



OO Analysis and Design with UML 2 and UP

Dr. Jim Arlow,
Zuhlke Engineering Limited



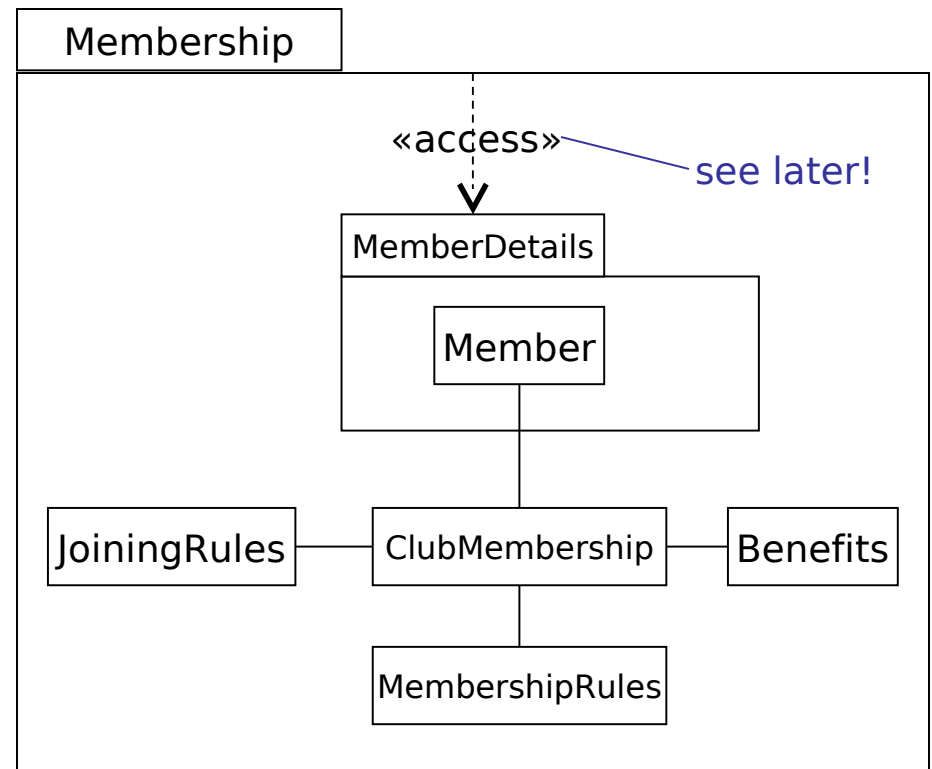
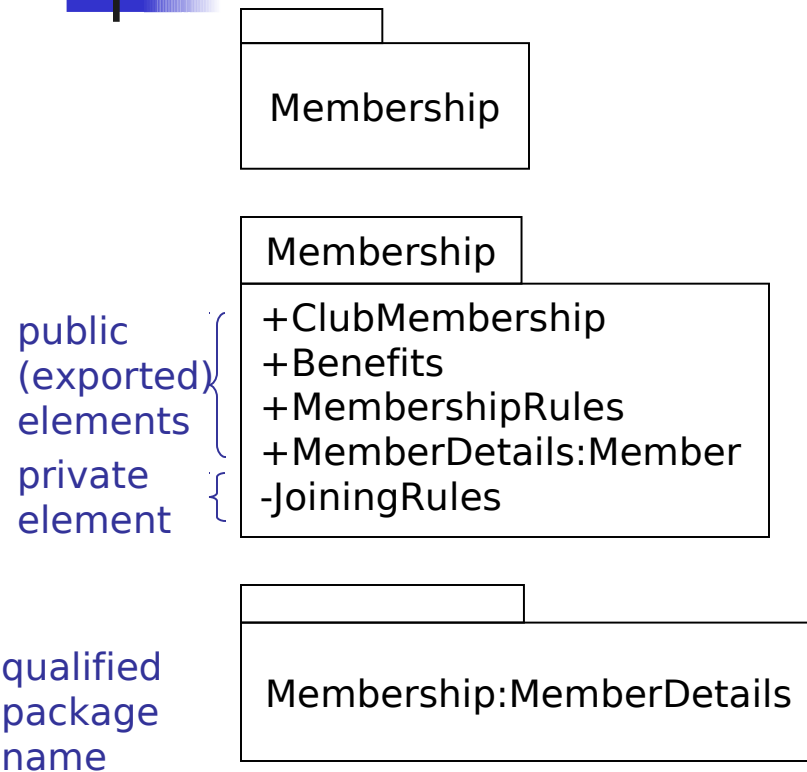
Analysis - packages



