

SYSC-3120—Software Requirements Engineering

Part III - Object-Oriented Analysis

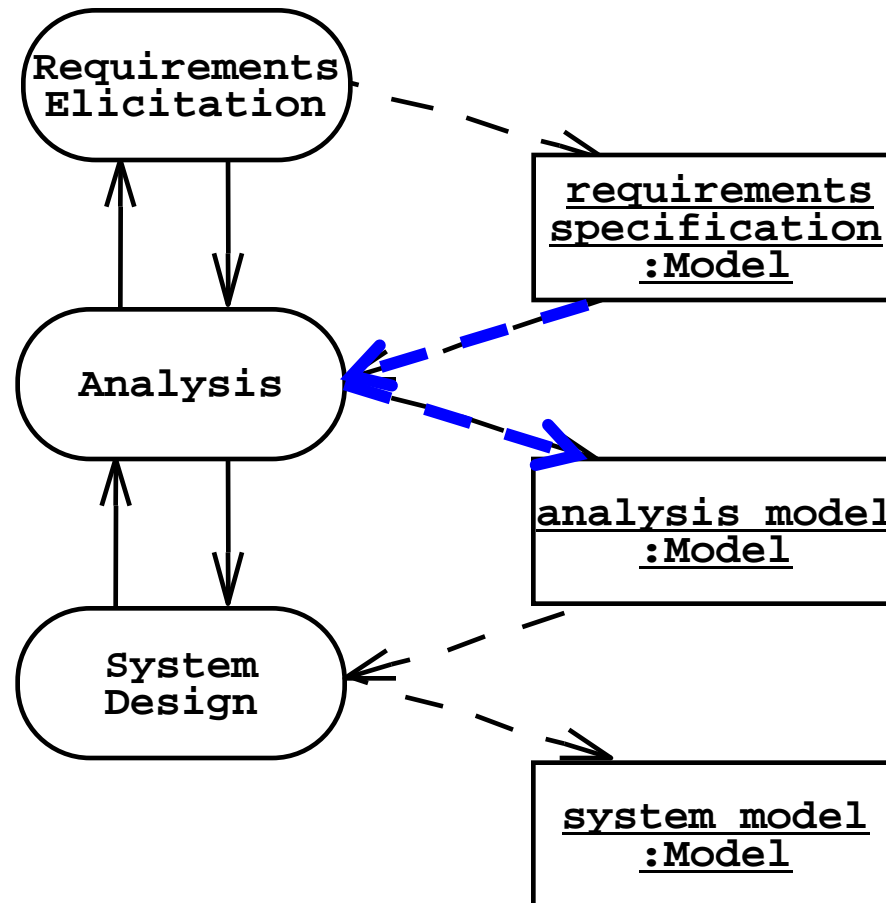
SYSC-3120—Software Requirements Engineering

- Overview
- Analysis Concepts
 - Modeling structure: class diagram
 - Modeling interaction: sequence diagram
 - Modeling state-based behavior: state machine diagram
- Analysis Process
 - Finding objects/classes (heuristics)
 - Finding relationships: associations and attributes (heuristics)
 - Interactions/behavior (heuristics)
 - Responsibilities
 - Analysis review

Overview

- Transform the **specifications** of the system into a form suitable to designers, from the requirement elicitation results (use case model)
- Systematic procedure, heuristics
- Analysis of **problem domain**, as opposed to **solution domain** (Design)
- Model composed of static and dynamic UML models
 - Static model: classes relationships attributes (modeling system structure)
 - Dynamic model: object behavior, interactions between objects (modeling system behavior)
- The result is a model that is (expected to be) correct, complete, consistent and verifiable.

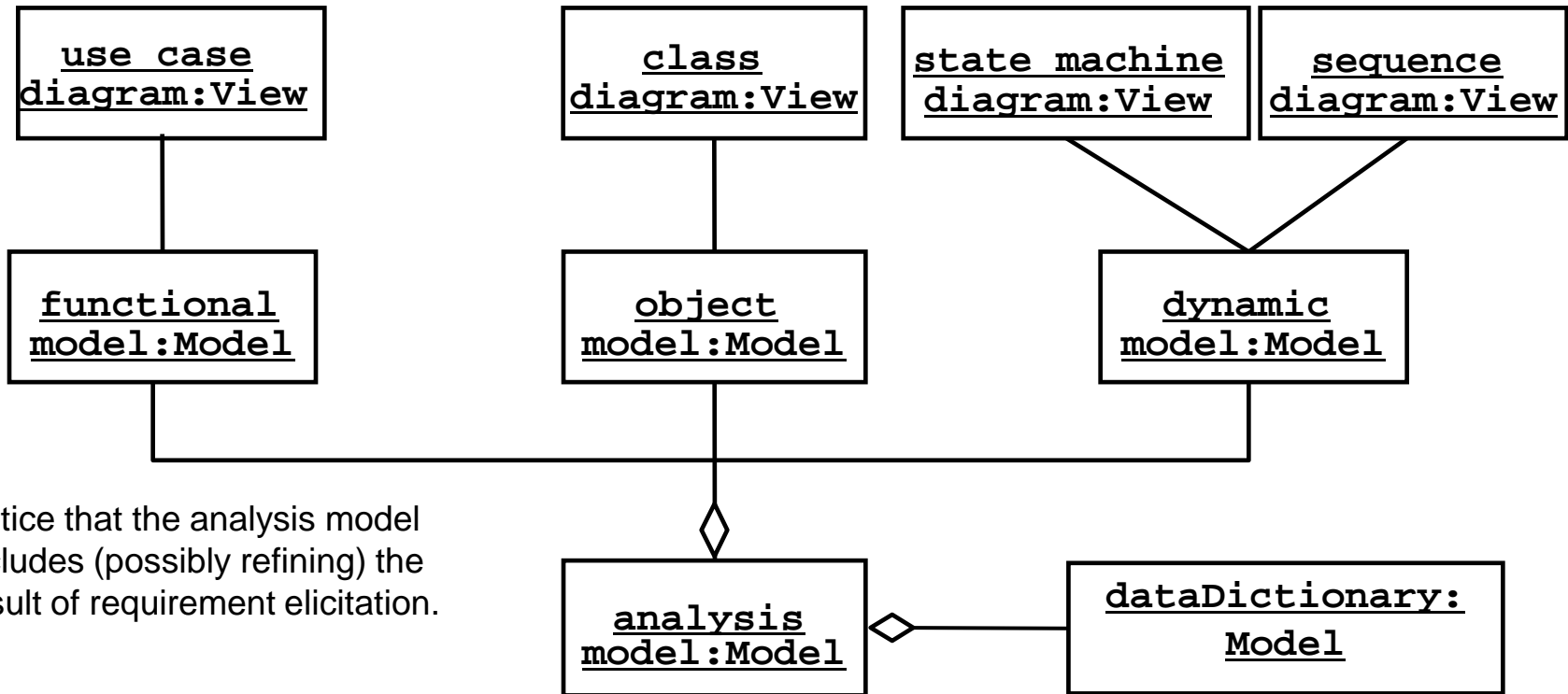
Overview (Bruegge and Dutoit, 2000)



Object-Oriented, UML-based Analysis Model (Bruegge and Dutoit, 2000)

Including:

- Operation's pre- and post-conditions
- Class invariants



Notice that the analysis model includes (possibly refining) the result of requirement elicitation.

Ideally automatically generated

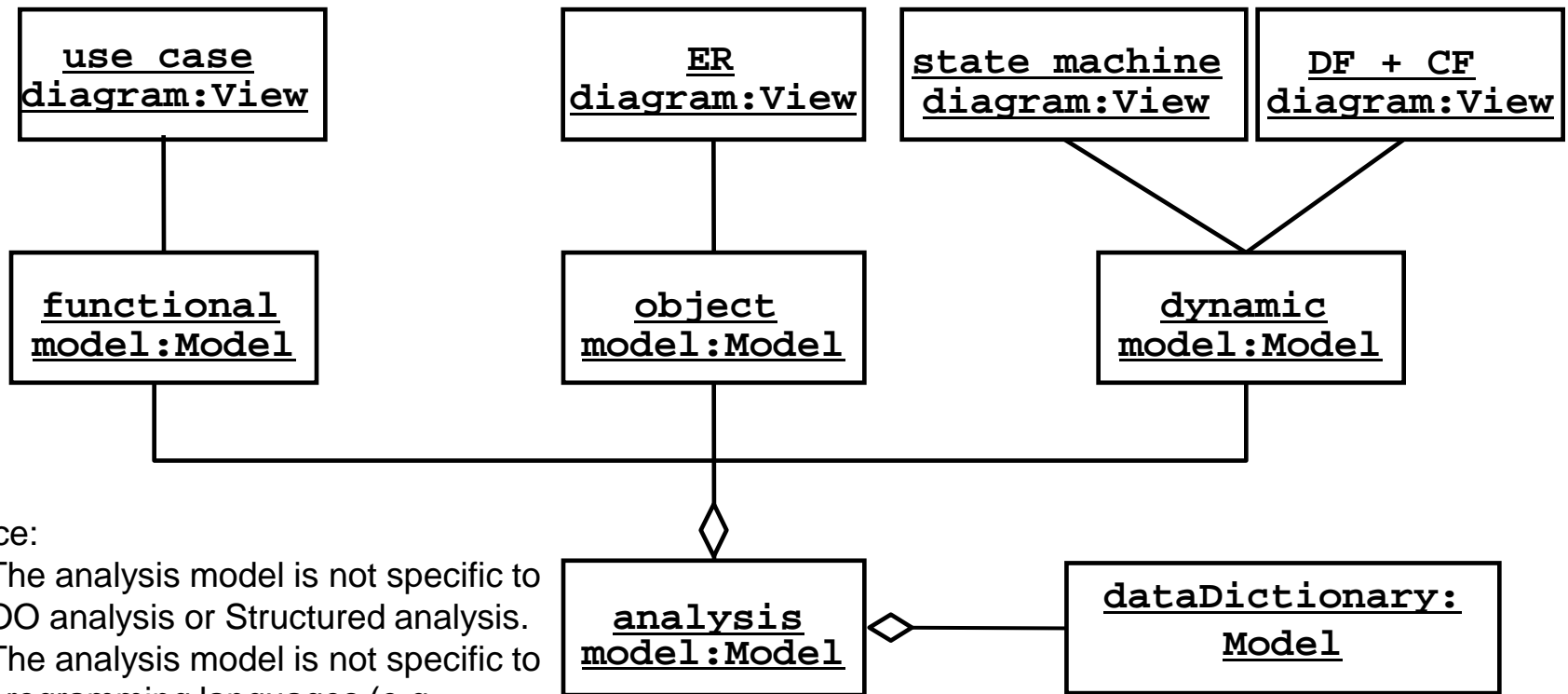
Analysis Model

(alternative to Object-Oriented, UML-based)

- **Structured Analysis**: data and processes manipulating data are considered separately
 - Data objects are modeled with attributes and relationships
 - E.g., with an Entity Relationship (ER) diagram
 - Processes are modeled to show how they transform data
 - E.g., Data Flow (DF) model, Control Flow (CF) model, lead to functional decomposition of the system's responsibilities
- Related development processes:
 - Structured analysis, structured design (SA/SD)
 - Structured analysis design technique (SADT)

Analysis Model

(alternative to Object-Oriented, UML-based)



Notice:

- The analysis model is not specific to OO analysis or Structured analysis.
- The analysis model is not specific to programming languages (e.g., C++/Java for OO analysis, C for Structured analysis)

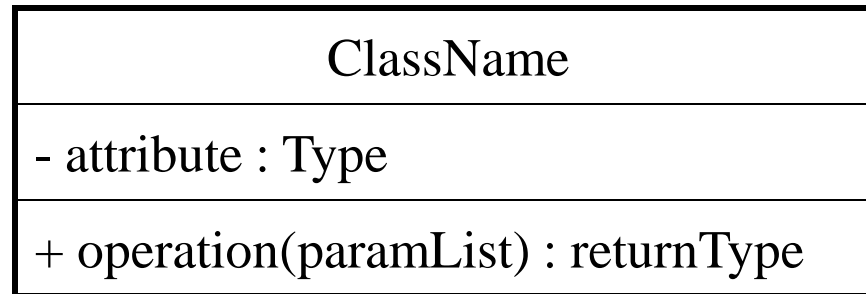
Ideally automatically generated

SYSC-3120—Software Requirements Engineering

- Overview
- Analysis Concepts
 - Modeling structure: class diagram
 - Modeling classes
 - Different kinds of relationships
 - OCL: Better specifying operations/classes, constraining class diagram
 - Liskov substitution principle
 - Modeling interaction: sequence diagram
 - Modeling state-based behavior: state machine diagram
- Analysis Process
 - Finding objects/classes (heuristics)
 - ...

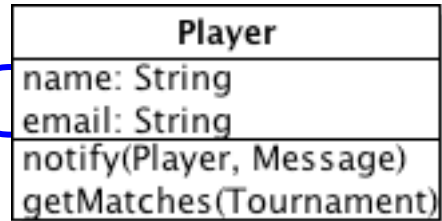
Class Definition

- UML definition: “description of a set of objects that share the same attributes, operations, and relationships [OMG UML Guide, 1999]
- Class structure: a 3-part box
 - Attribute: name, type, visibility
 - Operation: name, parameter list (name, type, direction), return type, visibility



- Class name in italic = abstract class

Class Definition



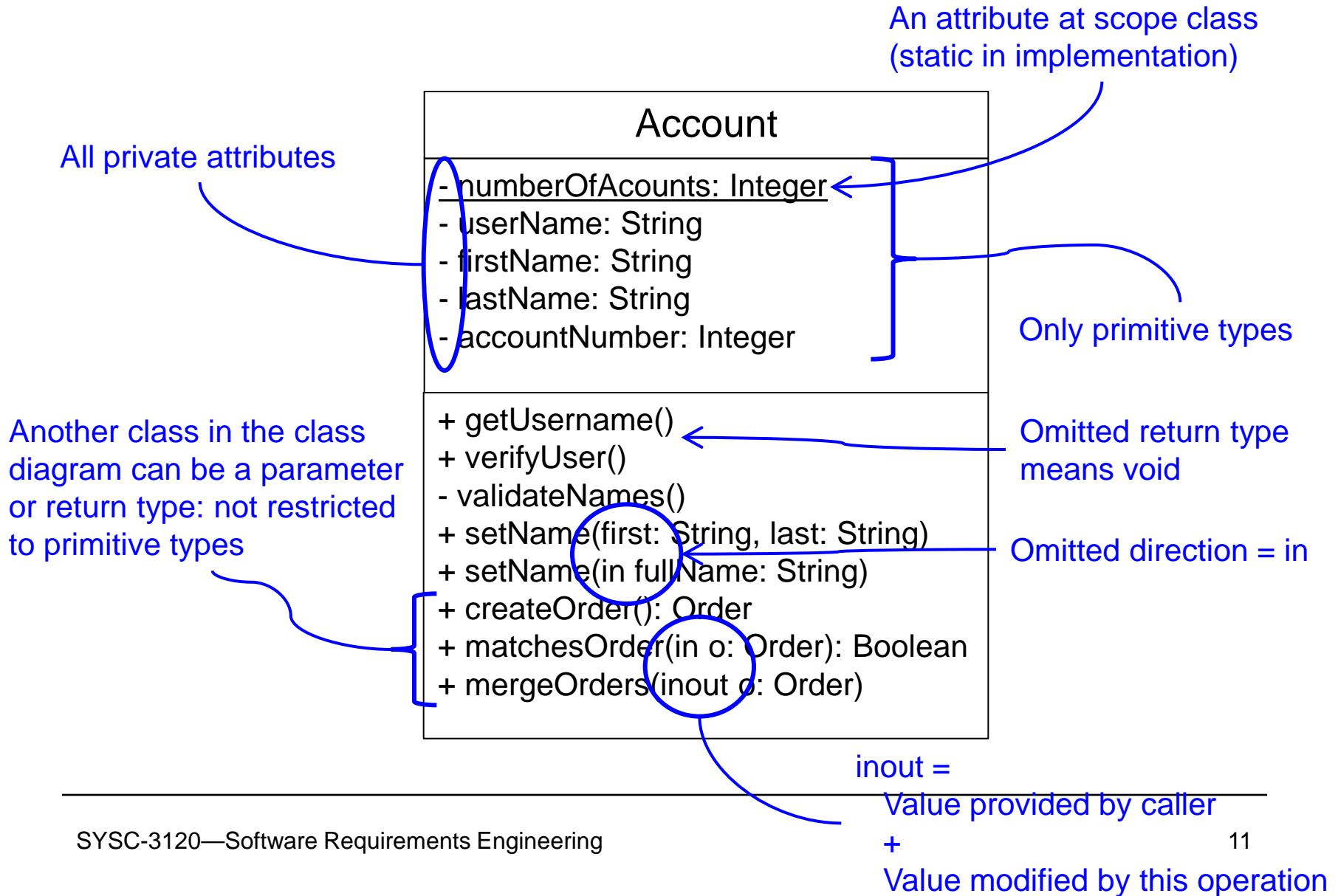
An attribute

- has a name
- has a type:
 - only primitive types allowed
 - Boolean, String, Integer, Real
- has a visibility
 - + name: type a public attribute (default)
 - name: type a private attribute
 - # name: type a protected attribute
 - ~ name: type a package-visible attribute
- class vs. object scope (i.e., static)
 - Class scope = underlined definition
 - Implementation: class ≈ static

An operation

- has a name
- has a return type (optional)
- has a visibility
- class vs. object scope
- has parameters (optional)
 - name (optional)
 - type: primitive types, class types
 - direction: in/out/inout
 - in = passed by caller (default when omitted)
 - out = like a returned value
 - inout = both
- E.g.:
 - + setSize(in name:String)
 - + getSize(): Rectangle

Class Definition—Example



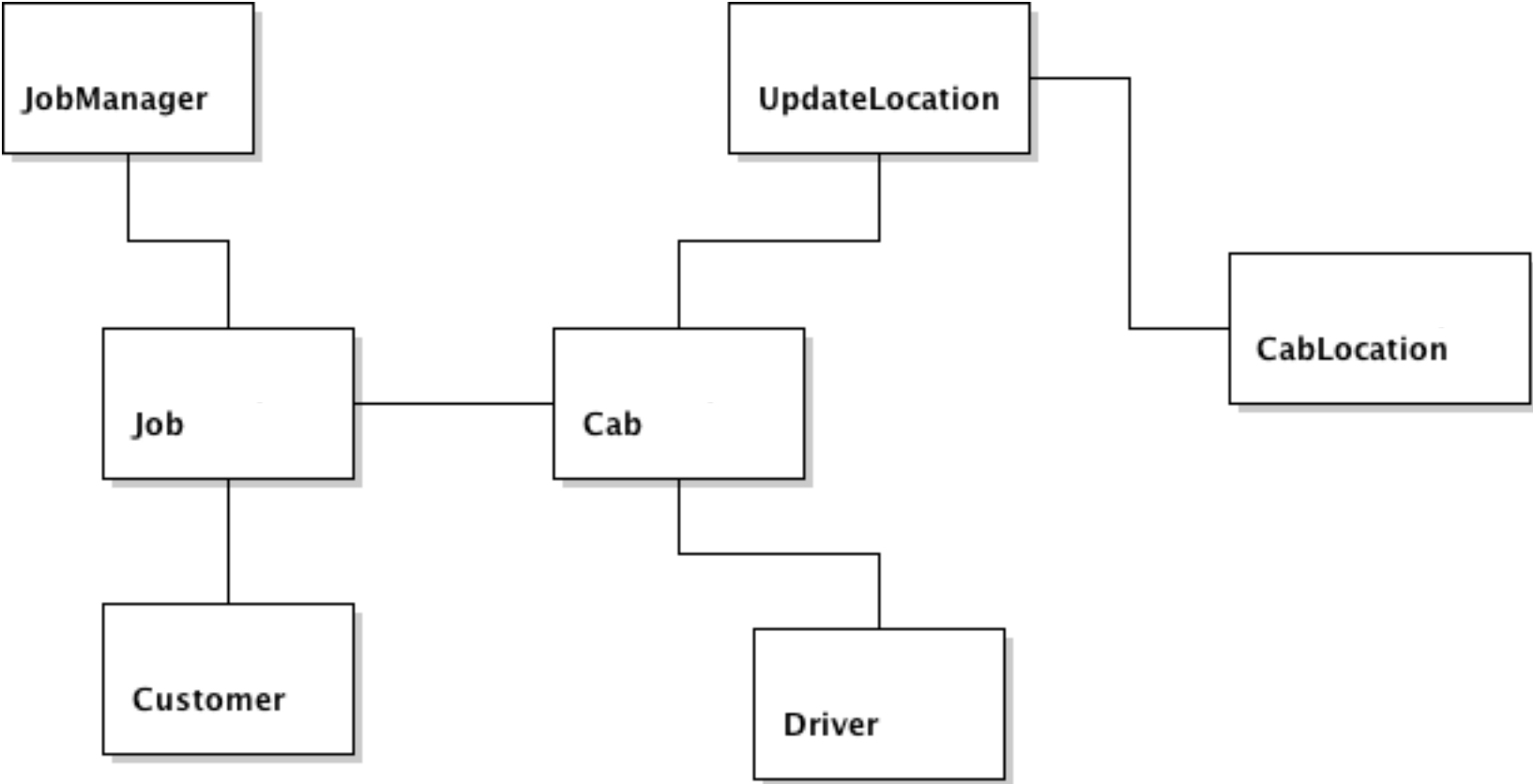
Class/Object Taxonomy

- Objectives:
 - facilitate the identification of classes/objects' responsibilities
- Result: classification of classes
 - *Entity Class/Object*
 - Represents the persistent information tracked by the system (Application domain objects, “Business objects”)
 - *Control Class/Object*
 - Represents the control tasks performed by the system, contains the logic and determines the order of object interactions
 - Implements the logic of use cases
 - *Boundary Class/Object*
 - Represents the interaction between the actors and the system: user interface object, device interface object, system interface object
 - Only transmits data, without change of semantics, possibly with change of format
- There are other class taxonomies

Use of Class/Object Taxonomy

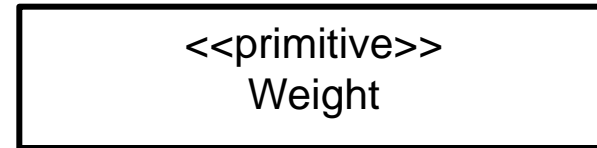
- Model that is more resilient to change.
 - The boundary objects are more likely to change than the control
 - The control objects are more likely to change than the application logic and entity objects
 - Limit impact (propagation) of changes when errors are corrected or requirements change
- Helps identify classes/objects and clearly identify responsibilities
- Helps read class diagram (using stereotypes)
 - Use string <<Boundary>> before the class name
 - Use string <<Control>> before the class name
 - Use string <<Entity>> before the class name
- Helps verification:
 - Clear responsibilities
 - bypass the boundary classes
- Clarify object interactions in sequence diagrams (see later)

Class Taxonomy

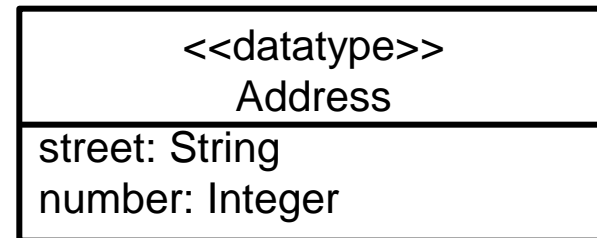


User-defined Types

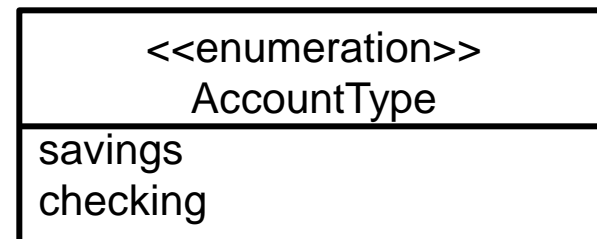
- Primitive Types
 - you can define your own primitive type



- Data Types
 - structured type



- Enumeration



- These can be used as attribute types, in addition to the previously mentioned primitive types. **No other class allowed!**

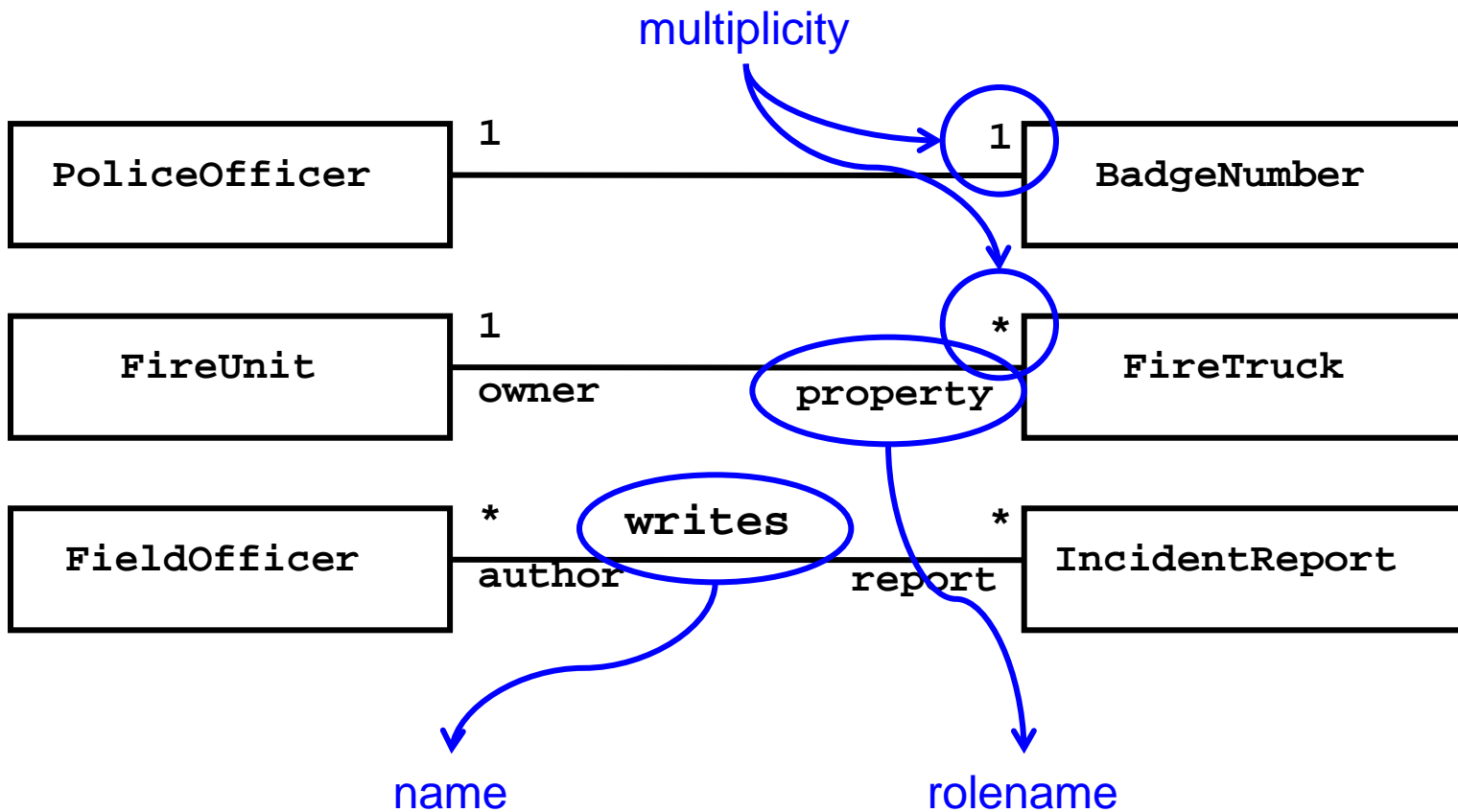
SYSC-3120—Software Requirements Engineering

- Overview
- Analysis Concepts
 - Modeling structure: class diagram
 - Modeling classes
 - Different kinds of relationships
 - Association
 - Generalization and Realization
 - Dependency
 - OCL: Better specifying operations/classes, constraining class diagram
 - Liskov substitution principle
 - Modeling interaction: sequence diagram
 - Modeling state-based behavior: state machine diagram
- Analysis Process
 - Finding objects/classes (heuristics)
 - ...

Association

- Models the possibility of links between instances of two or more (associated) classes
 - Describes a group of links (between instances) with common structure and semantics
 - Mathematically correspond to a set of tuples (relations)
- Multiplicities indicate the size of tuples
- Different kinds of associations:
 - Plain association
 - Aggregation
 - Composition
- Can have a name (not that useful)
- Can have rolename on each end (very useful)
- Can have a direction (rather a design decision)

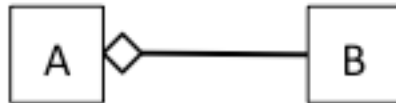
Associations—Other Examples (the FRIEND system)



Association: kinds and direction



Plain association: A and B are associated



Aggregation: A is an aggregate of B(s)



Composition: A is composed of B(s)

We will discuss semantics differences later

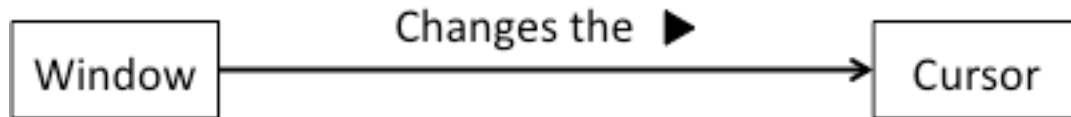


A and B “know” each other



A “knows” B, but B does not “know” A

Association: name

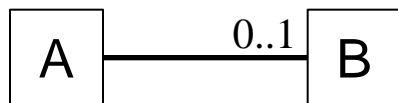


- An association can have a name.
- A name is optional.
- When using a name, add an indication (triangle) to help “read” the association.
- The association name appears in the middle between the two classes

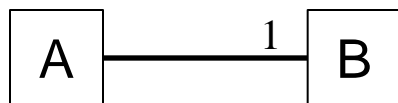
Association: multiplicities

- Models the possibility of links between instances of two or more (associated) classes
 - Describes a group of links (between instances) with common structure and semantics
 - Mathematically correspond to a set of tuples (relations)
- Multiplicities indicate the size of tuples
- Multiplicities at both ends of the association (line).
- Multiplicities are strongly recommended!
 - See later
- The default (omitted) multiplicity is 1.
 - See alternatives next
- They specify the number of instances of the classes at each end of the association that can be linked at runtime. (Not at any instant in time though: see later)

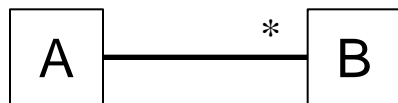
Association: multiplicities



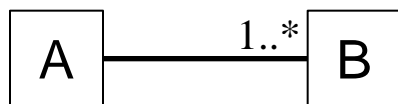
Between operation executions (when objects are idle), an instance of A is linked to 0 or 1 (not more) instance of B



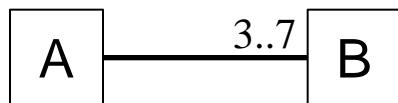
Between operation executions (when objects are idle), an instance of A is linked to exactly 1 instance of B



Between operation executions (when objects are idle), an instance of A is linked to 0 or any arbitrary number of instances of B

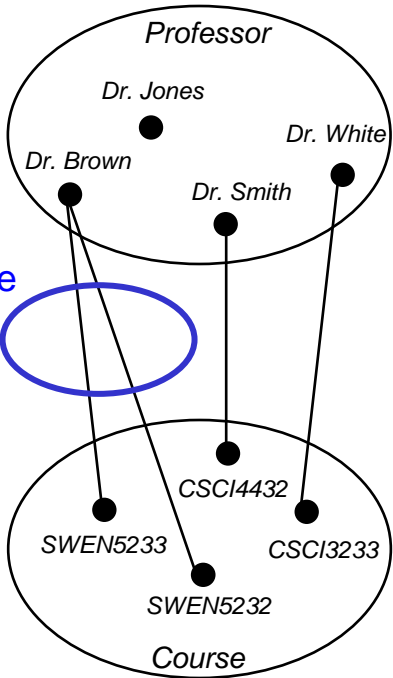
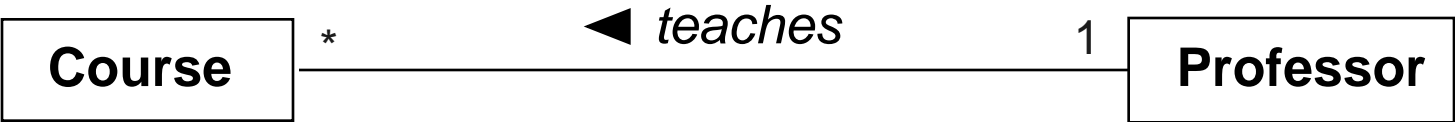


Between operation executions (when objects are idle), an instance of A is linked to at least 1 instance of B, and possibly any arbitrary number of instances of B



Between operation executions (when objects are idle), an instance of A is linked to at least 3 but no more than 7 instances of B

Association: multiplicities

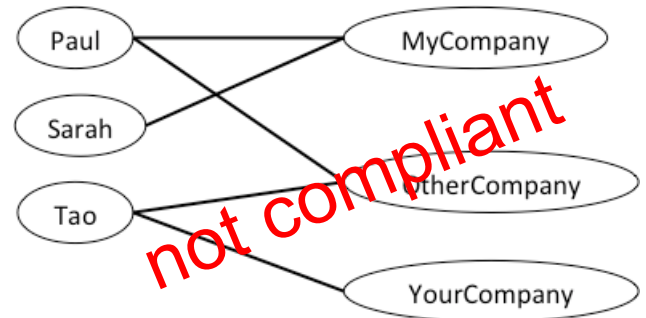
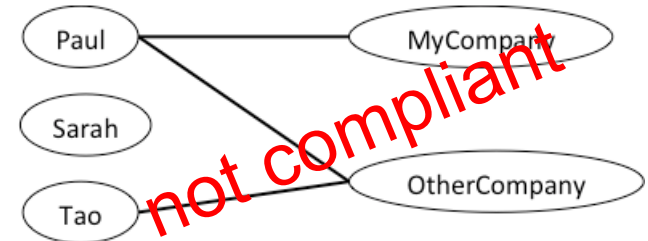
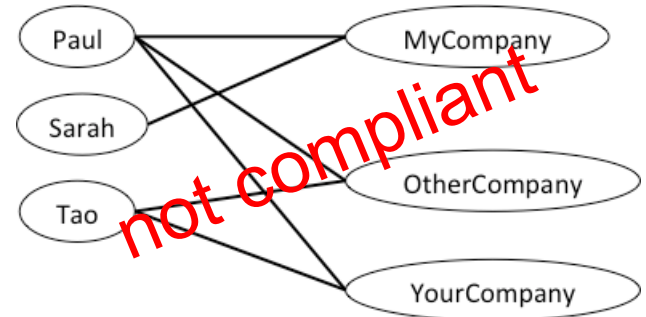
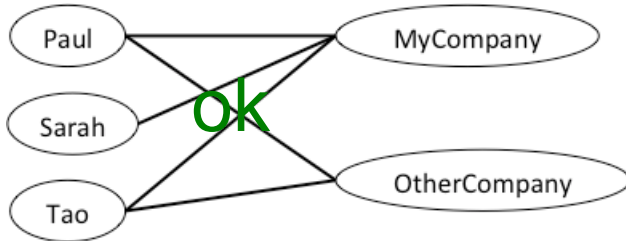
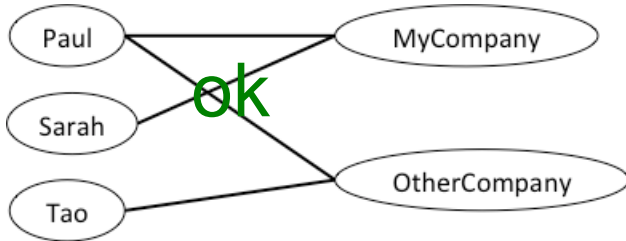
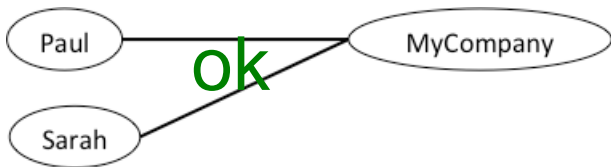
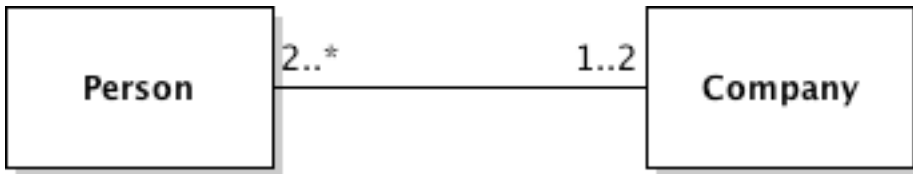


