Object Oriented Analysis

Lecture 5

Outline

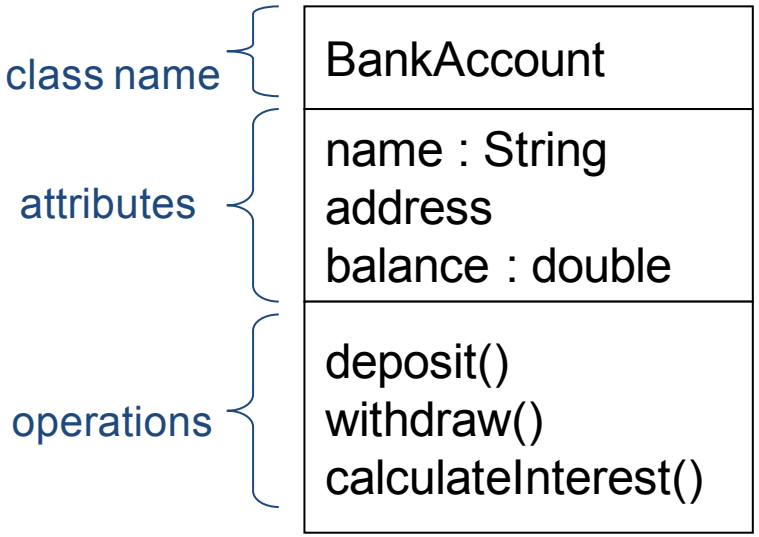
- Objects and classes [Lecture 4]
- Finding analysis classes
- Relationships between objects and classes
  - Links
  - Associations
  - Dependencies
- Inheritance and polymorphism
- Interaction diagrams

Finding Analysis Classes

Lecture 5/Part 1

What are Analysis classes?

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



Specify attribute types if you know what they are.

What makes a good analysis class?

- Its name reflects its intent
- It is a *crisp abstraction* that models one specific element of the problem domain
  - It maps onto a clearly identifiable feature of the problem domain
- It has *high cohesion*
  - Cohesion is the degree to which a class models a single abstraction
  - Cohesion is the degree to which the *responsibilities* of the class are semantically related
- It has *low coupling*
  - Coupling is the degree to which one class depends on others

Rules of thumb

- ✧ 3 to 5 responsibilities per class
- ✧ Each class collaborates with others
- ✧ Beware many very small classes
- ✧ Beware few but very large classes
- ✧ Beware of “functoids”
- ✧ Beware of “omnipotent” classes
- ✧ Avoid deep inheritance trees

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

Finding classes

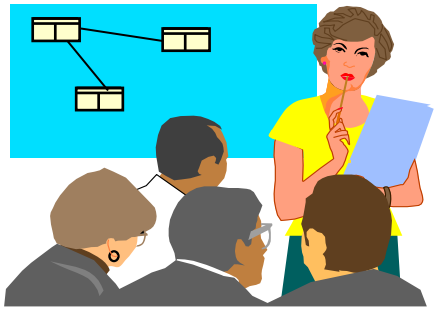
- ✧ Perform noun/verb analysis on documents:
  - Nouns are candidate **classes**
  - Verbs are candidate **responsibilities**

  What documents can be studied?
- ✧ Perform CRC card analysis
  - Class, Responsibilities and Collaborators
  - A two phase brainstorming technique using sticky notes – first brainstorm and then analyse the dat

Class Name: BankAccount	
Responsibilities:	Collaborators:
Maintain balance	Bank

things the class does

things the class works with



Other sources of classes

- Physical objects
- Paperwork, forms
  - Be careful when relying on processes that need to change
- Known interfaces to the outside world
- Conceptual entities that form a cohesive abstraction

- **With all techniques, beware of spurious classes**
  - Look for *synonyms* - different words that mean the same
  - Look for *homonyms* - the same word meaning different things

Key points

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

Relationships Between Objects and Classes

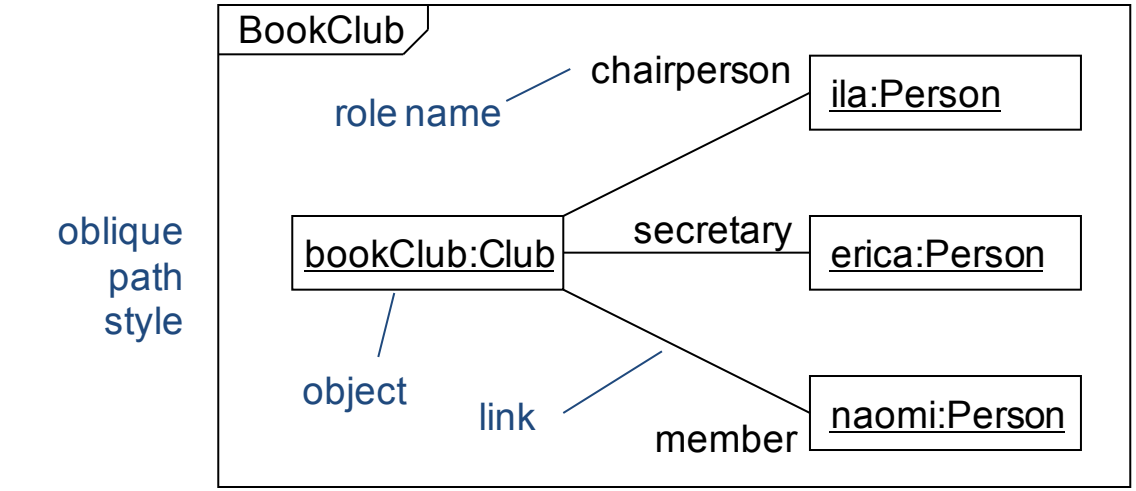
Lecture 5/Part 2

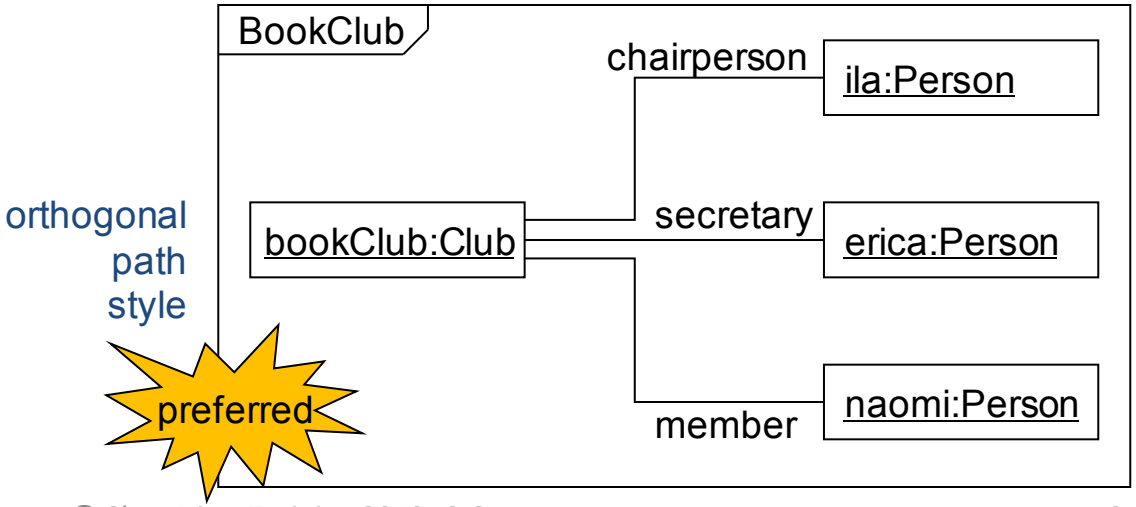
What is a link?

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

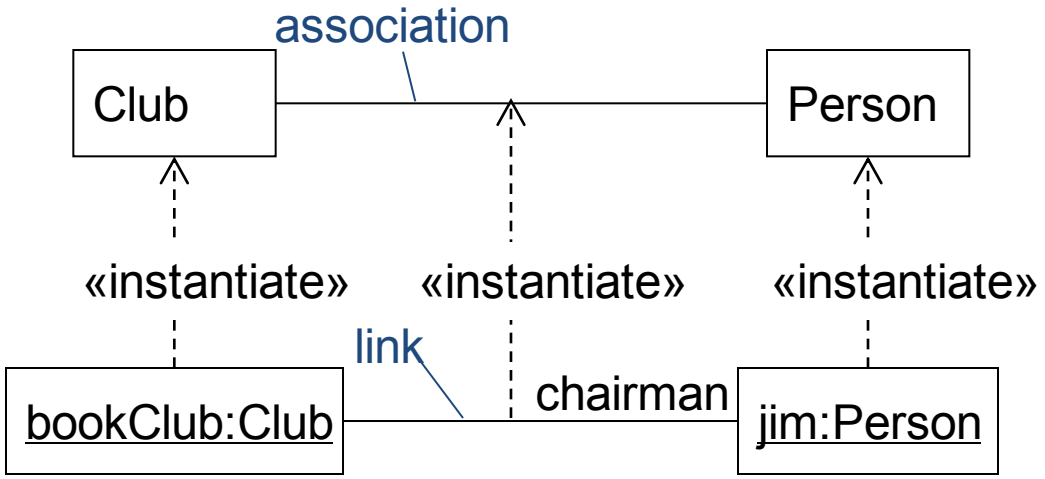
Object diagrams

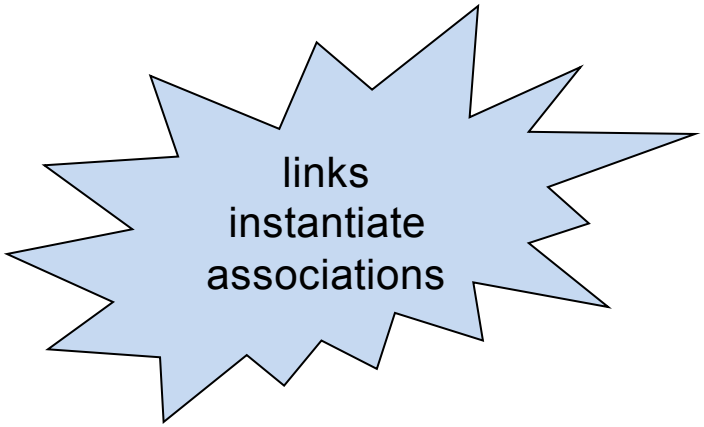
- Paths in UML diagrams (lines to you and me!) can be drawn as orthogonal, oblique or curved lines
- We can combine paths into a tree *if* each path has the same properties





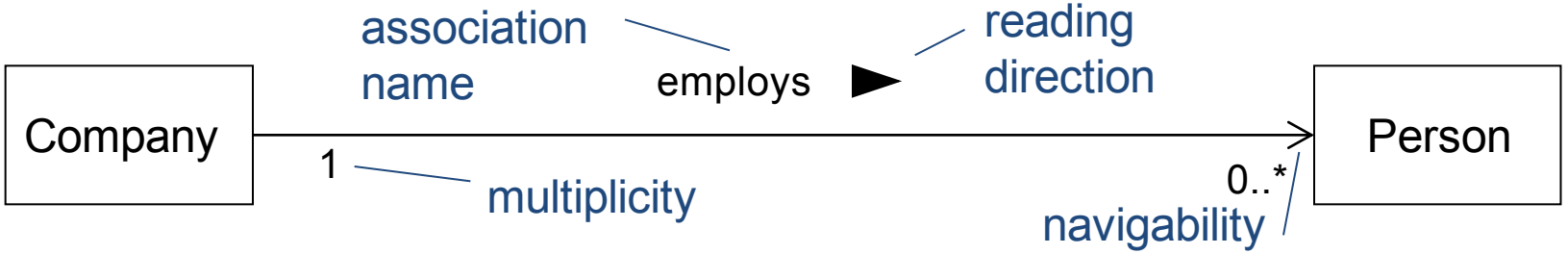
What is an association?