Analysis packages

- A package is a *general purpose* mechanism for organising model elements into groups
 - Group semantically related elements
 - Define a “semantic boundary” in the model
 - Provide units for parallel working and configuration management
 - Each package defines an *encapsulated namespace* i.e. all names must be unique within the package
- In UML 2 a package is a purely logical grouping mechanism
 - Use components for physical grouping
- *Every* model element is owned by exactly one package
 - A hierarchy rooted in a top level package that can be stereotyped «topLevel»
- Analysis packages contain:
 - Use cases, analysis classes, use case realizations, analysis packages

Package syntax

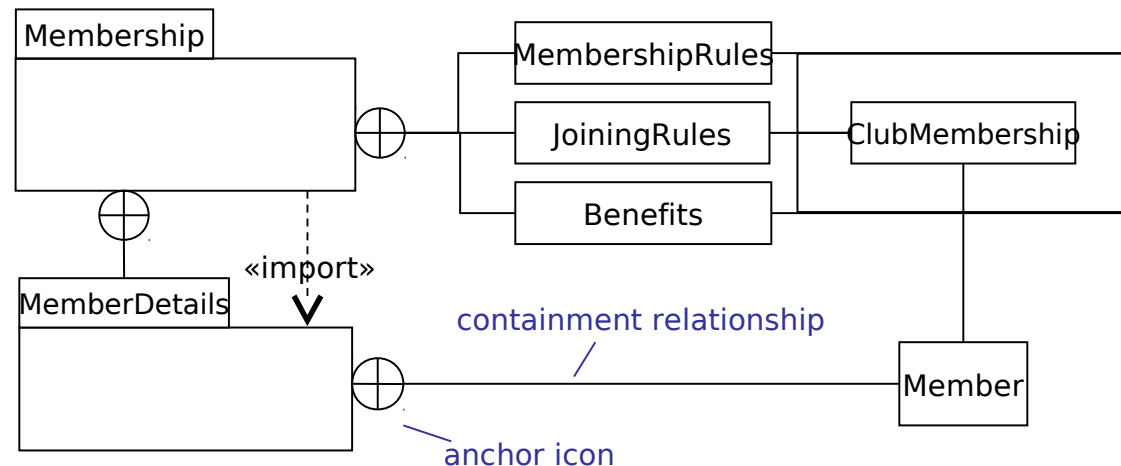
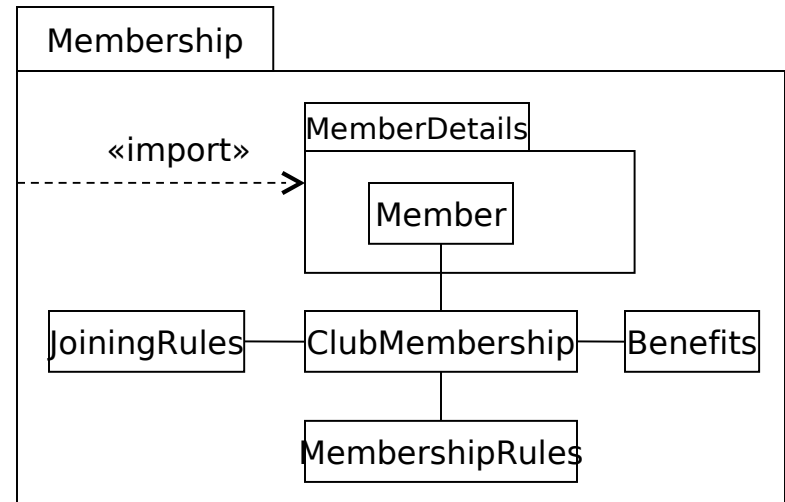


standard UML 2 package stereotypes

<code>«framework»</code>	A package that contains model elements that specify a reusable architecture
<code>«modelLibrary»</code>	A package that contains elements that are intended to be reused by other packages Analogous to a class library in Java, C# etc.

