - Specification
  - Realization
- Models are used to show different aspects of a system
  - Trace

Advanced Modeling with UML

- Part 1: Model Management
- Part 2: Extension Mechanisms and Profiles

Bran Selic, Rational Software
bran@objectime.com

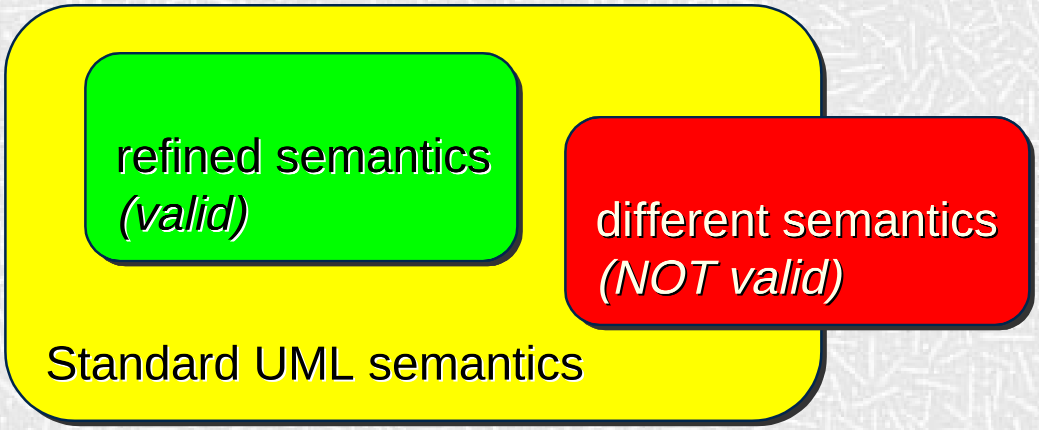
- Part 3: Object Constraint Language (OCL)

Semantic Variations in UML

- Semantic aspects that are:
  - undefined (e.g., scheduling discipline), or
  - ambiguous (multiple interpretations/possibilities)
- Why?
  - Different domains require different specializations
  - Extend the applicability and utility of UML to a very broad spectrum of domains
    - …while avoiding the “PL/I syndrome”

Extensibility Mechanisms

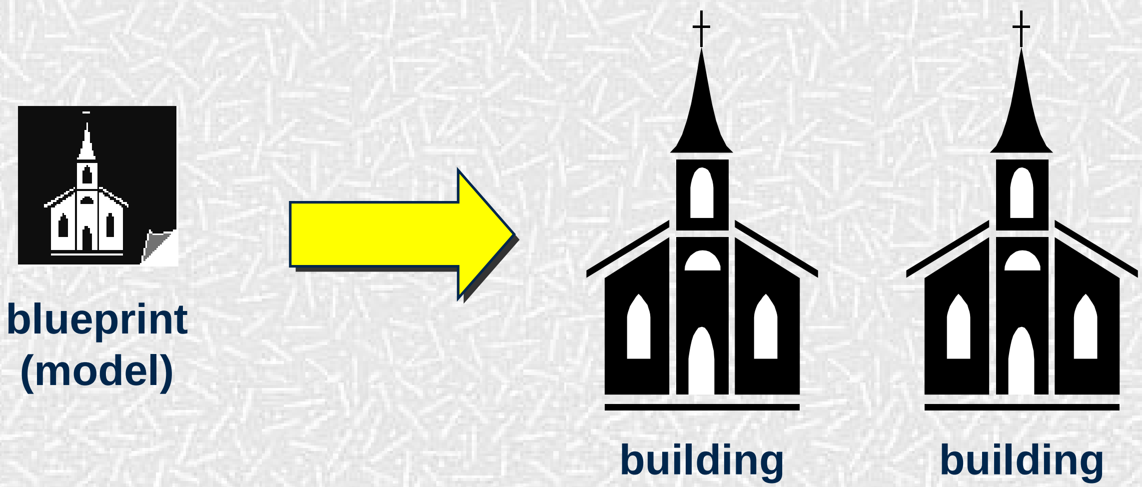
- Used for **refining** the general UML semantics
  - *must be consistent with general UML semantics!*



- Purpose:
  - To obtain specialized domain-specific or even application-specific variations of general-purpose modeling concepts

Models

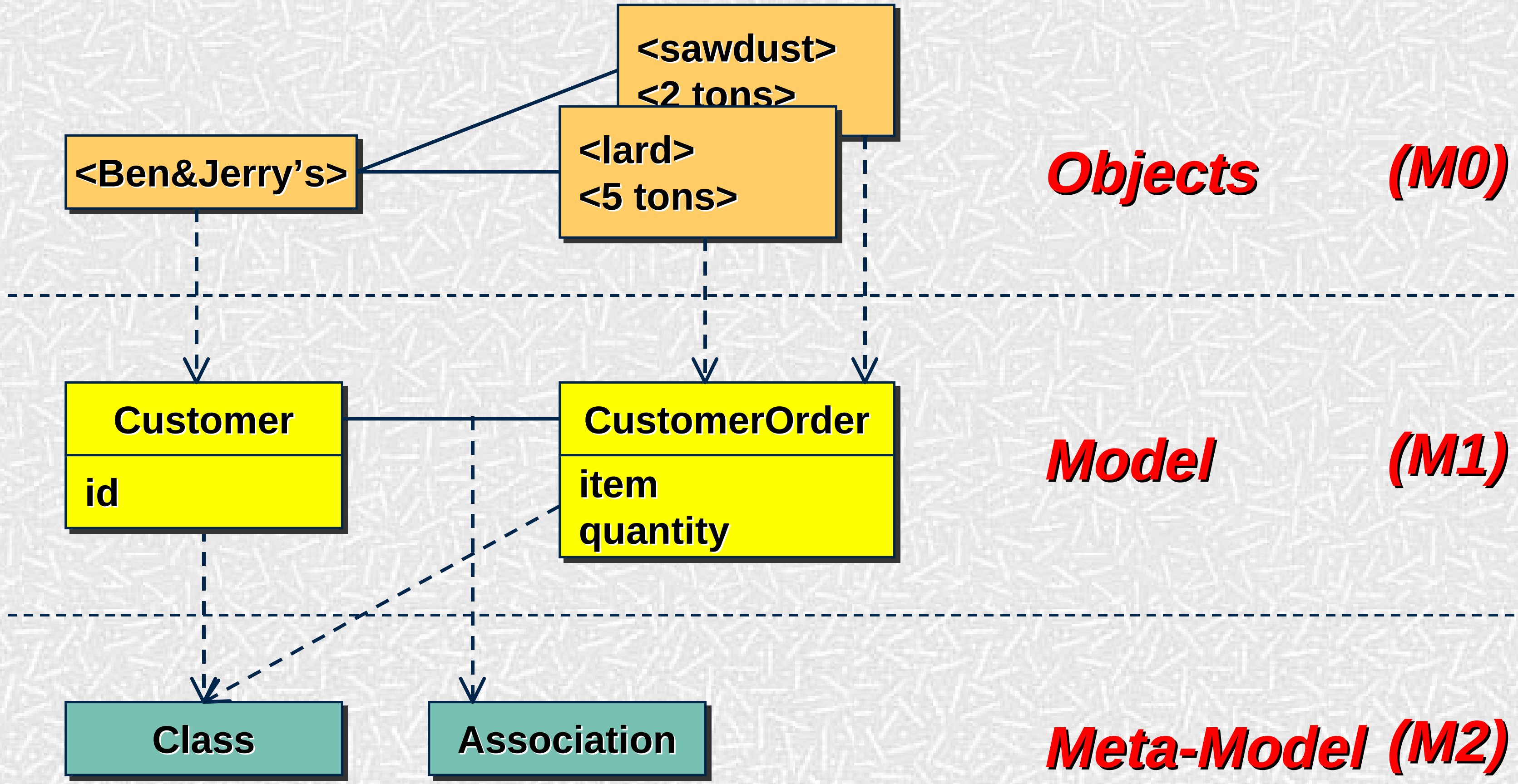
- A **model** is a description of something
  - *“a pattern for something to be made”* (Merriam-Webster)
  - model ≠ thing that is modeled



