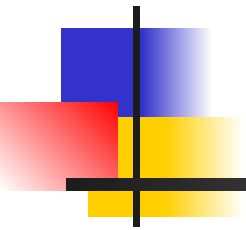
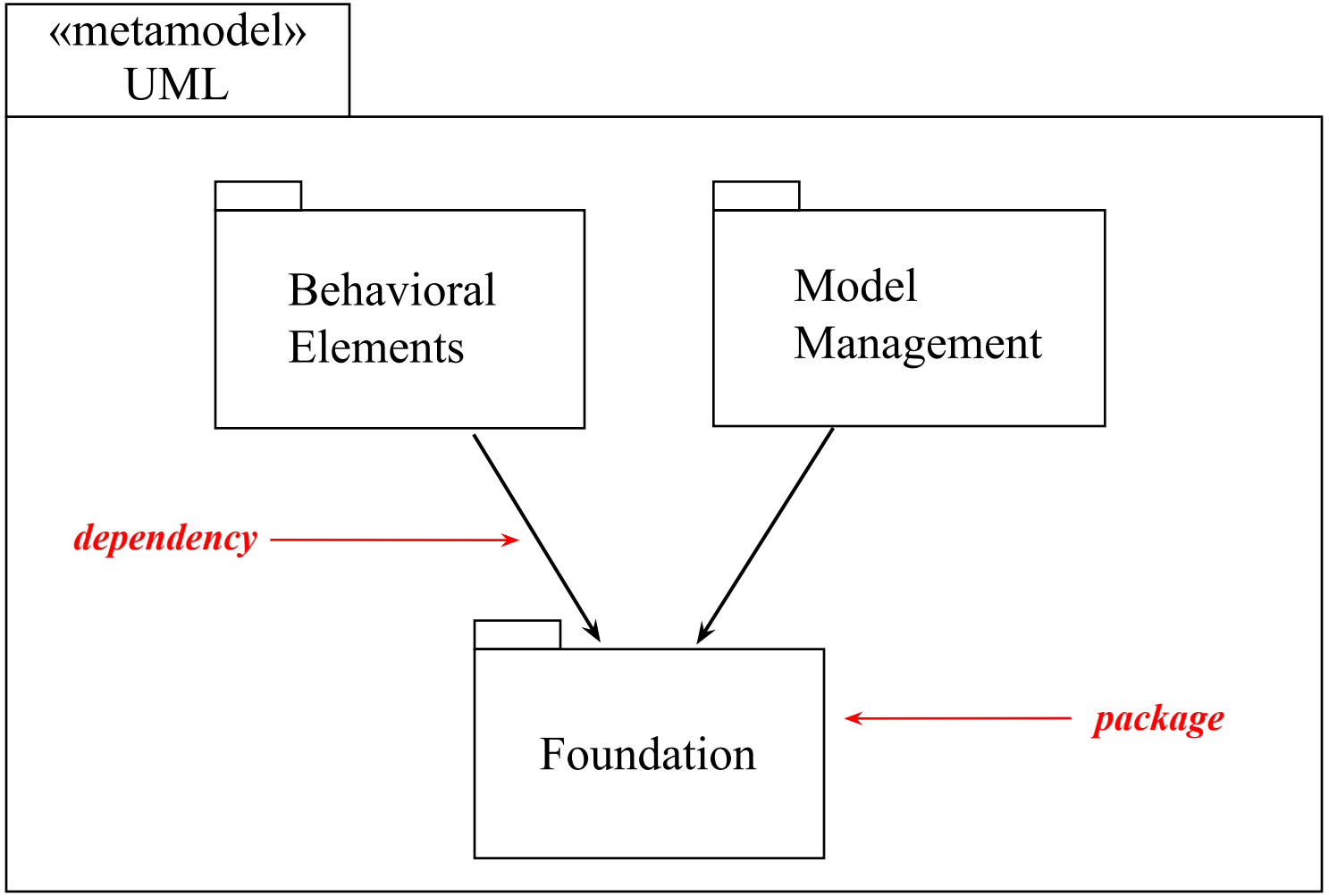
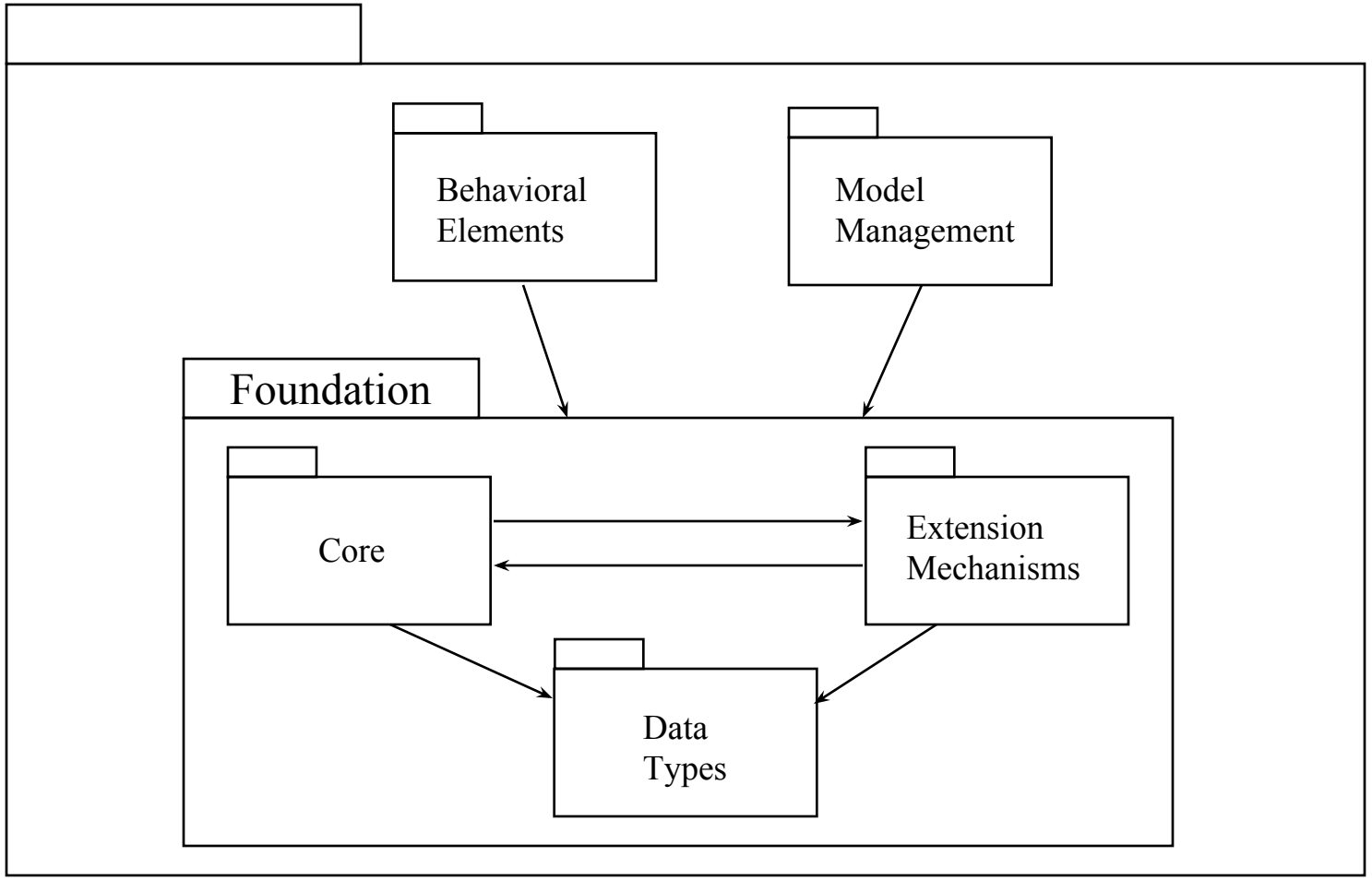


UNIT III
Packages & Package Diagram

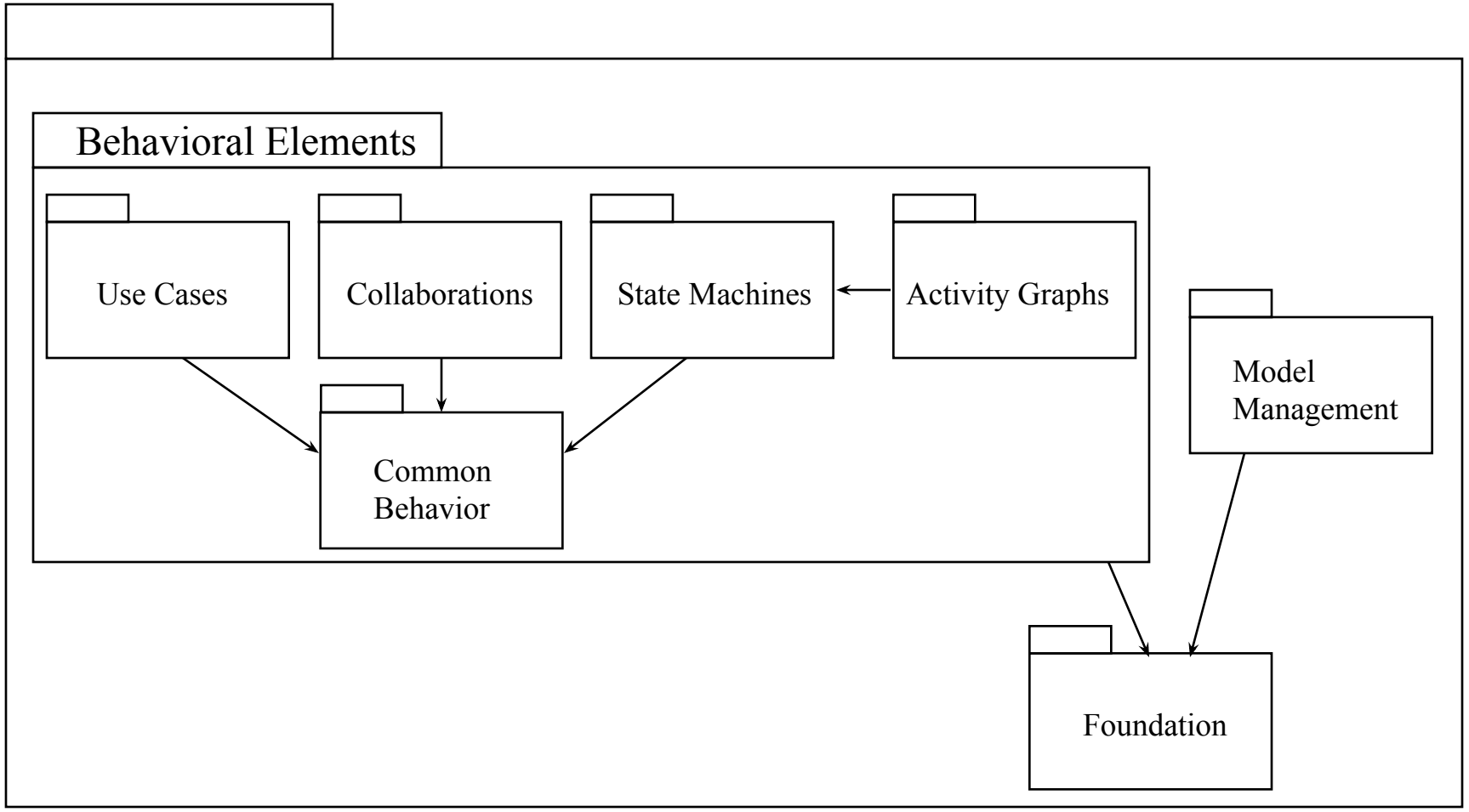
UML Overview

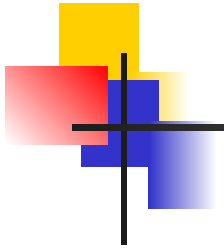


UML Overview



UML Overview

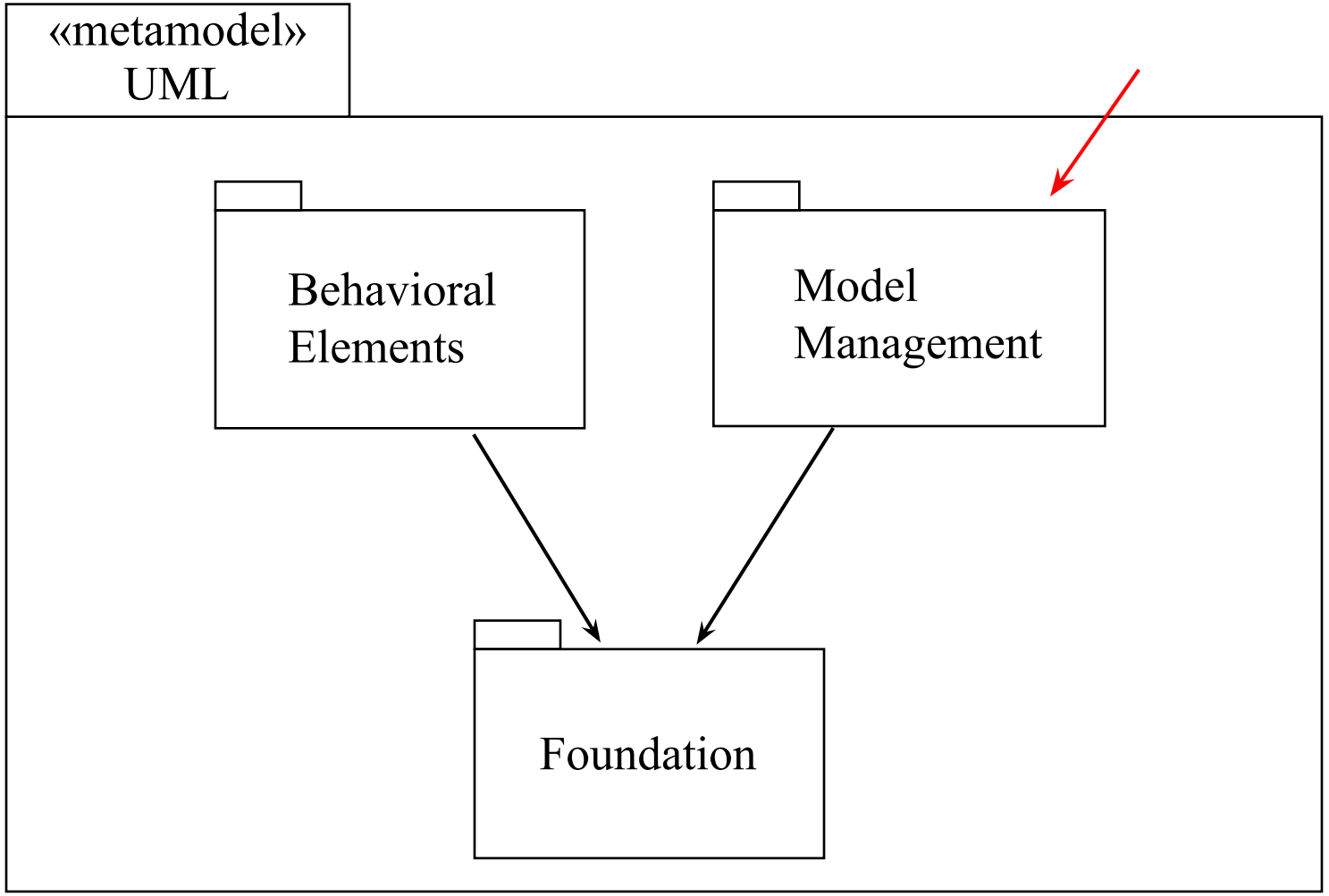




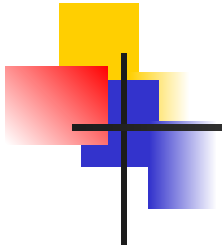
Advanced Modeling with UML

- Model Management

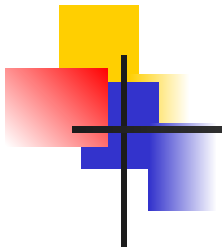
UML Overview



Model Management Overview

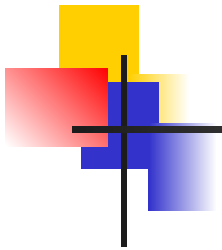


- Main UML constructs used for model management:
  - Package
  - Subsystem
  - Model



Unifying Concepts

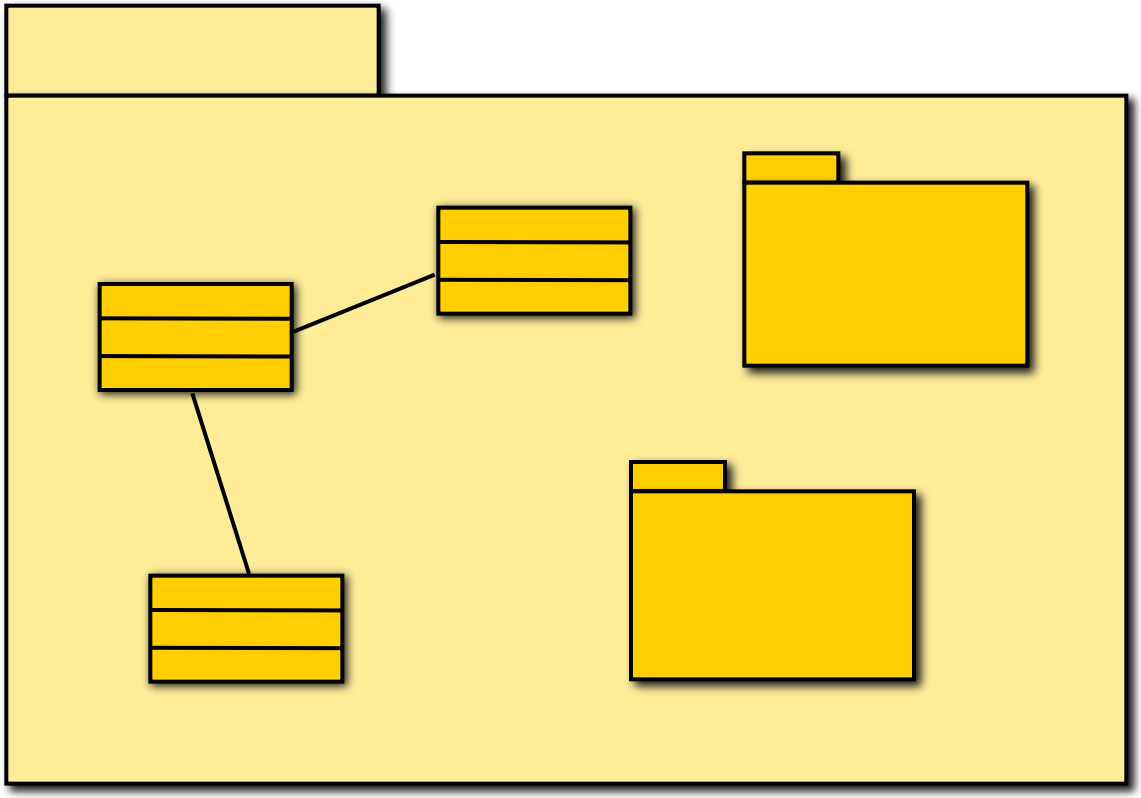
- Packages, Subsystems, and Models
  - Group other model elements together
  - Each groups elements for a different reason (providing different semantics)
- Other grouping elements in UML include:
  - Classes
  - Components



Package

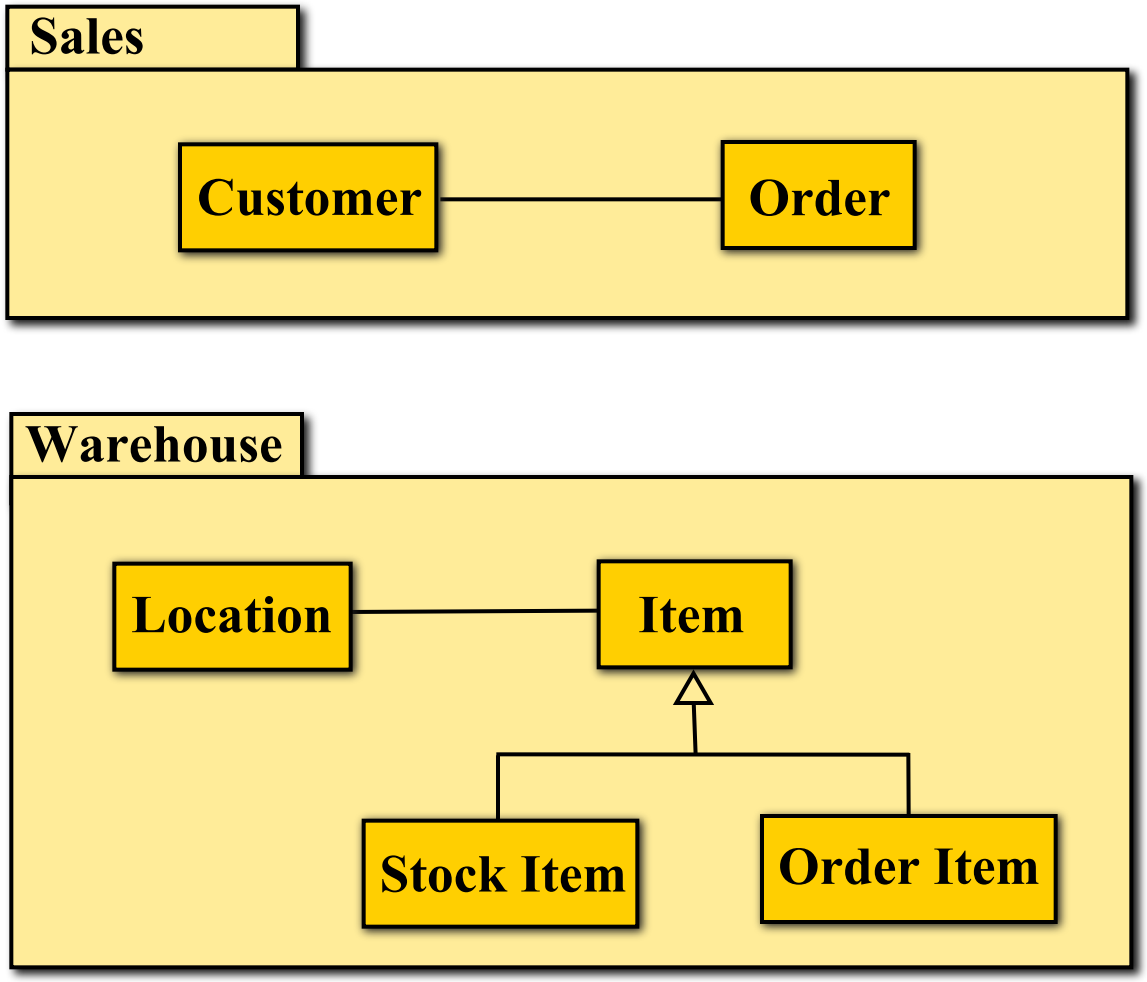
- What are Packages?
- Core Concepts
- Diagram Tour
- When to Use Packages
- Modeling Tips

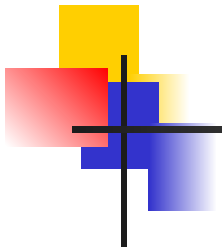
Package



A package is a grouping of model elements

Package – Example

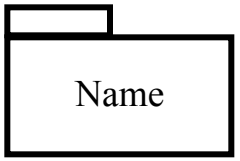
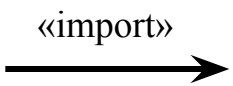
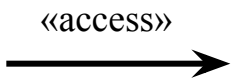


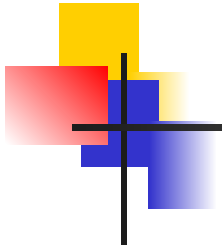


Package

- A package can contain model elements of different kinds
  - Including other packages to create hierarchies
- A package defines a namespace for its contents
- Packages can be used for various purposes

Core Concepts

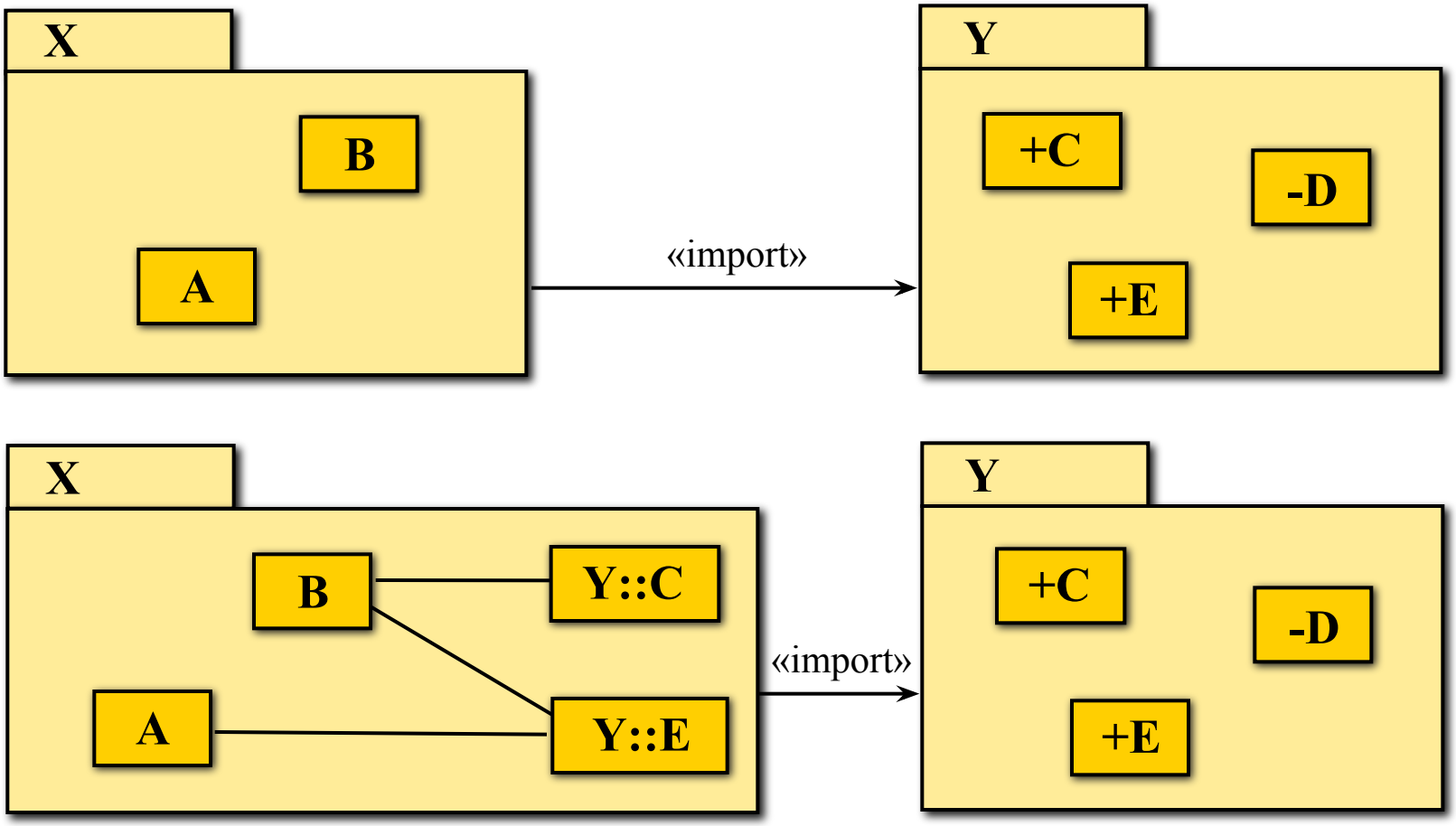
Construct	Description	Syntax
Package	A grouping of model elements.	
Import	A dependency indicating that the public contents of the target package are added to the namespace of the source package.	
Access	A dependency indicating that the public contents of the target package are available in the namespace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