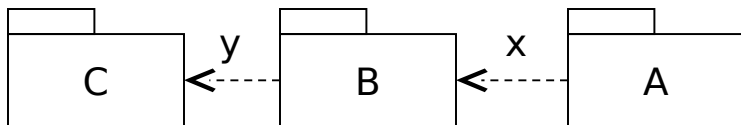
Nested packages

- If an element is visible within a package then it is visible within all nested packages
 - e.g. Benefits is visible within MemberDetails
- Show containment using nesting or the containment relationship
- Use «access» or «import» to merge the namespace of nested packages with the parent namespace



Package dependencies

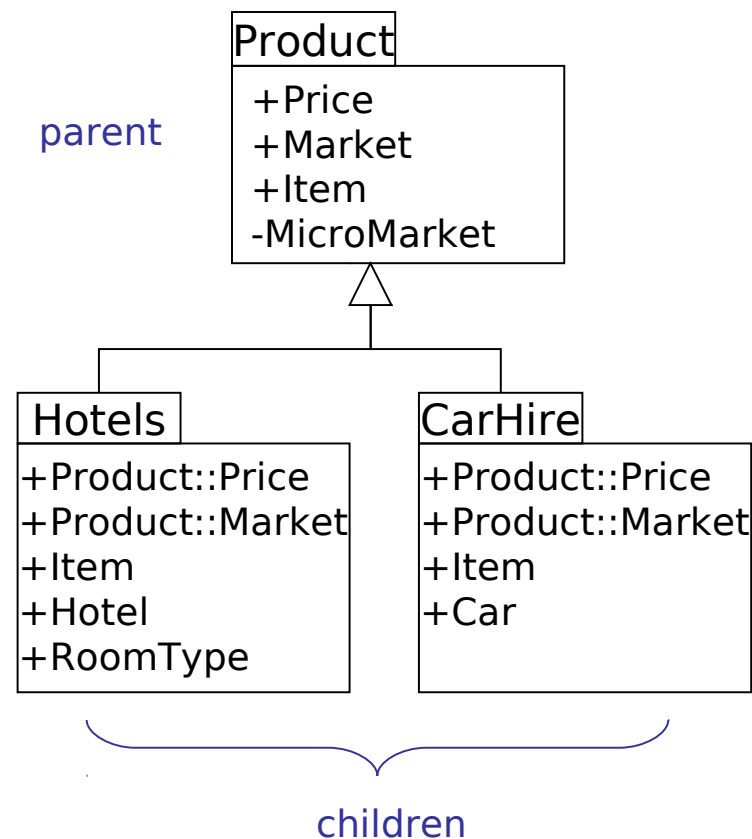
dependency	semantics
<p>Supplier ← «use» Client</p>	An element in the client uses an element in the supplier in some way. The client depends on the supplier. Transitive.
<p>Supplier ← «import» Client</p>	Public elements of the supplier namespace are added as public elements to the client namespace. Transitive.
<p>Supplier ← «access» Client not transitive</p>	Public elements of the supplier namespace are added as private elements to the client namespace. Not transitive.
<p>Analysis Model ← «trace» Design Model</p>	«trace» usually represents an historical development of one element into another more refined version. It is an extra-model relationship. Transitive.
<p>Supplier ← «merge» Client</p>	The client package merges the public contents of its supplier packages. This is a complex relationship only used for metamodeling - you can ignore it.