- model ≠ thing that is modeled

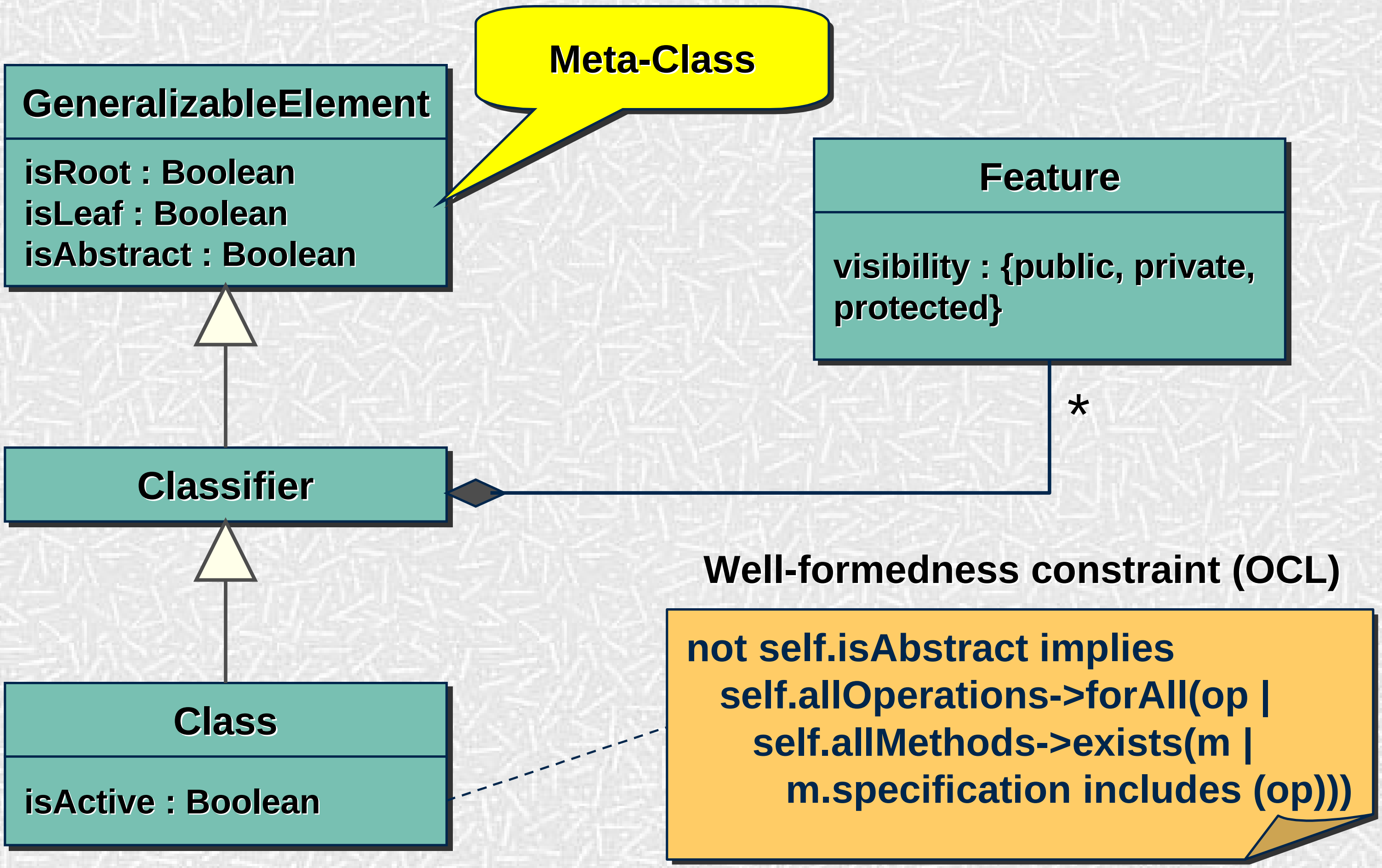
Meta-Models

- Models of models (modeling tools)



The UML Meta-Model

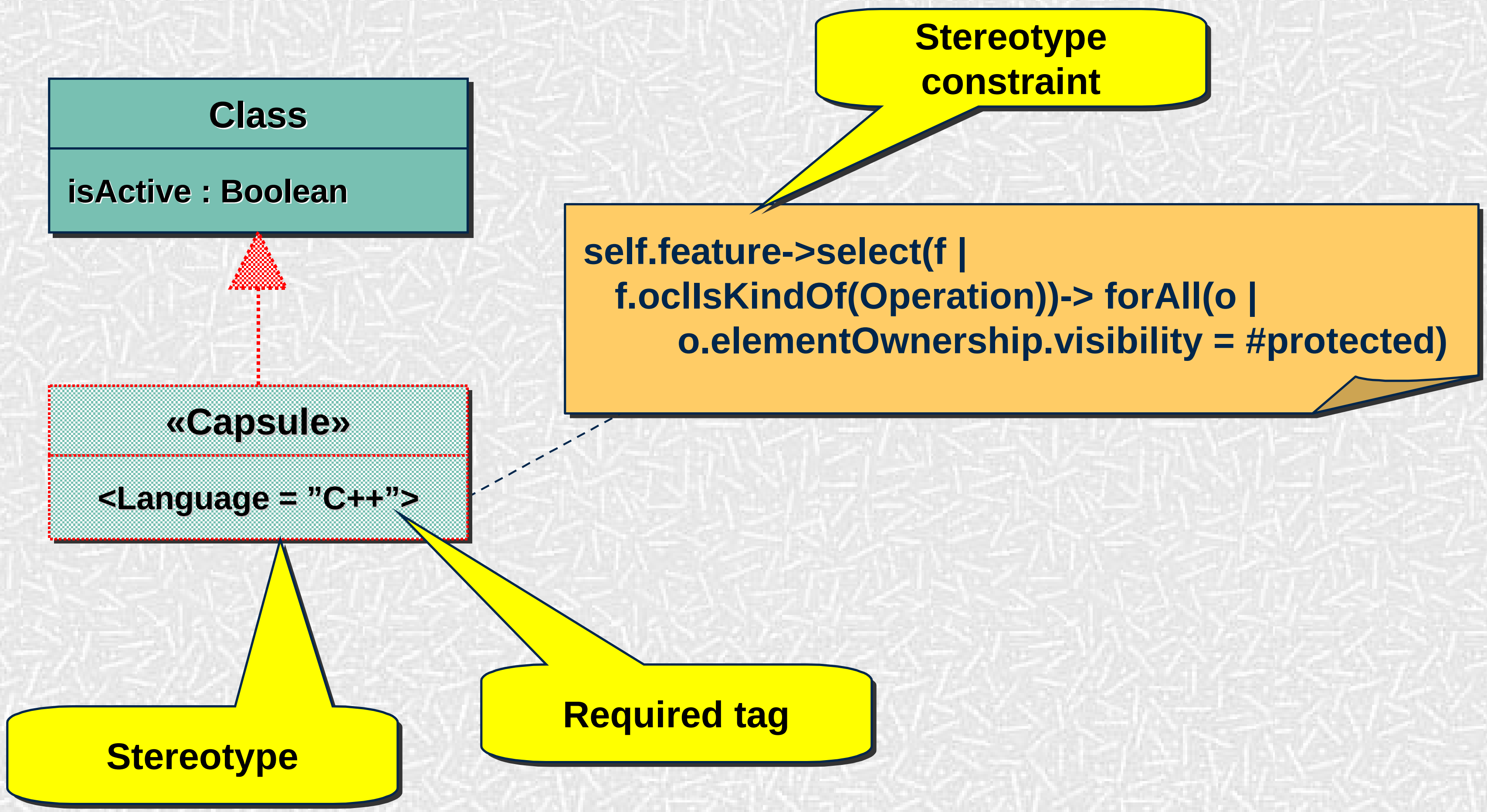
- Expressed using a very small subset of UML



The Three Basic Mechanisms

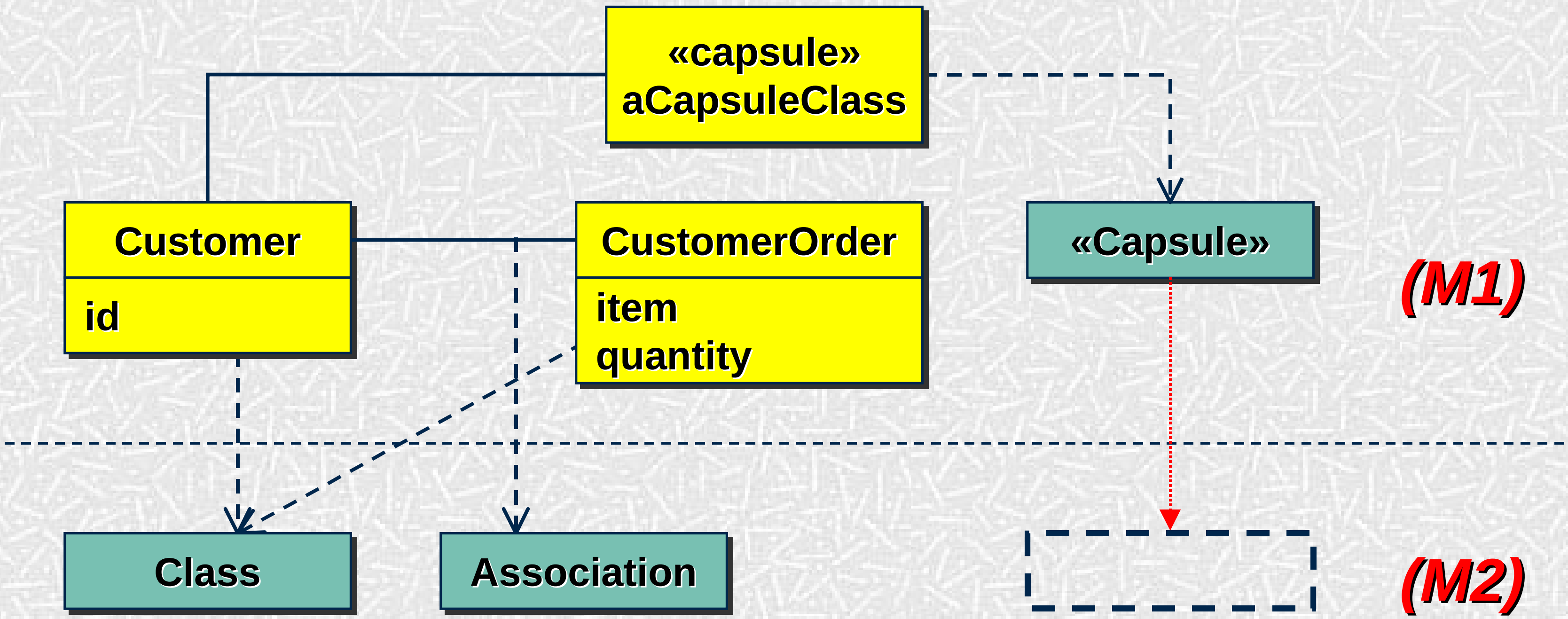
- Stereotypes
  - used to refine meta-classes (or other stereotypes) by defining supplemental semantics
- Constraints
  - predicates (e.g., OCL expressions) that reduce semantic variation
  - can be attached to any meta-class or stereotype
- Tagged Values
  - individual modifiers with user-defined semantics
  - can be attached to any meta-class or stereotype