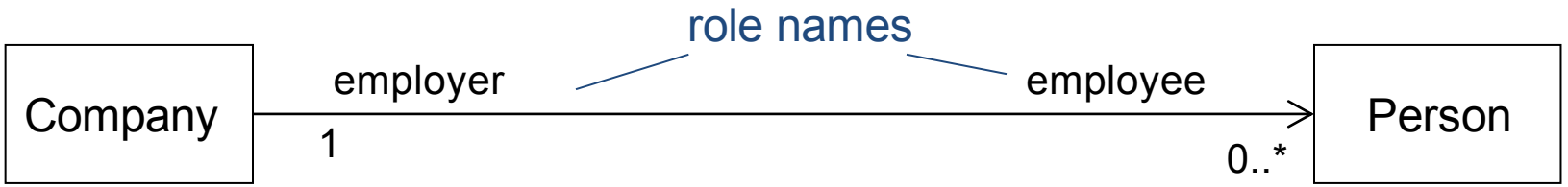




- Associations are relationships between classes
- Associations between classes indicate that there **may be** links between objects of those classes, while links indicates that there **must be** associations
- Can there be a communication between objects of two classes that have no association between them?

Association syntax



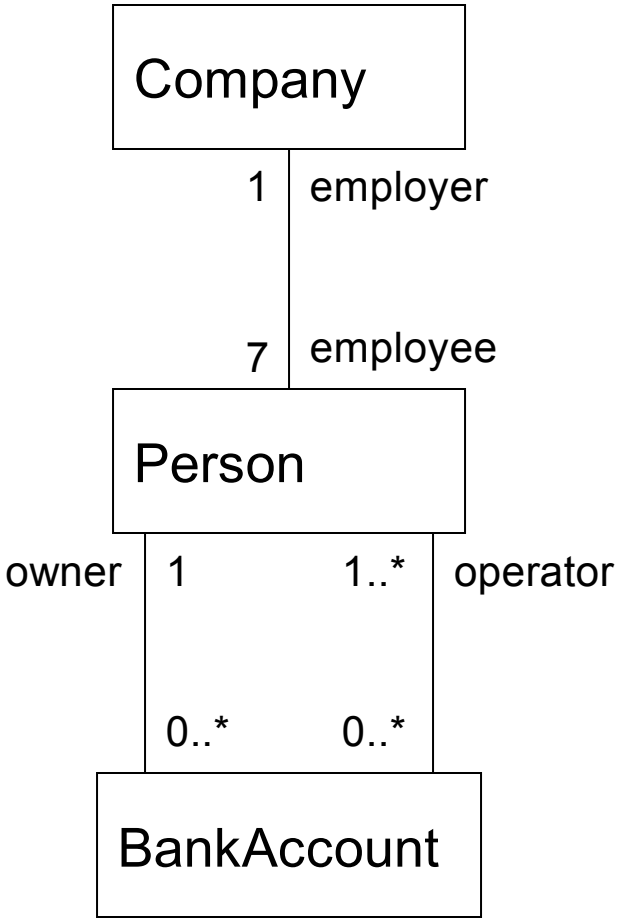


- An association can have role names *or* an association name
- Multiplicity is a constraint that specifies the number of objects that can participate in a relationship at any point in time
  - If multiplicity is not explicitly stated in the model then it is undecided – *there is no default multiplicity*

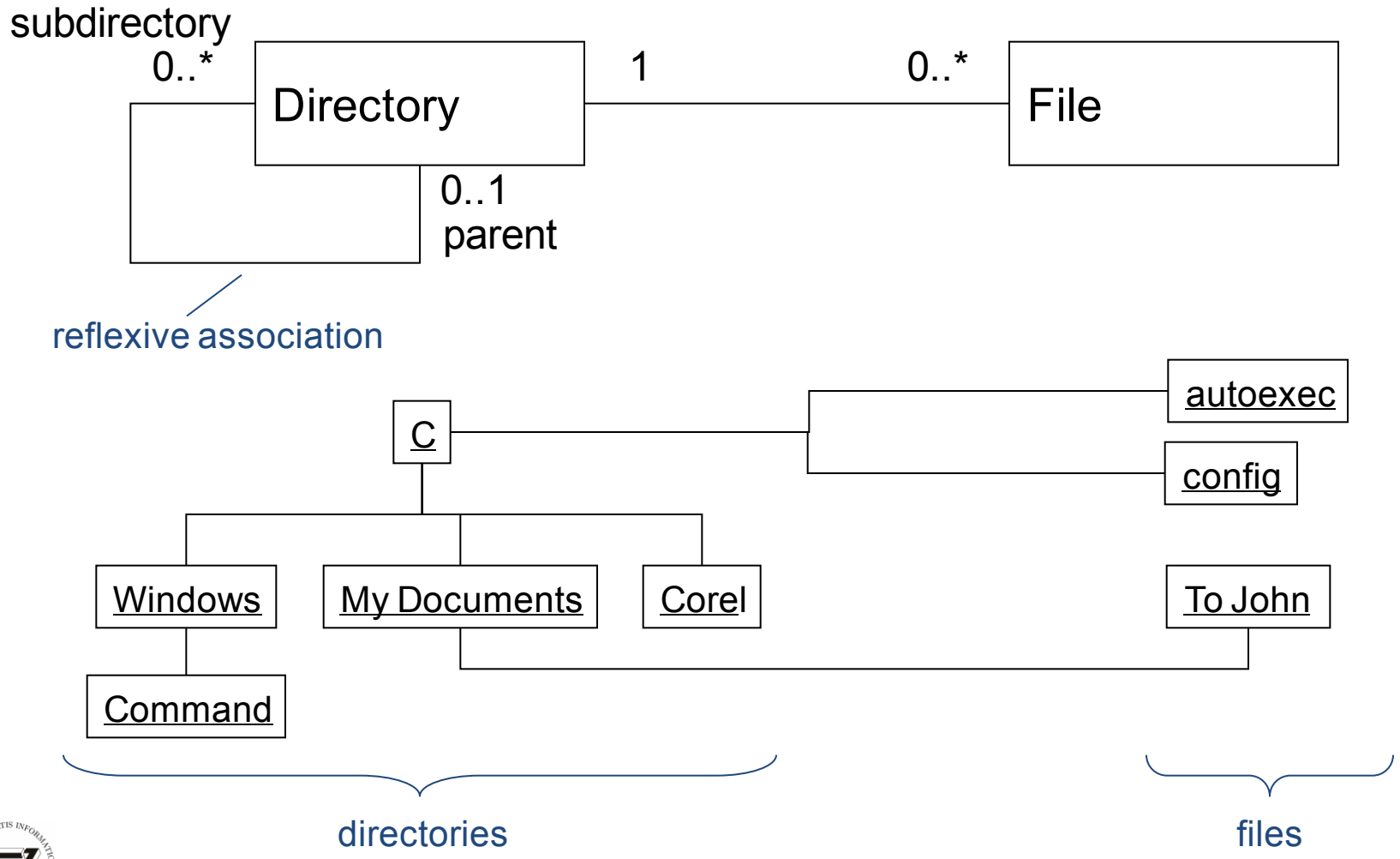
multiplicity: min..max	
0..1	zero or 1
1	exactly 1
0..*	zero or more
1..*	1 or more
1..6	1 to 6

Multiplicity exercise

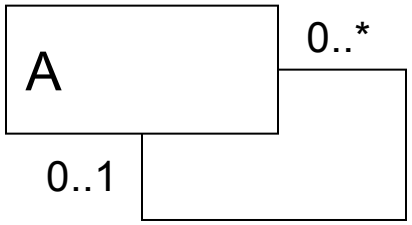
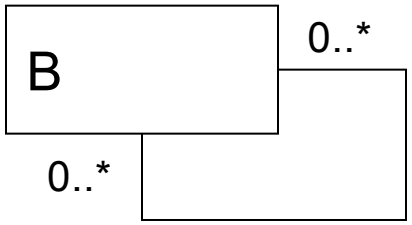
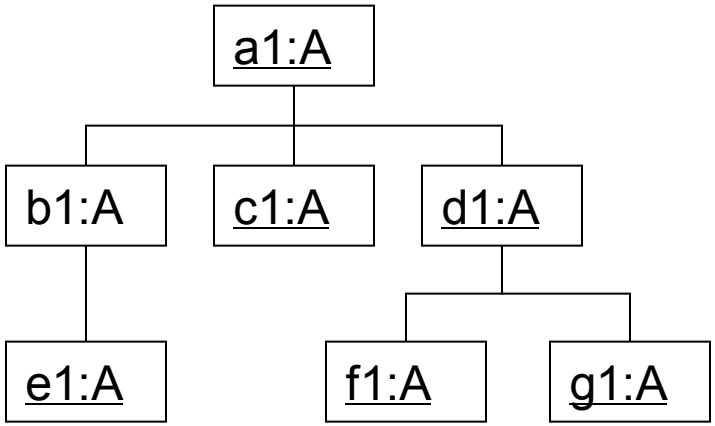
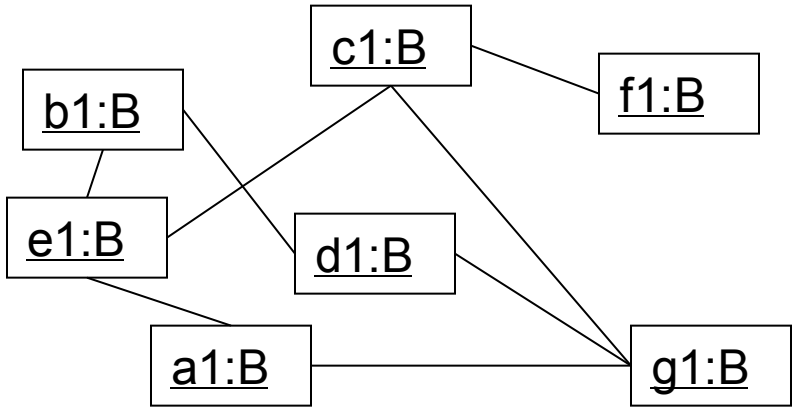
- How many
  - Employees can a Company have?
  - Employers can a Person have?
  - Owners can a BankAccount have?
  - Operators can a BankAccount have?
  - BankAccounts can a Person have?
  - BankAccounts can a Person operate?



Reflexive associations: file system example



Hierarchies and networks

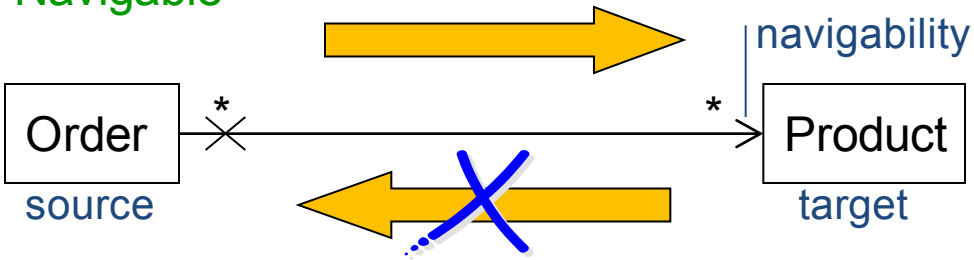
hierarchy	network
	
	
In an association hierarchy, each object has *zero or one* object directly above it.	In an association network, each object has zero or many objects directly above it.

