SYSC-3020—Introduction to Software Engineering

Part III - Object-Oriented Analysis

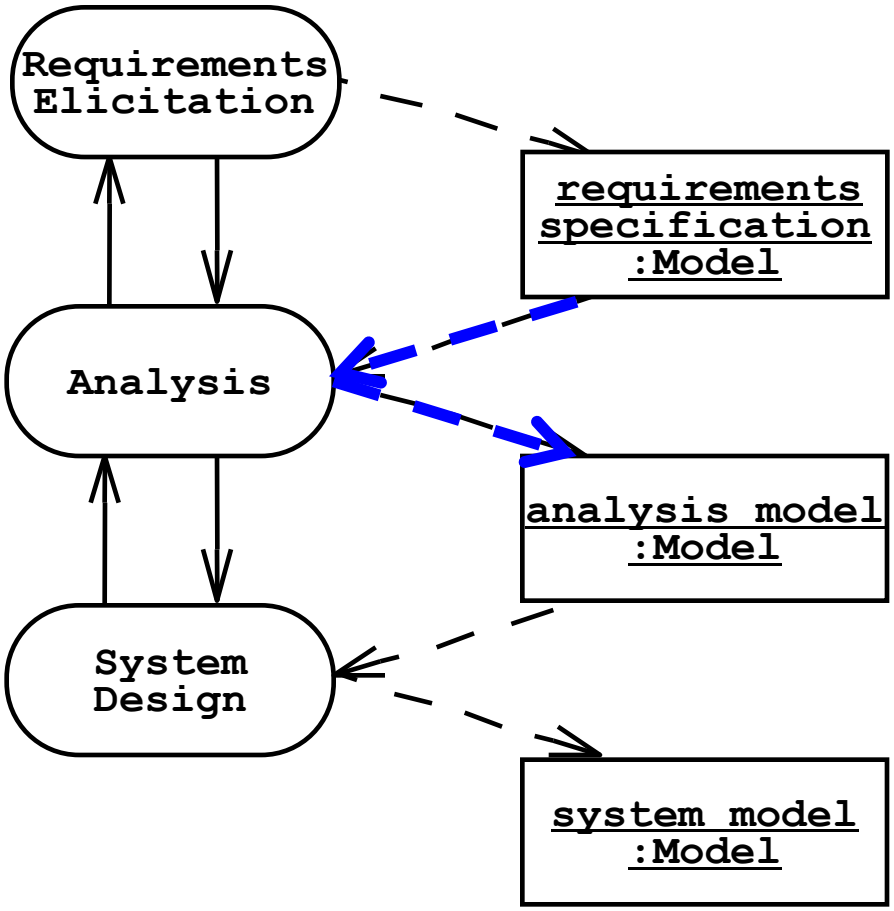
SYSC-3020—Introduction to Software Engineering

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

Overview

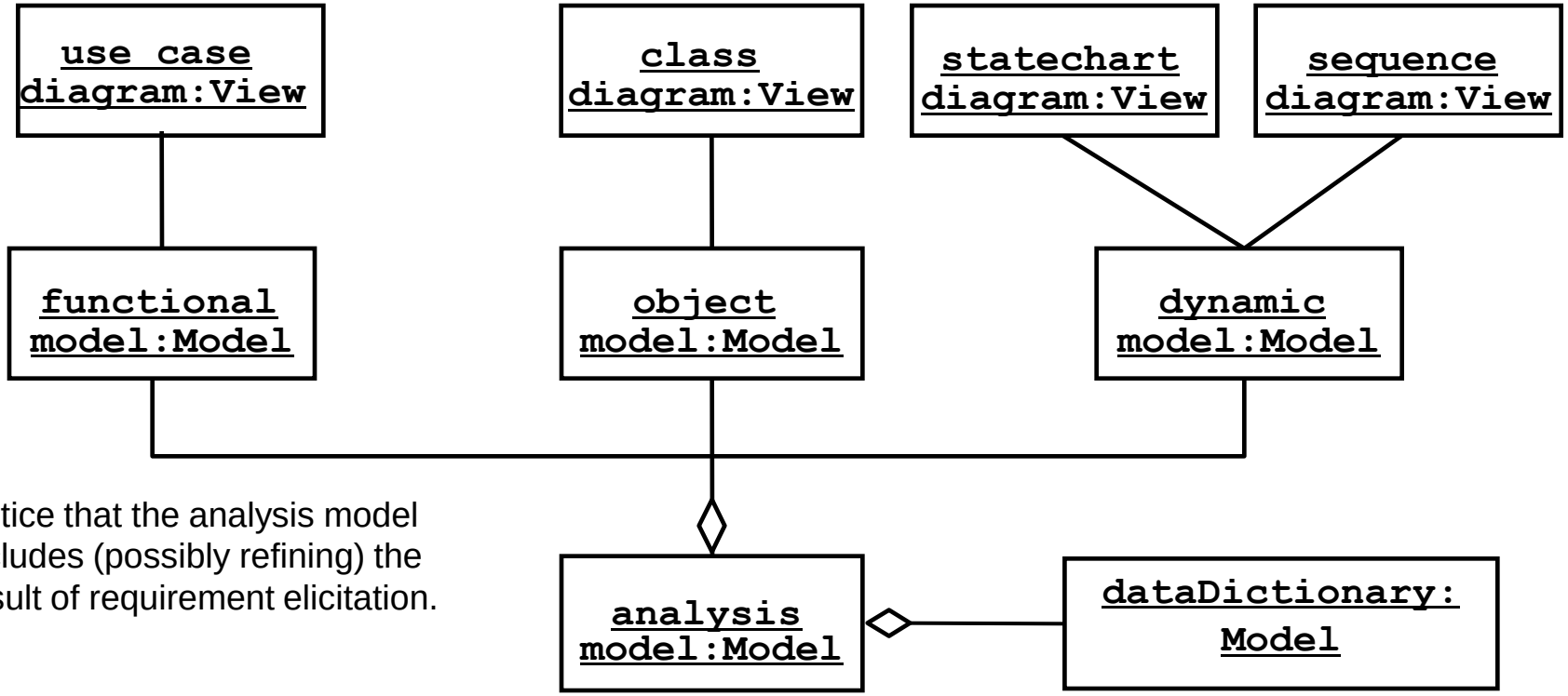
- Transform the specifications of the system into a form suitable to designers, from the requirement elicitation results (use case model)
- Systematic procedure, heuristics
- Analysis of *problem* domain, as opposed to *solution* domain (Design)
- Model composed of static and dynamic UML models
  - Static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

Overview (Bruegge and Dutoit, 2000)



Analysis Model (Bruegge and Dutoit, 2000)

Including:
-Operation's pre and potconditons
-Class invariants



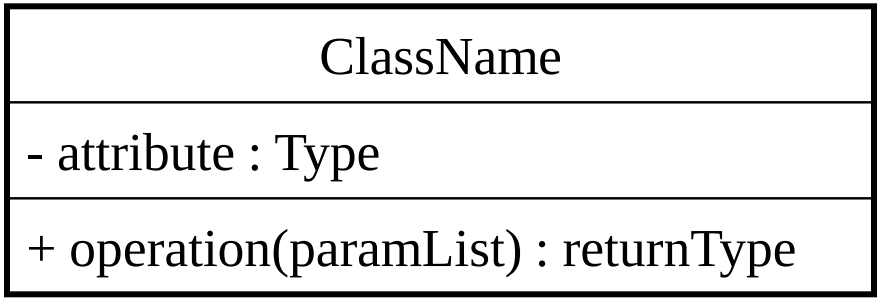
Notice that the analysis model includes (possibly refining) the result of requirement elicitation.

SYSC-3020—Introduction to Software Engineering

- Overview
- Analysis Concepts
  - Different kinds of classes
  - Different kinds of relationships
  - Modeling interaction
  - Modeling state-based behavior
- Analysis Process
  - Finding objects/classes (heuristics)
  - Finding relationships: associations and attributes (heuristics)
  - …

Class Definition (reminder)

- 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



- Reminder: attribute types can only be primitive (Boolean, Real, Integer, String)

Class/Object Taxonomy

- Objectives:
  - facilitate the identification of classes/objects
  - facilitate the identification of their role (class responsibilities)
- Result: classification of classes
  - *Entity Class/Object*
    - Represent the persistent information tracked by the system (Application domain objects, “Business objects”)
  - *Boundary Class/Object*
    - Represent the interaction between the actors and the system: user interface object, device interface object, system interface object
  - *Control Class/Object*
    - Represent the control tasks performed by the system, contains the logic and determines the order of object interactions
- There are other class taxonomies

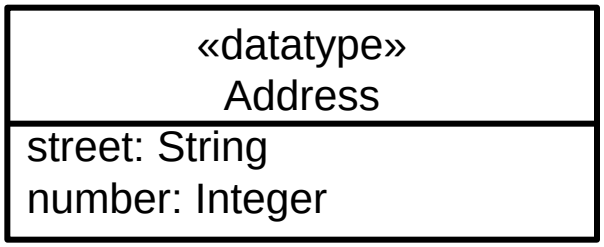
Use of Class/Object Taxonomy

- 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 responsibiltiies
  - bypass the boundary classes
- Clarify object interactions in sequence diagrams (see later)

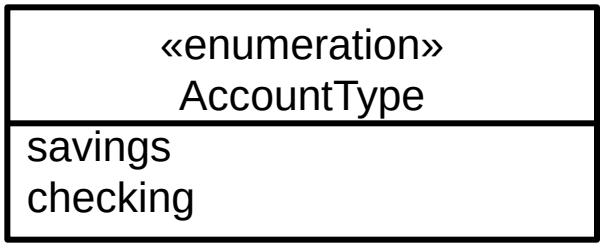
User-defined Types

- Primitive Types

- Data Types



- Enumeration



- These can be used as attribute types, in addition to the previously mentioned primitive types.

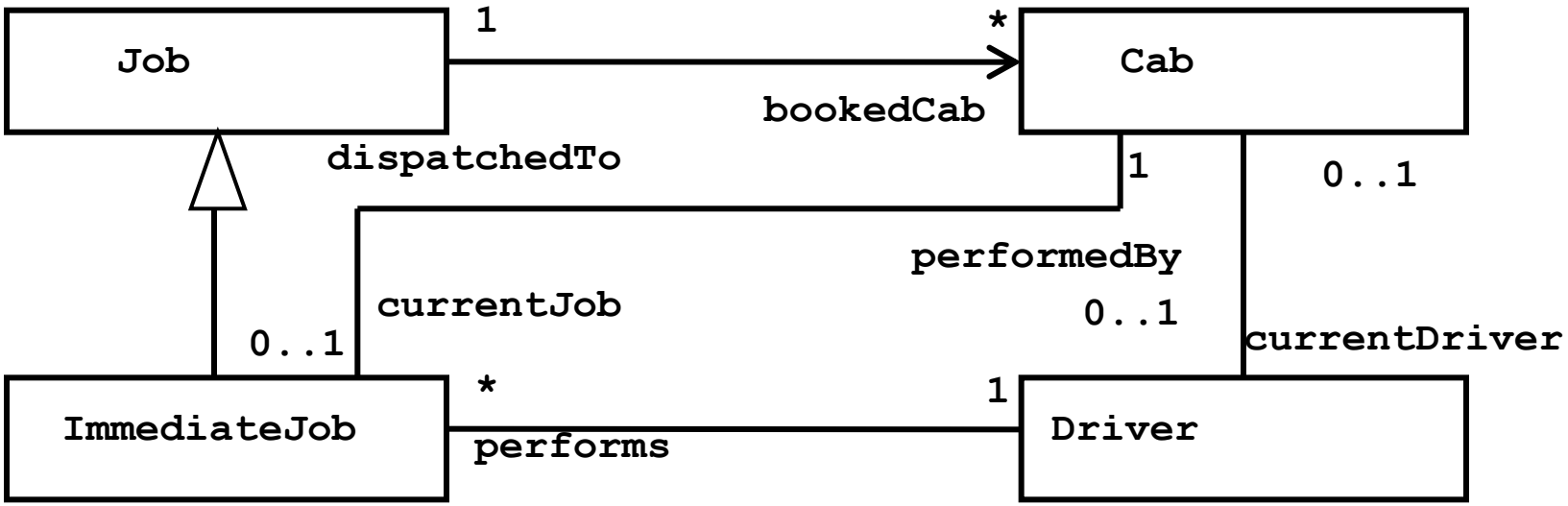
SYSC-3020—Introduction to Software Engineering

- Overview
- Analysis Concepts
  - Different kinds of classes
  - Different kinds of relationships
    - Association
    - Generalization and Realization
    - Dependency
  - Modeling interaction
  - Modeling state-based behavior
- Analysis Process
  - Finding objects/classes (heuristics)
  - Finding relationships : associations and attributes (heuristics)
  - …

Association

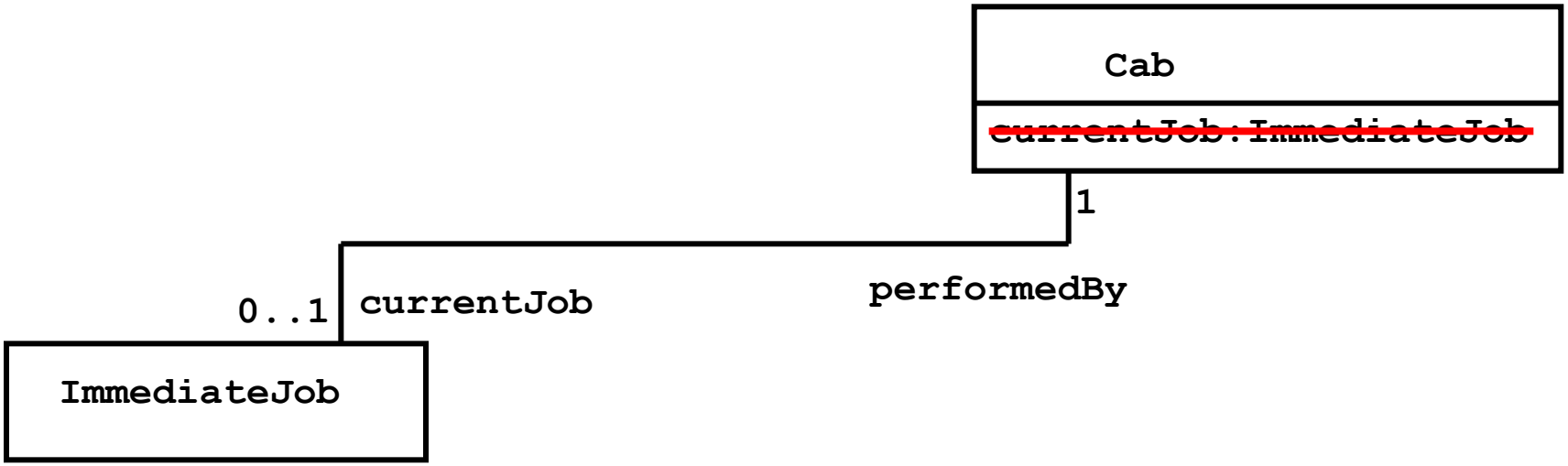
- Models the possibility of links between instances of two or more (associated) classes
  - Describes a group of links (between instances) with common structure and semantics
  - Mathematically correspond to a set of tuples (relations)
- Multiplicities indicate the size of tuples
- Can have a name (not that useful)
- Can have **rolename** on each end (very useful)

Rolename == Attribute (during implementation)



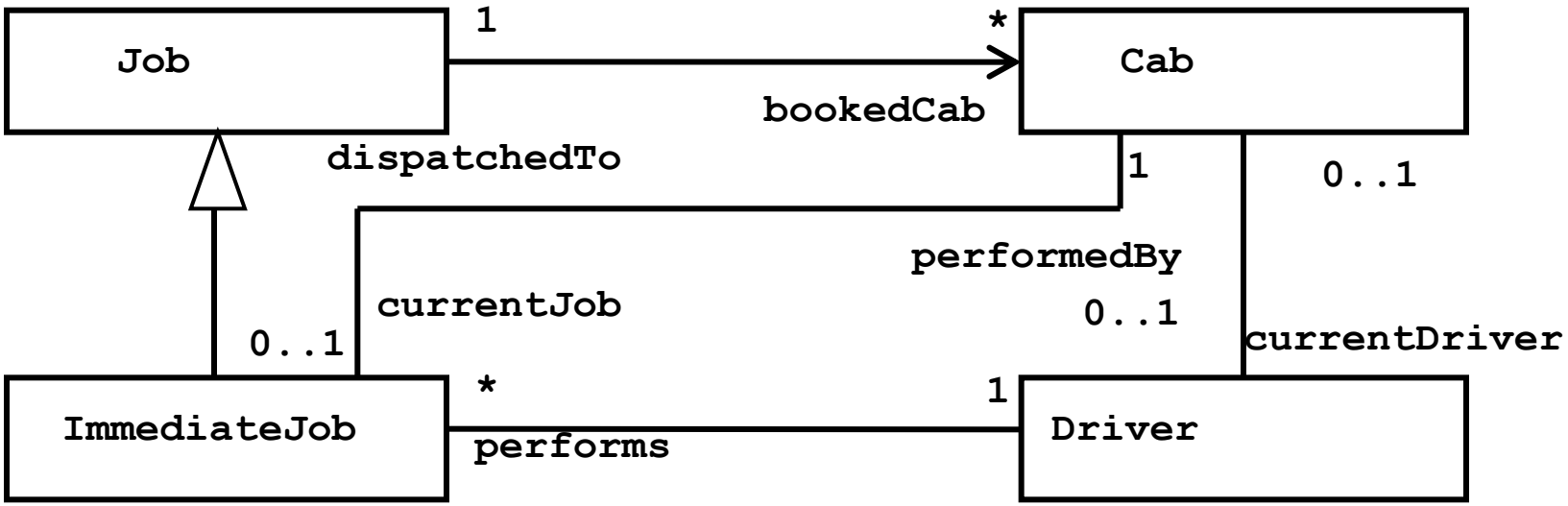
- Class **`ImmediateJob`** has (implementation) an attribute of type **`Cab`** named `performedBy`.
- Class **`Cab`** has an attribute of type **`ImmediateJob`** named `currentJob`.
- Class **`Job`** has an attribute named `bookedCab`. Its type is a collection type (a **`Set`**). Due to multiplicity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

Rolename == Attribute (during implementation)



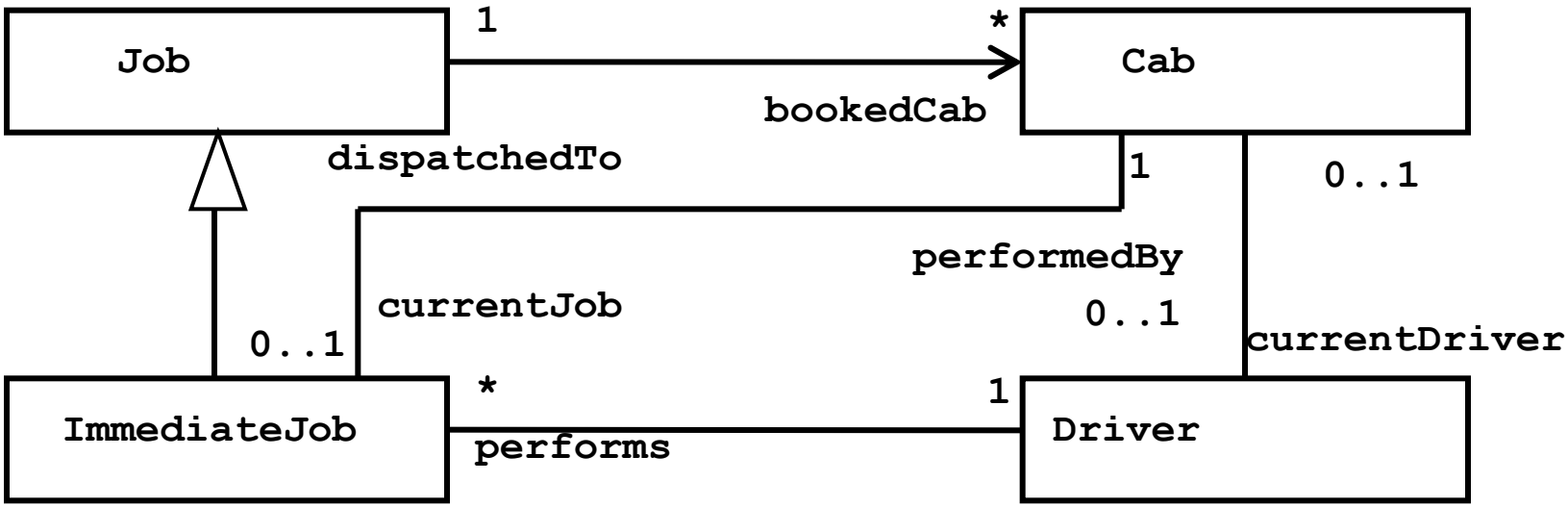
- Class Cab has
  - An attribute called currentJob (of type ImmediateJob) as part of its definition
  - An attribute called currentJob (of type ImmediateJob) because of its association with class ImmediateJob
  - There is duplication of information!
- Recall: attributes can only have primitive types.