This is one instance of the association

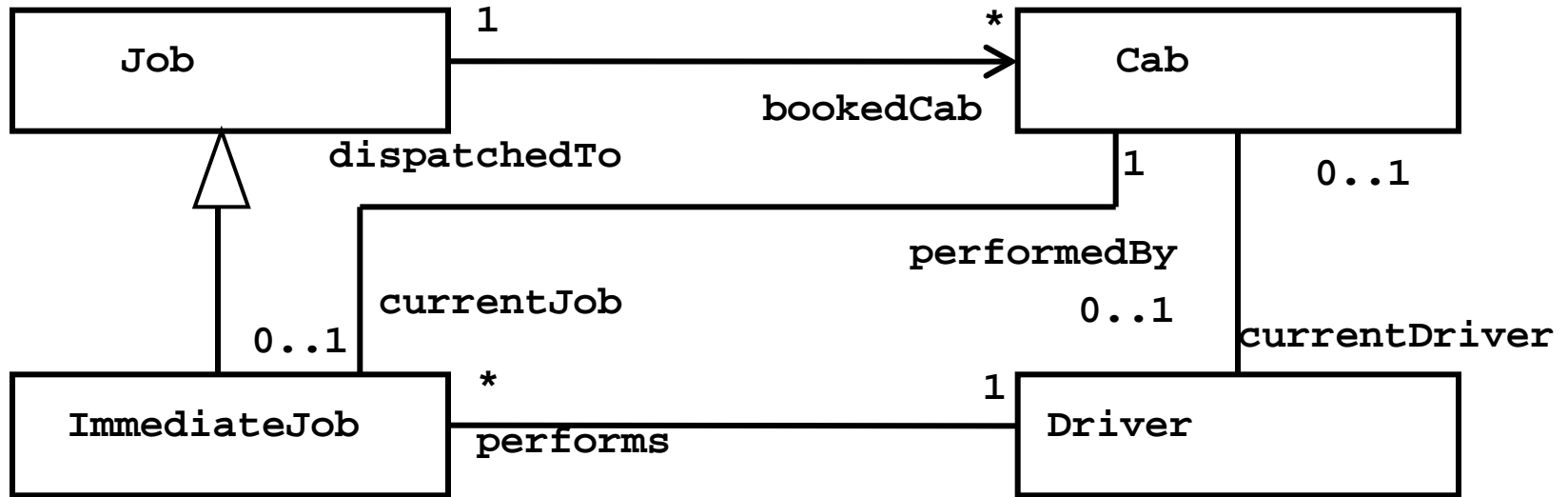
<i>teaches</i>	
Dr. White	CSCI3233
Dr. Smith	CSCI4432
Dr. Brown	SWEN5232
Dr. Brown	SWEN5233

Semantics of one association: SET
i.e., no duplicate

Association Multiplicities—Exercise

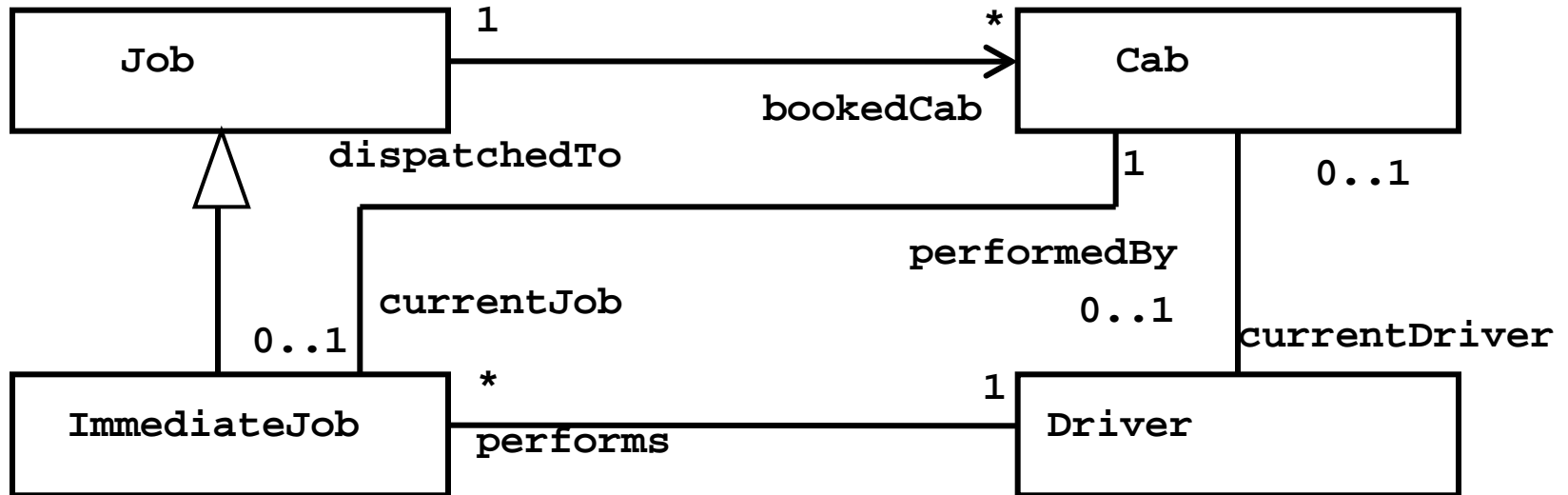


Multiplicities have Semantics



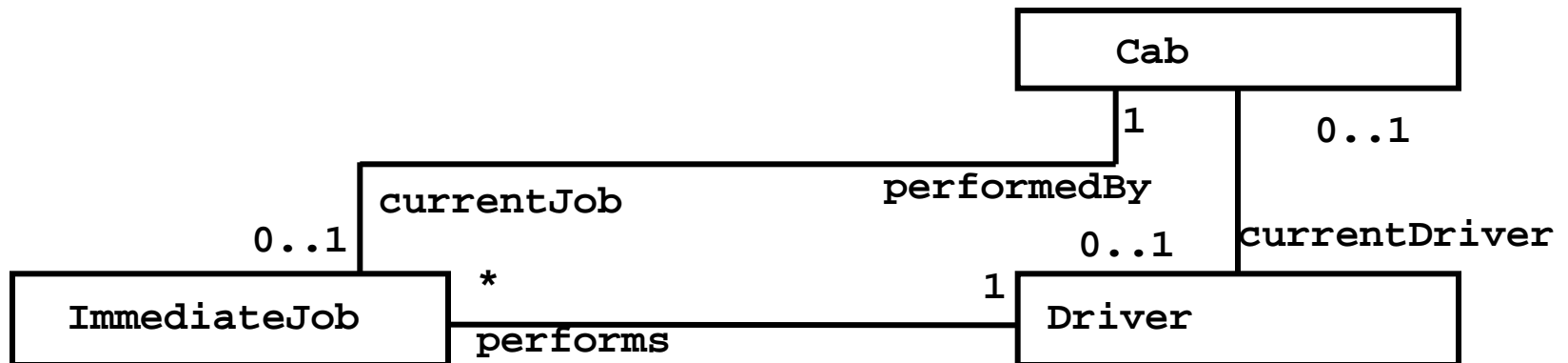
- A Job instance is linked to 0 or more Cab instances.
- A Cab instance is linked to exactly 1 Job instance.
- An ImmediateJob instance is linked to exactly one Cab instance.
- A Cab instance is linked to 0 or 1 (but not more) ImmediateJob instance.
- ...
- At what time during the execution of instances/objects do these conditions hold?

Multiplicities have Semantics (cont.)



- A Cab instance is linked to exactly 1 Job instance.
- A Cab instance is linked to 0 or 1 (but not more) ImmediateJob instance.
- **When?**
 - At the end of the construction of a Cab instance.
 - Before and after the execution of any public operation of Cab.
- **But not** (necessarily)
 - During the execution of the constructor or any public operation of Cab.
 - Before and after the execution of non-public operations of Cab.

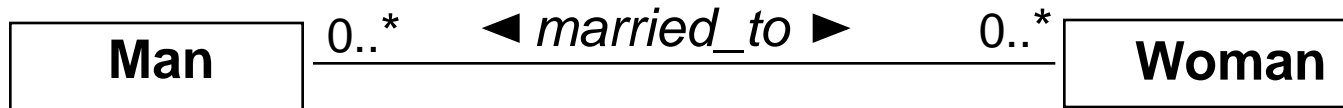
Multiplicities have Semantics (cont.)



- Multiplicities have huge consequences and **can easily be wrong**.
- An ImmediateJob is linked to exactly one Cab instance
- An ImmediateJob is linked to exactly one Driver instance
 - A Driver instance is linked to 0 or 1 Cab.
 - i.e., a Driver instance may not be linked to any Cab.
- Path ImmediateJob→Driver→Cab specifies that
 - An ImmediateJob is performed by a Driver but the Driver may not be in a Cab!
 - **There is a problem!**
- Different paths, when having the same semantics, must be consistent
 - We can call these paths *redundant paths*
- **During Analysis: avoid redundant paths!**

Multiplicity: Snapshot versus History ?

- What is the scope of multiplicity constraints ?
- Should they cover past, present and future or just present ?
- Answer depends on the context, and what you want to model

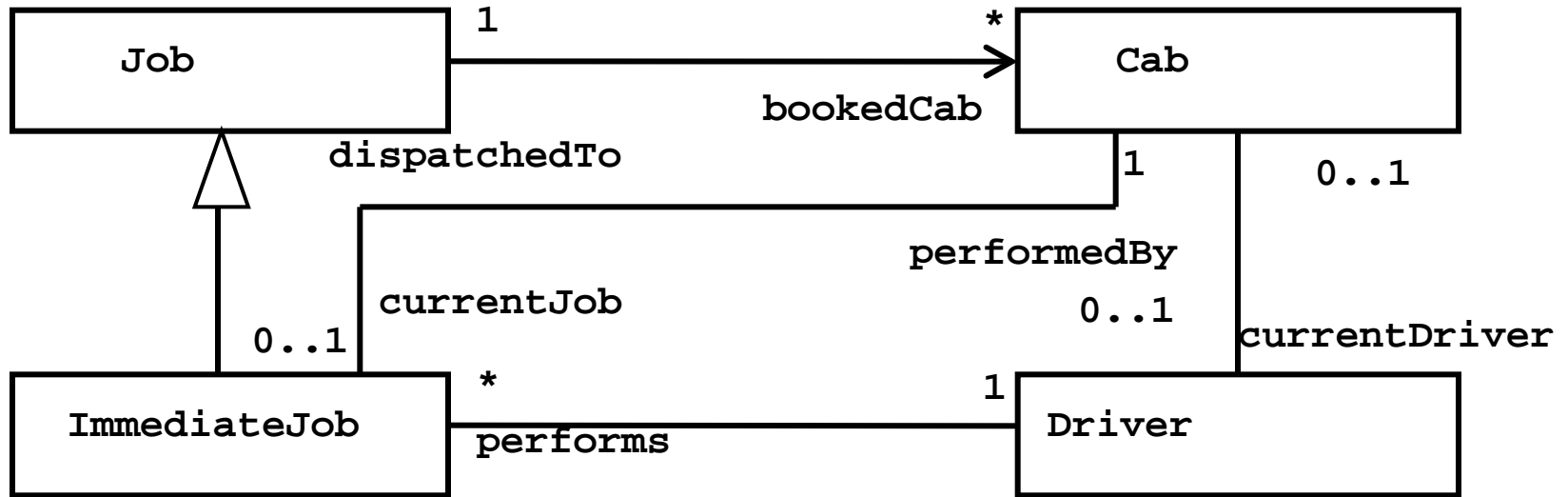


=> when a history of marriages is useful (e.g., police)



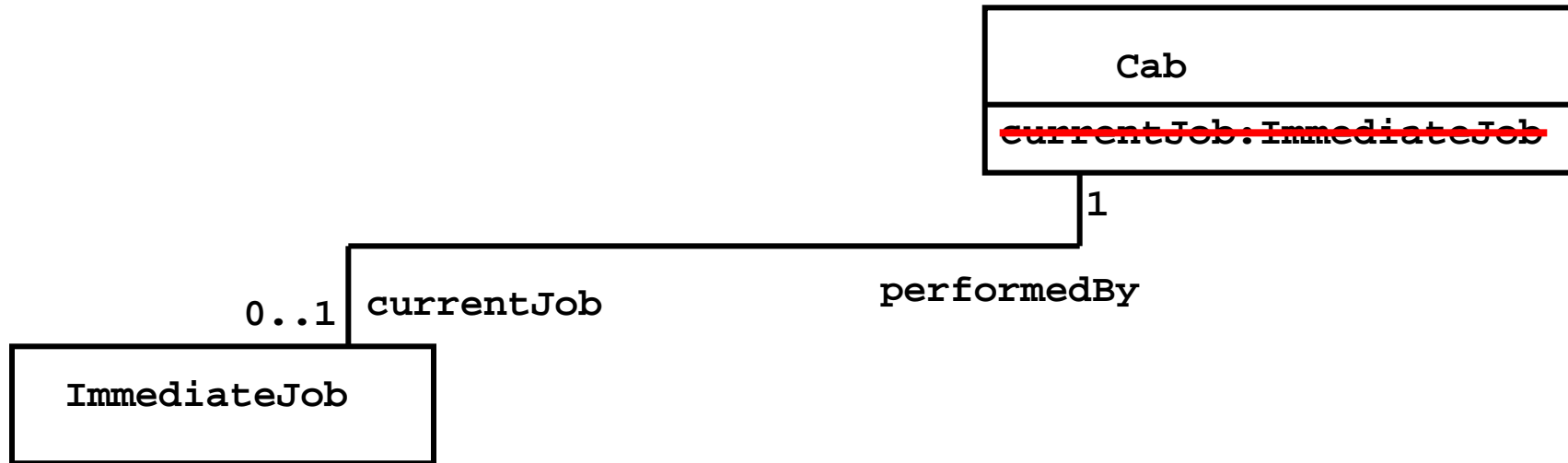
=> when only current marital status needed (e.g., bank)

Rolename == Attribute (during implementation)



- Class **ImmediateJob** has (implementation) an attribute of type **Cab** named **performedBy**.
- Class **Cab** has an attribute of type **ImmediateJob** named **currentJob**.
- Class **Job** has an attribute named **bookedCab**. Its type is a collection type (a Set). Due to multiplicity *.
- Class **Cab** does not have an attribute of class **Job** (due to the navigation of the association)

Rolename == Attribute (during implementation)



- Class Cab has
 - An attribute called currentJob (of type ImmediateJob) as part of its definition
 - An attribute called currentJob (of type ImmediateJob) because of its association with class ImmediateJob
 - **There is duplication of information!**
- Recall: attributes can only have primitive types (or enumeration, ... types). No other user-defined class types.

Association Classes (Fowler)



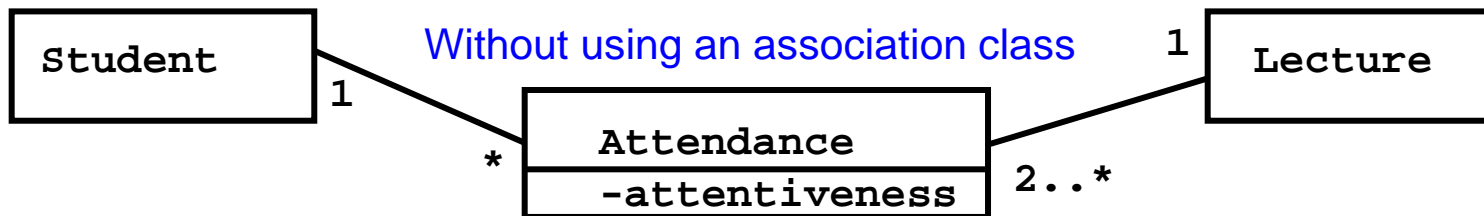
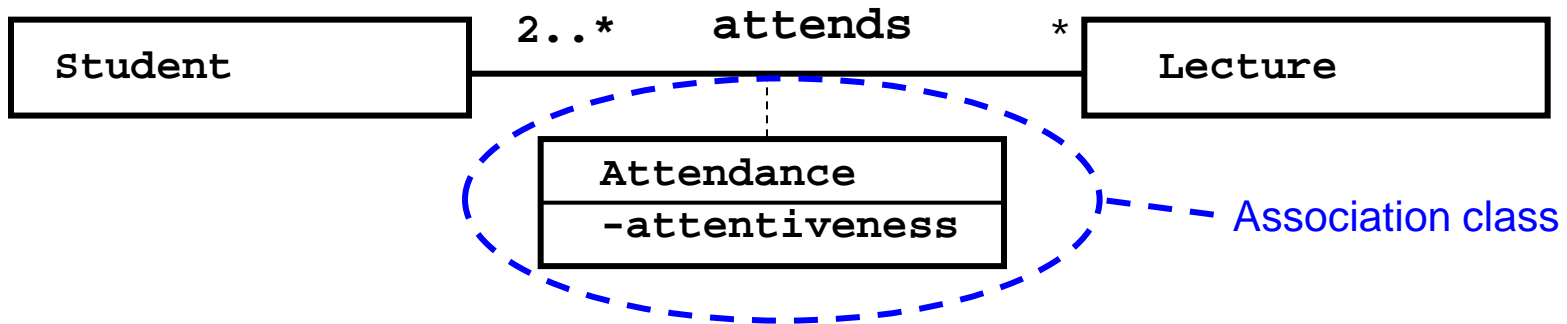
One wants to model the fact that students are more or less attentive.
Is this a property (attribute) of class Student?

- No: attentiveness depends on lectures

Is this a property (attribute) of class Lecture?

- No: attentiveness depends on students

Attentiveness is a property of the association Student—Lecture



Whole-Part Class Relationship

Four semantics possible

- ExclusiveOwns (e.g. Book has Chapter)
 - Existence-dependency (deleting a composite \Rightarrow deleting the components)
 - Transitivity
 - Asymmetry
 - Fixed property
- Owns (e.g. Car has Tire)
 - No fixed property
- Has (e.g. Division has Department)
 - No existence dependency
 - No fixed property
- Member (e.g. Meeting has Chairperson)
 - No special properties except membership



Composition



Aggregation
(shared in UML 2)

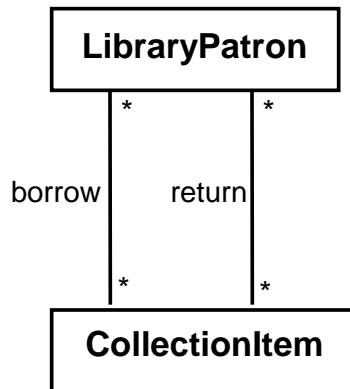


Whole-Part Class Relationship

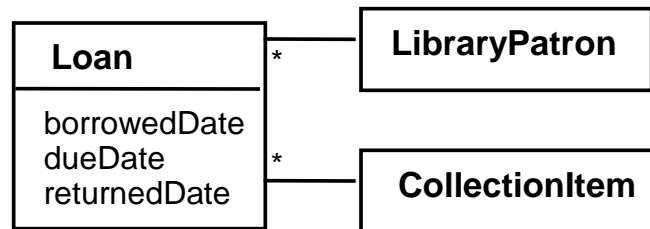
- Finding the right whole-part class relationship between class A and class B is context dependent
 - Depends on the specifics of your application
- For instance: Class Car is associated with class Tire
 - You have to build a software for a company building cars
 - From the point of view of this company, once the car is built, the link Car-Tire does not change
 - ExclusiveOwns 
 - You have to build a software for a company recycling cars
 - Tires in good shapes can be reused.
 - Has 

Association vs. Action

- A common mistake is to represent *actions* as if they were associations



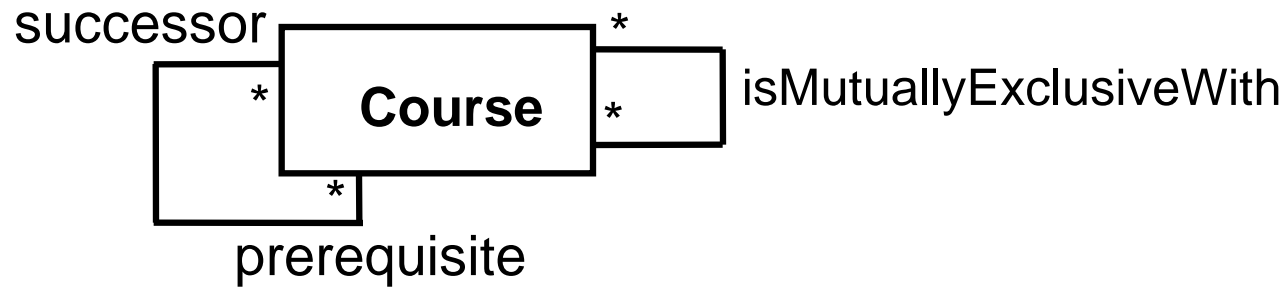
Bad, due to the use of associations that are actions



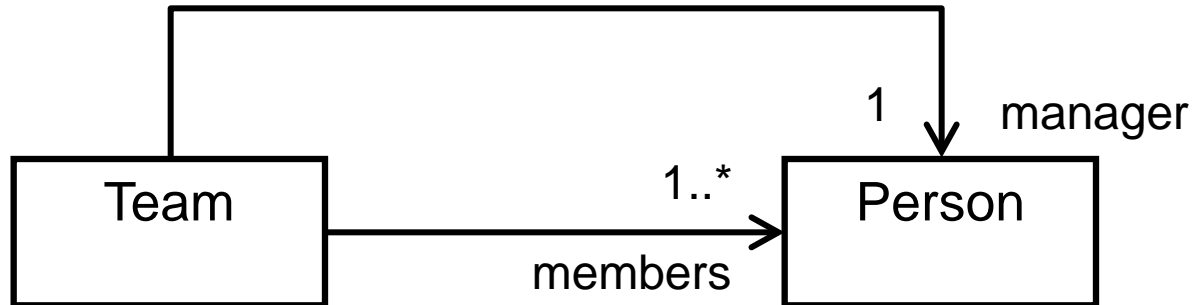
Better: The **borrow** operation creates a **Loan**, and the **return** operation sets the **returnedDate** attribute.

Reflexive associations

- It is possible for an association to connect a class to itself

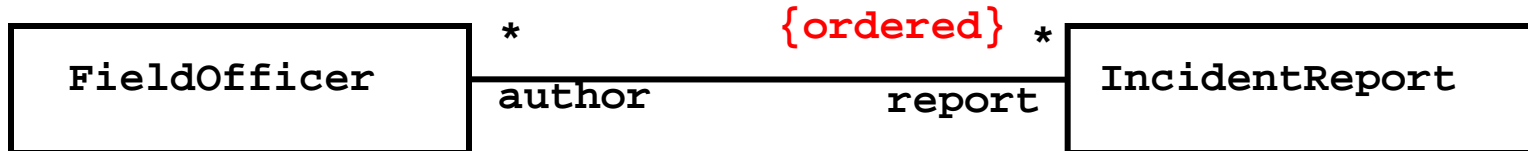


Role name mandatory in context



Constraining Associations: ordered

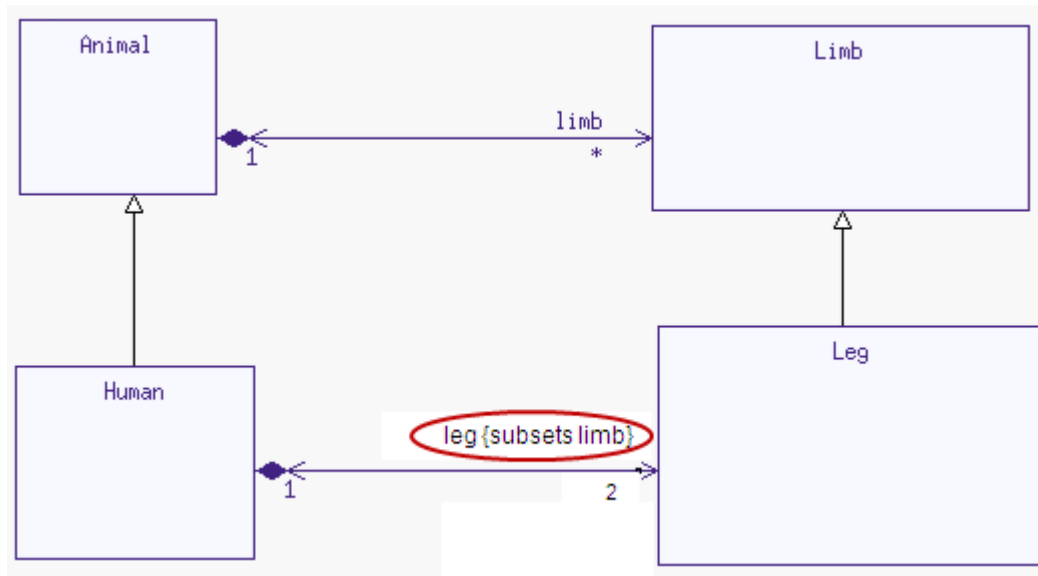
- Many different ways to specify a constraint on an association.
- One important one (see discussion on OCL): **{ordered}**
 - Specifies that the instances are ordered
 - Note: there is a difference between ordered and sorted
 - Sorting requires a sorting criterion (e.g., alphabetical)
 - Ordered simply means we have a first, second, ... last element.



- A FieldOfficer instance is linked to 0 or more IncidentReport instances, i.e., a set of IncidentReport instances.
- That set is ordered, i.e., it is a sequence and we can identify/access the first element of the set, the second, ... the last.
- No sorting criterion!

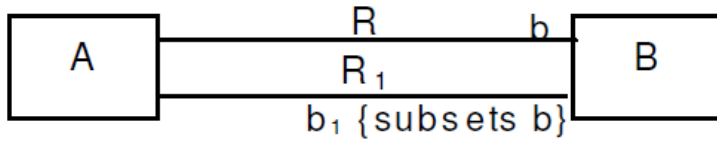
Constraining Associations: subsets

- Syntax `<end-name1> {subsets <end-name2>}`
 - placed near an association end `<end-name1>` (*the subsetting end*) indicates that it subsets `<end-name2>` (*the subsetted end*)
- Effect: the set of instances of an association is a subset of the set of instances of another association.
 - *inclusion constraint* between the subsetted and the subsetting associations
 - population of the subsetting end must be included in that of the subsetted end.

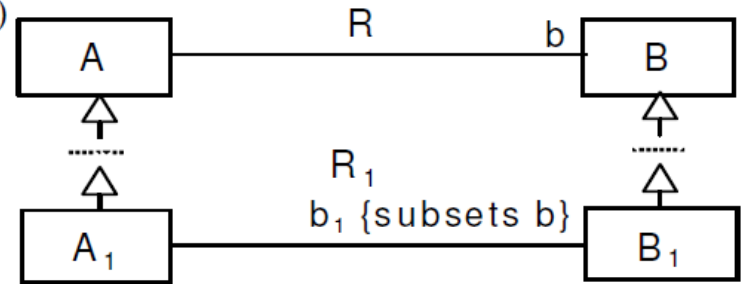


Association subsetting notation

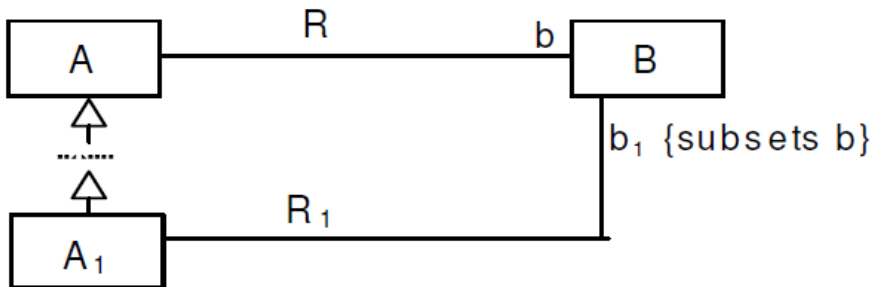
a)



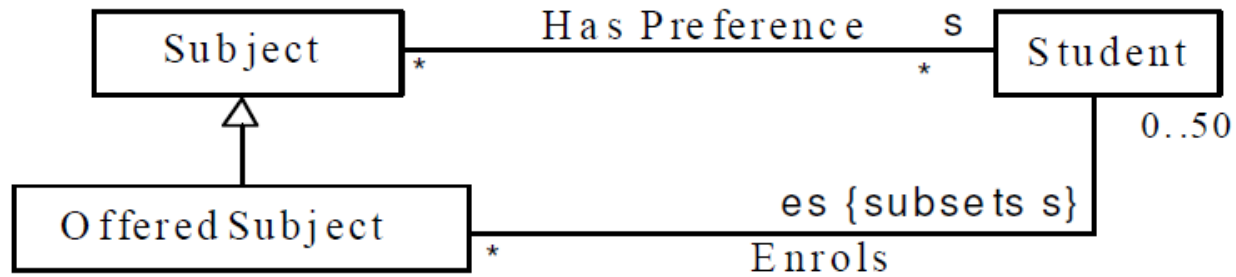
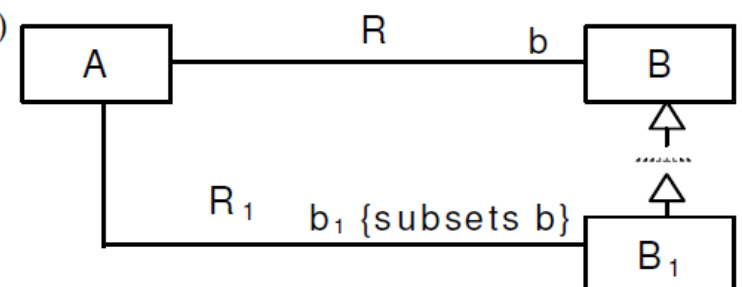
b)



c)



d)



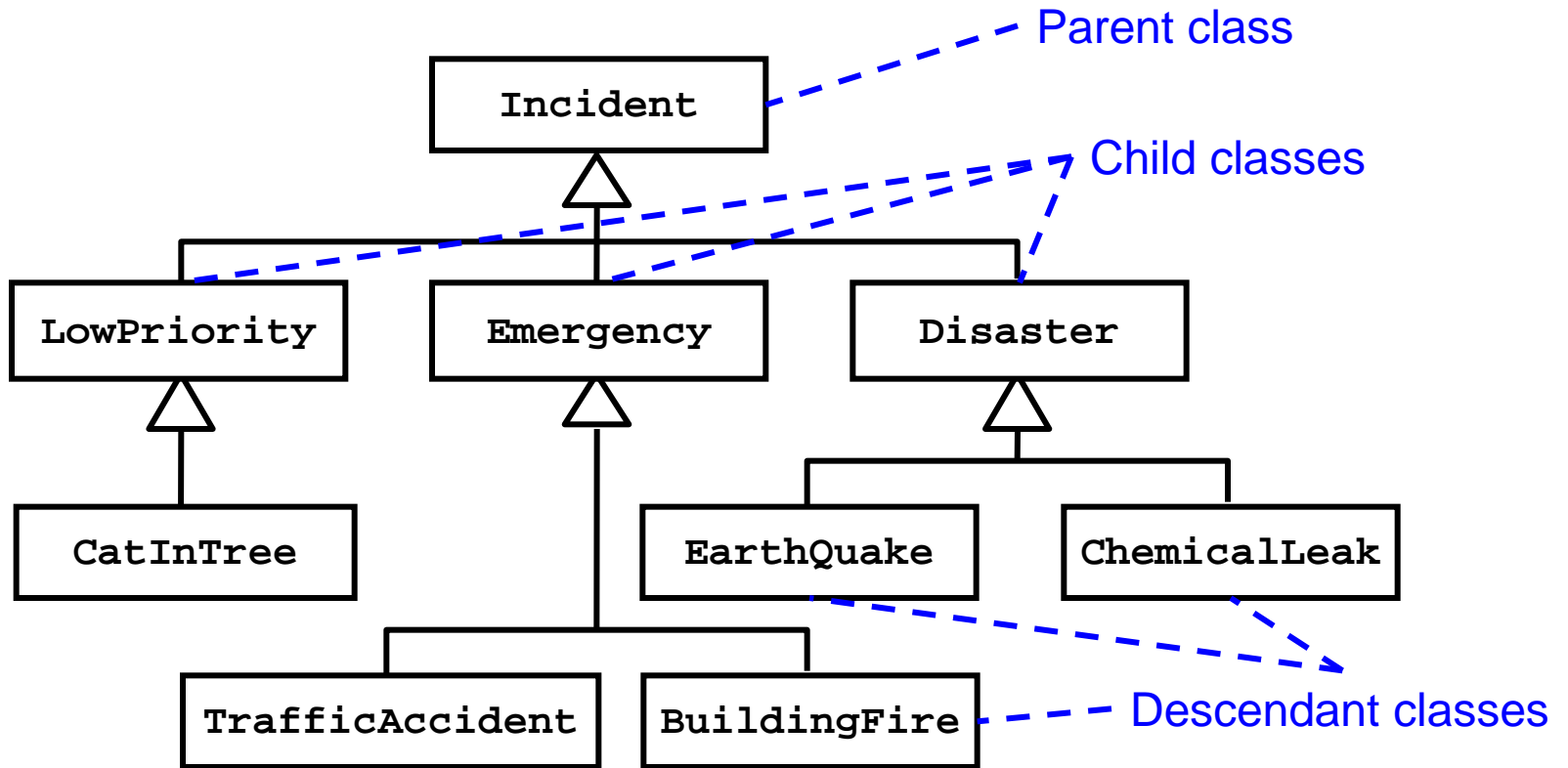
SYSC-3120—Software Requirements Engineering

- Overview
- Analysis Concepts
 - Modeling structure: class diagram
 - Modeling classes
 - Different kinds of relationships
 - Association
 - Generalization and Realization
 - Dependency
 - OCL: Better specifying operations/classes, constraining class diagram
 - Liskov substitution principle
 - Modeling interaction: sequence diagram
 - Modeling state-based behavior: state machine diagram
- Analysis Process
 - Finding objects/classes (heuristics)
 - ...