Navigability

- Navigability indicates that it is possible to traverse from an object of the *source* class to objects of the *target* class

An Order object stores a list of Products

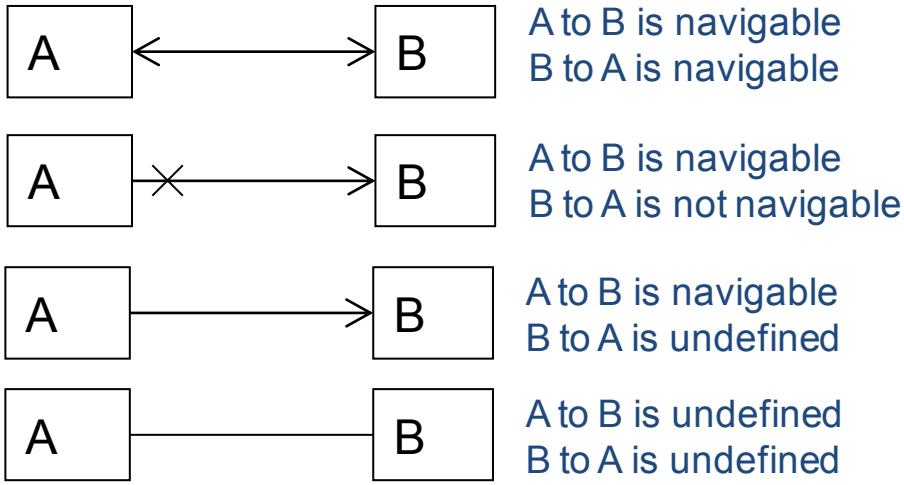
Navigable



Not navigable

A Product object does not store a list of Orders

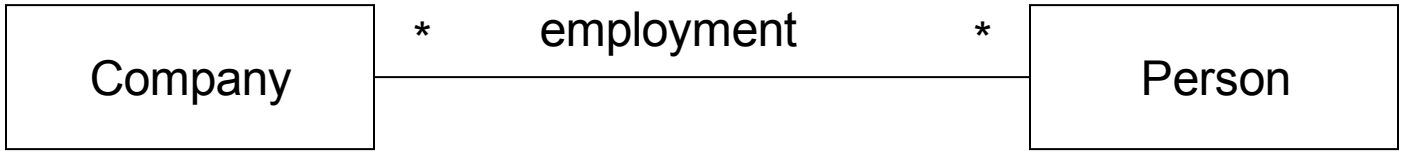
- Can there be a communication in a direction not supported by the navigability?
- Are some of the cases on the right equivalent?



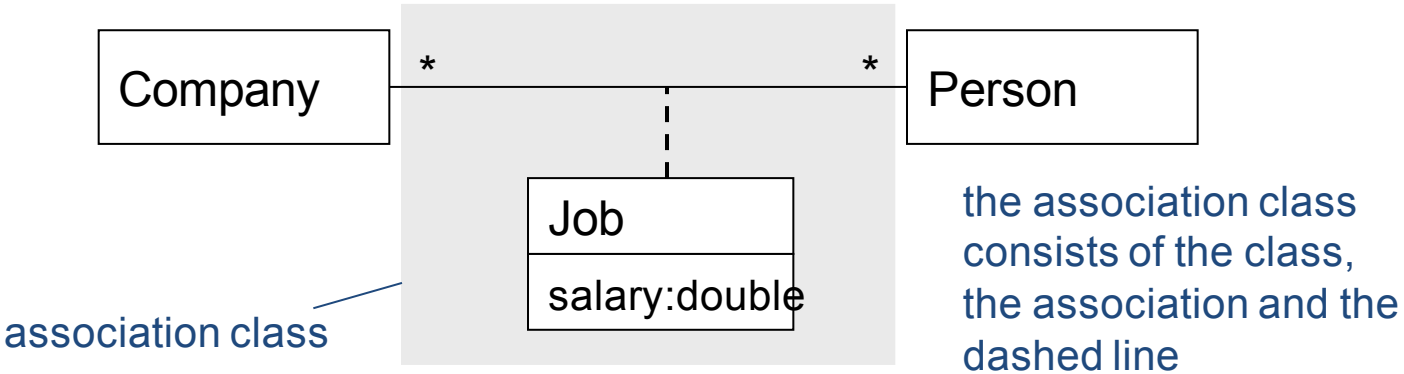
Associations and attributes

- An association is (through its role name) a substitutional representation of an attribute
  - **Use associations when:**
    - The target class is an important part of the model
    - The target class is a class that you have designed yourself and must be shown on the model
  - **Use attributes when:**
    - The target class is *not* an important part of the model e.g. a primitive type such as number, string
    - The target class is just an implementation detail such as a bought-in component or a library component e.g. Java.util.Vector (from the Java standard libraries)

Association classes

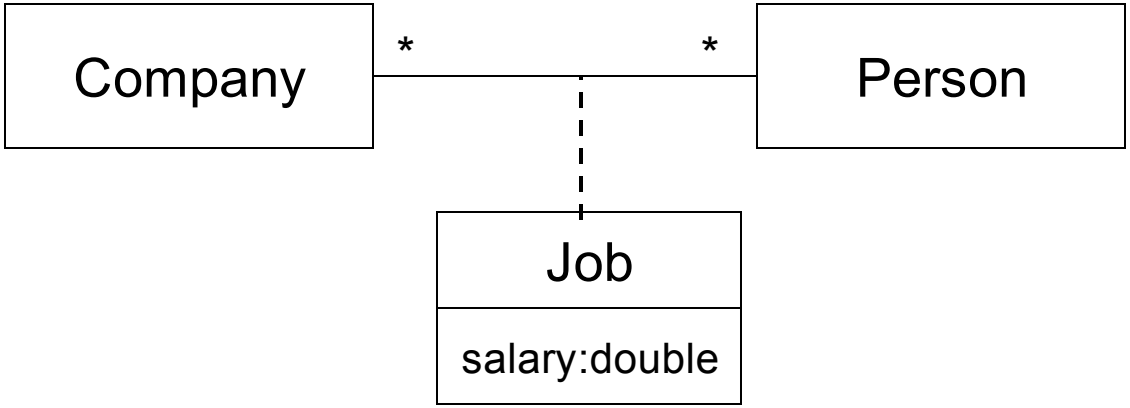


- Where do we record the Person's salary?
- We model the association itself as an association class. Exactly one instance of this class exists for each link between a Person and a Company.
- We can place the salary and any other attributes or operations which are really features of the association into this class

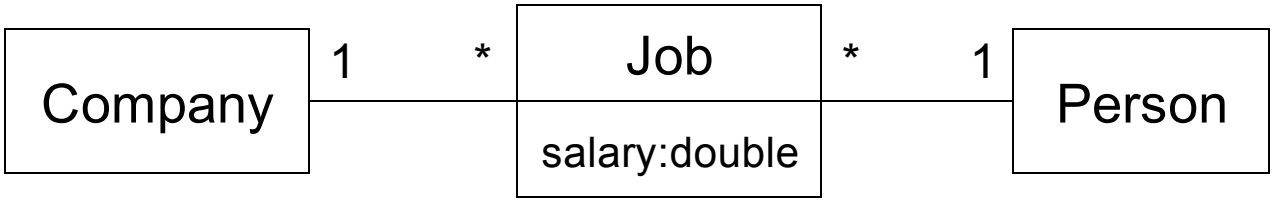


Using association classes

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



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

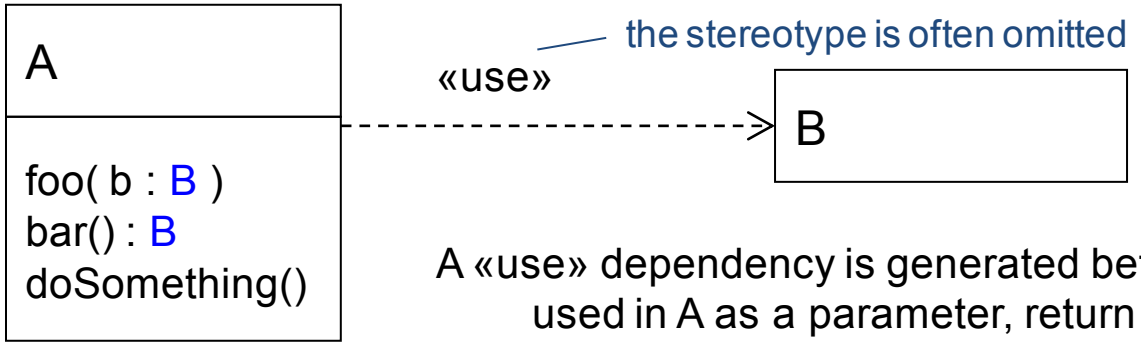


Dependencies

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

Usage dependencies

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




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

A «use» dependency is generated between A and B when B is used in A as a parameter, return value or inside method body

Abstraction and permission dependencies

- Abstraction dependencies
  - «trace» - the client and the supplier represent the same concept but at different points in development
  - «substitute» - the client may be substituted for the supplier at runtime. The client and supplier must realize a common contract. Use in environments that *don't* support specialization/generalization
  - «refine» - the client represents a fuller specification of the supplier
  - «derive» - the client may be derived from the supplier. The client is logically redundant, but may appear for implementation reasons
- Permission dependencies
  - «access» the public contents of the supplier package are added as private elements to the namespace of the client package
  - «import» the public contents of the supplier package are added as public elements to the namespace of the client package
  - «permit» the client element has access to the supplier element despite the declared visibility of the supplier

Key points

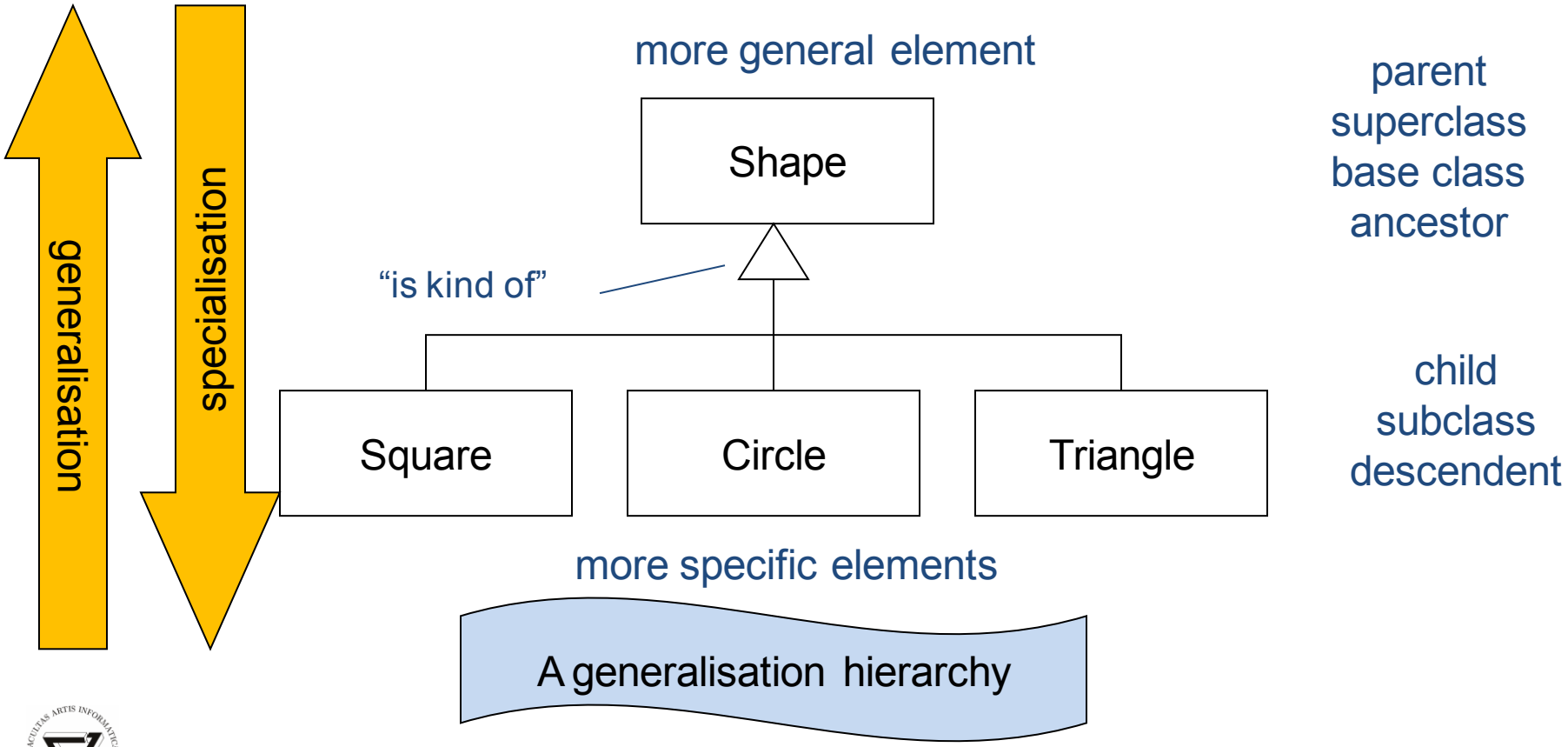
- Links – relationships between objects
- Associations – relationships between classes
  - role names
  - multiplicity
  - navigability
  - association classes
- Dependencies – relationships between model elements
  - usage
  - abstraction
  - permission