Multiplicities have Semantics



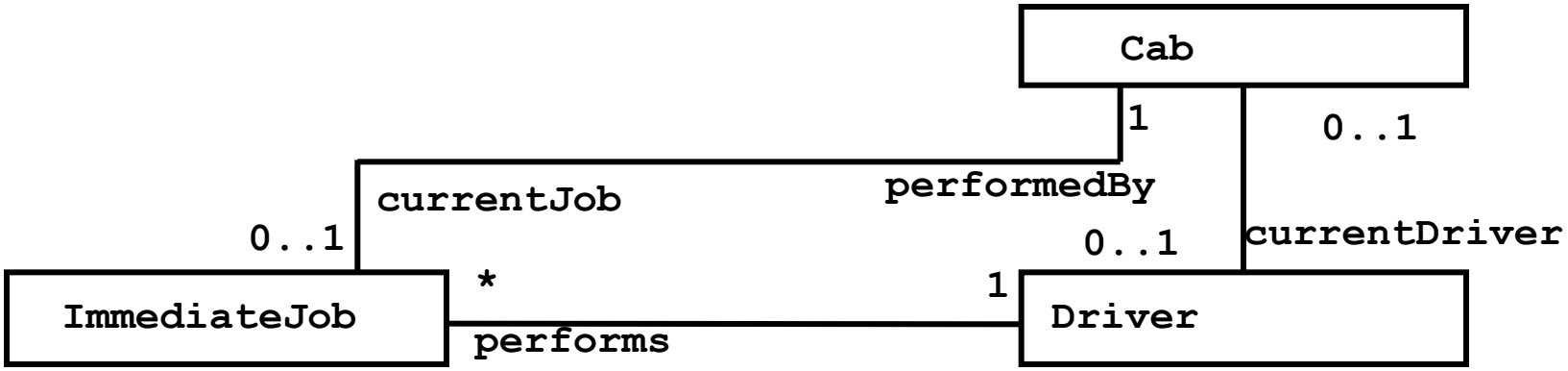
- A Job instance is linked to 0 or more Cab instances.
- A Cab instance is linked to exactly 1 Job instance.
- An ImmediateJob instance is linked to exactly one Cab instance.
- A Cab instance is linked to 0 or 1 (but not more) ImmediateJob instance.
- …
- At what time during the execution of instances/objects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

Multiplicities have Semantics (cont.)



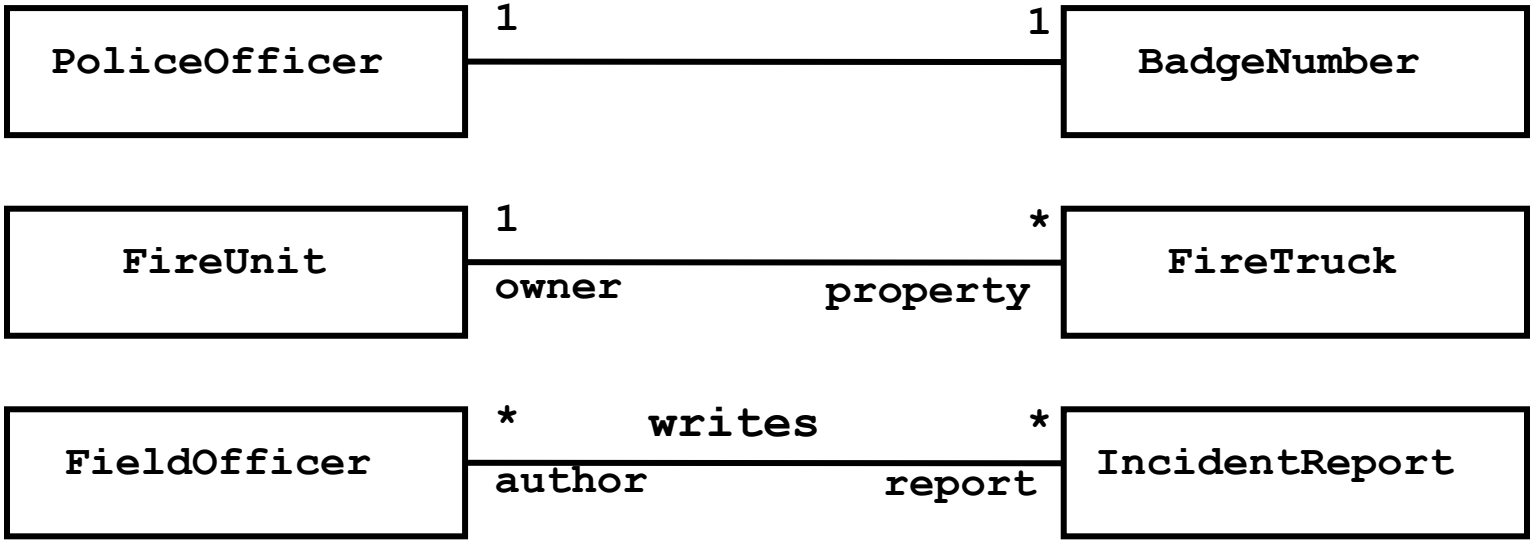
- A Cab instance is linked to exactly 1 Job instance.
- A Cab instance is linked to 0 or 1 (but not more) ImmediateJob instance.
- When?
  - At the end of the construction of a Cab instance.
  - Before and after the execution of any public operation of Cab.
- But not (necessarily)
  - During the execution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

Multiplicities have Semantics (cont.)



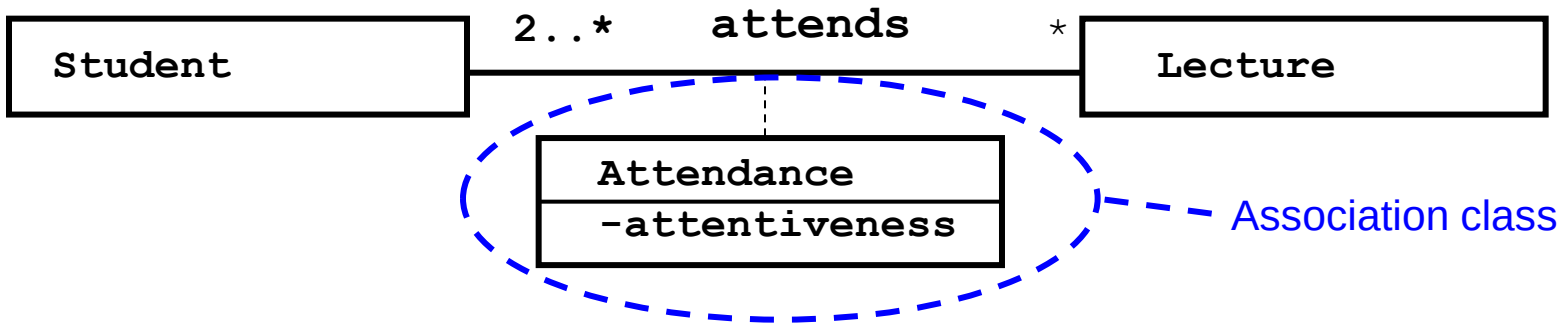
- Multiplicities have huge consequences and **can easily be wrong**.
- An ImmediateJob is linked to exactly one Cab instance
- An ImmediateJob is linked to exactly one Driver instance
  - A Driver instance is linked to 0 or 1 Cab.
  - i.e., a Driver instance may not be linked to any Cab.
- Path ImmediateJob➔Driver➔Cab 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!

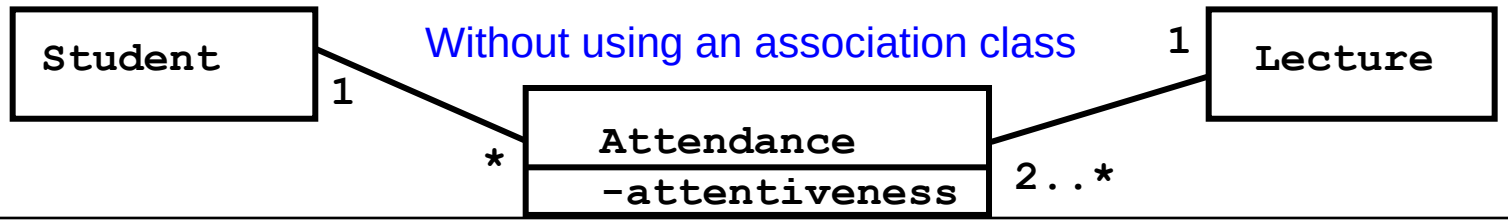
Associations—Other Examples (the FRIEND system)



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

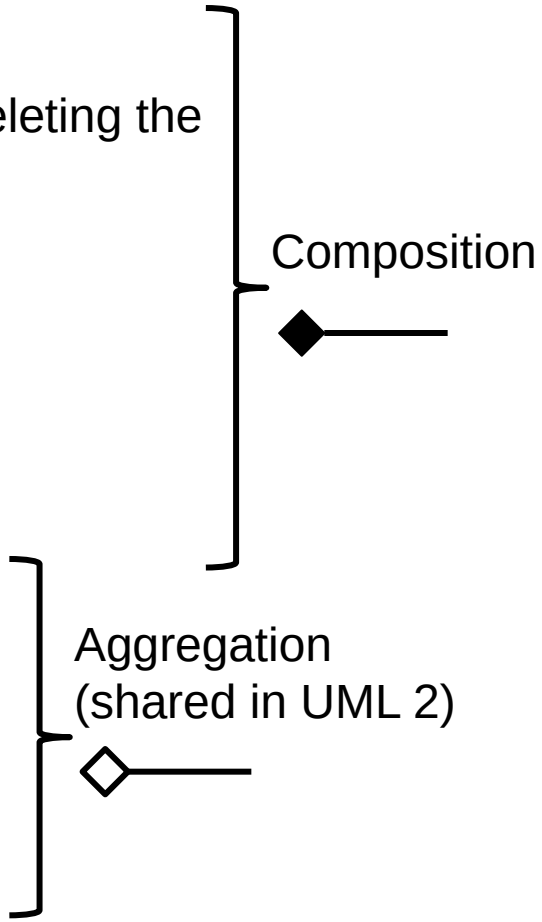




Whole-Part Class Relationship

Four semantics possible

- ExclusiveOwns (e.g. `Book has Chapter`)
  - Existence-dependency (deleting a composite ⇒ deleting the components)
  - Transitivity
  - Asymmetricity
  - 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



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 

SYSC-3020—Introduction to Software Engineering

- Overview
- Analysis Concepts
  - Different kinds of classes
  - Different kinds of relationships
    - Association
    - Generalization and Realization
    - Dependency
  - Modeling interaction
  - Modeling state-based behavior
- Analysis Process
  - Finding objects/classes (heuristics)
  - Finding relationships : associations and attributes (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 superclass variable
- 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

Generalization

- Generalization is a relationship between a more general element and a more specific element, where the more specific element is *entirely consistent* with the more general element but contains more information.
- The two elements obey the **substitutability** principle – we can use the more specific element where the more general element is expected without breaking the system.
- Generalization is a much stronger type of relationship than association.
- Generalization implies the very highest level of dependency and therefore *coupling* between two elements.
- Five types (which follow)

Interface Inheritance

- **Interface inheritance: subtyping, type inheritance**
  - This is a “harmless” form of inheritance
  - A subclass inherits attribute types and operation signatures (operation names plus formal arguments)
  - A subclass is said to support a superclass interface
  - The implementation of inherited operations is deferred until later
  - The **interfaces** are normally declared through an **abstract class**
    - We can say – with one proviso – that an interface is an abstract class
    - The proviso is that an abstract class can provide partial implementations for some operations whereas a pure interface defers the definition of all operations

Implementation Inheritance

- Generalization can be used (deliberately or not) to imply code reuse and it is then realized by an **implementation inheritance**
- A very powerful, sometimes dangerously powerful, interpretation of generalization
  - It is also the “default” interpretation of generalization
  - Combines the superclass properties in the subclasses and allows **overriding** them with new implementations, when necessary
  - Allows sharing of property descriptions, code reuse, and polymorphism

Extension Inheritance

- Inheritance as an **incremental definition** of a class
- A subclass **is-kind-of** superclass
- This is a **proper use** of implementation inheritance
- The **overriding** of properties should be used with care. It should only be allowed to make properties **more specific** (e.g. to produce additional outputs, to make implementations of operations more efficient), *not to change the meaning* of a property.

Restriction Inheritance

- Inheritance as a **restriction** mechanism whereby some inherited properties are suppressed in the subclass
- This is a **problematic use** of inheritance
- A superclass object can still be substituted by a subclass object provided that whoever is using the object is aware of the suppressed properties

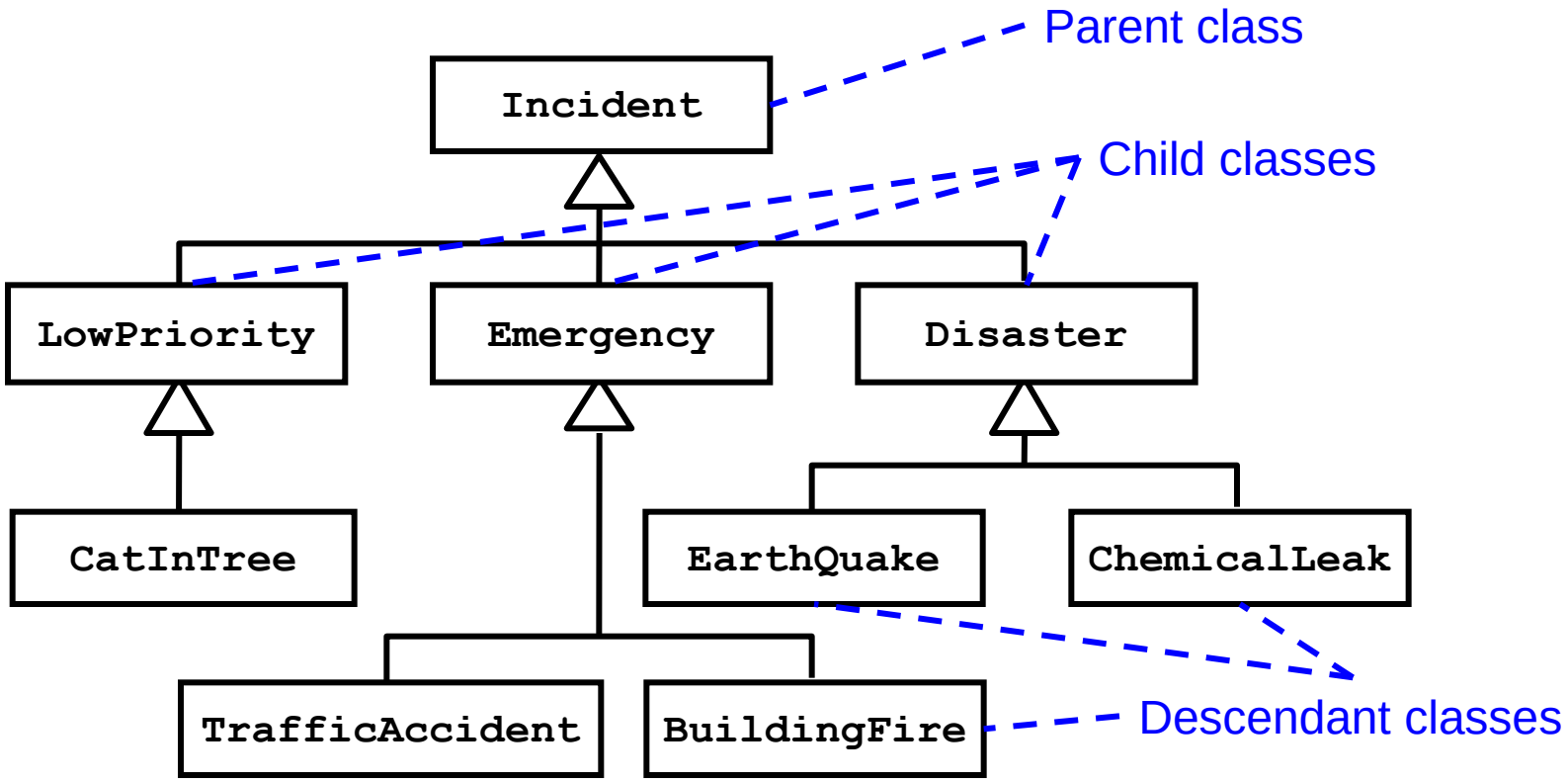
Convenience Inheritance

- When inheritance is used for the purpose of reusing code but does not implement a proper generalization.
  - Subclass violates is-a rule.
- Example: ***Set*** implemented by inheriting from ***Hashtable***
- Convenience inheritance results in operations being inherited but meaningless in the context of the subclass. This is potentially dangerous as it can create confusion when one modifies or uses the subclass
- Convenience inheritance should be avoided and substituted with *Delegation*
  - A class is said to delegate to another class if it implements an operation by merely resending a message to another class.
  - Shown as an association (has-a).