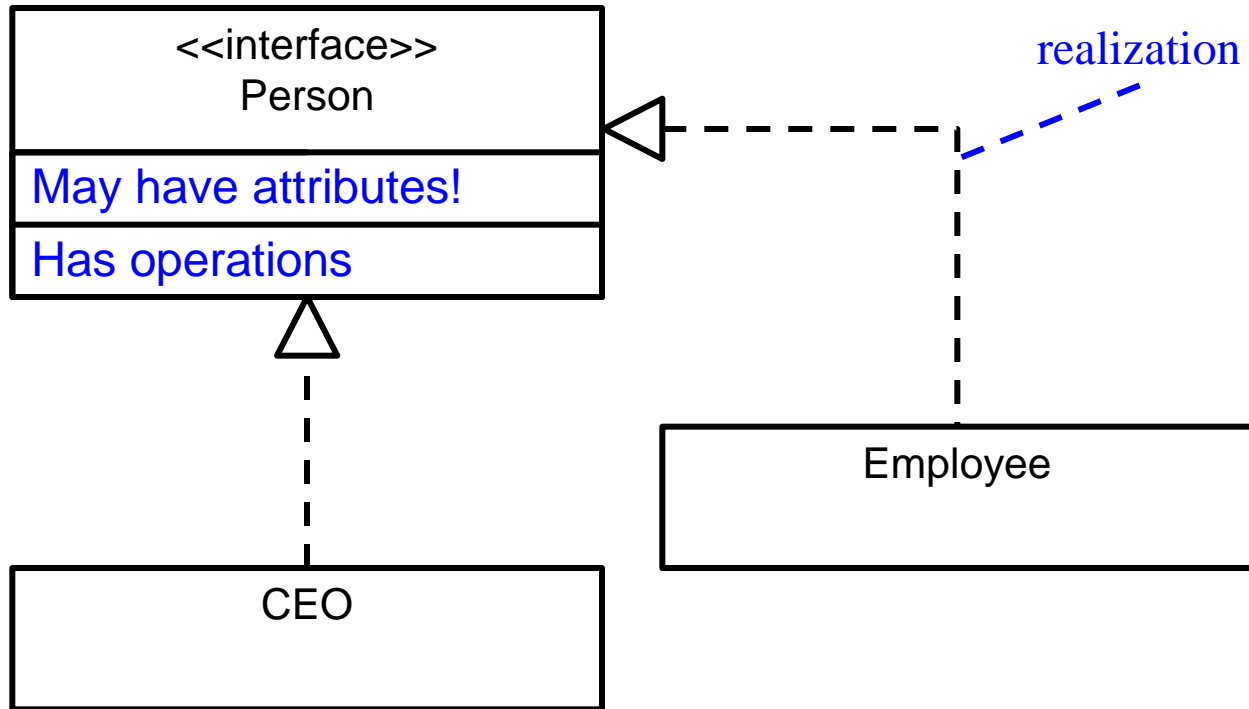
Generalization / Specialization

- The UML terminology for inheritance (almost)
- Common features abstracted into a more generic class
- Subclasses inherit (reuse) superclass features
 - Attributes
 - Operations
- Substitutability
 - Subclass object is a legal value for a variable whose type is the superclass
- Polymorphism
 - The same operation can have different implementations in different classes
- Abstract operation
 - Implementation provided in subclasses
- Abstract class
 - Class with no direct instance objects
 - A class with an abstract operation is abstract

FRIEND Example



Interface and Realization

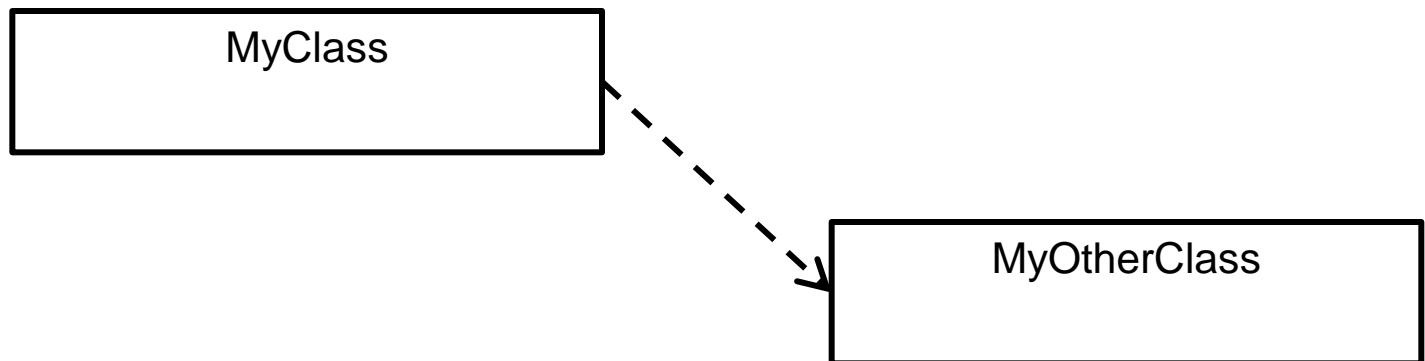


SYSC-3120—Software Requirements Engineering

- Overview
- Analysis Concepts
 - Modeling structure: class diagram
 - Modeling classes
 - Different kinds of relationships
 - Association
 - Generalization and Realization
 - Dependency
 - OCL: Better specifying operations/classes, constraining class diagram
 - Liskov substitution principle
 - Modeling interaction: sequence diagram
 - Modeling state-based behavior: state machine diagram
- Analysis Process
 - Finding objects/classes (heuristics)
 - ...

Dependency

- Use a dependency when
 1. A client class needs to call services on a server class
 2. You do not have (i.e., need) an association (i.e., attribute) between the client and the server
 - i.e., the interaction is transient
- In other words, there is no need for an association but instances of the two classes need to communicate with one another anyway.



Association vs. Dependency

- When do one needs an association?
- When do one needs a dependency?

- Association:
 - When a link between two objects has to survive the end of execution of an operation.
- Dependency:
 - When a link between two objects does not need to survive the execution of an operation (transient).

SYSC-3120—Software Requirements Engineering

- Overview
- Analysis Concepts
 - Modeling structure: class diagram
 - Modeling classes
 - Different kinds of relationships
 - OCL: Better specifying operations/classes, constraining class diagram
 - Liskov substitution principle
 - Modeling interaction: sequence diagram
 - Modeling state-based behavior: state machine diagram
- Analysis Process
 - Finding objects/classes (heuristics)
 - Finding relationships : associations and attributes (heuristics)
 - ...

Constraints: Motivations

- Constraints on UML model elements: conditions that must be true about some aspect of the system
 - Recall the ImmediateJob/Driver/Cab example
- Examples:
 - Precisely specifying operations
 - An operation's input must satisfy a specific condition
 - Precisely specifying conditions in other diagrams
 - Conditions in sequence diagrams, in state machine diagrams
 - Constraining class diagram
 - In the CarMatch system, an individual may not have more than 10 agreements for car sharing
 - When an ImmediateJob is created, it is linked to a Car instance and that Car instance must be linked to a Driver instance.

Constraints: Motivations

- Precise constraints make the analysis (and design) more precise and rigorous
- Complement the UML graphical notation by associating properties with model elements
 - e.g., methods, classes, transitions
- Help verification and validation of models

- Different types of constraints
 - *Contracts* for operations
 - *Conditions* in sequence diagrams (see later)
 - *Guard* conditions in state machine diagrams (see later)
 - Class and state *invariants* in state machine diagrams (see later)

Analysis (and Design) by Contract

- Specifying software \approx specifying operation-to-operation communications
 - What are the responsibilities and rights of communicating operations?
 - Formalized by Design by Contract (DbC)
 - Operation's precondition:
 - binds the client,
 - defines the responsibility of any (other) operation calling this operation.
 - defines the rights of the operation being called, i.e., it has the right to expect that the condition(s) stated in its precondition hold
 - Operation's postcondition:
 - binds the method being specified,
 - defines the conditions that must be ensured by this operation at the end of its execution.
 - defines the rights of the calling operation, i.e., it has the right to expect that, assuming the condition(s) stated in the precondition of the called operation were satisfied, the conditions stated in the called operation's postcondition hold
-

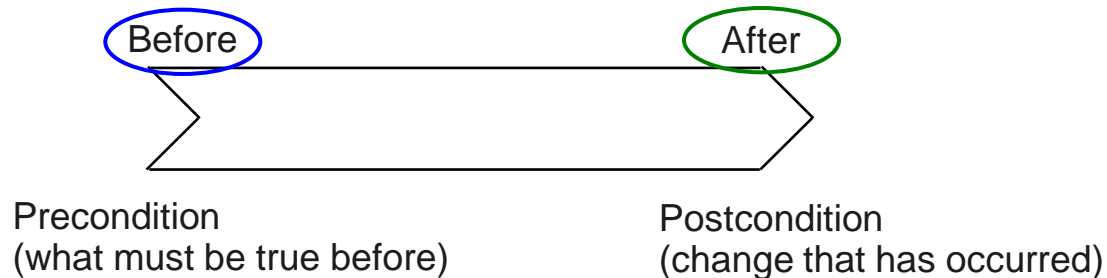
Design by Contract (Rights and Responsibilities)

	Obligations/Responsibilities	Benefits/Rights
Operation performing the call	To satisfy the precondition of the operation being called.	Expects/relies on what is stated in the postcondition of the operation being called
Operation being called	To ensure conditions stated in the postcondition hold.	Can assume it is called while its precondition holds.

Design by Contract (Rights and Responsibilities)

```
Contractor :: put (element: T, key: STRING)
-- insert element x with given key
```

	Obligations	Benefits
Client	Call put only on a non-full table	Get modified table in which x is associated with key
Contractor	Insert x so that it may be retrieved through key	No need to deal with the case in which the table is full before insertion

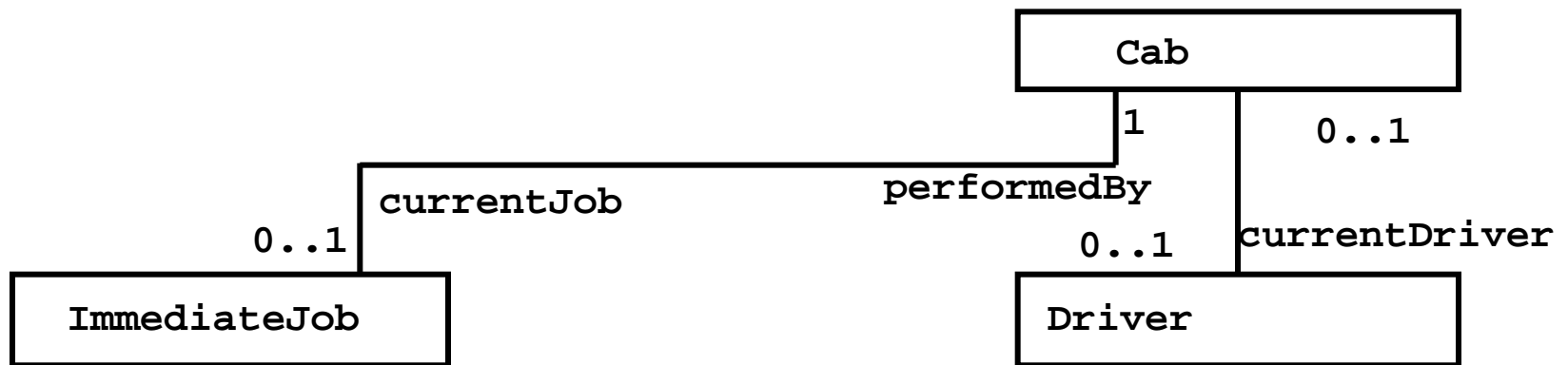


Design by Contract (Rights and Responsibilities)

- Precondition: what must be true before
 - In terms of parameter values
 - In terms of system state
 - E.g., other objects' attribute's values
 - E.g., links between objects
 - In terms of relations between those
- Postcondition: what must be true after / changes that occurred
 - In terms of return value
 - In terms of out/inout parameters
 - In terms of system state
 - In terms of relations between those
 - In terms of relations to what was true before

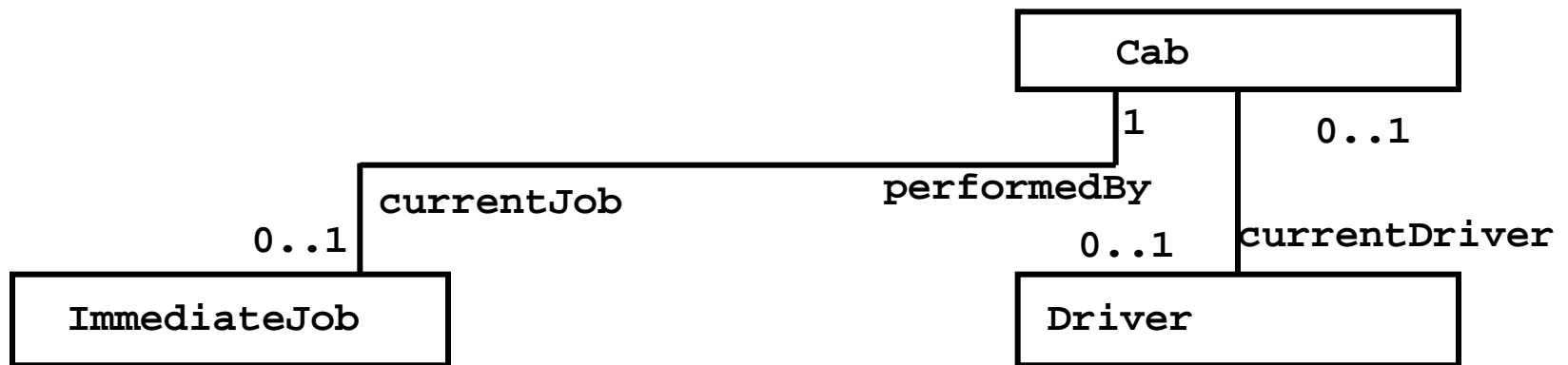
Design by Contract (constraining object relations)

- Class diagram notation insufficient to specify everything we need to specify
- Recall the ImmediateJob-Cab-Driver example
 - Multiplicities are adequate: a Cab may not have a Driver (lower bound of 0)
 - But a legal instantiation (according to multiplicities) does not necessarily make sense: an ImmediateJob is linked to a Cab, which is linked to no Driver.
- No UML class diagram notation to specify that:
 - When a Cab is linked to an ImmediateJob, then it must also be linked to a Driver.



Design by Contract (constraining object relations)

- No UML class diagram notation to specify that:
 - When a Cab is linked to an ImmediateJob, then it must also be linked to a Driver.
- What is this property?
 - This is not a property (pre/post) of any operation in those classes
 - This is a property of any instance of Cab
 - This is a property of class Cab: an (class) **invariant**



Class Invariant

- Condition that must always be met by all instances of a class
- Described using a Boolean expression that evaluates to true if the invariant is met
- Invariants must be true all the time,
 - except during the execution of an operation where the invariant can be temporarily violated.
- If the pre- and post-conditions are satisfied, then the class invariant must be preserved by an operation
- A violated invariant suggests an illegal system state

- Note: multiplicities are in fact invariants.

Usage of Contracts/Constraints

- Analysis/Design:
 - Understand and document clearly operations' intent and purpose
 - Understand and document clearly class characteristics
- Coding:
 - Guide programmer to an appropriate implementation (i.e. method)
 - Black-box specification of methods
- System robustness:
 - Check invariants and pre-conditions at run time before the execution of operations, possibly raise exceptions and display error messages.
 - Check invariants and post-conditions at the end of method executions.
- Testing:
 - Verify that the method does what was originally intended
 - Help debugging.

Object Constraint Language (OCL)

- Part of the UML standard
- Formal, mathematical language
- Inspired by the work on formal specification methods
- Not a programming language but a typed, declarative language
- OCL contains set operators and logic connectives.
 - Based on Set theory and first order logic

OCL Expression—Context

- An OCL expression needs a **context**
 - Determines the model element being constrained
 - Determines the scope of the expression
 - Determines what (named model element) can be accessed in the expression.
- We will consider two kinds of context
 - A class, to define an invariant
 - Rather: any instance of the class
 - An operation, to define a precondition, a postcondition
- Syntax: (Class name can be fully qualified to account for package information.)

context *ClassName*

inv: *OCL-expression*

context *ClassName::operationName(...)*

pre: *OCL-expression*

post: *OCL-expression*

OCL Expression—Context

- An OCL expression needs a **context**
 - What (named model element) can be accessed in the expression?

Class context (rather instance context)

- Class scope attributes
- Instance scope attributes
- Rolenames of associations
- Including what is inherited
- Navigation through the class (rather instance) diagram
- Other class names

Operation context

- Class scope attributes
- Instance scope attributes
- Rolenames of associations
- Including what is inherited
- Navigation through the class (rather instance) diagram
- Other class names
- Operation parameters
- Values the above had before the execution of the operation
- Operation return value

Class/Instance Scope Model Element

- Context (either class or operation) gives access to
 - Class scope attributes
 - Instance scope attributes
 - Role name of associations (at the other end of the associations from the context)
 - Including what is inherited
- **self** refers to the instance of the context

context SavingsAccount

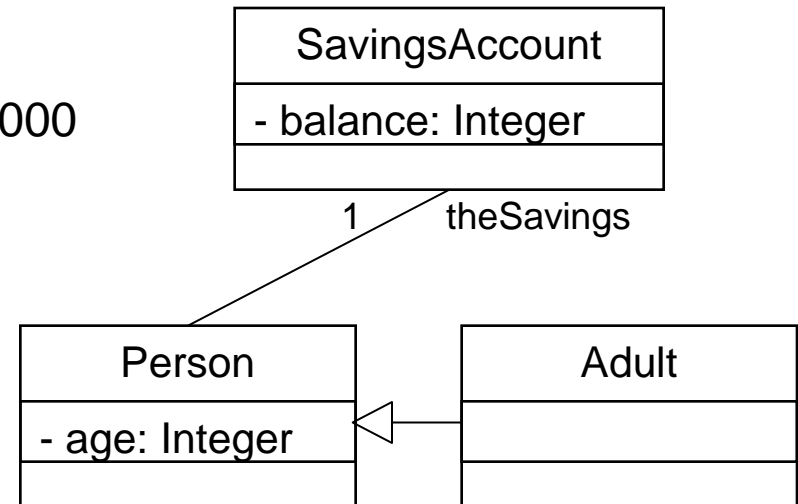
inv: self.balance>0 and self.balance<25000

context Adult

inv: self.age>=18

context Person

inv: self.theSavings = ...



OCL Types

- OCL is a typed language
- Available types?
 - Primitive types: Boolean, Real, Integer, String
 - Collection types: Set, OrderedSet, Bag, Ordered Bag
 - *Set*: cannot contain duplicate items
 - *Bag*: can contain duplicate items
 - *OrderedSet*: elements of a set are ordered, cannot contain duplicate items (recall the `{ordered}` constraint)
 - *OrderedBag*, a.k.a., *Sequence*, elements of a bag are ordered, can contain duplicate items (recall the `{ordered}` constraint)
 - Any user-defined class from the class diagram
- Predefined types (primitives and collection types) have predefined operations

Boolean Type

a	b	a = b	a <> b	a and b	a or b	a xor b	a implies b (not (a) or b)	not(a)
true	true	true	false	true	true	false	true	false
true	false	false	true	false	true	true	false	false
false	true	false	true	false	true	true	true	true
false	false	true	false	false	false	false	true	true

- The '=' is a first order logic equality, not the assignment '=' of programming languages.

Collection Types

Operation	Description
size	The number of elements in the collection
count(object)	The number of occurrences of object in the collection.
includes(object)	True if the object is an element of the collection.
includesAll(collection)	True if all elements of the parameter collection are present in the current collection.
isEmpty	True if the collection contains no elements.
notEmpty	True if the collection contains one or more elements.
iterate(expression)	Expression is evaluated for every element in the collection.
sum	The addition of all elements in the collection.
exists(expression)	True if expression is true for at least one element in the collection.
forAll(expression)	True if expression is true for all elements.

- When using a collection operation, use the **invocation operator** “->”
 - e.g., aColl->size() > 0
 - e.g., aColl->isEmpty = false

Collection Types

Operation	Set	OrderedSet	Bag	Sequence	Description
=, <>	X	X	X	X	Equal, not equal
-	X	X	-	-	{1,4,6} - {4,5}={6}
symetricDifference(coll)	X	-	-	-	{1,4,6} symDiff {4,5}={1,5,6}
append(object), prepend(object)	-	X	-	X	Self-explanatory
asBag()	X	X	X	X	Results in a bag collection
asOrderedSet()	X	X	X	X	Removes duplicates, order is random
asSequence()	X	X	X	X	Order is random
asSet()	X	X	X	X	Removes duplicates
excluding(object)	X	X	X	X	Removes the object from the collection (set) or any duplicate of it (bag)
first(), last()	-	X	-	X	Self-explanatory
including(object)	X	X	X	X	Adding an element
at(index), insertAt(index, object)	-	X	-	X	Self-explanatory
intersection(coll)	X	-	X	-	Self-explanatory
union(coll)	X	X	X	X	Self-explanatory

Navigation Expression

- Navigation is the process whereby you follow links from a source object to one or more target objects.
- OCL navigation expressions can refer to any of the following:
 - Classes and interfaces
 - Attributes
 - Association ends
 - Query operations
 - these are operations that have the property isQuery set to true.
 - they do not change attribute values, contents of collection, ...
 - i.e., no side effect when calling them.
 - e.g., getName()

Navigation Expression

- To navigate associations, use:
 - Class name or rolename
 - Using the dot notation
 - e.g., self.roleName1.roleName2
- No difference between composition, aggregation and associations as far as navigation expressions are concerned
- No difference whether association is navigable or not
 - We specify a condition/constraint on instances
- No difference whether attributes are private or not
- Can traverse several associations in one expression

Navigation—Simple Examples

context SavingsAccount

inv: self.balance>0 and self.balance<25000

context Adult

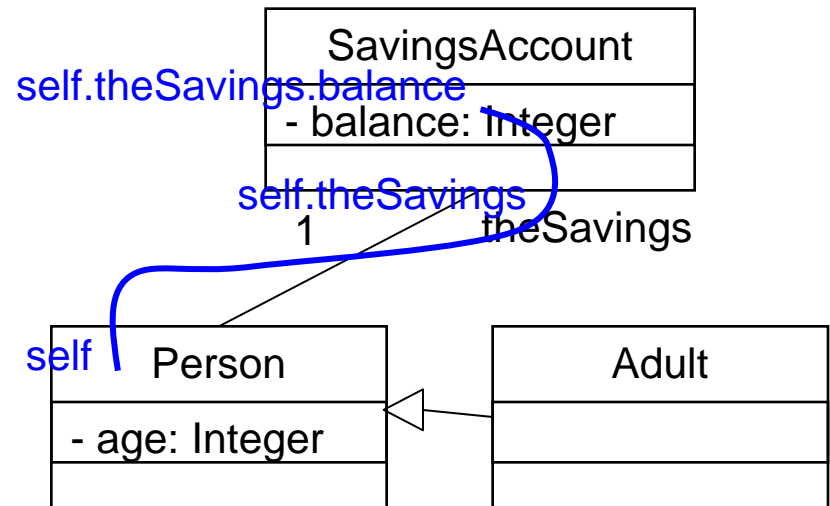
inv: self.age>=18

context Person

inv: self.theSavings -> size()==1

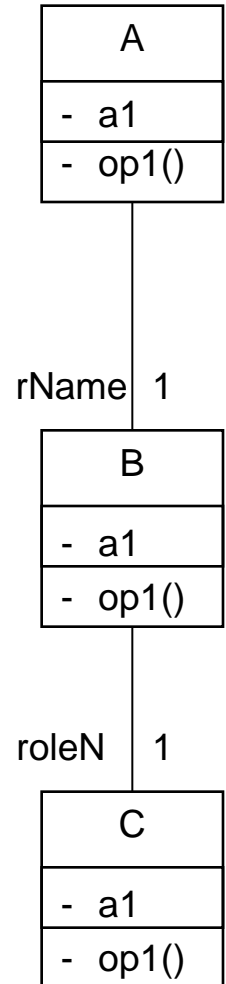
context Person

inv: self.theSavings.balance > 0 and ...



Navigation Expressions

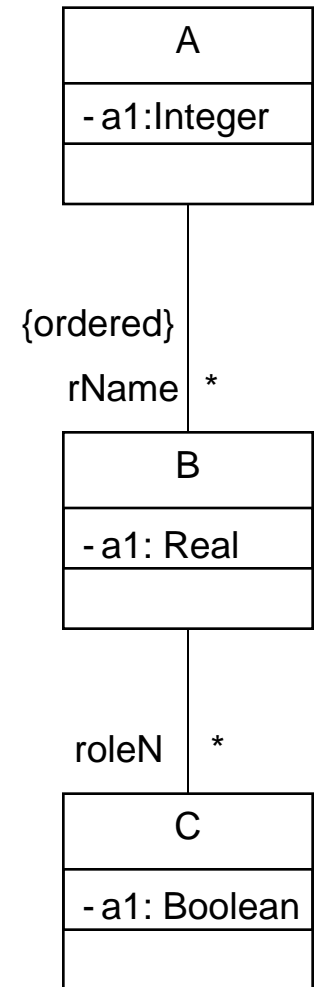
Context	Navigation expression	Semantics
A	self	The contextual instance, an instance of class A.
	self.a1	The value of attribute a1 of the contextual instance
	self.op1()	The result of invoking op1() on the contextual instance. Must not have any side effect.
	self.rName	The instance of class B linked to the contextual instance. One (and only one) instance of B because of multiplicity 1.
	self.B	Same as self.rName. (Requires that we have only one association between A and B.)
	self.rName.a1	The value of attribute a1 of the B instance linked to the contextual instance.
	self.rName.op1()	The result of invoking op1() on the B instance linked to the contextual instance. Must not have any side effect.
	self.rName.roleN	The instance of C linked to the instance of B linked to the contextual instance. One (and only one) instance of C because of multiplicities 1.



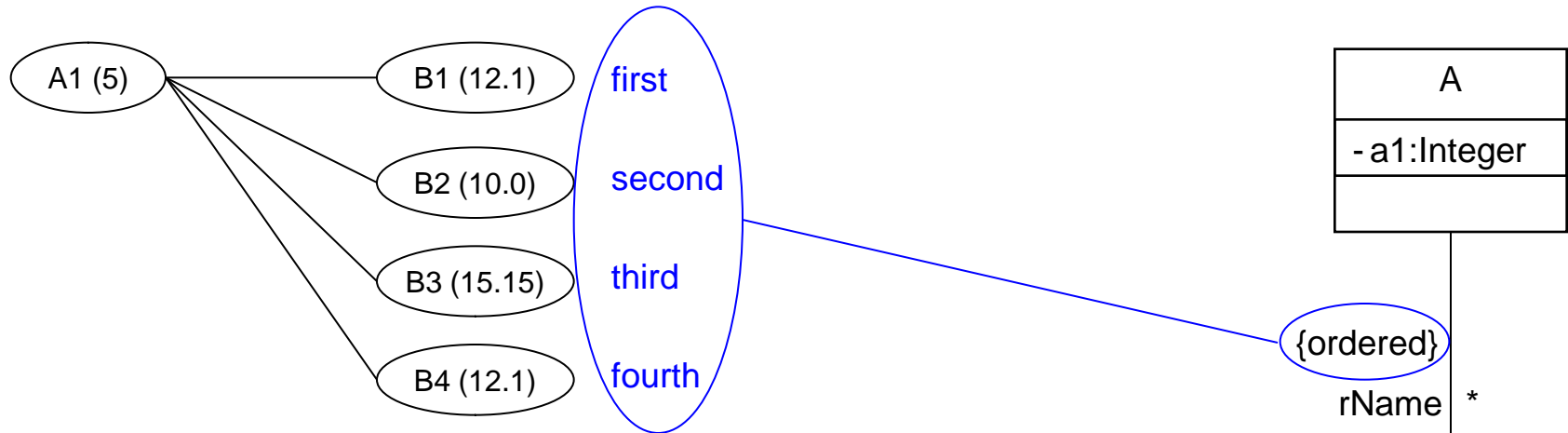
Result of Navigation—Collection

- The result of a navigation is:
 - An instance if navigating associations with multiplicity one only
 - A collection otherwise
 - Navigating only one association: a Set or an OrderedSet
 - Navigating two (or more) associations: a Bag or a Sequence

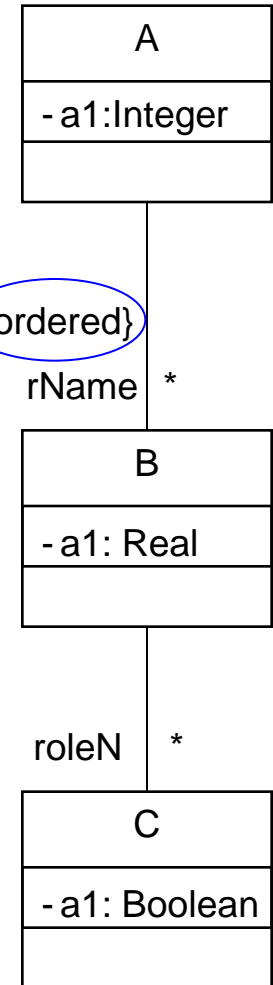
Context	Navigation expression	Resulting type
A	self	An instance of A
A	self.a1	One integer value
A	self.rName	An OrderedSet of B instances
B	self.roleN	A Set of C instances
A	self.rName.a1	A Sequence of Real values
A	self.rName.roleN	A Bag of C instances



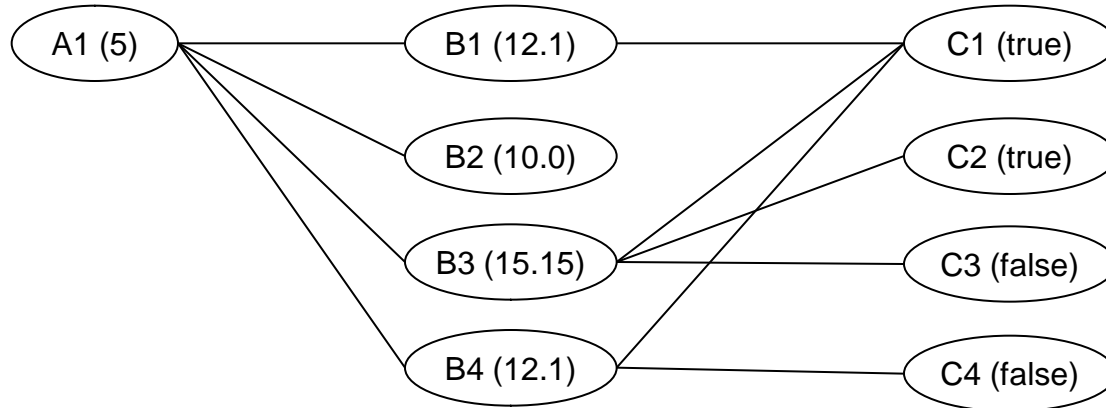
Result of Navigation—Collection



Navigation expression	Result	Resulting type
self	A1	An instance of A
self.a1	5	One integer value
self.rName	{B1, B2, B3, B4}	An OrderedSet (no duplicate) of B instances
self.rName.a1	{12.1, 10.0, 15.15, 12.1}	A Sequence (may have duplicates) of Real values



Result of Navigation—Collection

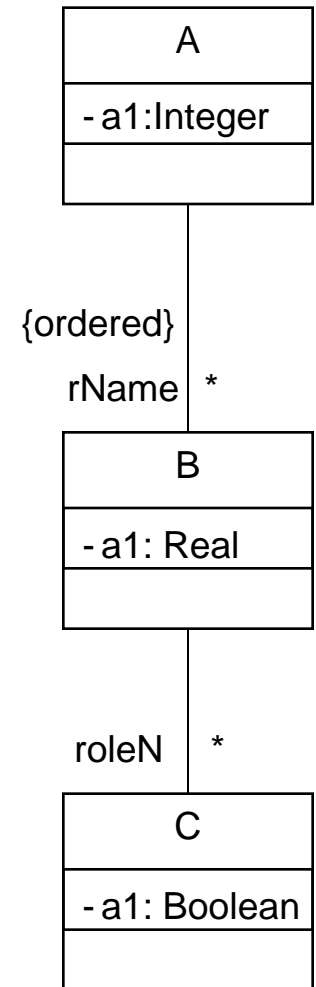


Navigation expression	Result	Resulting type
self.rName.roleN	{C1, C1, C2, C3, C1, C4}	A Bag (no order, can have duplicates) of C instances

$\text{self.rName.roleN} = \text{B1.roleN} \cup \text{B2.roleN} \cup \text{B3.roleN} \cup \text{B4.roleN}$
 $= \{C1\} \cup \{C1, C2, C3\} \cup \{C1, C4\}$

(Not the OCL notation!)

If you want to remove duplicates: `self.rName.roleN->asSet()`



Navigating—Association Classes, Generalizations, Enumerations

- How to navigate to/from an association class?

Context Student

self.attends ...

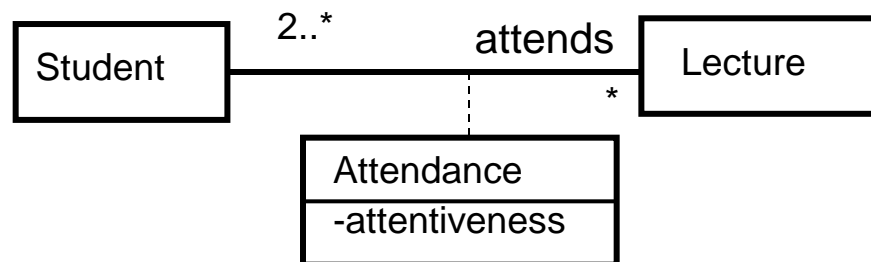
self.Attendance ... (collection of instances)

Context Attendance

self.Student ... (one instance)

self.Lecture ... (one instance)

self.attends ... (one instance)



Navigating—Association Classes, Generalizations, Enumerations

- How to navigate to descendants?