Example: A Special Type of Class



Extensibility Method

- Refinements are specified at the Model (M1) level but apply to the Meta-Model level (M2)
  - does not require “meta-modeling” CASE tools
  - can be exchanged with models

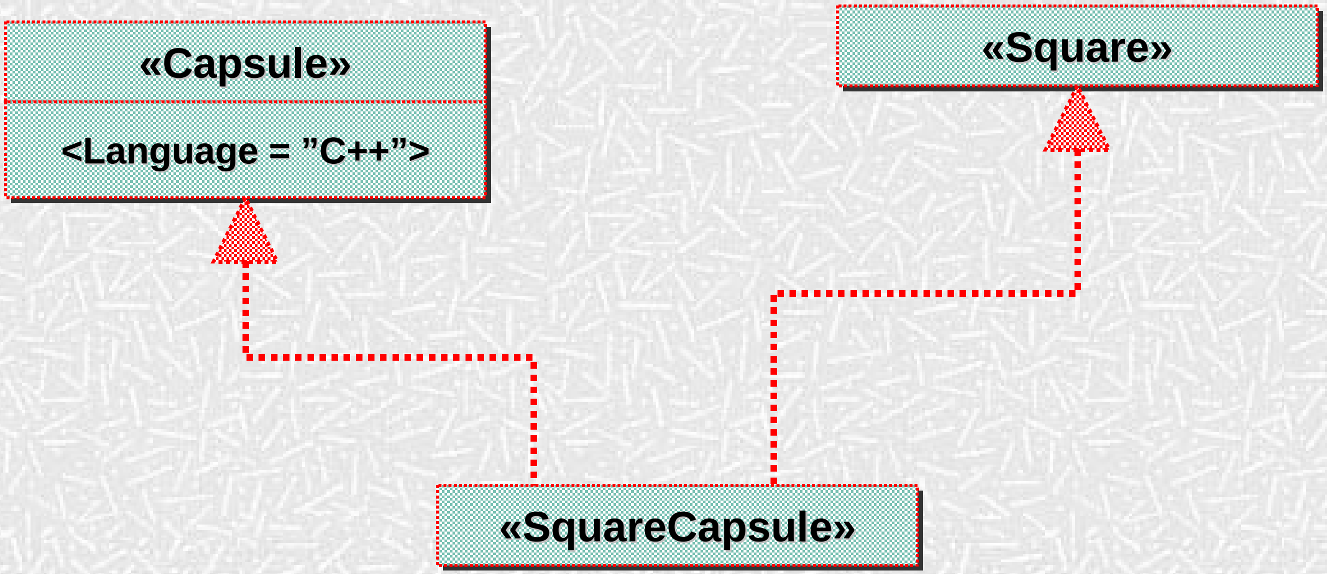


Stereotypes

- Used to define derivative modeling concepts based on existing generic modeling concepts
- Defined by:
  - base (meta-)class = UML meta-class or stereotype
  - constraints
  - required tags (0..*)
    - often used for modeling pseudo-attributes
  - icon
- A model element can have at most one stereotype

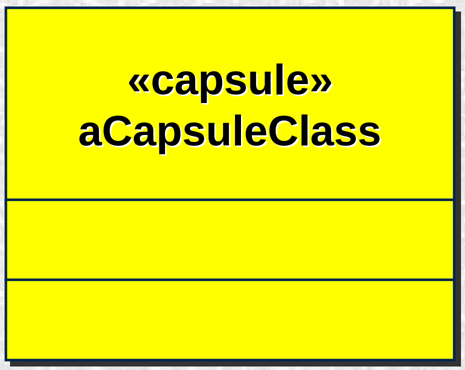
Heuristic: Combining Stereotypes

- Through multiple inheritance:

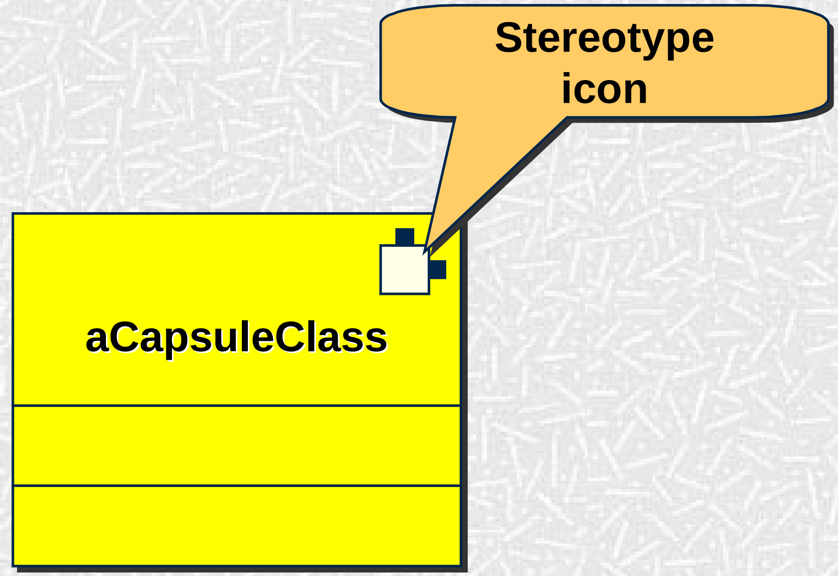


Stereotype Notation

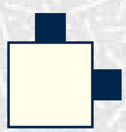
- Several choices



(a) with guillemets



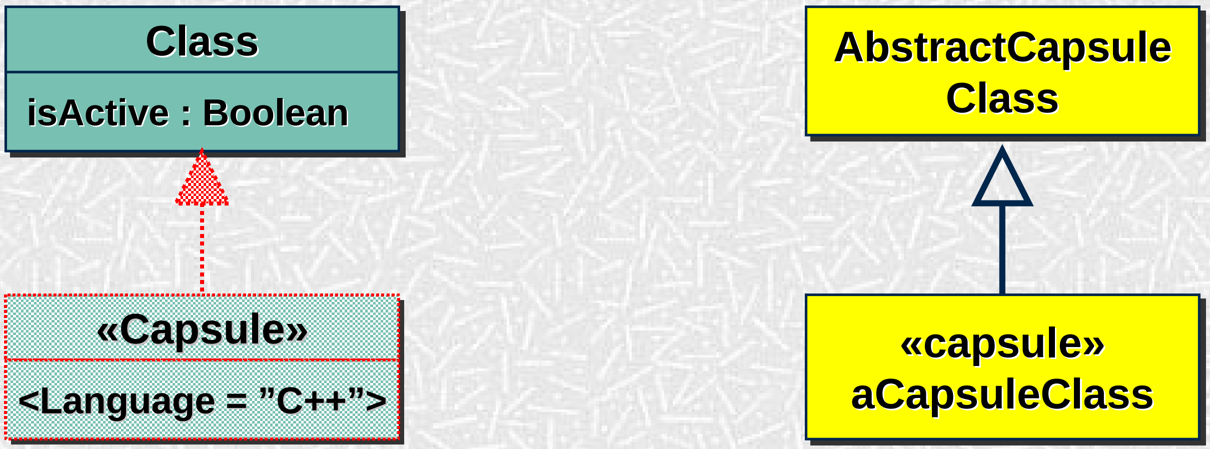
(b) with icon



(c) iconified form

Heuristic: When to Stereotype?

- Abstract class or stereotype?



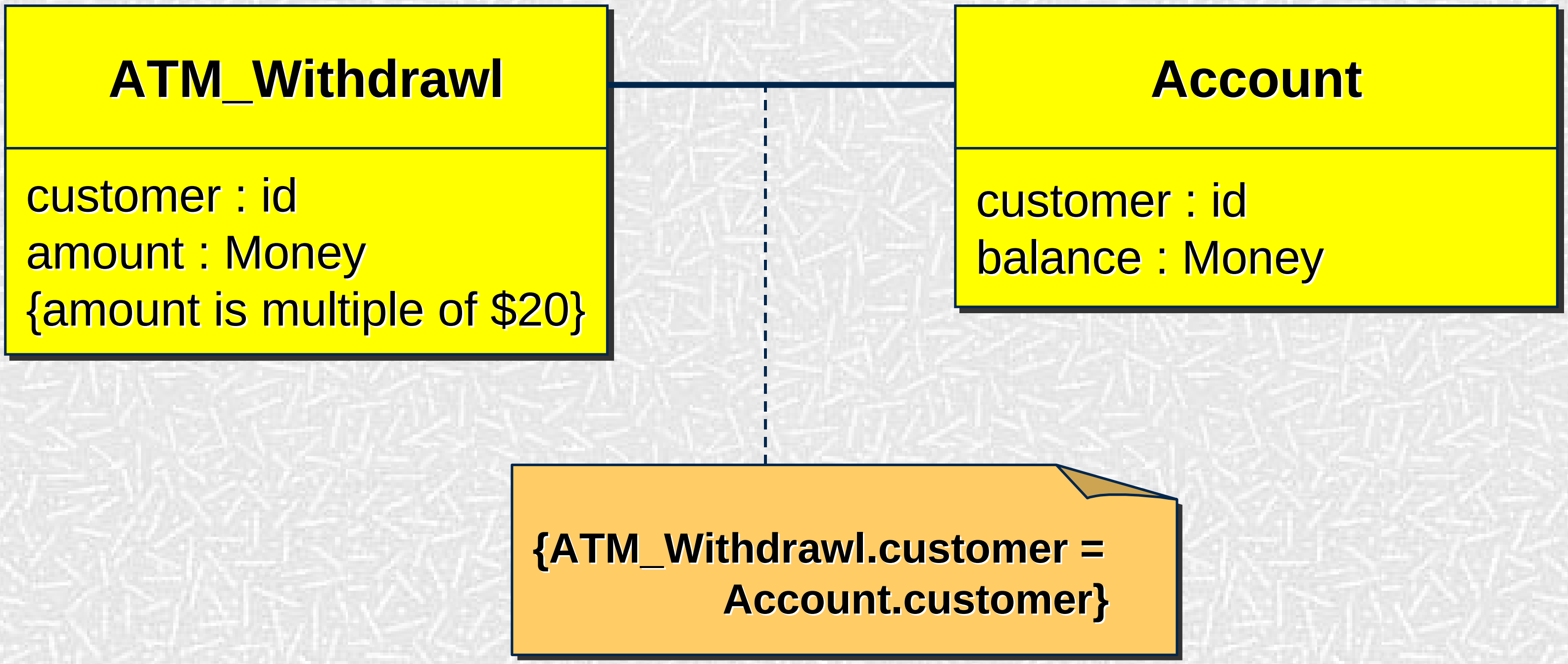
- Stereotypes typically used where one or more tools need to support (validate, enforce) the supplementary semantics
  - basis for further standardization