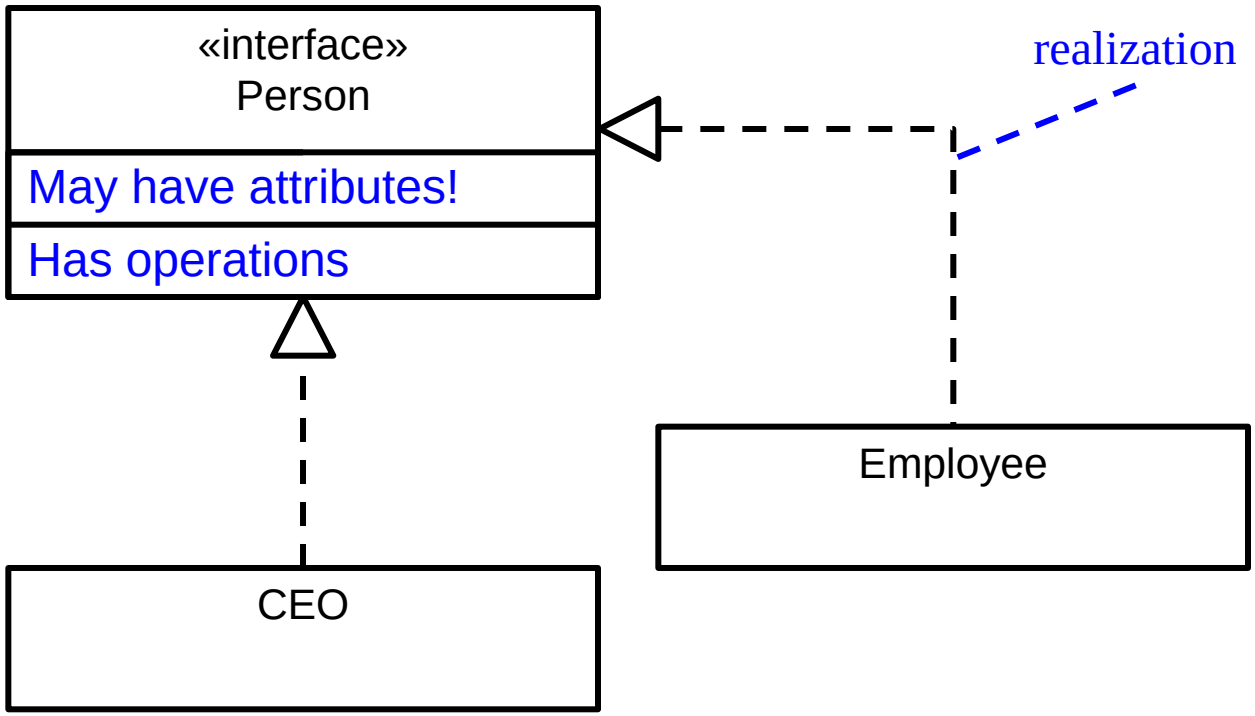
Liskov substitution principle

- *Liskov substitution principle* states that, if a client code uses the methods provided by a superclass, then developers should be able to add new subclasses without having to change the client code.
- 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: a client method written in terms of superclass *T* must be able to use instances of S without knowing whether the instances are of *S* or *T*.
- *S* is a said to be a *subtype* of *T*
- 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 (Open-Closed principle)
- Any future extension (new subclasses) **will *not* affect** parent methods (Open-Closed principle)

FRIEND Example



Interface and Realization

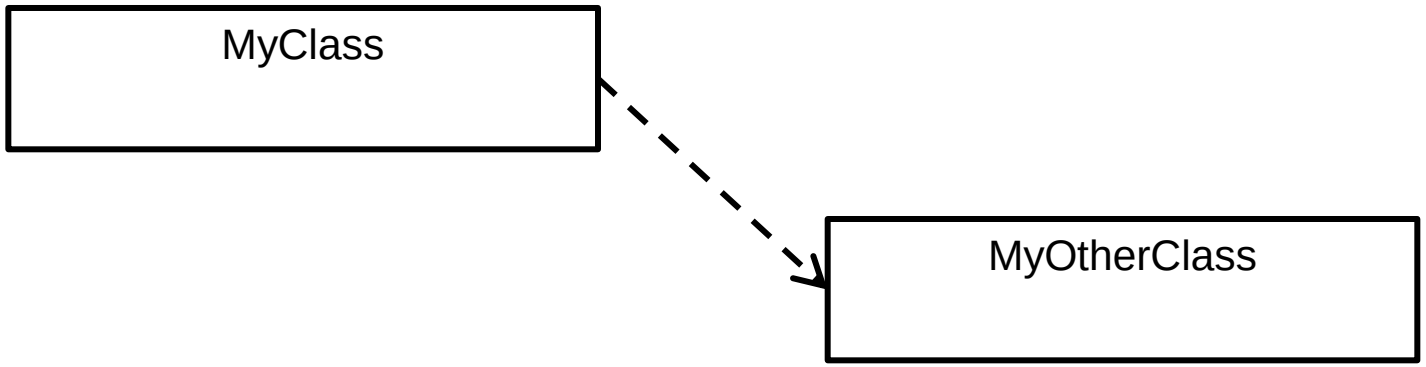


SYSC-3020—Introduction to Software Engineering

- Overview
- Analysis Concepts
  - Different kinds of classes
  - Different kinds of relationships
    - Association
    - Generalization and Realization
    - Dependency
  - Modeling interaction
  - Modeling state-based behavior
- Analysis Process
  - Finding objects/classes (heuristics)
  - Finding relationships : associations and attributes (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
- In other words, there is no need for an association but instances of the two classes need to communicate with one another anyway.
  - Eg: `myClass:operation( MyOtherClass arg )`



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.
  - Remember the analogy between association's rolenames and attributes.
- Dependency:
  - When a link between two objects does not need to survive the execution of an operation. (i.e., the second class is used as a parameter for an operation in the first class).

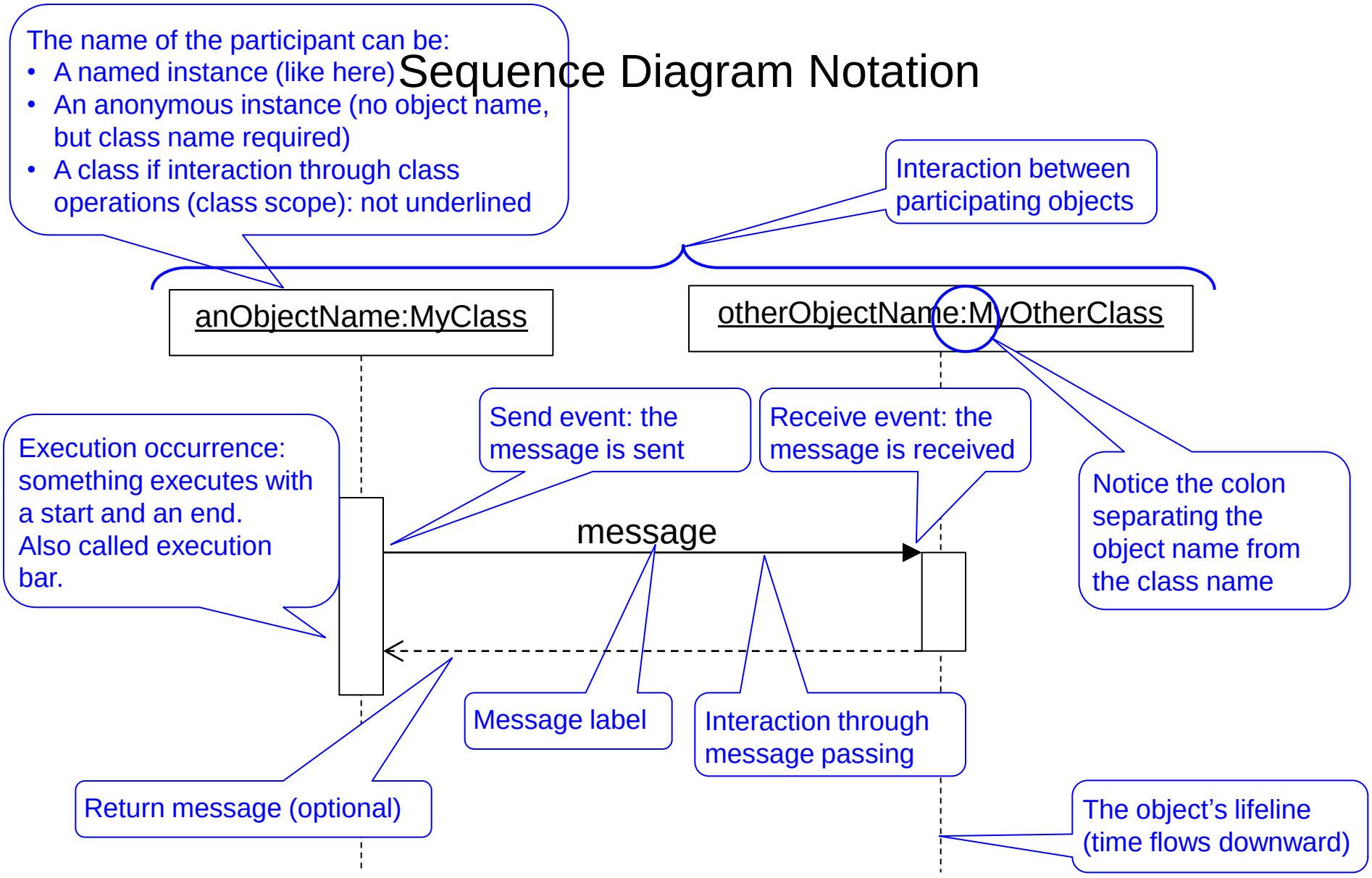
SYSC-3020—Introduction to Software Engineering

- Overview
- Analysis Concepts
  - Different kinds of classes
  - Different kinds of relationships
  - Modeling interaction
  - Modeling state-based behavior
- Analysis Process
  - Finding objects/classes (heuristics)
  - Finding relationships : associations and attributes (heuristics)
  - …

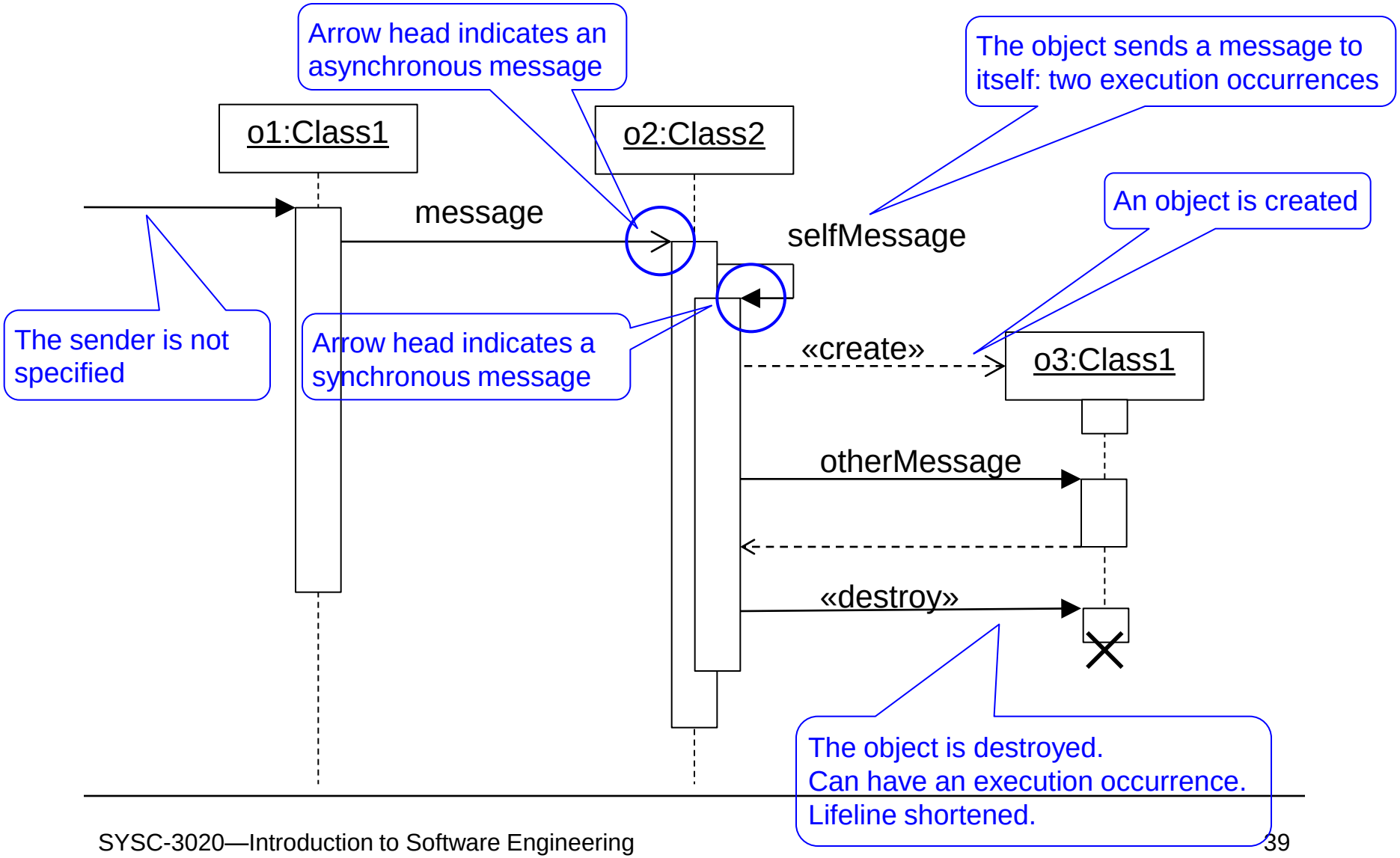
Modeling Interactions

- 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

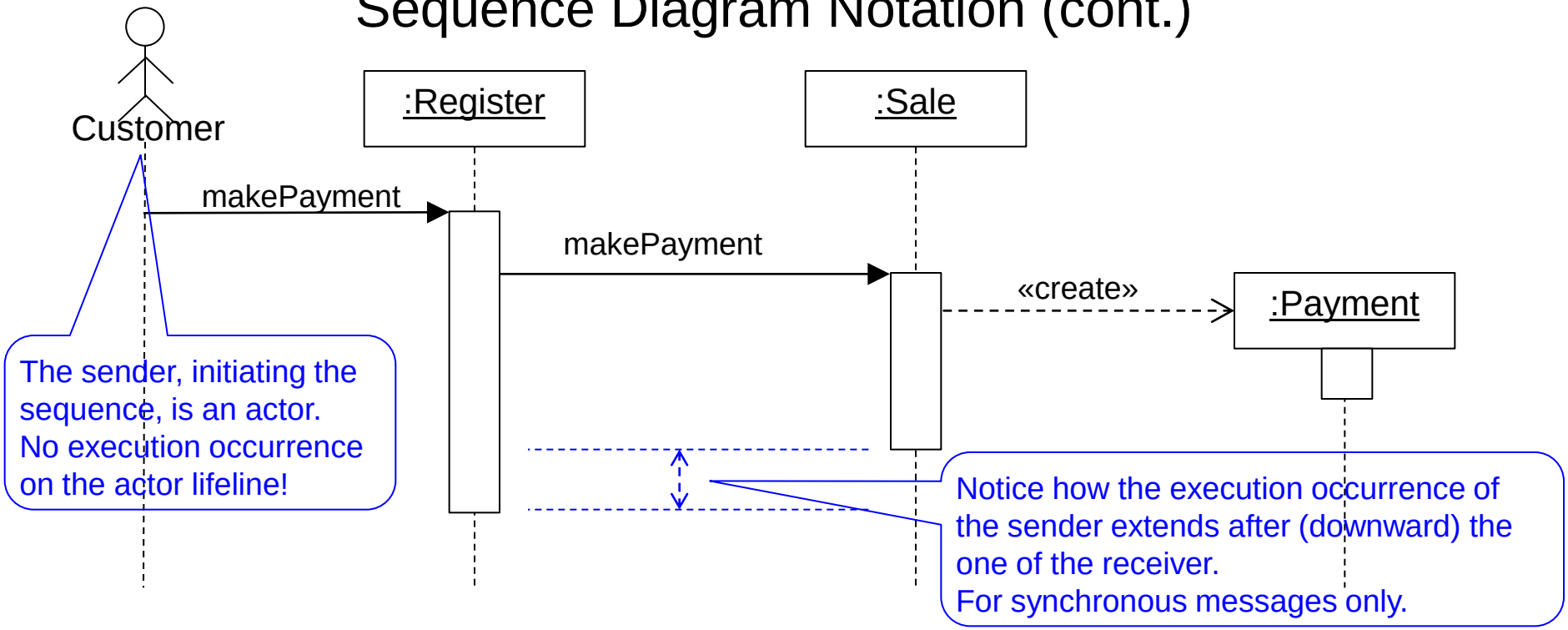
Sequence Diagram Notation



Sequence Diagram Notation (cont.)

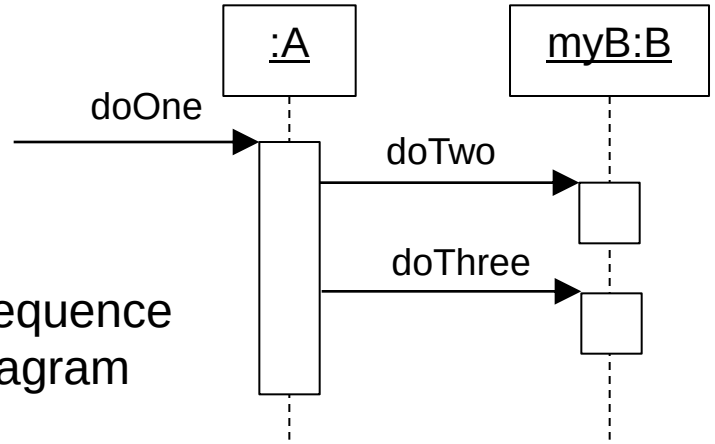


Sequence Diagram Notation (cont.)

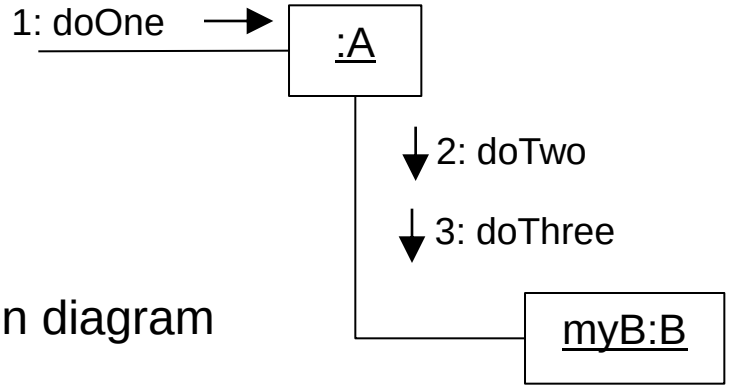


- The sequence diagram is read as follows:
  - The message *makePayment* is sent to an instance of a *Register*. The sender is an actor.
  - The *Register* instance sends the *makePayment* message to a *Sale* instance.
  - The *Sale* instance creates an instance of a *Payment*.

Example of Sequence & Communication Diagrams



Sequence diagram



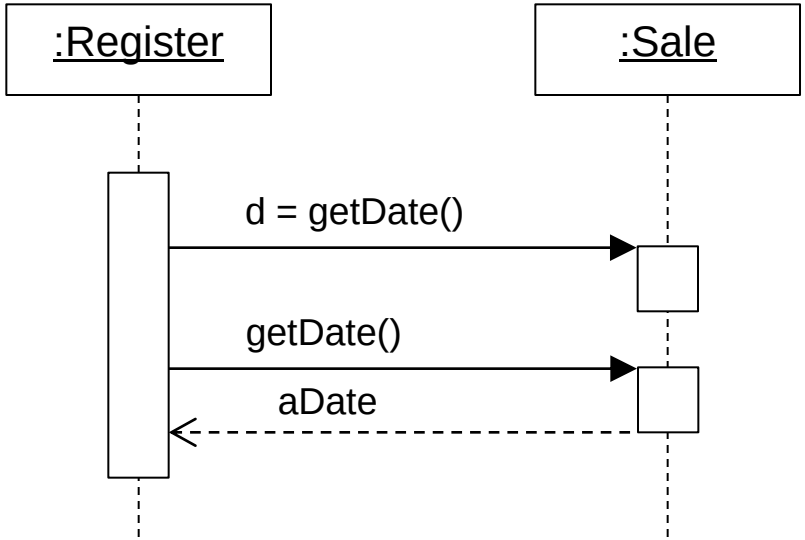
Equivalent communication diagram

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

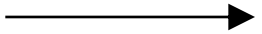
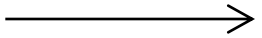
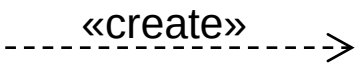
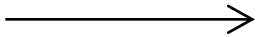
Operation to be invoked or signal to be emitted

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.

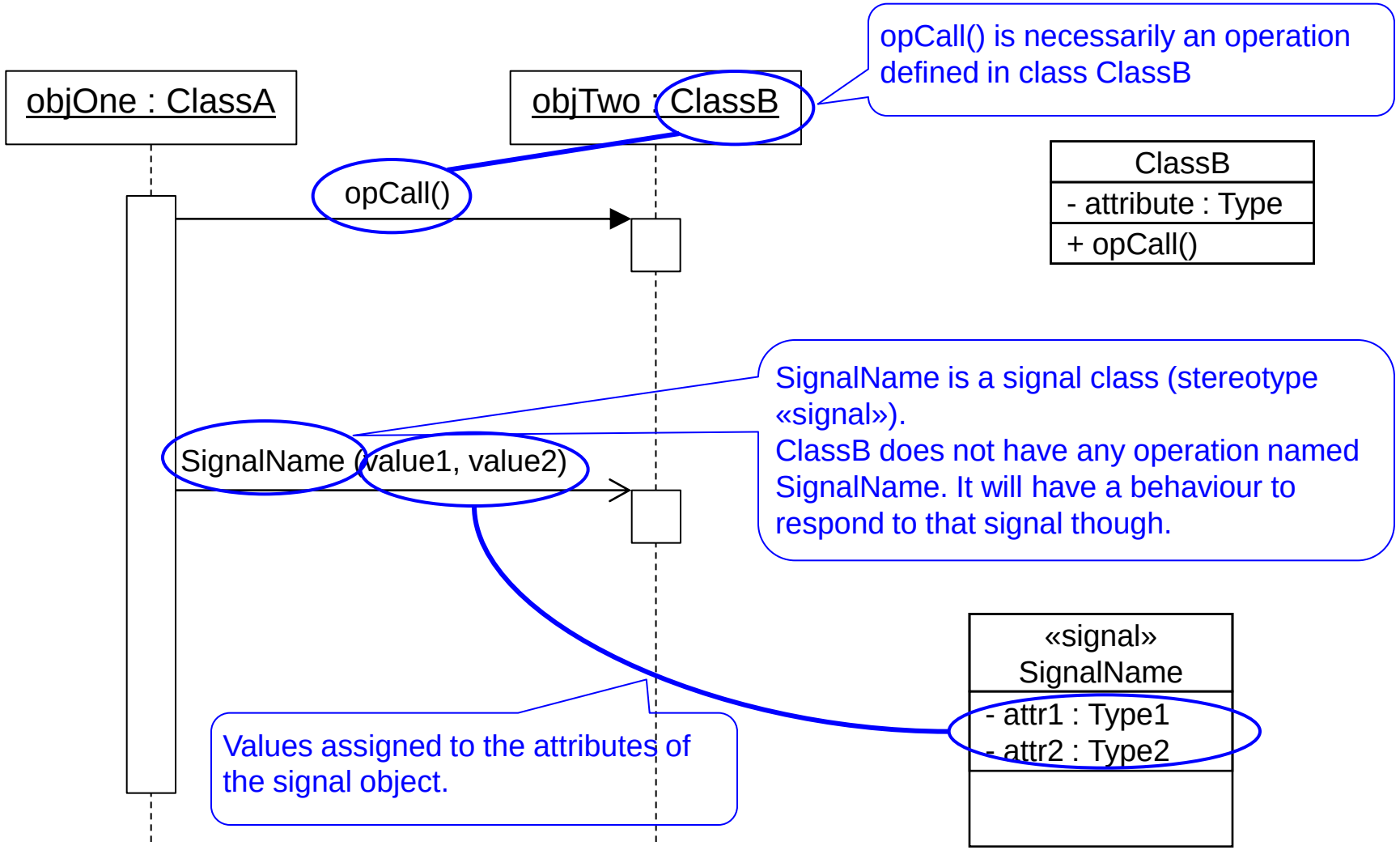
Optional
Value being returned (not the variable being assigned)

Coma-separated list
Argument can be:
- a parameter of the sending execution occurrence,
- an attribute,
- a local variable from assigned by a previous message.