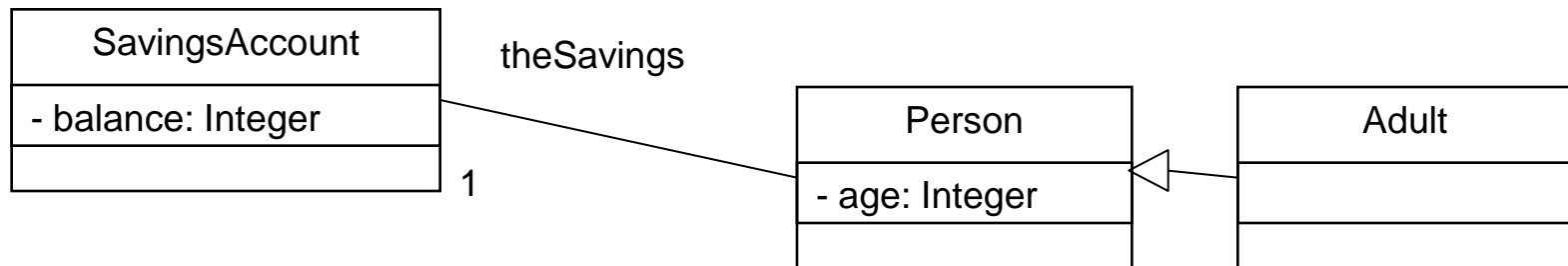
Context SavingsAccount

self.Person

(the collection of Person instances linked to this SavingsAccount, including Adult instances)

self.Adult

(the collection of Adult instances which are in the self.Person collection—subset)

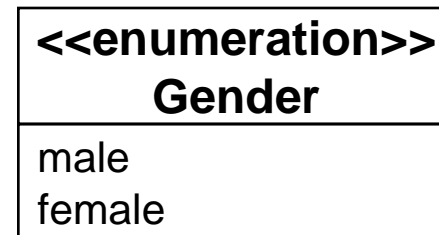
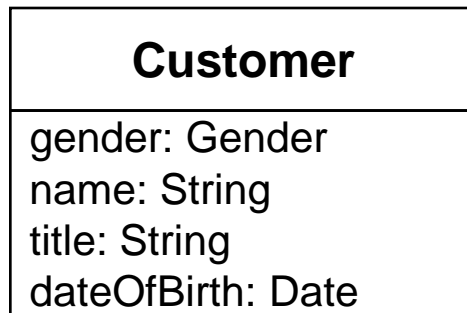


Navigating—Association Classes, Generalizations, Enumerations

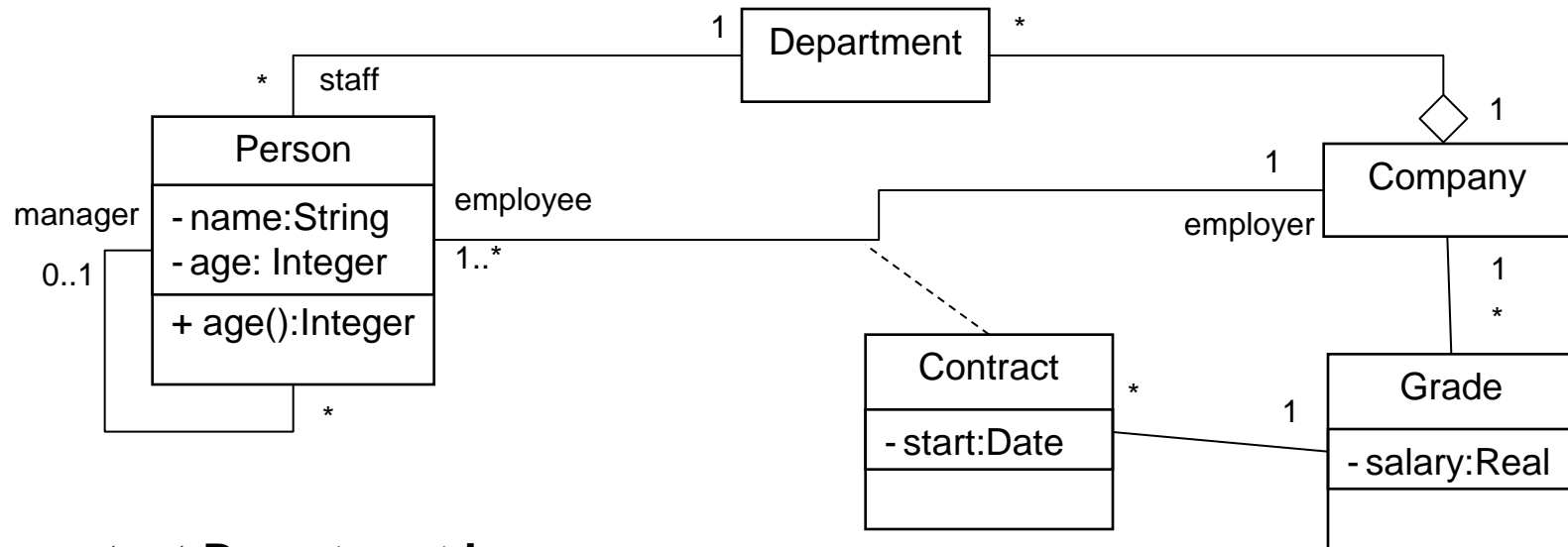
- How to use enumeration values in an OCL expression?

context Customer **inv** :

gender = Gender::male implies title = 'Mr.'



Example Class Diagram



context Department inv:

`staff.Contract.Grade.salary->sum()`

The total amount of money spent on salaries in the department.

context Department inv:

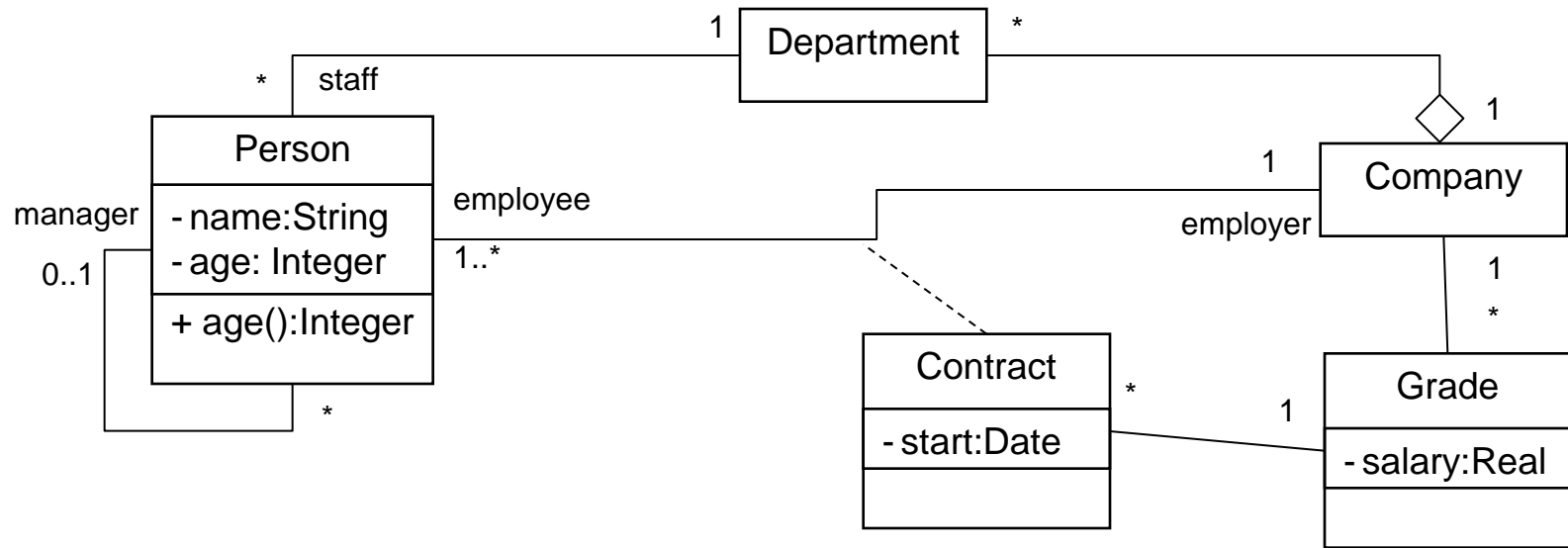
`staff.Contract.Grade->asSet()->size()`

The total number of different salary grades for the personnel in the department.

Subset Selection

- Sometimes necessary to consider only a subset of objects returned by a navigation
`navigationExpression->select (contextForExpression | BooleanExpression)`
- Operation “select” applies a Boolean expression to each object of a collection and returns those objects for which the expression is true
- Declaration of local variable: Context for navigation in the Boolean expression.
- Alternative: rejecting some instances
`navigationExpression->reject (contextForExpression | BooleanExpression)`

Subset Selection



context Company

```
self.employee->select(p:Person | p.Contract.Grade.salary > 50000)
```

Which employees are paid that much?

context Company

```
employee->select(p:Person | p.Contract.Grade.salary > 50000).manager->asSet()
```

Which managers allowed such salaries? 😊

Subset Selection

The context in the select operation is optional.
But it is strongly recommended to have it.

context Company

```
self.employee->select(Contract.Grade.salary > 50000)
```

context Company

```
employee->select(Contract.Grade.salary > 50000).manager->asSet()
```

Subset Selection

Select vs. Reject

context Company

```
self.employee->select(Contract.Grade.salary > 50000)
```

context Company

```
self.employee->reject( not (Contract.Grade.salary > 50000) )
```

context Company

```
self.employee->reject(Contract.Grade.salary <= 50000)
```


Creating a Collection

- Sometimes necessary to create a collection of objects by collecting data from an existing collection
`navigationExpression->collect (contextForExpression | navigationExpression)`
- Takes a navigation expression as argument and returns a bag consisting of the values of the expression for each object in the original collection

context Department **inv:**

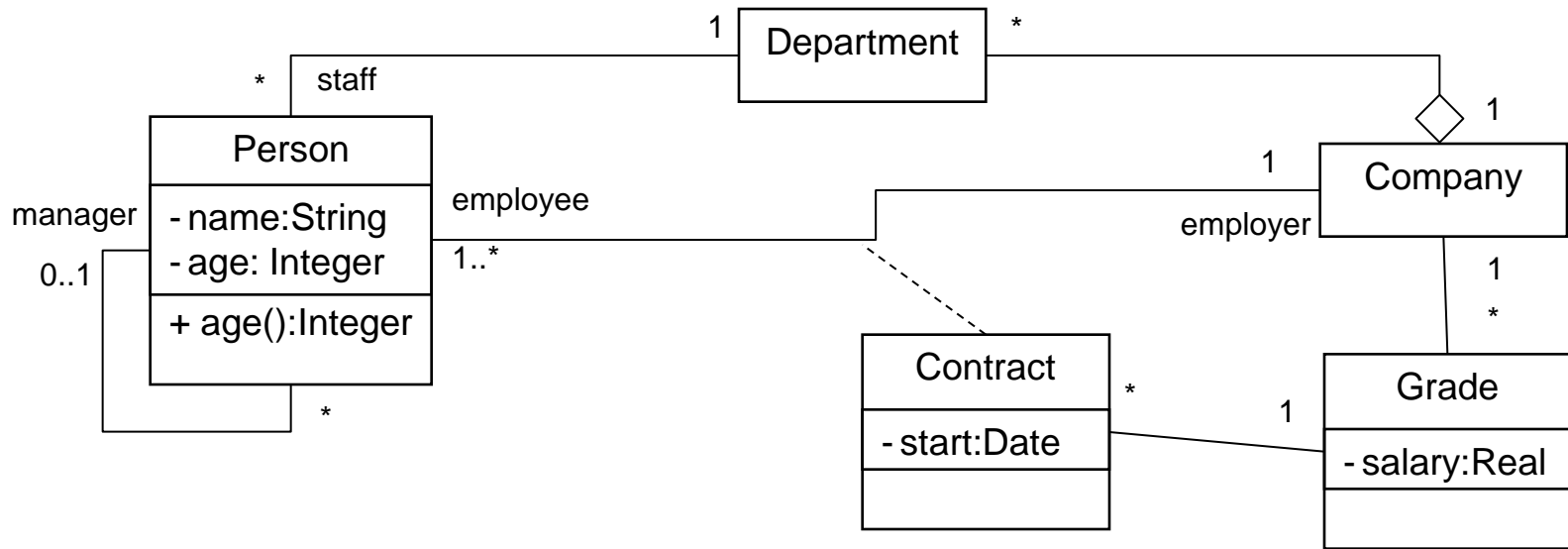
```
self.staff->collect(p:Person | p.age())
```

- Expression can perform additional calculations

context Company **inv:**

```
self.Contract.Grade->collect(g:Grade | salary*1.1)  
->sum()
```

Creating a Collection



context Department

```
self.staff->collect(p:Person | p.age())
```

The bag of age values for the staff of the department.

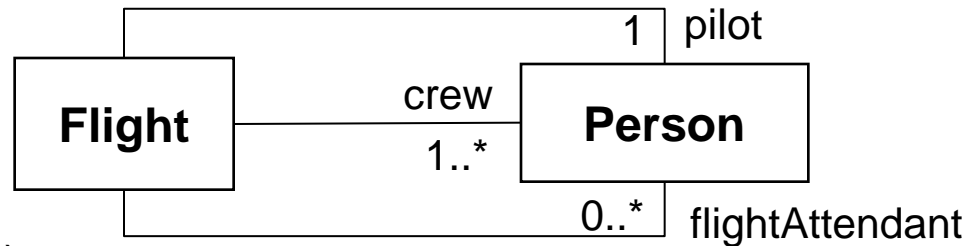
context Company

```
self.Contract.Grade->collect(g:Grade | salary*1.1)->sum()
```

The expression within the collect(...) can perform a computation.

Checking Collection Contents: includes(), includesAll()

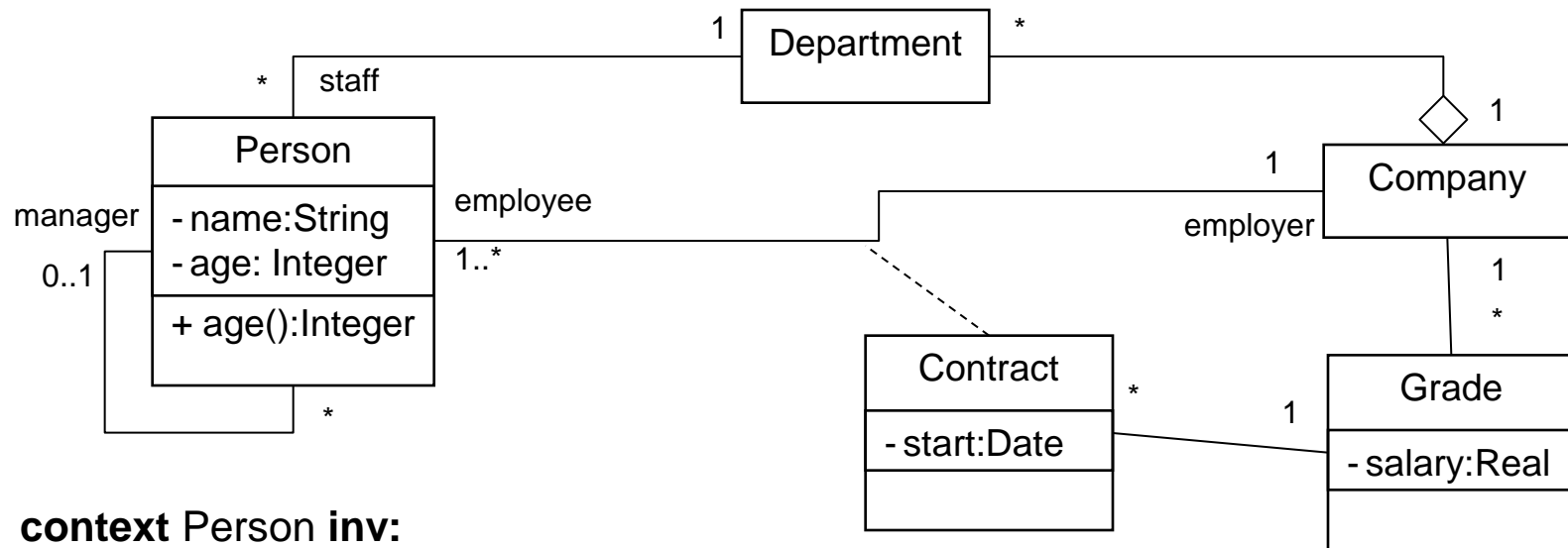
- Checking whether an object is part of a collection
`navigationExpression1->includes (navigationExpression2)`
Returns true if the result (one object) of `navigationExpression2` is in the collection resulting from `navigationExpression1`
- Checking whether a collection of objects is part of a collection
`navigationExpression1->includesAll (navigationExpression2)`
Returns true if the result (a collection) of `navigationExpression2` is in the collection resulting from `navigationExpression1`



context Flight inv:
`self.crew -> includes (self.pilot)`

context Flight inv:
`self.crew -> includesAll (self.flightAttendants)`

Examples of Basic Constraints



context Person inv:

self.employer = self.Department.Company

context Company inv:

self.employee->select(p:Person|p.age() < 18)->isEmpty()

context Person inv:

self.employer.Grade->includes(self.contract.grade)

context Department inv:

self.Company.employee->includesAll(self.staff)

Examples of Basic Constraints

context Customer

inv: title = if isMale then 'Mr.' else 'Ms.' endif

inv: age >= 18 and age < 66

inv: name.size < 100

Customer
name: String title: String age: Integer isMale: Boolean

context Person **inv:**

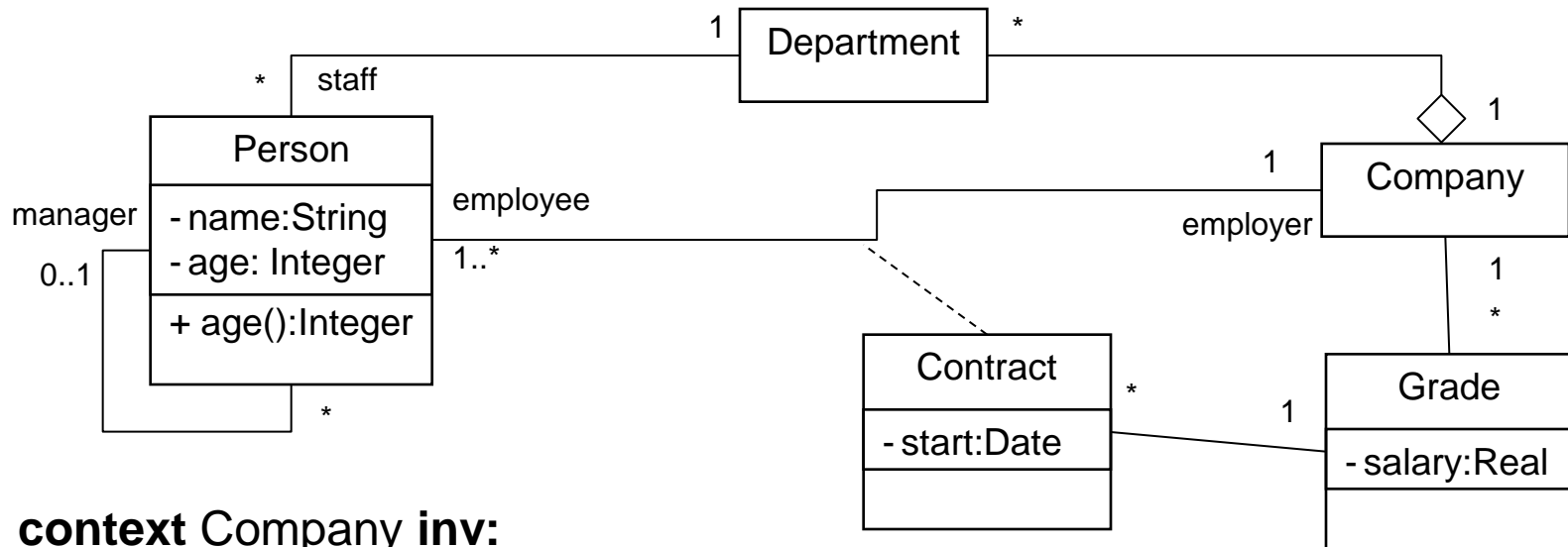
self.age() > 50 implies self.Contract.Grade.salary > 25000

Checking Constraint on the Elements of a Collection

- Apply a Boolean expression to every element in a collection and return a Boolean value
- `forall` returns true if the specified Boolean expression is true for **every** member of the collection
- `exists` returns true if the specified Boolean expression is true for **at least** one member of the collection

- When a condition has to be true for every instance of a class use:
`ClassName.allInstances`

Checking Constraint on the Elements of a Collection



context Company inv:

```
self.Grade->forAll(g:Grade | not g.contract->isEmpty())
```

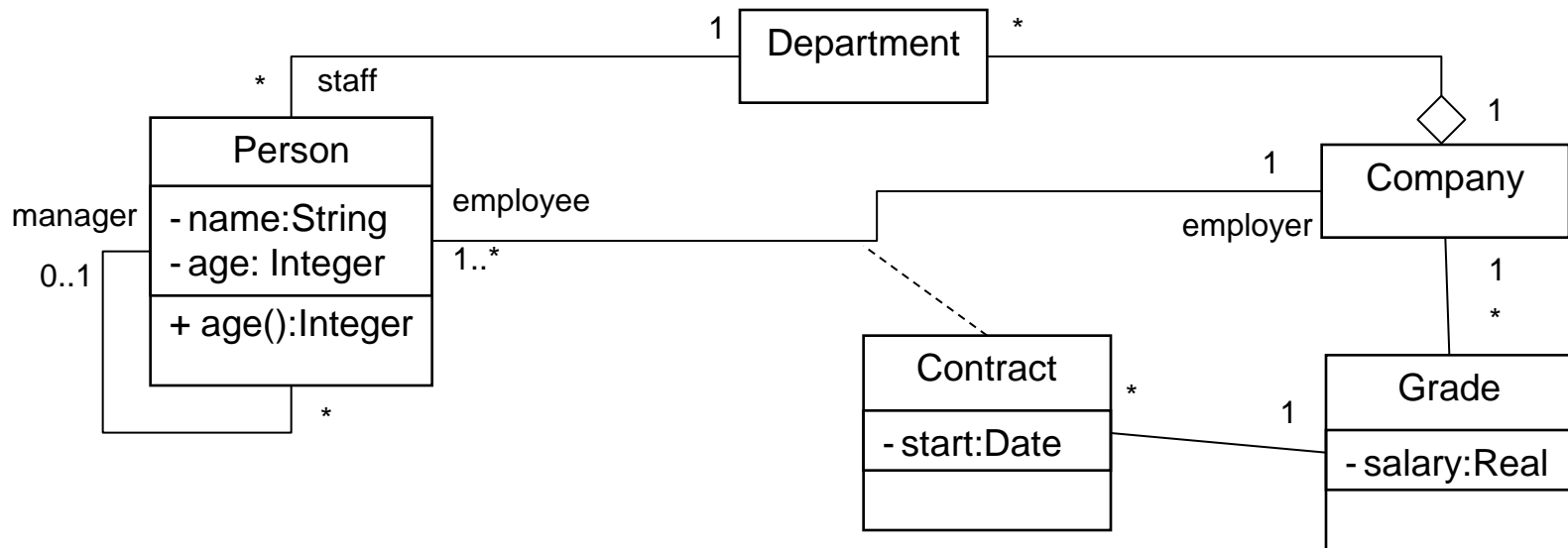
Each Grade instance of the Company is linked to at least one Contract instance. Shouldn't the multiplicity be 1..* instead of * then?

context Department inv:

```
staff->exists(e:Person | e.manager->isEmpty())
```

In a Department, there is one staff member who does not have a manager. Would that be the manager?

Checking Constraint on the Elements of a Collection



context Grade inv:

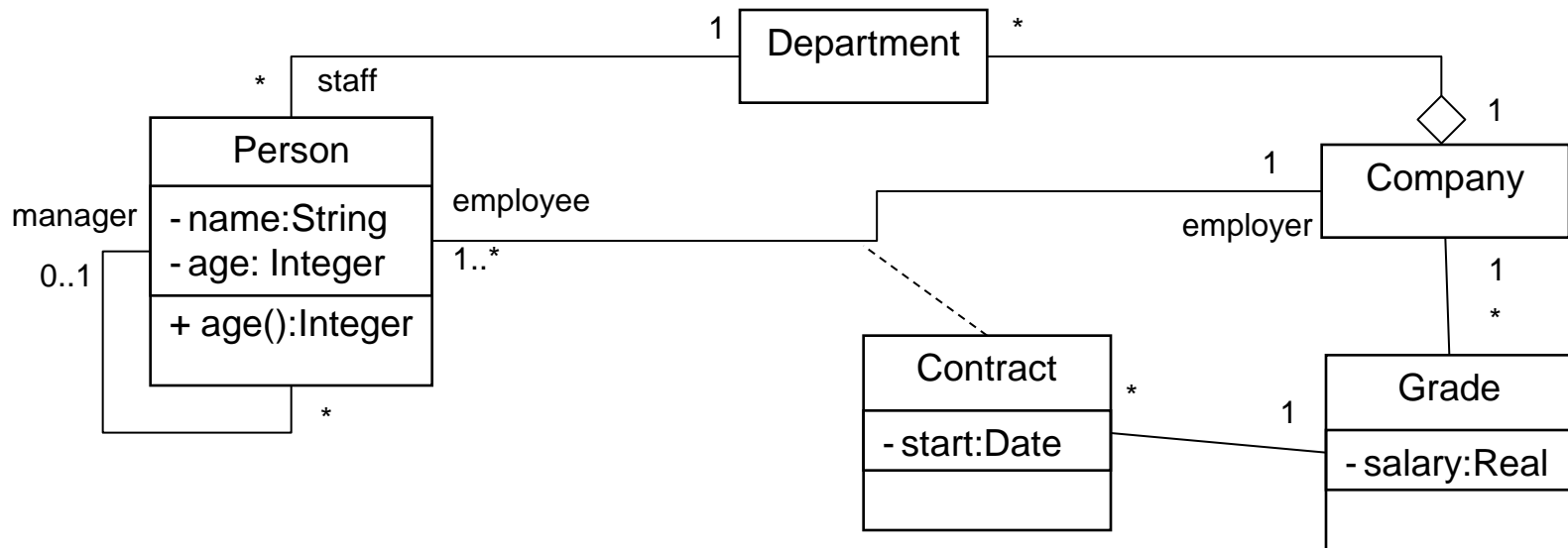
```
Grade.allInstances->forall(g : Grade |  
    g <> self implies g.salary <> self.salary)
```

In the system, no two different Grade instances have the same salary attribute value.

context Grade inv:

```
Grade.allInstances->forall(g1, g2 | g1 <> g2 implies g1.salary <> g2.salary)
```


Checking Constraint on the Elements of a Collection



context Grade inv:

salary > 20000

Two equivalent expressions.

A constraint on a class applies to all the instances of that class.

context Grade inv:

Grade.allInstances->forAll(g:grade | g.salary >20000)

Postcondition—A Specific Notation

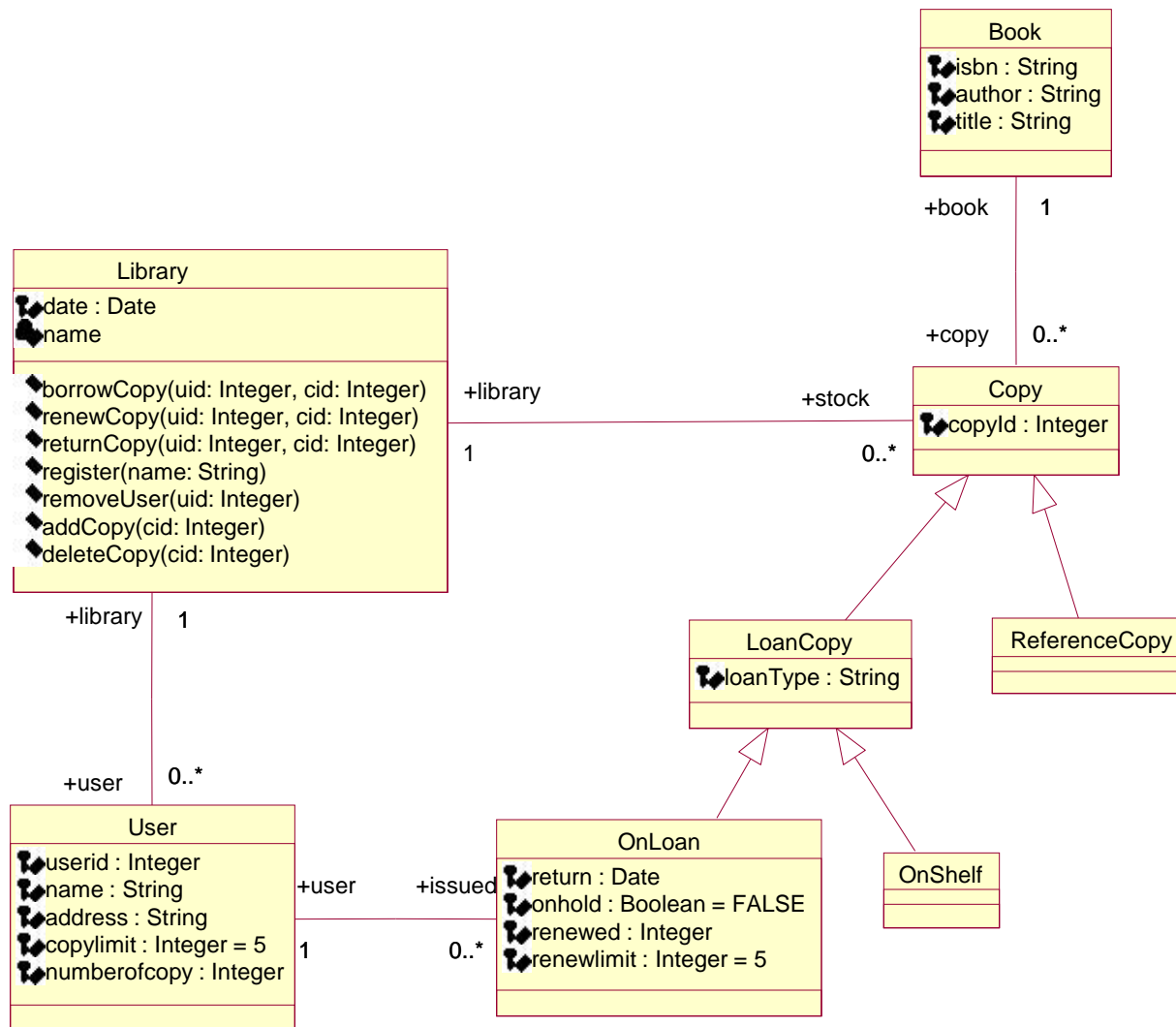
- In a postcondition, we usually want to relate the value of an attribute (or link, or inout parameter) to its value before the call.
 - E.g., balance has been reduced by the amount of the withdrawal.
 - $\text{new_balance} = \text{old_balance} - \text{amount}$
- Use postfix notation **@pre** to refer to a value before the execution of a method.

context SavingsAccount::withdraw(amt)

pre: amt < balance

post: balance = **balance@pre** - amt

A more Complex Example—A Library System



Contracts for Library::borrowCopy

context Library::borrowCopy(uid, cid)

pre :

self.user->exists(user : User | user.userid = uid and not
user.numberofcopy = user.limit)

and

self.OnShelf->exists(onshef : OnShelf | onshel.f.copylid = cid)

post :

not self.OnShelf->exists(onshef : OnSelf | onshel.f.copylid = cid)

and

self.OnLoan->exists(onloan : OnLoan | onloan.copylid = cid)

and

self.user->exists(user : User | user.userid = uid

and user.numberofcopy = user.numberofcopy@pre + 1

and user.OnLoan->exists(onloan : Onloan | onloan.copylid = cid))

SYSC-3120—Software Requirements Engineering

- Overview
- Analysis Concepts
 - Modeling structure: class diagram
 - Modeling interaction: sequence diagram
 - Modeling state-based behavior: state machine diagram
- Analysis Process
 - Finding objects/classes (heuristics)
 - Finding relationships : associations and attributes (heuristics)
 - ...

Modeling Interactions

- Interaction diagrams are used to illustrate how objects interact via messages.
 - They are used for dynamic object modeling.
 - Two types of interaction diagrams
 - Sequence Diagram
 - Show an exchange of messages between objects arranged in a time sequence
 - Collaboration Diagrams
 - Emphasize the relationships between objects along which the messages are exchanged
 - More useful in design
 - Sequence and collaboration diagrams can be used interchangeably
 - Some CASE tools allow (automatically) creating one from the other.
 - Collaboration diagram used for structuring, used mostly during design
 - Can be used to determine operations in classes
-

Sequence Diagram Notation

The name of the participant can be:

- A named instance (like here)
- An anonymous instance (no object name, but class name required)
- A class if interaction through class operations (class scope): not underlined

Interaction between participating objects

anObjectName:MyClass

otherObjectName:MyOtherClass

Execution occurrence: something executes with a start and an end. Also called execution bar.