Inheritance and polymorphism

Lecture 5/Part 3

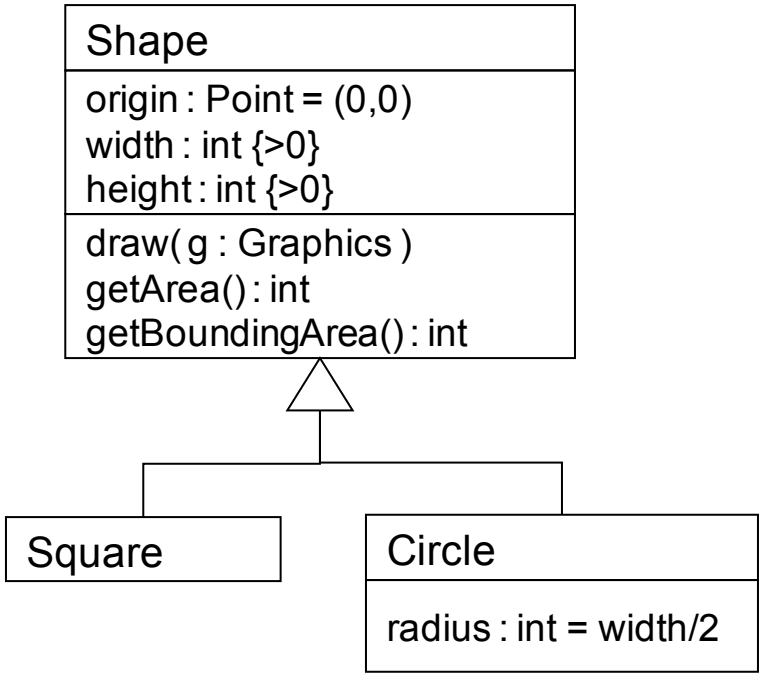
Generalisation

A relationship between a more general element and a more specific element (with more information)



Class inheritance

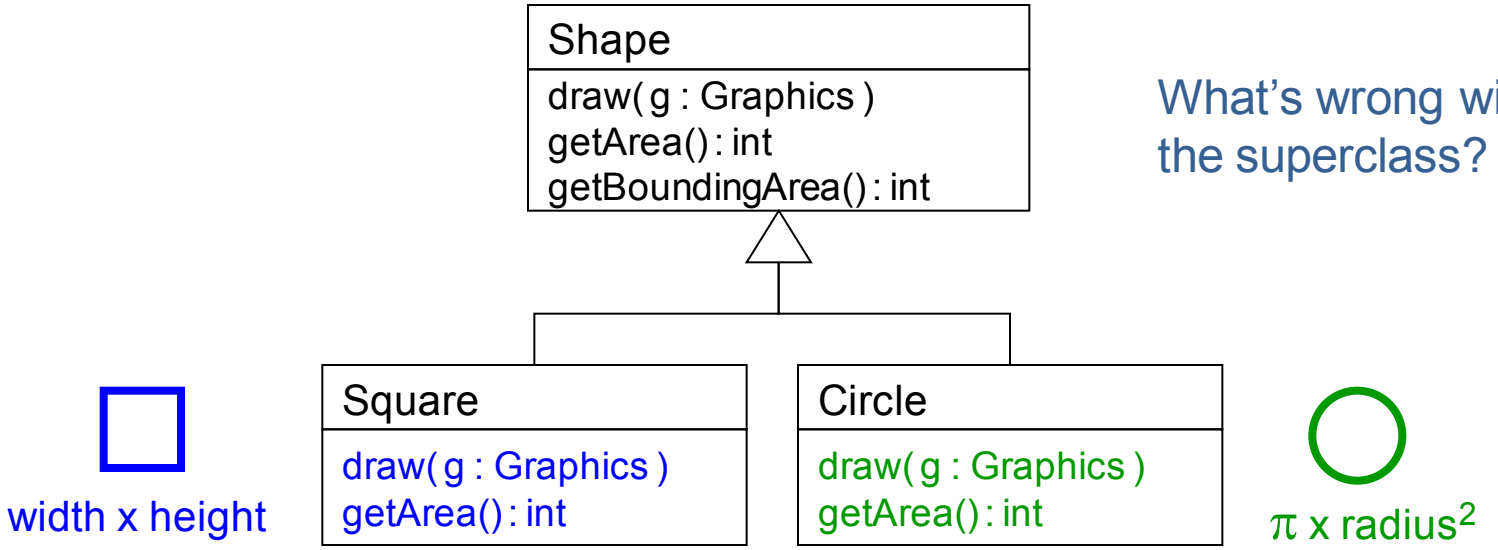
- Subclasses inherit *all* features of their superclasses:
  - attributes
  - operations
  - relationships
  - stereotypes, tags, constraints
- Subclasses can add new features
- Subclasses can override superclass operations
- We can use a subclass instance *anywhere* a superclass instance is expected



Substitutability Principle

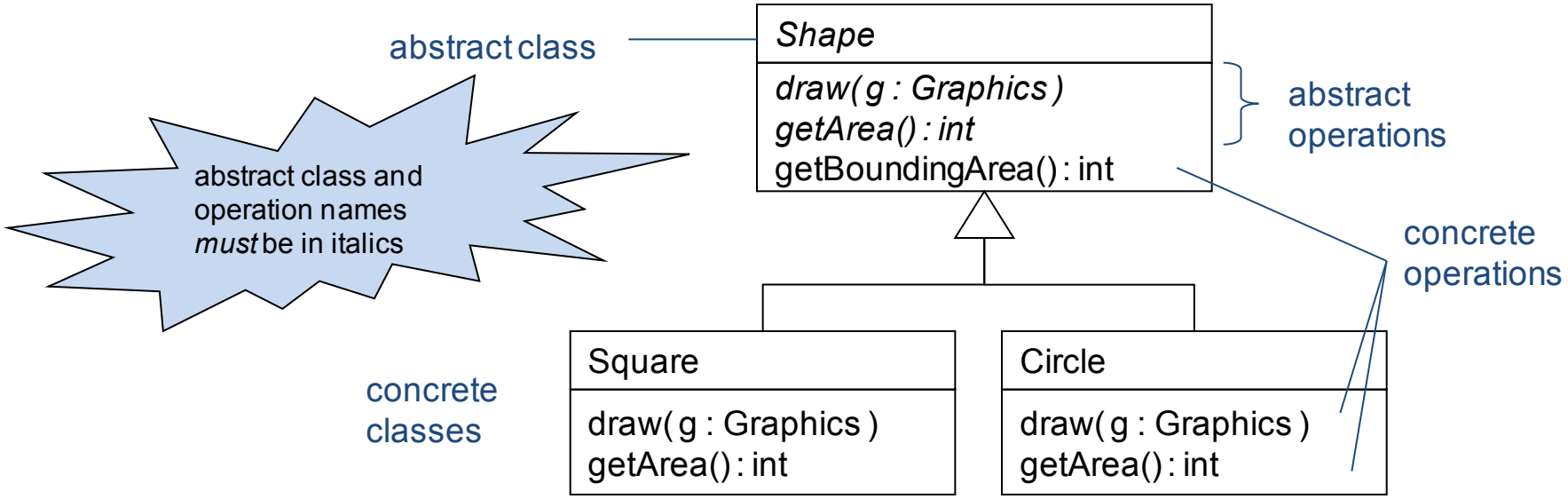
What's wrong with these subclasses?

Overriding



- Subclasses often need to *override* superclass behaviour
- To override a superclass operation, a subclass must provide an operation with the same signature
  - The operation signature is the operation name, return type and types of all the parameters

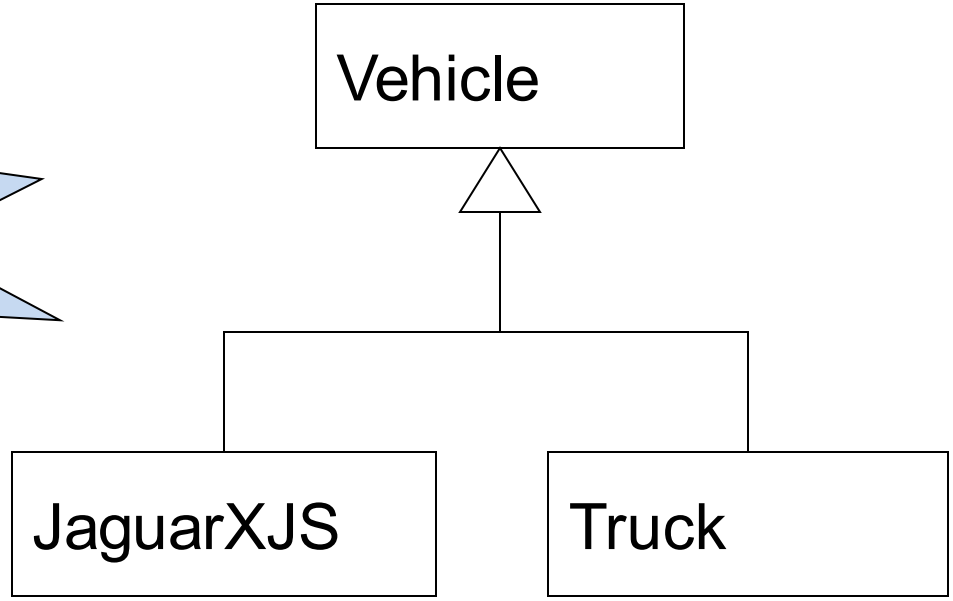
Abstract operations & classes



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

Exercise

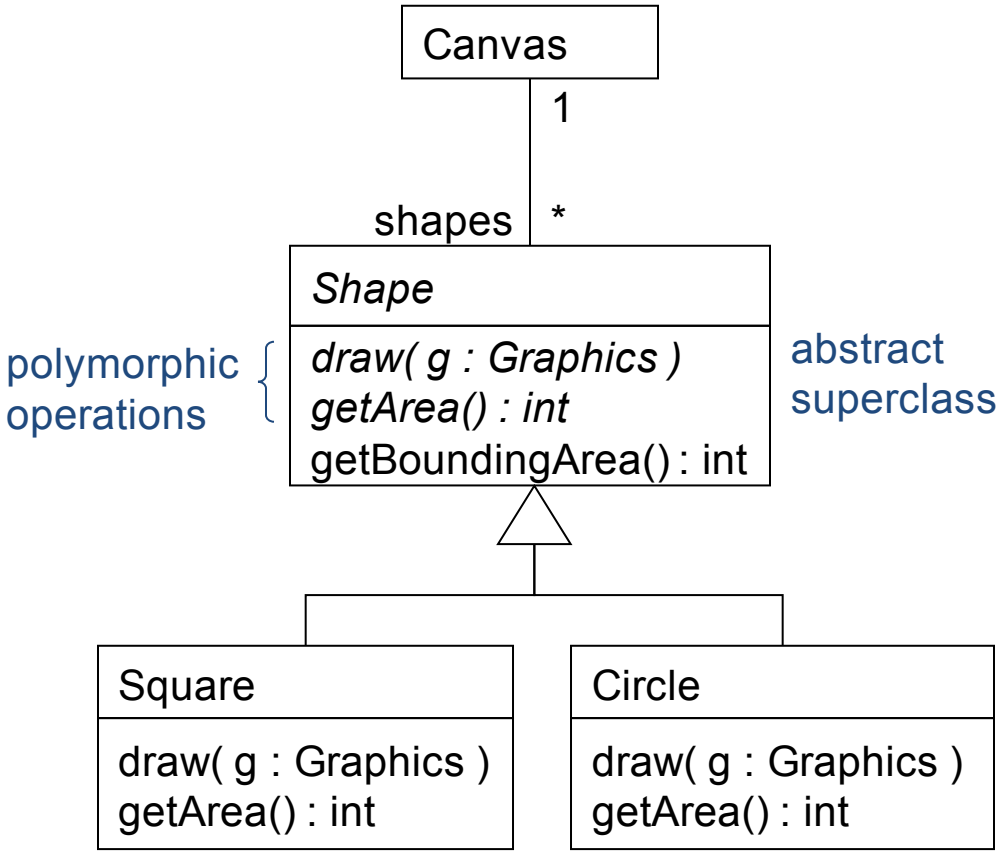
what's wrong with this model?