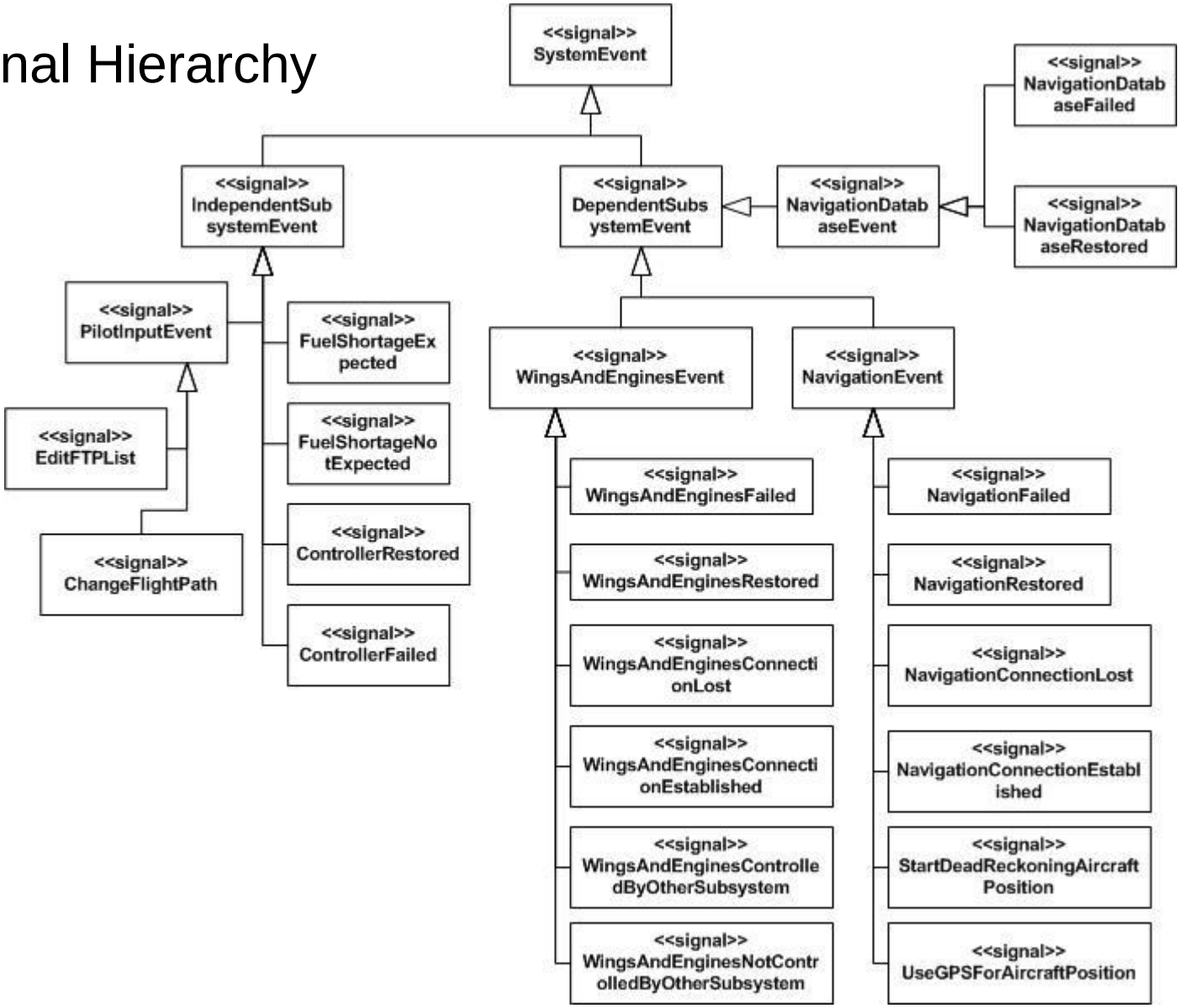
Different categories of Messages

- 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

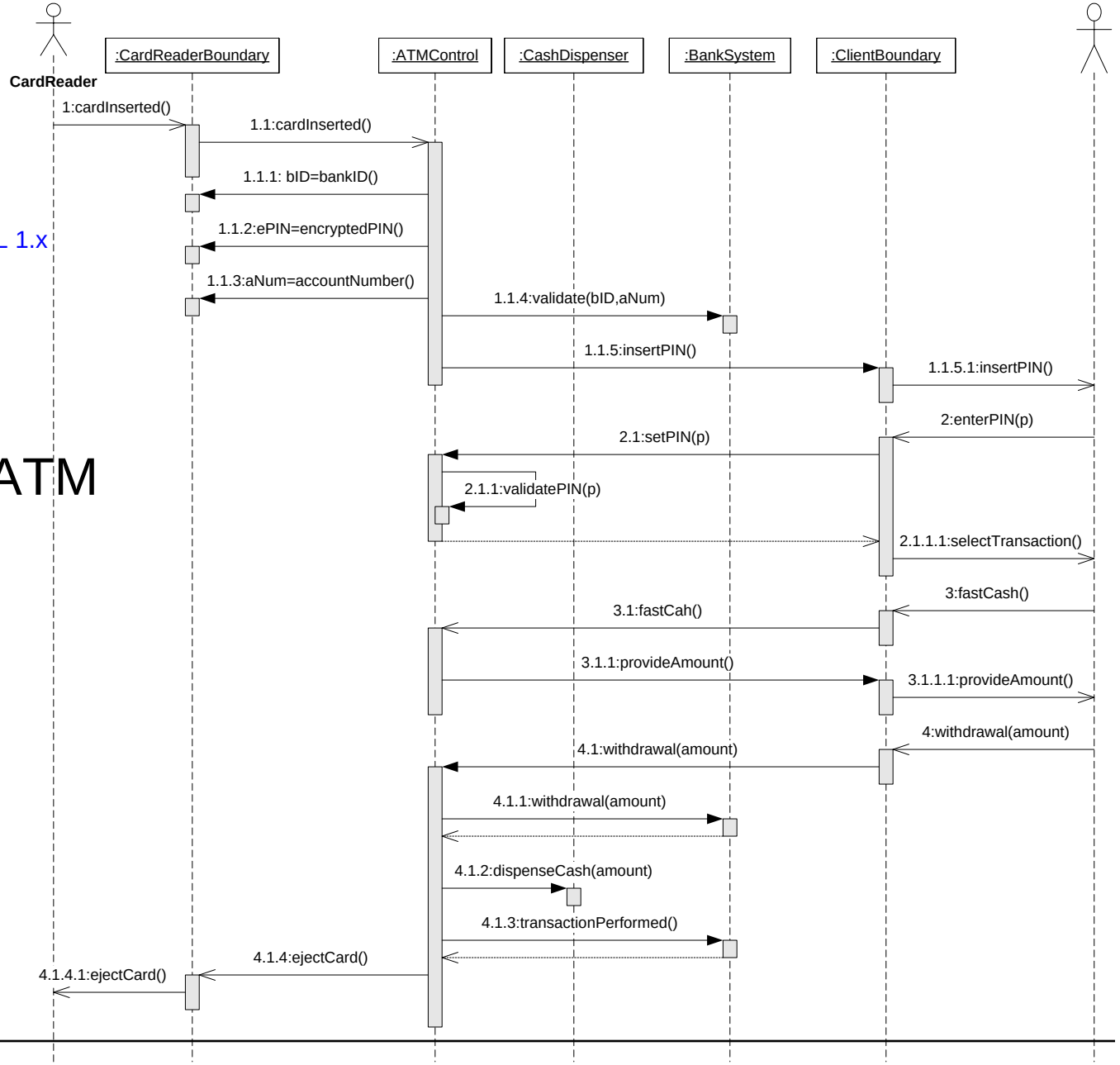


Signal Hierarchy

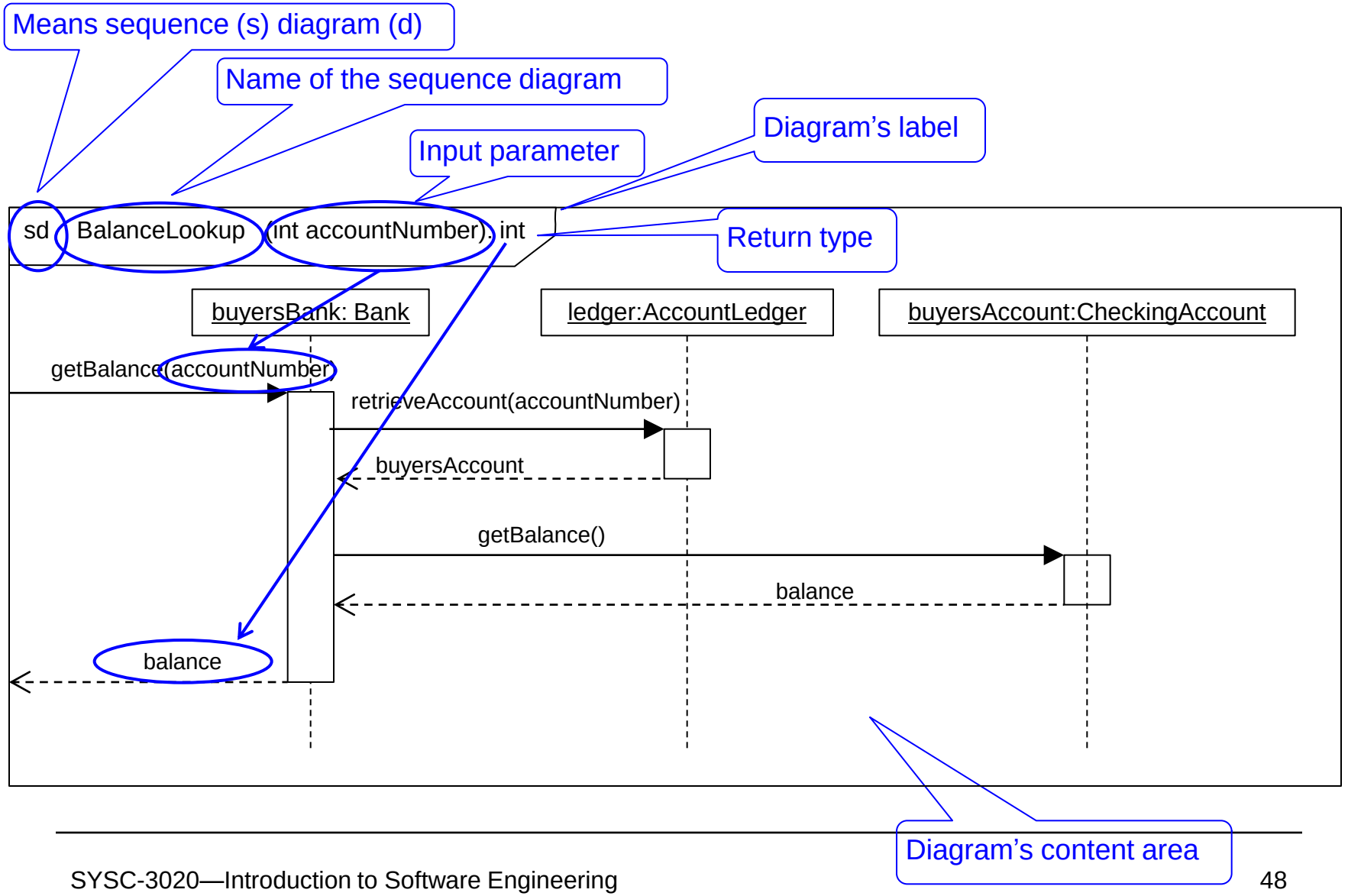


Message numbering is UML 1.x

Example - ATM



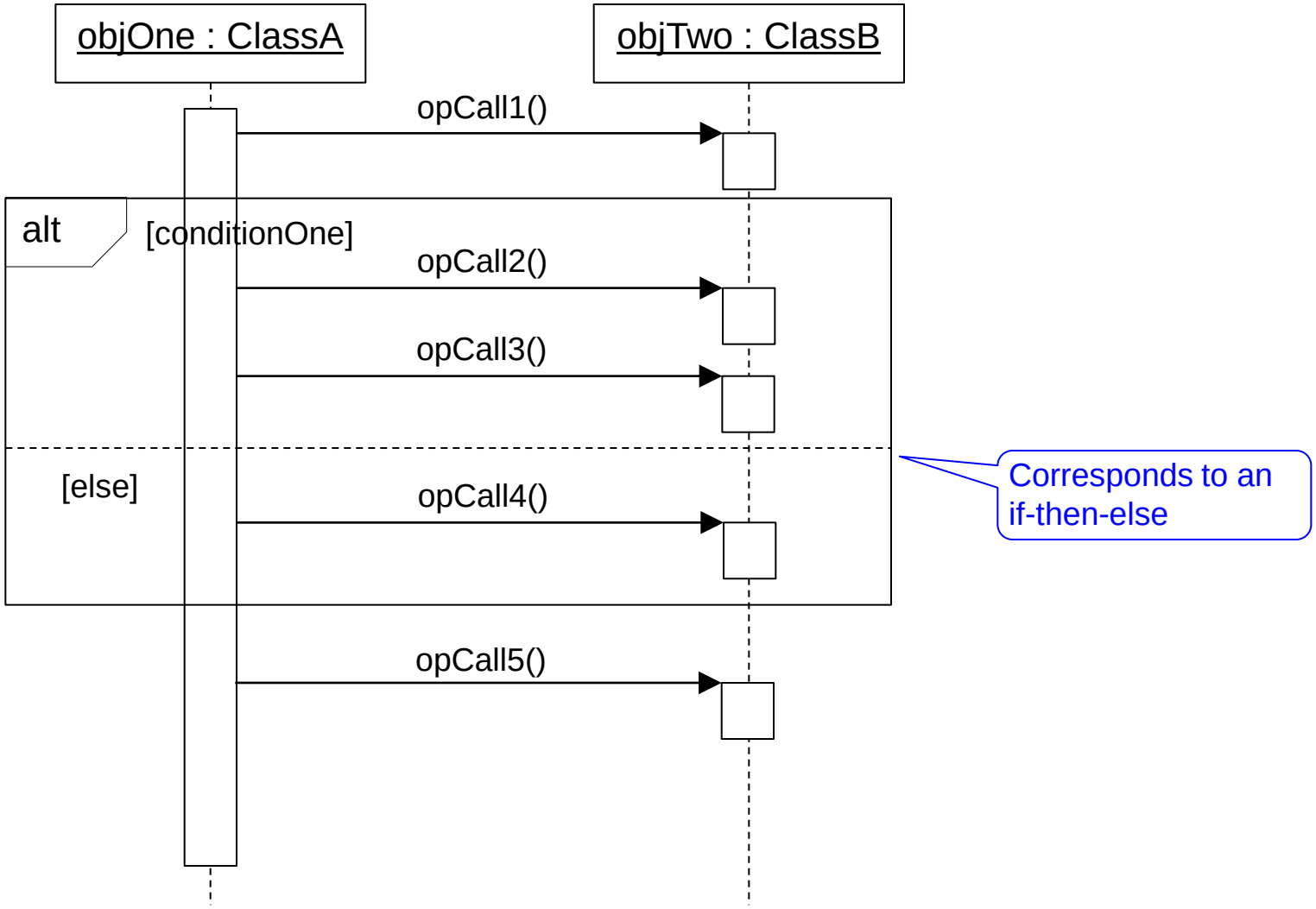
Sequence Diagram's Frame



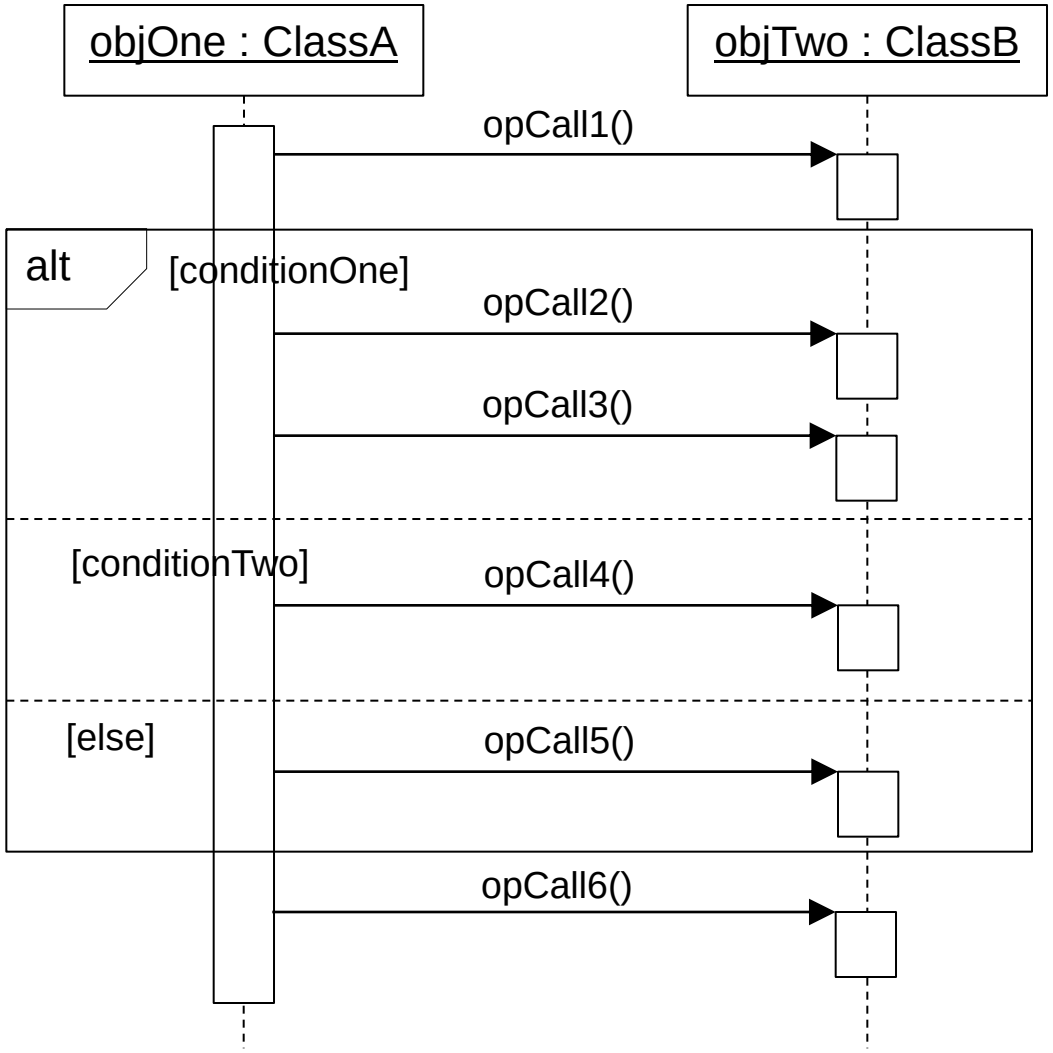
Other Frames—Control Flow