Tagged Values and Constraints

- Tagged values:
  - consist of a **tag** and **value** pair
  - often used to model stereotype attributes
  - arbitrary domain-specific semantics
    - instructions to a code generator (“debug_flag = true”)
    - project management data (“status = unit_tested”)
    - etc.
- Constraints
  - formal or informal expressions
  - must not contradict inherited base semantics

Constraint Notation

- Enclosed in braces “{…}”
- Can appear in various places in a model



UML Profiles

- A package of related specializations of general UML concepts that capture domain-specific variations and usage patterns
  - *A domain-specific interpretation of UML*
- Profiles currently being defined by the OMG:
  - EDOC
  - Real-Time
  - CORBA
  - ...

Advanced Modeling with UML

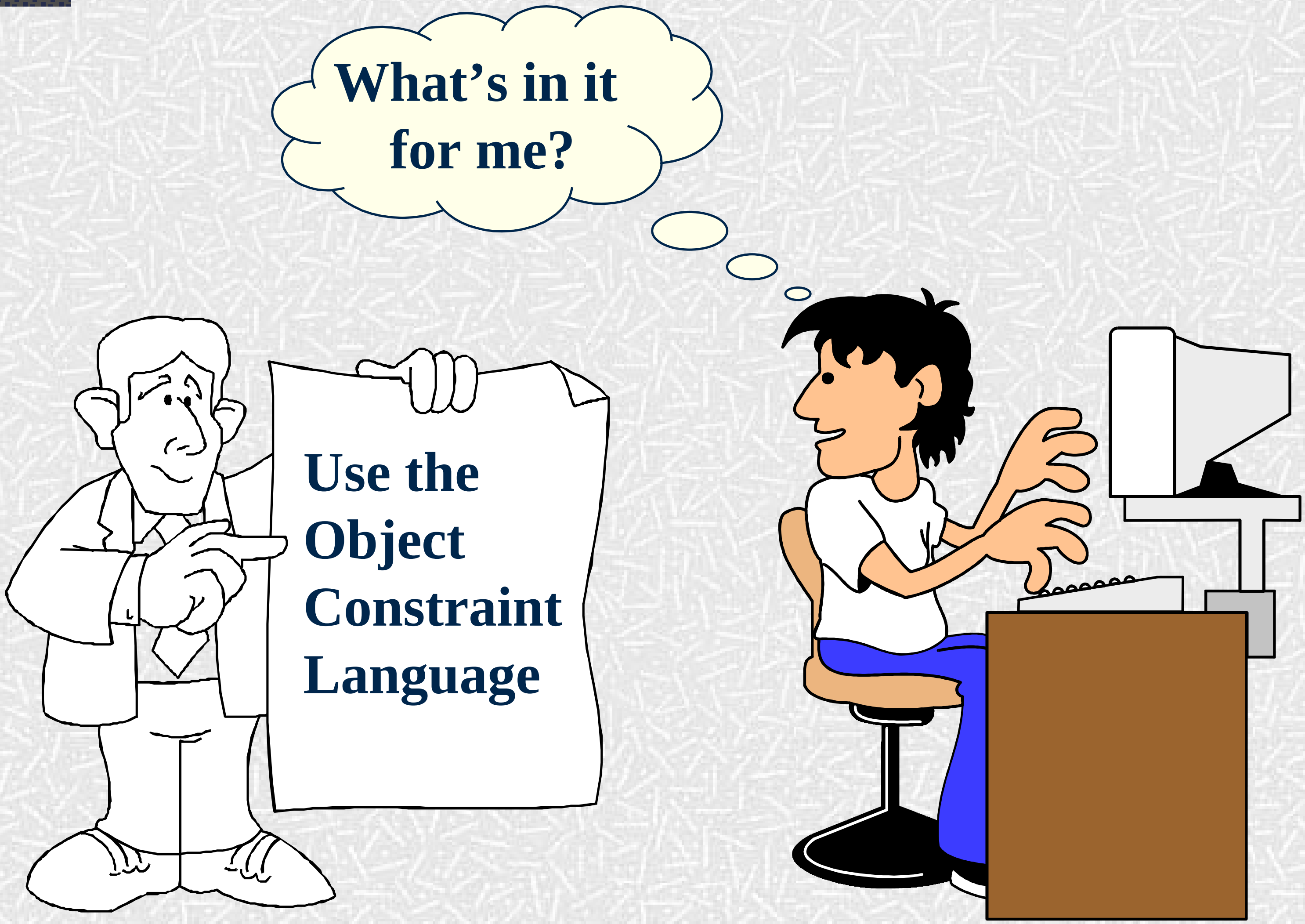
- Part 1: Model Management
- Part 2: Extension Mechanisms and Profiles
- Part 3: Object Constraint Language (OCL)

Jos Warmer, Klasse Objecten
j.warmer@klasse.nl

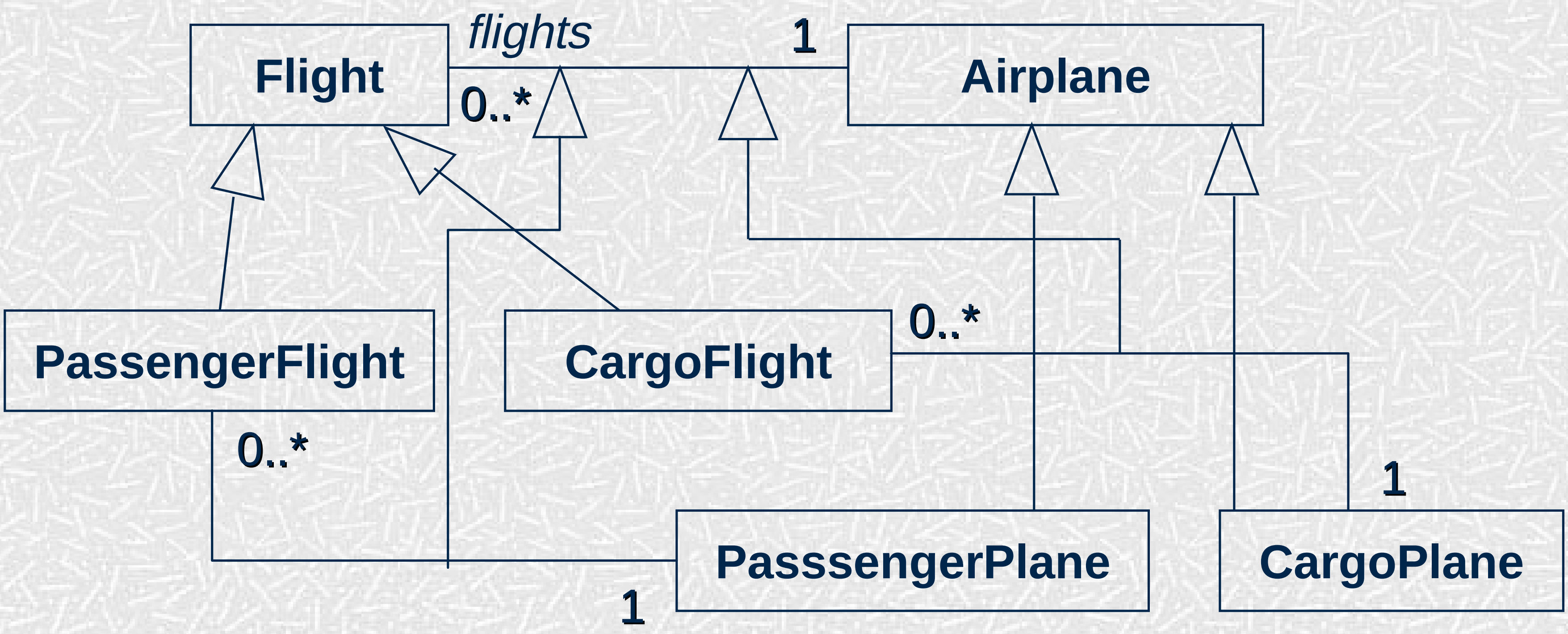
Overview

- What are constraints
- Core OCL Concepts
- Advanced OCL Concepts
- Wrap up

Why use OCL ?