Visibility

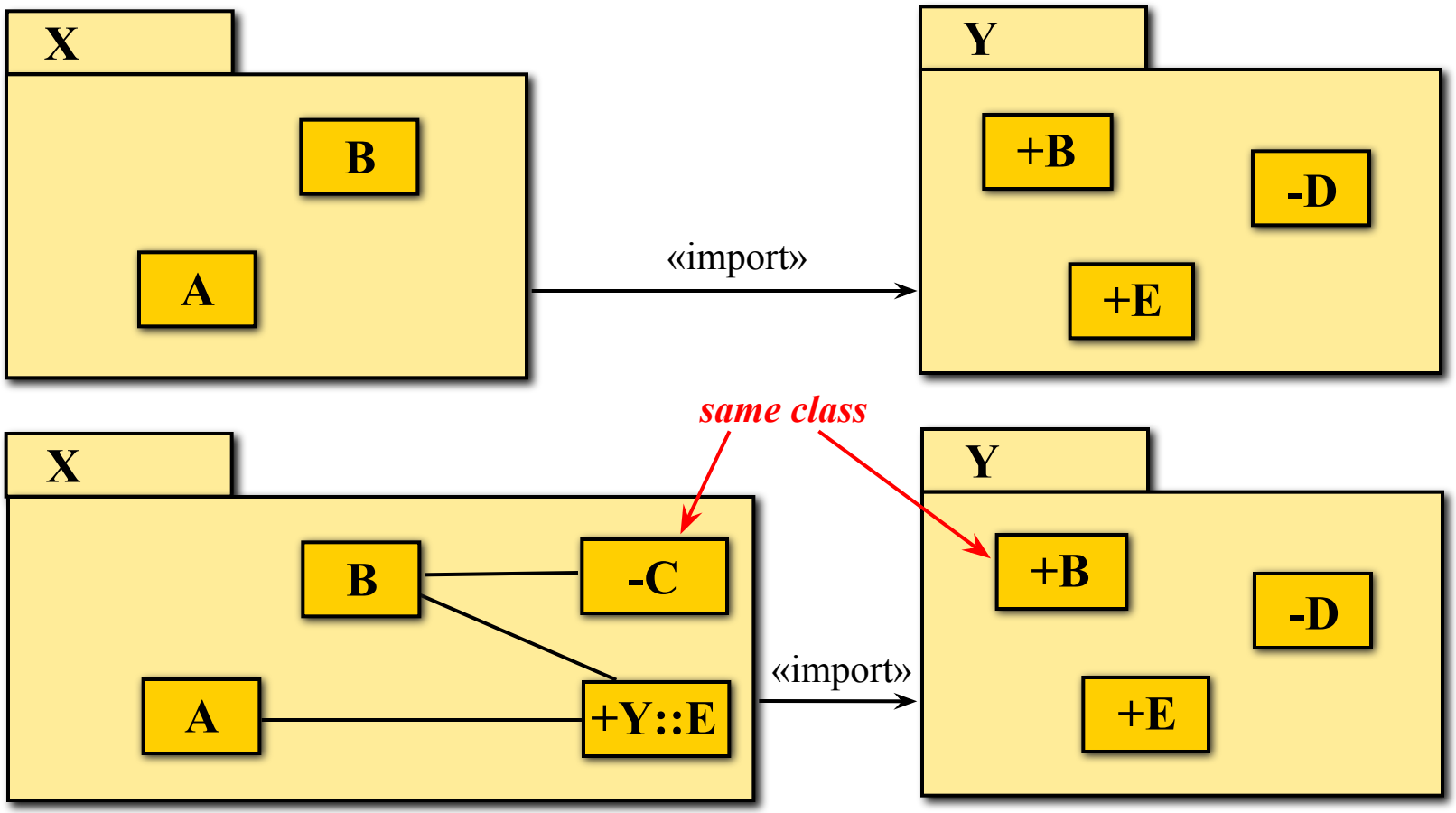
- Each contained element has a visibility relative to the containing package
  - A *public* element is visible to elements outside the package, denoted by '+'
  - A *protected* element is visible only to elements within inheriting packages, denoted by '#'
  - A *private* element is not visible at all to elements outside the package, denoted by '-'
- Same syntax for visibility of attributes and operations in classes

Import



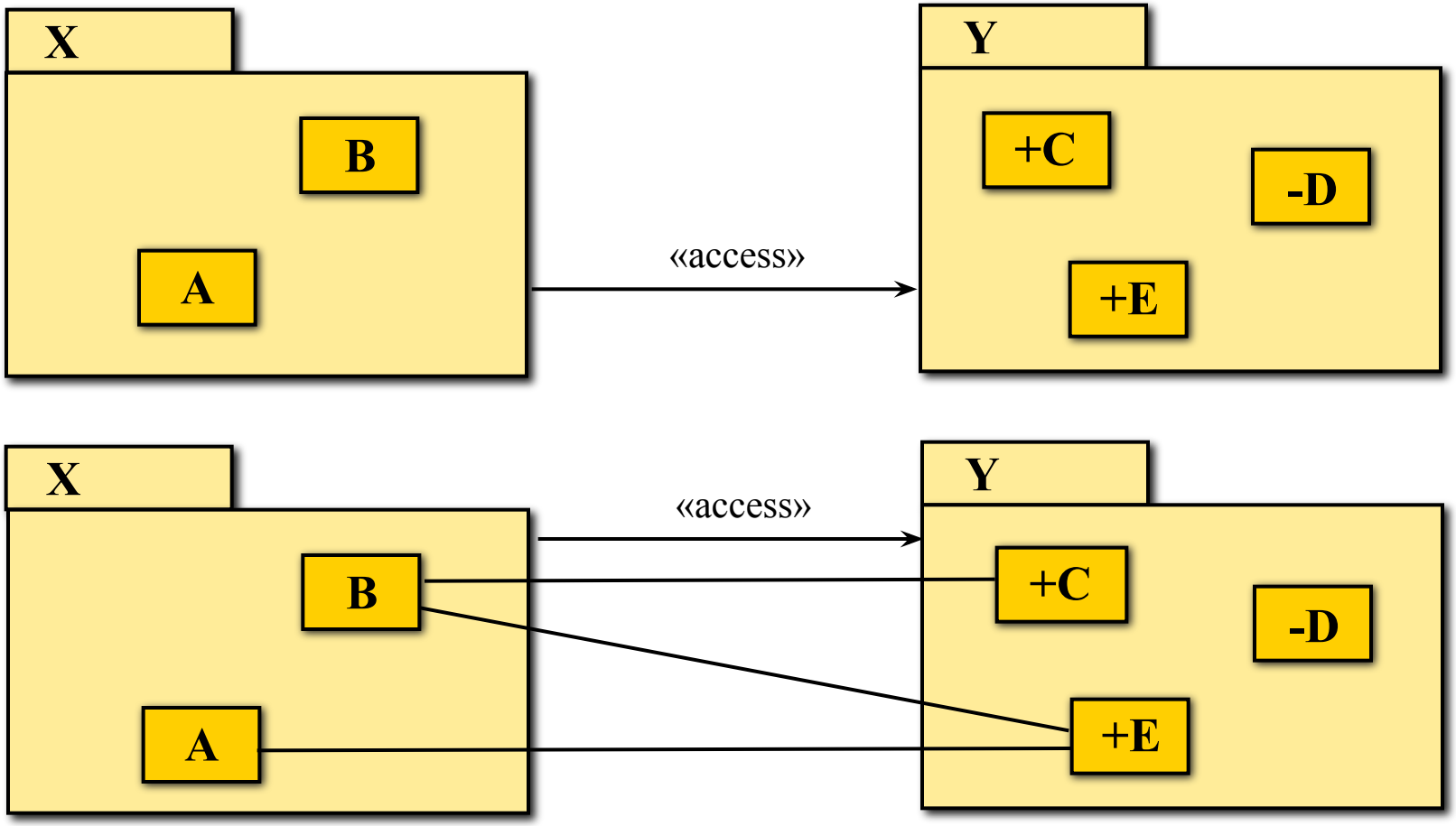
The associations are owned by package X

Import – Alias



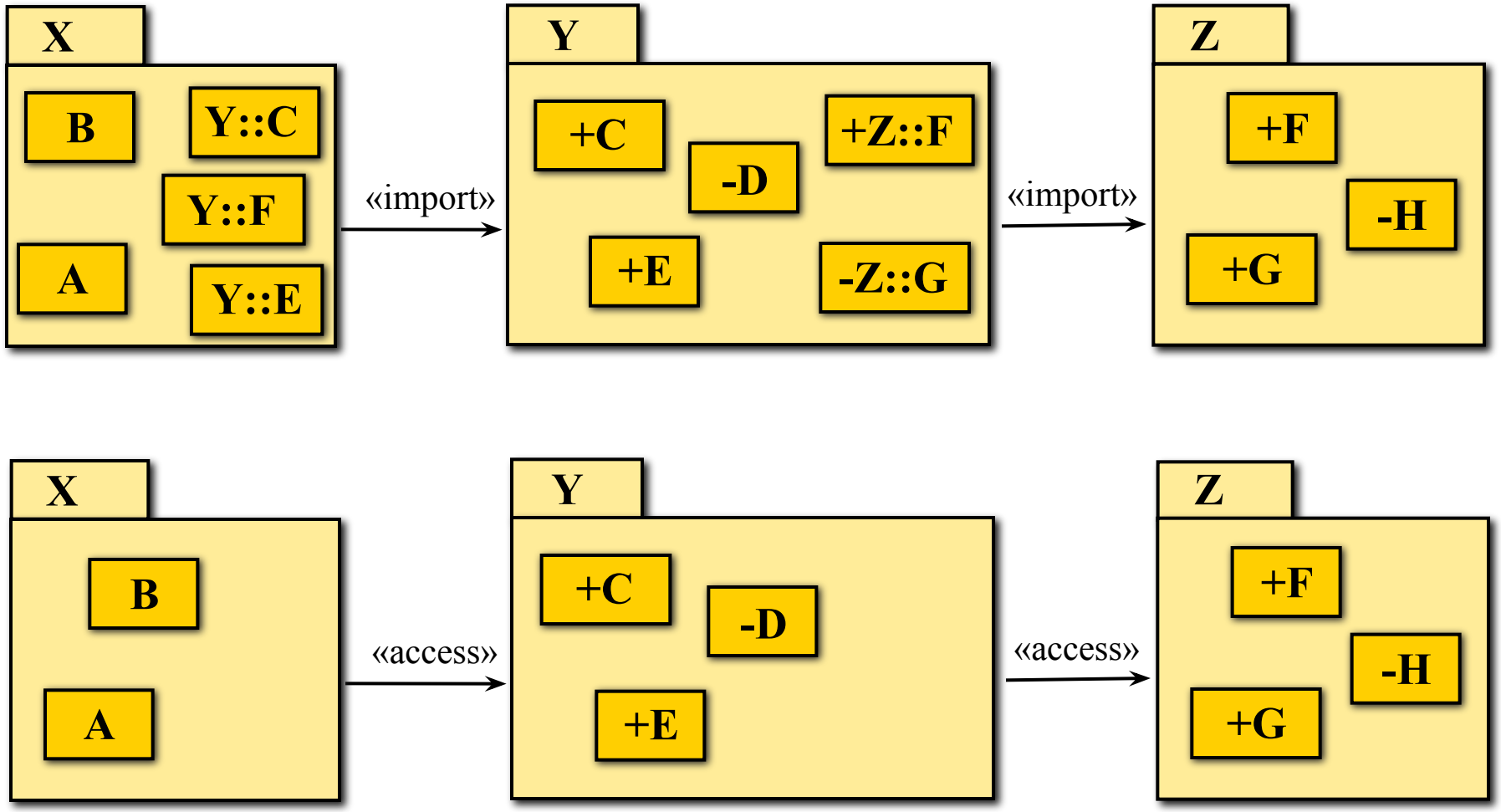
An imported element can be given a local alias and a local visibility

Access



The associations are owned by package X

Import vs. Access



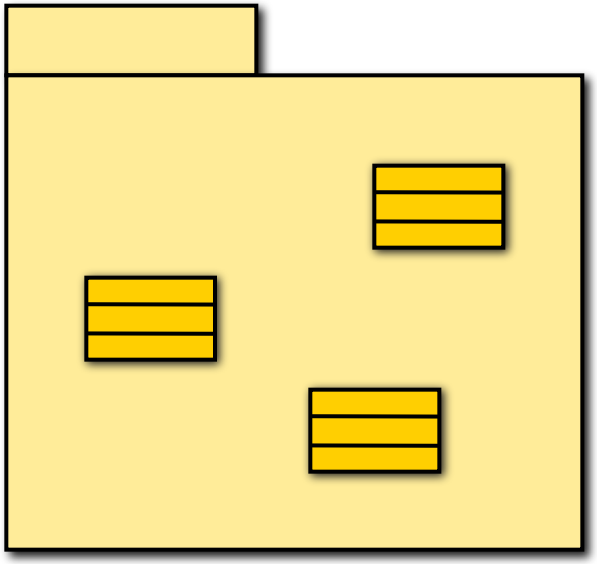
Package Inheritance

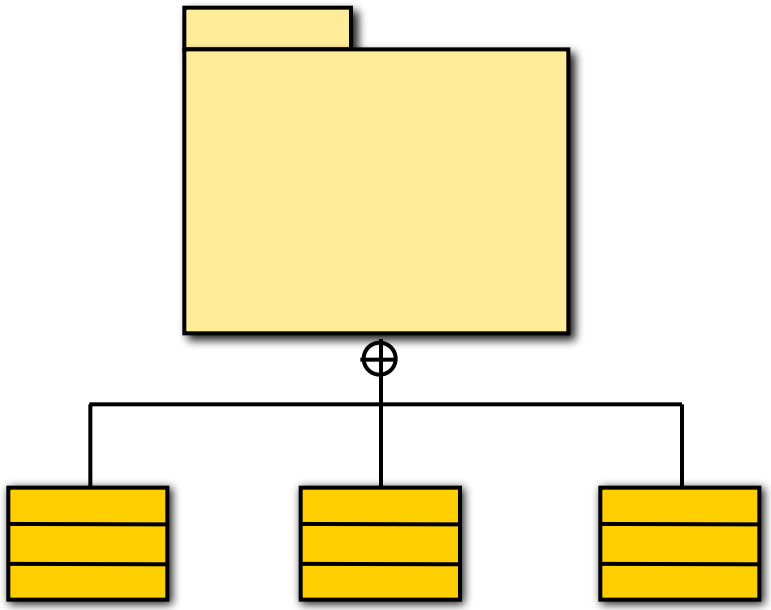
- A package with a generalization to another package inherits public and protected elements that are
  - owned or
  - imported

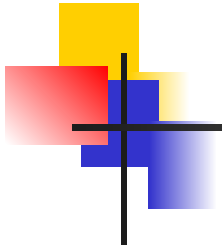
  by the inherited package

Diagram Tour

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





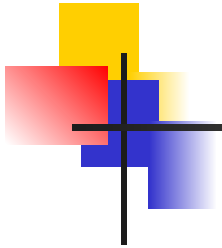


When to Use Packages

- 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

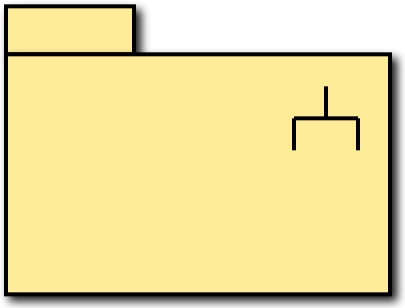
- Gather model elements with strong cohesion in one package
- Keep model elements with low coupling in different packages
- Minimize relationships, especially associations, between model elements in different packages
- Namespace implication: an element imported into a package does not “know” how it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

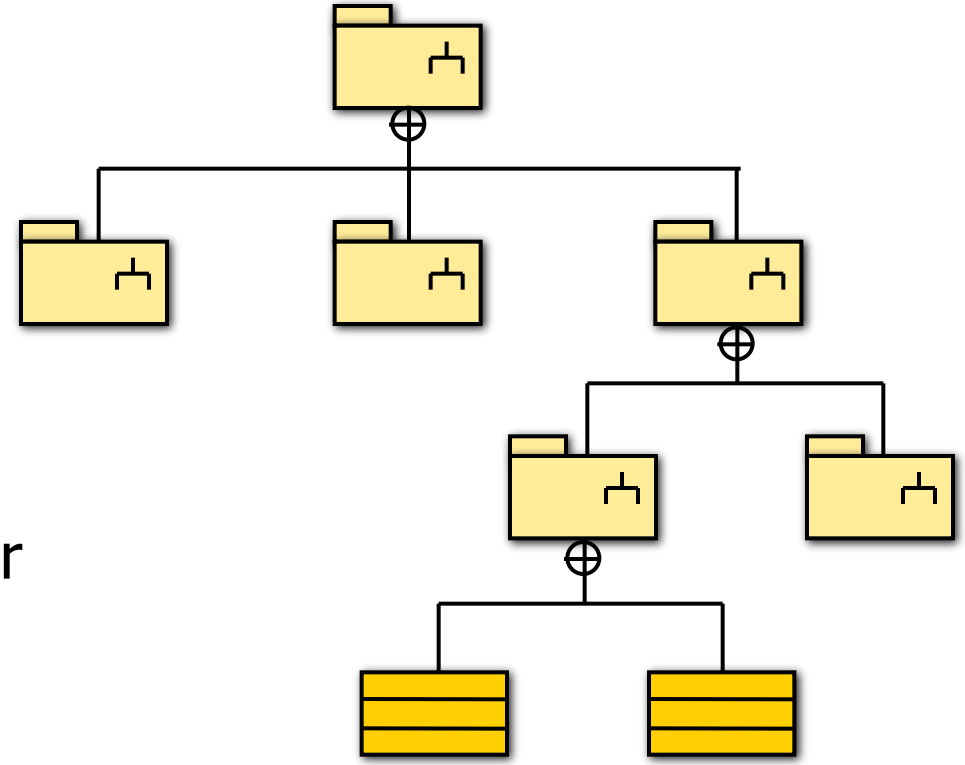


Subsystem

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

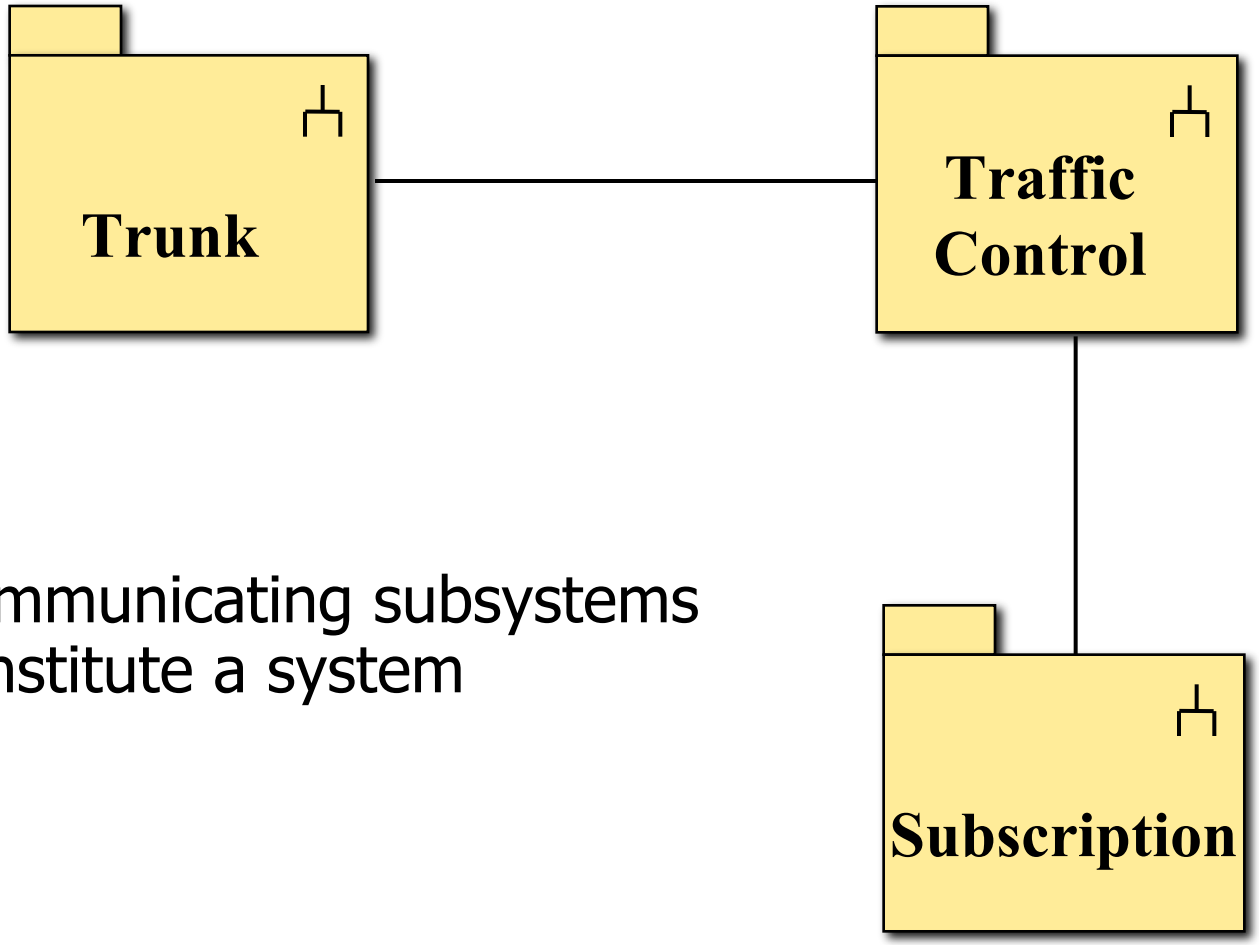
Subsystem





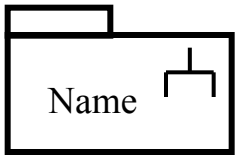
Subsystems are used for system decomposition

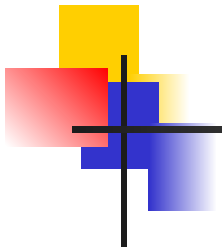
Subsystem – Example



Communicating subsystems constitute a system

Core Concepts

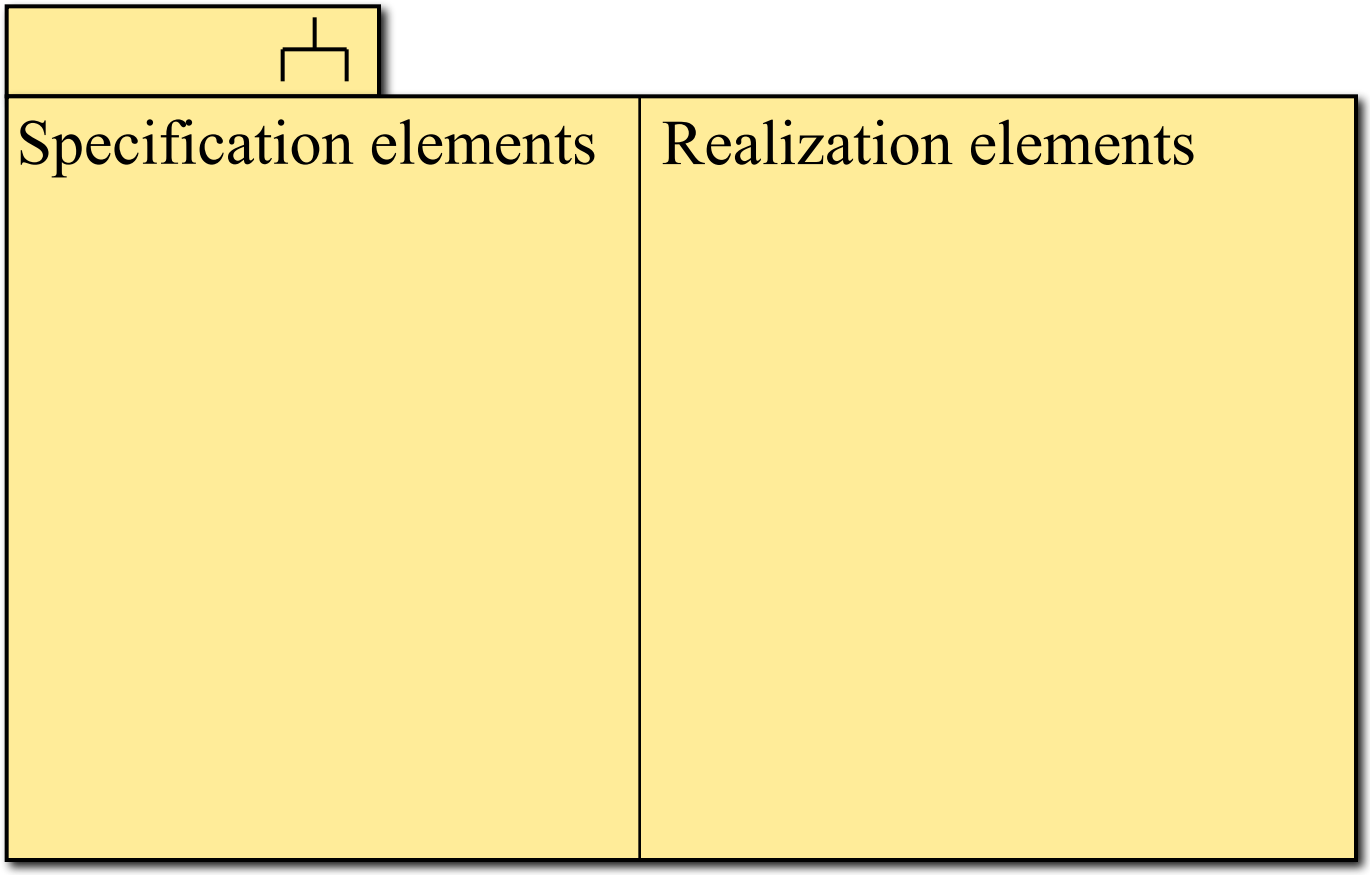
Construct	Description	Syntax
Subsystem	A grouping of model elements that represents a behavioral unit in a physical system.	