Polymorphism

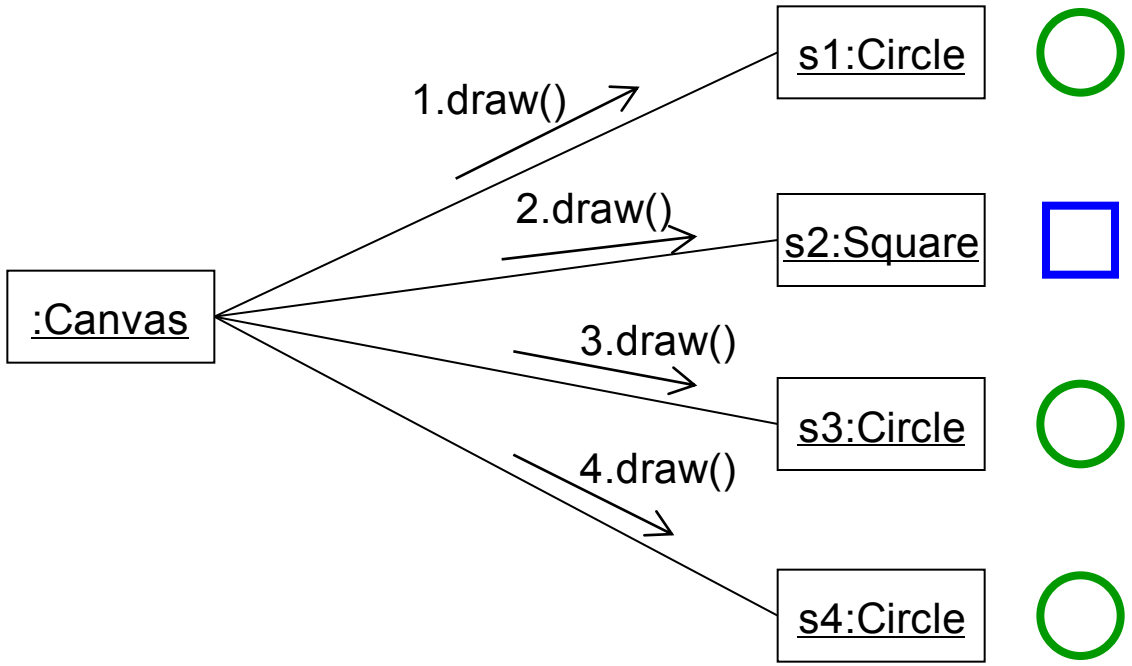
- Polymorphism = "many forms"
  - A polymorphic operation has many implementations
  - Square and Circle provide implementations for the polymorphic operations *Shape::draw()* and *Shape::getArea()*
- The operation in Shape superclass defines a contract for the subclasses.

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

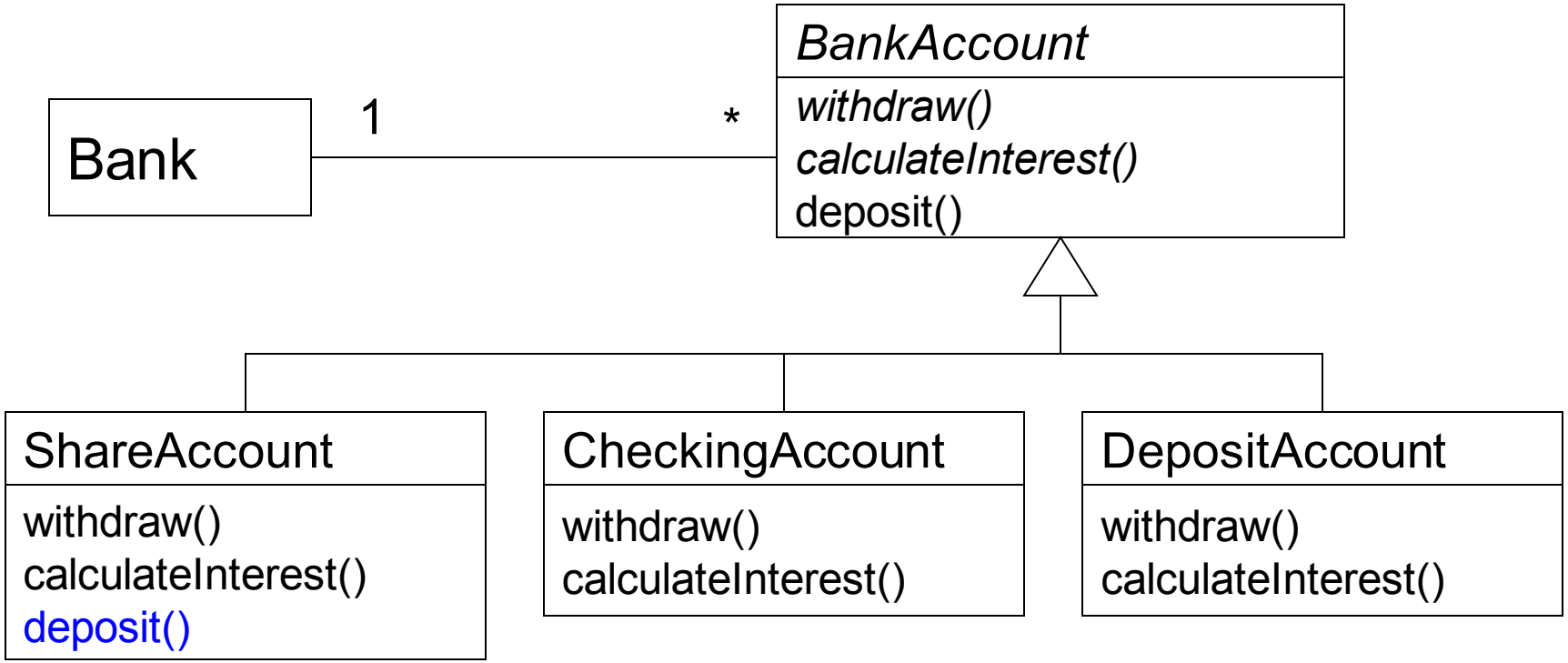


What happens?

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



BankAccount example



- We have overridden the deposit() operation even though it is *not* abstract.

Key points

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

Interaction Diagrams

Lecture 5/Part 4

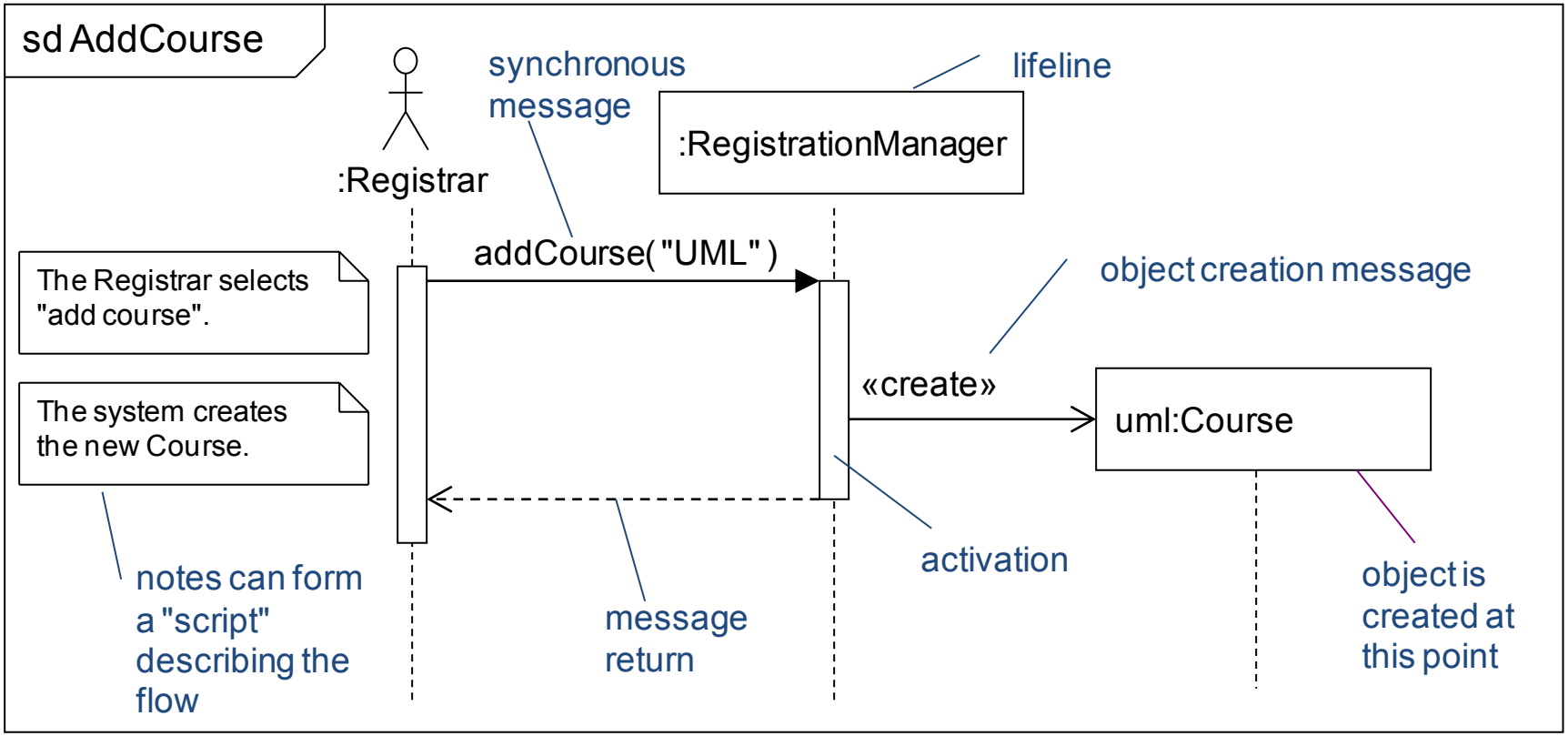
Use Case realization

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

Interaction diagrams

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

Sequence diagram syntax



- Interactions are captured via **lifelines** (participants in the interaction) and **messages** (communications between lifelines)
- Activations indicate when a lifeline has focus of control - they are often omitted from sequence diagrams