- 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

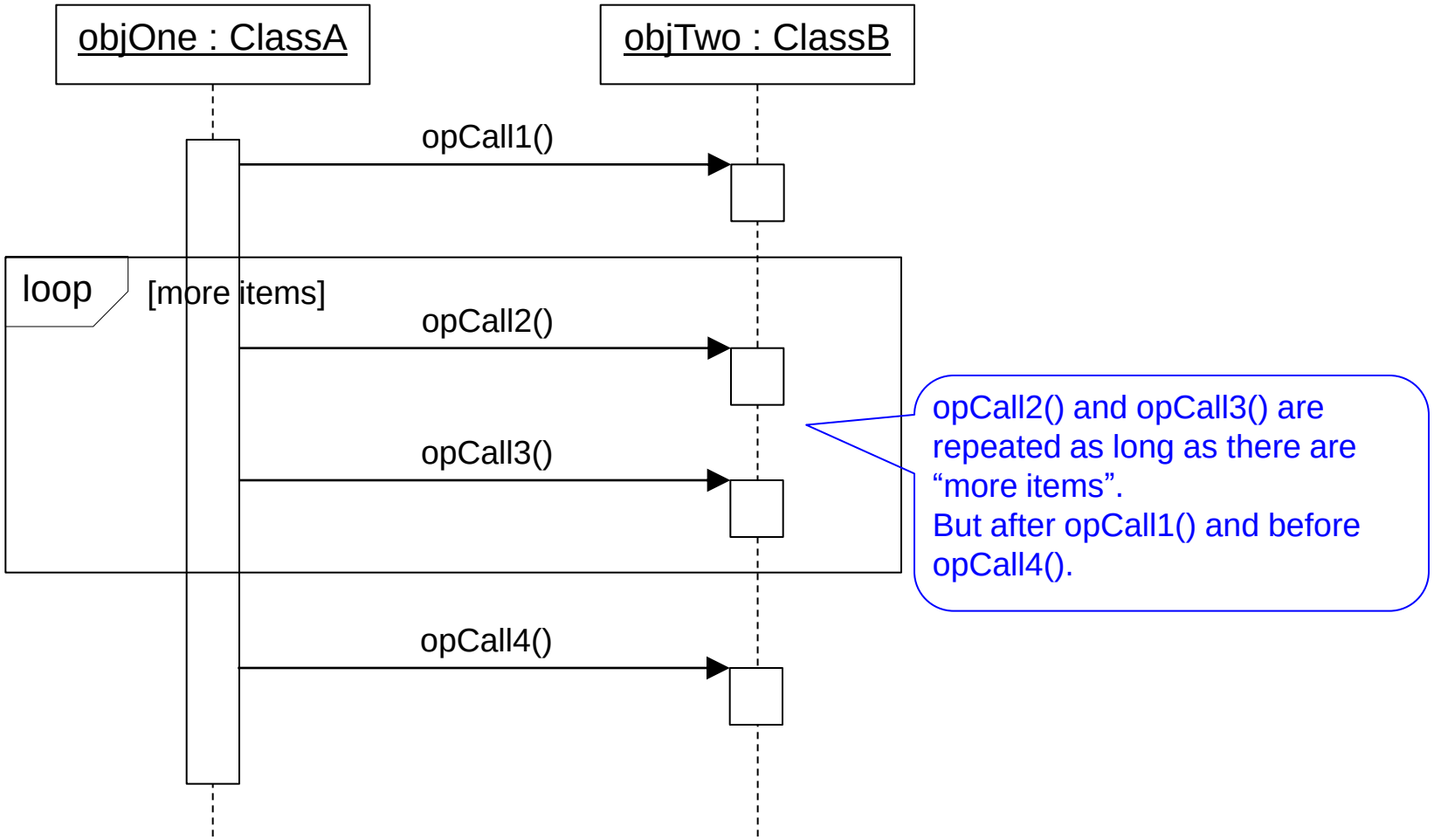
Other Frames—Control Flow (cont.)



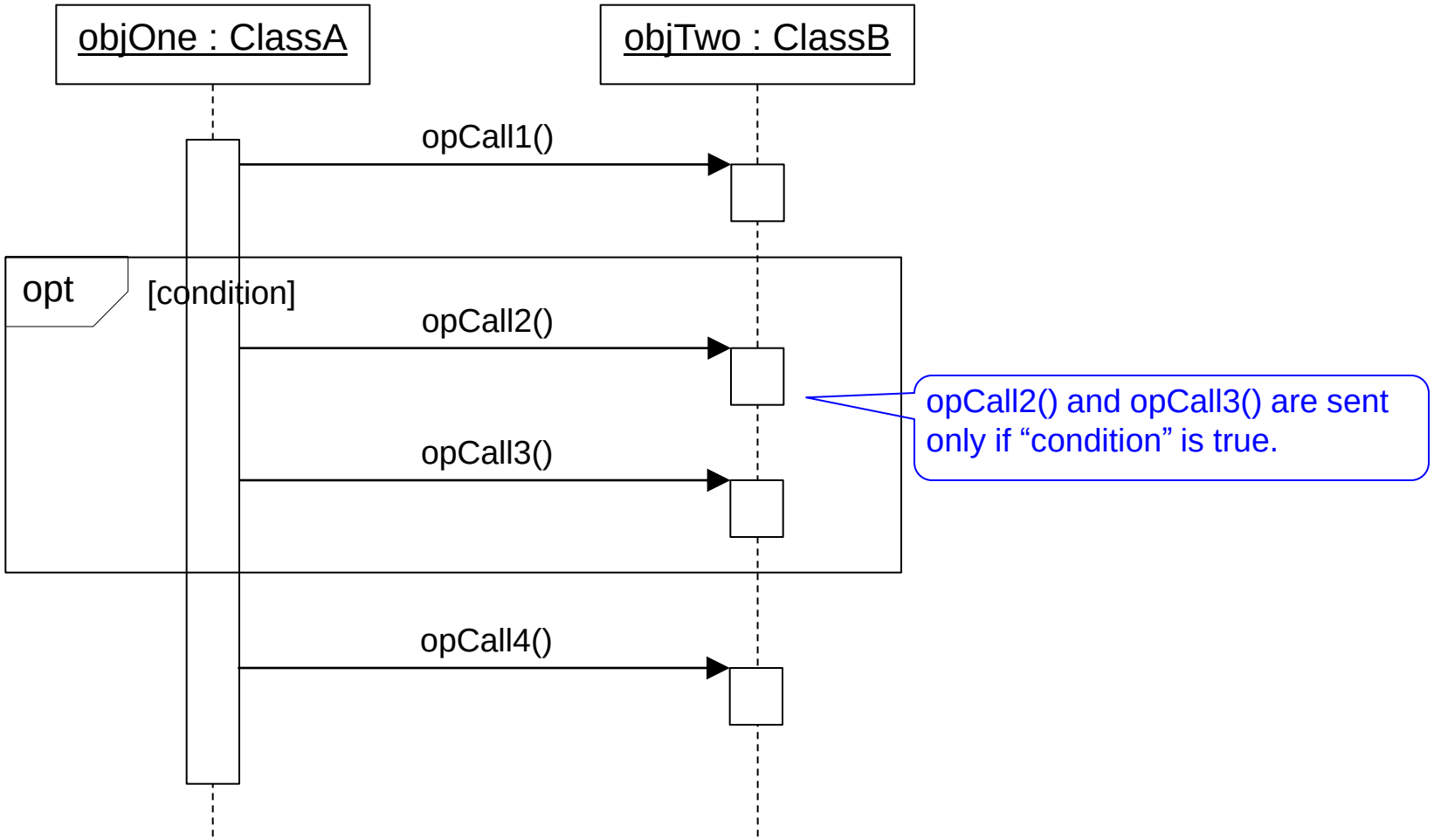
Other Frames—Control Flow (cont.)



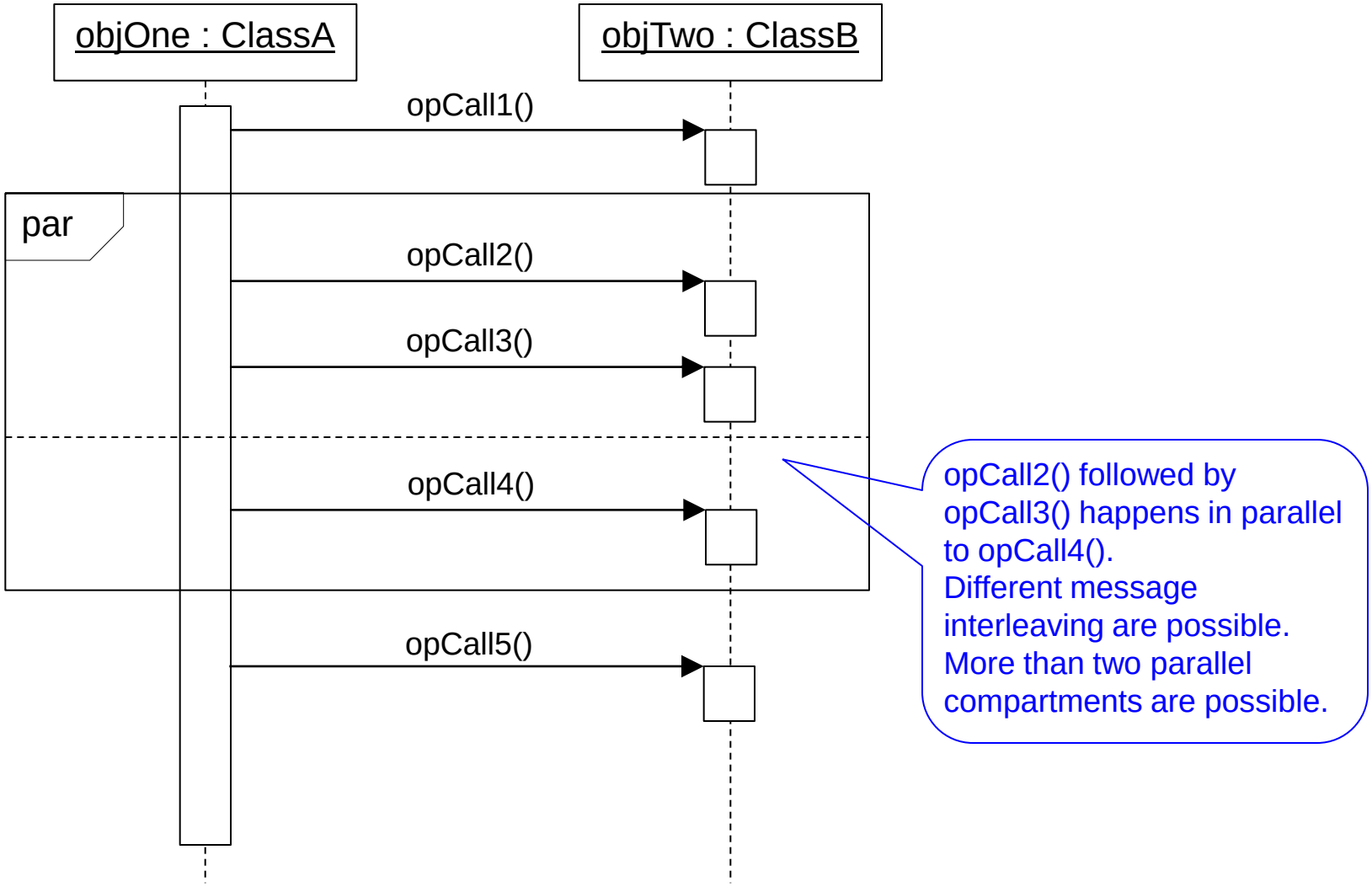
Other Frames—Control Flow (cont.)



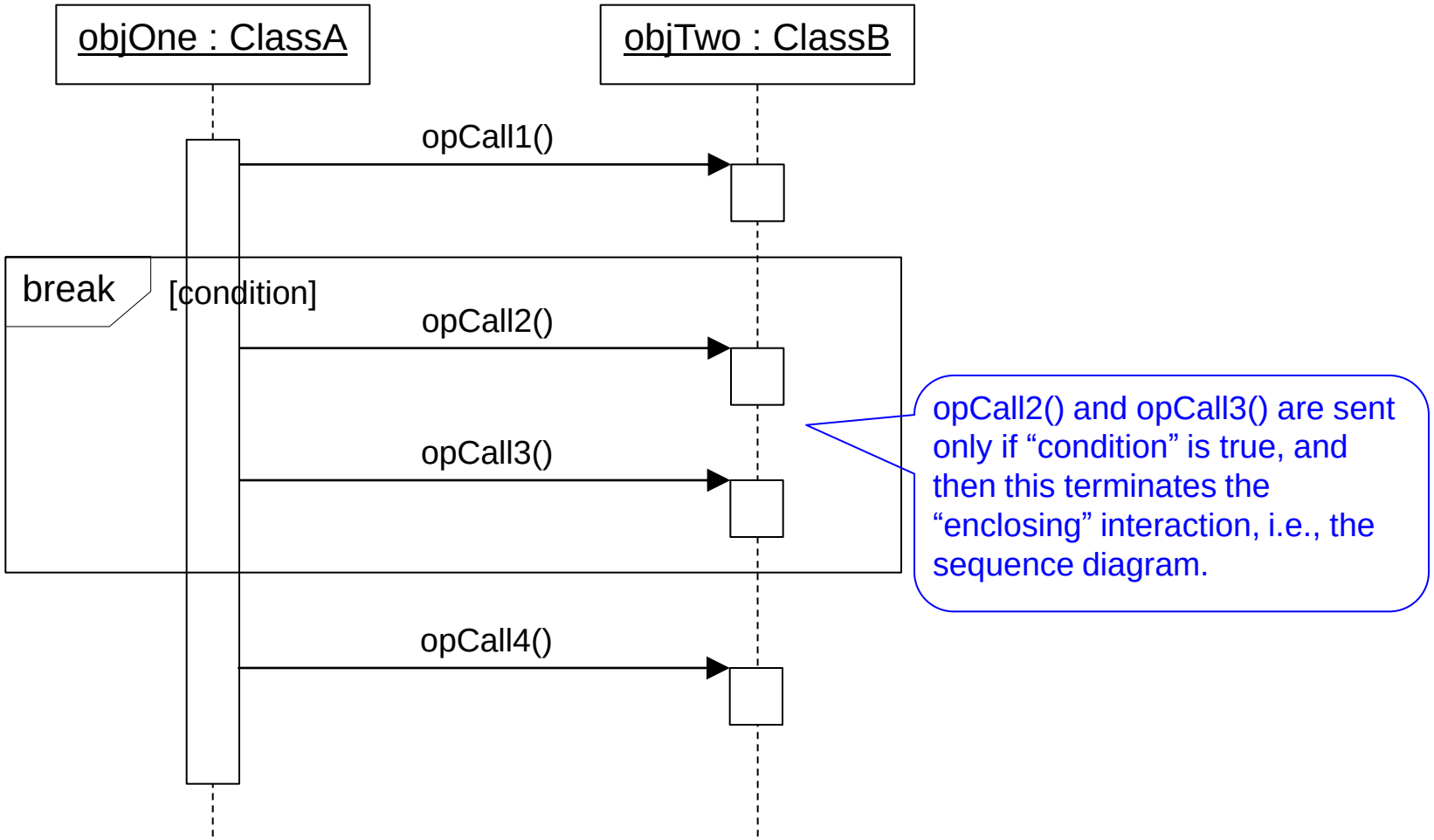
Other Frames—Control Flow (cont.)



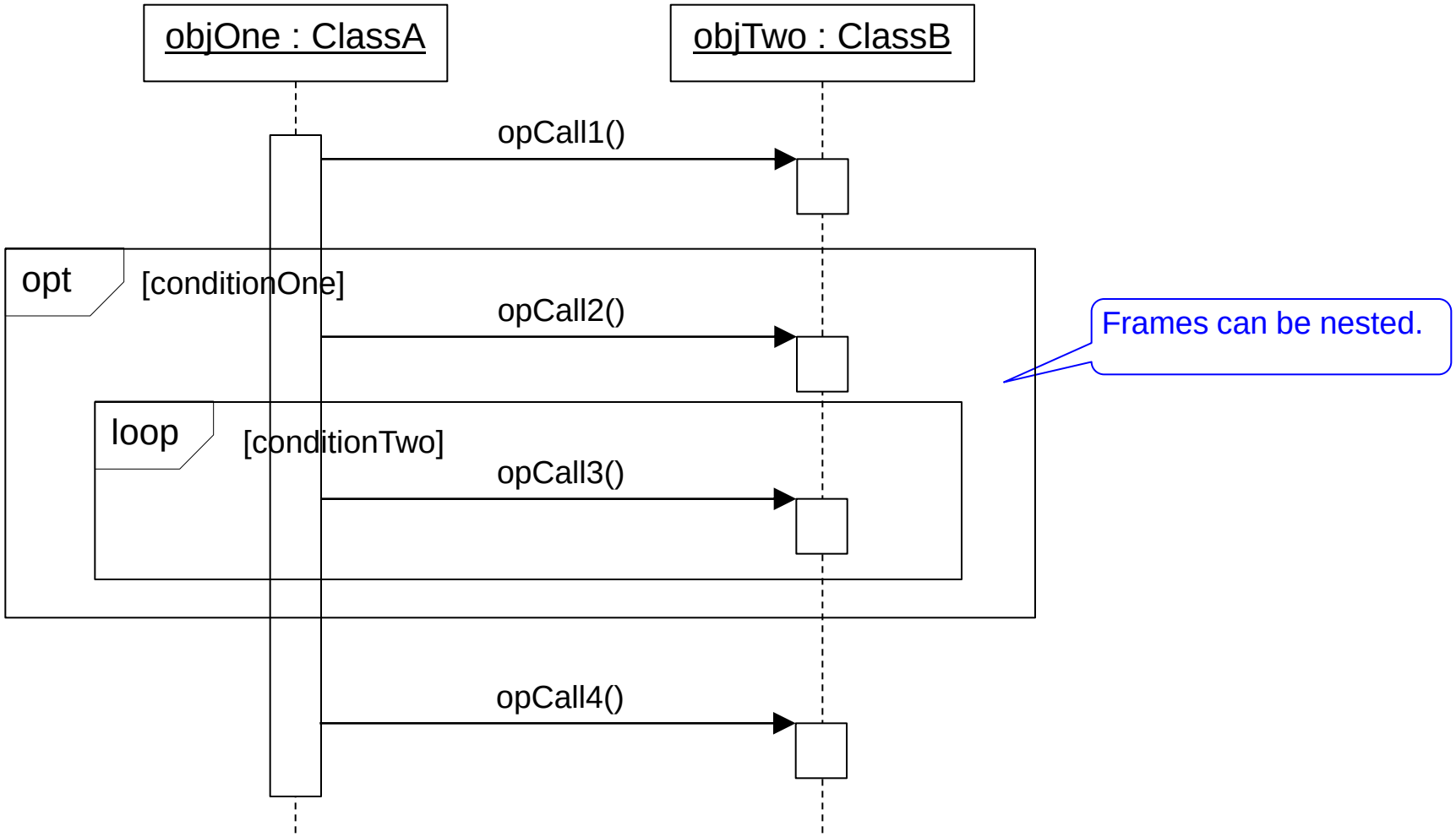
Other Frames—Control Flow (cont.)