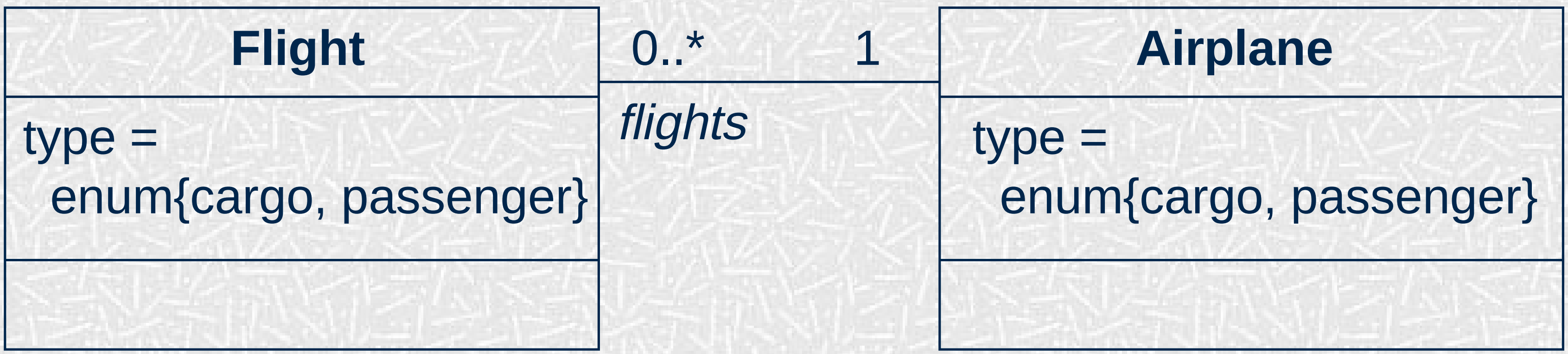


That’s why !!



Flight
flights
1
Airplane
0..*
PassengerFlight
CargoFlight
0..*
0..*
1
PasssengerPlane
CargoPlane
1

Diagram with invariants



context Flight
inv: type = #cargo implies airplane.type = #cargo
inv: type = #passenger implies airplane.type = #passenger

Definition of constraint

- "A constraint is a restriction on one or more values of (part of) an object-oriented model or system."

Different kinds of constraints

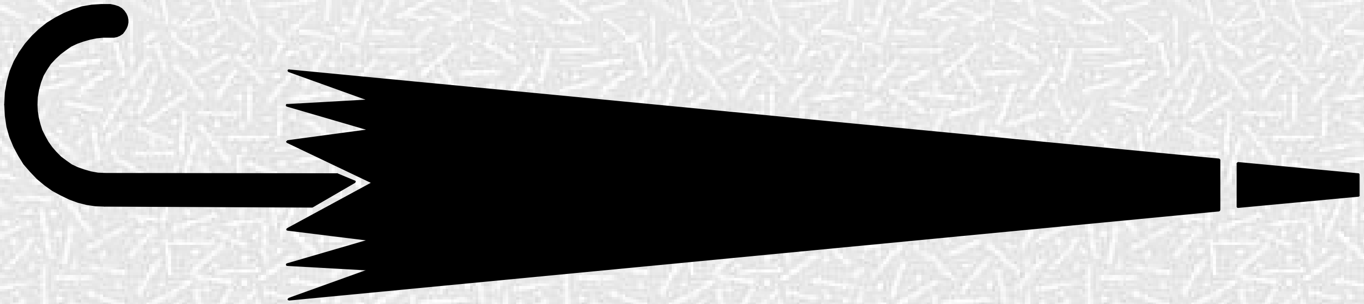
- Class invariant
  - a constraint that must always be met by all instances of the class
- Precondition of an operation
  - a constraint that must always be true BEFORE the execution of the operation
- Postcondition of an operation
  - a constraint that must always be true AFTER the execution of the operation

Constraint stereotypes

- UML defines three standard stereotypes for constraints:
  - invariant
  - precondition
  - postcondition

What is OCL?

- OCL is
  - a textual language to describe constraints
  - the constraint language of the UML
- Formal but easy to use
  - unambiguous
  - no side effects



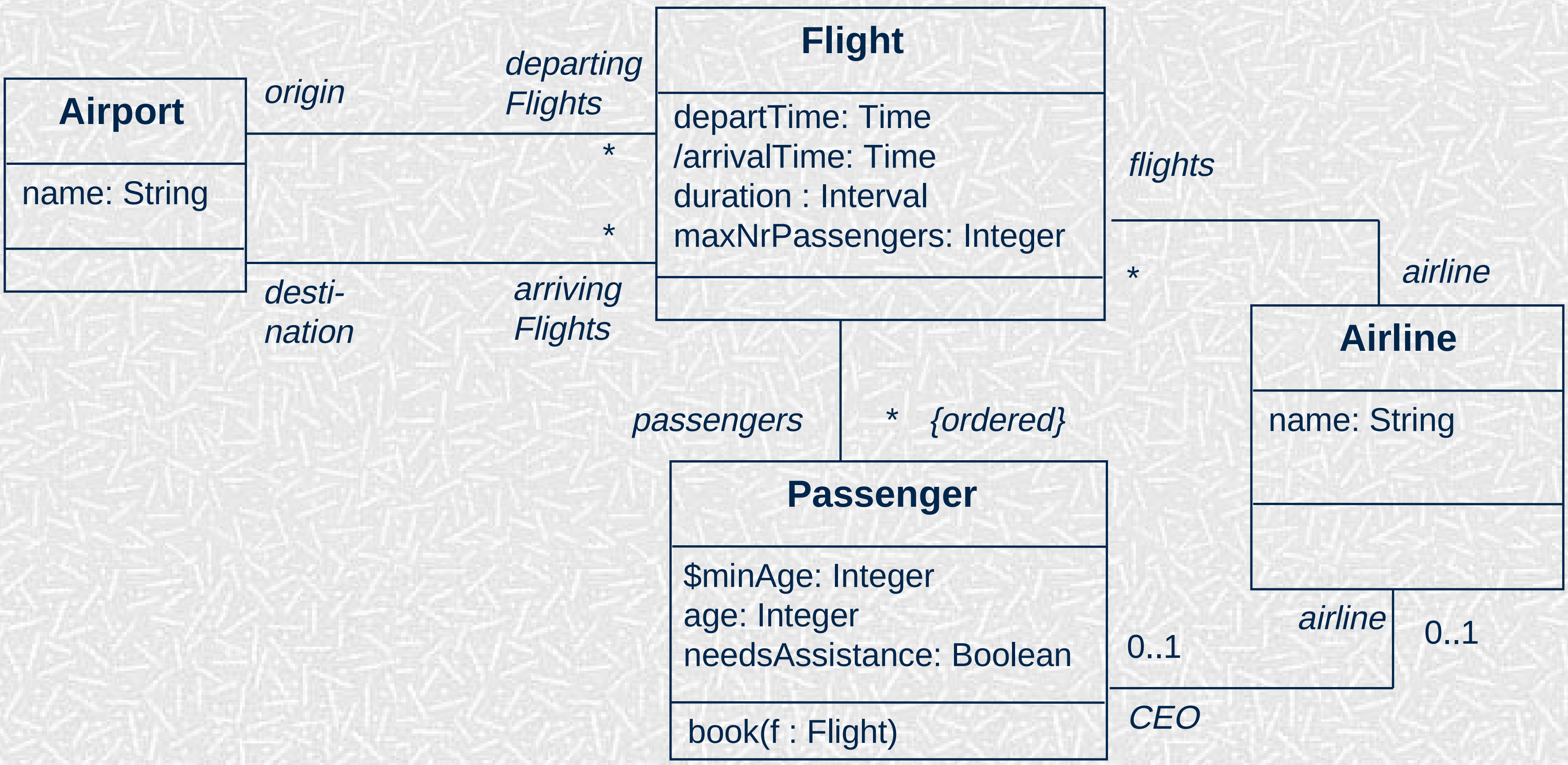
Constraints and the UML model

- OCL expressions are always bound to a UML model

Overview

- What are constraints
- Core OCL Concepts
- Advanced OCL Concepts
- Wrap up

Example model



Constraint context and self

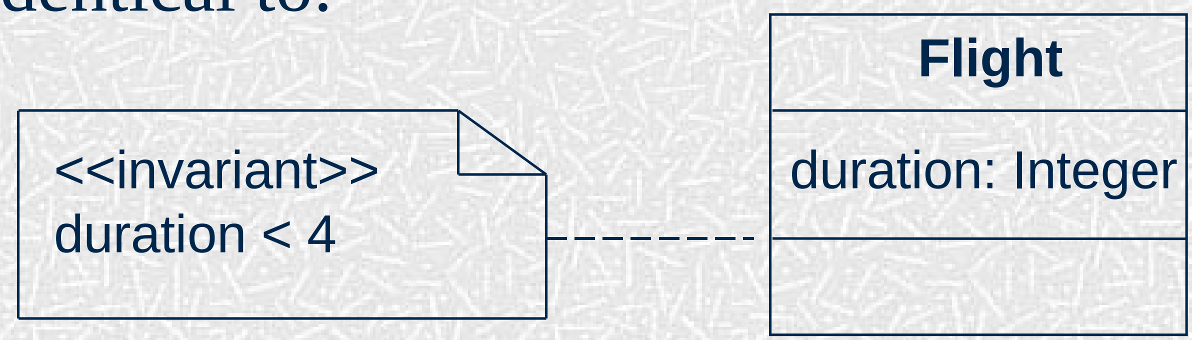
- Every OCL expression is bound to a specific context.
- The context may be denoted within the expression using the keyword ‘self’.