Subsystem Aspects

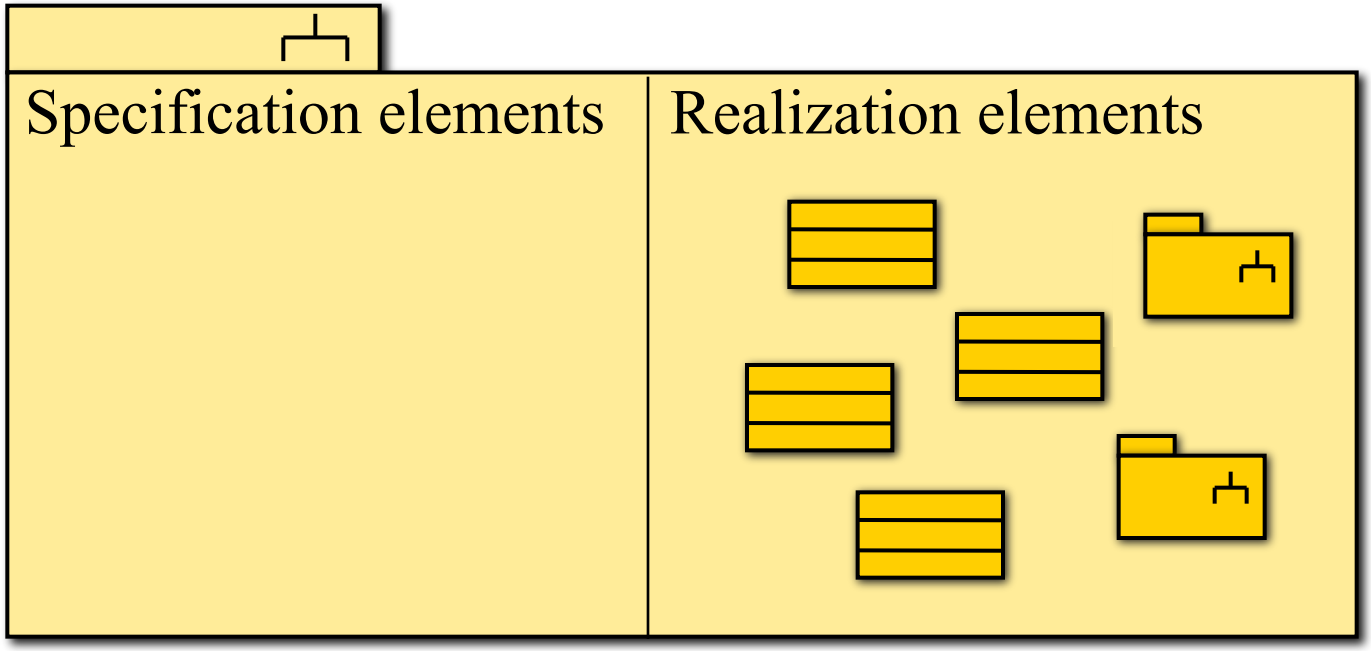
- A subsystem has two aspects:
  - An external view, showing the services provided by the subsystem
  - An internal view, showing the realization of the subsystem
- There is a mapping between the two aspects

Subsystem Aspects



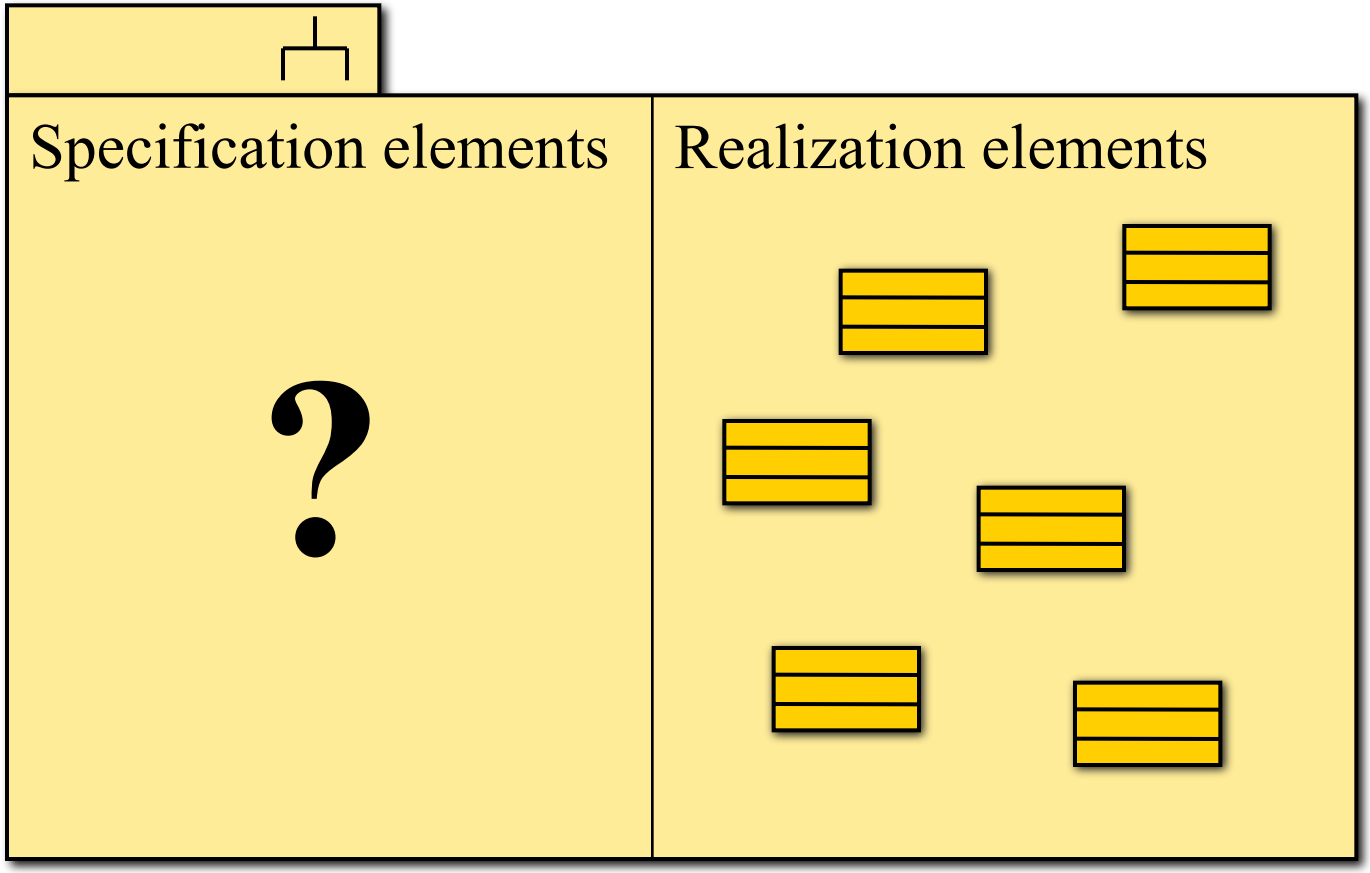
A subsystem has a specification and a realization to represent the two views

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

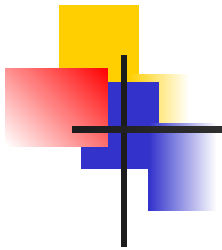
Subsystem Specification



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

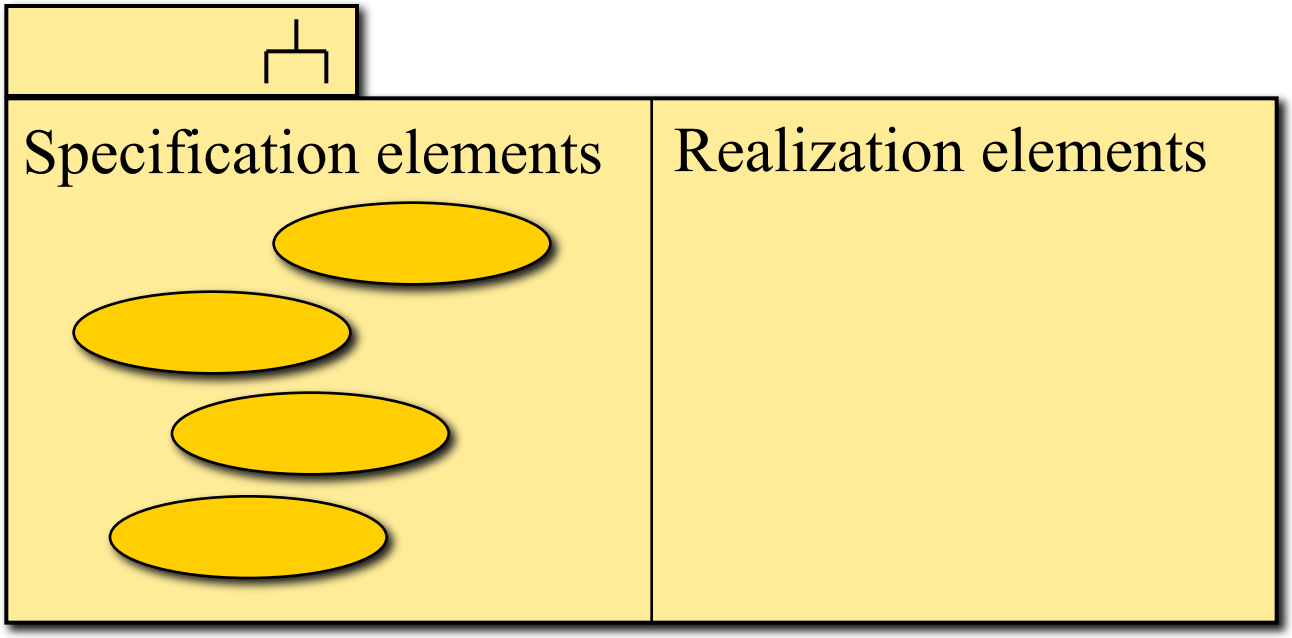


Specification Techniques

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

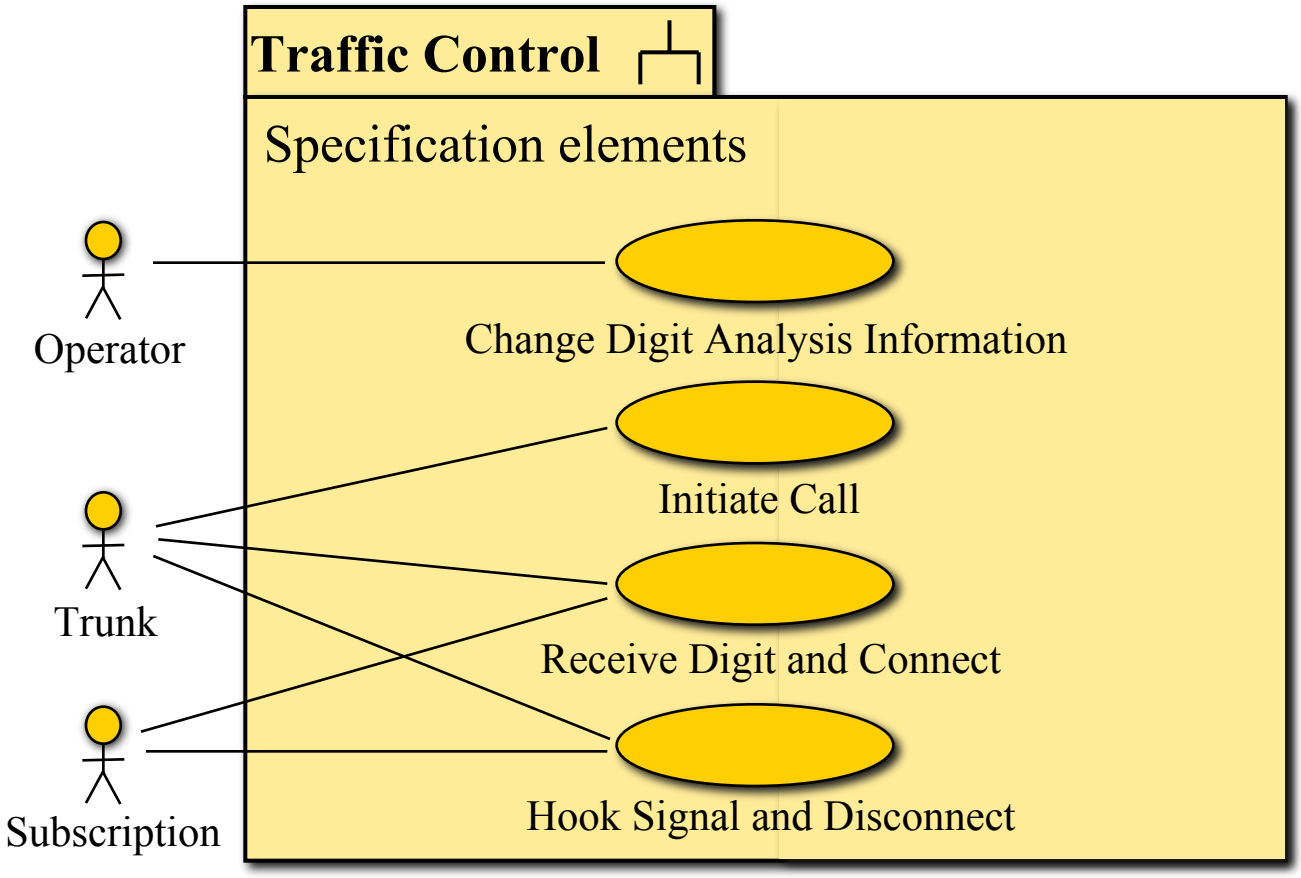
…and combinations of these.

Use Case Approach

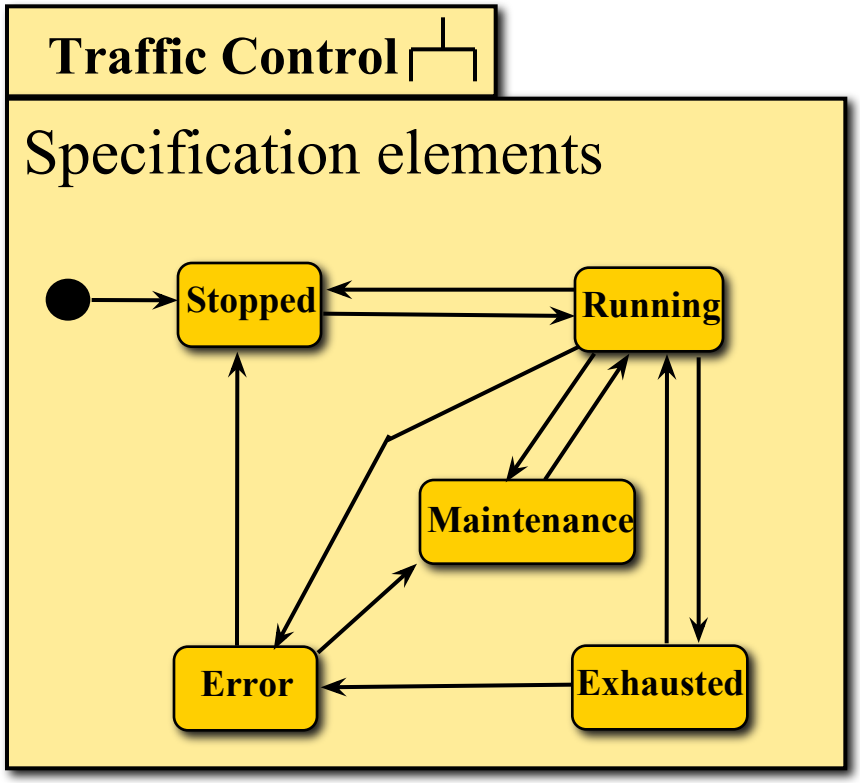


- For subsystem services used in certain sequences
- When the specification is to be understood by non-technical people

Use Case Approach – Example

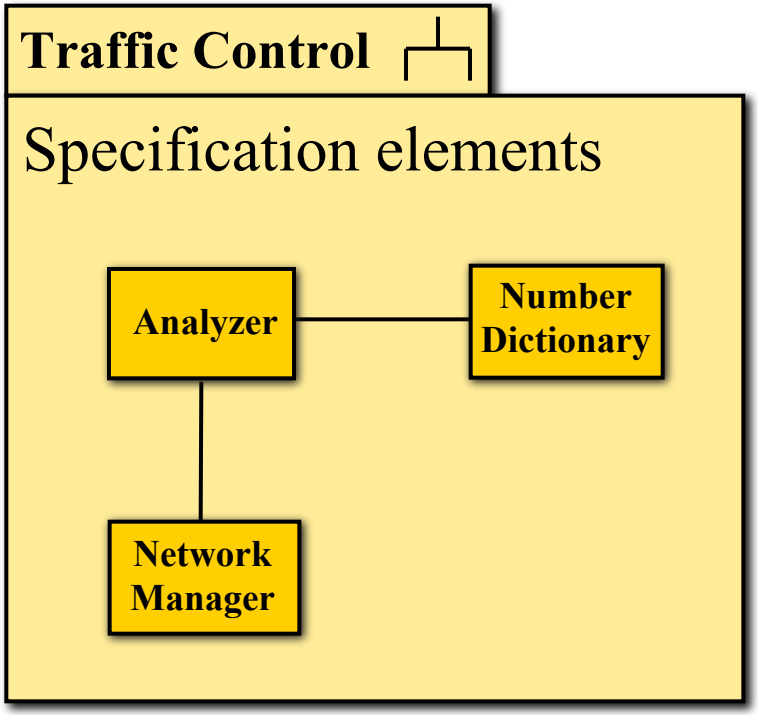


State Machine Approach



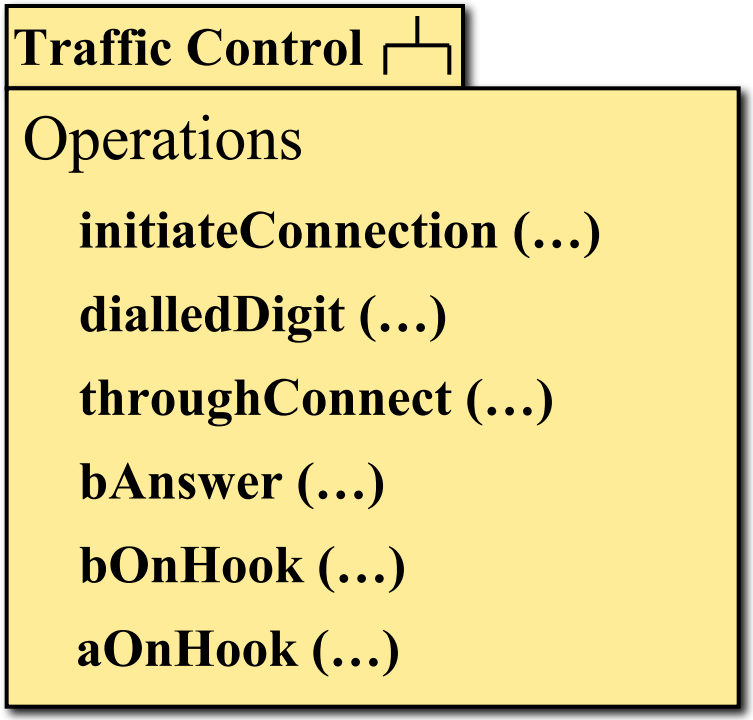
- For subsystems with state dependent behavior
- Focuses on the states of the subsystem and the transitions between them

Logical Class Approach



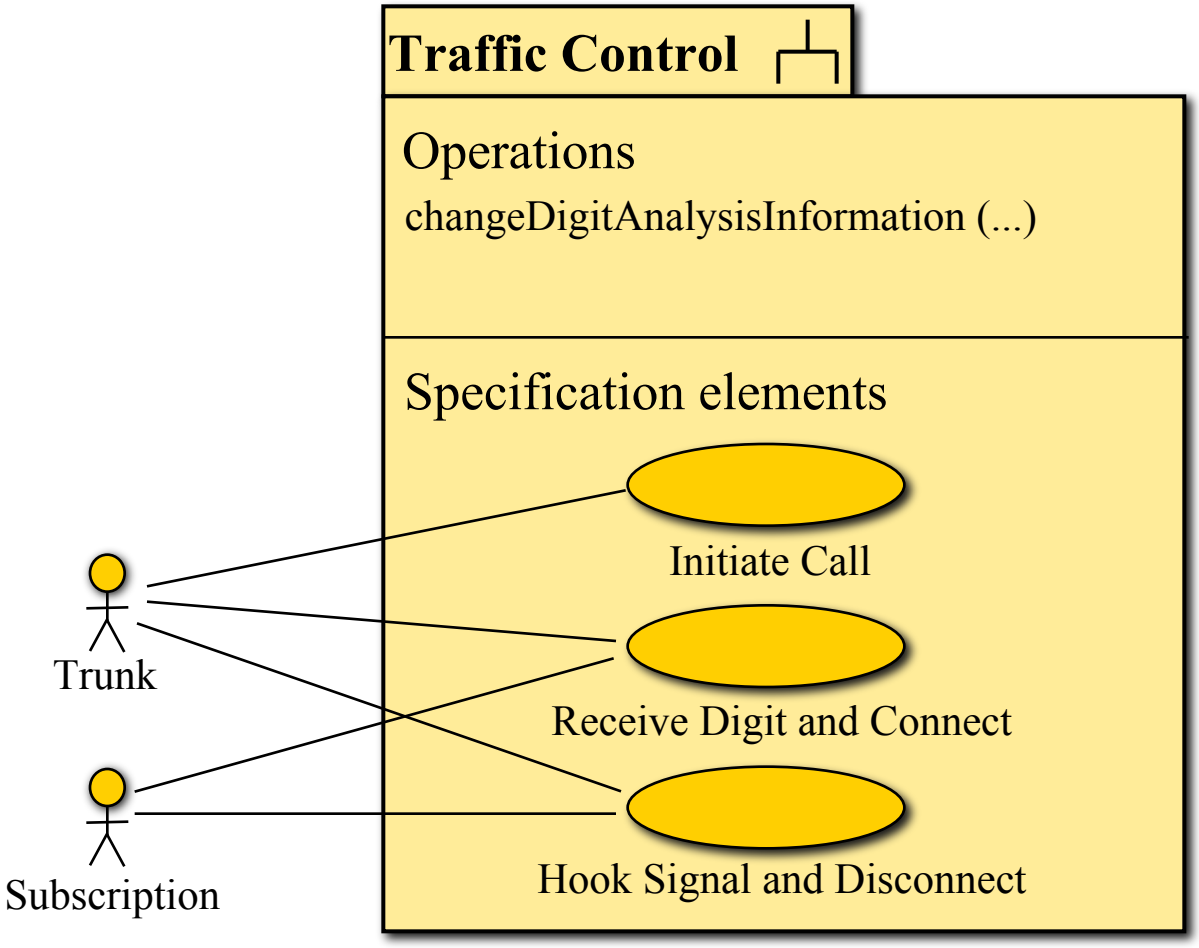
- When usage of the subsystem is perceived as manipulation of objects
- When the requirements are guided by a particular standard

Operation Approach

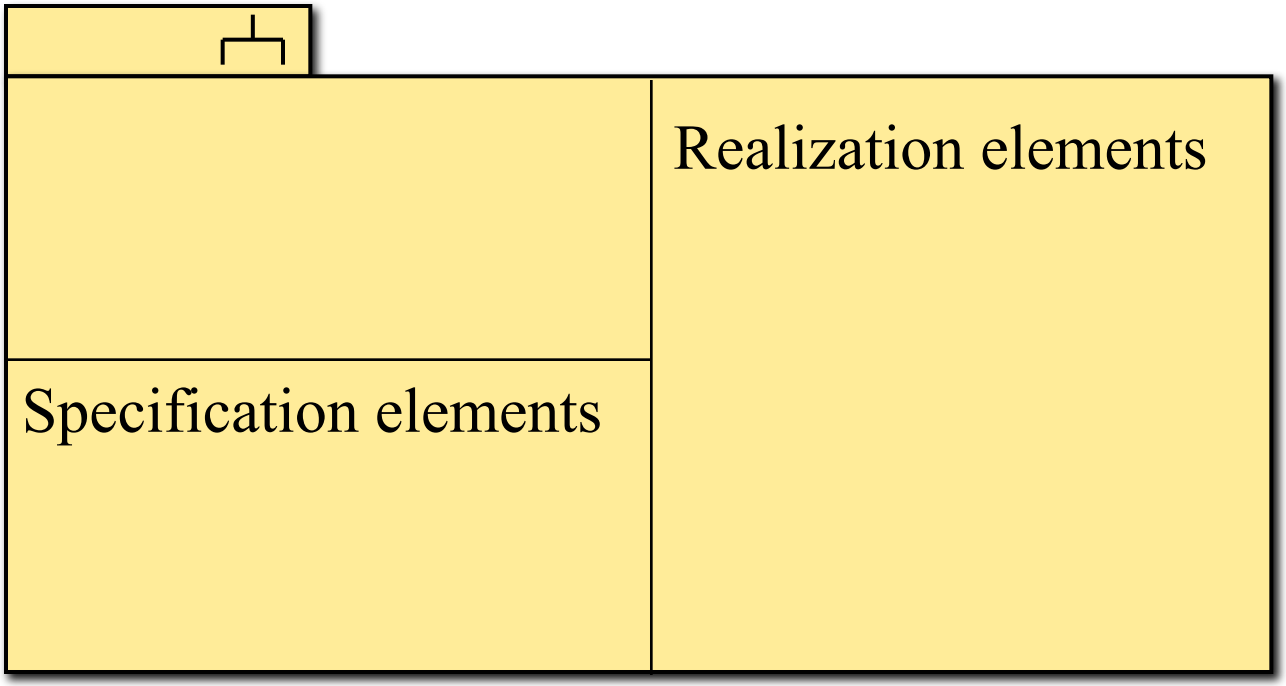


- For subsystems providing simple, "atomic" services
- When the operations are invoked independently