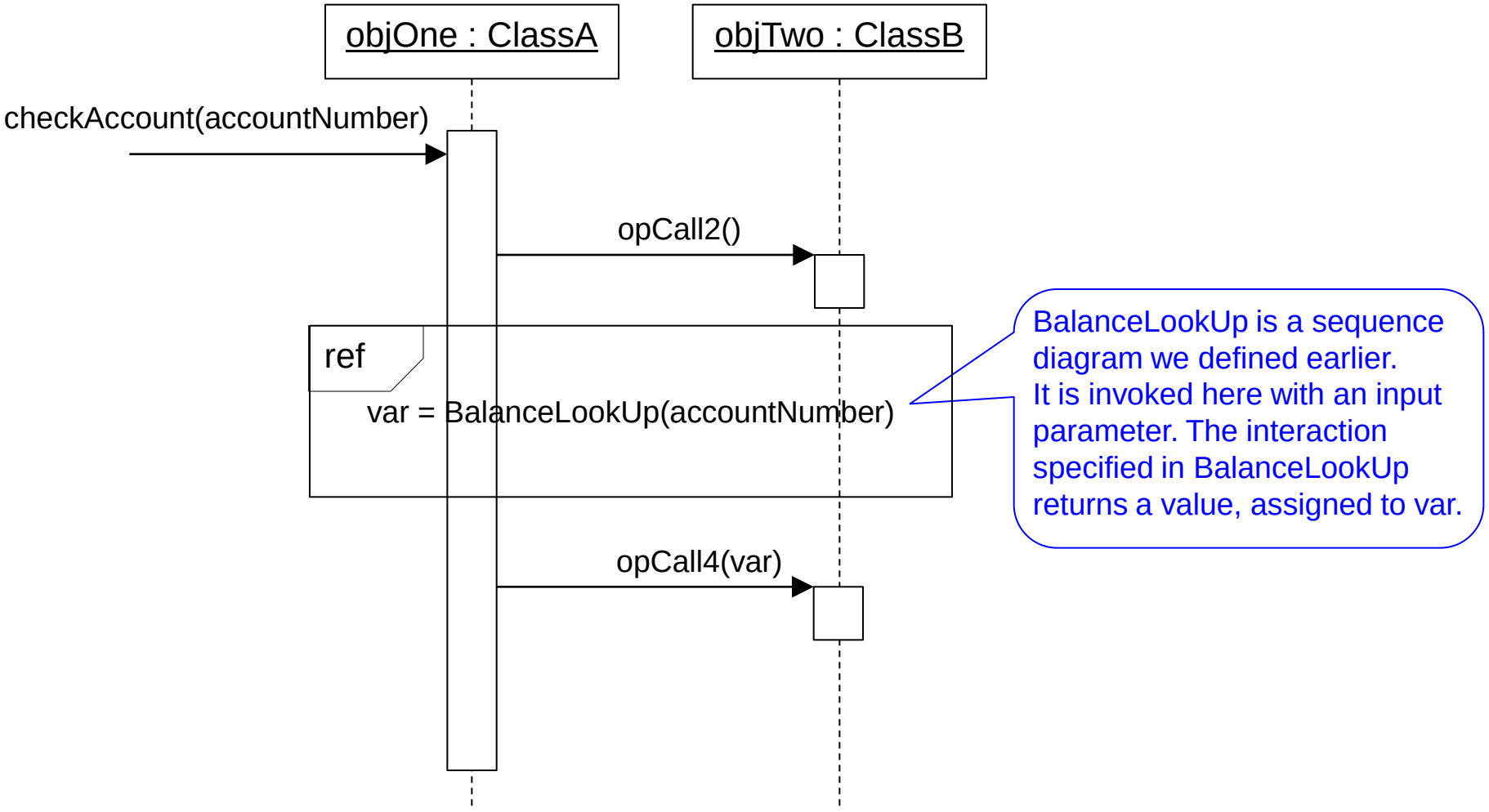
Other Frames—Control Flow (cont.)



Other Frames—Control Flow (cont.)



Other Frames—Control Flow (cont.)



SYSC-3020—Introduction to Software Engineering

- Overview
- Analysis Concepts
  - Different kinds of classes
  - Different kinds of relationships
  - Modeling interaction
  - Modeling state-based behavior
- 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: numberOfElements=0
  - The data structure is full: numberOfElements=maxCapacity
  - The data structure is partially full: numberOfElements>0 and numberOfElements<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 state**s** 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.

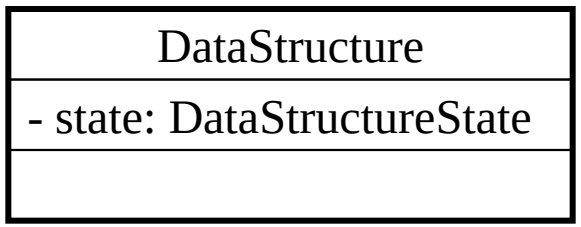
State vs. Class Invariant

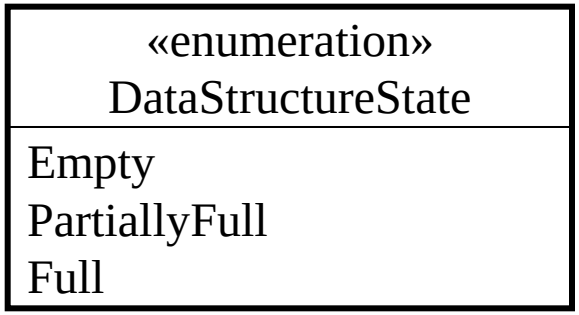
- Often useful to add a state attribute to the class

- Consider a data structure that has a maximum capacity.
- State invariant for state Empty:

  state=DataStructureState.Empty ⇔ numberOfElements=0
- Class invariant:

  state=DataStructureState.Empty
  xor state=DataStructureState.PartiallyFull
  xor state=DataStructureState.Full



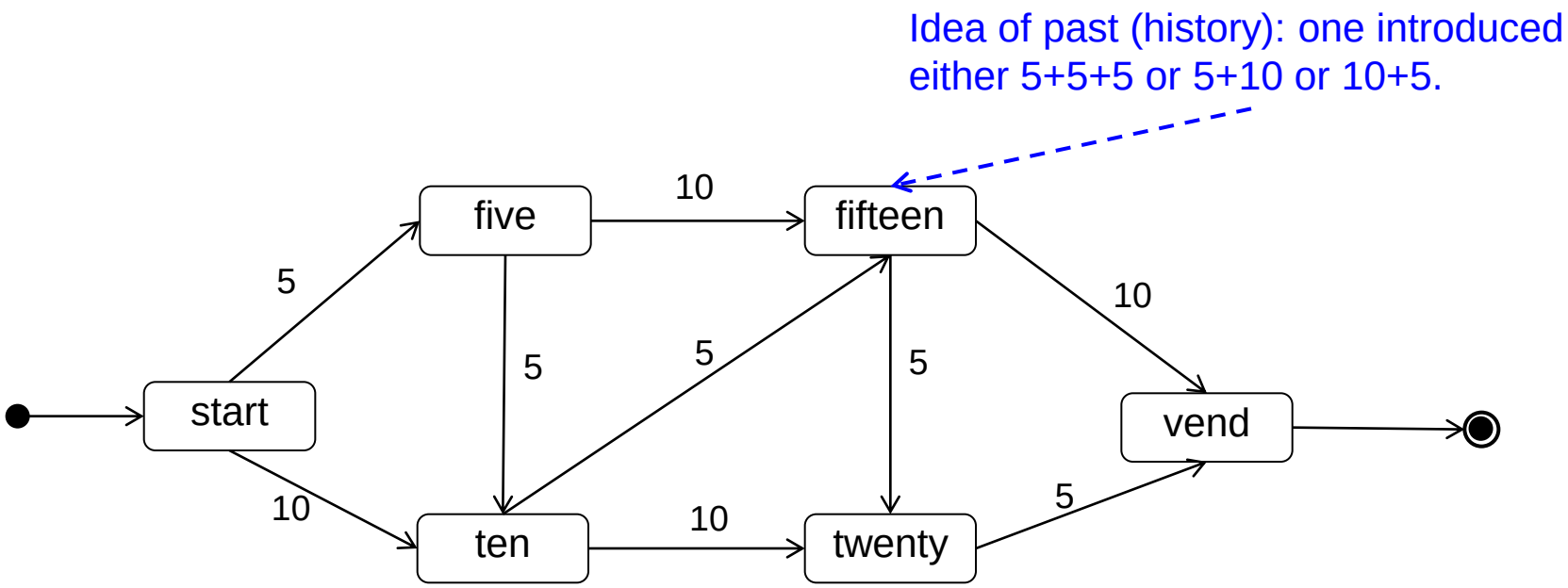


Finite State Machine

- The object (or component) being modeled can only assume a finite number of existence conditions called states
- The object behaviour is a given state is (distinguishable from other states' behaviour) essentially identical and 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 drink) is 25 cents.



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-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/behavior (heuristics)
  - Responsibilities
  - 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.

Common Noun (blue) Doing verbs (red) Being Verbs (green)
Having verbs (yellow) Modal verbs (purple) 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”
- 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 system interface with actors
- Each actor interacts with at least one boundary object
- They transform the actor information to be used by entity and control 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 (eg. 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 DispatcherSation when EmergencyReport is received.
    - Not mentioned in the use case description, but dispatcher needs an interface to view emergency an 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
- 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 *FieldOfficer* and Dispatcher stations

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?)

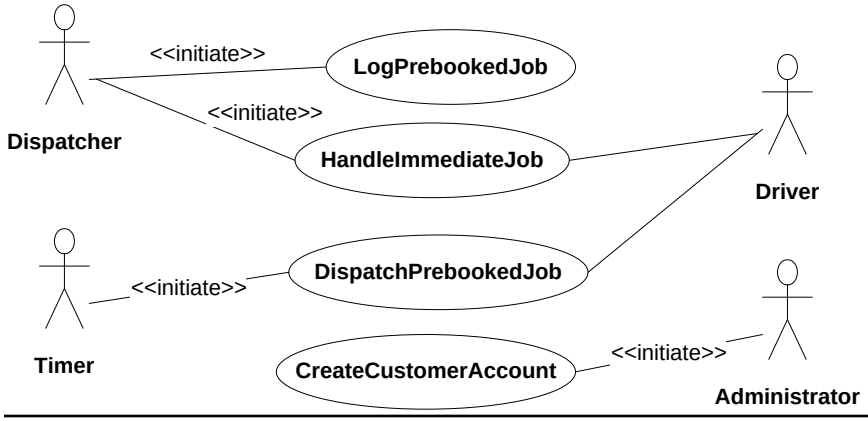
➢ You can use table(s) to report on such information.

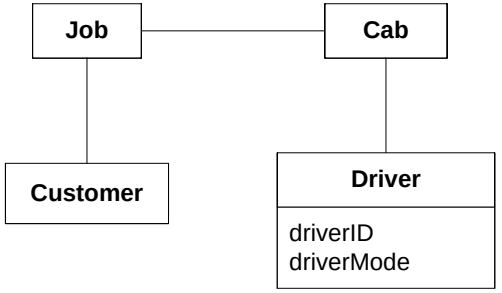
Finding Classes: Advice from Authors

- 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.

Common Difficulty: 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





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/behavior (heuristics)
  - Responsibilities
  - 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.

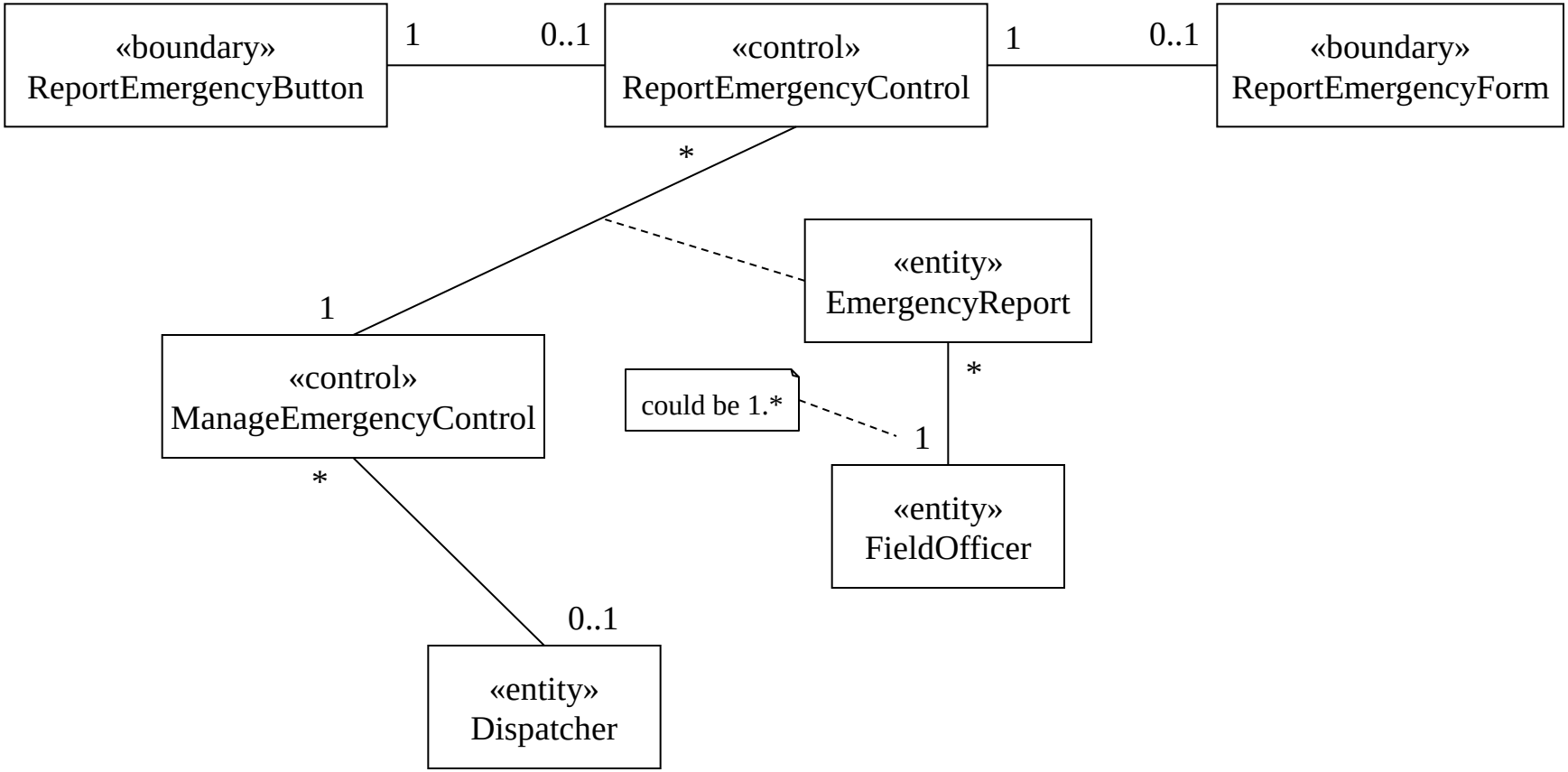
Association Heuristics: Initial Identification

- 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

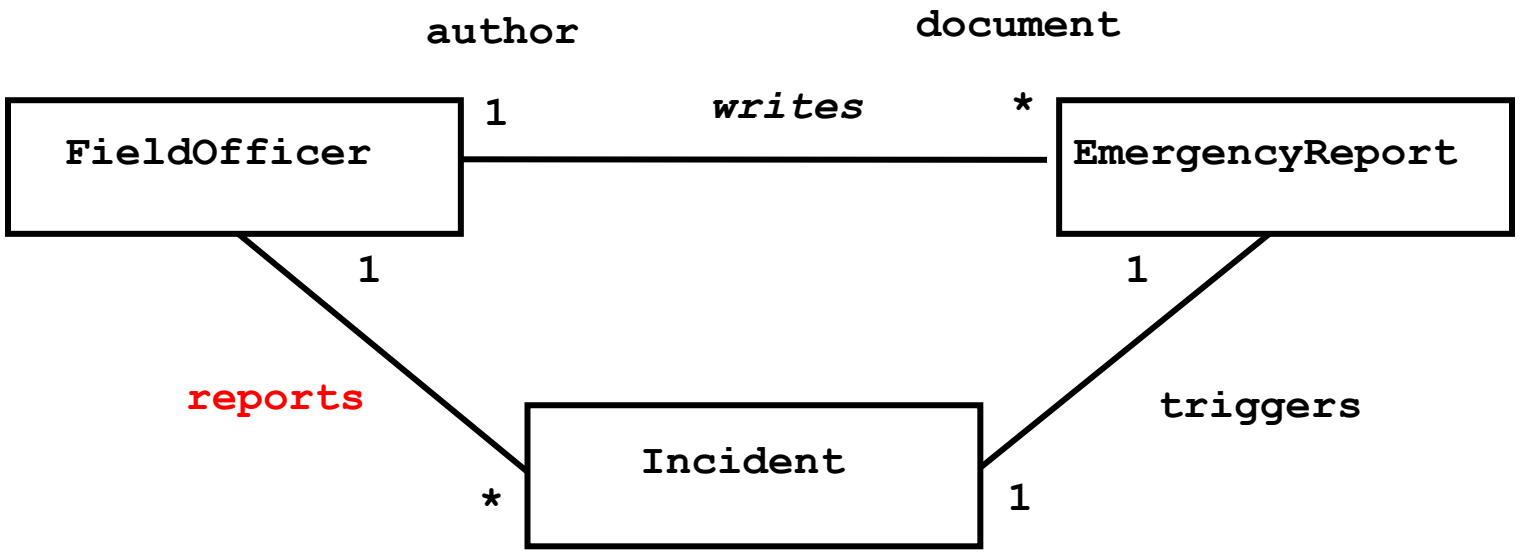
Association Heuristics: Refinement

- 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 (no message can be sent from one object to another)
  - 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



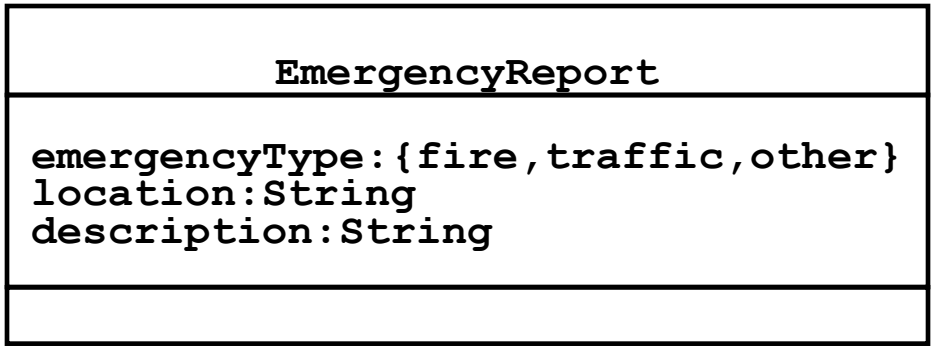
FRIEND Example



Identify Attributes

- "The name of the tournament" : Not all nouns become classes
- "… complete the form by specifying the type" : Not all verbs become associations.
- Attributes are properties of individual objects
  - Property is a partial aspect of an object (incomplete)
- 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.
- Least stable part of analysis object model

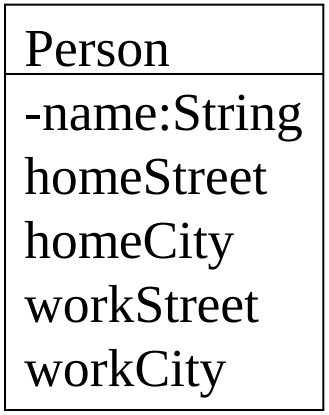
FRIEND Example

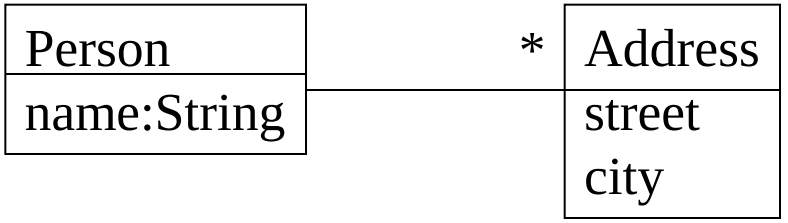


- Describe each attribute in data dictionary
  - –Name, brief description, type (legal values)

Attribute Heuristics (I)

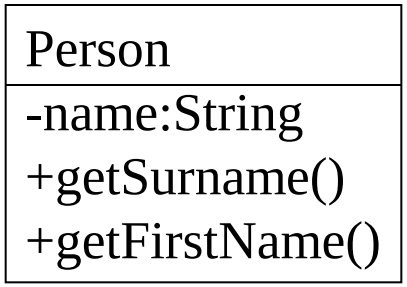
- Properties generally have simple types (or at least conceptually atomic)
  - If attribute is an object, use association instead
  - Exceptions: Strings, address, date
  - e.g., *FieldOfficer* who authored an *EmergencyReport*
- Word analysis: Possessive phrases and adjective phrases
- Nouns that are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

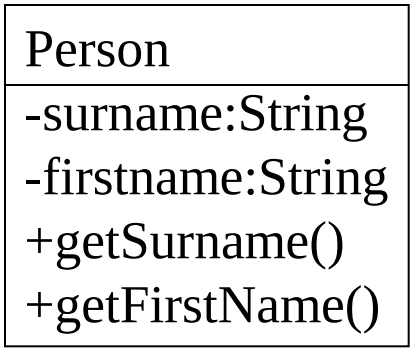




Attribute Heuristics (II)

- Attributes should not have an implicit internal structure.