Lifelines

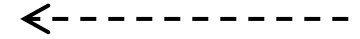
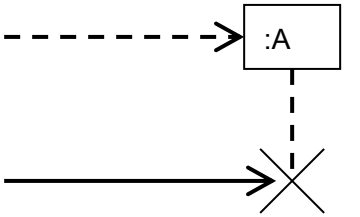
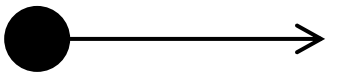
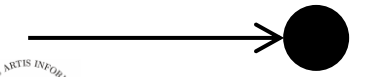
jimsAccount [id = "1234"] : Account

name selector type

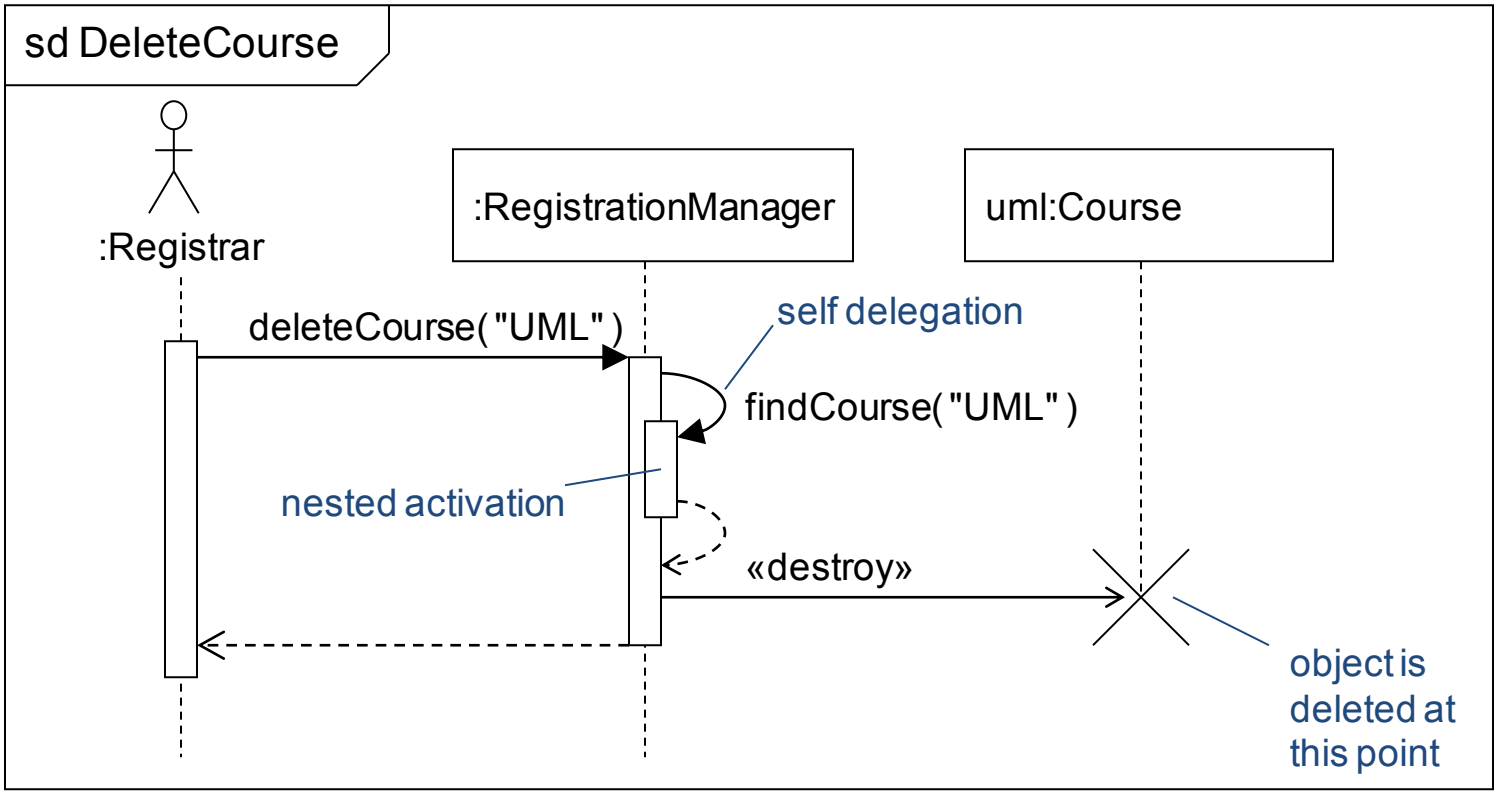
- A lifeline represents a single participant in an interaction
  - Shows how a classifier instance may participate in the interaction
- Lifelines have:
  - name - the name used to refer to the lifeline in the interaction
  - selector - a boolean condition that selects a specific instance
  - type - the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

Messages

- A message represents a communication between two lifelines

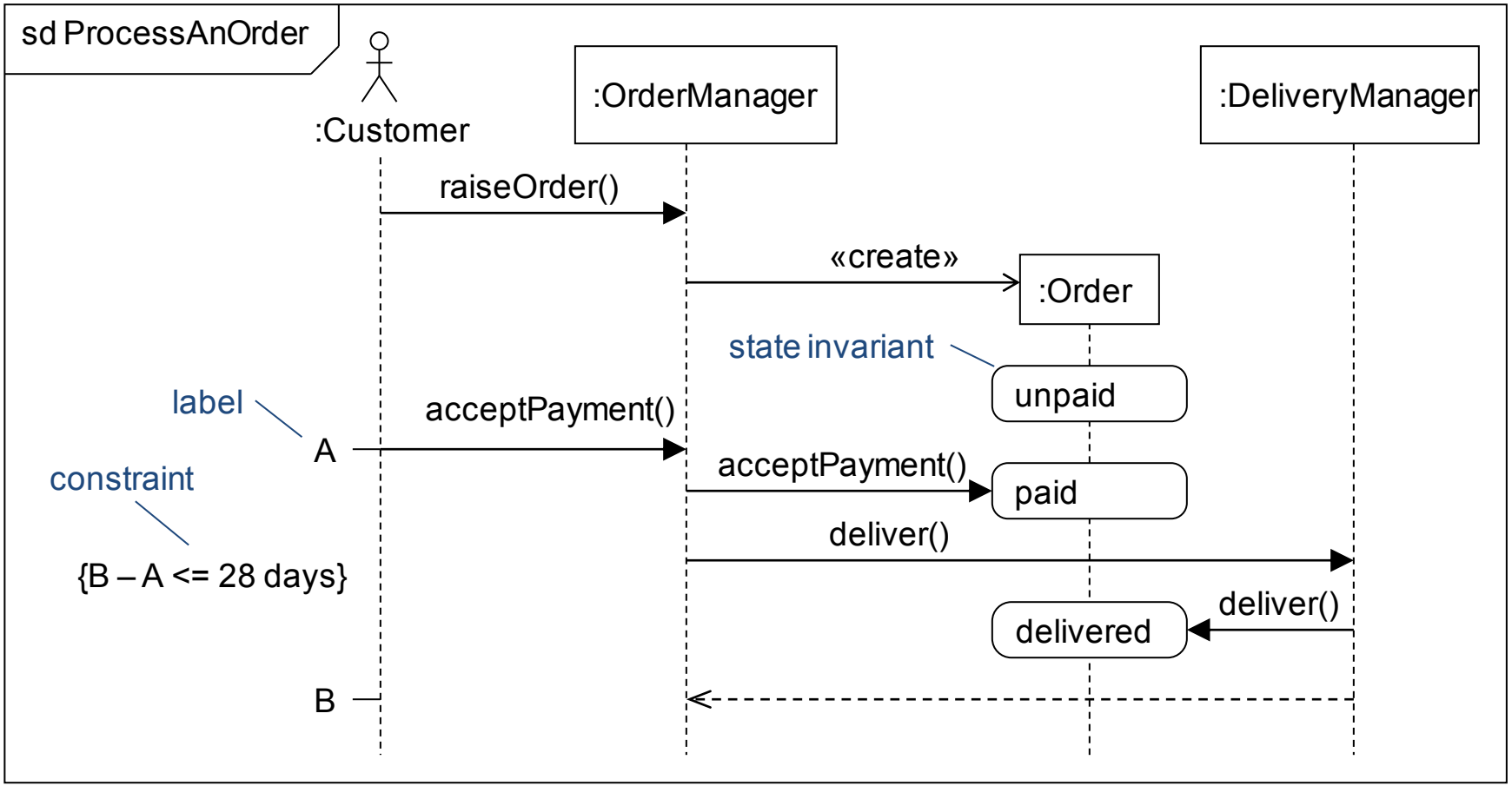
arrow type	type of message	semantics
	synchronous message	calling an operation synchronously the sender waits for the receiver to complete
	asynchronous send	calling an operation asynchronously, sending a signal the sender *does not* wait for the receiver to complete
	message return	returning from a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

Deletion and self-delegation



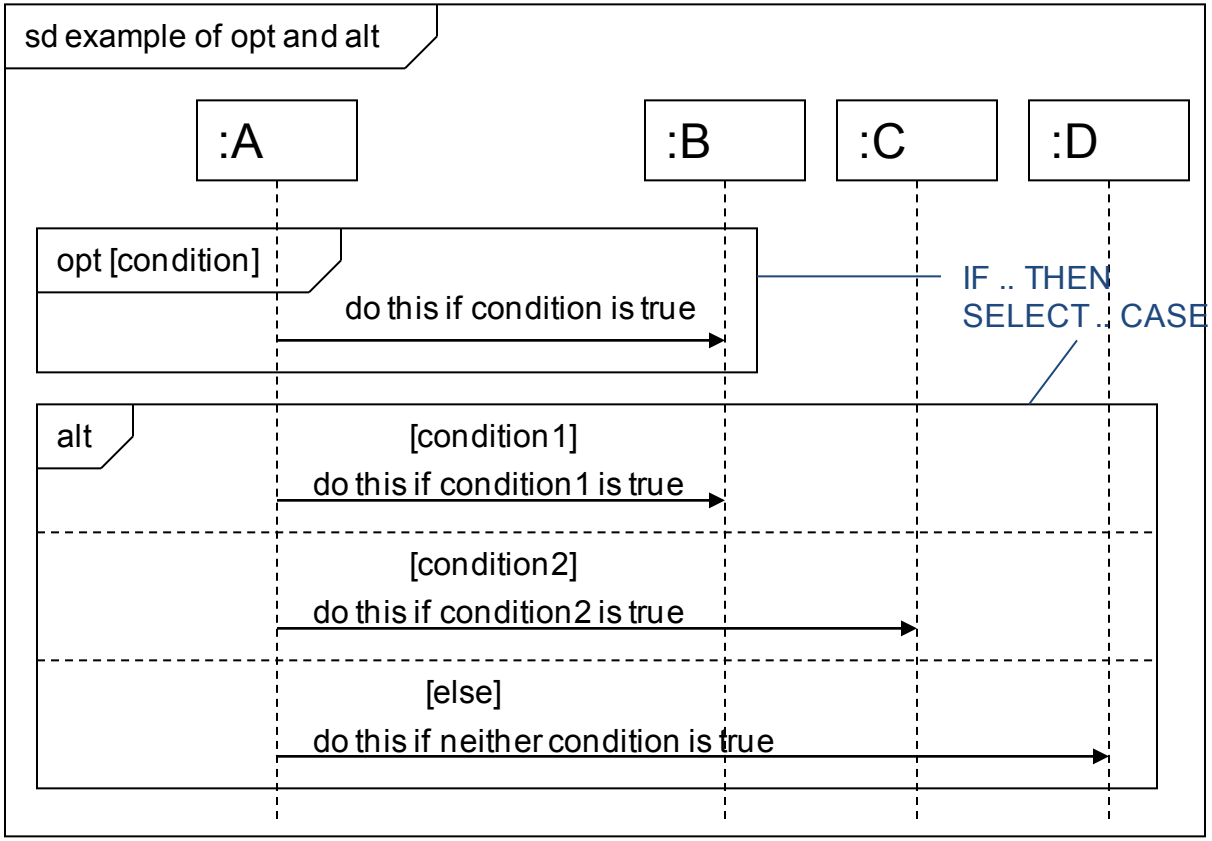
- Self delegation is when a lifeline sends a message to itself
  - Generates a nested activation