Mixing Techniques

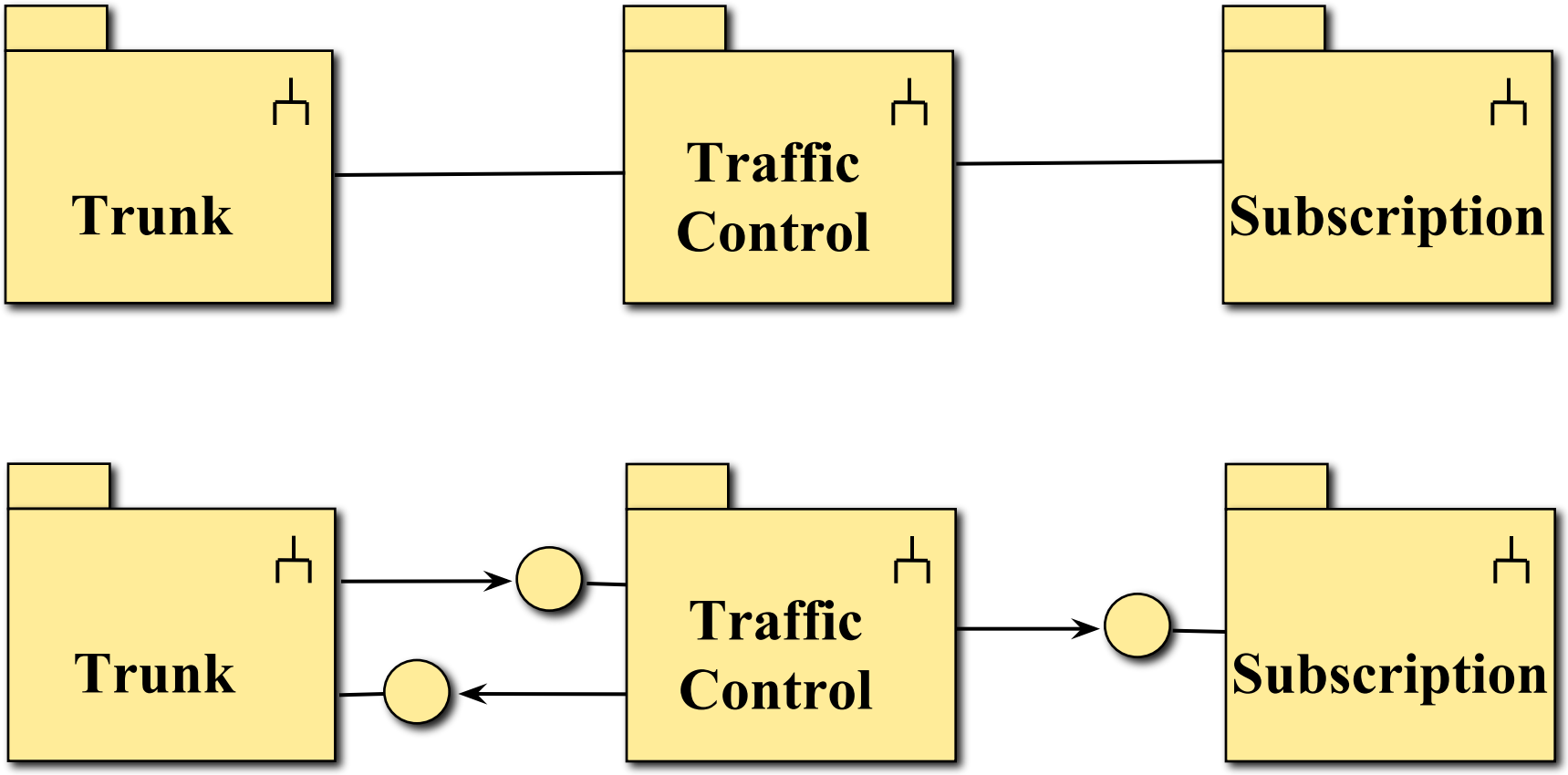


Complete Subsystem Notation

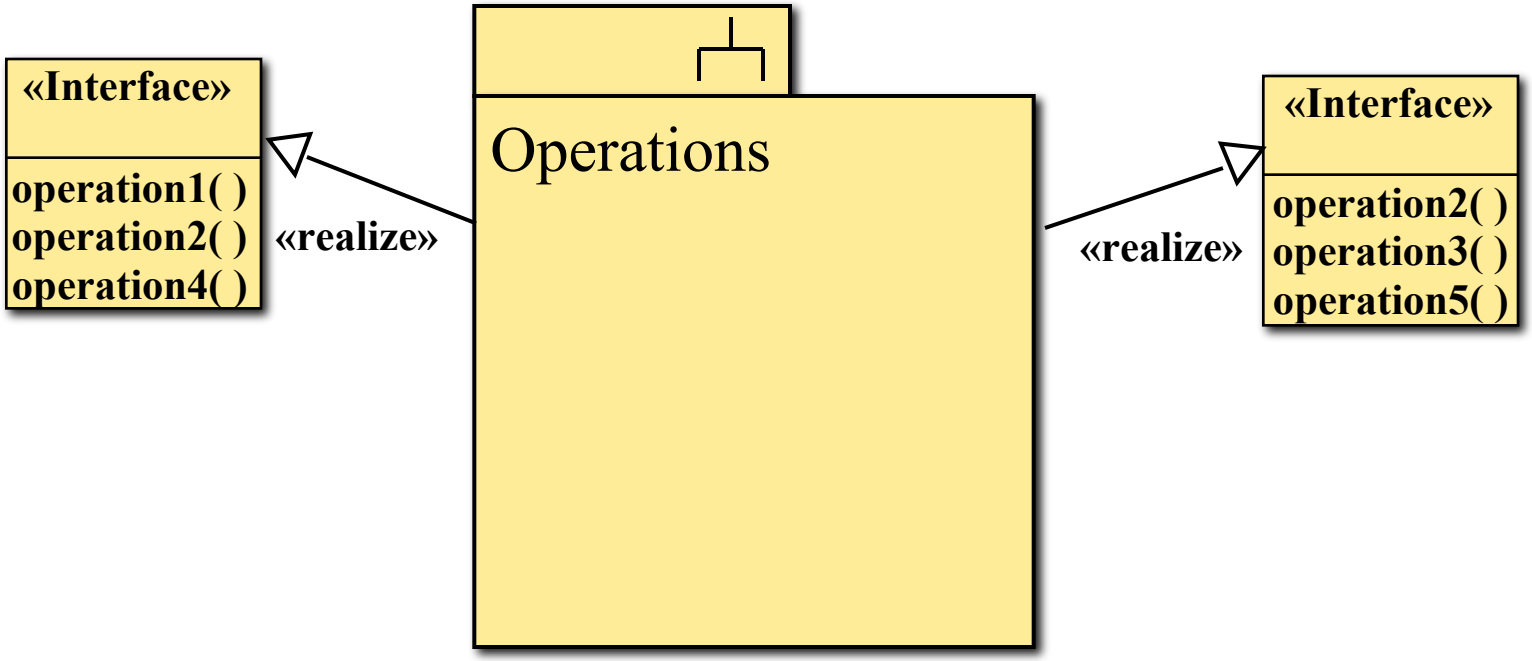


- The complete subsystem symbol has three pre-defined compartments
- Each of the compartments may be optionally omitted from the diagram

Subsystem Interfaces

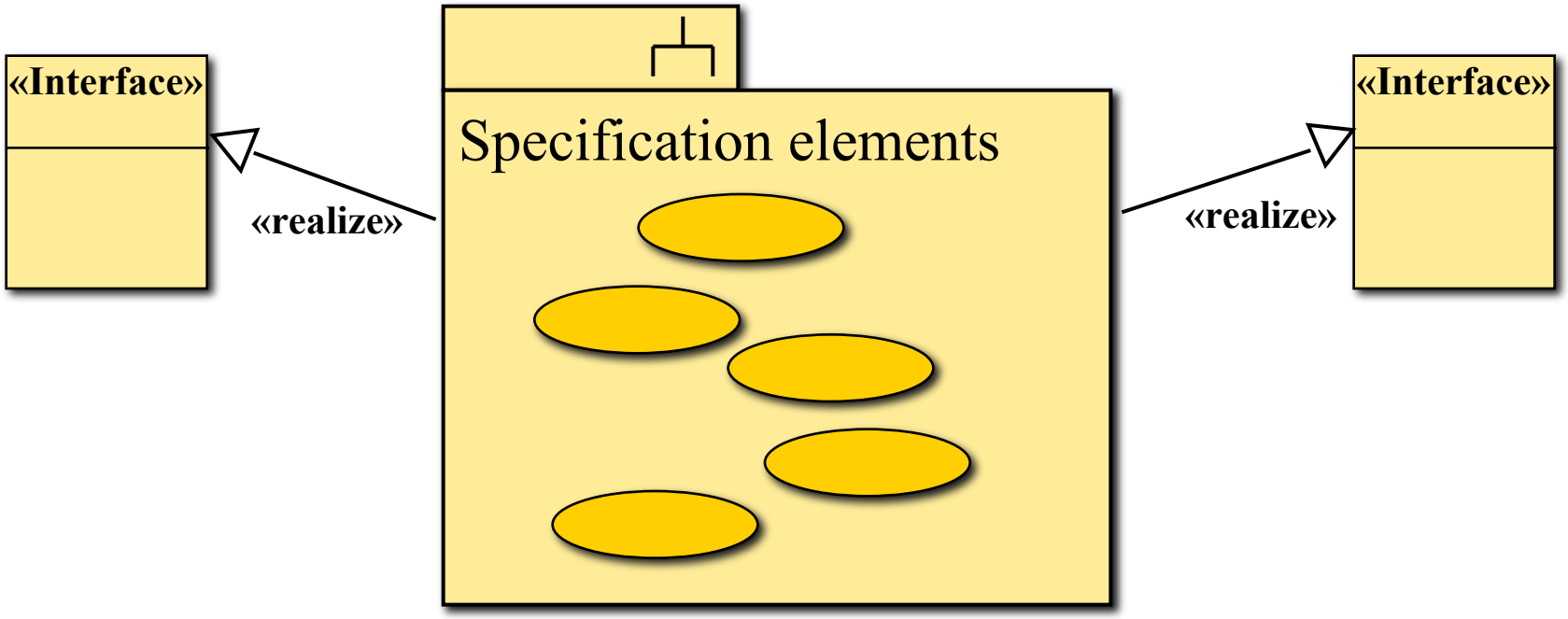


Operations and Interfaces



The subsystem must support all operations in the offered interfaces

Subsystem Interfaces

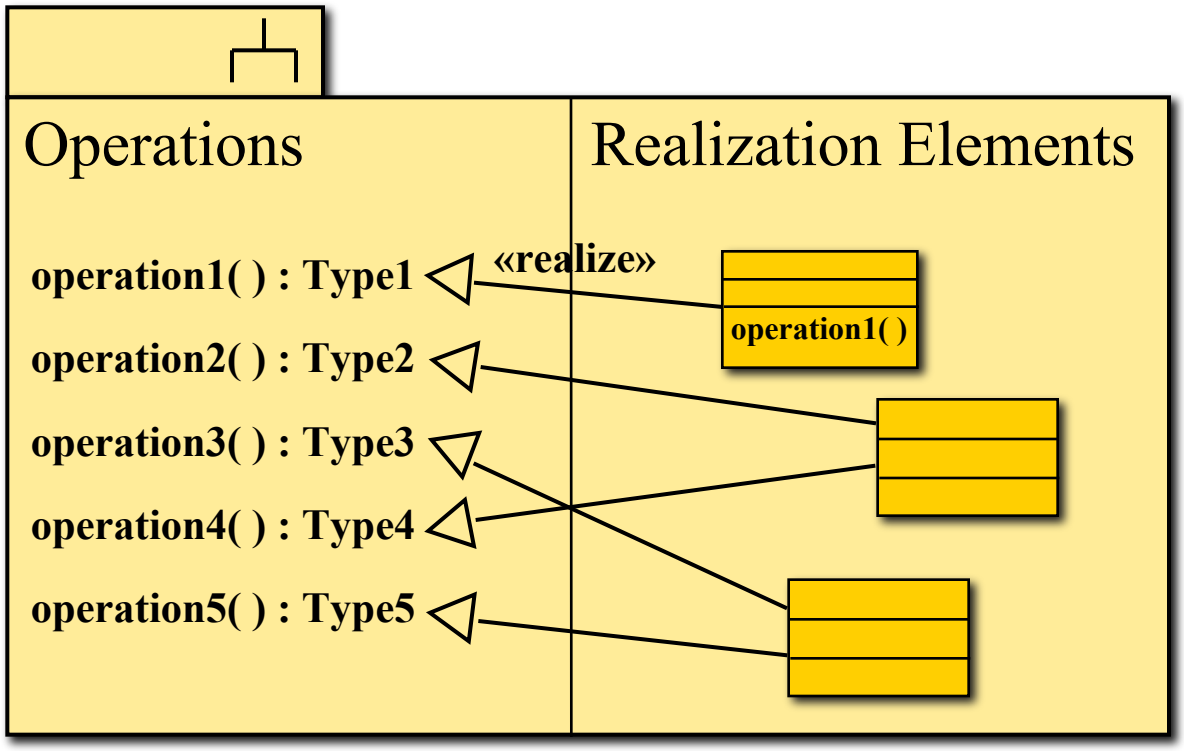


An interface operation may alternatively be supported by a specification element of the subsystem

Specification – Realization

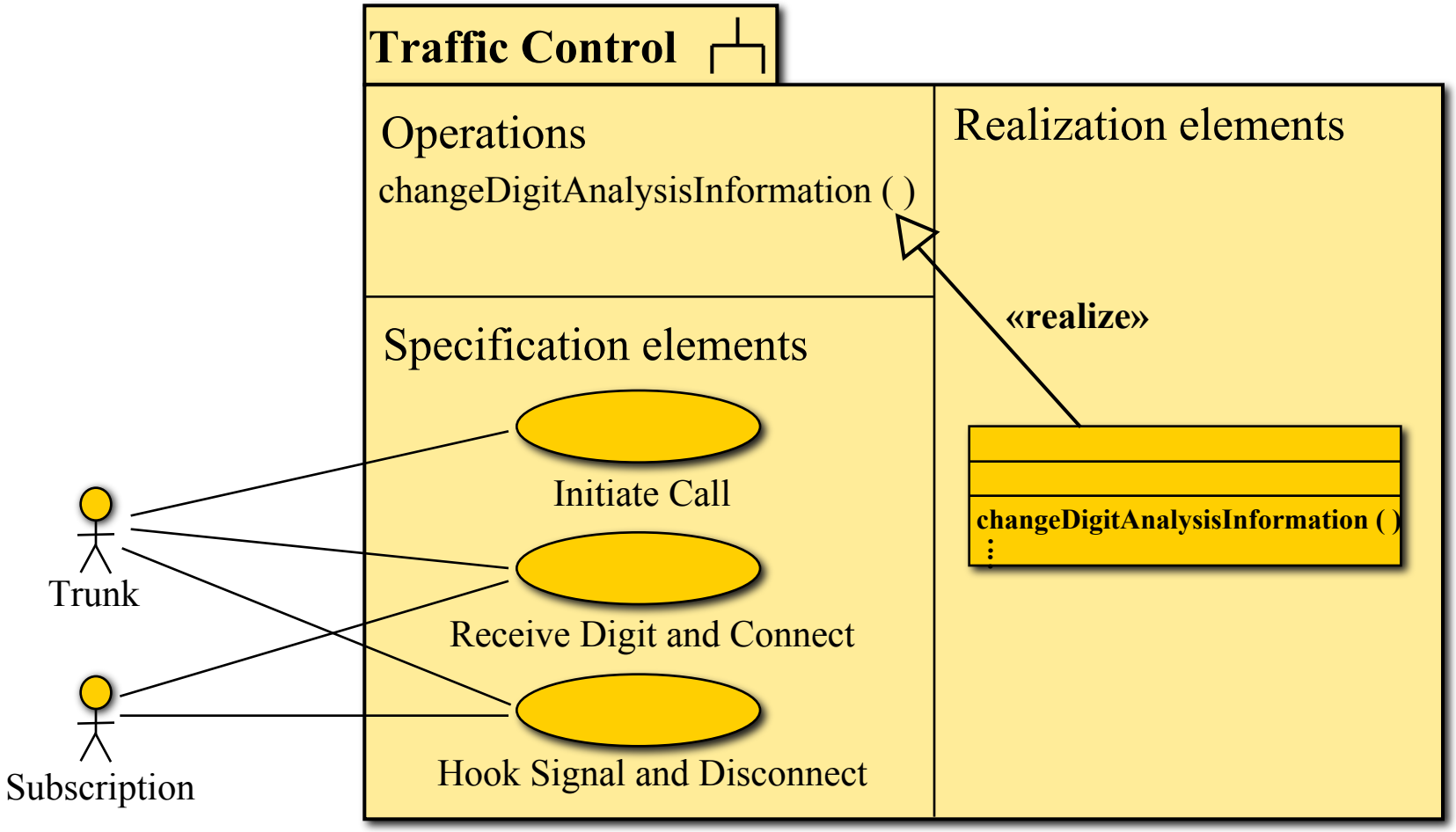
- The specification and the realization must be consistent
- The mapping between the specification and the realization can be expressed by:
  - realization relationships
  - collaborations

Realize Relationship



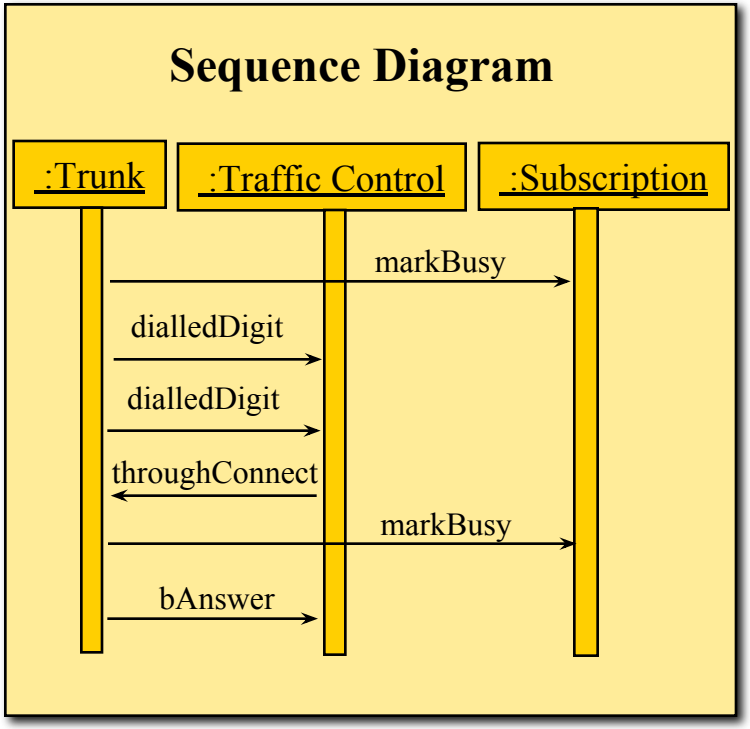
Realization is particularly useful in simple mappings