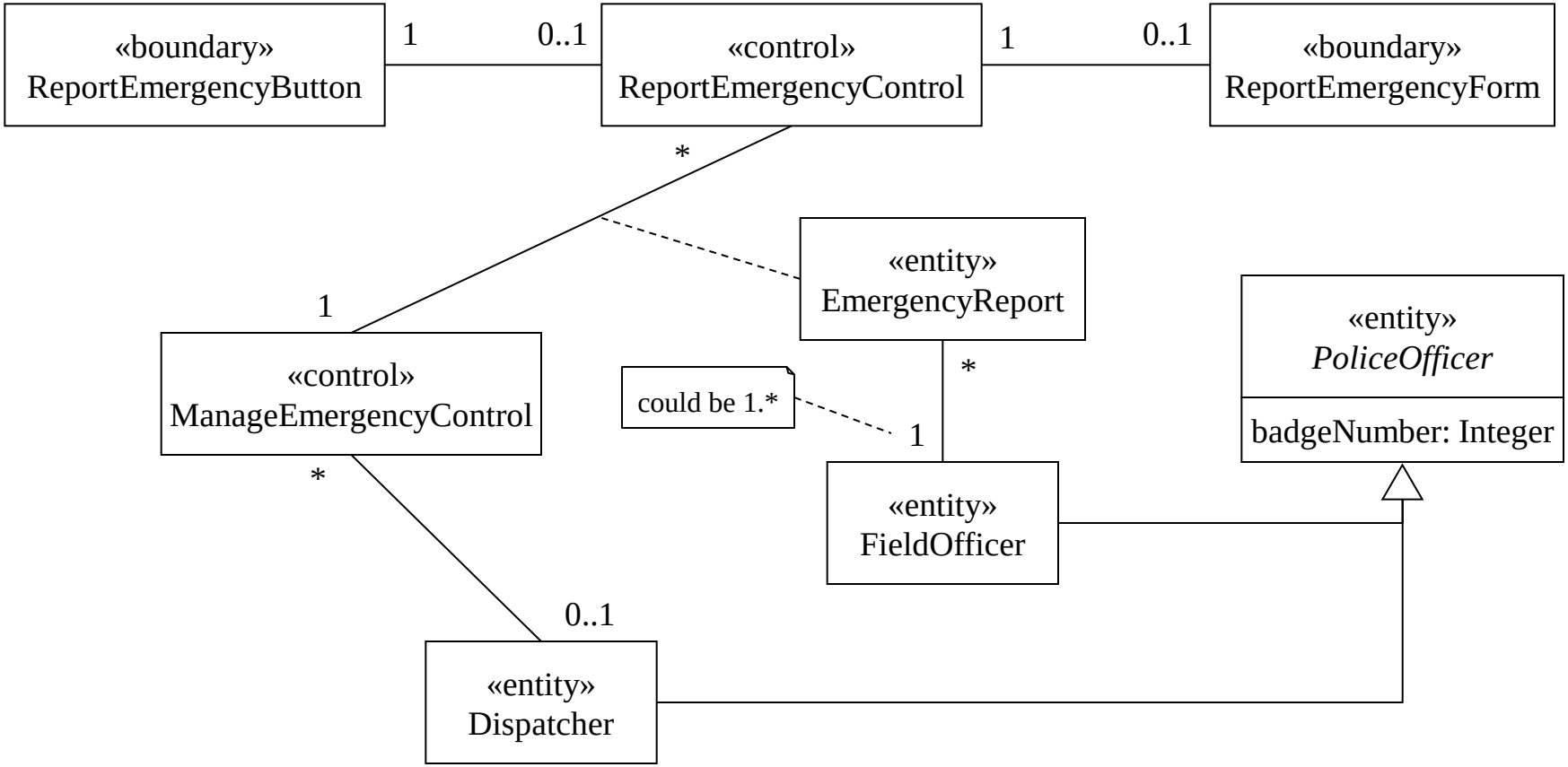


- Represent stored state as attribute of entity object

Generalization

- Eliminate redundancy in the analysis model
- Share attributes, operations
- *Dispatchers* and *Fieldofficers* both have *badgeNumber* to identify them within the city. They are both *PoliceOfficer*
- Abstract *PoliceOfficer* class, containing common functionality and attributes

PoliceOfficer



SYSC-3020—Introduction to Software Engineering

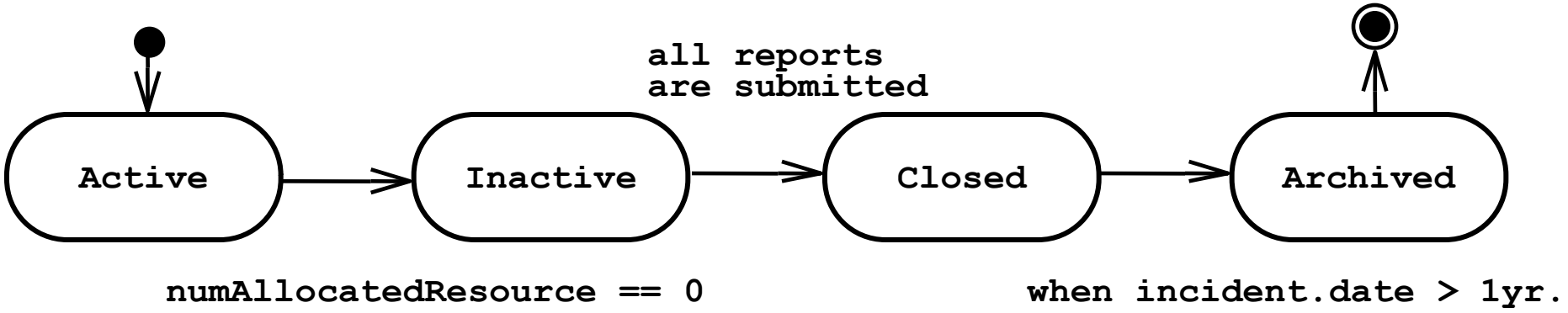
- Overview
- Analysis Concepts
  - Different kinds of classes
  - Different kinds of relationships
  - Modeling interaction
  - Modeling state-based behaviour
- Analysis Process
  - Finding objects/classes (heuristics)
  - Finding relationships: associations and attributes (heuristics)
  - Interactions/behavior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

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 to communicating operations.
- Sequence diagram shows some responsibilities of interacting operations.
- ➢ Sequence of messages must match operations pre and postconditions!

Library System (excerpt)

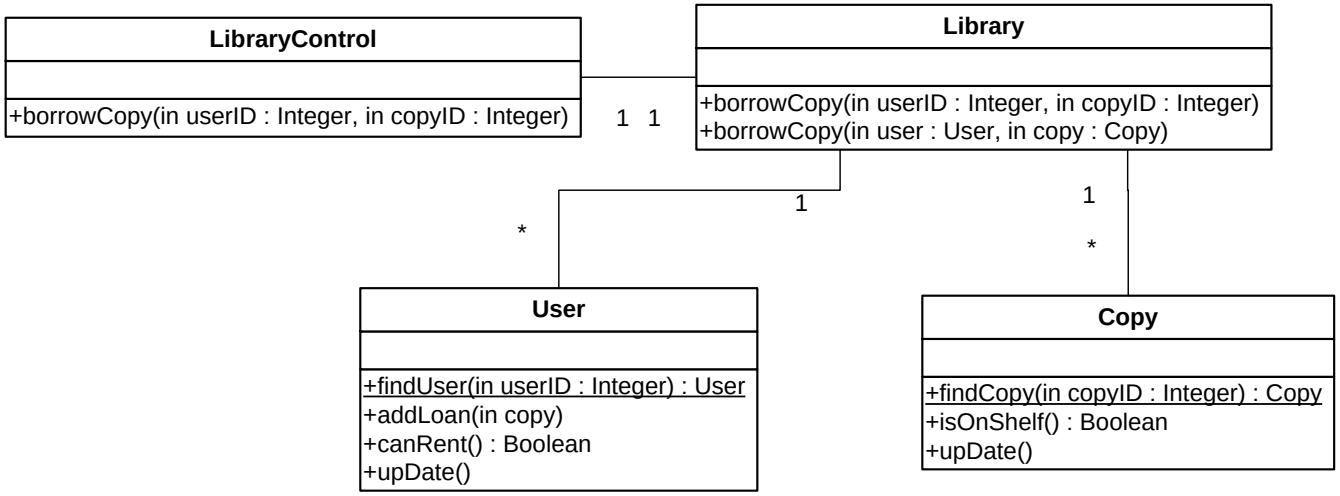
Operation borrowCopy(uid, cid) in class Library

Precondition:

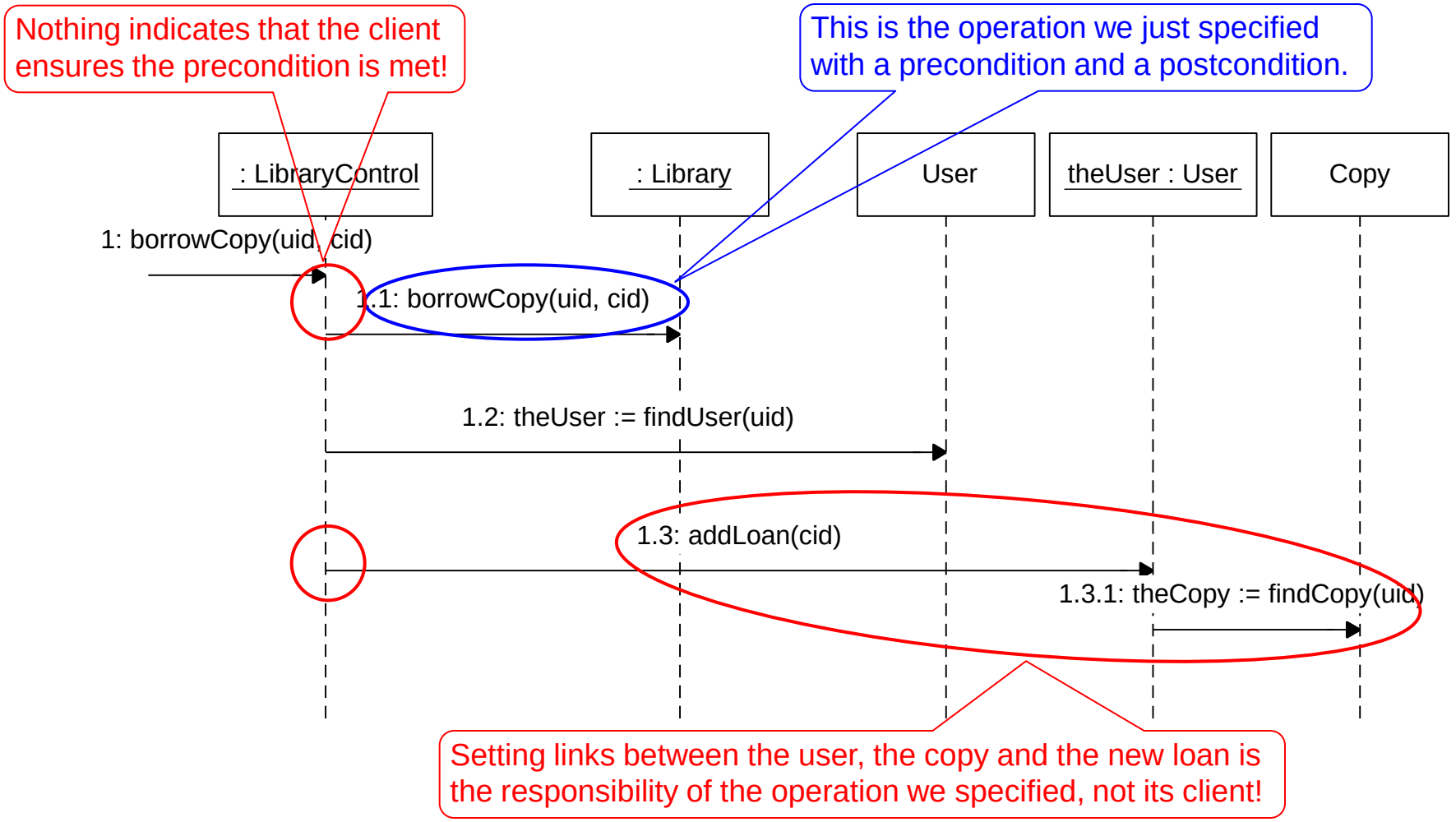
- There is a user with id uid who has not reached his/her maximum number of allowed rentals.
- There is a book with id cid on shelves (ready to rent).

Postcondition:

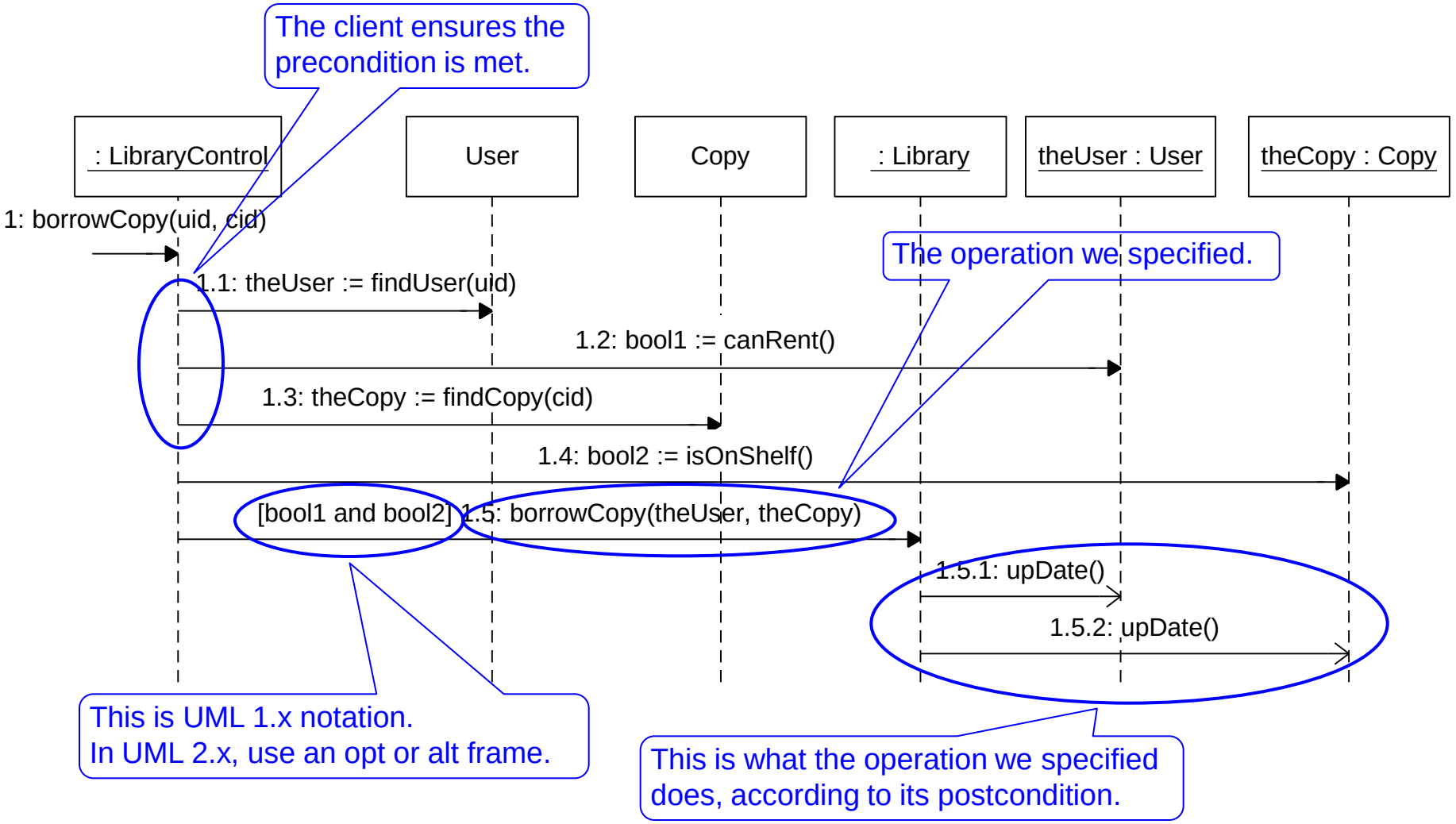
- There is no book on shelves with id cid.
- There is a loan object for a book with id cid.
- The user with id uid is renting one more copy and is linked to a loan for a book with id cid.



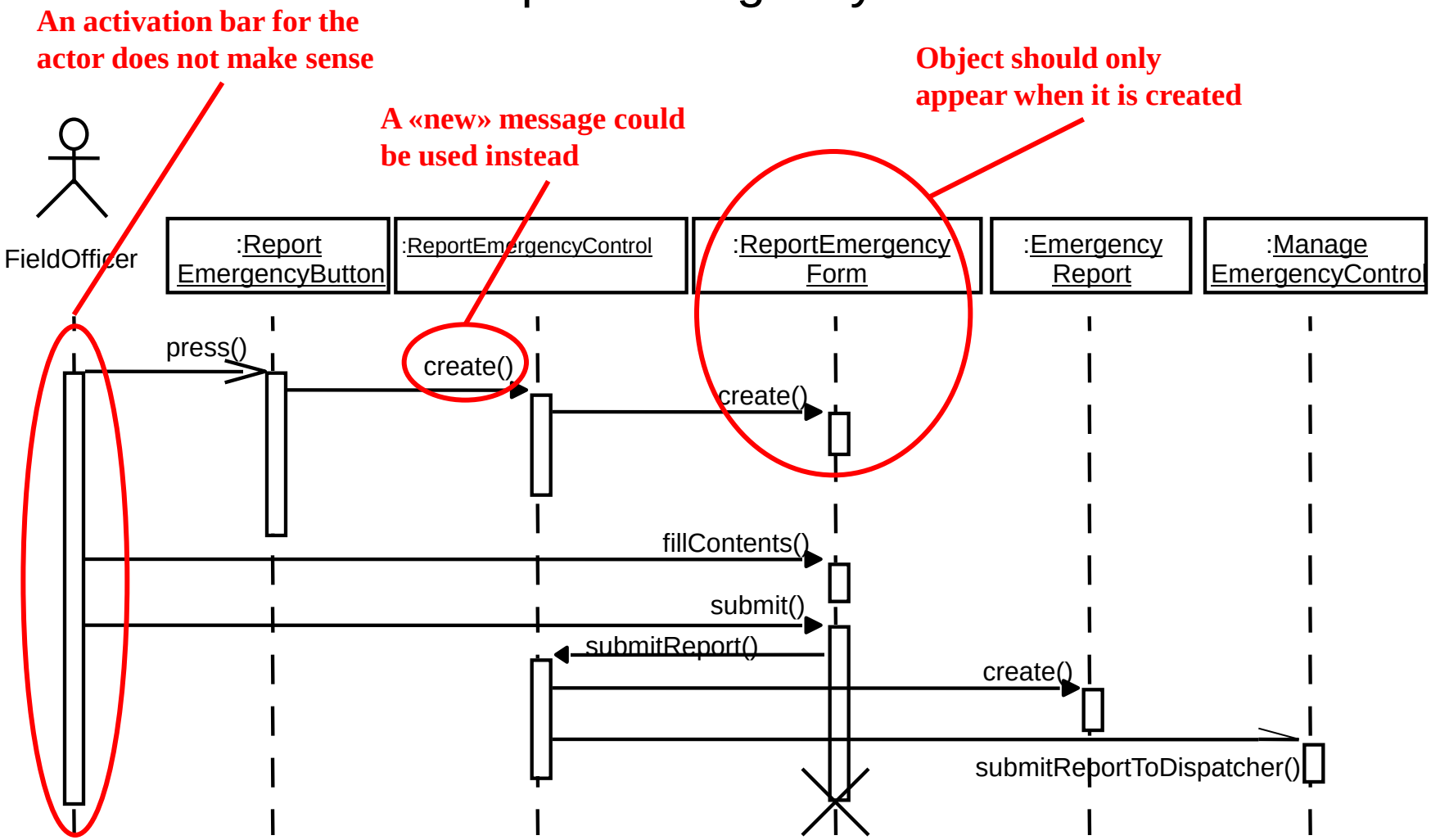
Sequence Diagram for Borrowing (excerpt)