State invariants and constraints



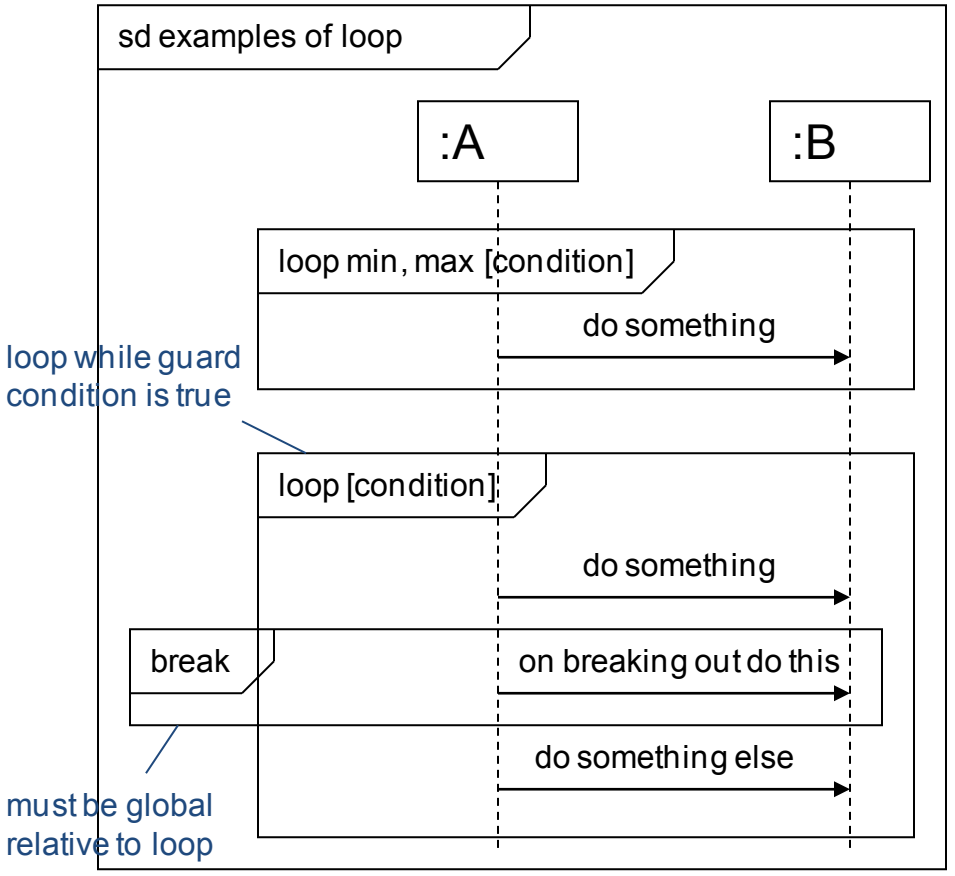
Combined fragments – opt and alt

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



Combined fragments – loop and break

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



Loop idioms

type of loop	semantics	loop expression
infinite loop	keep looping forever	loop *
for i = 1 to n {body}	repeat (n) times	loop n
while(booleanExpression) {body}	repeat while booleanExpression is true	loop [booleanExpression]

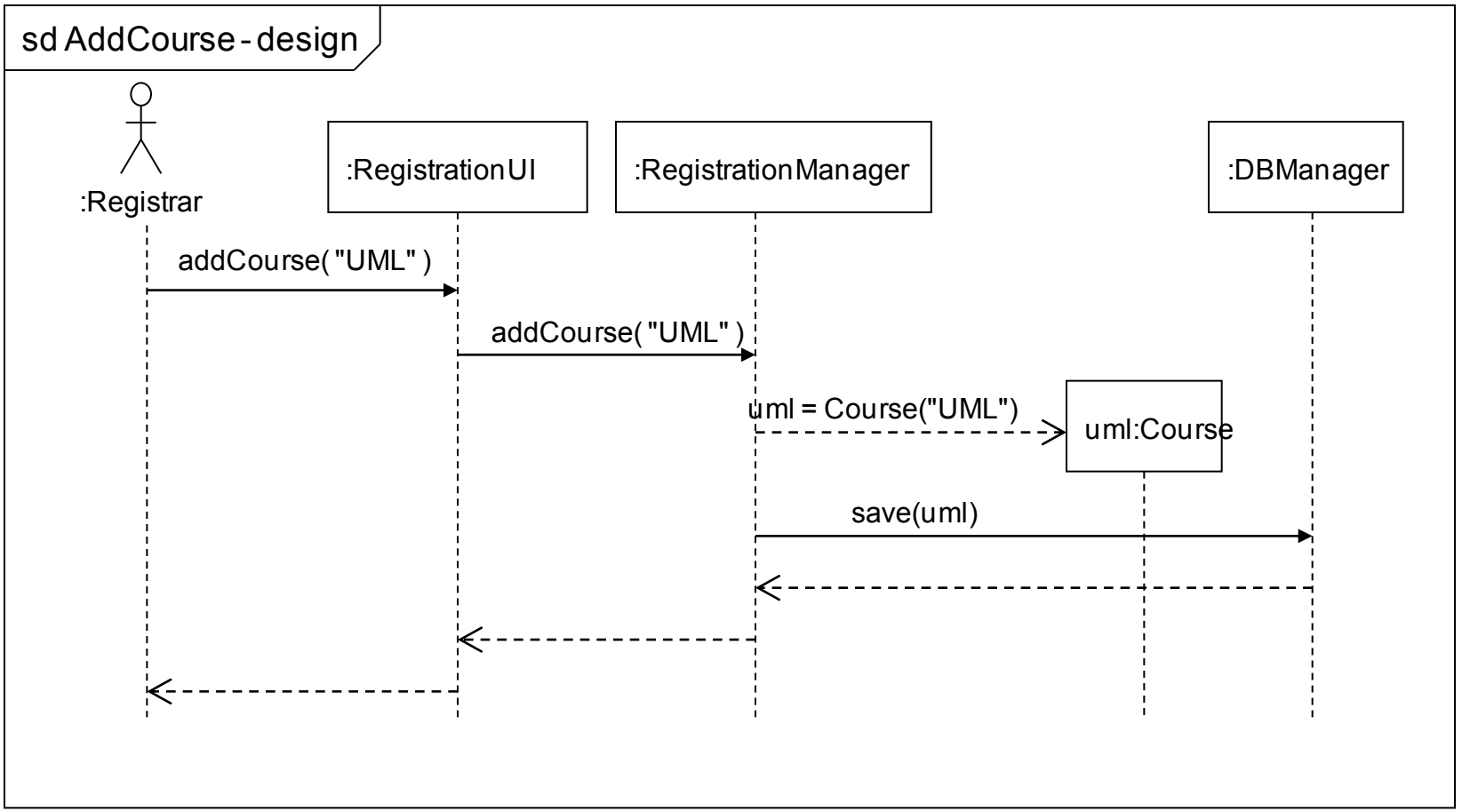
repeat
{body}
while(booleanExpression)
	execute once then repeat while booleanExpression is true	loop 1, * [booleanExpression]

forEach object in collection {body}	Execute the loop once for each object in a collection	loop [for each object in collection]
forEach object in ObjectType {body}	Execute the loop once for each object of a particular type	loop [for each object in :ObjectType]

The rest of the operators

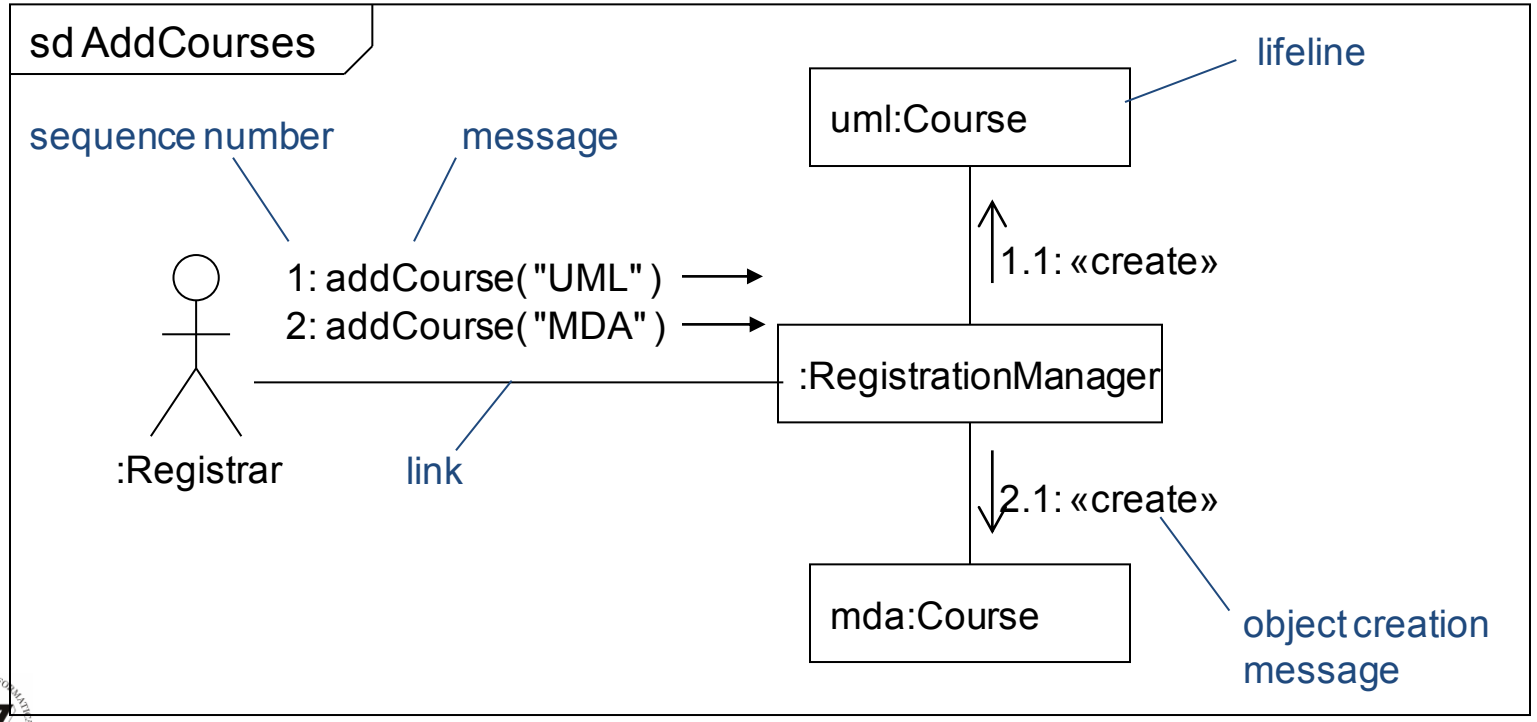
operator	long name	semantics
par	parallel	Both operands execute in parallel
seq	weak sequencing	The operands execute in parallel subject to the constraint that event occurrences on the *same* lifeline from *different* operands must happen in the same sequence as the operands
ref	reference	The combined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