Notation

- Constraints may be denoted within the UML model or in a separate document.
  - the expression:

    context Flight inv: self.duration < 4
  - is identical to:

    context Flight inv: duration < 4
  - is identical to:



Elements of an OCL expression

- In an OCL expression these elements may be used:
  - basic types: String, Boolean, Integer, Real.
  - classifiers from the UML model and their features
    - attributes, and class attributes
    - query operations, and class query operations
  - associations from the UML model

Example: OCL basic types

```
context Airline inv:
name.toLower = ‘klm’

context Passenger inv:
age >= ((9.6 - 3.5)* 3.1).floor implies
mature = true
```

Model classes and attributes

- “Normal” attributes

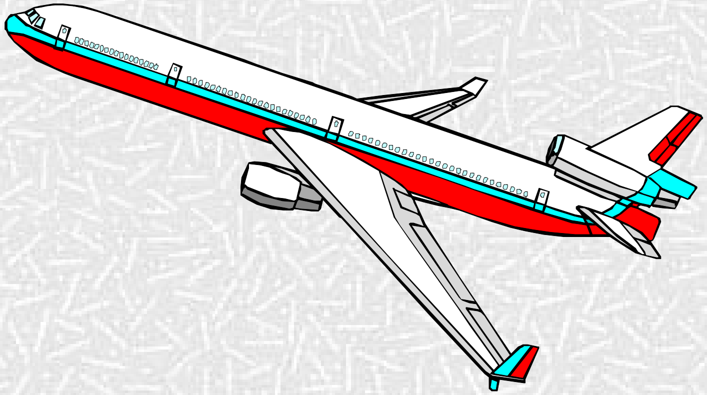
  context Flight inv:

  self.maxNrPassengers <= 1000

- Class attributes

  context Passenger inv:

  age >= Passenger.minAge



Example: query operations

context Flight inv:

self.departTime.difference(self.arrivalTime)

.equals(self.duration)

Time
$midnight: Time month : String day : Integer year : Integer hour : Integer minute : Integer
difference(t:Time):Interval before(t: Time): Boolean plus(d : Interval) : Time

Interval
nrOfDays : Integer nrOfHours : Integer nrOfMinutes : Integer
equals(i:Interval):Boolean $Interval(d, h, m : Integer) : Interval

Associations and navigations

- Every association is a navigation path.
- The context of the expression is the starting point.

- Role names are used to identify the navigated association.

Example: navigations

- Navigations

```
context Flight
inv: origin <> destination
inv: origin.name = ‘Amsterdam’

context Flight
inv: airline.name = ‘KLM’
```

Association classes

context Person inv:

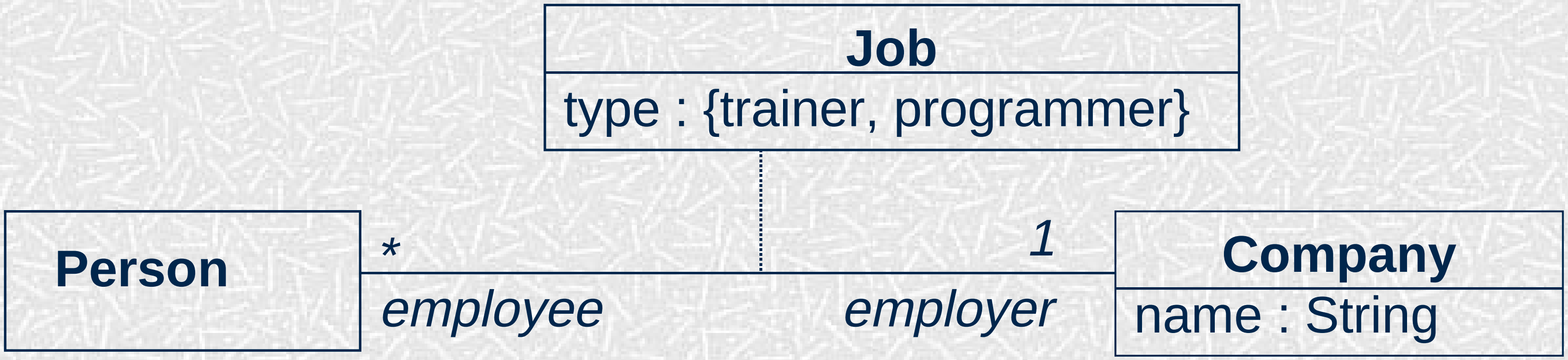
if employer.name = ‘Klasse Objecten’ then

job.type = #trainer

else

job.type = #programmer

endif



The OCL Collection types

- What are constraints
- Core OCL Concepts
  - Collections
- Advanced OCL Concepts
- Wrap up

Three subtypes to Collection

- Set:
  - arrivingFlights(from the context Airport)
- Bag:
  - arrivingFlights.duration (from the context Airport)
- Sequence:
  - passengers (from the context Flight)

Collection operations

- OCL has a great number of predefined operations on the collections types.

- Syntax:

collection->operation

The collect operation

- Syntax:

  collection->collect(elem : T | expr)

  collection->collect(elem | expr)

  collection->collect(expr)

- Shorthand:

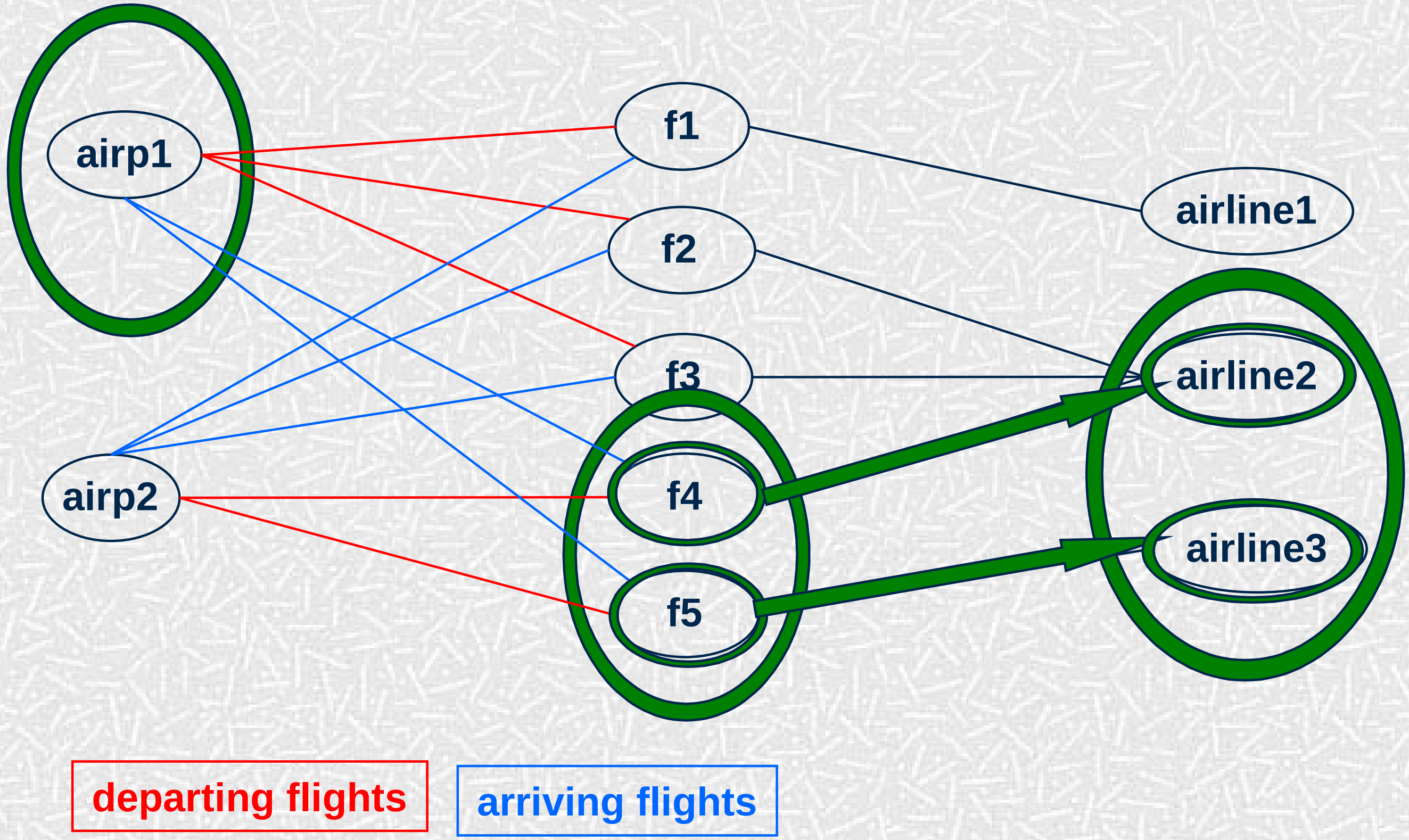
  collection.expr

- The *collect* operation results in the collection of the values resulting evaluating *expr* for all elements in the *collection*

Example: collect operation

context Airport inv:

self.arrivingFlights->collect(airLine)->notEmpty



The select operation

- Syntax:

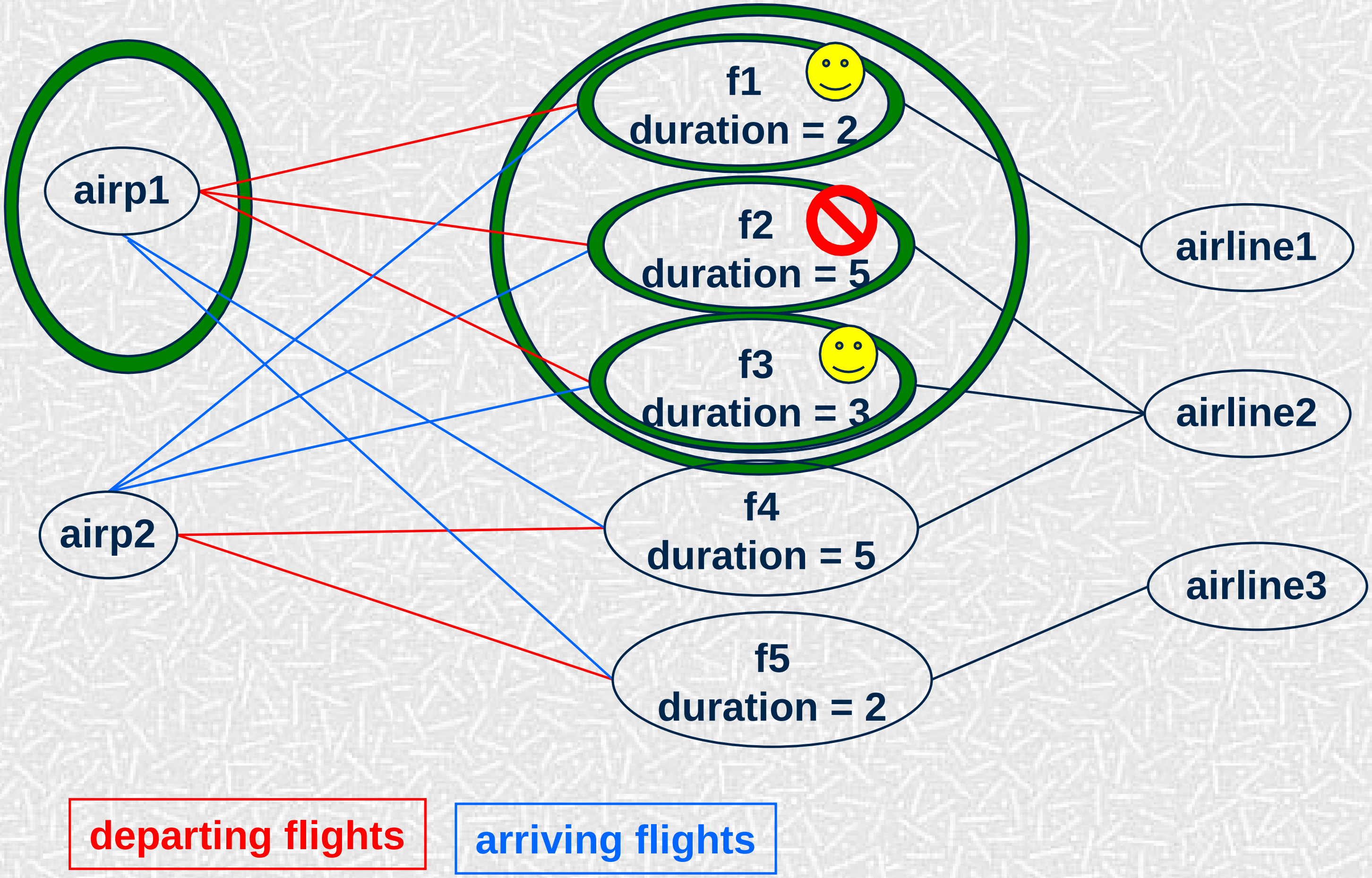
```
collection->select(elem : T | expression)
collection->select(elem | expression)
collection->select(expression)
```

- The *select* operation results in the subset of all elements for which *expression* is true

Example: collect operation

context Airport inv:

self.departingFlights->select(duration<4)->notEmpty



The forAll operation

- Syntax:

```
collection->forAll(elem : T | expr)
collection->forAll(elem | expr)
collection->forAll(expr)
```

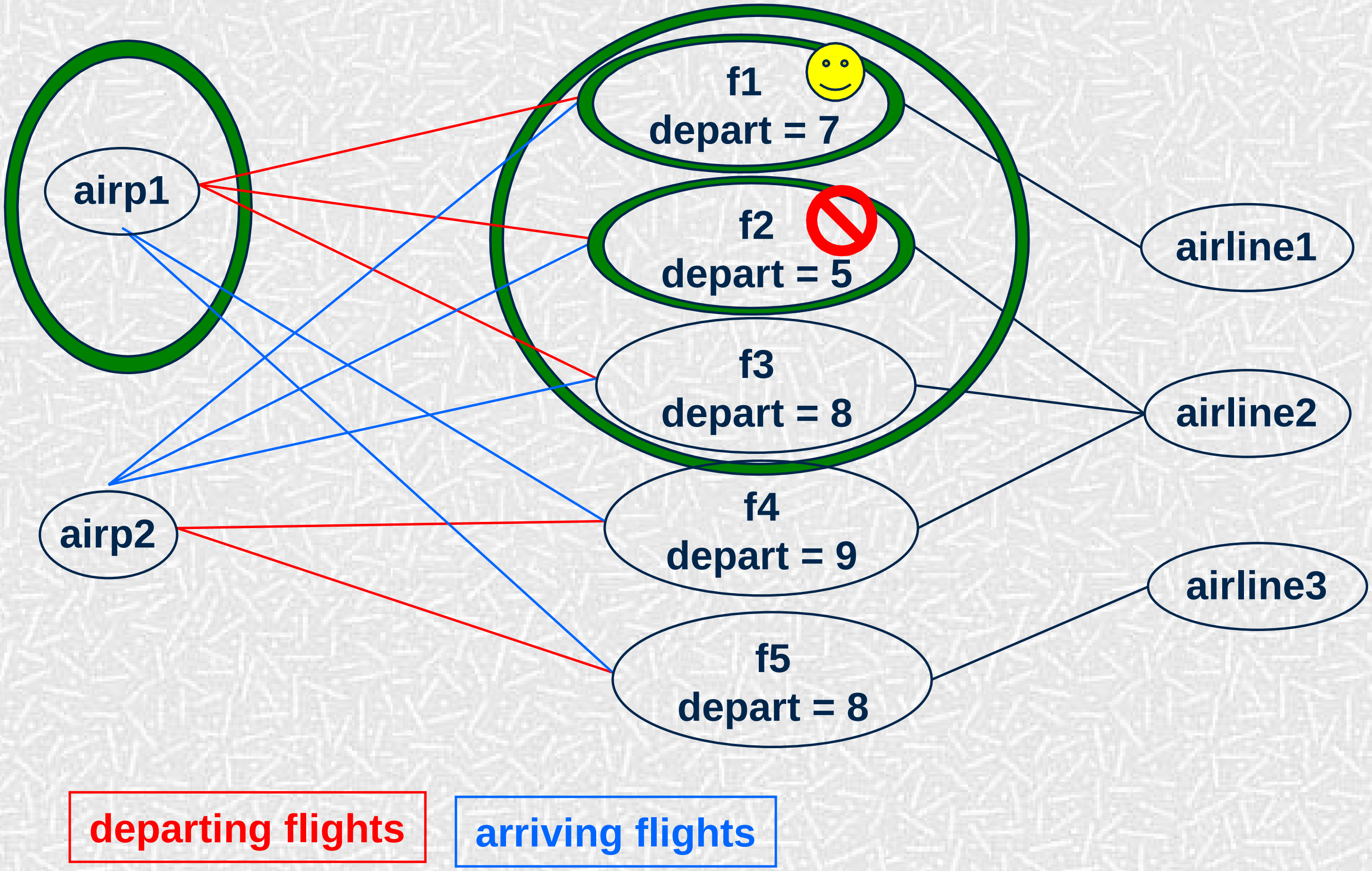
- The *forAll* operation results in true if *expr* is true for all elements of the collection

$\forall$

Example: forAll operation

context Airport inv:

self.departingFlights->forAll(departTime.hour>6)



The exists operation

- Syntax:

```
collection->exists(elem : T | expr)
collection->exists(elem | expr)
collection->exists(expr)
```

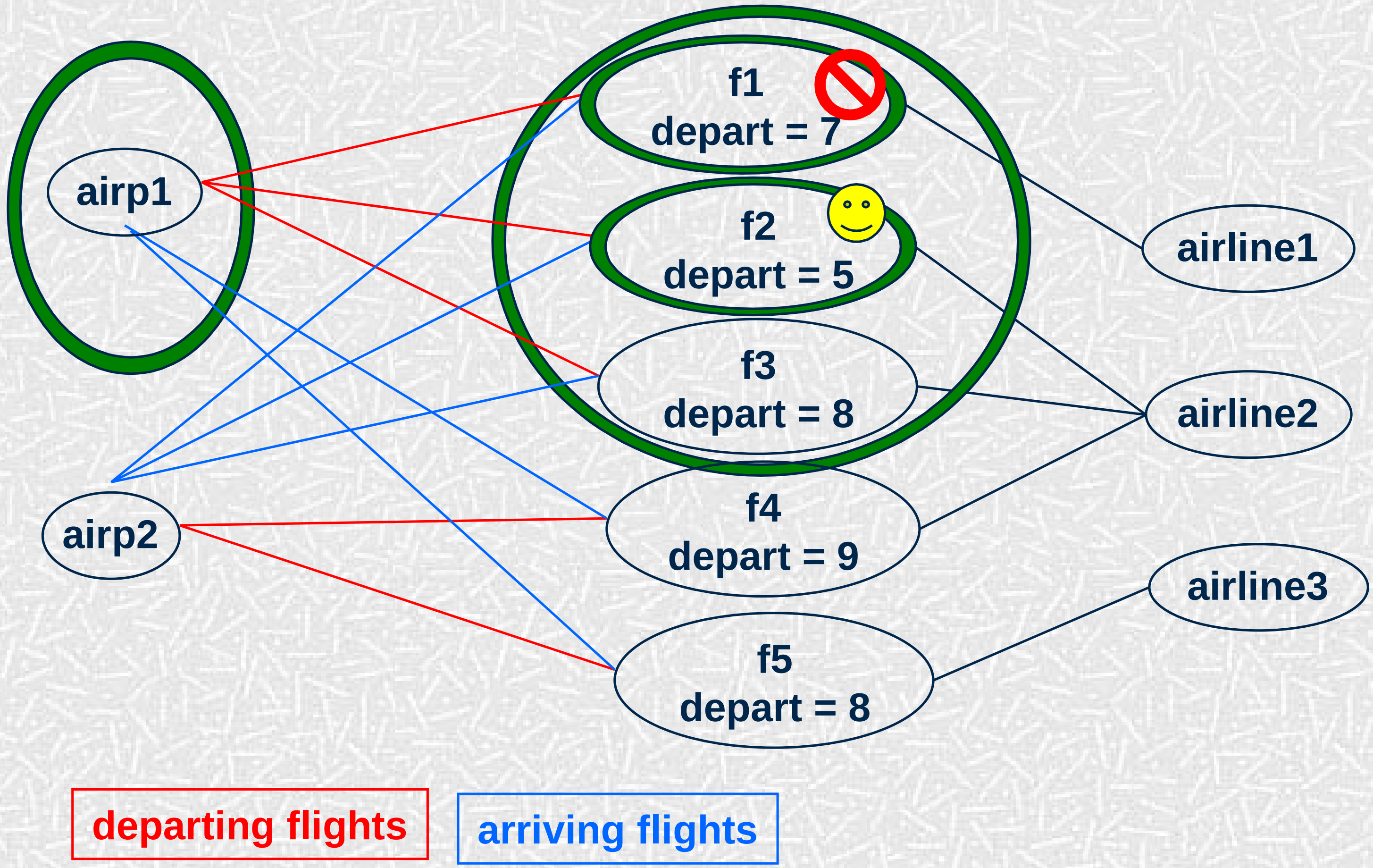
- The *exists* operation results in true if there is at least one element in the collection for which the expression *expr* is true.