Send event: the message is sent

Receive event: the message is received

Notice the colon separating the object name from the class name

message

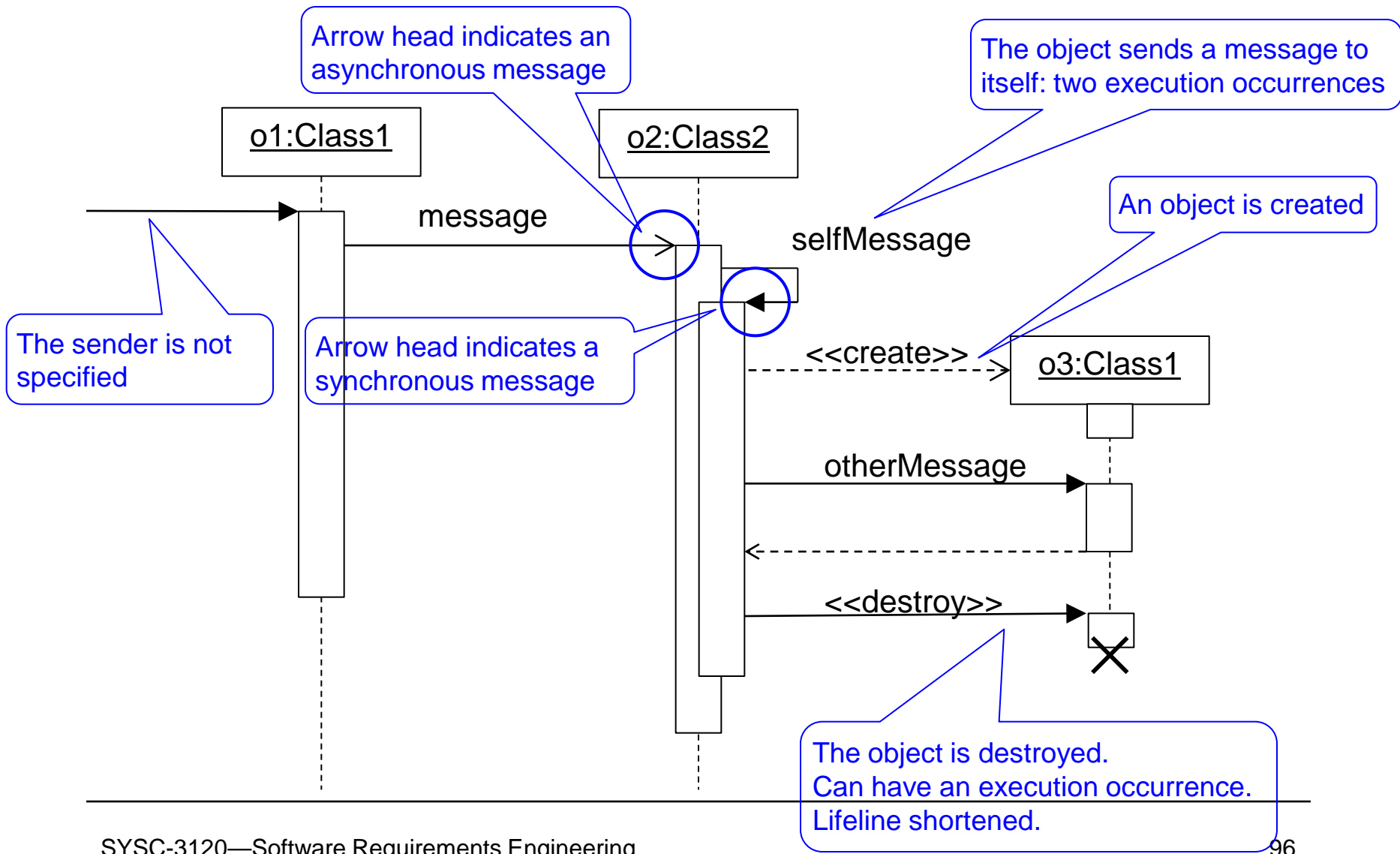
Message label

Interaction through message passing

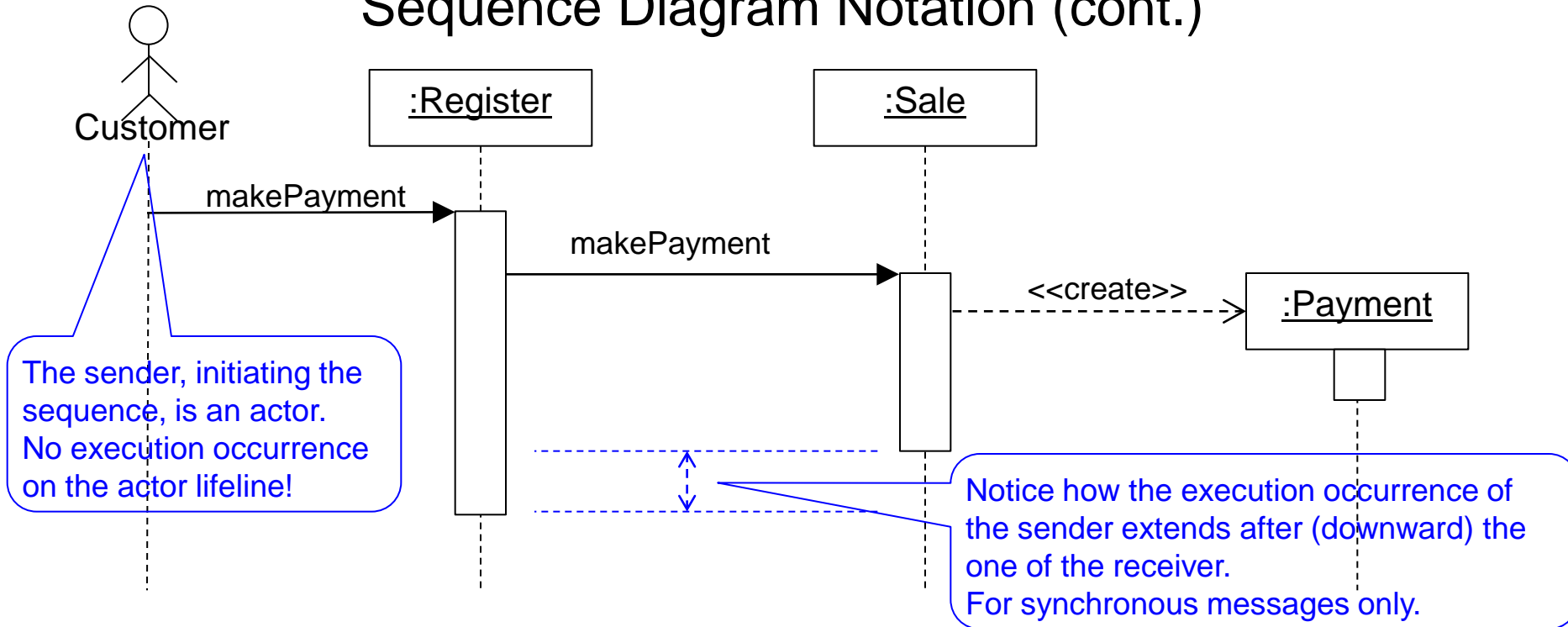
Return message (optional)

The object's lifeline (time flows downward)

Sequence Diagram Notation (cont.)

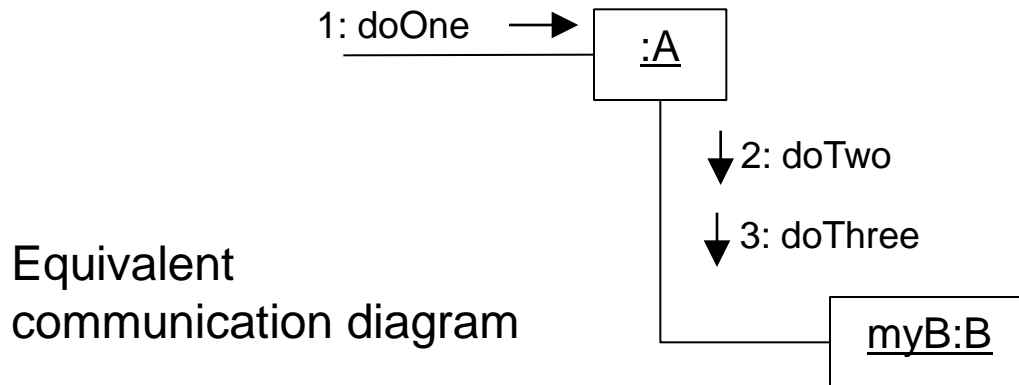
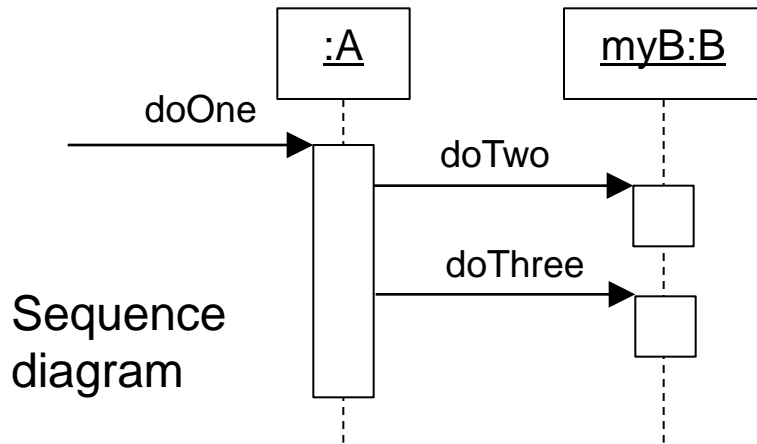


Sequence Diagram Notation (cont.)



- The sequence diagram is read as follows:
 - The message *makePayment* is sent to an instance of a *Register*. The sender is an actor.
 - The *Register* instance sends the *makePayment* message to a *Sale* instance.
 - The *Sale* instance creates an instance of a *Payment*.

Example of Sequence & Communication Diagrams

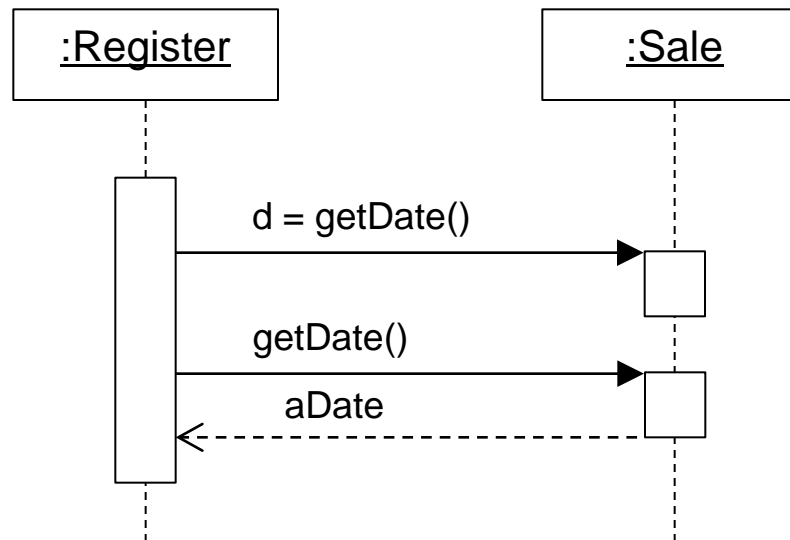


What might this represent in code? e.g. java

```
public class A {
    private B myB = new B();
    public void doOne() {
        myB.doTwo();
        myB.doThree();
    }
    // ...
}
```

Lifelines: Reply or Returns

- There are two ways to show the return result from a message:
 - Using the message syntax *returnVar = message(parameter)*
 - Using a reply (or return) message line at the end of an execution specification bar.
- Both are common in practice. The first one is preferred because it is less effort and the diagram is not cluttered.



Message Label

- General syntax

`aNamedVariable` = `signal_or_operation_name` (`arguments`) :
`return_value`

Operation to be invoked or
signal to be emitted

Optional: where the result is stored.

Can be:

- A local variable of the interaction (execution occurrence initiating the message).
- An attribute of the initiating lifeline.

Optional
Value being returned (not
the variable being assigned)

Coma-separated list

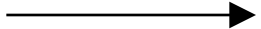
Argument can be:

- a parameter of the sending execution occurrence,
- an attribute,
- a local variable from assigned by a previous message.

Different categories of Messages

- Message can denote the following interactions:

- Call

- Denotes (Usually) synchronous invocation of an operation 
 - The return message can return some values to the caller or it can just acknowledge that the operation completed

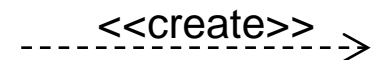
- The return message is optional

- May also be an asynchronous invocation 

- The sender continues without waiting

- No return message

- <<create>>



- <<destroy>>

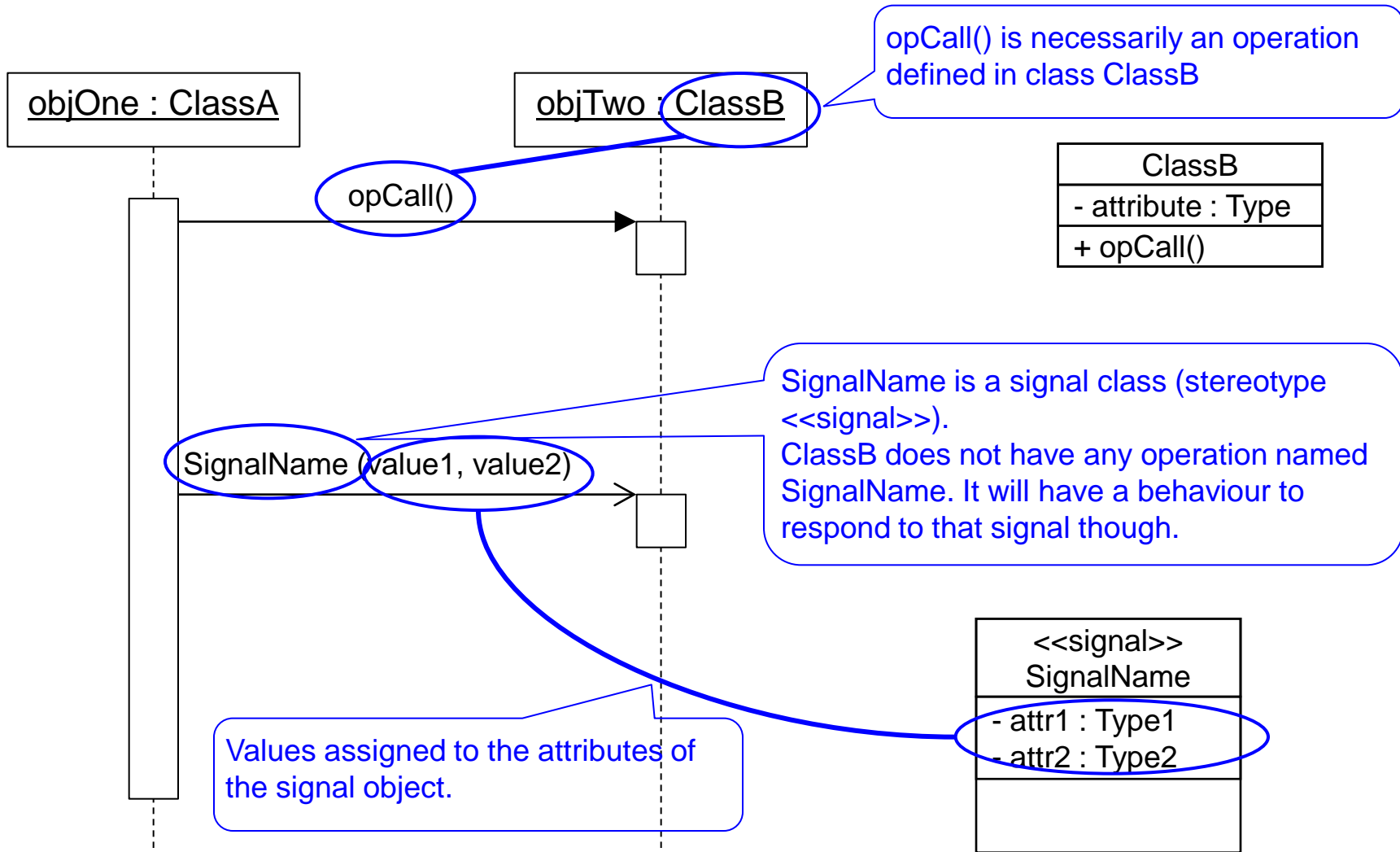


- Signal

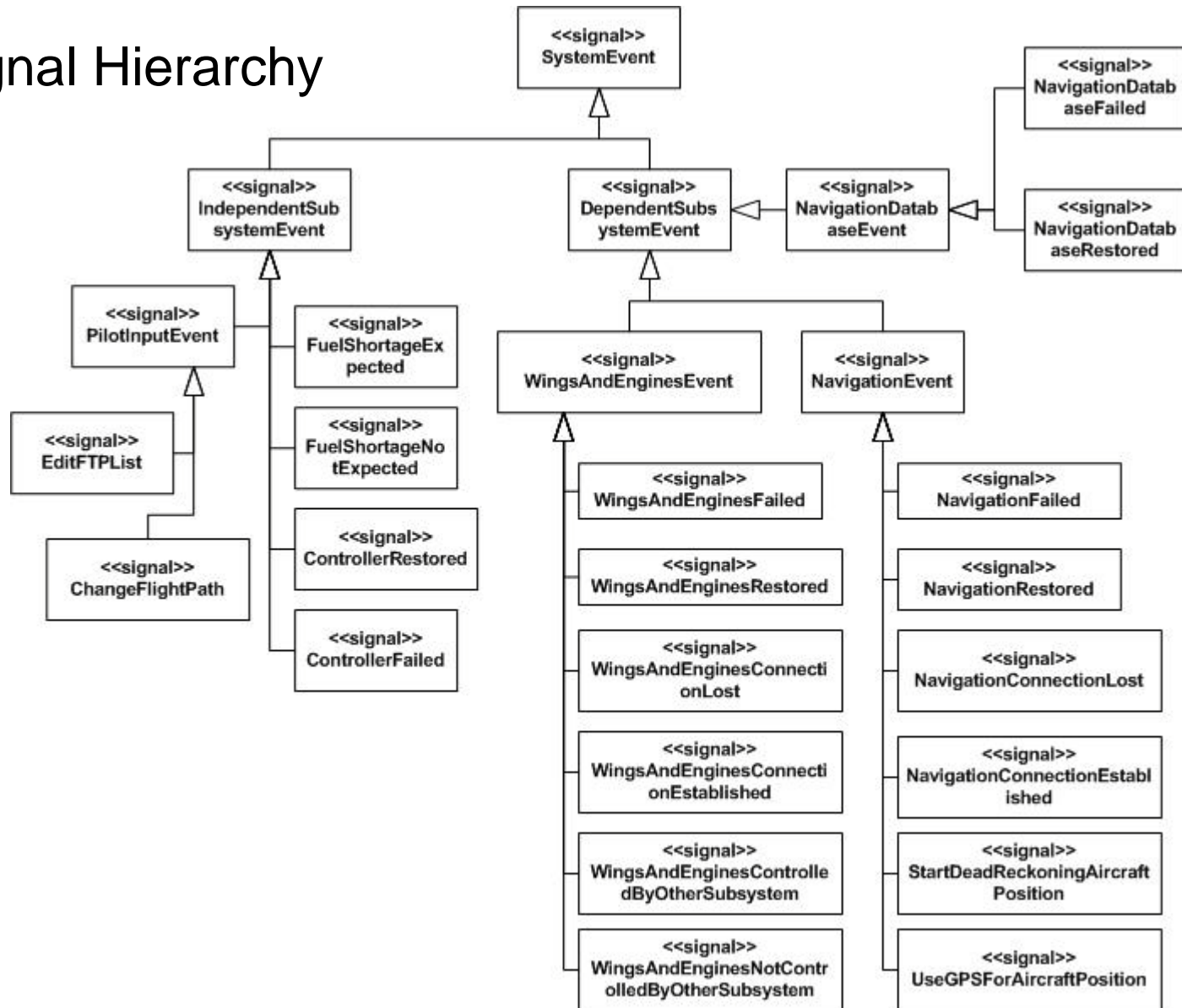
- Denotes asynchronous inter-object communication 

- The sender continues executing after sending the signal message

Call vs. Signal

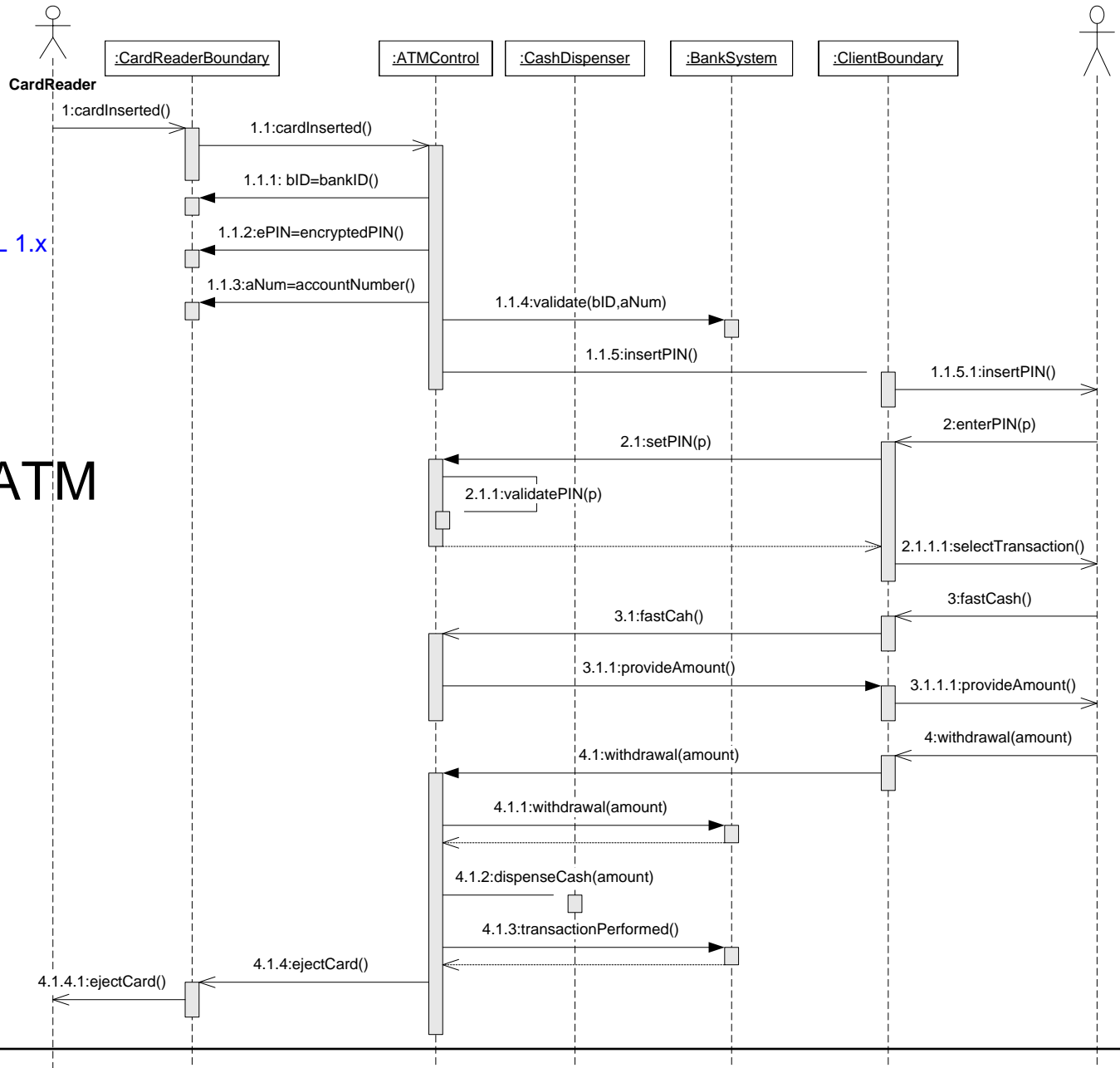


Signal Hierarchy

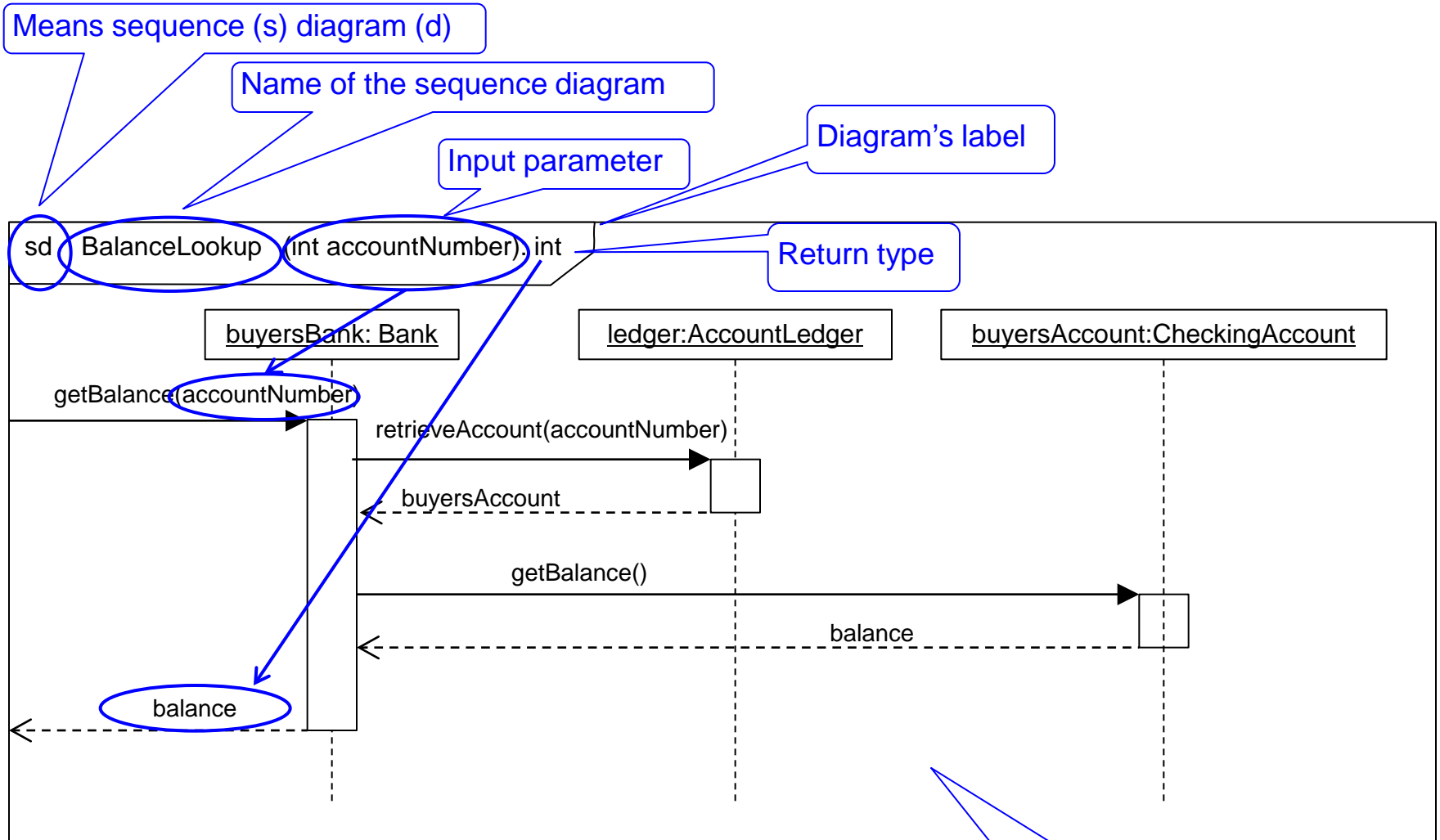


Message numbering is UML 1.x

Example - ATM



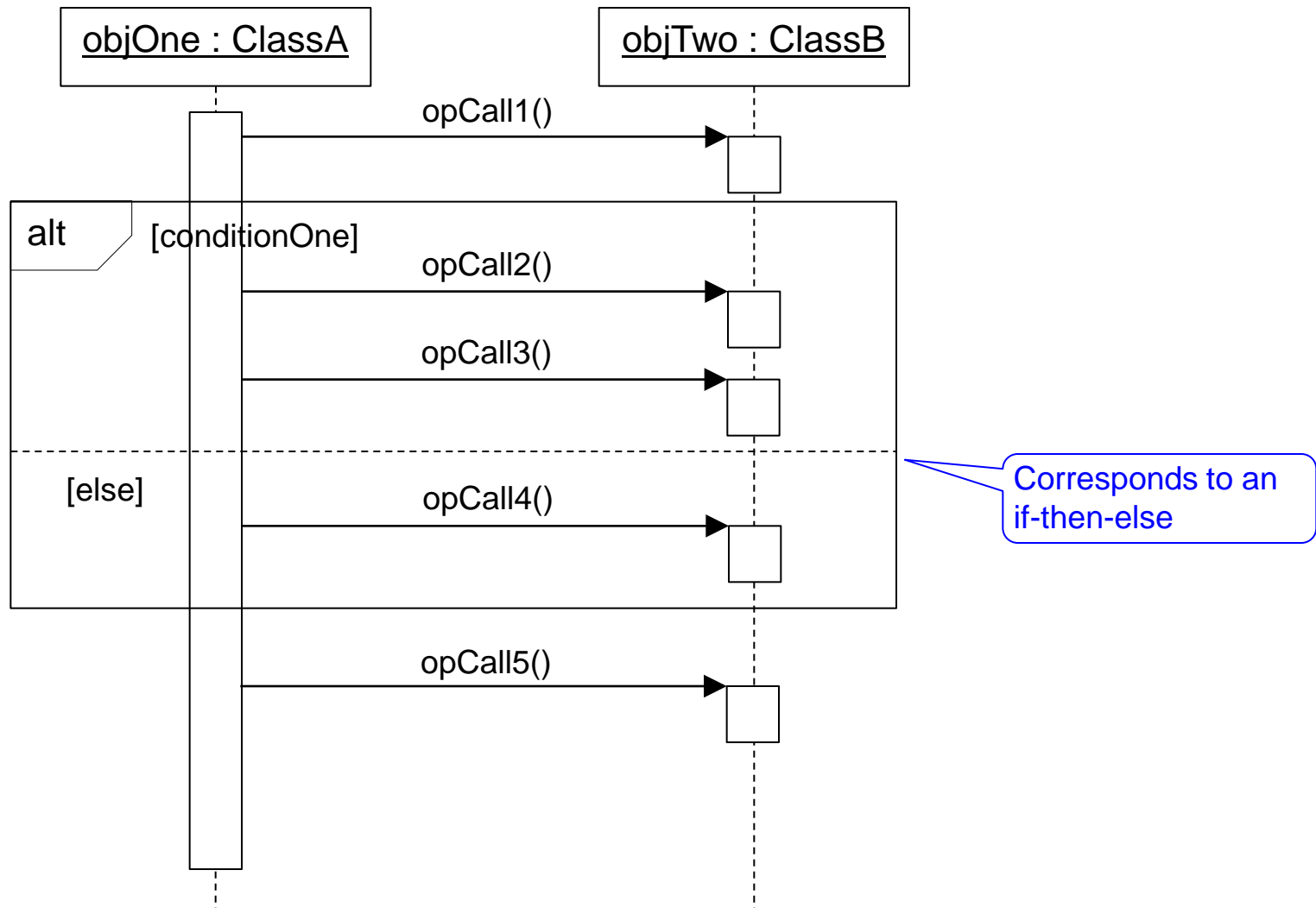
Sequence Diagram's Frame



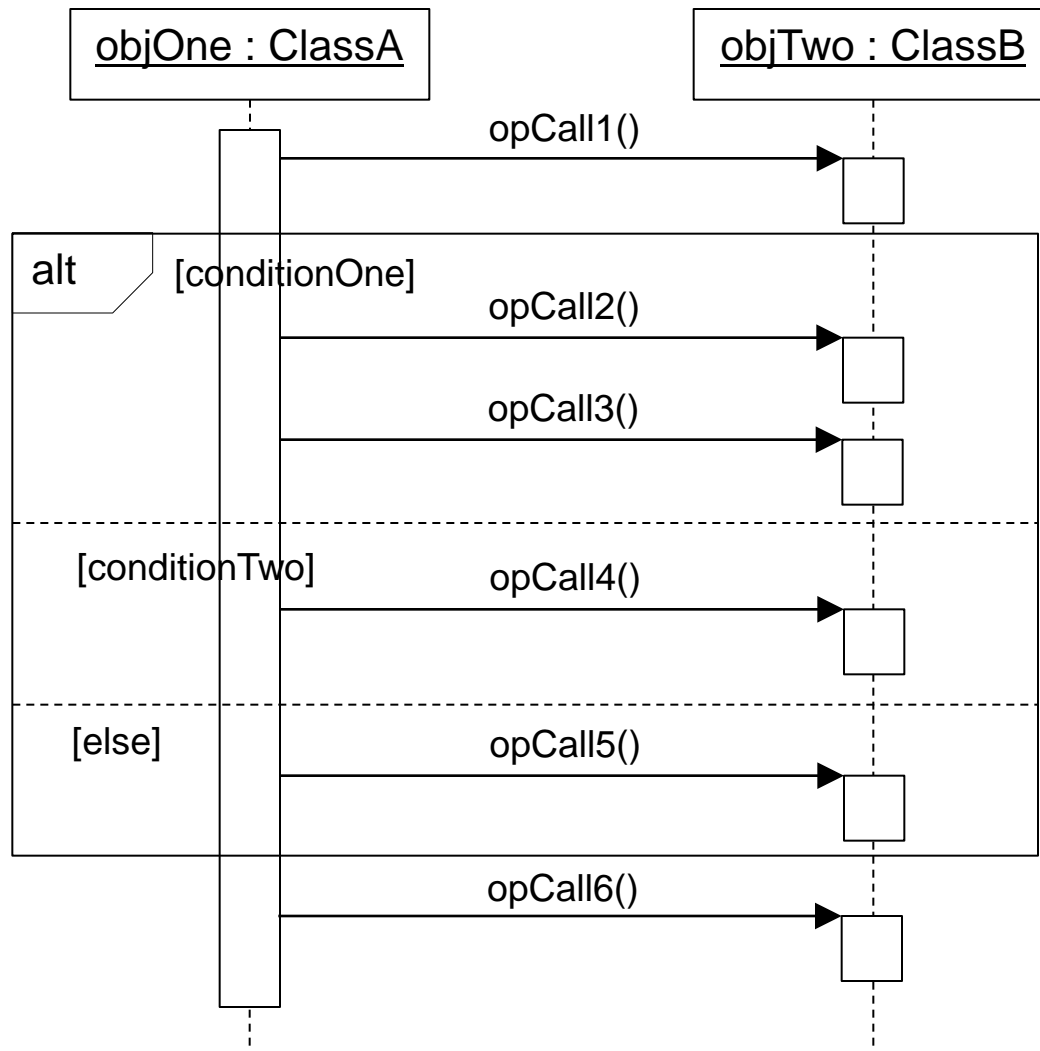
Other Frames—Control Flow

- To support conditional and looping constructs (among many other things), the UML 2.0 uses **frames**.
- Frames are regions or fragments of the diagrams; they have an operator or label (such as *loop*) and guard (conditional clause).
- Different kinds of frame exist (not exhaustive list):
 - **alt**: At most one operand's condition will evaluate to true. If there is an else operand and none of the other operands have executed, then the else will be executed.
 - **loop**: Fragment will be executed repeatedly. That is, loop fragment while guard is true. Can also write *loop(n)* to indicate looping n times
 - **opt**: A choice in which either this fragment will execute or it will not, depending on whether the guard is true or not
 - **par**: Operands execute in parallel. Any interleaving between operands is possible; order within operands maintained
 - **break**: A fragment with a condition which, if it is true, will be executed and will break out of the enclosing fragment
 - **ref**: a sequence diagram is invoked within another sequence diagram

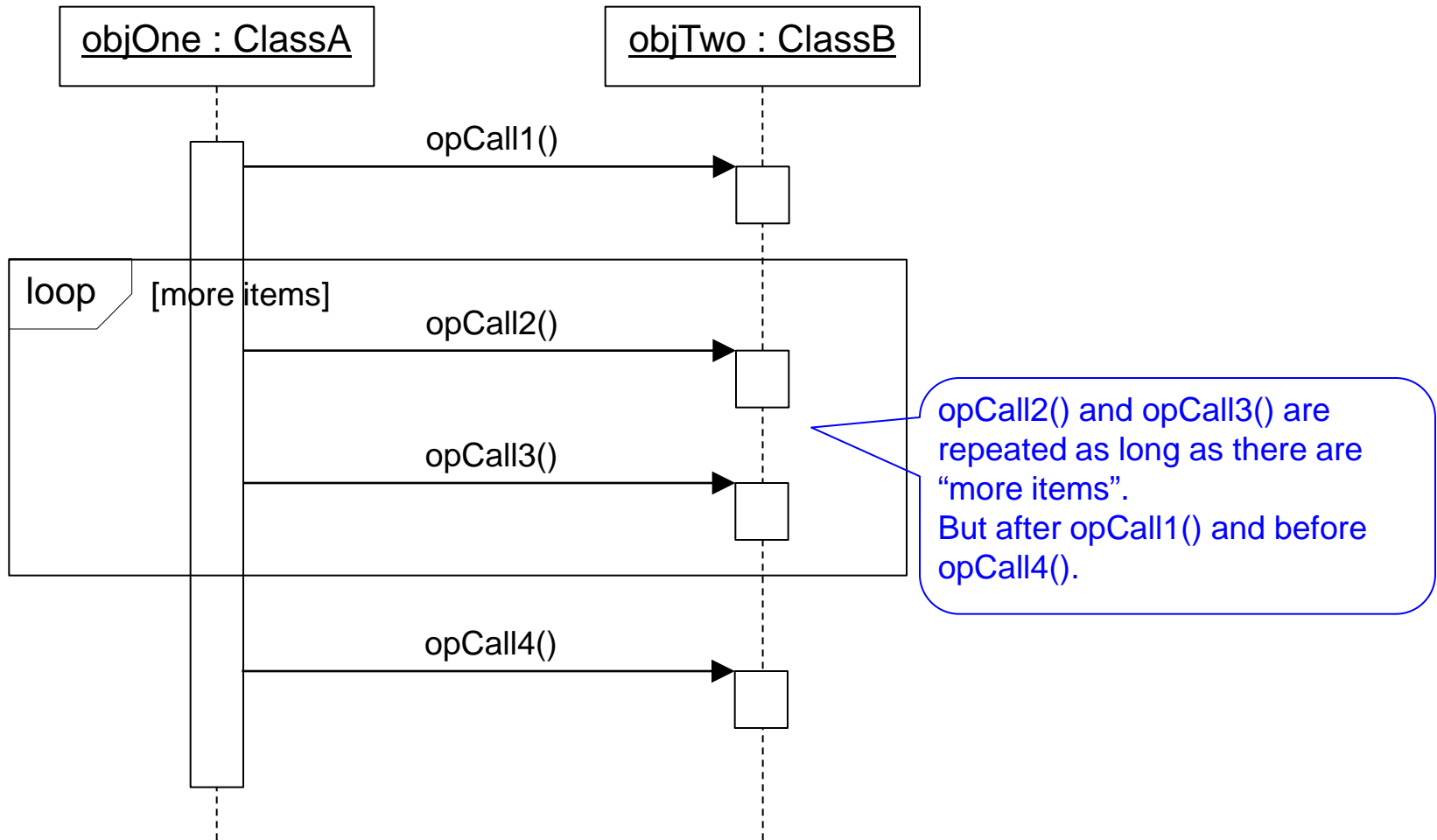
Other Frames—Control Flow (cont.)