Sequence diagrams in design



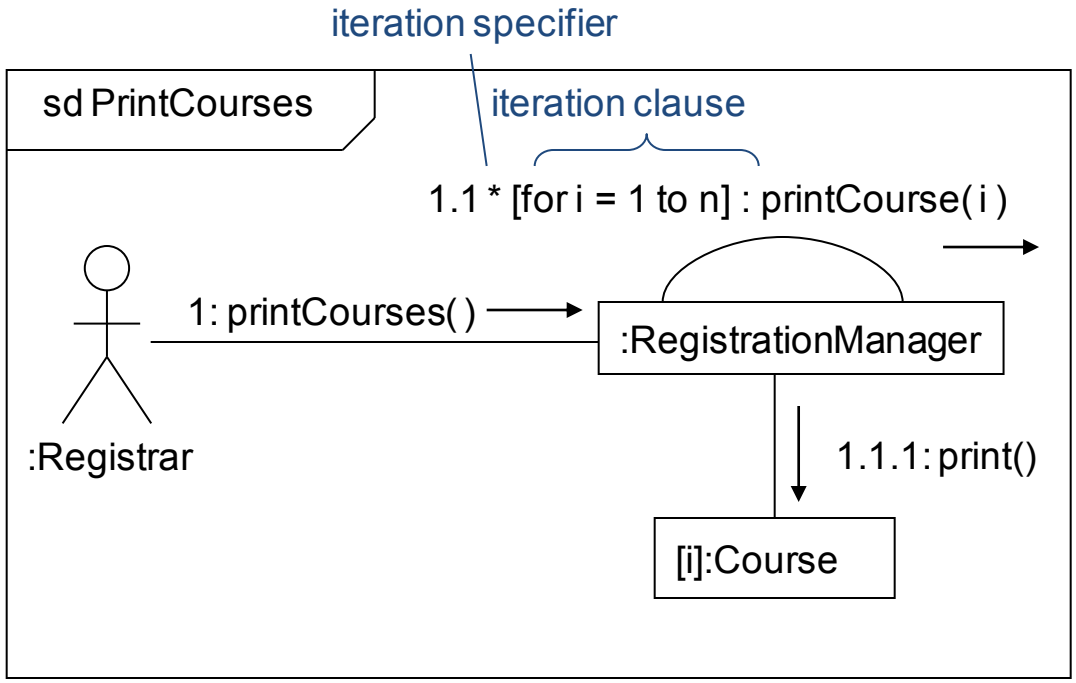
Communication diagram syntax

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

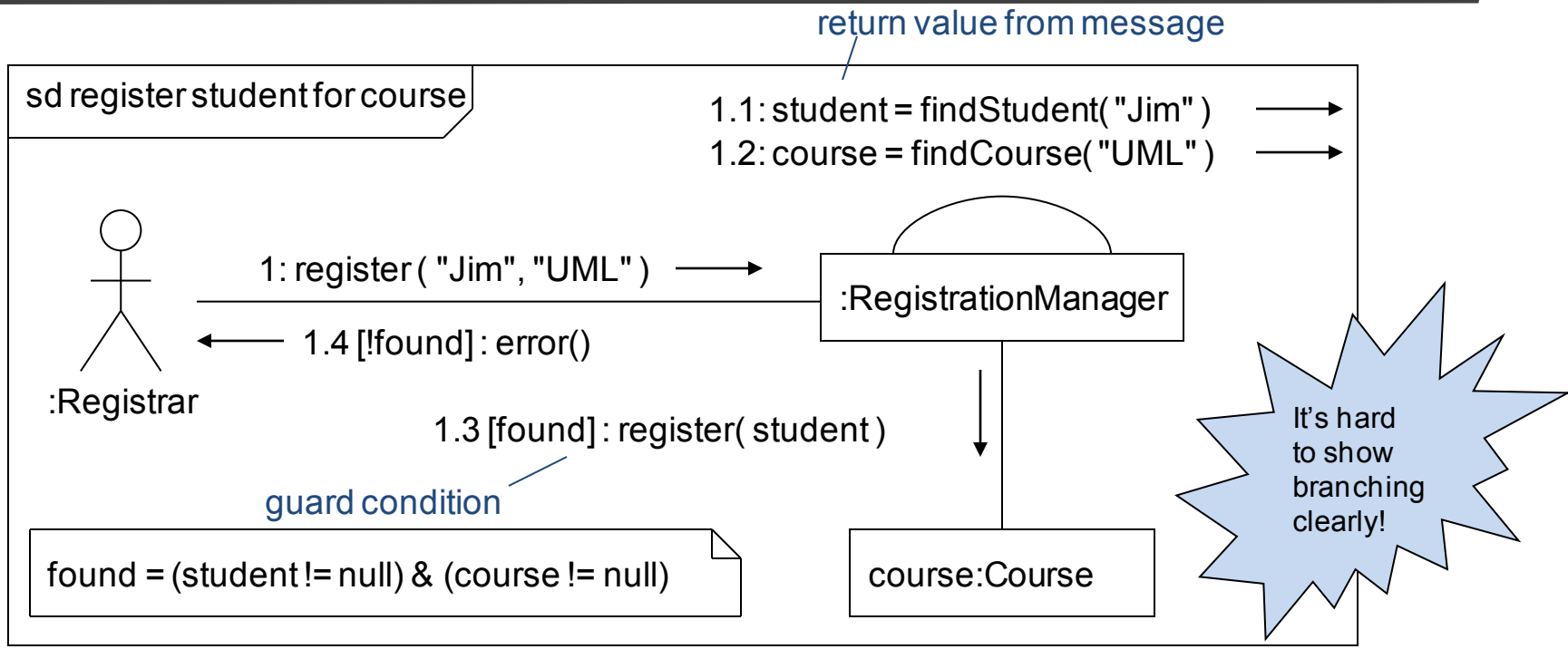


Iteration

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



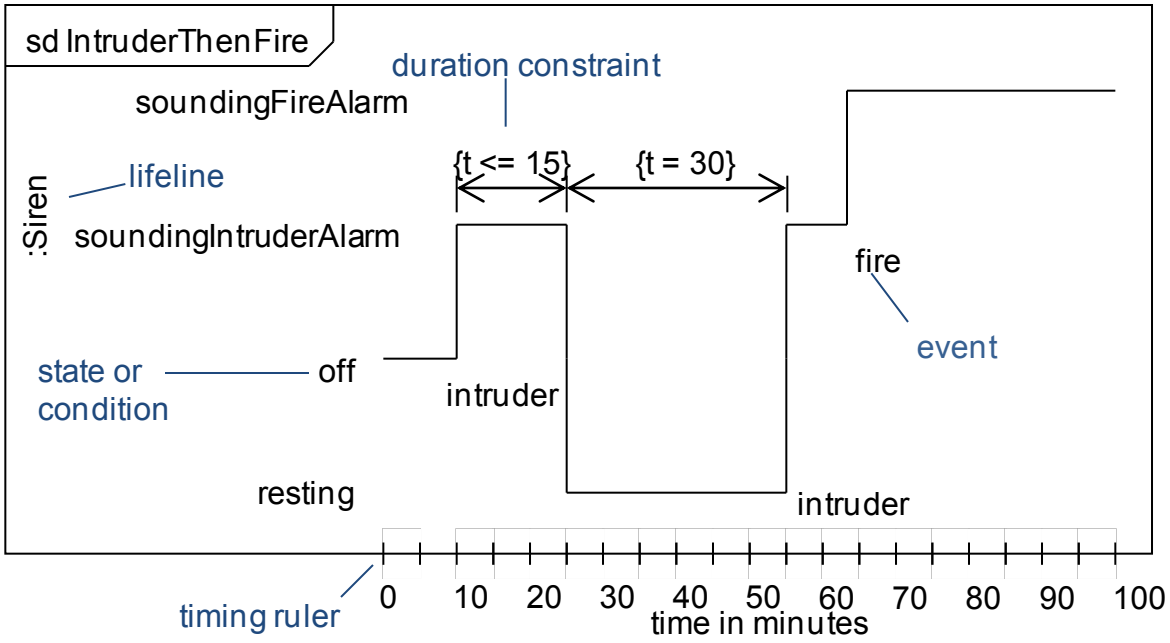
Branching



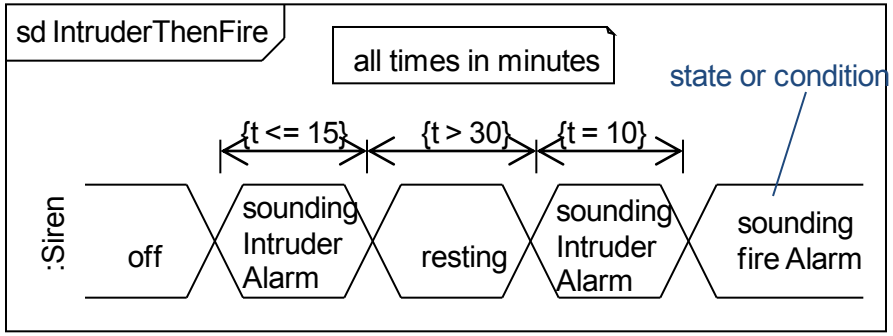
- Branching is modelled by prefixing the sequence number with a *guard condition*
  - There is no prescribed UML syntax for guard conditions
  - In the example above, we use the variable found. This is true if both the student and the course are found, otherwise it is false

Timing diagrams

- Emphasize the real-time aspects of an interaction
- Used to model timing constraints
- Lifelines, their states or conditions are drawn vertically, time horizontally
- It's important to state the time units you use in the timing diagram

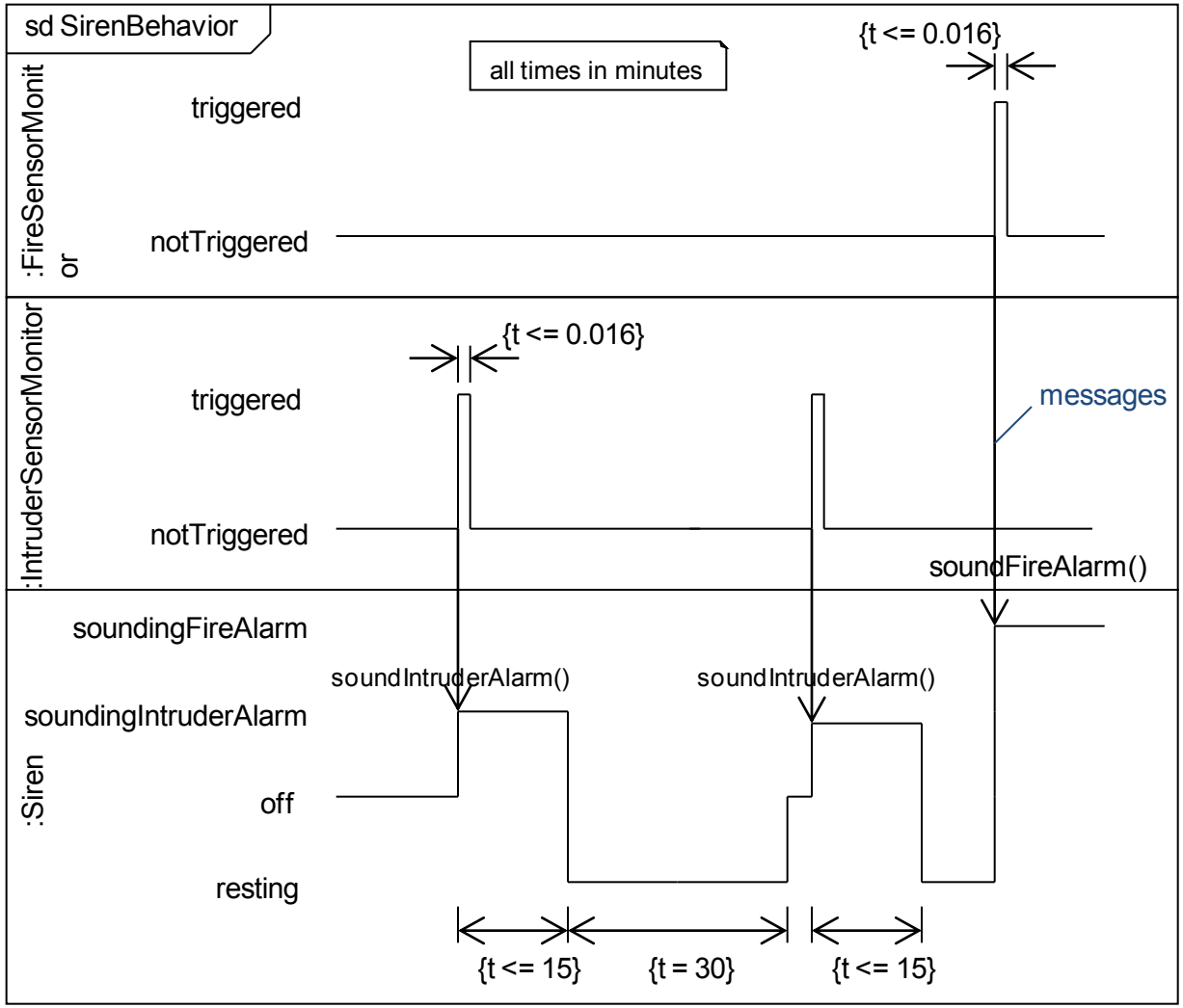


compact form



Messages on timing diagrams

- You can show messages between lifelines on timing diagrams
- Each lifeline has its own partition



Key points

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