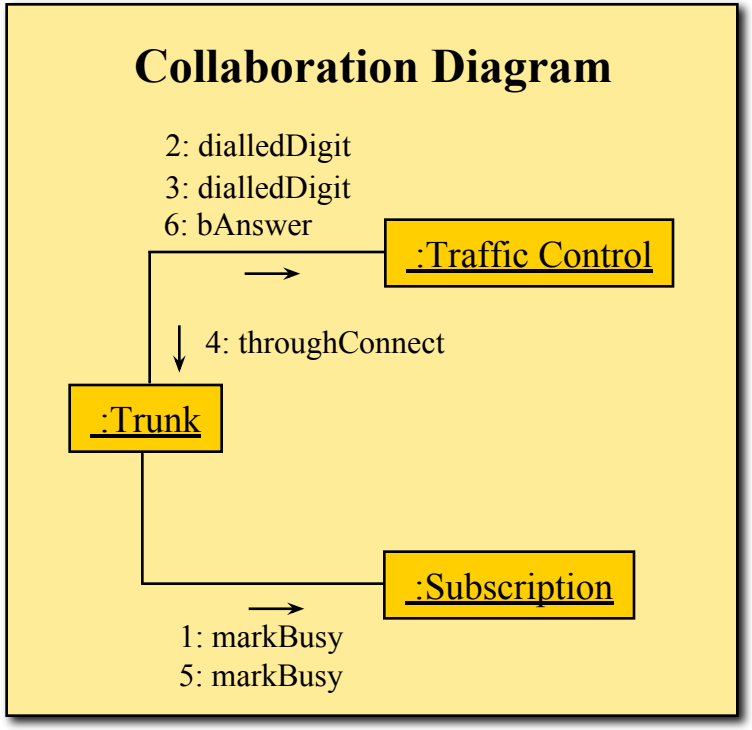
Realize – Example



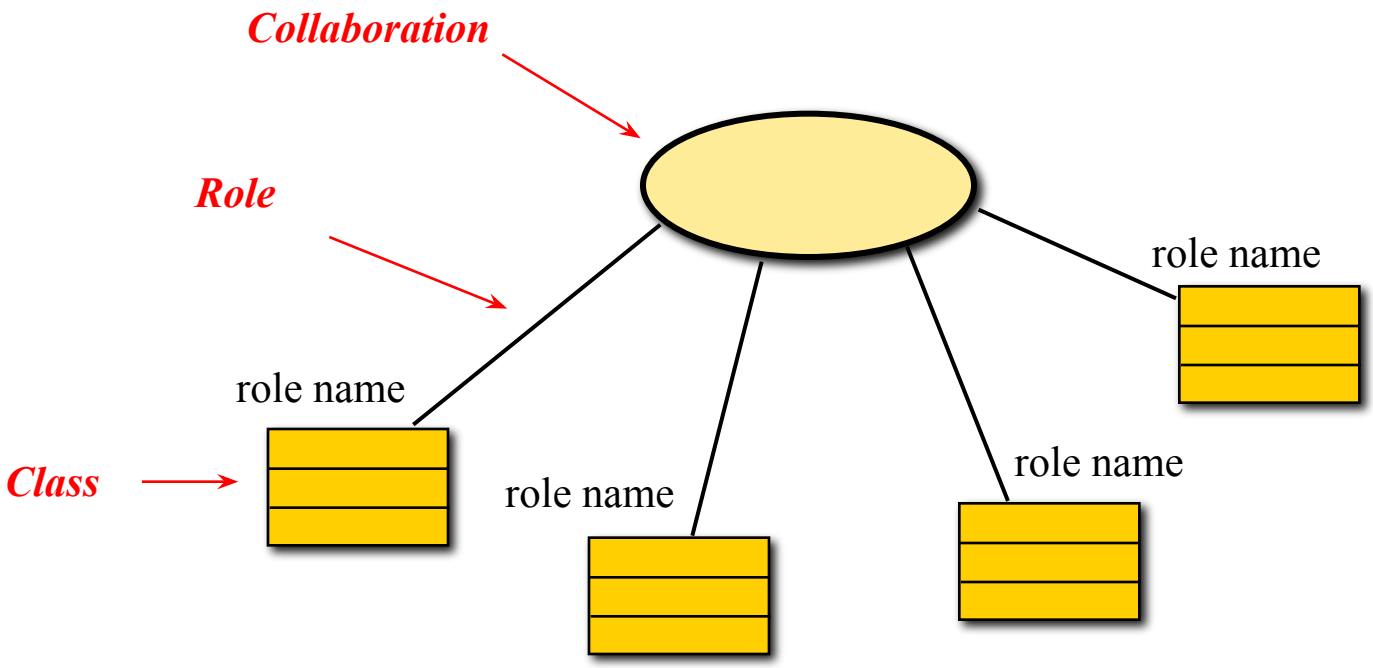
Collaboration

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



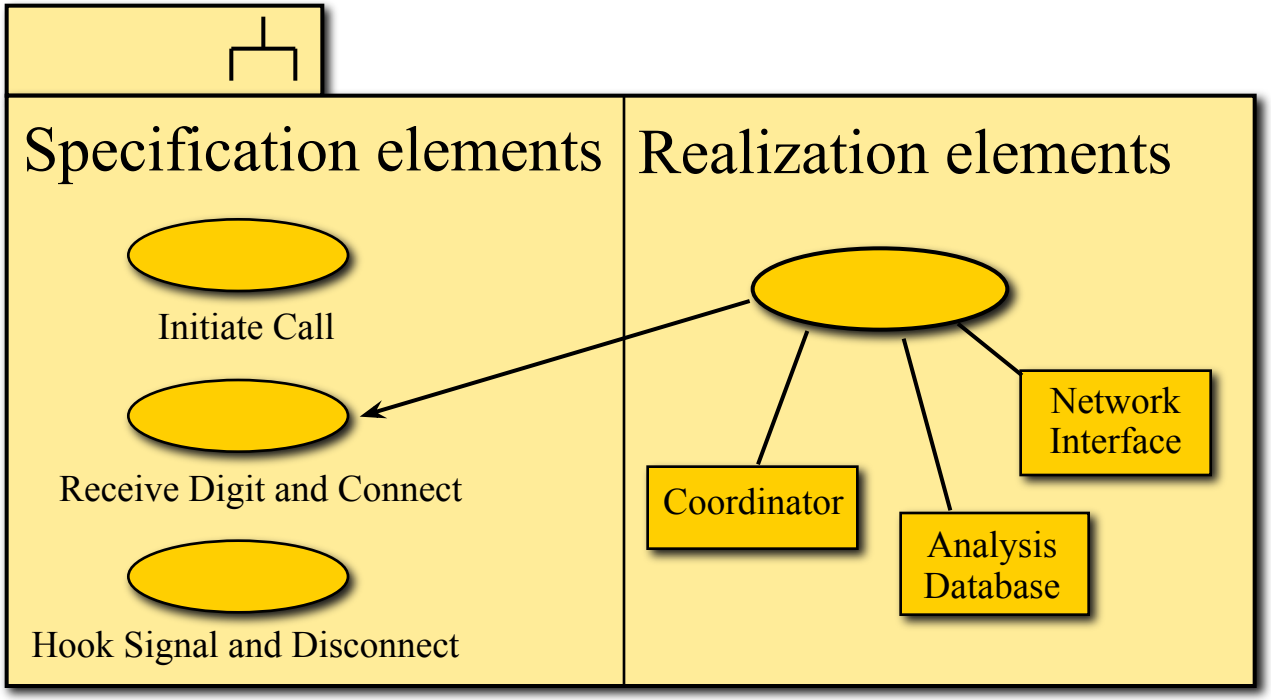


Collaboration – Notation



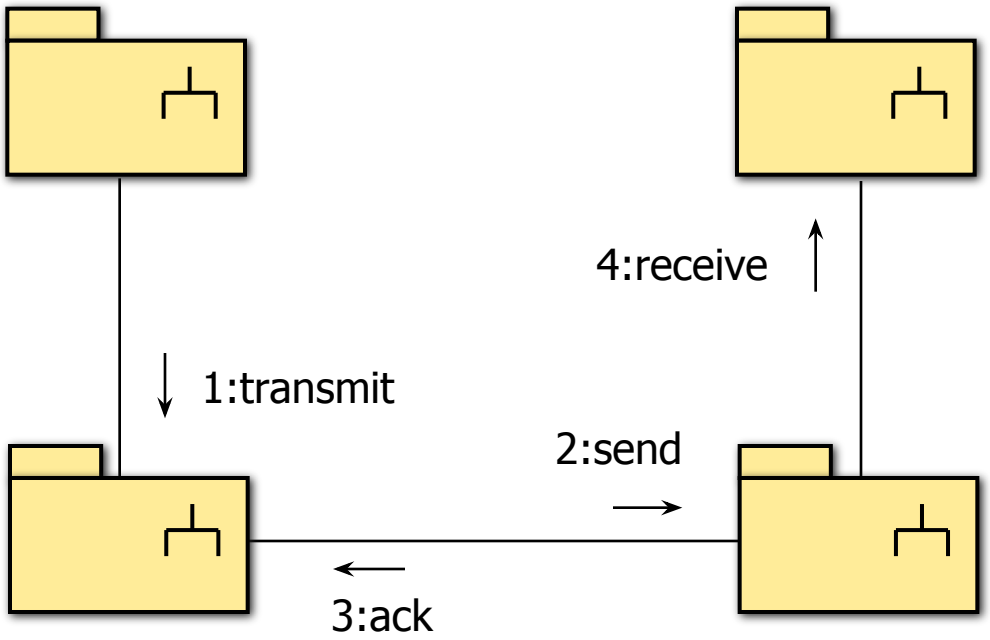
A collaboration and its participants

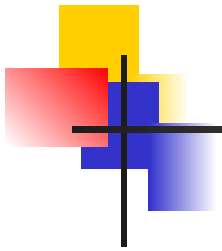
Collaboration – Example



Collaborations are useful in more complex situations

Subsystem Interaction

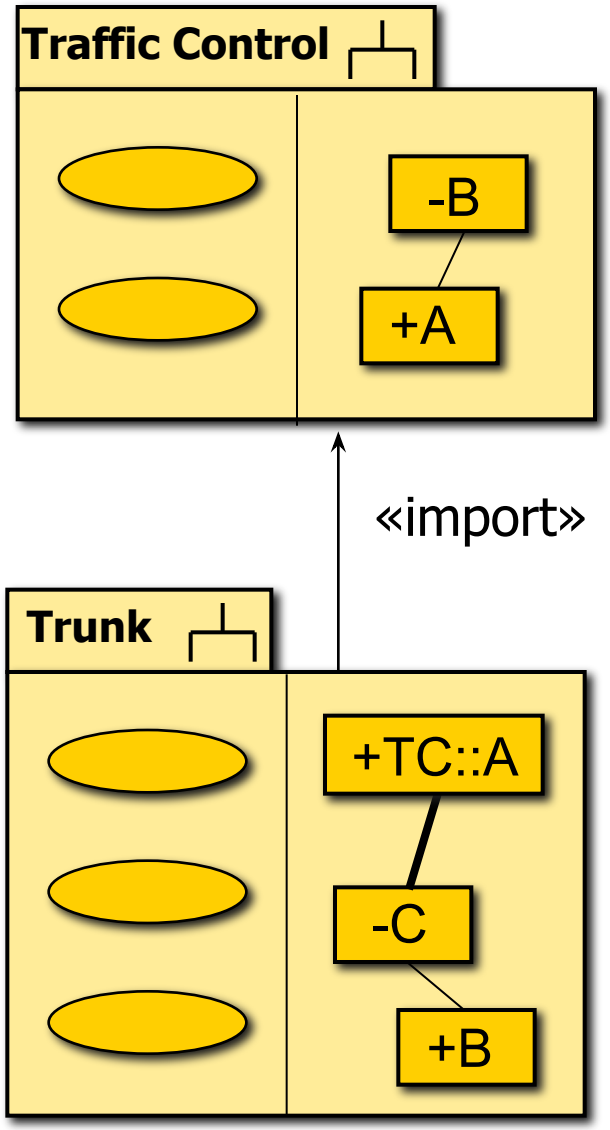




Communicating with Subsystems

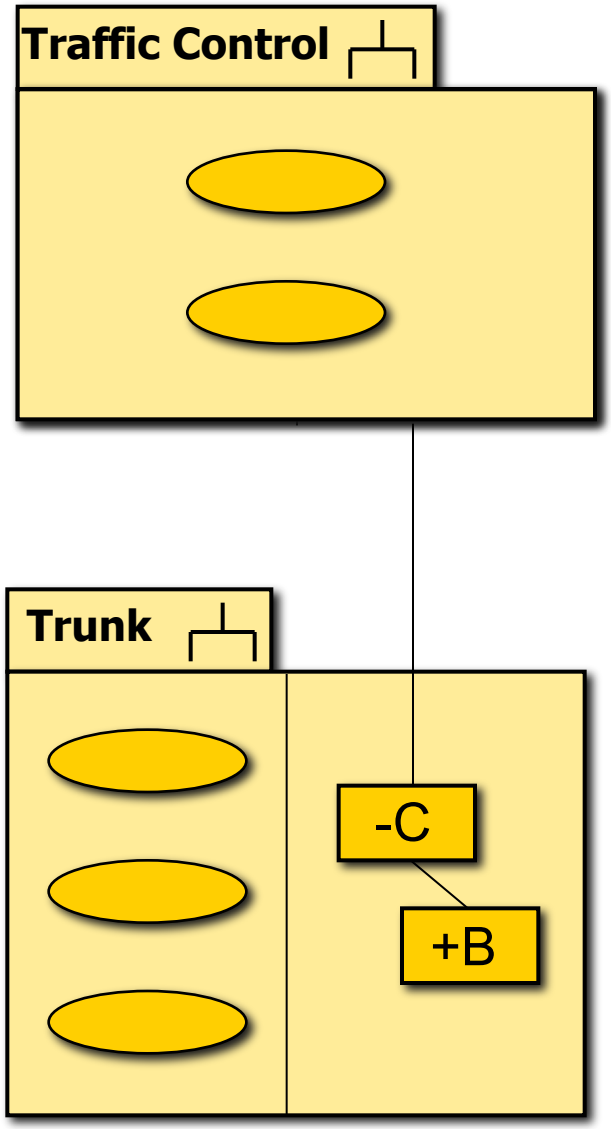
- Two approaches:
  - Open subsystem - public elements are accessed directly
  - Closed subsystem - access via the subsystem itself

Open Subsystems



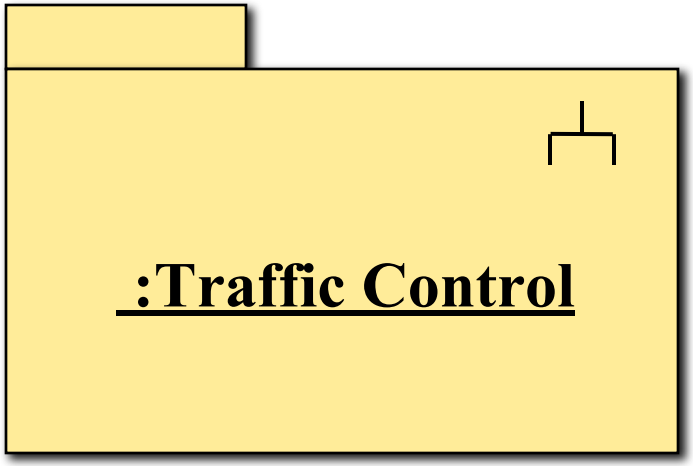
- A subsystem is called *open* if its realization is used directly by the environment
- The specification acts
  - as an overview of the subsystem
  - as a requirements specification

Closed Subsystems



- A subsystem is called *closed* if its realization is not directly used by the environment
- The specification acts
  - as an overview of the subsystem
  - as a requirements specification
  - as a specification of how to use the subsystem

Subsystem Instance



A subsystem instance is the runtime representation of a subsystem. Its task is to handle incoming stimuli.

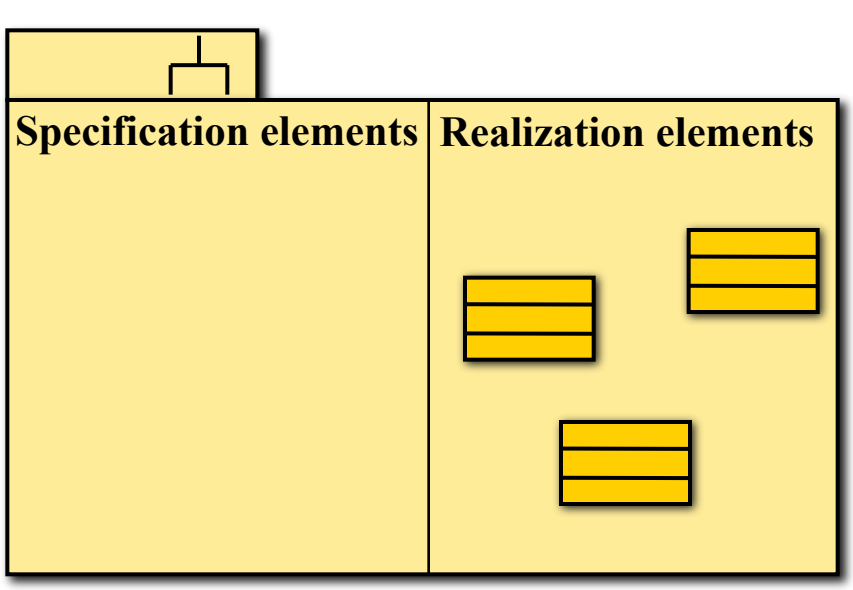
Subsystem Inheritance

- A subsystem with a generalization to another subsystem inherits public and protected elements that are
  - owned or
  - imported

  by the inherited subsystem
- Both specification elements and realization elements are inherited
- Operations are also inherited

Diagram Tour

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



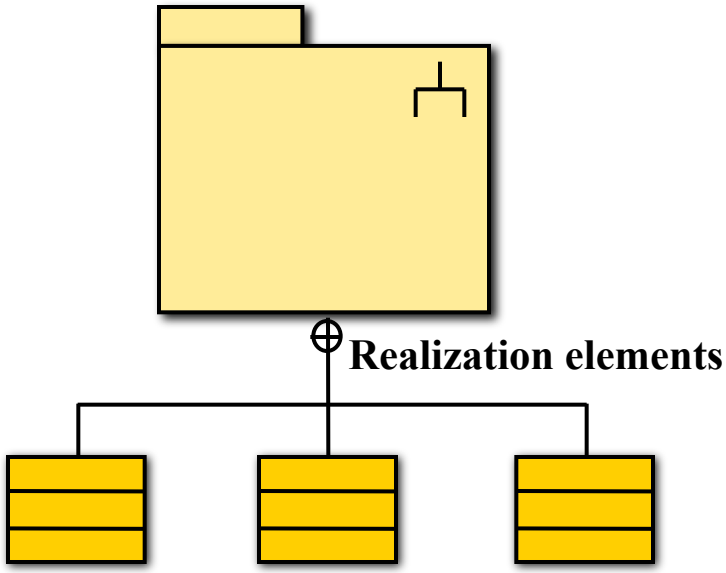
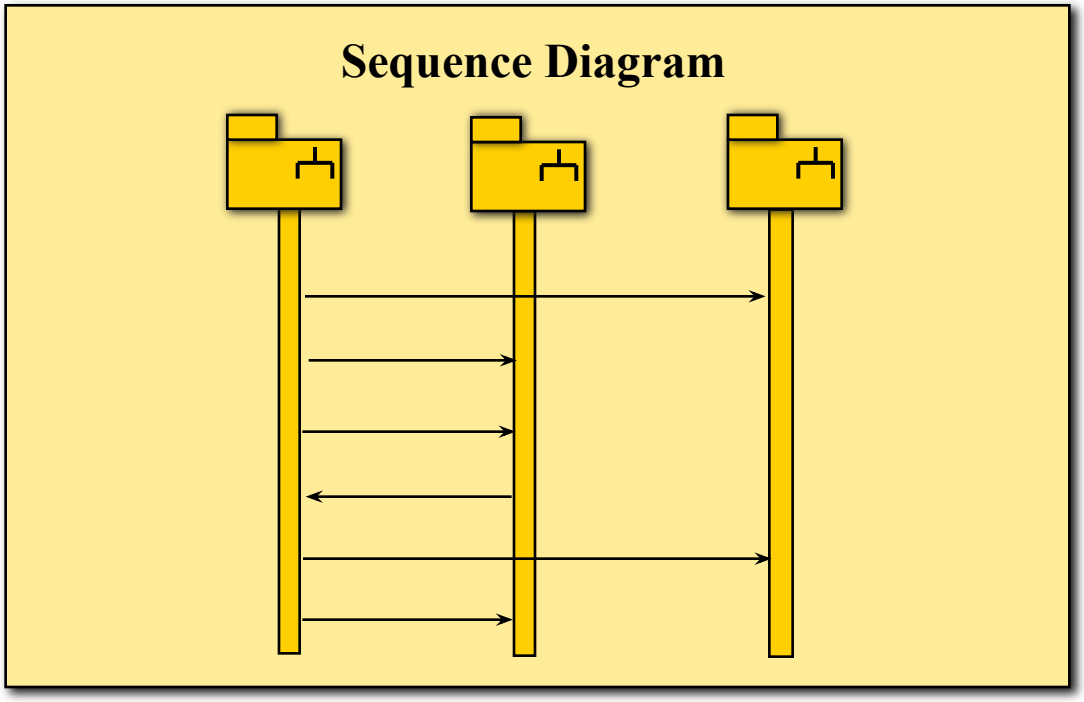


Diagram Tour – continued

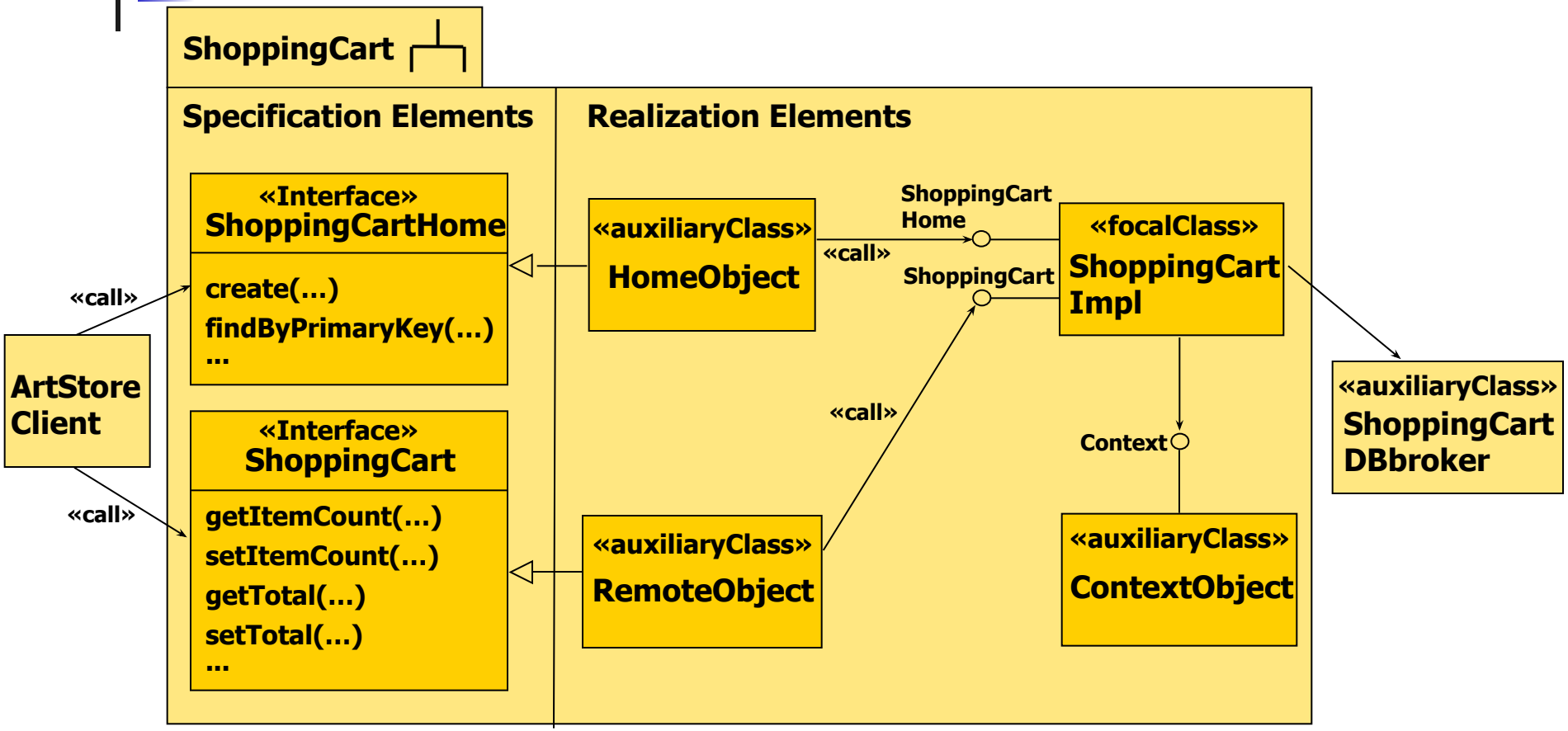
- Subsystems can be shown in interaction diagrams
  - collaboration diagrams
  - sequence diagrams



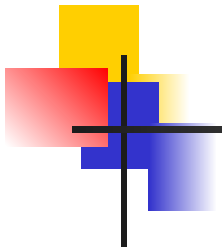
When to Use Subsystems

- To express how a large system is decomposed into smaller parts
- Distributed development
- To express how a set of modules are composed into a large system
- For component based development

Component Based Development

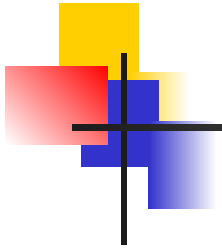


An EJB component modeled using a subsystem and classes stereotyped «focalClass» or «auxiliaryClass»



Modeling Tips – Subsystem

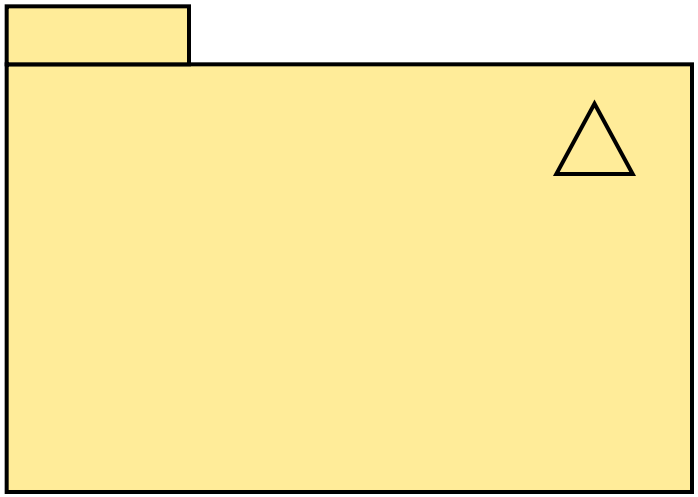
- Define a subsystem for each separate part of a large system
- Choose specification technique depending on factors like kind of system and kind of subsystem
- Realize each subsystem independently, using the specification as a requirements specification



Model

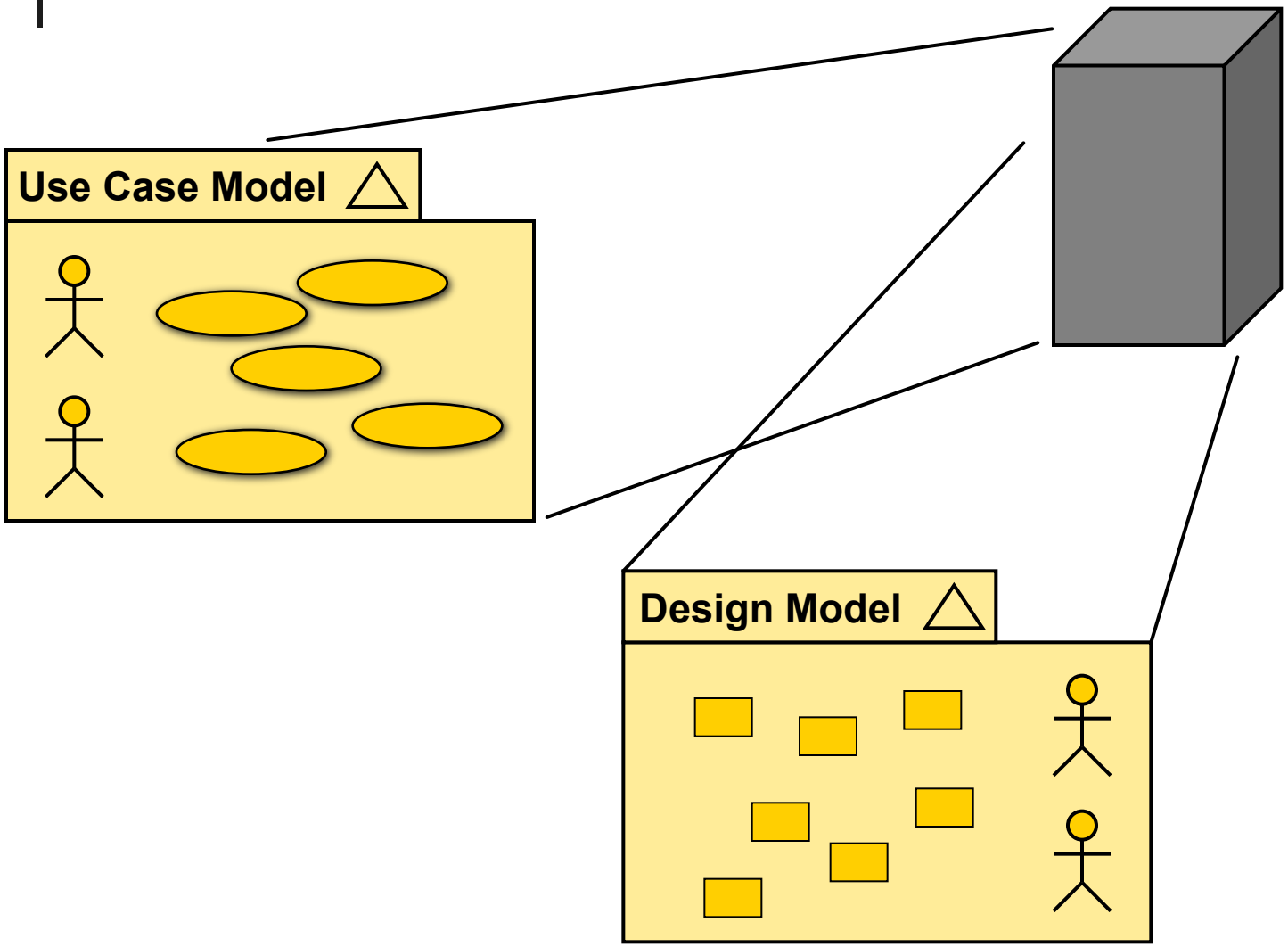
- What are Models?
- Core Concepts
- Diagram Tour
- When to Use Models
- Modeling Tips

Model

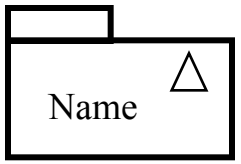
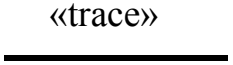


A model captures a view of a system, with a certain purpose. It is a complete description of those aspects of the system relevant to the purpose of the model, at the appropriate level of detail.

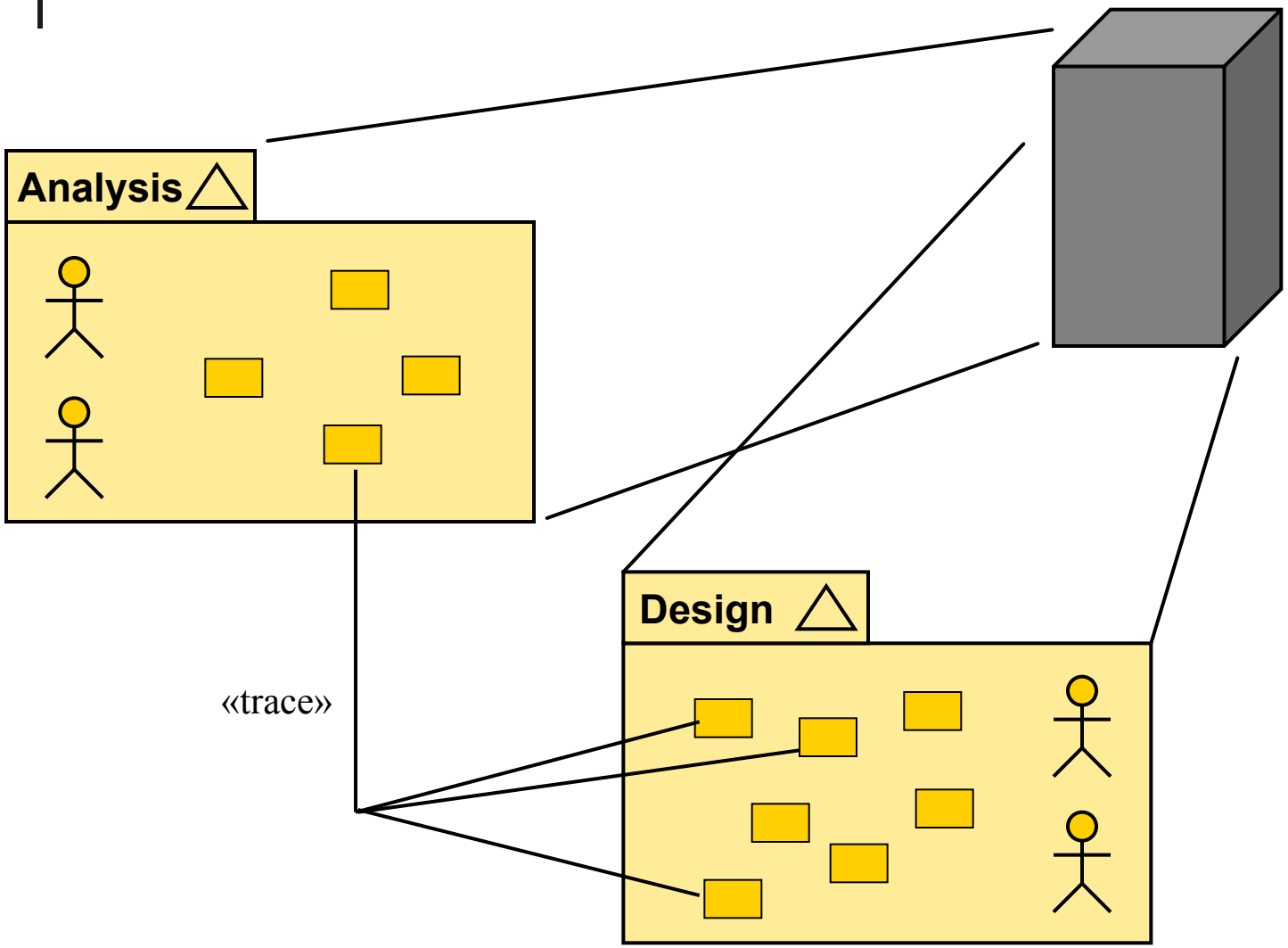
Model – Example



Core Concepts

Construct	Description	Syntax
Model	A view of a system, with a certain purpose determining what aspects of the system are described and at what level of detail.	
Trace	A dependency connecting model elements that represent the same concept within different models. Traces are usually non-directed.	