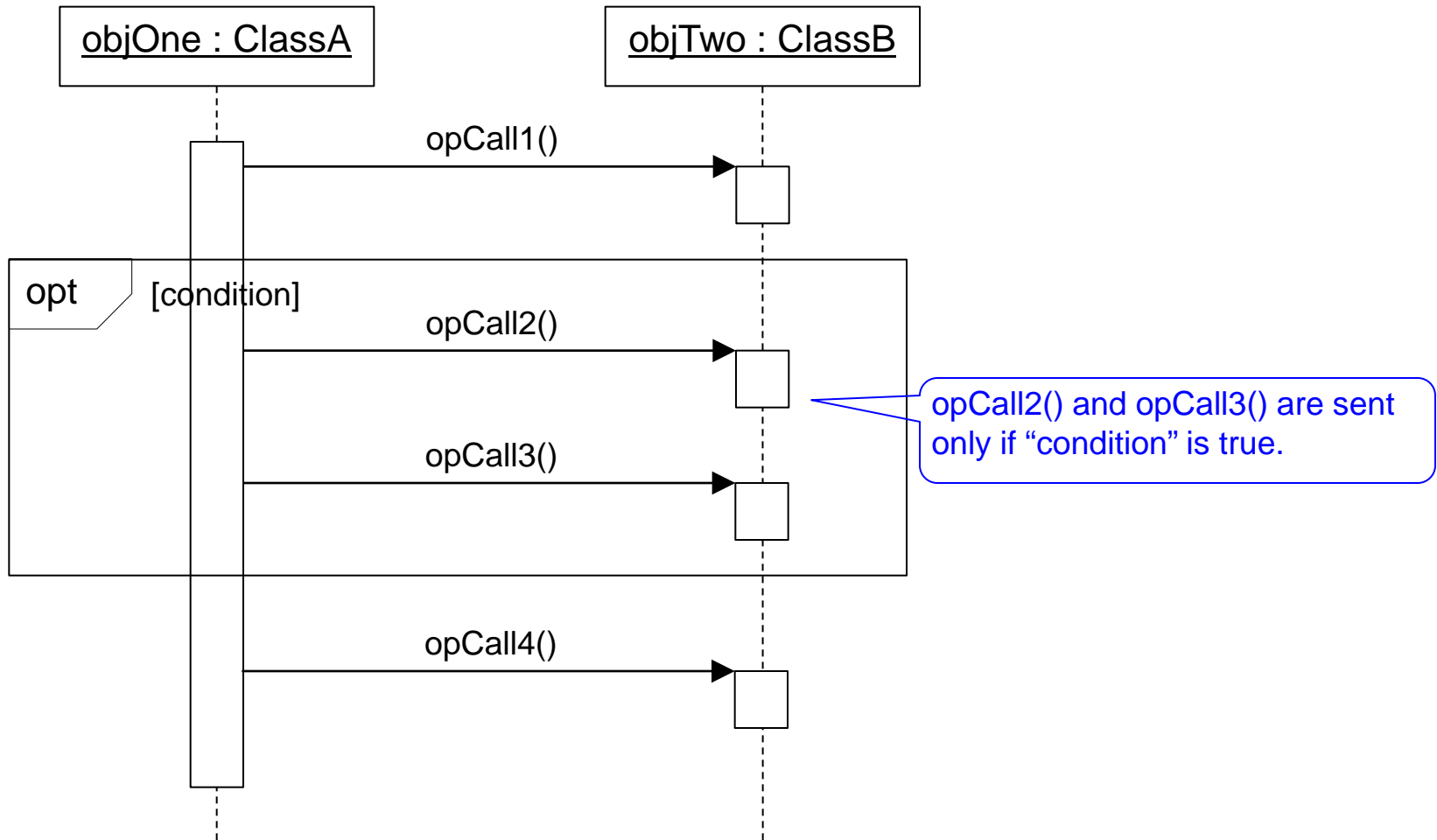
Other Frames—Control Flow (cont.)



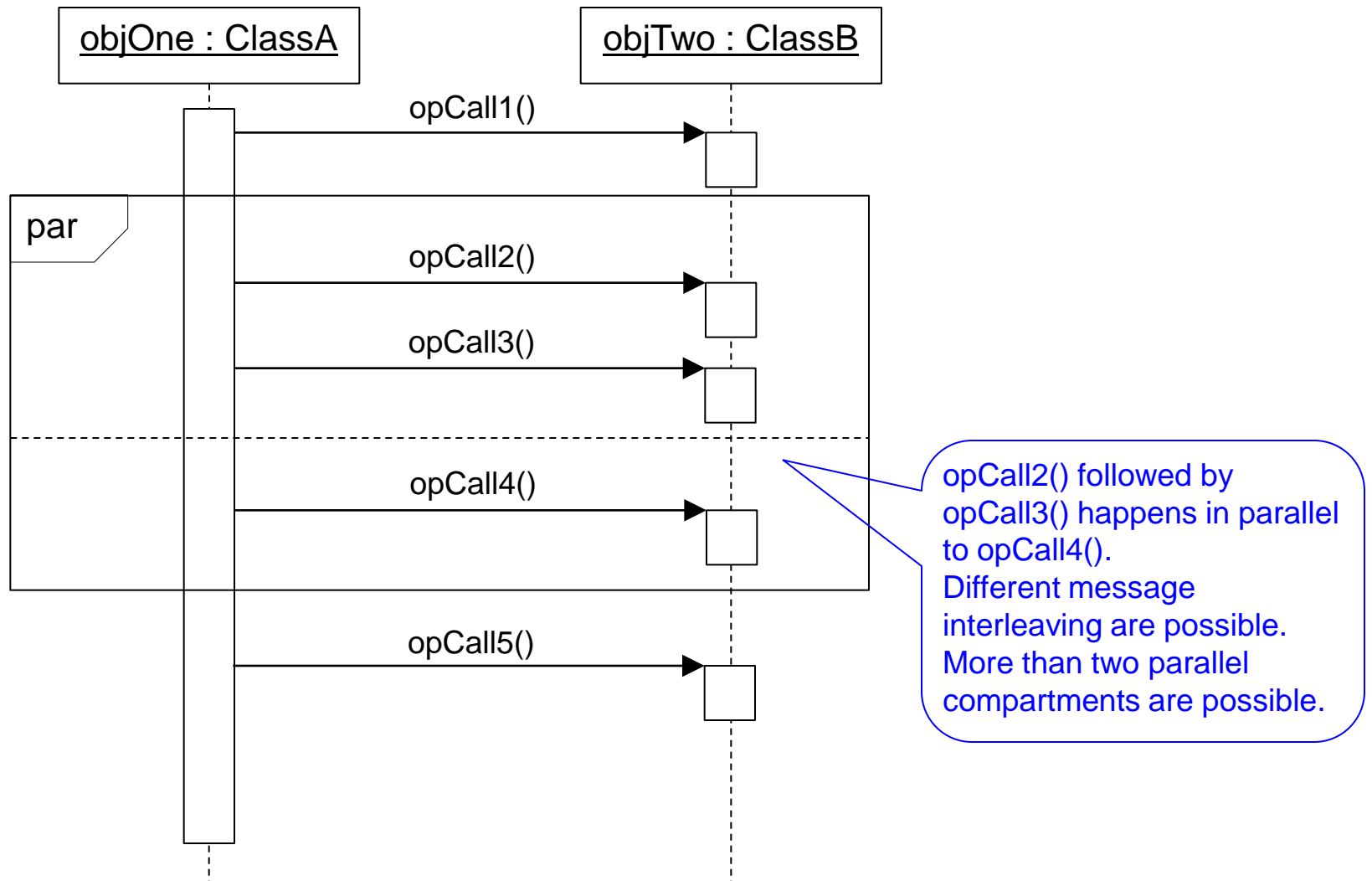
Other Frames—Control Flow (cont.)



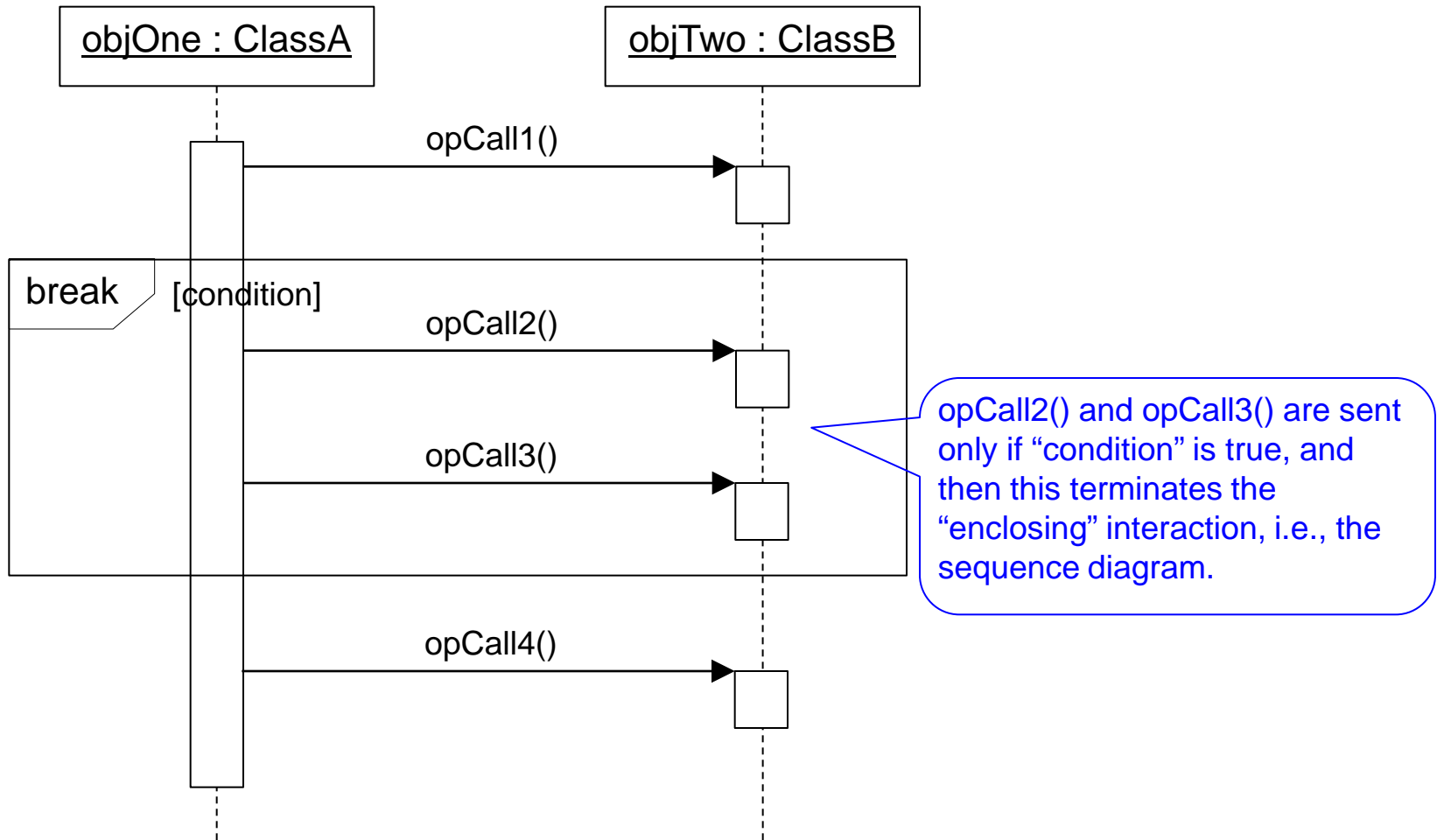
Other Frames—Control Flow (cont.)



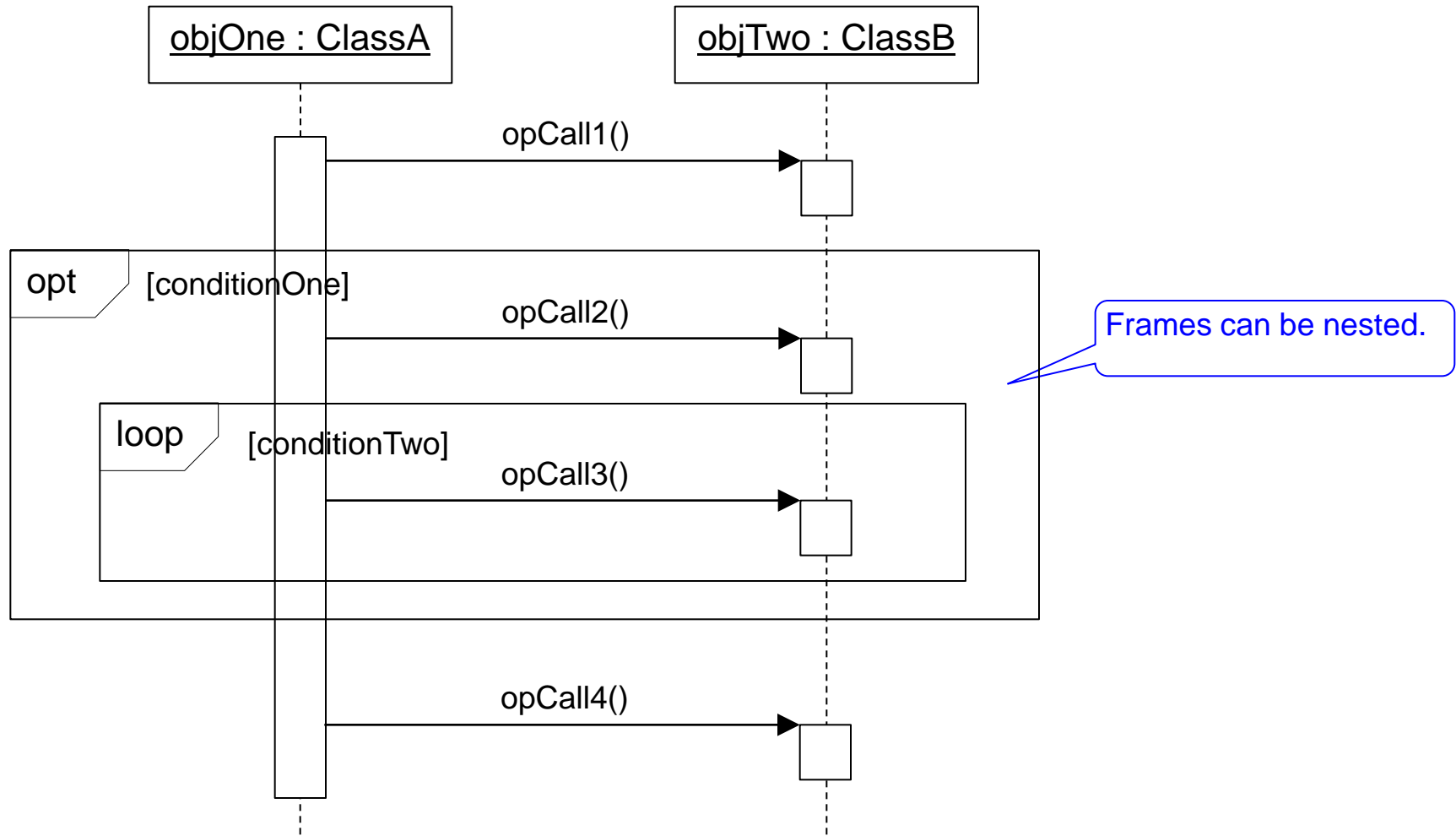
Other Frames—Control Flow (cont.)



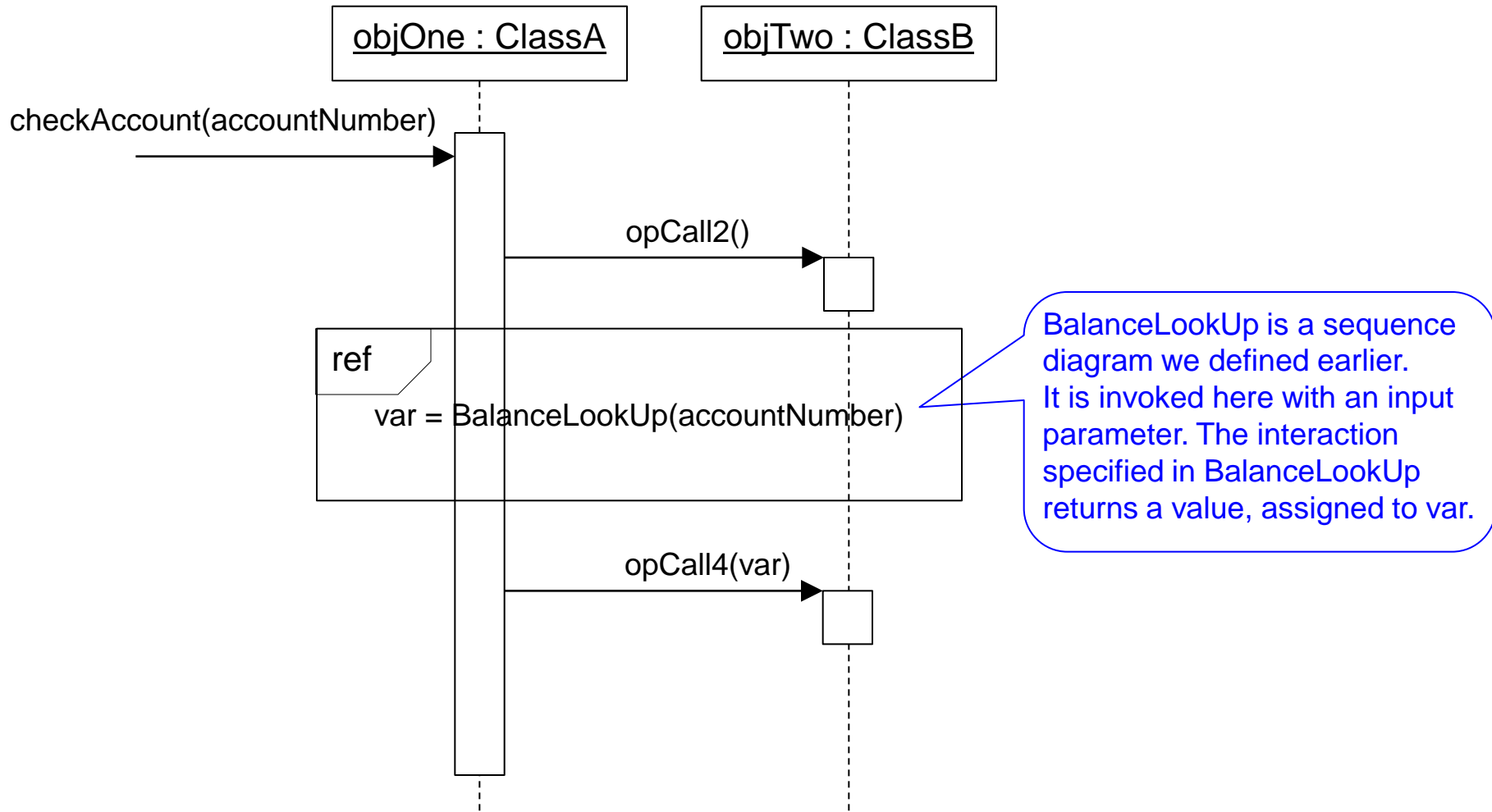
Other Frames—Control Flow (cont.)



Other Frames—Control Flow (cont.)



Other Frames—Control Flow (cont.)



SYSC-3120—Software Requirements Engineering

- Overview
- Analysis Concepts
 - Modeling structure: class diagram
 - Modeling interaction: sequence diagram
 - Modeling state-based behavior: state machine diagram
- Analysis Process
 - Finding objects/classes (heuristics)
 - Finding relationships : associations and attributes (heuristics)
 - ...

Different Kinds of Object Behaviours

[Douglas, Wagner et al]

- **Simple behaviour:** object performs services on request and keeps no memory of previous services
 - e.g., a simple math function such as sine, or square root
 - returns the value measured from a sensor at a given instant in time
 - returns value of object attribute.
- **State behaviour** (a.k.a., state-driven, reactive): the way the object performs services depends on what happened in the past (memory), i.e., what other services have occurred before
 - e.g., a cruise control
 - an elevator control
- **Continuous behaviour:** current output depends on the previous history in a way that does not lend itself to discretization (as in state behaviour)
 - e.g., digital filter

Notion of State in a Finite State Machine

- State = information about past history.
condition that persists for a significant period of time
- All states represent **all possible situations** in which the state machine may ever be.
- Contains a kind of **memory**: how the state machine can have reached the present situation.
- As the application runs the state changes from time to time, and outputs may depend on the current state as well as on the inputs.
- States are **distinguishable**: i.e., they observably differ from one another in either one (or several) of:
 - The events they accept
 - The transition they take as a result of accepting those events
 - A transition is a response to an event that causes a state change
 - The actions they perform.

State = Condition: Condition on What?

- The current state of an object is determined by:
 - the current value of the object's attributes (state variables)
 - the current value (and contents) of links that it has with other objects
 - Possibly the current value of other (linked) objects' attributes
 - Example:
 - class `StaffMember` has an attribute `startDate`
 - `startDate` determines whether a `StaffMember` object is in the *probationary state*:
 - The `StaffMember` object is in the `Probationary` state for the first six months of employment.
 - While in this state, a staff member has different employment rights.
 - Some attributes and links of an object are significant for the determination of its state while others are not.
 - `staffName` and `staffNo` attributes of a `StaffMember` object have no impact upon its state
 - Often: several attributes' and links' values are used to define a state
-

State Conditions are Distinguishable

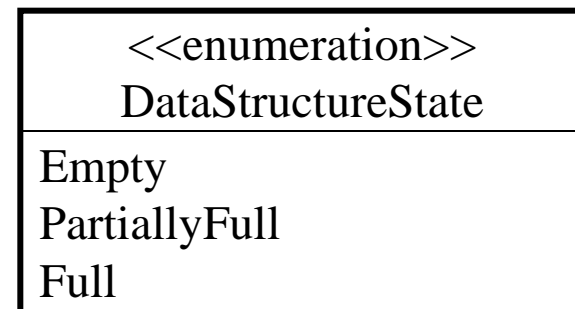
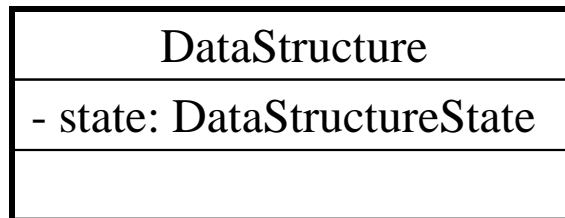
- Since any two different states are distinguishable
Since a state defines a condition
Then, **conditions are distinguishable**
- Consider a data structure that has a maximum capacity.
- One can define three states:
 - The data structure is empty: $\text{numberOfElements}=0$
 - The data structure is full: $\text{numberOfElements}=\text{maxCapacity}$
 - The data structure is partially full:
 $\text{numberOfElements}>0$ and $\text{numberOfElements}<\text{maxCapacity}$
- Conditions are distinguishable since numberOfElements can only satisfy one (and only one) of the three conditions.

State vs. Class Invariant

- State condition = State invariant
 - i.e., a condition that does not vary (invariant) while the object is in the state
- Class invariant = what states an object can be in
- Consider a data structure that has a maximum capacity.
- State invariant for state Empty: numberOfElements=0
- Class invariant: the object is either Empty, Full or PartiallyFull.

State vs. Class Invariant

- Often useful to add a **state** attribute to the class
- Consider a data structure that has a maximum capacity.
- State invariant for state Empty:
state=DataStructureState.Empty \Leftrightarrow numberOfElements=0
- Class invariant:
state=DataStructureState.Empty
xor state=DataStructureState.PartiallyFull
xor state=DataStructureState.Full



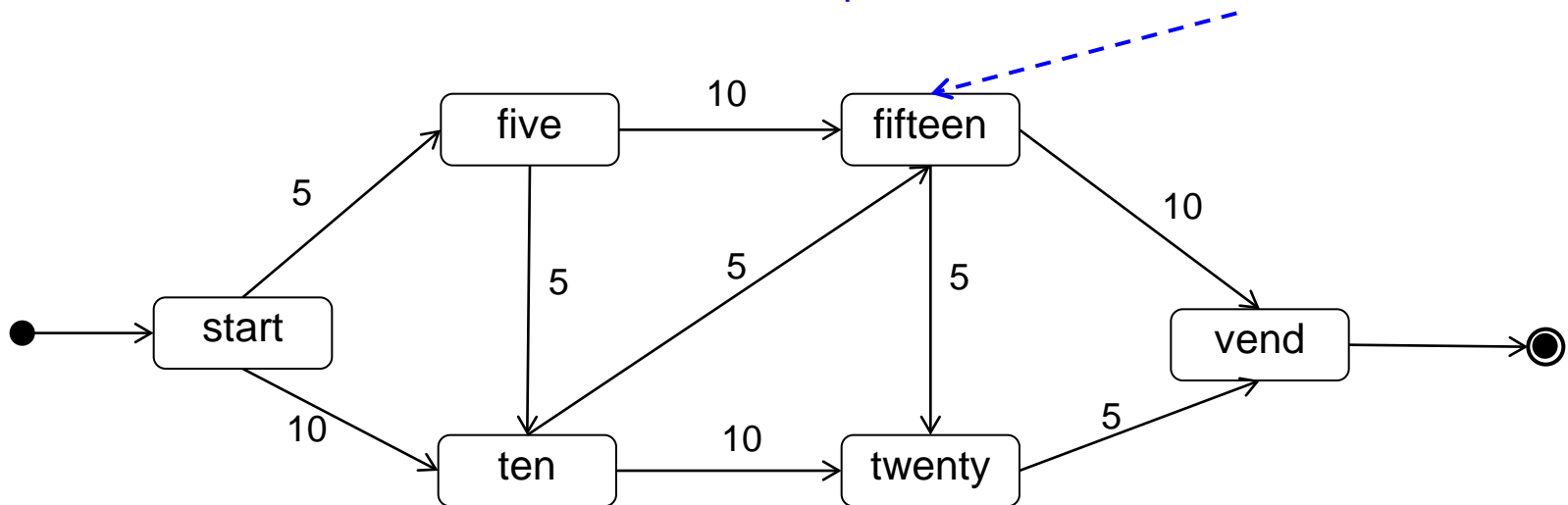
Finite State Machine

- The object (or component) being modeled can only assume a finite number of existence conditions called states
- The object behaviour in a given state is (distinguishable from other states' behaviour) defined by:
 - The messages and events accepted
 - The actions associated with each incoming event
 - The state's reachability graph (i.e., how state can change)
 - The set of transitions
- An object spend all its time in states
 - I.e., transitions take (approximately) zero time
- The object may change state only in a finite number of well-defined ways, called transitions
- Transitions are enabled by events: a response to an event that causes a change in state
- An object cannot be in two different states at the same time.
 - One (and only one) state condition holds at a given instant

A Simple Finite State Machine

A control system has to count the amount of money dropped into a vending machine. Only 5 and 10 cent coins are accepted. The correct, recognized sum (e.g., to deliver a stamp) is 25 cents.

Idea of past (history) defined by successive inputs: 5+5+5 or 5+10 or 10+5.



Transitions and Events

- Transition: the act of changing state
 - A transition is initiated by an event.
 - Four kinds of events in UML:
 - Signal event:
 - An occurrence of interest arising asynchronously from outside the scope of the state machine
 - Call event:
 - An explicit synchronous notification of an object by another
 - Change event:
 - An event based on the changing of an attribute value
 - Time event:
 - Either the elapse of a specific duration or the arrival of an absolute time
 - **Warning: an event is just that**
 - Something that occurs at a particular instant
 - Recall: transitions take (approximately) zero time
-

SYSC-3120—Software Requirements Engineering

- Overview
- Analysis Concepts
 - Modeling structure: class diagram
 - Modeling interaction: sequence diagram
 - Modeling state-based behaviour: state machine diagram
- **Analysis Process**
 - Finding objects/classes (heuristics)
 - Finding relationships : associations and attributes (heuristics)
 - Interactions/behavior (heuristics)
 - Responsibilities and consistency
 - Analysis review

Object Modeling

- Steps during object modeling
 1. Class identification
 2. Find the attributes
 3. Find the methods
 4. Find the associations between classes
 5. Review (iterate, iterate, iterate)
- Order of steps
 - Order of steps secondary, only a heuristic
 - Iteration is important
 - Static model will be refined when devising dynamic models

Class Modeling during Analysis

- Classes, their *invariant*, associations, and attributes are determined at this stage
- *Associations* among classes
- *Attributes* of classes
- But also system *operations* and their *contracts*, i.e., pre- and post-conditions
- Invariant: condition that must remain true under any circumstances for an instance of the class
- Pre-condition: condition that must be true for the operation to execute correctly
- Post-condition: the effect of executing the operation in terms of system state change and outputs

Finding classes (I)

- Identify participating objects in each use case
 - Pick one (there are many, so prioritize!)
- They will correspond to the main concepts of the application domain
 - Always use application domain terms
- They are named, described, consolidated into the data dictionary (glossary)
 - If two use cases refer to the same concept, the corresponding object should be the same.
 - If two objects share the same name and do not correspond to the same concept, one or both concepts are renamed to acknowledge and emphasize their difference.
- Benefits of a data dictionary:
 - consistent set of definitions for all developers
 - single term for each concept
 - precise and clear official meaning
- Definitions of objects and attributes may be reviewed by the users
- Initial Analysis model, several iterations

Finding classes (II)

- General Advice
 - Find the nouns in the use cases (e.g., Incident)
- Systematic Processes
 - Abbott's Textual Analysis
 - CRC cards (Class-Role-Collaboration)
- Heuristics
 - Heuristics for Entity
 - Heuristics for Boundary
 - Heuristics for Control

Example: Report Emergency

Flow of Events

1. FieldOfficer activates the “Report Emergency” function of terminal
2. FRIEND responds by presenting a form to the officer, including location, incident description, resource request and hazardous material fields.
3. FieldOfficer completes form by specifying minimally the emergency type and description fields. May also describe possible responses to the emergency situation and request specific resources. Once form is completed, FieldOfficer submits the form.
4. FRIEND receives form and notifies Dispatcher
5. Dispatcher reviews submitted information and creates an Incident in the database by invoking the OpenIncident use case. All information contained in form is automatically included in the Incident. Dispatcher selects a response by allocating resources to the Incident (with AllocateResources use case) and acknowledges the emergency report with a short message to FieldOfficer
6. FRIEND display acknowledgement and selected response to FieldOfficer.

Textual Analysis

Mapping parts of speech to object model components [Abbot 1983]

Examples from ReportEmergency Use Case

Part of speech	Model component	Example
• proper noun	• instance	• Alice
• common noun	• class	• FieldOfficer
• doing verb	• method	• submit
• being verb	• inheritance	• is a kind of
• having verb	• aggregation	• has, includes
• modal verb	• constraint	• must be
• adjective	• attribute	• incident description

Example: ReportEmergency

Flow of Events

1. FieldOfficer activates the “Report Emergency” function of terminal
2. FRIEND responds by presenting a form to the officer, including location, incident description, resource request and hazardous material fields
3. FieldOfficer completes form by specifying minimally the emergency type and description fields. May also describe possible responses to the emergency situation and request specific resources. Once form is completed, FieldOfficer submits the form.
4. FRIEND receives form and notifies Dispatcher
5. Dispatcher reviews submitted information and creates an Incident in the database by invoking the OpenIncident use case. All information contained in form is automatically included in the Incident. Dispatcher selects a response by allocating resources to the Incident (with AllocateResources use case) and acknowledges the emergency report with a short message to FieldOfficer
6. FRIEND displays acknowledgement and selected response to FieldOfficer.

Can you identify Common Noun, Doing verbs, Being Verbs, Having verbs, Modal verbs, Adjectives ?

Example: ReportEmergency

Flow of Events

1. FieldOfficer **activates** the “**Report Emergency**” **function of terminal**
2. FRIEND **responds** by presenting a **form** to the **officer**, **including** location, incident description, resource request and hazardous material fields
3. FieldOfficer **completes form by specifying minimally** the **emergency type** and **description fields**. **May also describe possible responses to the emergency situation** and **request specific resources**. **Once form is completed**, FieldOfficer **submits** the **form**.
4. FRIEND **receives form** and **notifies** Dispatcher
5. Dispatcher **reviews submitted information** and **creates** an **Incident** in the database by invoking the OpenIncident use case. **All information contained in form** is **automatically included** in the **Incident**. Dispatcher **selects** a **response** by **allocating resources** to the **Incident** (with AllocateResources use case) and **acknowledges** the **emergency report** with a **short message** to **FieldOfficer**
6. FRIEND **displays acknowledgement** and **selected response** to **FieldOfficer**.

Common Noun (blue)

Having verbs (yellow)

Doing verbs (red)

Modal verbs (purple)

Being Verbs (green)

Adjectives (orange)

Pros and Cons

- + Focus on users' terms
 - Model quality depends on analyst writing style
 - Natural language is inherently imprecise
 - More nouns than relevant classes
-
- Usually imply clarifying and rephrasing the scenarios and use cases with users, and use a data dictionary
 - Use of heuristics is necessary with natural language

Heuristics for Entity Objects

- Find terms that developers or users need to clarify in order to understand the flow of events
 - e.g., “information submitted by *FieldOfficer*”
 - Clarify as “EmergencyReport”
 - Note: Emergency Report is not mentioned in the use case description, but is referred to as “information submitted by the *FieldOfficer*”
- Recurring nouns in use cases
 - e.g., *Incident*
- Real world entities that the system needs to keep track of
 - e.g., *FieldOfficer*, *Dispatcher*
- Real world procedures that the system needs to keep track of
 - e.g., *EmergencyOperationsPlan*

ReportEmergency: (entity objects only)

- Dispatcher:
 - Police officer who manages Incidents ...
- EmergencyReport:
 - Initial report about an Incident from a FieldOfficer to a Dispatcher.
- FieldOfficer:
 - Police or fire officer on Duty.
- Incident:
 - Situation requiring attention from a FieldOfficer.