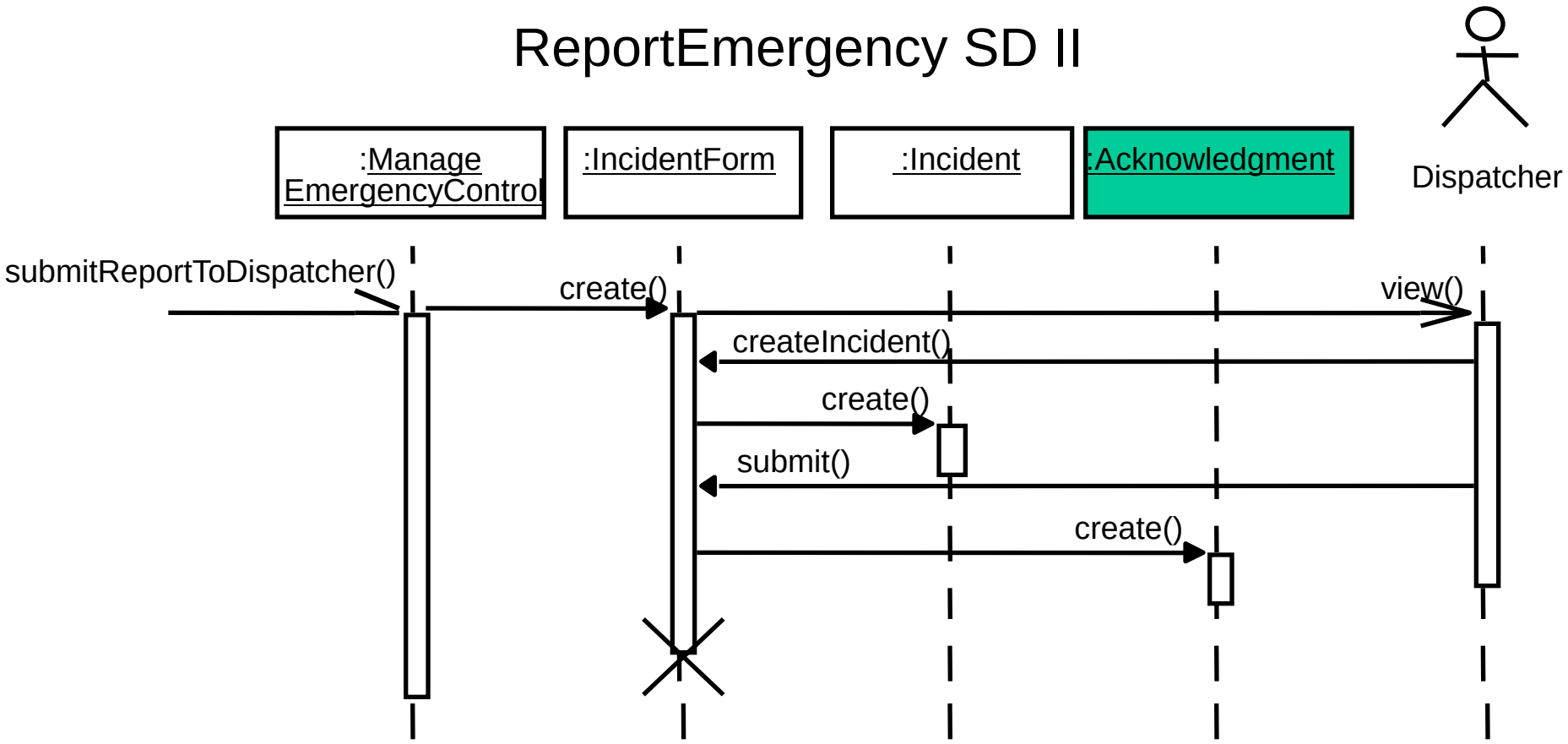
Sequence Diagram for Borrowing (excerpt)



ReportEmergency SD

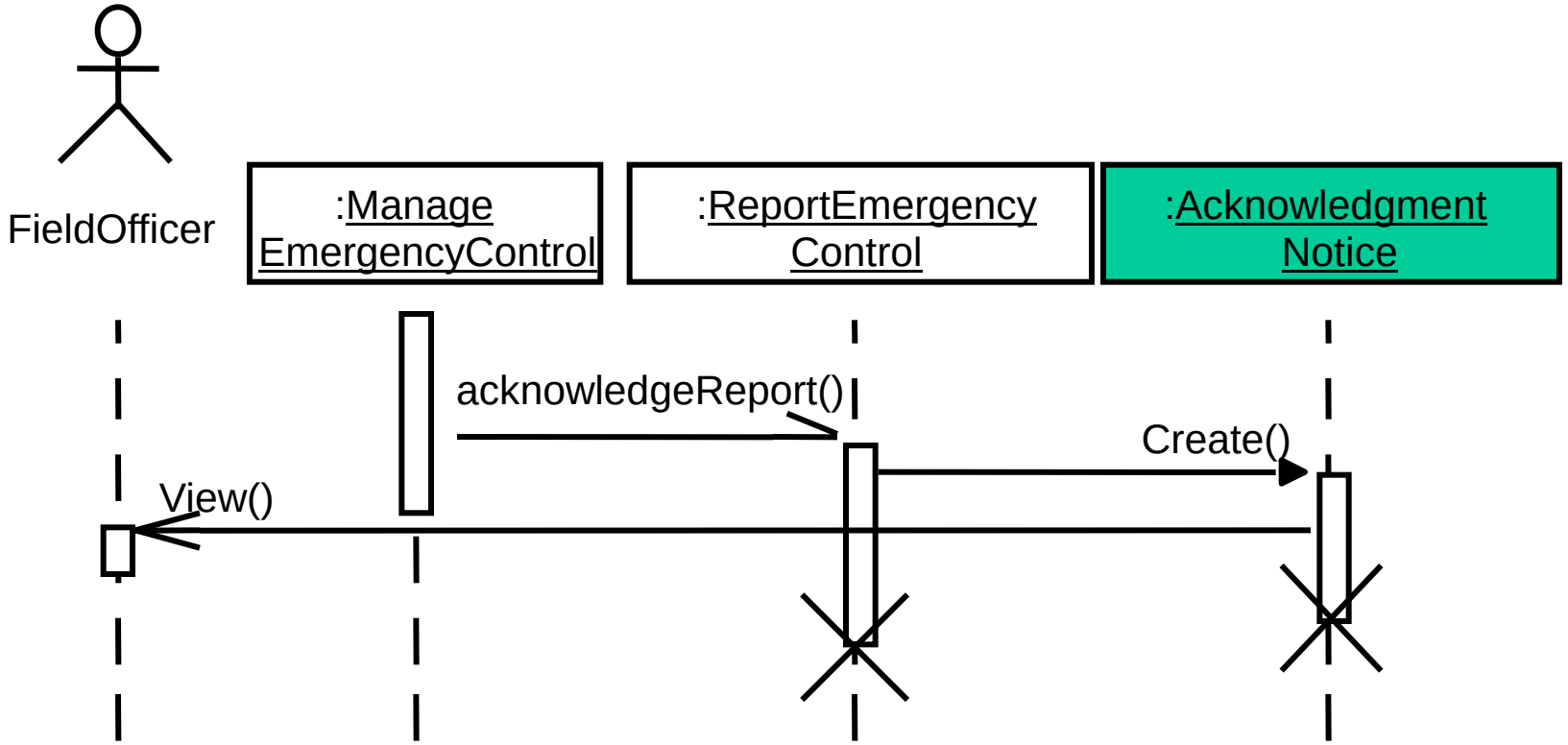


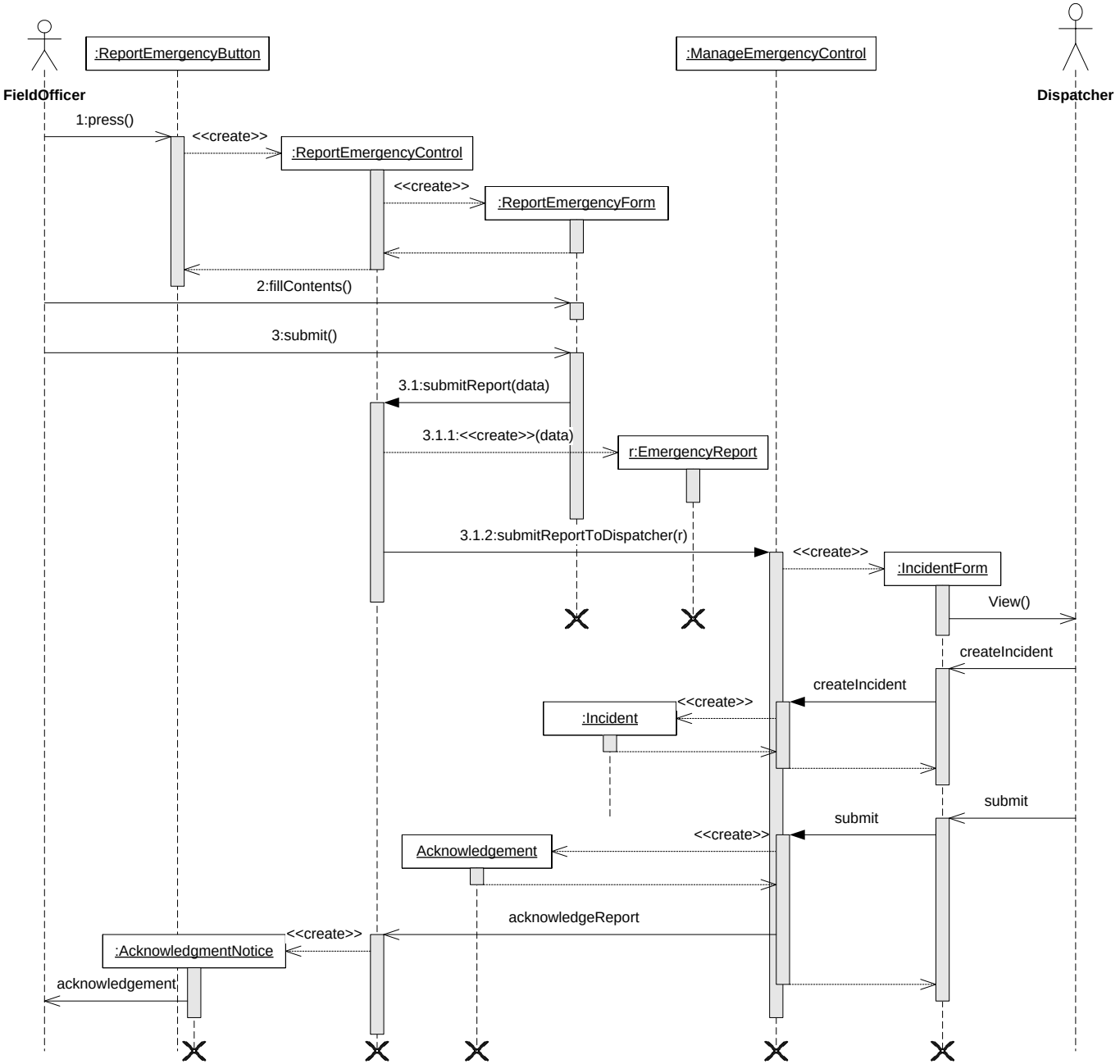
ReportEmergency SD II



- 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 III




FieldOfficer
:ReportEmergencyButton
:ManageEmergencyControl
Dispatcher
1:press()
<<create>>
:ReportEmergencyControl
<<create>>
:ReportEmergencyForm
2:fillContents()
3:submit()
3.1:submitReport(data)
3.1.1:<<create>>(data)
r:EmergencyReport
3.1.2:submitReportToDispatcher(r)
<<create>>
:IncidentForm
View()
createIncident
createIncident
<<create>>
:Incident
submit
submit
<<create>>
Acknowledgement
acknowledgeReport
<<create>>
:AcknowledgmentNotice
acknowledgement

Heuristics

- First column: actor who initiates the use case
- Second column: boundary object
- Third column: control object that manages the rest of the use case
- Control objects are created by boundary objects initiating use cases
- Other boundary objects are created by control 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-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

Object Responsibilities

- Sequence diagrams imply we distribute the behavior of the use case across participating objects.
- 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

- *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

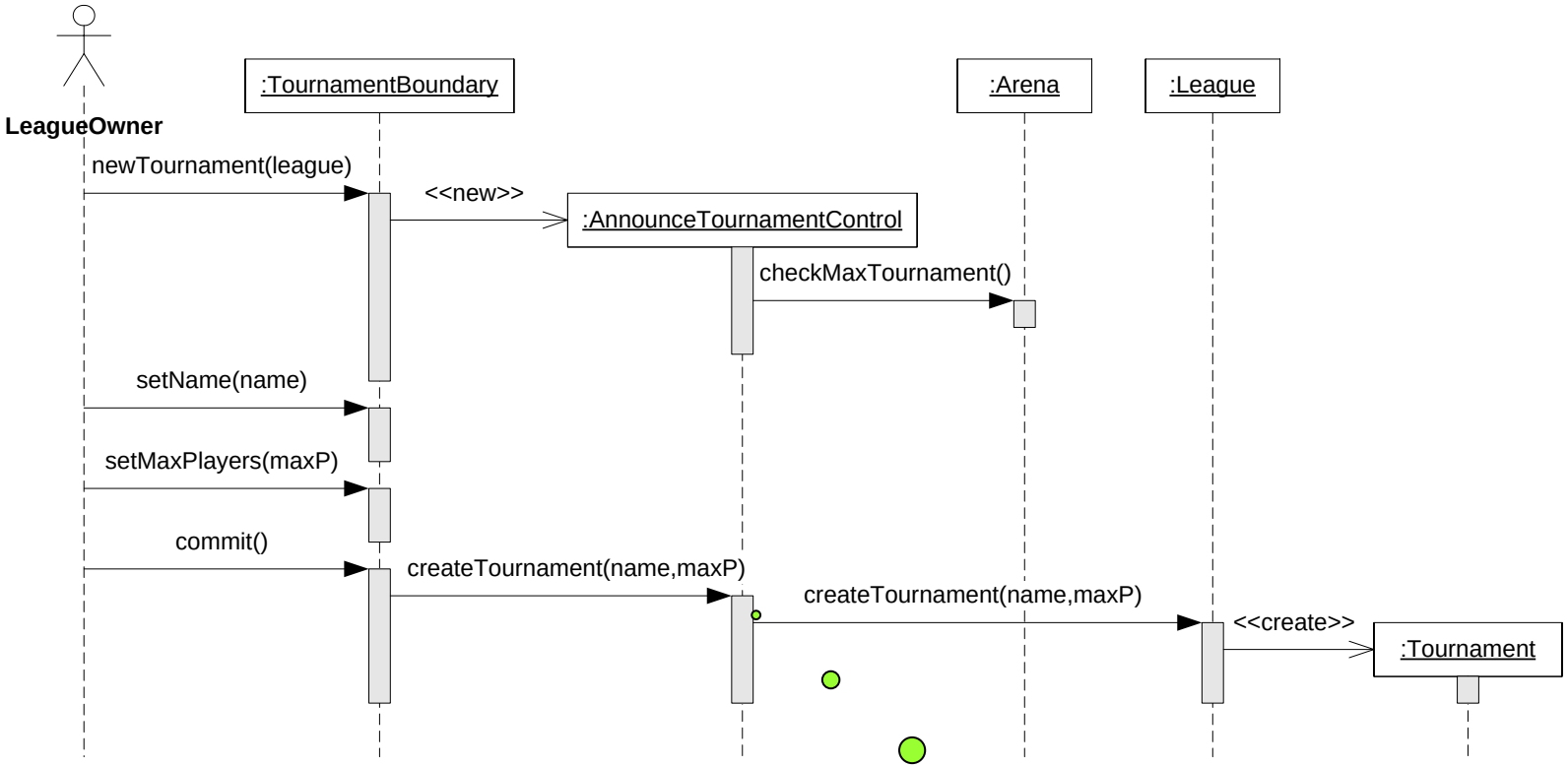
Cross-Checking

- Sequence diagrams can be used to help check the completeness / correctness of the use case model and class diagrams.
- 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)

Impact on ARENA's Object Model (ctd)

- The Sequence Diagram also supplied us with a lot of new events
  - newTournament(league)
  - setName(name)
  - setMaxPlayers(max)
  - Commit
  - checkMaxTournaments()
  - createTournament
- Question: Who owns these events?
- Answer: For each object that receives an event there is a public operation in the associated class.
  - The name of the operation is usually the name of the event.

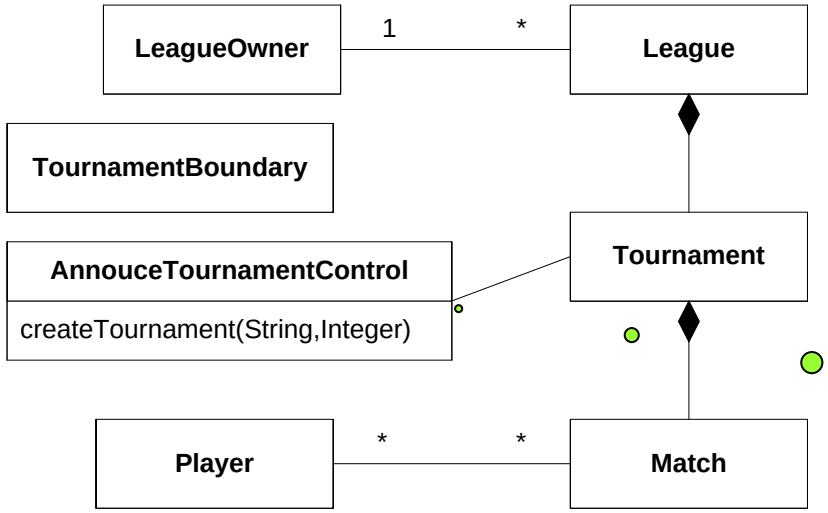
Arena Case study (cont.)



createTournament(String,Integer) is a public operation of AnnouceTournamentControl.

Arena Case study (cont.)

Updated class diagram



New operation and association

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 and Stair Diagrams (Ivar Jacobsen)

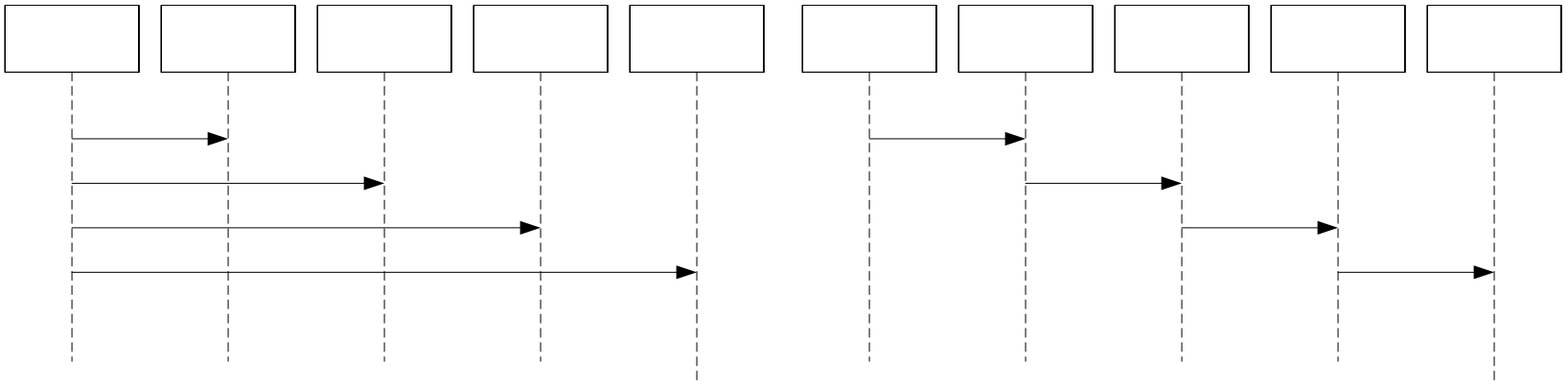
Fork and Stair Diagram

Fork Diagram

- Much of the dynamic behavior is placed in a single object, usually the control object. It 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 hel with a specific 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:
- Decentralized control structure
  - The operations have a strong connection
  - The operations will always be performed in the same order
- Centralized control structure (better support of change)
  - The operations can ***change*** order
  - New operations can be inserted as a result of new requirements

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