Trace



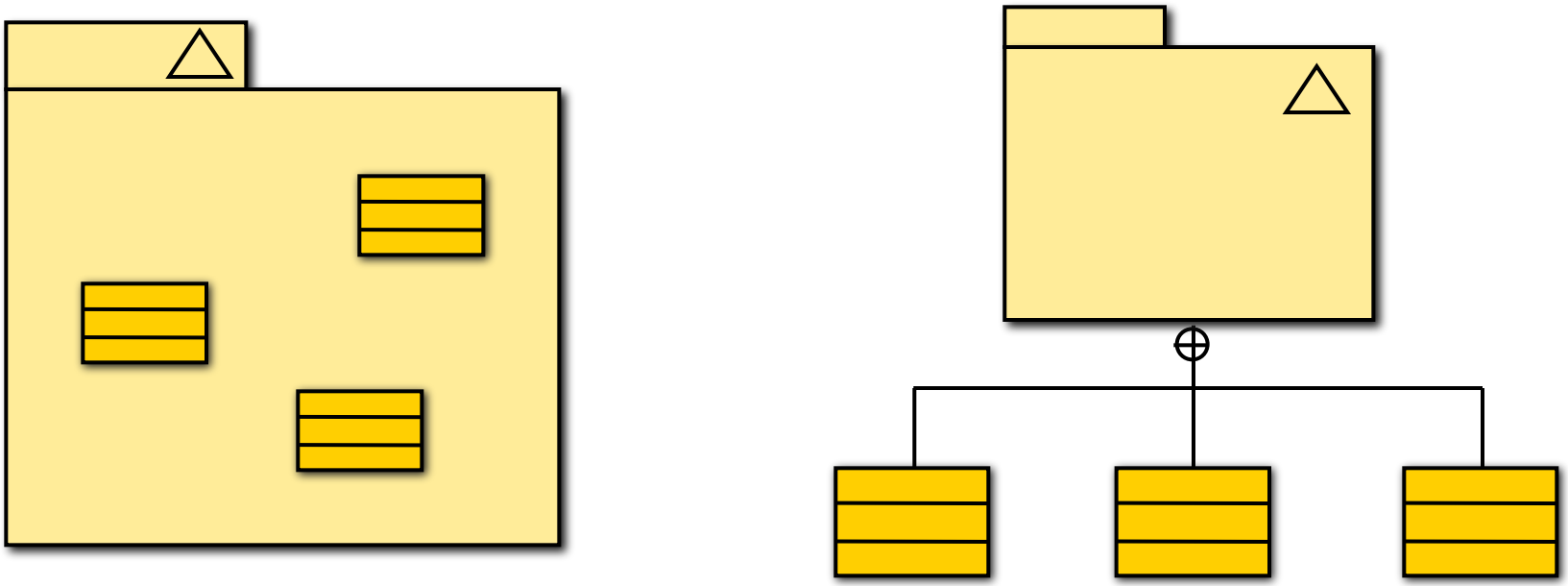
Model Inheritance

- A model with a generalization to another model inherits public and protected elements that are
  - owned or
  - imported

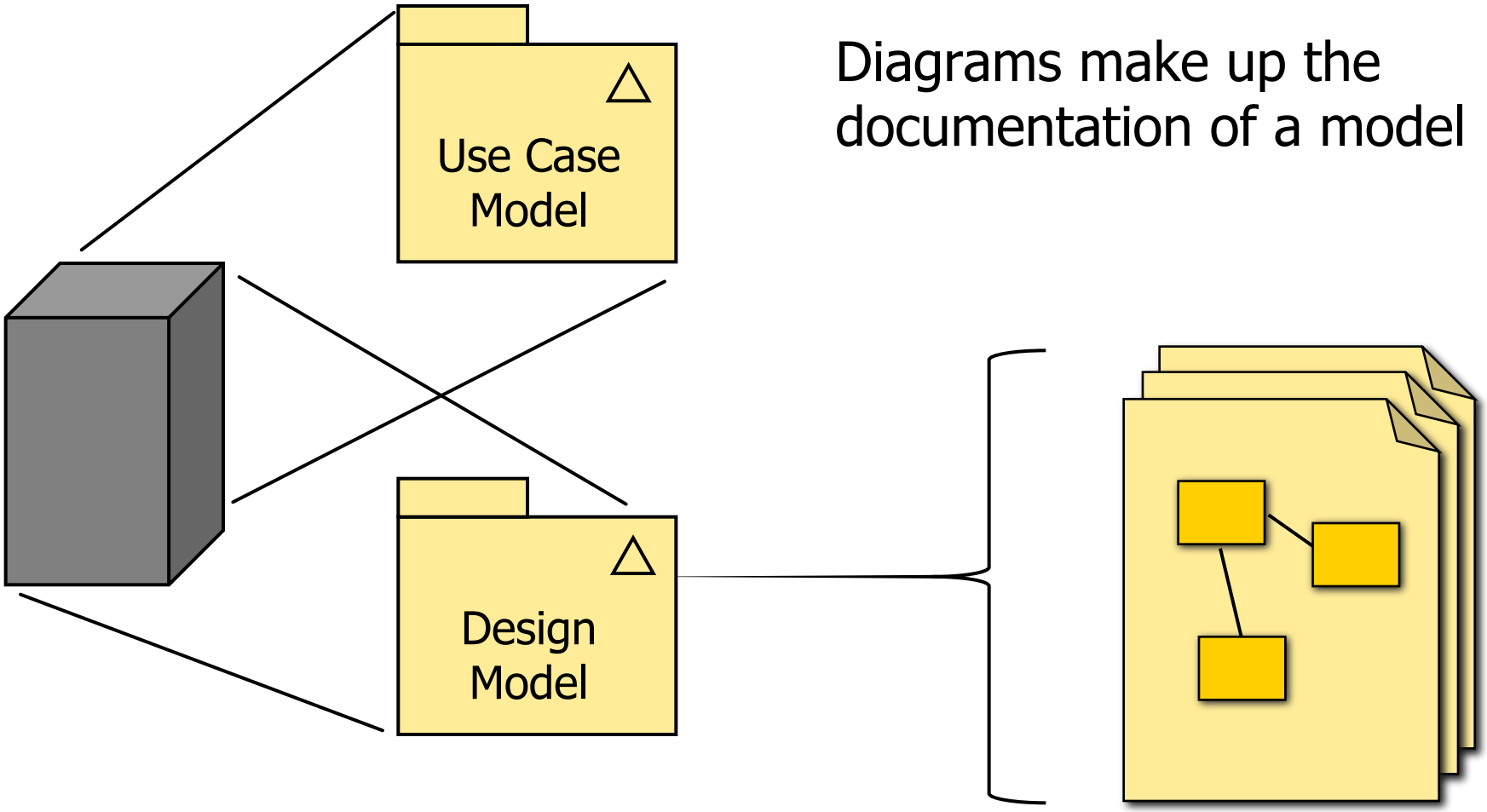
  by the inherited model

Diagram Tour

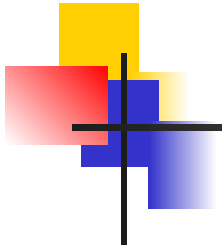
- Models as such are seldom shown in diagrams
- Two equivalent ways to show containment:



Model vs. Diagram



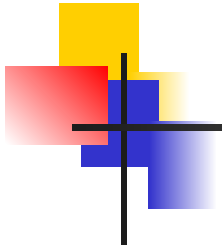
Diagrams make up the documentation of a model



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

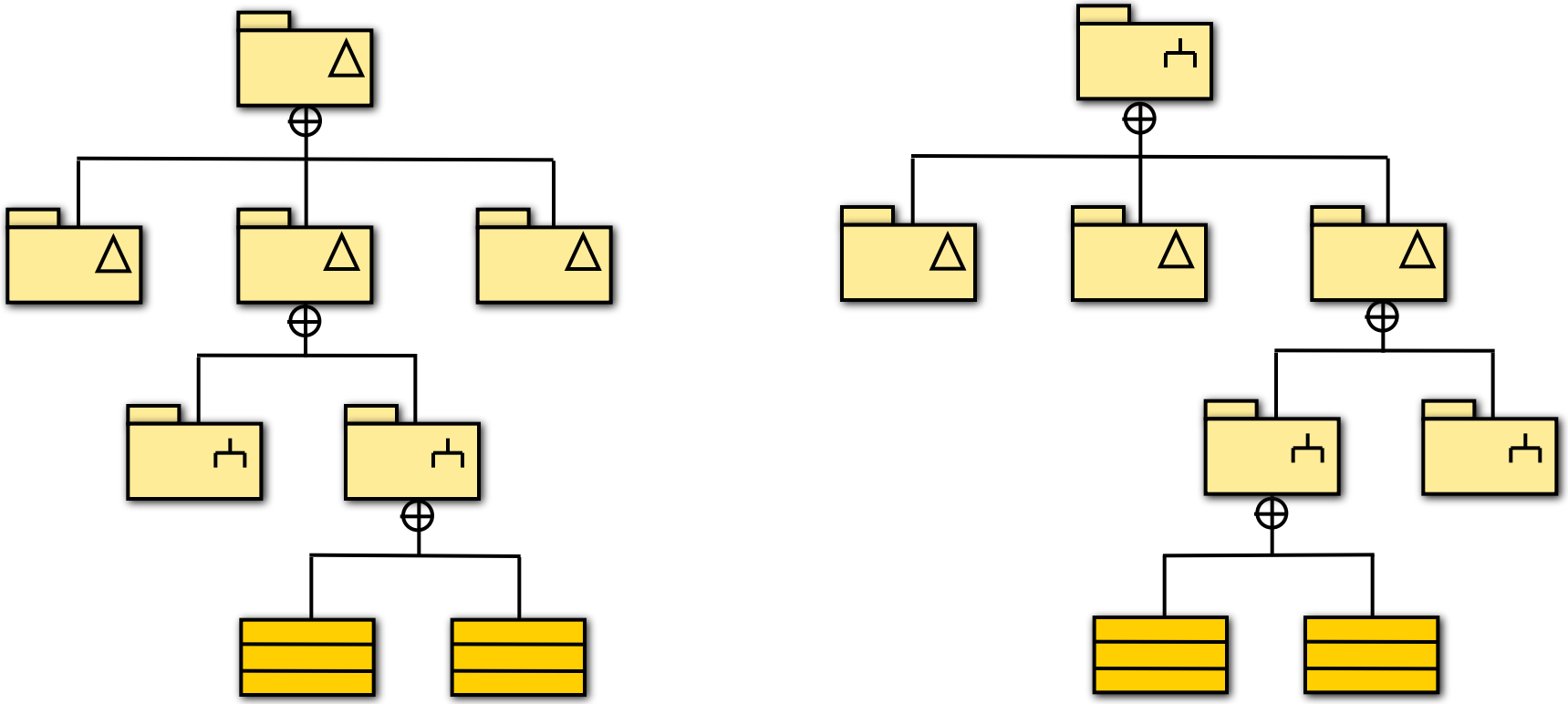
Modeling Tips – Model



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

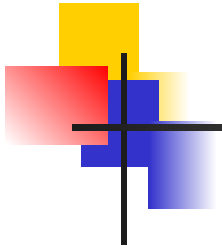
Models and Subsystems

Models and subsystems can be combined in hierarchies:



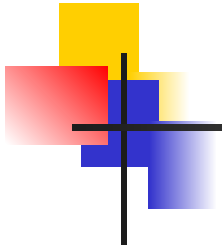
Wrap Up Model Management

- Packages are used to organize a large set of model elements
  - Visibility
  - Import
  - Access
- Subsystems are used to represent parts of a system
  - Specification
  - Realization
- Models are used to capture different views of a system
  - Trace



Advanced Modeling with UML

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

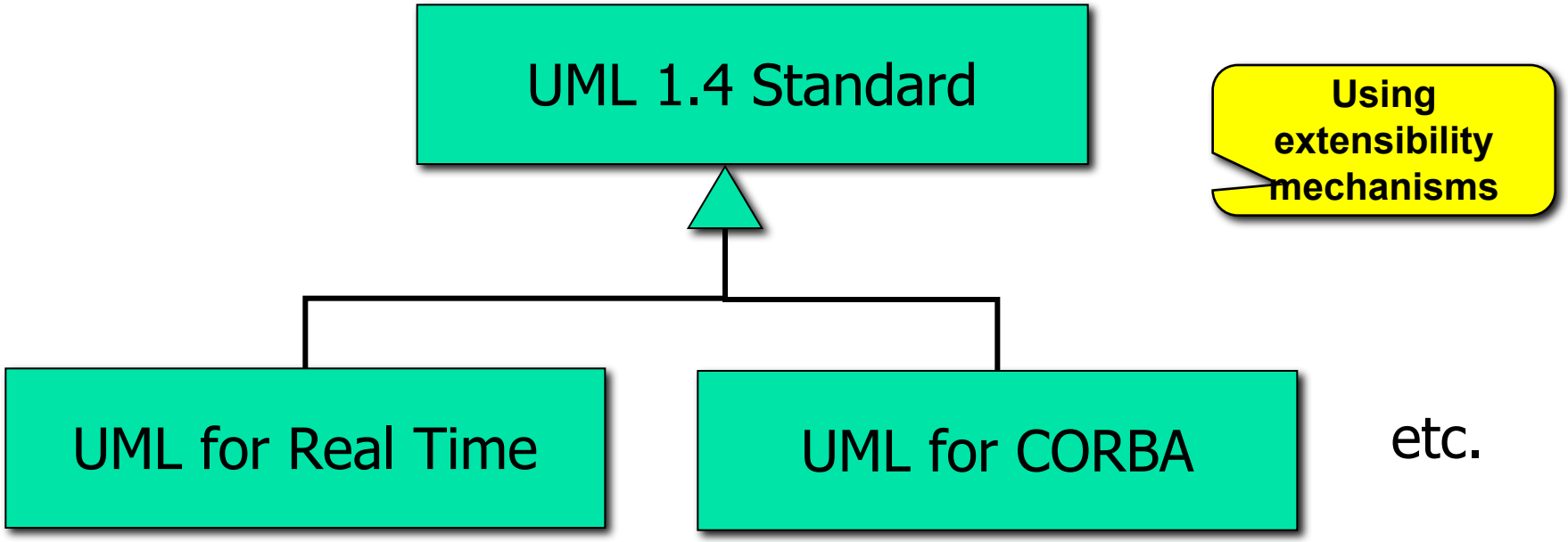


Semantic Variations in UML

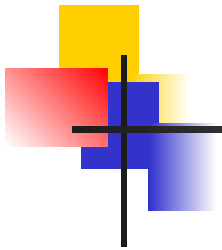
- UML contains semantic aspects that are:
  - undefined (e.g., scheduling discipline), or
  - ambiguous (multiple choices/interpretations)
- Why is this the case?
  - UML can't address every domain perfectly
  - Different domains require different specializations
- But, if UML can be extended in arbitrary ways, what is the value of having a standard?

UML as a "Family of Languages"

- The standard can be specialized for different domains
  - in effect: refinements of the standard



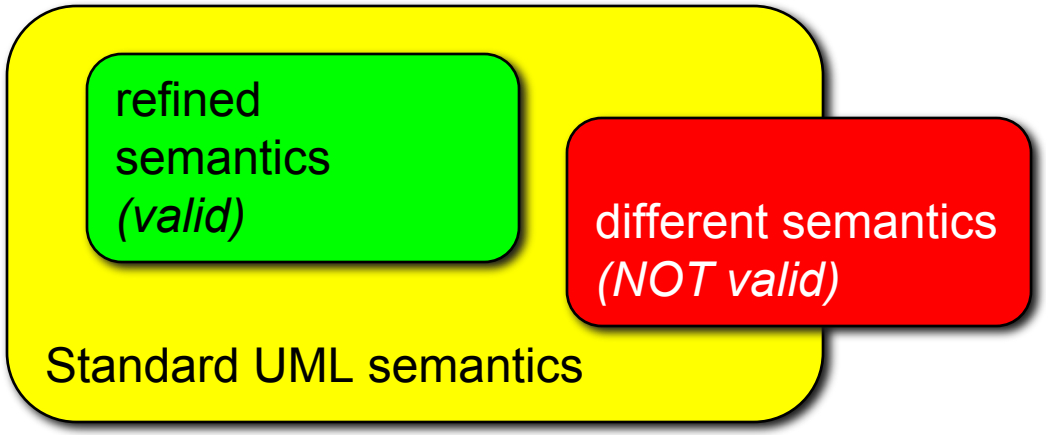
Extensibility Mechanisms



- Extensibility Mechanisms
  - Allow Modelers to *refine* the semantics of UML for a specific domain
- Extensions *cannot* violate the standard UML semantics
  - Enforces a consistent core of concepts and semantics for every variation
  - Prevents meta-model explosion (Using UML to model everything and anything)

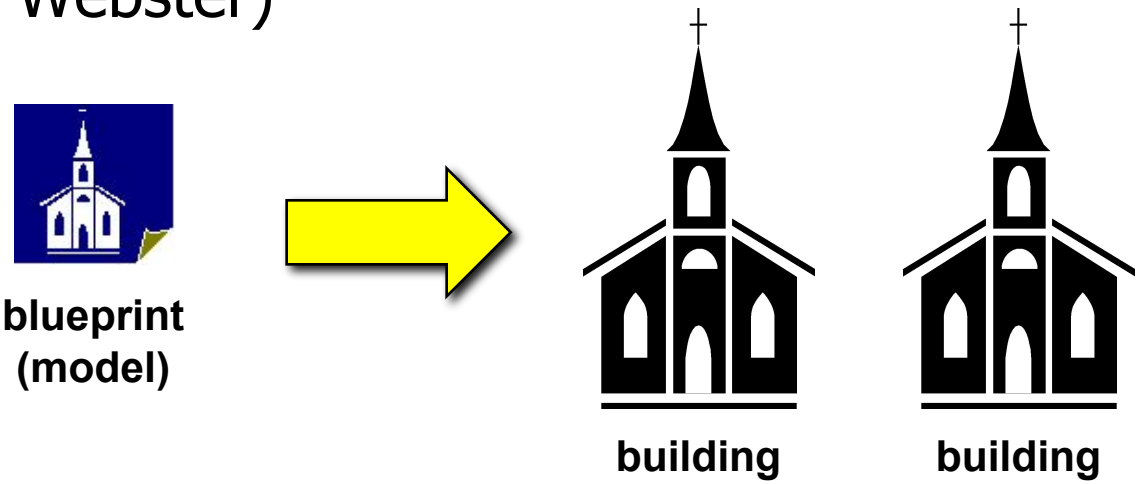
How UML Extensibility Works

- The standard UML semantics can be viewed as defining a space of possible interpretations



Models

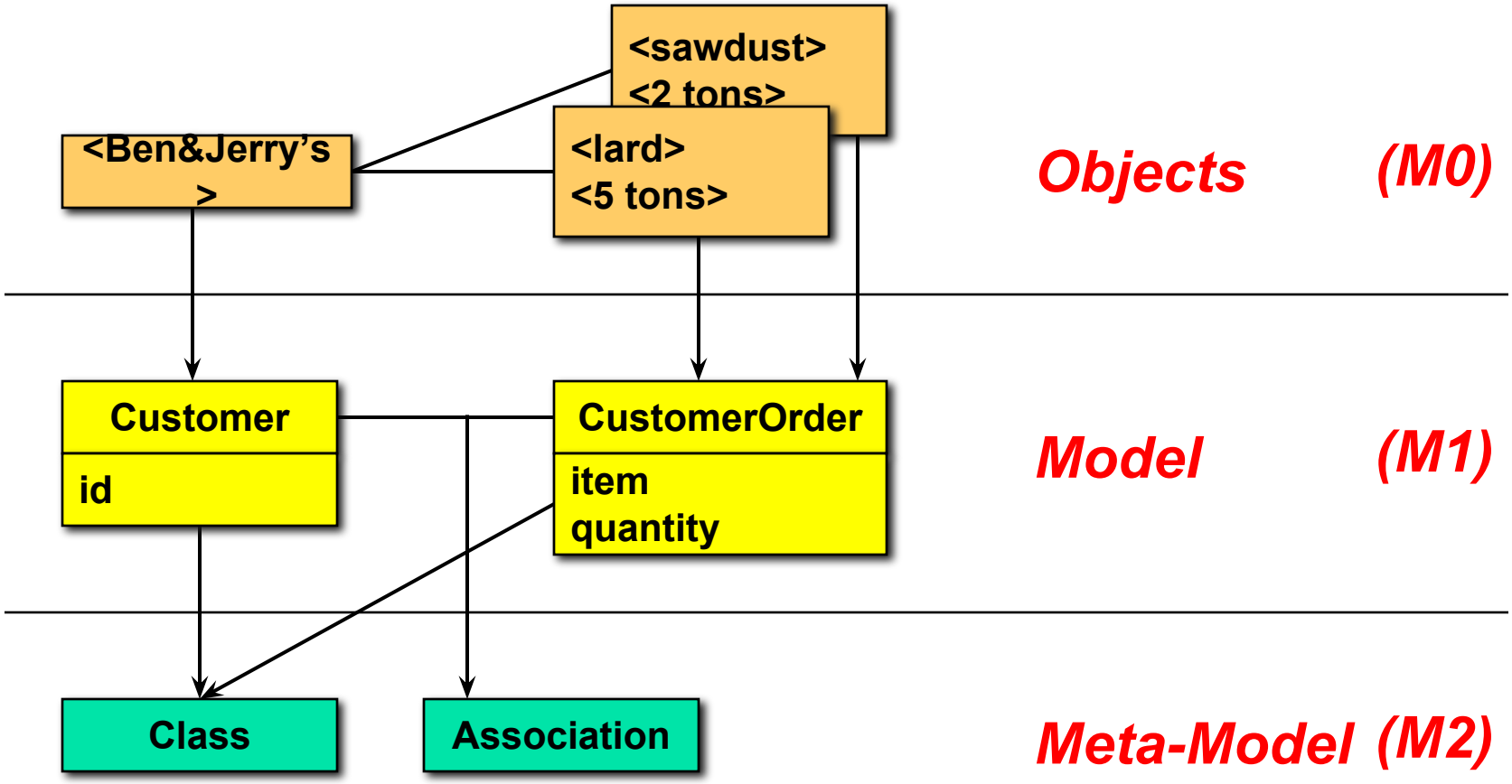
- A *model* is a description of something
  - *"a pattern for something to be made"* (Merriam-Webster)



- model ≠ thing that is modeled
  - The Map is Not The Territory

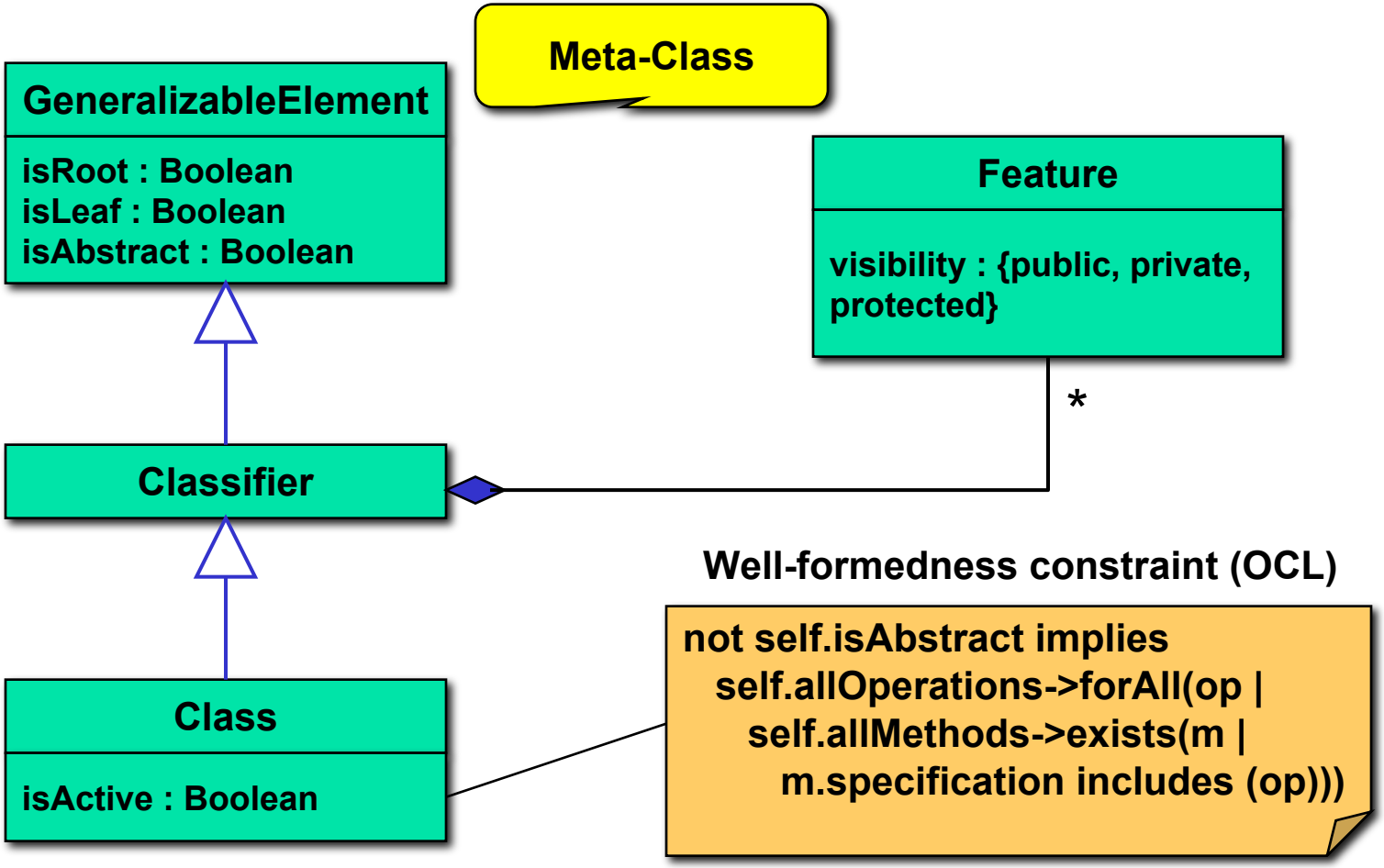
Meta-Models

- Are Simply Models of Models



The UML Meta-Model

- Is a UML Model of UML



The UML Meta-Model

- Put another way, UML uses (a part of) itself as a Meta-language
  - Other examples: Scheme, Lisp, Smalltalk
- This small subset of UML is used to describe all of UML
  - The subset contains classes, associations, operations, constraints, generalization, etc.
  - This core subset is related to MOF (more on MOF in Part 4)