Heuristics for Boundary Objects

- Represent the *interfaces* between system and actors
 - Each actor interacts with at least one boundary object
 - They transform the actor information to be used by entity and control objects inside the system
 - Examples of boundary objects:
 - Forms and windows the users need to enter data into the system
 - e.g., `EmergencyReportForm`
 - Notices and messages the system uses to respond to the user
 - e.g., `AcknowledgmentNotice`
 - Data sources or sinks
 - e.g. `Printer`
 - Beware: Model the user interface at coarse level
 - Do not model the visual aspects at this stage
 - Visual aspects are dealt with by a GUI subsystem, e.g., based on SWING in Java, which is the intermediary between the user and the interface class
-

ReportEmergency : (Boundary)

- AcknowledgementNotice:
 - Notice used for displaying the Dispatcher's acknowledgement to the FieldOfficer
- DispatcherStation:
 - Computer used by the Dispatcher
- FieldOfficerStation:
 - Mobile Computer used by the Dispatcher
- ReportEmergencyButton:
 - Button used by FieldOfficer to initiate ReportEmergency use case.
- IncidentForm:
 - Form used for the creation of Incidents, presented to Dispatcher on DispatcherStation when EmergencyReport is received.
 - Not mentioned in the use case description, but dispatcher needs an interface to view emergency report
- EmergencyReportForm:
 - Form used for input of the ReportEmergency, presented to FieldOfficer on the FieldOfficerStation.

Heuristics for Control Objects

- Responsible for coordinating boundary and entity objects
 - e.g., `ElevatorController` in Elevator class diagram
- Control objects do not have usually a counterpart in real world
- Creation/Destruction (usually)
 - created when a user session or a use case scenario starts
 - ceases to exist at the end of the session or use case scenario
- Collect information from boundary objects and dispatch them to entity and application logic objects
- Collect information from entity or other control objects and dispatch them to boundary objects.

Heuristics for Control Objects

- Identify one control object per use case or more if the use case is complex
- Identify one control object per actor in the use case
 - e.g., FRIEND: *ReportEmergencyControl* for the *FieldOfficer* and *ManageEmergencyControl* for the *Dispatcher*
- The life span of a control object should be determined by the use case or extent of user session

FRIEND Example: (Control)

- *ReportEmergencyControl*:
 - Manages the report emergency reporting function on the *FieldOfficerStation*. The object is created when the *FieldOfficer* selects the “report Emergency” button.
- *ManageEmergencyControl*:
 - Manages the report emergency reporting function on the *DispatcherStation*. This object is created when an *EmergencyReport* is received.
- Two control objects for one Use Case due to distribution of *FieldOfficerStation* and *DispatcherStation*.

Cross-Checking

Cross-checking use cases and participating objects:

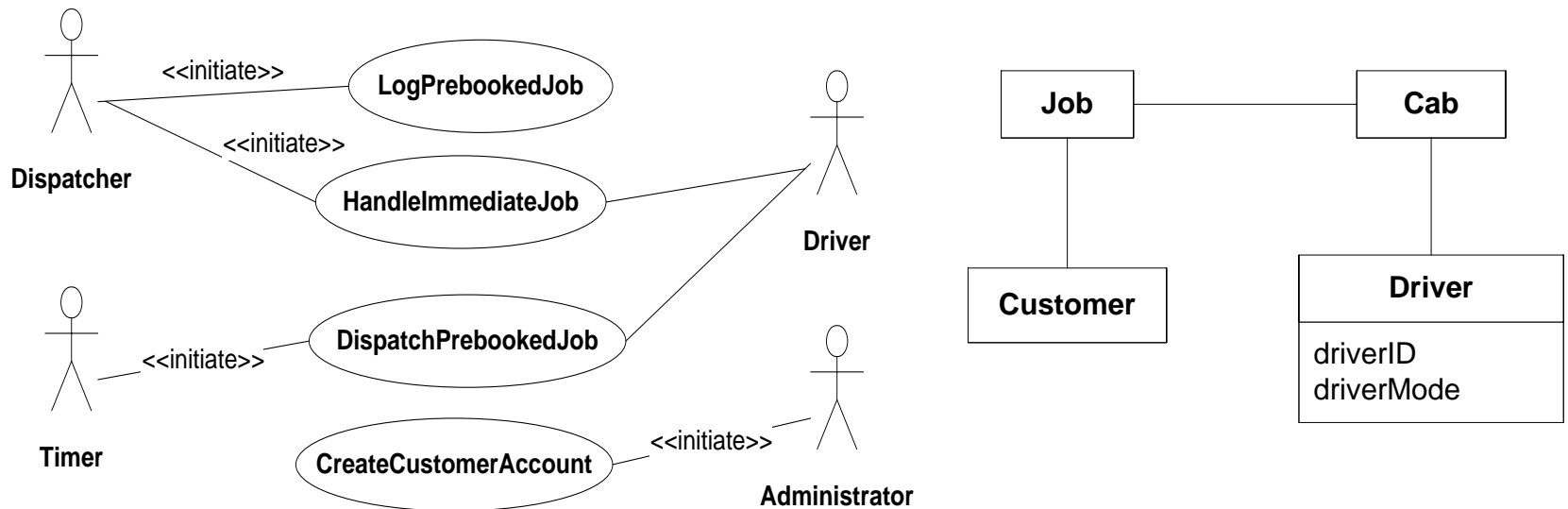
- Which use cases create this object (i.e., during which use cases are the values of the object attributes entered in the system)? Which actors can access this information?
 - Which use cases modify and destroy this object (i.e., which use cases edit or remove this information from the system)? Which actor can initiate these use cases?
 - Is this object needed (i.e., is there at least one use case that depends on this information?)
- We can use table(s) to report on such information.

Finding Classes: Advice from Authors

- Lethbridge
 - You might choose to be liberal in building the initial list of classes (keeping all possible candidates) or you might choose to be strict (keeping only if you are definite)
 - Suggestion: Be liberal. Easy to eliminate classes during a review.
 - As a rule of thumb, a class is only needed **in a domain model** if you have to store or manipulate instances of it **in order to implement a requirement**.
 - Common Difficulty: Deciding whether to have classes in a domain model that represent actors
 - Example: Security or Instant Messaging System
 - Example: Drawing Package
 - Example: Managing Corporate Accounts.

Classes for actors? Example from Cab Lab

- From the Cab dispatching system:
 - We should have a dispatcher class (because we have a driver class and they are both actors)
 - Not necessarily: only if the system has to store data about the dispatcher (e.g., id, password)
 - We should have a customer actor (because we have a customer class)
 - No: the customer is not interacting with the system (unless paying with credit card)



SYSC-3120—Software Requirements Engineering

- Overview
- Analysis Concepts
 - Modeling structure: class diagram
 - Modeling interaction: sequence diagram
 - Modeling state-based behaviour: state machine diagram
- **Analysis Process**
 - Finding objects/classes (heuristics)
 - **Finding relationships : associations and attributes (heuristics)**
 - Interactions/behavior (heuristics)
 - Responsibilities and consistency
 - Analysis review

Identify Associations

- Identify classes that need to know about another class instances,
 - e.g., they create, access, destroy instances of that class
 - e.g., *EmergencyReport* can be created by *FieldOfficer*
- Association properties:
 - name, roles, multiplicities, navigation
- An iterative process :
 - Initial identification
 - Then, refinement (analyzing and verifying the associations)
- For every association, ask yourself : Is it relevant to the application ?
 - Is it needed to implement some requirement ?
 - If there is no requirement, you are simply complicating the model.

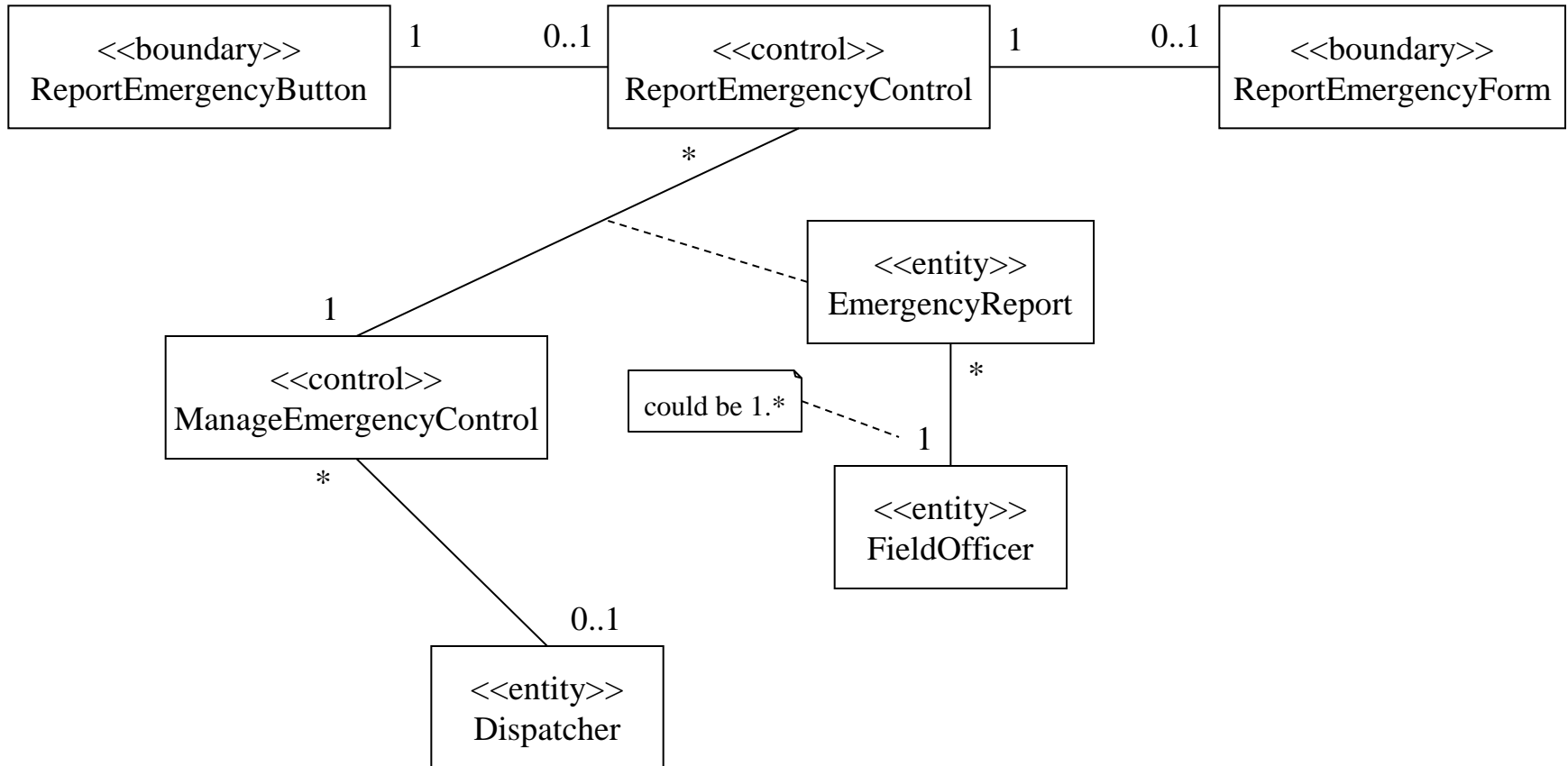
Association Heuristics: Initial Identification

- Start with class(es) that are most central to the system.
 - Start with associations between entity classes.
 - Work outwards.
- Initially, examine *verb phrases* in scenarios and Use Case descriptions
 - e.g., `FieldOfficer submits an EmergencyReport`
- Name associations and Roles precisely
 - If you omit, association defaults to “has” (not always informative)
 - Add sufficient names to make the association clear and unambiguous.
- Do not worry about multiplicity until the set of associations is stable
 - In general, take a non-restrictive approach to multiplicity
 - Begin with *, rather than 1..n
 - Don't worry about part-whole (aggregation vs composition)
- Do not worry about directionality of association, until design
 - By default, associations are bi-directional.

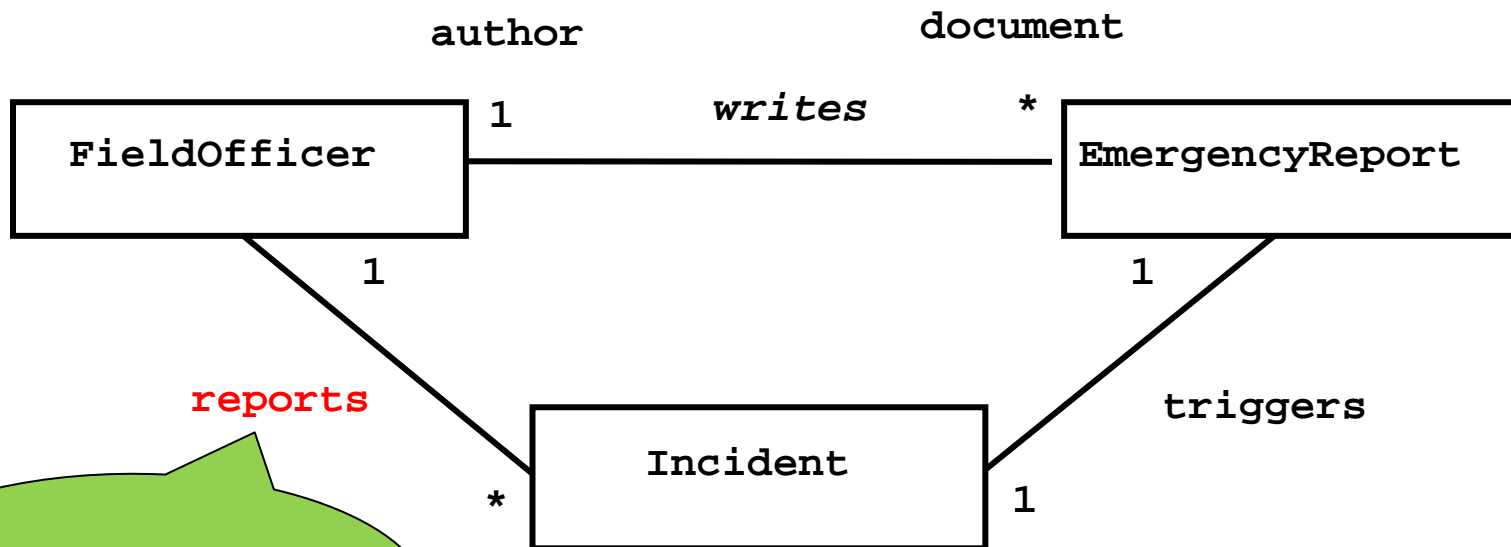
Association Heuristics: Refinement

- Eliminate associations that can be derived from other associations (avoid redundancy during Analysis)
- Analyze multiplicity
 - Read every association in both directions. Does it make sense ?
 - Consider association class for any many-to-many association
- Analyze each entity class to see how it is identified
 - Most entity objects have an identifying characteristics
 - Use *qualifiers* as often as possible to identify key attributes
 - Reduces multiplicity values
- Use sequence diagrams
 - Uncover missing associations (between objects exchanging messages)
 - An association is legitimate only if its links survive beyond the execution of a single operation
 - If information does not need to be stored, perhaps you can eliminate the association

Identify Associations



FRIEND Example



reports

Is this association necessary?
Should it exist after the
incident was created?

Identify Attributes

- Attributes are properties of individual objects
 - Property is a partial aspect of an object
- Some nouns become classes, and some become attributes
 - E.g., “the **name** of the **tournament...**”
- Some verbs correspond to associations, but not all
 - “... complete the form by **specifying** the **type**”
- Identify associations before attributes
 - Do not confuse associated objects and attributes
- For every attribute, ask yourself : Is it relevant to the application ?
 - Is it needed to implement some requirement ?
 - If there is no requirement, you are simply complicating the model.
- Attributes are the least stable part of analysis object model

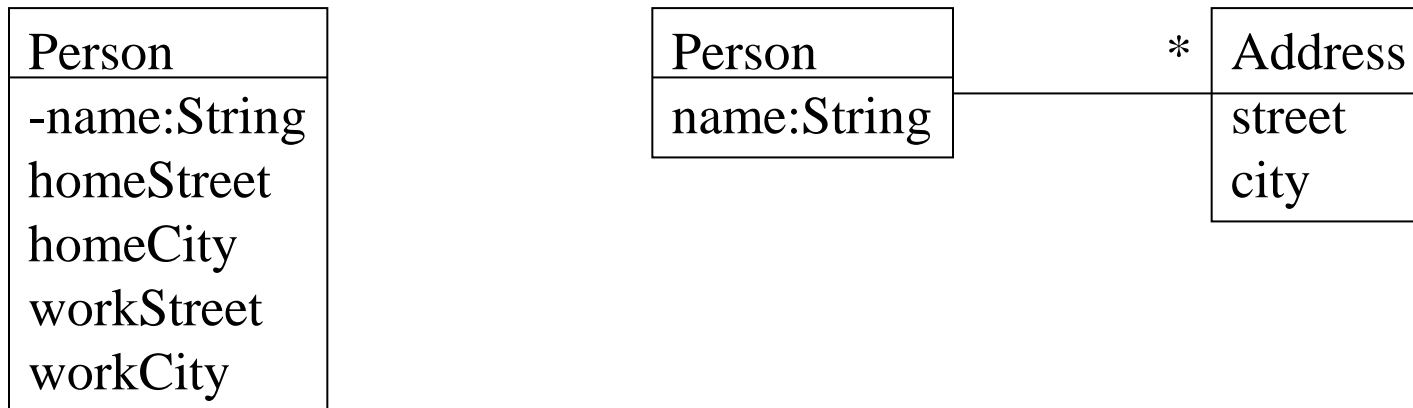
FRIEND Example

EmergencyReport
<code>emergencyType: {fire, traffic, other}</code> <code>location: String</code> <code>description: String</code>

- Describe each attribute in data dictionary
 - name, brief description, type (legal values)

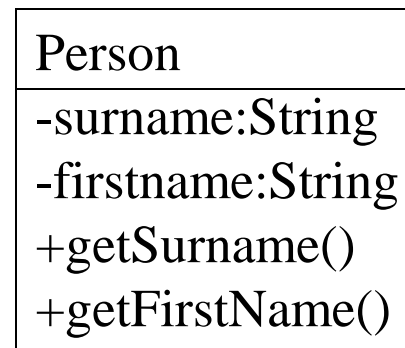
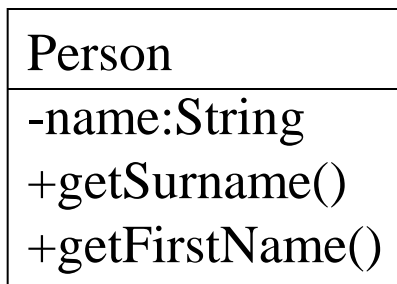
Attribute Heuristics (I)

- Properties generally have simple types (or at least conceptually atomic)
 - If attribute is an object, use association instead
 - Exceptions: address, date
 - e.g., *FieldOfficer* who authored an *EmergencyReport*
- Word analysis:
 - possessive phrases (e.g., the description OF THE emergency)
 - adjective phrases: (e.g., the emergency description)
- Nouns that are collections are associations, not attributes
 - Attribute name should not be plural



Attribute Heuristics (II)

- Attributes should not have an implicit internal structure.

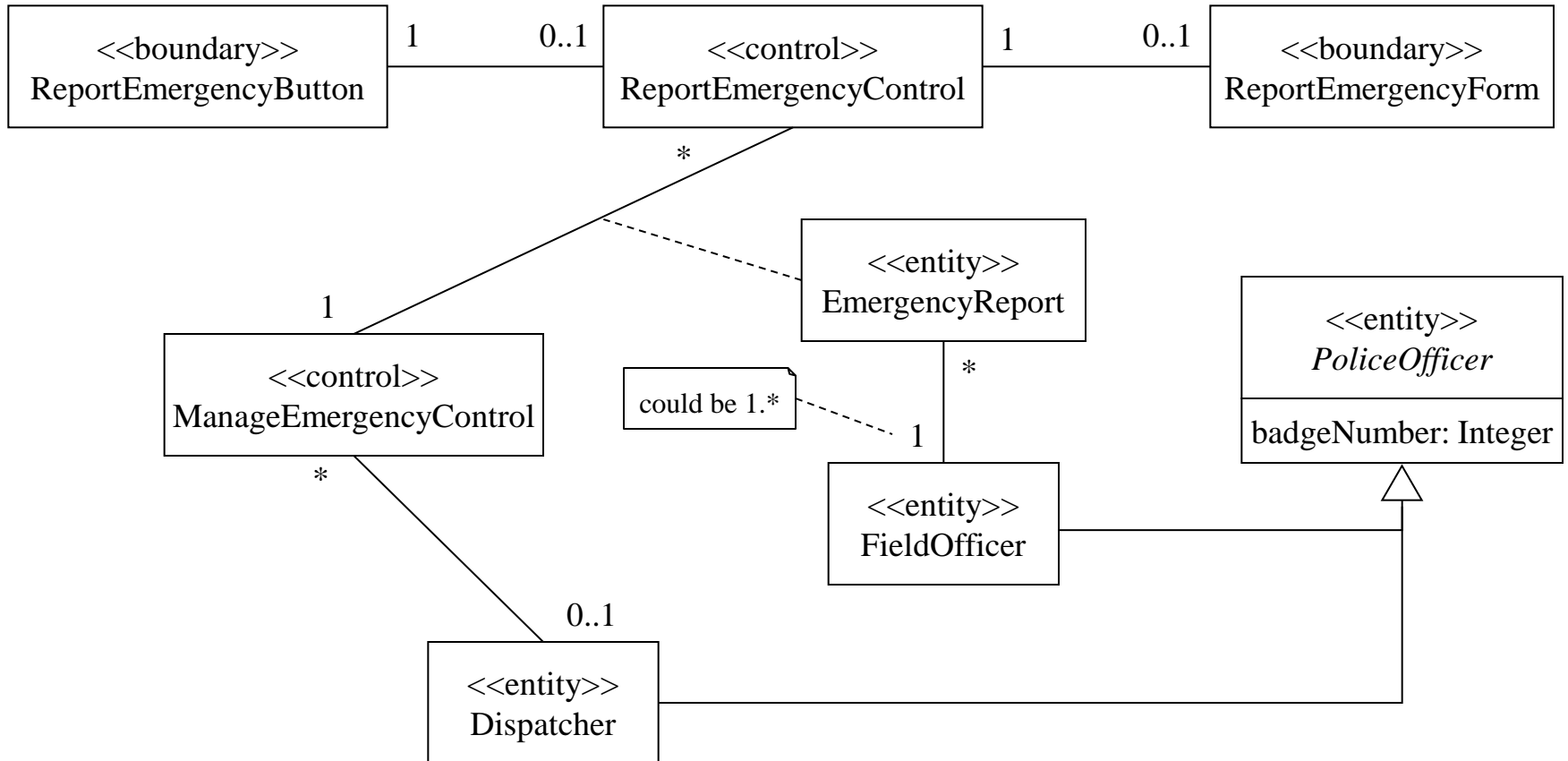


- Represent stored state as attribute of entity object

Generalization

- Eliminate redundancy in the analysis model
- Share attributes, operations, associations
- Example:
 - *Dispatchers* and *Fieldofficers* both have *badgeNumber* to identify them within the city. They are both *PoliceOfficer*
 - Abstract *PoliceOfficer* class, containing common functionality and attributes
 - UML Class diagram notation: abstract class name in italics

PoliceOfficer



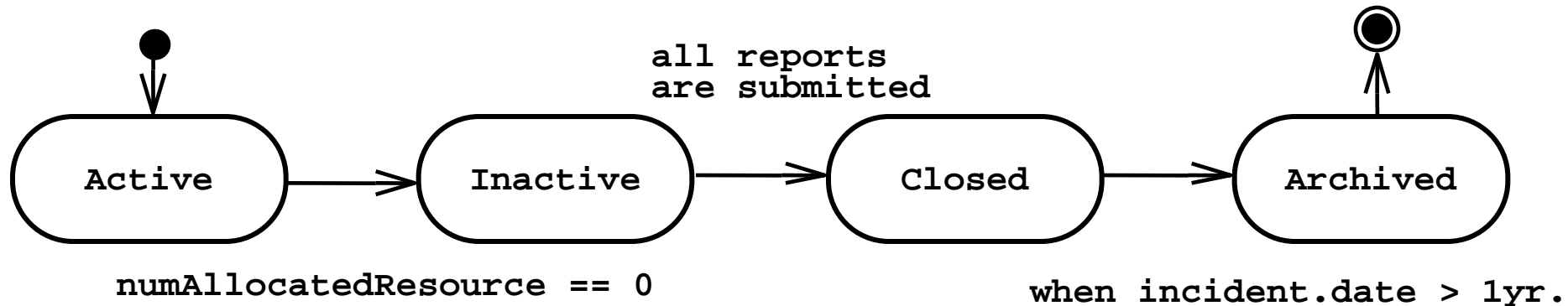
SYSC-3120—Software Requirements Engineering

- Overview
- Analysis Concepts
 - Modeling structure: class diagram
 - Modeling interaction: sequence diagram
 - Modeling state-based behaviour: state machine diagram
- **Analysis Process**
 - Finding objects/classes (heuristics)
 - Finding relationships: associations and attributes (heuristics)
 - **Interactions/behavior (heuristics)**
 - Responsibilities and consistency
 - Analysis review

Modeling Object Behavior

- Sequence diagrams represent behavior from the perspective of single use case
 - Shows how the behaviour of a use case is distributed among participating objects.
- Statechart diagrams capture behavior from the perspective of a single object
 - Focus only on objects with non-trivial behavior (multi-modal, state-dependent)
- Help identify missing Use Cases
- Help identify missing objects and/or operations
- Build more formal description of the object behavior

Incident Statechart



- Are there Use Cases for documenting, closing, and archiving Incidents?
- The Active state could be further refined and decomposed: nested statecharts
- Transitions conditions can and should be described more formally. (OCL – see later)

Use of Object Types (cont.)

In sequence diagrams

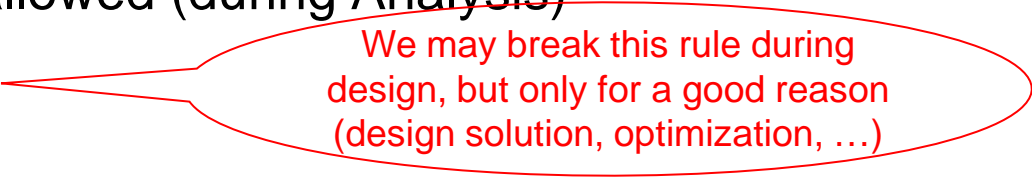
- **Boundary** objects communicate with **Control** objects.
- **Control** objects communicate with **Boundary**, **Control** and **Entity** objects.
- **Entity** objects communicate with other **Entity** objects.

These suggest directions of messages in sequence diagrams:

- Boundary → Control
- Control → Boundary, Control → Control, Control → Entity
- Entity → Entity

But the following are not allowed (during Analysis)

- Boundary → Entity
- Entity → Control
- Entity → Boundary



We may break this rule during design, but only for a good reason (design solution, optimization, ...)

This suggests that Control objects transform information received from Entity objects and send it to Boundary objects.

Sequence Diagram vs Pre/Postconditions

- Pre/postconditions specify responsibilities of communicating operations.
- Sequence diagram shows some responsibilities of interacting operations.
- Sequence of messages must match the pre/postconditions of operations!

Library System (excerpt)

Operation `borrowCopy(uid, cid)` in class Library

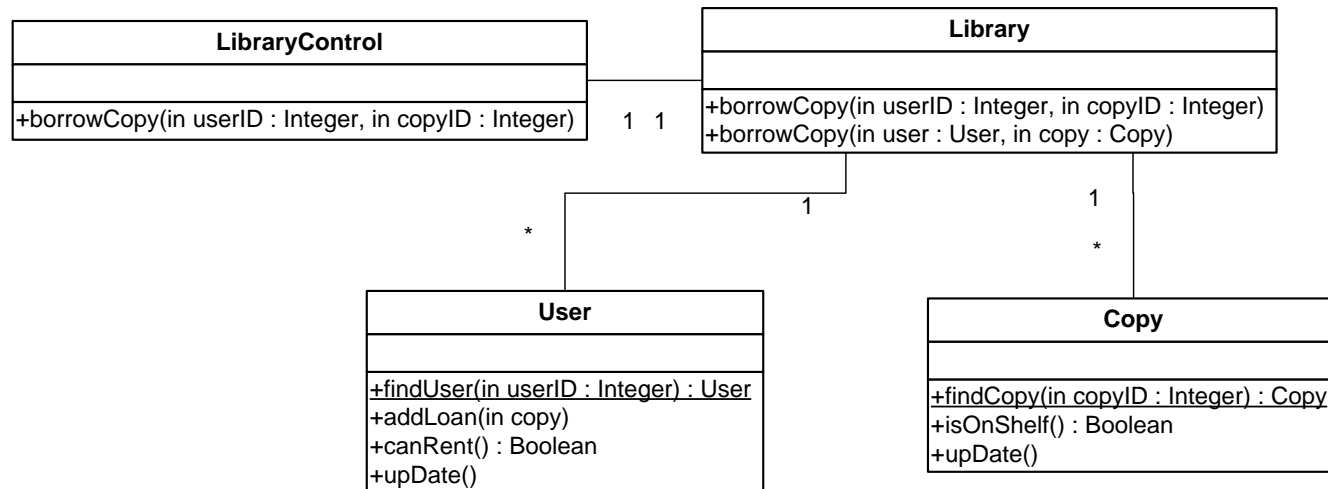
Precondition:

- There is a user with id `uid` who has not reached his/her maximum number of allowed rentals.
- There is a book with id `cid` on shelves (ready to borrow).

Postcondition:

- There is no book on shelves with id `cid`.
- There is a loan object for a book with id `cid`.
- The user with id `uid` is borrowing one more copy and is linked to a loan for a book with id `cid`.

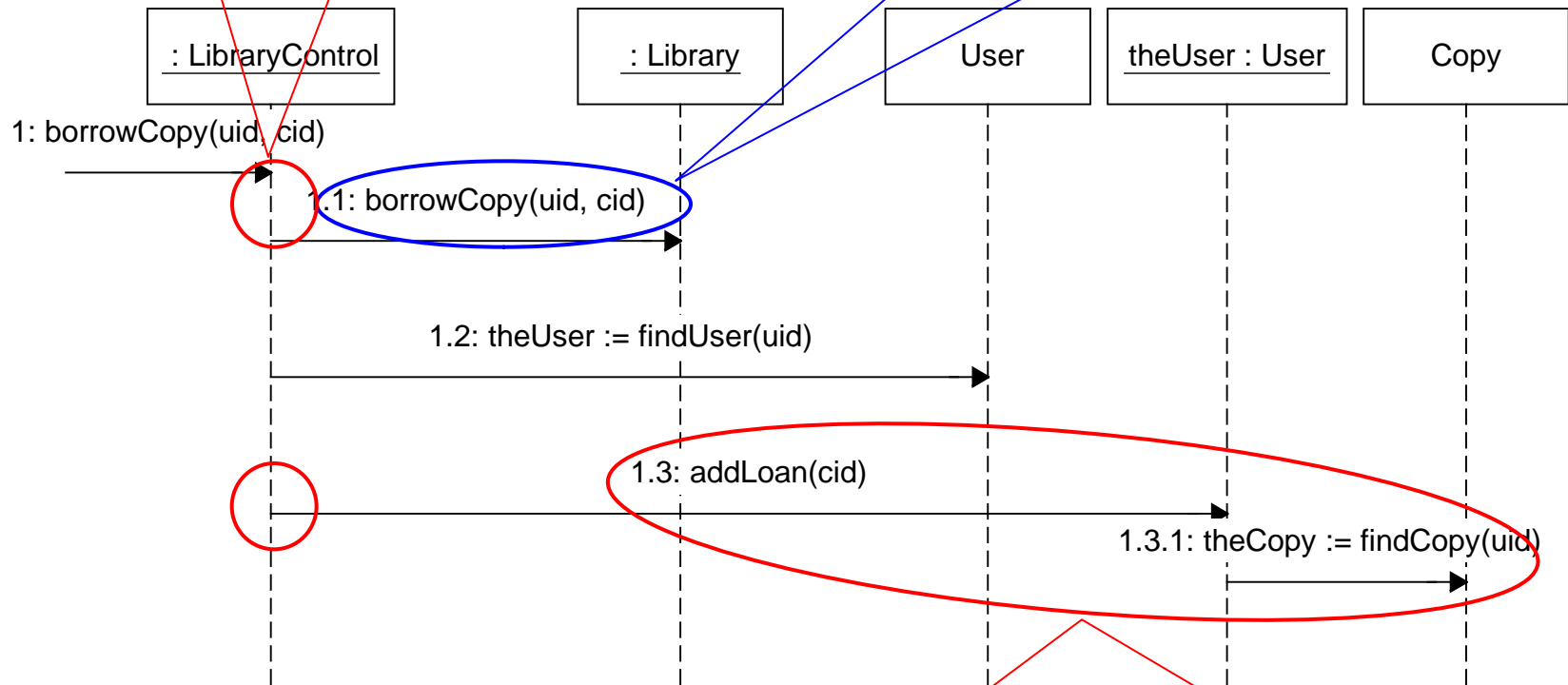
Can you write these in OCL? (see slide #92)



Sequence Diagram for Borrowing (excerpt)

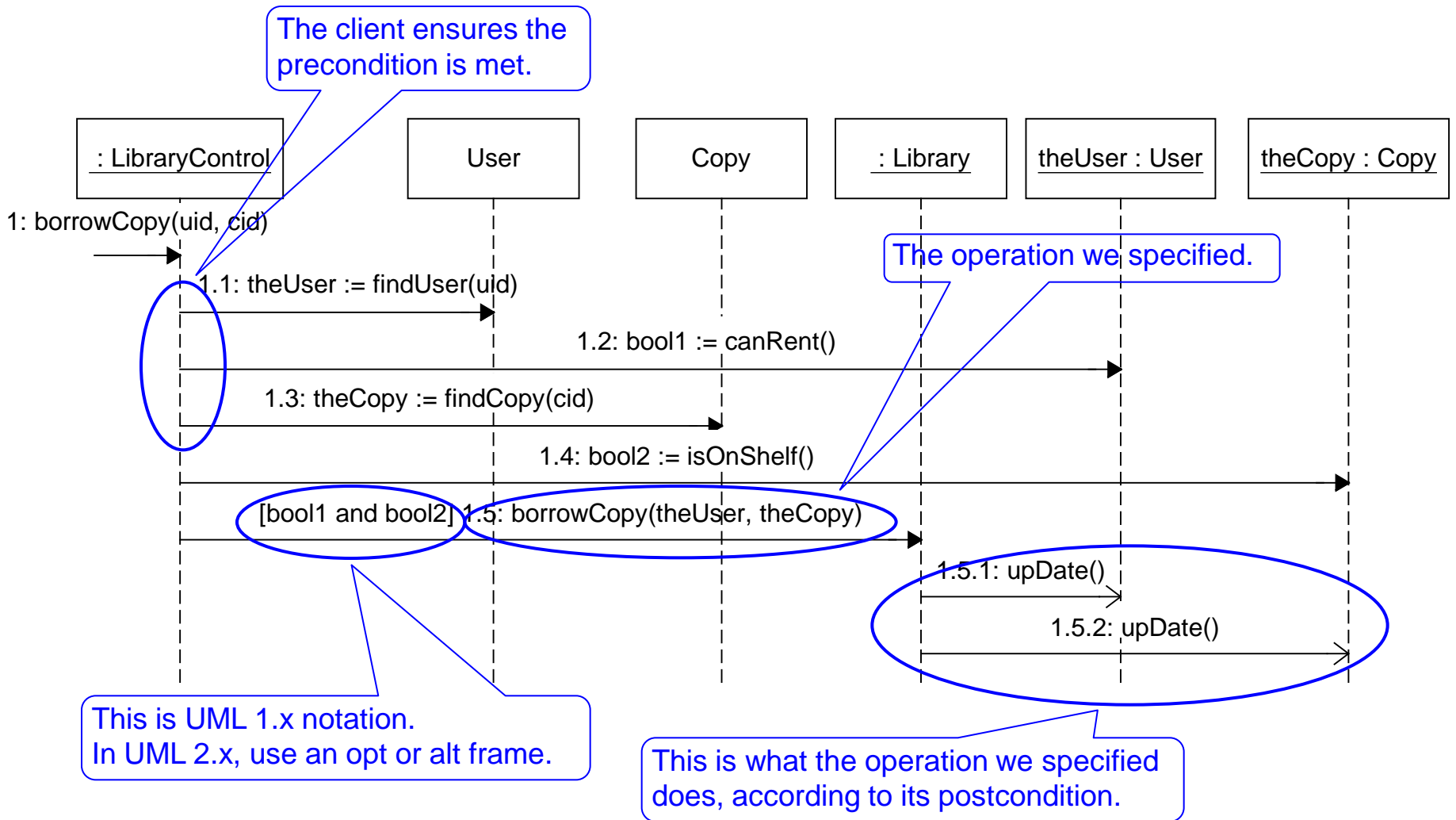
Nothing indicates that the client ensures the precondition is met!