$\exists$

Example: exists operation

context Airport inv:

self.departingFlights->exists(departTime.hour<6)



Example: exists operation

```
context Airport inv:
self.departingFlights ->
        exists(departTime.hour < 6)
```

Other collection operations

- *isEmpty*: true if collection has no elements
- *notEmpty*: true if collection has at least one element
- *size*: number of elements in collection
- *count(elem)*: number of occurences of elem in collection
- *includes(elem)*: true if elem is in collection
- *excludes(elem)*: true if elem is not in collection
- *includesAll(coll)*: true if all elements of coll are in collection

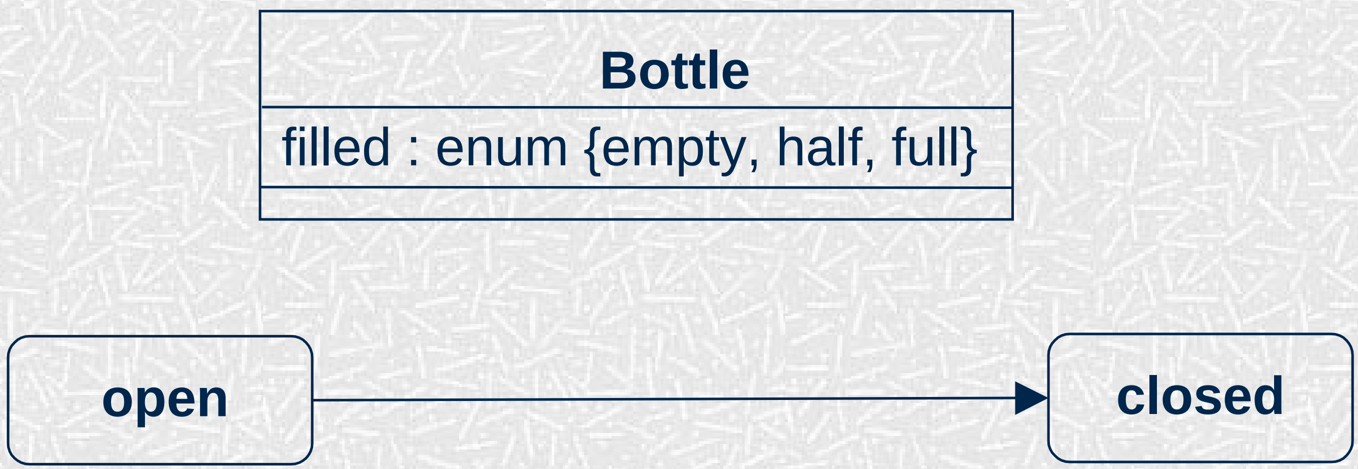
Result in postcondition

- Example pre and postcondition

```
context Airline::servedAirports() : Set(Airport)
pre :  -- none
post: result = flights.destination->asSet
```

Statechart: referring to states

- The operation *oclInState* returns true if the object is in the specified state.



context Bottle inv:
self.oclInState(closed) implies filled = #full

Local variables

- The Let construct defines variables local to one constraint:

Let var : Type = <expression1> in <expression2>

Iterate

- The *iterate* operation for collections is the most generic and complex building block.

```
collection->iterate(elem : Type;
                    answer : Type = <value> |
    <expression-with-elem-and-answer>)
```

Iterate example

- Example iterate:

```
context Airline inv:
flights->select(maxNrPassengers > 150)->notEmpty
```

- Is identical to:

```
context Airline inv:
flights->iterate(f : Flight; answer : Set(Flight) = Set{ } |
if f.maxNrPassengers > 150 then
   answer->including(f)
else answer endif )->notEmpty
```

Inheritance of constraints

- Guiding principle Liskovs Substitution Principle (LSP):
  - "Whenever an instance of a class is expected, one can always substitute an instance of any of its subclasses."

Inheritance of constraints

- Consequences of LSP for invariants:
  - An invariant is always inherited by each subclass.
  - Subclasses may strengthen the invariant.

- Consequences of LSP for preconditions and postconditions:
  - A precondition may be weakened
  - A postcondition may be strengthened

Wrap up

- What are constraints
- Core OCL Concepts
- Advanced OCL Concepts
- Wrap up

Current Developments

- Feedback from several OCL implementors handled in UML-RTF
  - e.g. the grammar has some loose ends
  - typical tool-related issues
- Development of OCL metamodel
  - currently concrete syntax only
  - will result in abstract syntax
- OCL Workshop with pUML group
  - formalization of OCL

OCL Tools

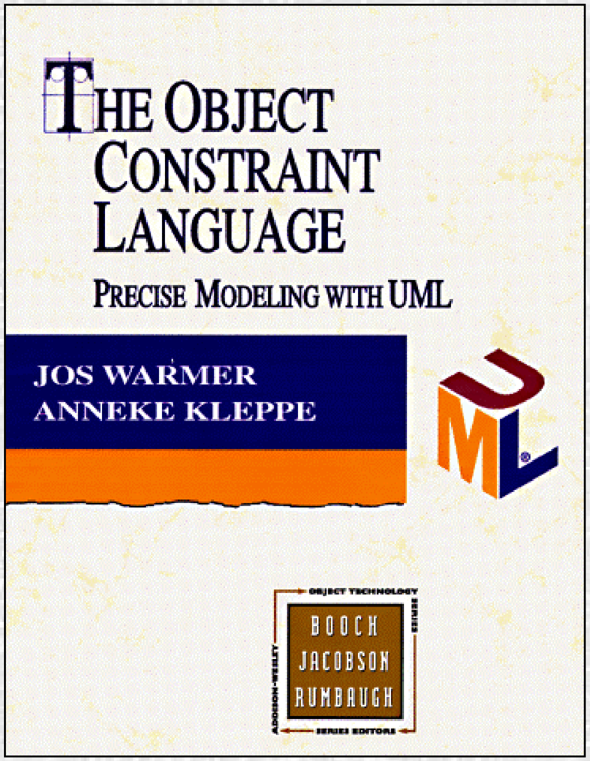
- Cybernetics
  - ww.cybernetic.org
- University of Dresden
  - www-st.inf.tu-dresden.de/ocl/
- Boldsoft
  - www.boldsoft.com
- ICON computing
  - www.iconcomp.com
- Royal Dutch Navy
- Others … …

Conclusions and Tips

- OCL invariants allow you to
  - model more precisely
  - stay implementation independent
- OCL pre- and postconditions allow you to
  - specify contracts (design by contract)
  - precisely specify interfaces of components
- OCL usage tips
  - keep constraints simple
  - always combine natural language with OCL
  - use a tool to check your OCL

Further resources on OCL

- The Object Constraint Language
  - ISBN 0-201-37940-6
- OCL home page
  - www.klasse.nl/ocl/index.htm



Preview - Next Tutorial

Metadata Integration with UML, XMI, and MOF

- 4-Layer Metamodel Architecture
- UML CORBAfacility
- UML XMI DTD
- Meta Object Facility

Further Info

Web

- OMG UML Resource Page
  - www.omg.org/uml/
- UML Tutorial 1 (OMG Document **omg/99-11-04**)
  - www.omg.org/cgi-bin/doc?omg/99-11-04
- UML Tutorial 2 (OMG Document **omg/00-01-01**)
  - www.omg.org/cgi-bin/doc?omg/00-01-01
- UML Tutorial 3 (will be posted)

Email

- Karin Palmkvist: karin.palmkvist@enea.se
- Bran Selic: bran@objectime.com
- Jos Warmer: j.warmer@klasse.nl

Conferences & workshops

- UML World 2000, NYC, June '00
- UML '00, York, England, Oct. '00

Questions

Tool demonstrations

- For anyone interested there are tool demonstrations directly after this tutorial

  - University of Dresden
  - Cybernetics