The Basic Extensibility 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

Stereotypes

- Used to define specialized model elements based on a core UML model element
- Defined by:
  - Base metaclasses (or stereotype)
    - What element is specialized?
  - Constraints:
    - What is special about this stereotype?
  - required tags (0..*)
    - What values does this stereotype need to know?
  - icon
    - How should I appear in a model?
- A model element can be stereotyped in multiple different ways

Example

- Capsule: A special type of concurrent object used in modeling certain real-time systems
- By definition, all classes of this type:
  - are active (concurrent)
  - have only features (attributes and operations) with protected visibility
  - have a special "language" characteristic used for code generation purposes
- In essence, a constrained form of the general UML Class concept

Example: Stereotype Definition

- Using a tabular form:

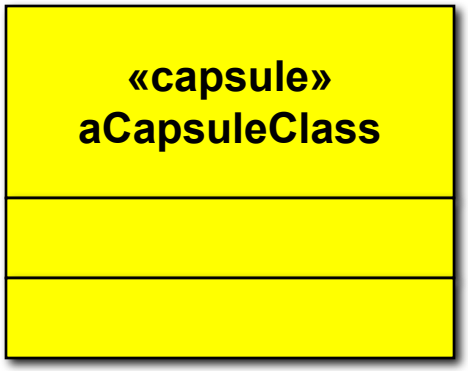
Stereotype	Base Class	Tags	Constraints
«capsule»	Class	language	isActive = true;
			self.feature->select(f | f.oclIsKindOf(Operation))-> forAll(o | o.elementOwnership.visibility = #protected)

Tag	Stereotype	Type	Multiplicity
language	«capsule»	String	0..1

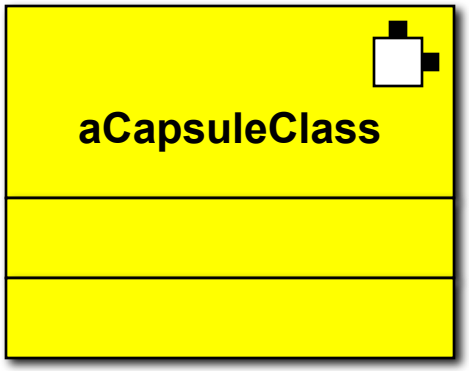
Stereotype Notation

- Several choices

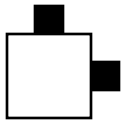
Stereotype icon



(a) with guillemets ("gwee-mays")



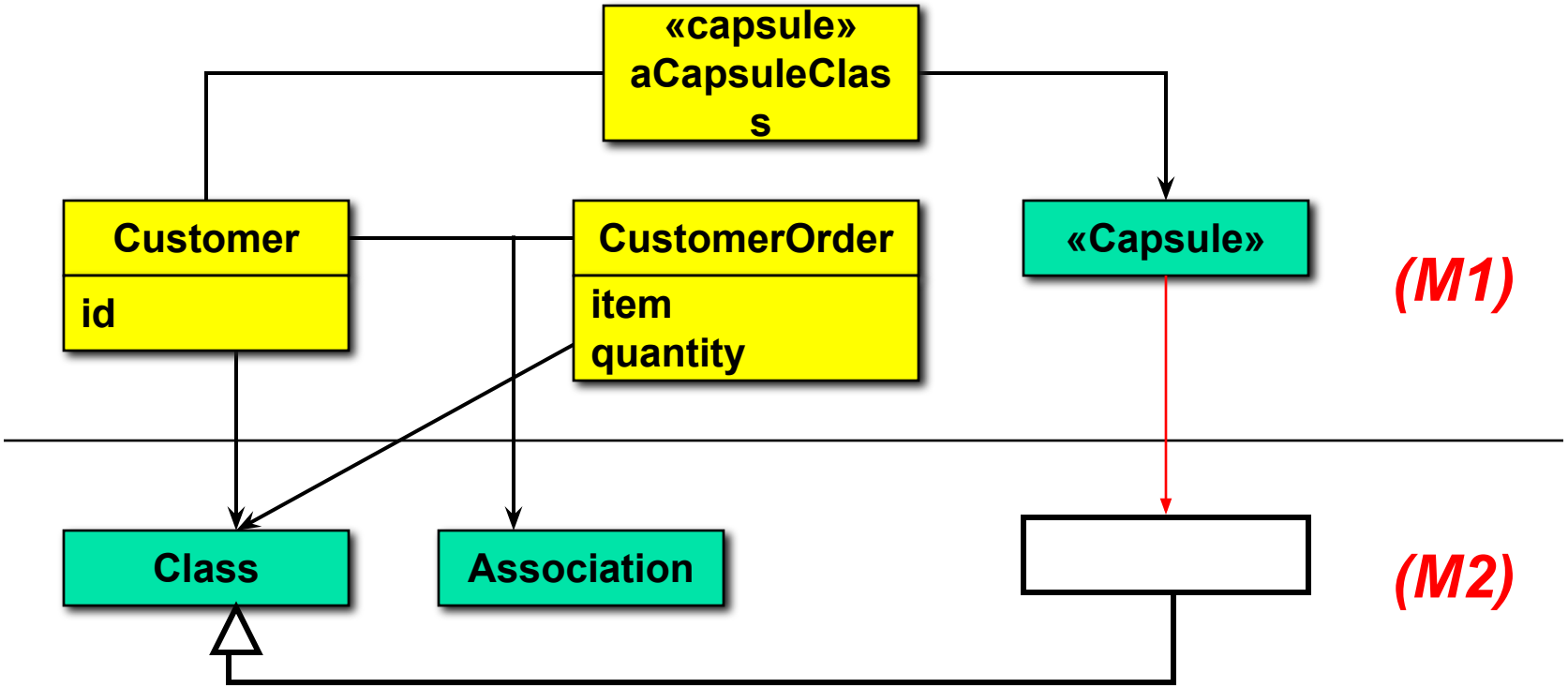
(b) with icon



(c) iconified form

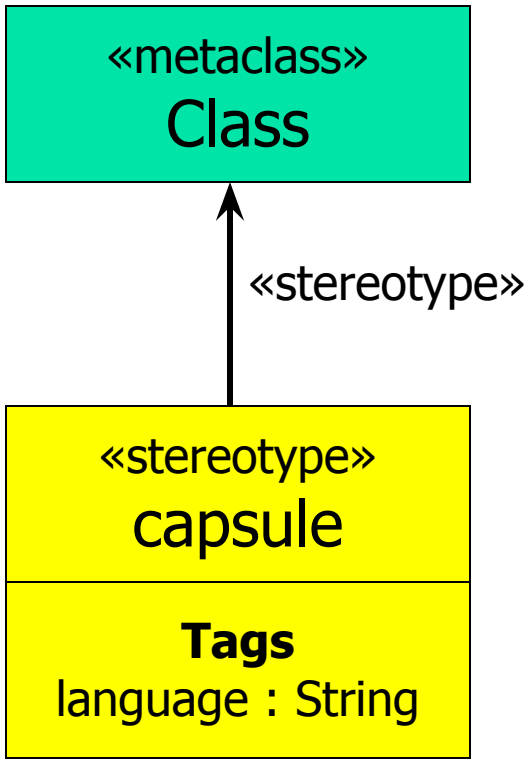
Extensibility Method

- Refinements are specified at the Model (M1) level but apply to the Meta-Model level (M2)
  - avoids need for "meta-modeling" CASE tools
  - can be exchanged with models



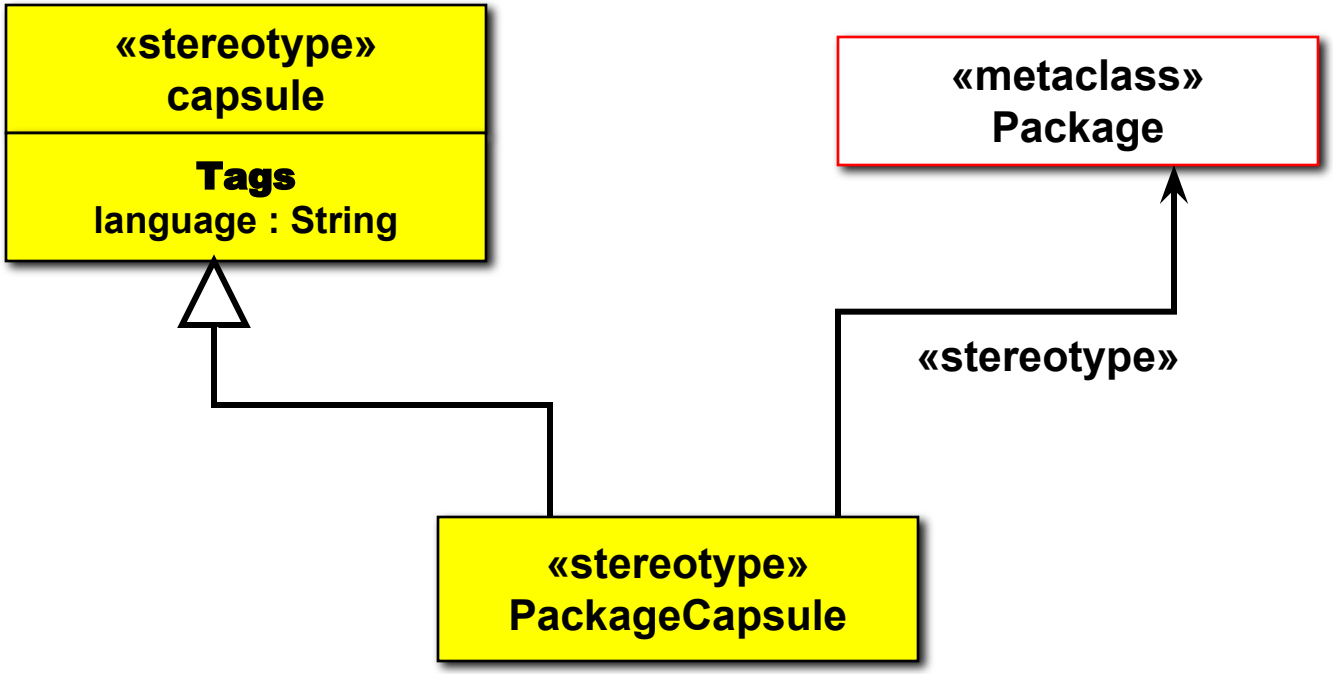
Graphical Definition

- Alternative to the tabular form
  - defined in a user (M1) model



Heuristic: Combining Stereotypes

- Through multiple inheritance:



When to Use Stereotypes?

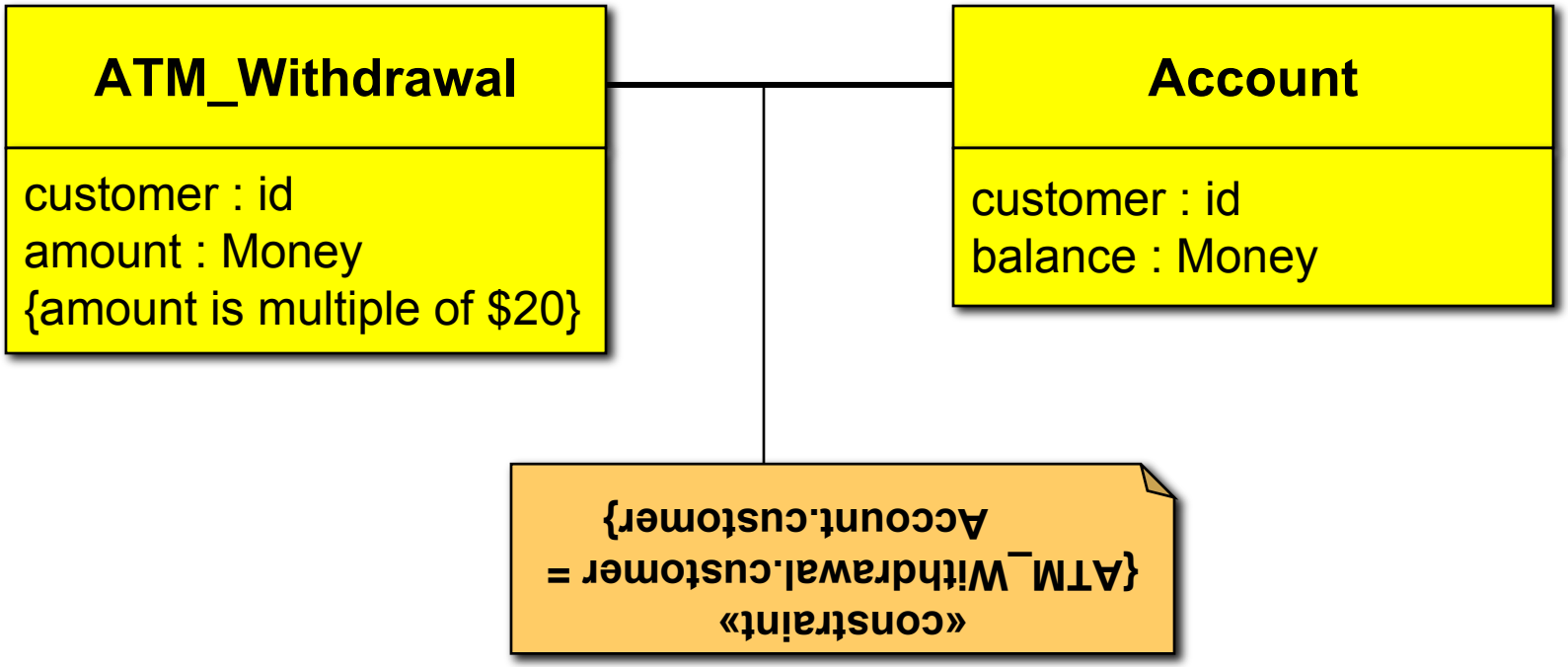
- Why not use normal subclassing instead?
- Use stereotypes when:
  - additional semantic constraints cannot be specified through standard M1-level modeling facilities
    - e.g. "all features have protected visibility"
  - the additional semantics have significance outside the scope of UML
    - e.g. instructions to a code generator "debugOn = true"

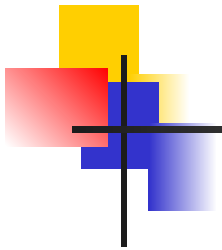
Tagged Values

- Consist of a *tag* and *value* pair
- Typed with a standard data type or M1 class name
- Typically used to model stereotype attributes
  - Additional information that is useful/required to implement/use the model
- May also be used independently of stereotypes
  - e.g., project management data ("status = unit_tested")

Constrains

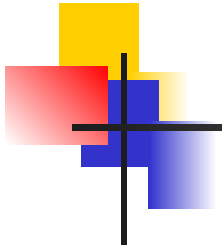
- Constraints
  - formal or informal expressions
  - must not contradict inherited base semantics





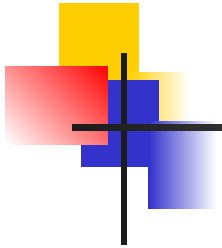
UML Profiles

- A package of related extensibility elements 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

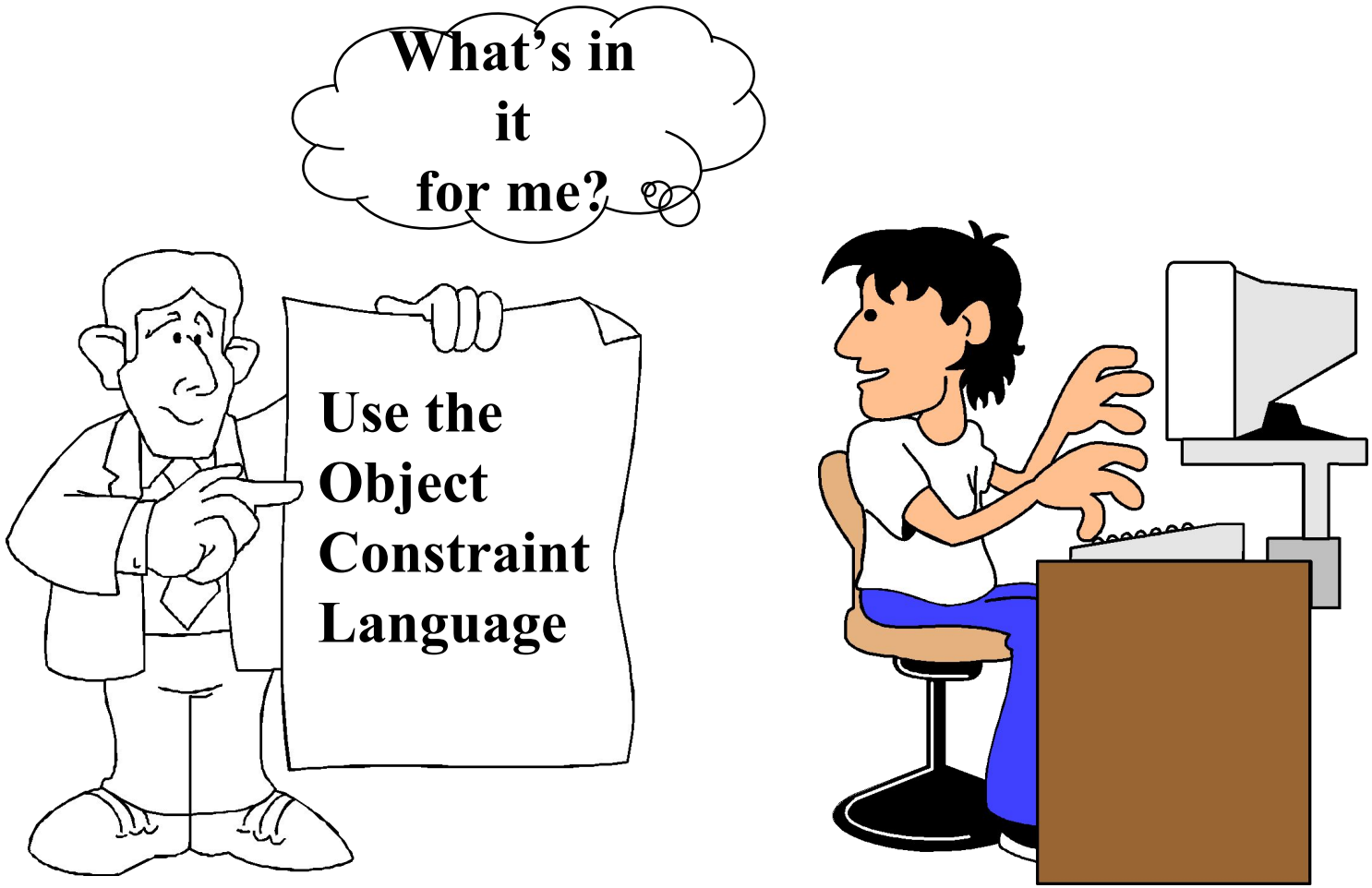
- Part 1: Model Management
- Part 2: Extension Mechanisms and Profiles
- Part 3: Object Constraint Language (OCL)



Overview

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

Why use OCL ?



Can we make this more precise?

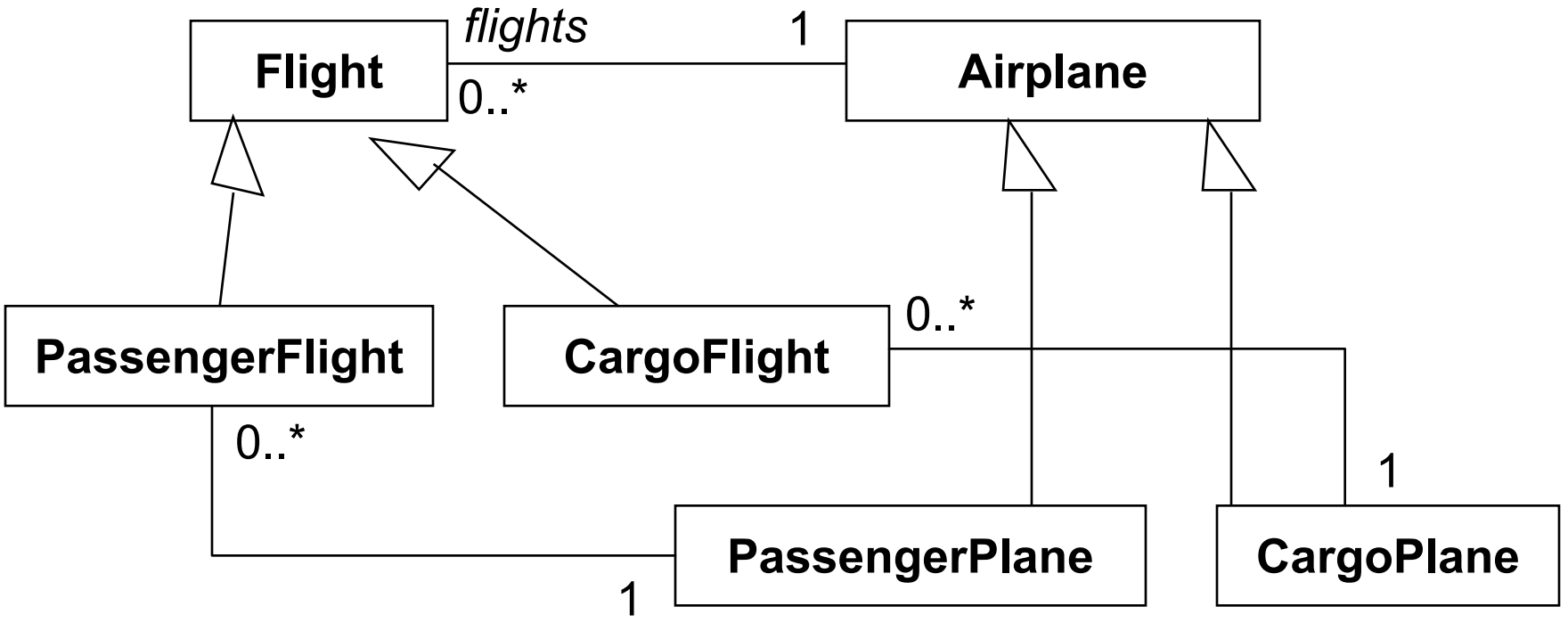
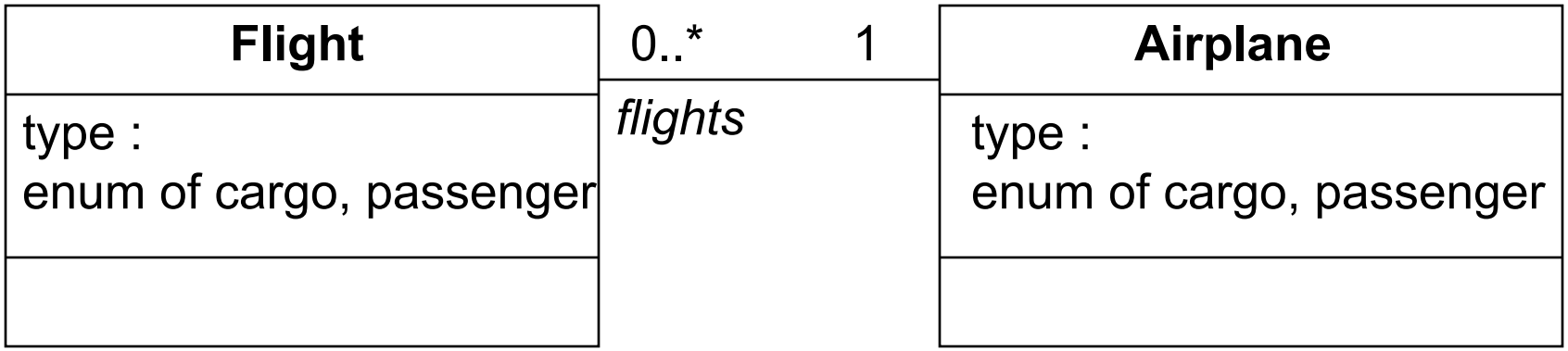


Diagram with added invariants



{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

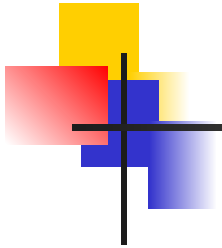
- UML defines three standard stereotypes for constraints:
  - invariant
  - precondition
  - postcondition

What is OCL?