This is the operation we just specified with a precondition and a postcondition.



Setting links between the user, the copy and the new loan is the responsibility of the operation we specified, not its client!

Improved Sequence Diagram for Borrowing (excerpt)

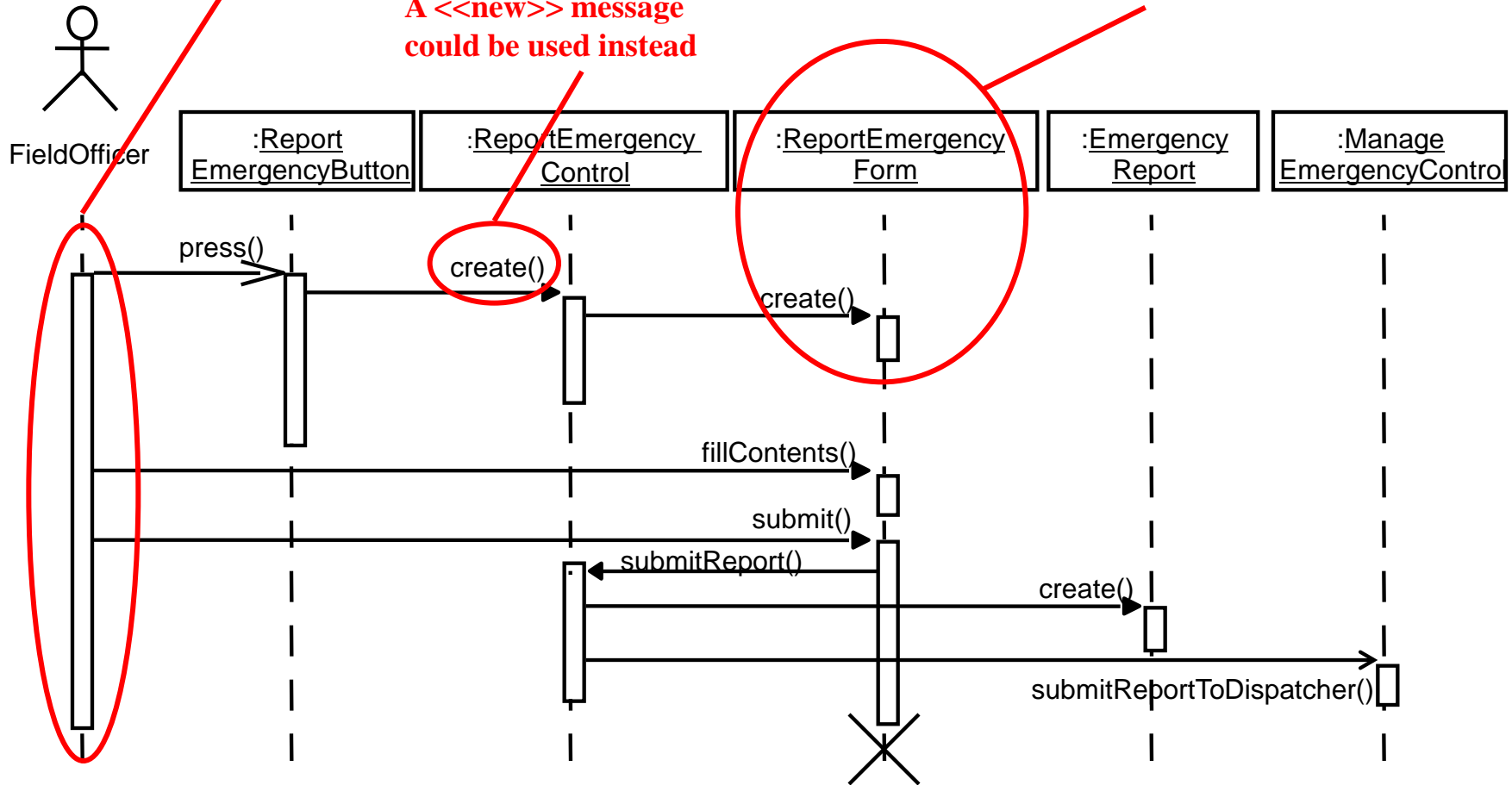


ReportEmergency SD (1)

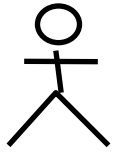
An activation bar for the actor does not make sense

A <<new>> message could be used instead

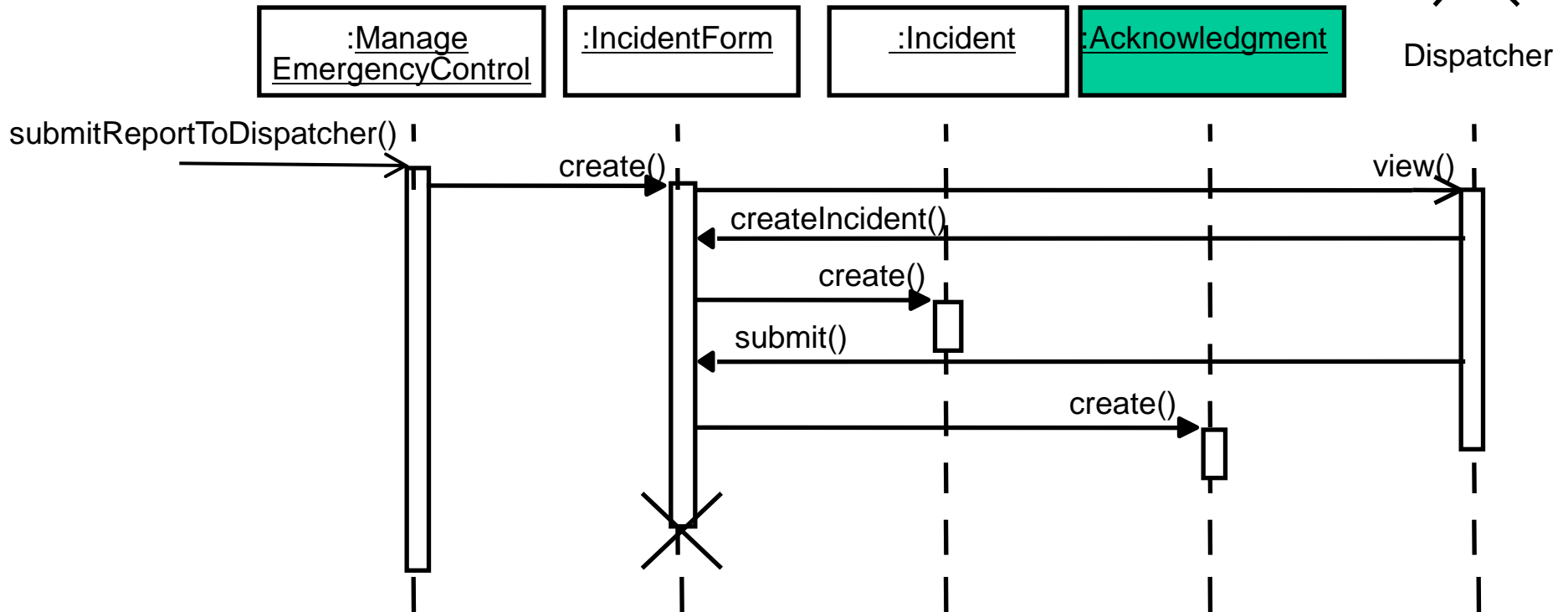
Object should only appear when it is created



ReportEmergency SD (2)

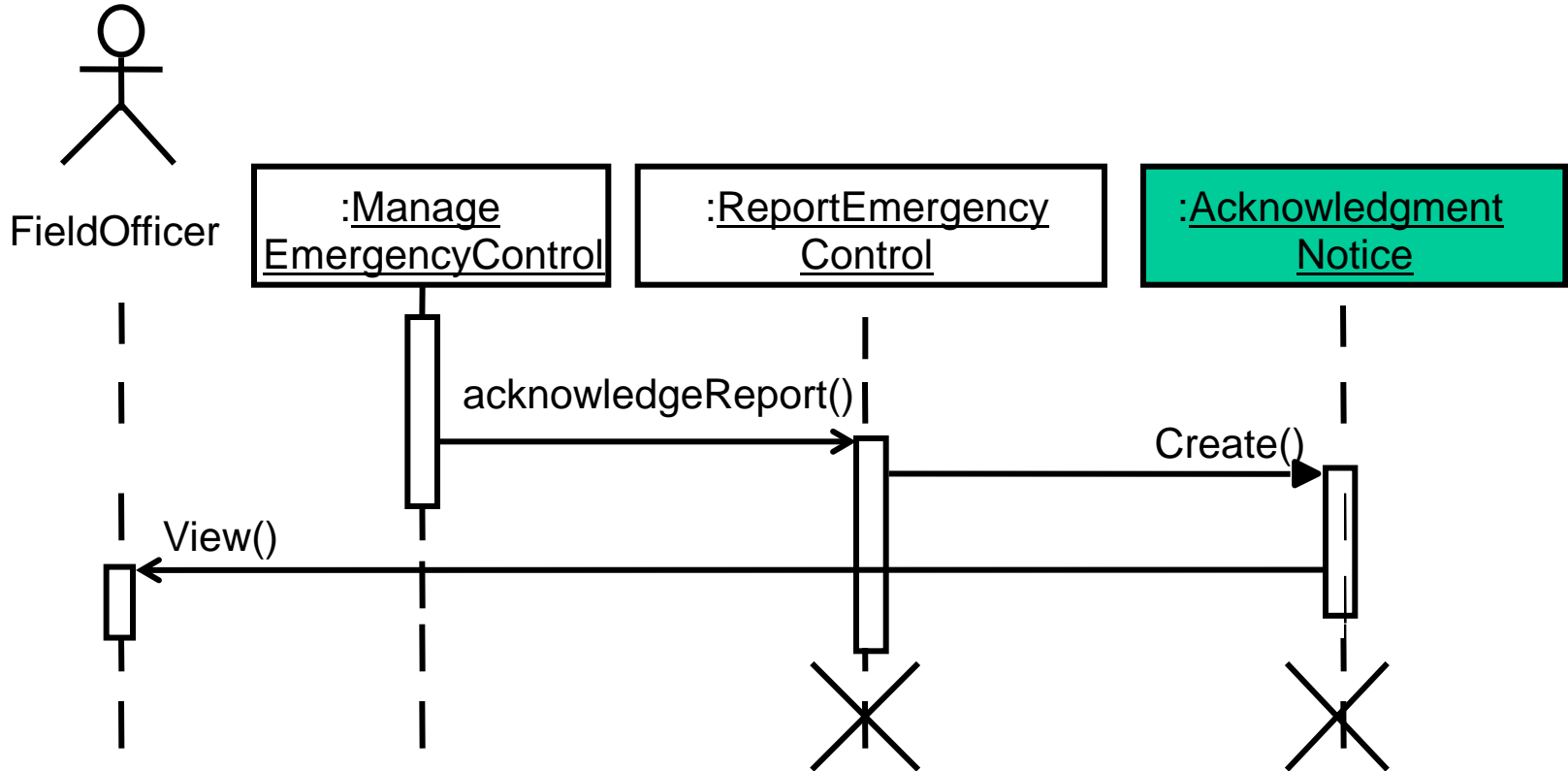


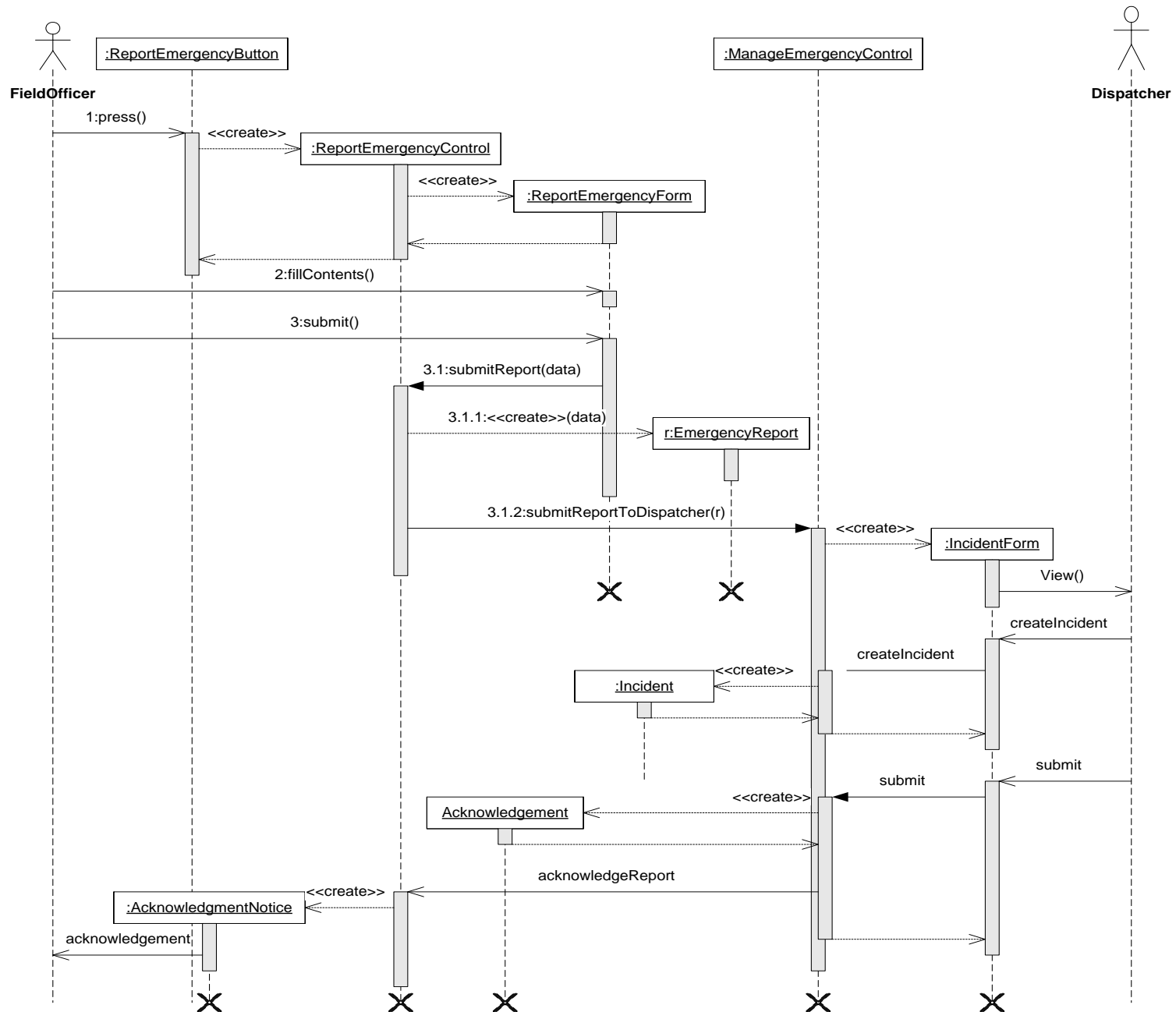
Dispatcher



- New entity object Acknowledgment that we forgot during our initial examination of the ReportEmergency use case.
 - It holds the information associated with an Acknowledgment (Entity object) and is created before the AcknowledgementNotice boundary object (next slide).
- We also need to clarify with the user what information is contained by an acknowledgement.

ReportEmergency SD (3)





Heuristics for arranging lifelines in SD

- Arranging lifelines
 - First column: **actor** who initiates the use case
 - Second column: **boundary object**
 - Third column: **control object** that manages the rest of the use case
- Object creation
 - Control objects are created by boundary objects initiating use cases
 - Other boundary objects are created by control objects
- Access to objects
 - Entity objects are accessed by control and boundary objects
 - Entity objects never access boundary or control objects

Change to ReportEmergency

- New Entity object:
 - *Acknowledgment* – Response of a *Dispatcher* to a *FieldOfficer*' s *EmergencyReport*. Contains resources allocated, predicted arrival time ...
- Modify Step 4 of *ReportEmergency* flow of events' description:
 - The acknowledgment indicates to the *FieldOfficer* that the *EmergencyReport* was received, an *Incident* created, and resources allocated to the *Incident*.

SYSC-3120—Software Requirements Engineering

- Overview
- Analysis Concepts
 - Modeling structure: class diagram
 - Modeling interaction: sequence diagram
 - Modeling state-based behaviour: state machine diagram
- **Analysis Process**
 - Finding objects/classes (heuristics)
 - Finding relationships: associations and attributes (heuristics)
 - Interactions/behaviour (heuristics)
 - **Responsibilities and consistency**
 - Analysis review

Specifying Responsibilities

- Sequence diagrams imply we distribute the behavior of the use case across participating objects.
 - An operation is responsible for interacting with some other object
- Responsibilities under the form of operations, are assigned to objects
- These operations may be shared by several Use Cases:
 - remove redundancies but consistency needs to be checked
- During analysis, sequence diagrams only focus on high-level behavior
 - implementation issues should not be addressed at this point

Specifying Responsibilities (cont.)

- *Pre-condition*: Conditions under which operations can be executed and yield a correct result
- *Post-Condition*: Conditions that are guaranteed true after execution of an operation
- *Class invariant*: Conditions that must remain true, at all times, for any instance of a class
- *Contract*: All of the above are referred to as a contract

Specifying Responsibilities (cont.)

- Class diagram
 - Shows which attributes are the responsibility of which class
 - Shows which operations are the responsibility of which class

 - But also

 - Multiplicities show what are the responsibilities of objects to maintain links to other objects
- State machine diagram
 - Show state-based behaviour of a given object

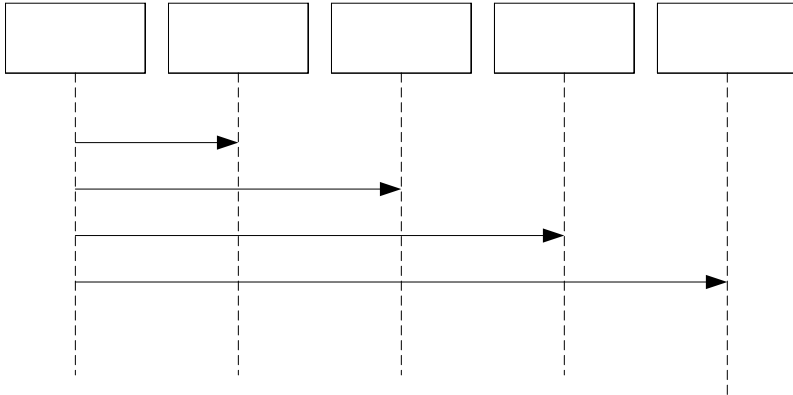
What else can we get out of sequence diagrams?

- Sequence diagrams are derived from the use cases. We therefore see the structure of the use cases.
- The structure of the sequence diagram helps us to determine how decentralized the system is.
- We distinguish two structures for sequence diagrams:
 - fork diagrams
 - stair diagrams (Ivar Jacobsen)

Fork and Stair Diagram

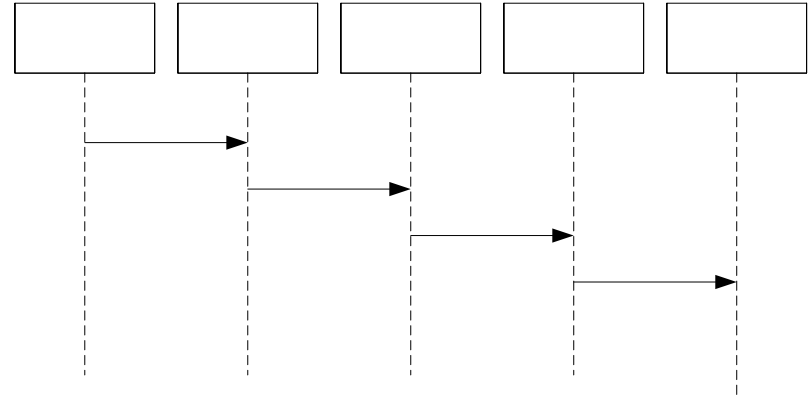
Fork Diagram

- Much of the dynamic behavior is placed in a single object, usually the control object.
- The control object knows all the other objects and often uses them for direct questions and commands.



Stair Diagram

- The dynamic behavior is distributed.
- Each object delegates some responsibility to other objects.
- Each object knows only a few of the other objects and knows which objects can help with a given behavior.



Fork or Stair?

- Which of these diagram types should be chosen?
- Object-oriented fans claim that the stair structure is better
 - The more the responsibility is spread out, the better
- However, this is not always true.
- Better heuristics for deciding between fork and stairs:
 - Decentralized control structure
 - The operations introduce a strong connection between caller and callee
 - It is harder to change the order of the operations (change involves many objects)
 - Centralized control structure (better support for change)
 - The operations can easily **change** order
 - New operations can be easily inserted as a result of new requirements

Cross-Checking

- Sequence diagrams can be used to help check the completeness / correctness of the use case model and class diagrams.
- Cross-checking:
 - Which Use Cases create this object? Which actors can access this information?
 - Which Use Cases modify and destroy this object? Which actors initiate these Use Cases?
 - Is this object needed? (at least one Use Case depends on this information)

Consistency Between UML Views

- UML views
 - Use case diagram
 - Use case descriptions
 - Class diagram(s)
 - Sequence diagram(s)
 - State machine diagram(s)
 - Data dictionary
- These view consider only one aspect of the system being built (separation of concern)
 - Functionalities (use case diagram + use case descriptions)
 - Structure (class diagram)
 - Interactions between objects(sequence diagrams)
 - State-based behaviour of an object (state machine diagrams)
 - Documentation: all the above + data dictionary
- Only **one** system is being built
 - Those views **must** be kept in sync.

Consistency Rules (excerpt)

	Use case diagram	Use case descr.	Sequence diagram	Class diagram	State machine diagram	Data dictionary
Use case diagram	NA	One use case = one description + Use case diagram matches description (e.g., actors)	One use case = one or more sequence diagrams + Use case diagram matches sequence diagram (e.g., actors)			Actors and use cases must appear in data dictionary
Use case descr.	See symmetric cell	NA	Flow of steps matches flow of messages	Entity classes, operations and attributes appear in uses case description		
Sequence diagram	See symmetric cell	See symmetric cell	NA	Objects must be instances of classes in class diagram + Need for association = need for message + Messages must appear in class diagram (operations or signals)	Scenarios in statecharts and sequence diagrams should match	Scenarios and contracts should match
Class diagram		See symmetric cell	See symmetric cell	NA	Operations, attributes, navigations used in state diagram found in class diagram	Classes, attributes, operations (including contracts) described
Statechart			See symmetric cell	See symmetric cell	NA	
Data dictionary	See symmetric cell		See symmetric cell	See symmetric cell	See symmetric cell	NA

SYSC-3020—Introduction to Software Engineering

- Overview
- Analysis Concepts
 - Different kinds of classes
 - Different kinds of relationships
 - Modeling interaction
 - Modeling state-based behaviour
- **Analysis Process**
 - Finding objects/classes (heuristics)
 - Finding relationships: associations and attributes (heuristics)
 - Interactions/behaviour (heuristics)
 - Responsibilities
 - **Analysis review**

Analysis Review - Correctness

- Is the data dictionary understandable by the user?
- Do abstract classes correspond to user-level concepts?
- Are all descriptions in accordance with the user's definitions
- Do all entity and boundary objects have meaningful noun phrases as names?
- Do all use cases and control objects have meaningful verb phrases as names?
- Are all error cases described and handled?
- Are system administration functions of the system described?

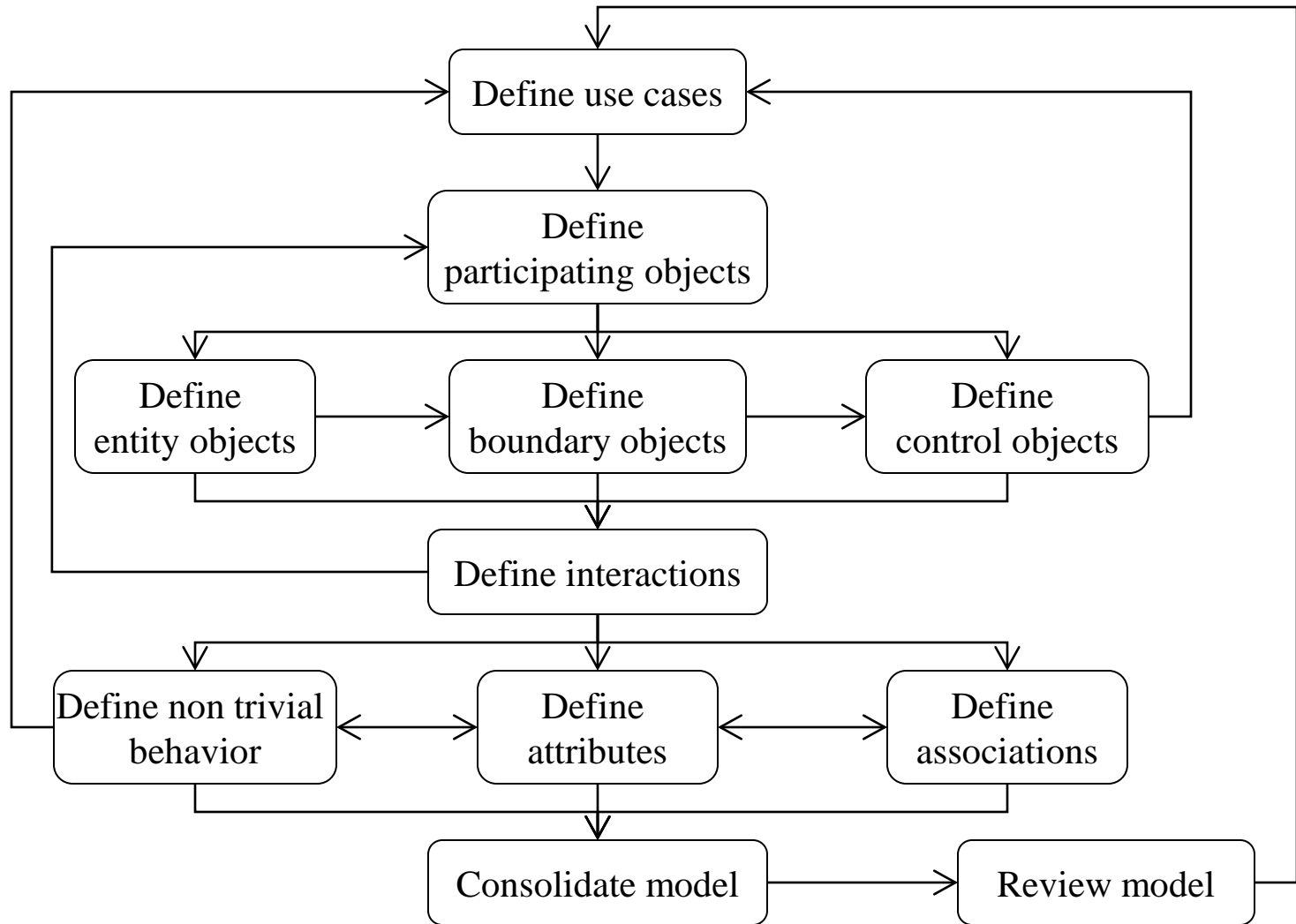
Analysis Review - Completeness

- For each object:
 - Is it needed by any use case? Where is it created, modified, destroyed?
- For each attribute:
 - When is it set? What is its type? Should it be a qualifier?
- For each association:
 - When is it traversed? Why was the specific multiplicity chosen? Can associations with one-to-many and many-to-many multiplicities be qualified?
- For each control object:
 - Does it have the necessary associations to access the objects participating in its corresponding use case?

Review-Consistency

- Are there multiple classes or use cases with the same name?
- Do model elements with similar names denote similar phenomena?
- Are all model elements described at the same level of detail?
- Are there classes with similar attributes and associations that are not in the same generalization hierarchy?

Analysis Activities

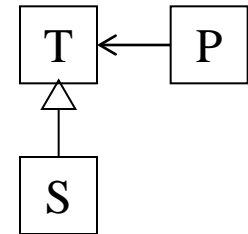


SYSC-3120—Software Requirements Engineering

- Overview
- Analysis Concepts
 - Modeling structure: class diagram
 - Modeling classes
 - Different kinds of relationships
 - OCL: Better specifying operation/classes, constraining class diagram
 - Liskov substitution principle
 - Modeling interaction: sequence diagram
 - Modeling state-based behavior: state machine diagram
- Analysis Process
 - Finding objects/classes (heuristics)
 - Finding relationships : associations and attributes (heuristics)
 - ...

Liskov Substitution Principle

- This principle defines the notions of generalization / specialization in a formal manner
- Class S is correctly defined as a specialization of class T if the following is true:
 - for each object s of class S there is an object t of class T such that the behavior of any program P defined in terms of T is unchanged if s is substituted for t (i.e., t is replaced by s).
- S is said to be a *subtype* of T
- All instances of a subclass can stand for instances of a superclass without any effect on client classes
- Any future extension (new subclasses) will not affect existing clients.



Lack of Substitutability

```
class Rectangle : public Shape {
private: int w, h;
public:
    virtual void set_width(int wi) {
        w=wi;
    }
    virtual void set_height(int he) {
        h=he;
    }
}
```

```
class Square : public Rectangle {
public:
    void set_width(int w) {
        Rectangle::set_height(w);
        Rectangle::set_width(w);
    }
    void set_height(int h) {
        set_width(h);
    }
}
```

```
void foo(Rectangle *r) { // This is the client
    r->set_width(5);
    r->set_height(4);
    assert((r->get_width()*r->get_height()) == 20); // Oracle
}
```

- If `r` is instantiated at run time with instance of `Rectangle`, the assertion is true.
 - If `r` is instantiated at run time with instance of `Square`, the behavior observed by client is different (`width*height == 16`)
 - May lead to problems
 - `Square` should be defined as subclass of `Shape`, not `Rectangle`
-

Rules

A number of rules must hold (have to be checked) to have substitutability

- *Signature Rule*:
 - The subtypes must have all the methods of the supertype, and the signatures of the subtypes methods must be *compatible* with the signatures of the corresponding supertypes methods
 - In Java, this is enforced as the subtype must have all the supertype methods, with identical signatures except that a subtype method can have fewer exceptions (compatibility stricter than necessary here)
- *Method Rule*:
 - Calls on these subtype methods must “behave like” calls to the corresponding supertype methods.
- *Properties Rule*:
 - The subtype must preserve the invariant of the supertype.

Method Rule

Method Rule can be expressed in pre- and post-conditions.

- The precondition is *weakened*
 - Weakening the precondition implies that the subtype method requires less from the caller
 - If methods $T::m()$ and $S::m()$ (overriding) have preconditions $PrC1$ and $PrC2$, respectively, then $PrC1 \Rightarrow PrC2$
 - The postcondition is *strengthened*
 - Strengthening means the subtype method returns more than the supertype method
 - If methods $T::m()$ and $S::m()$ (overriding) have postconditions $PoC1$ and $PoC2$, respectively, then $(PrC1 \wedge PoC2) \Rightarrow PoC1$
- The calling code depends on the postcondition of the supertype method, but only if the precondition is satisfied.

Liskov – Example 1: IntSet

```
public class IntSet {
    private Vector els; // the elements
    public IntSet() {...}
        // Post: Initializes this to be empty
    public void insert (int x) {...}
        // Post: Adds x to the elements of this
    public void remove (int x) {...}
        // Post: Remove x from the elements of this
    public boolean isIn (int x) {...}
        //Post: If x is in this returns true else returns false
    public int size () {...}
        //Post: Returns the cardinality of this
    public boolean subset (IntSet s) {...}
        //Post: Returns true if this is a subset of s else returns false
}
```

Liskov – Example 1: MaxIntSet

```
public class MaxIntSet extends IntSet {
    private int biggest; // biggest element if set not empty
    public MaxIntSet () {...} // call super()
    public Max () throws EmptyException {...}
        // new method
    public void insert (int x) {...}
        // overrides InSet::insert()
        // Additional Post: update biggest with x if x > biggest
    public void remove (int x) {...}
        // overrides InSet::remove()
        // Additional Post: update biggest with next biggest element in
        // this if x = biggest
}
```

We see that the post-conditions for `insert()` and `remove()` are stronger, that is they contain (imply) the post-conditions of the methods they override in the parent class.

Liskov – Example 2: LinkedList and Set

```
public class LinkedList {
    ...
    /** Adds an element to the end of the list
     * PRE:  element != null
     * POST: this.getLength() == old.getLength() + 1
     *       && this.contains(element) == true
     */
    public void addElement(Object element) { ... }
    ...
}
public class Set extends LinkedList {
    ...
    /** Adds element, provided element is not already in the set
     * PRE:  element != null && this.contains(element) == false
     * POST: this.getLength() == old.getLength() + 1
     *       && this.contains(element) == true
     */
    public void addElement(Object element) { ... }
    ...
}
```

The method rule is transgressed here: the precondition of `Set::addElement()` is stronger than the precondition of `LinkedList::addElement()`.

Properties Rule

- All methods of the subtype must preserve the invariant of the supertype
- The invariant of the subtype must imply the invariant of the supertype
- Example:
- Assume `FatSet` is a set of integers whose size is always at least 1.
 - The constructor and remove methods ensure this.
 - `Inv: FatSet.allInstances->size >= 1`
- `ThinSet` is also a set of integers but it can be empty and therefore cannot be a legal subtype of `FatSet`.

Property rule for InSet, MaxInSet

- Informal invariant description:
- Invariant of `InSet`, for any instance `i` :
 - `i.els != null` and
 - all elements of `i.els` are `Integers` and
 - there are no duplicates in `i.els`
- Invariant of `MaxIntSet`, for any instance `i` (assuming that `MaxIntSet` uses the iterator of `InSet` to traverse `els`) :
 - invariant of `InSet` and
 - `i.size > 0` and
 - for all integers `x` in `els`, `x <= i.biggest`
- The invariant of `MaxInSet` includes the invariant of `InSet` and therefore implies it.
- We comply with the property rule.