- OCL is
  - a textual language to describe constraints
  - the constraint language used in UML models
    - As well as the UML meta-model
- Formal but easy to use
  - unambiguous
  - no side effects

Constraints and the UML model

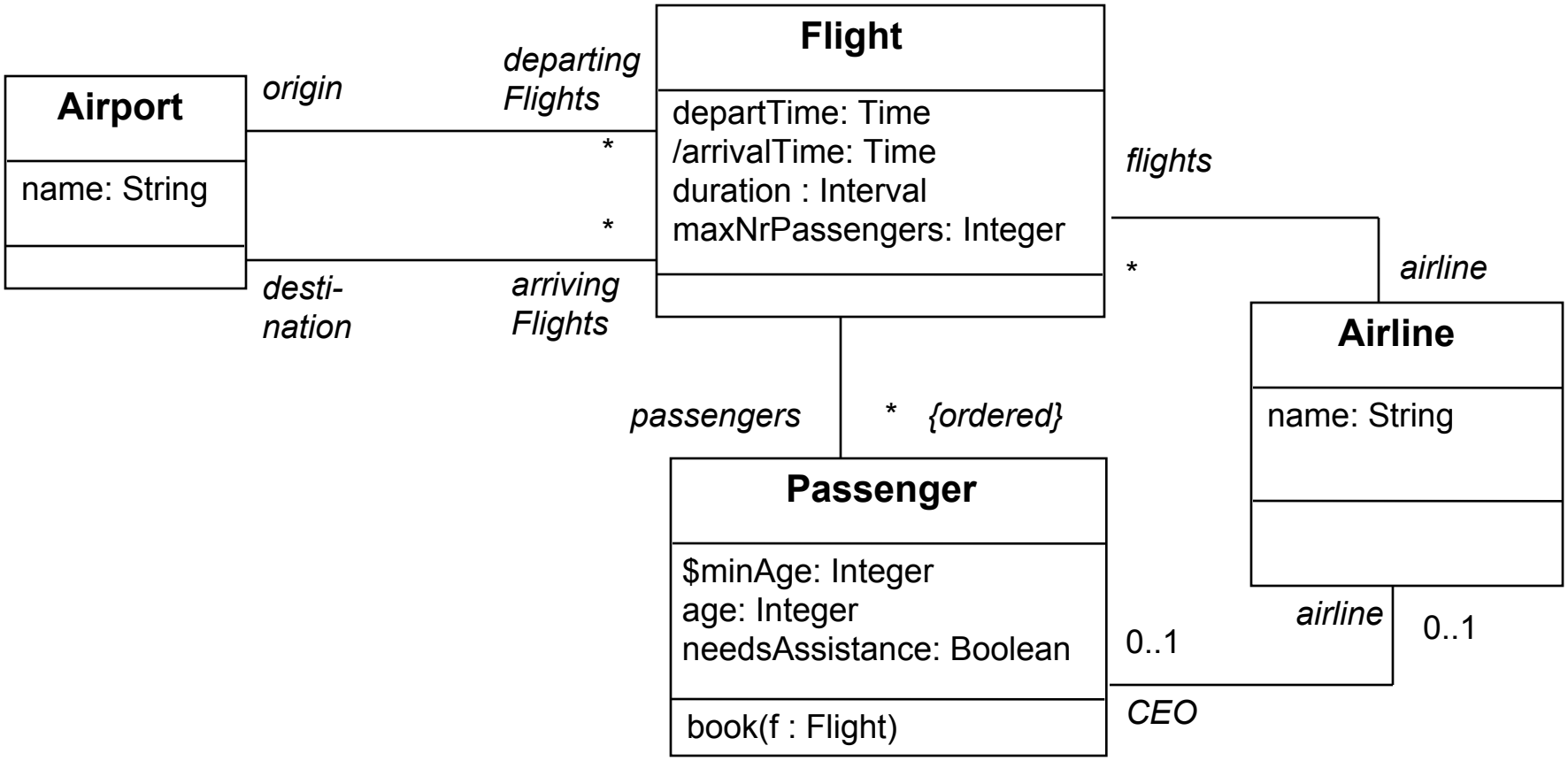
- OCL expressions are always bound to a UML model
  - OCL expressions can be bound to any model element in UML



Core OCL Concepts

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

Example model



Constraint context and self

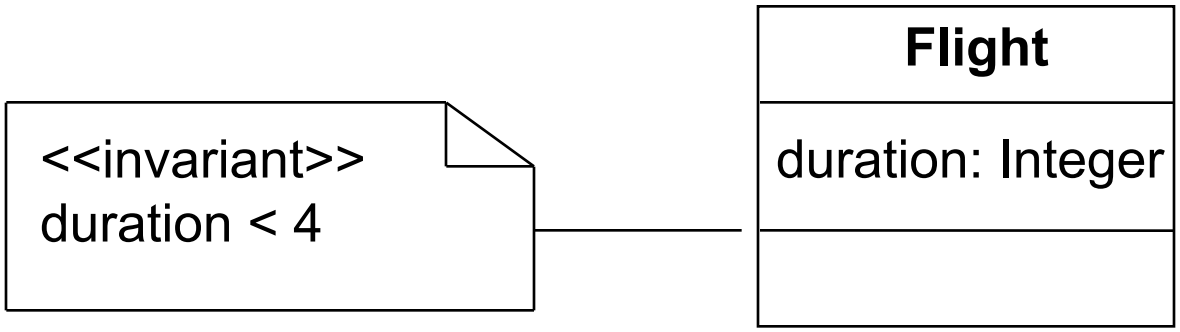
- Every OCL expression is bound to a specific context.
  - The context is often the element that the constraint is attached to
- The context may be denoted within the expression using the keyword ‘self’.
  - ‘self’ is implicit in all OCL expressions
  - Similar to`this’ in C++

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 (i.e., those operations that do not have side effects)
  - 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

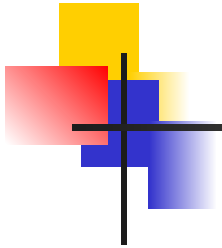
context Flight inv:

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

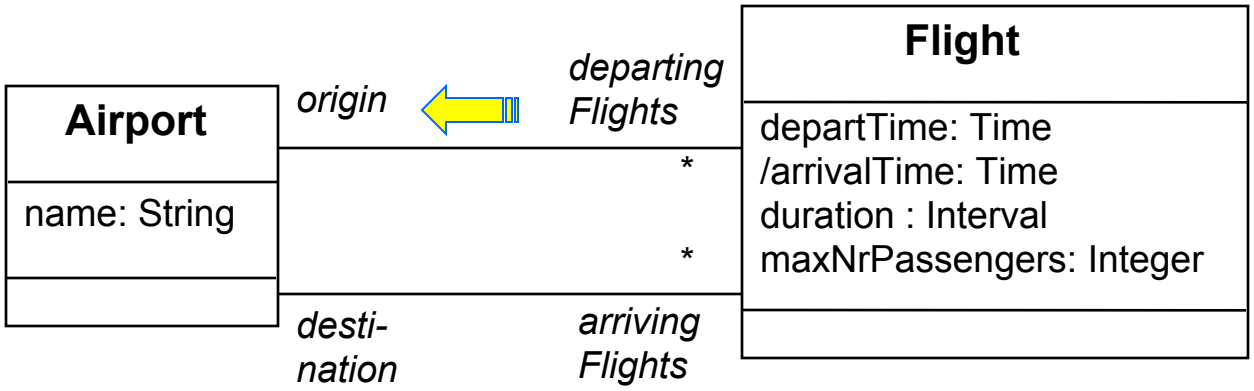
Interval
nrOfDays : Integer nrOfHours : Integer nrOfMinutes : Integer
equals(i:Interval):Boolean $Interval(d, h, m : Integer) : Interval

Associations and navigations



- Every association in the model is a navigation path.
- The context of the expression is the starting point.
- Role names are used to identify the navigated association.

Example: navigations



context Flight

inv: origin <> destination

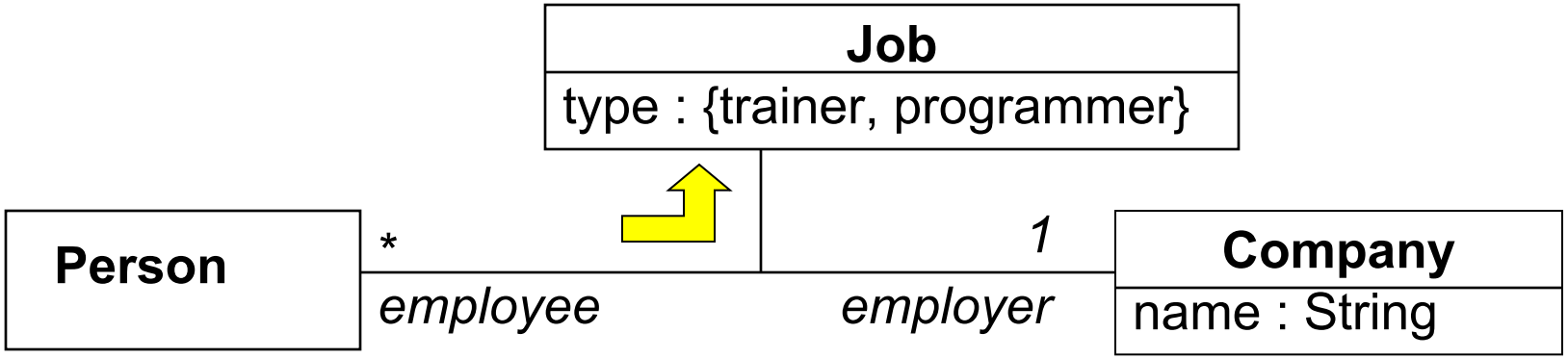
inv: origin.name = ‘Amsterdam’

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
```

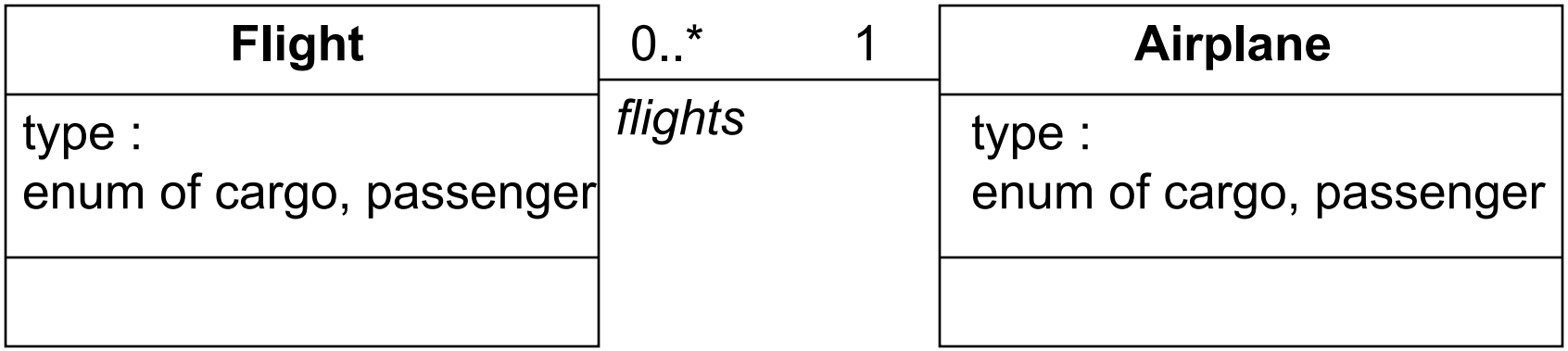


The OCL Collection types

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

Significance of Collections in OCL

- Most navigations return collections rather than single elements



Three Subtypes of Collection

- Set:
  - arrivingFlights(from the context Airport)
  - Non-ordered, unique
- Bag:
  - arrivingFlights.duration (from the context Airport)
  - Non-ordered, non-unique
- Sequence:
  - passengers (from the context Flight)
  - Ordered, non-unique

Collection operations

- OCL has a great number of predefined operations on the collection types.
- Syntax:
  - collection->operation

Use of the "->" (arrow) operator instead of the "." (dot) operator

The collect operation

- Syntax:

  collection->collect(elem : T | expr)

  collection->collect(elem | expr)

  collection->collect(expr)

- Shorthand:

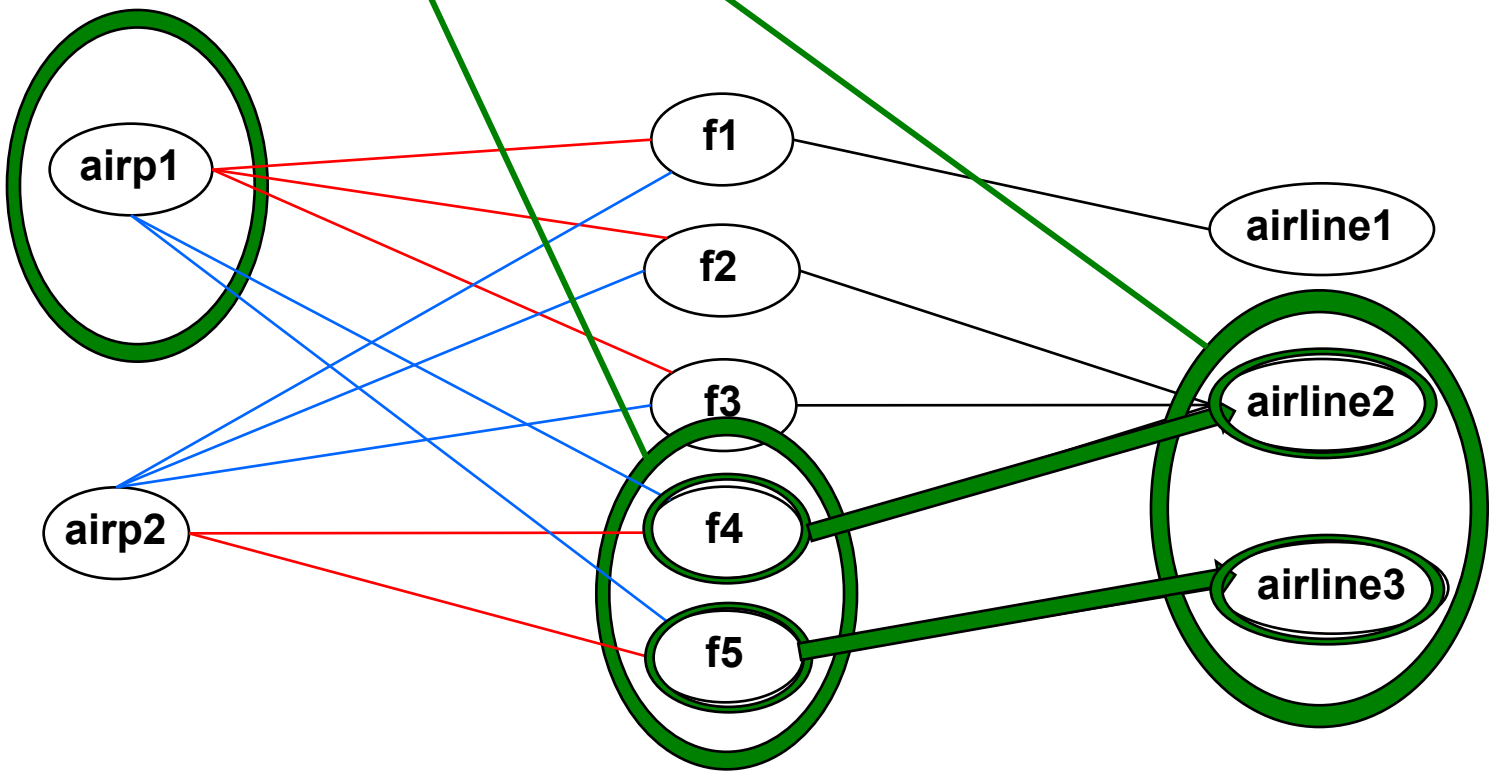
  collection.expr

- The *collect* operation results in the collection of the values resulting evaluating *expr* for all elements in the *collection*
- Shorthand often trips people up. Be Careful!

Example: collect operation

context Airport inv:

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



departing flights

arriving flights

The select operation

- Syntax:

  collection->select(elem : T | expression)

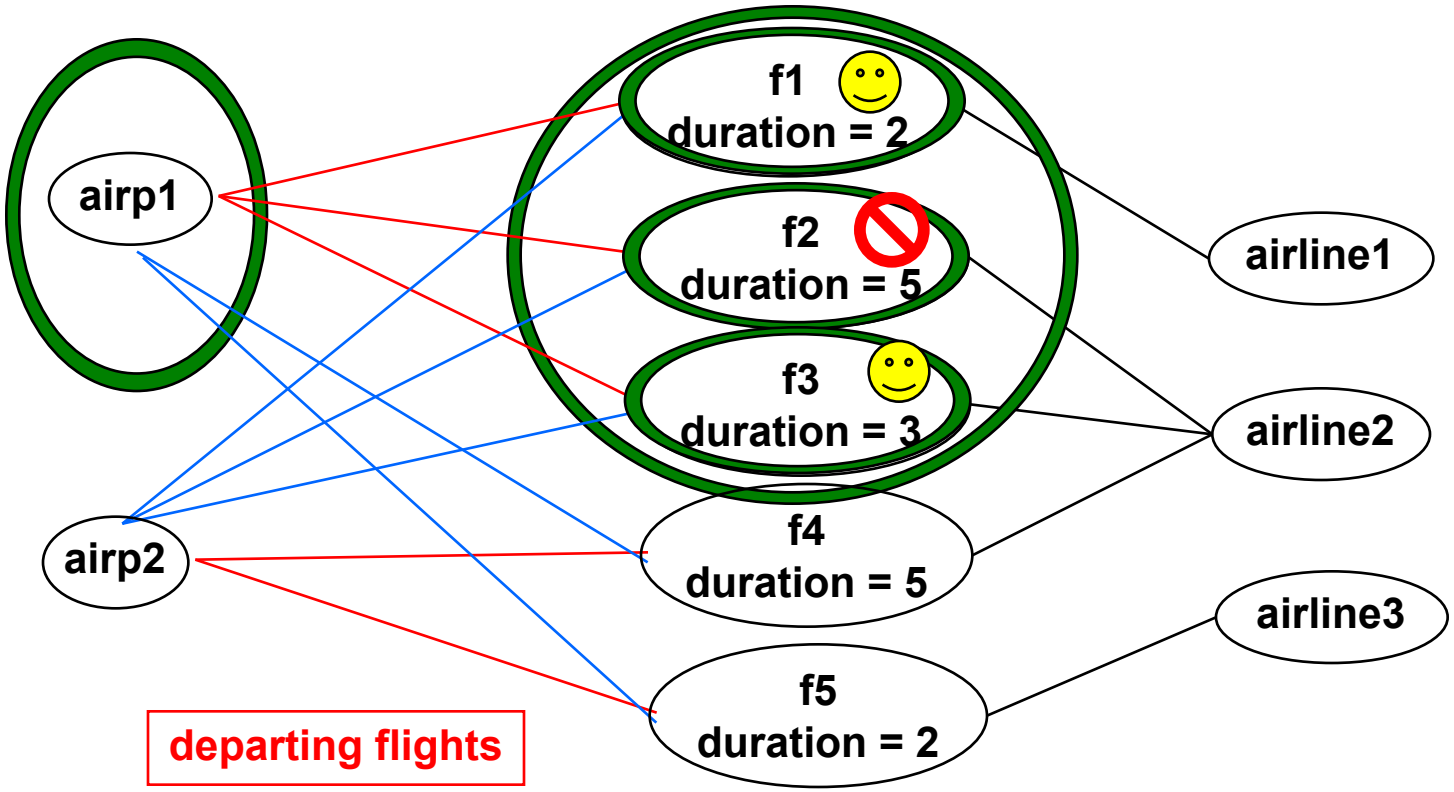
  collection->select(elem | expression)

  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

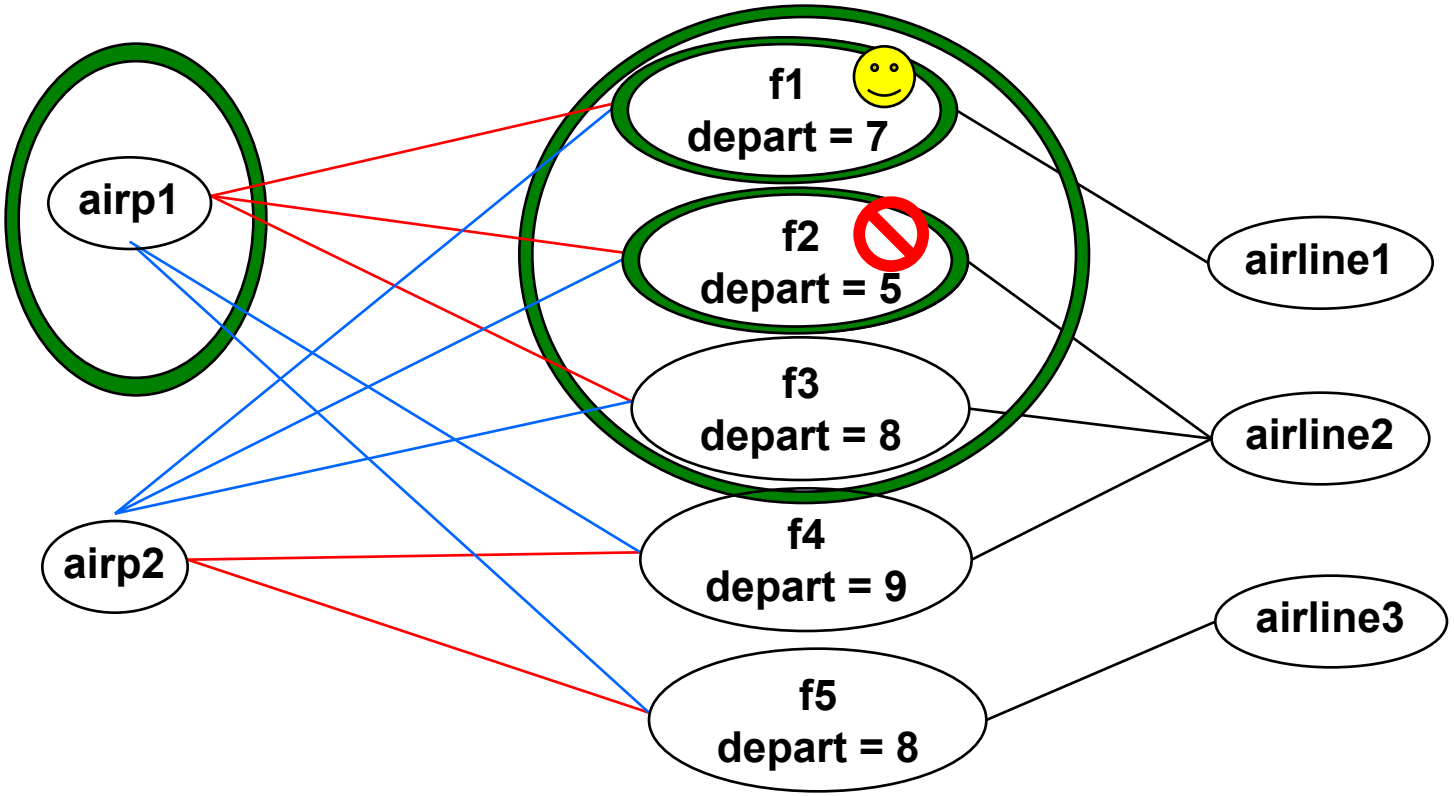


The forAll operation

- 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

Example: forAll operation

context Airport inv:
self.departingFlights->forAll(departTime.hour>6)



The exists operation

- Syntax:

  collection->exists(elem : T | expr)

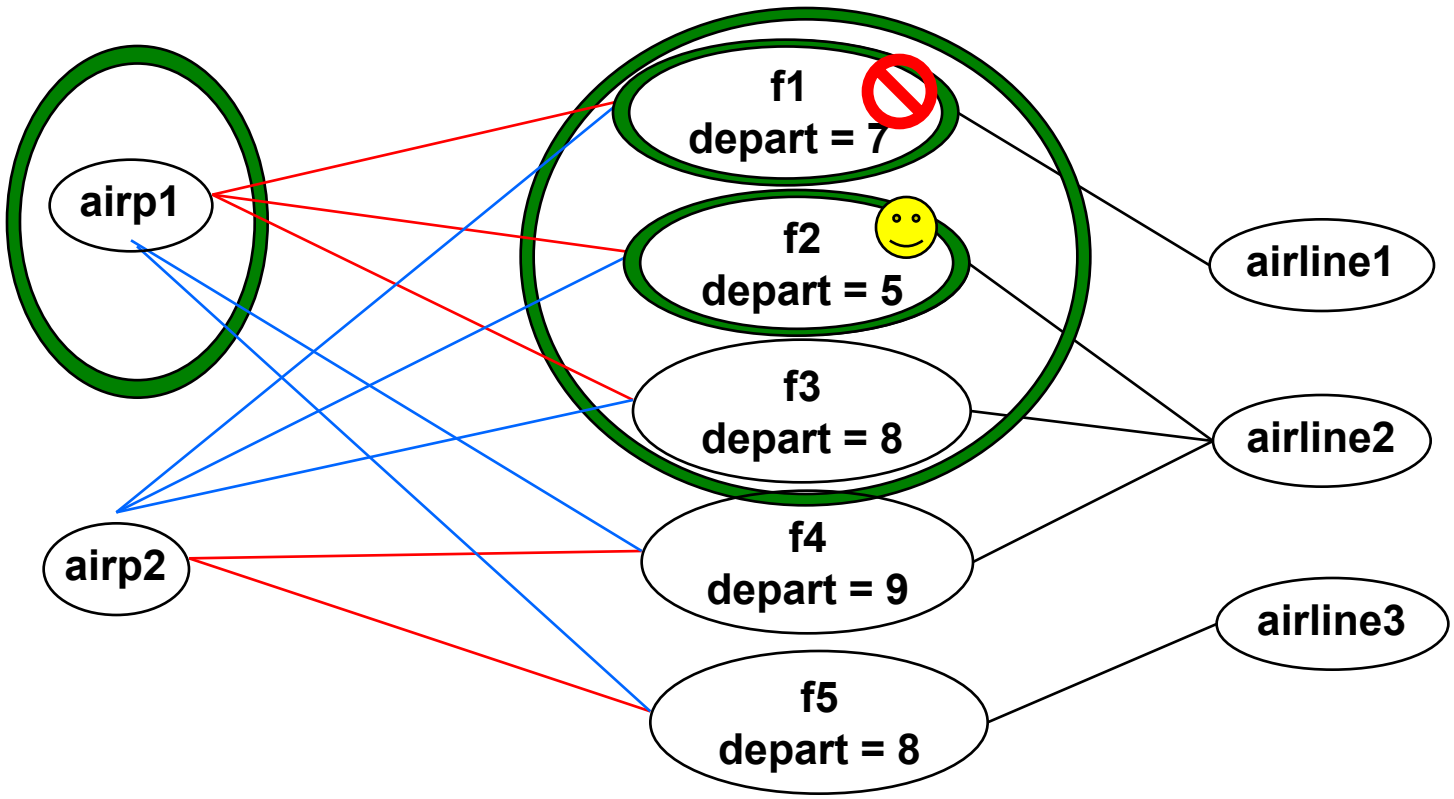
  collection->exists(elem | expr)

  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.

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

Advanced OCL Concepts

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

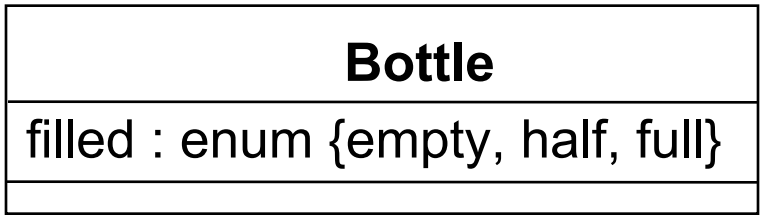
Result in postcondition

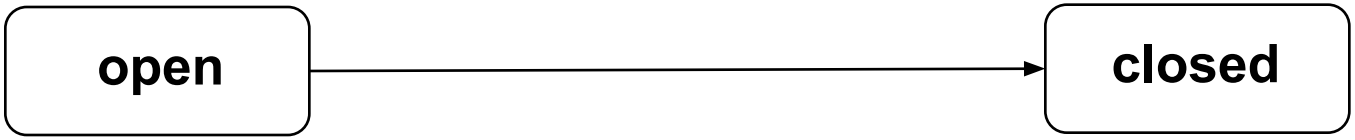
- Example pre and postcondition

```
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>

- Example:

  context Airport inv:

  Let supportedAirlines : Set (Airline) =
  self.arrivingFlights -> collect(airLine) in
  (supportedAirlines ->notEmpty) and
  (supportedAirlines ->size < 500)

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 Liskov's 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* (contravariance)
  - A postcondition may be strengthened (covariance)