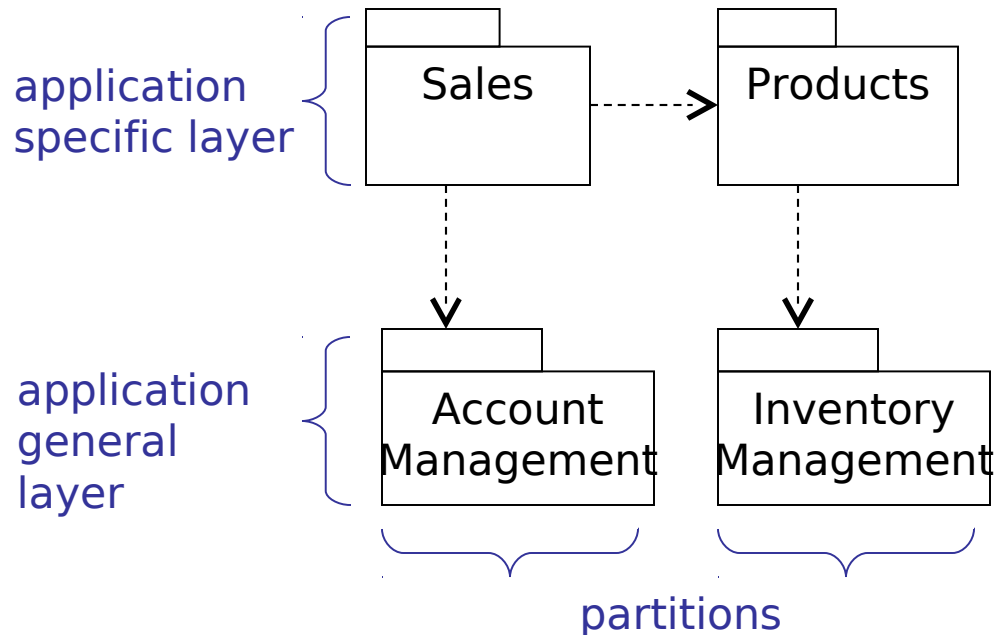
transitivity - if dependencies x and y are transitive, there is an *implicit* dependency between A and C

Package generalisation

- The more specialised child packages *inherit* the public and protected elements in their parent package
- Child packages may *override* elements in the parent package. Both Hotels and CarHire packages override Product::Item
- Child packages may *add* new elements. Hotels adds Hotel and RoomType, CarHire adds Car



Architectural analysis



- This involves organising the analysis classes into a set of cohesive packages
- The architecture should be *layered* and *partitioned* to separate concerns
 - It's useful to layer analysis models into application specific and application general layers
- Coupling between packages should be minimised
- Each package should have the minimum number of public or protected elements

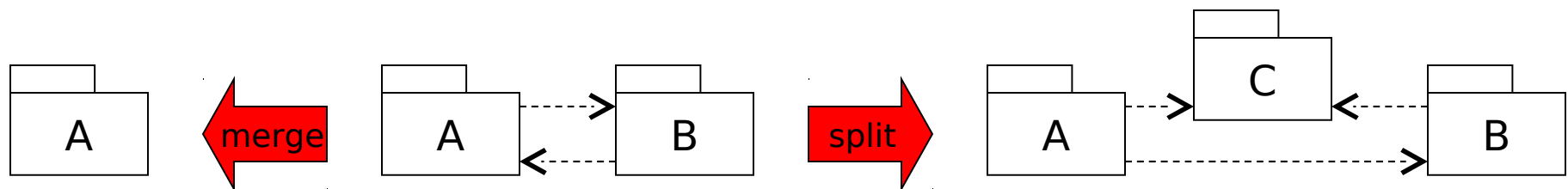


Finding analysis packages

- These are often discovered as the model matures
- We can use the natural groupings in the use case model to help identify analysis packages:
 - One or more use cases that support a particular business process or actor
 - Related use cases
- Analysis classes that realise these groupings will often be part of the same analysis package
- Be careful, as it is common for use cases to *cut across* analysis packages!
 - One class may realise several use cases that are allocated to different packages

Analysis packages: guidelines

- A cohesive group of closely related classes or a class hierarchy and supporting classes
- Minimise dependencies between packages
- Localise business processes in packages where possible
- Minimise nesting of packages
- Don't worry about dependency stereotypes
- Don't worry about package generalisation
- Refine package structure as analysis progresses
- 4 to 10 classes per package
- Avoid cyclic dependencies!





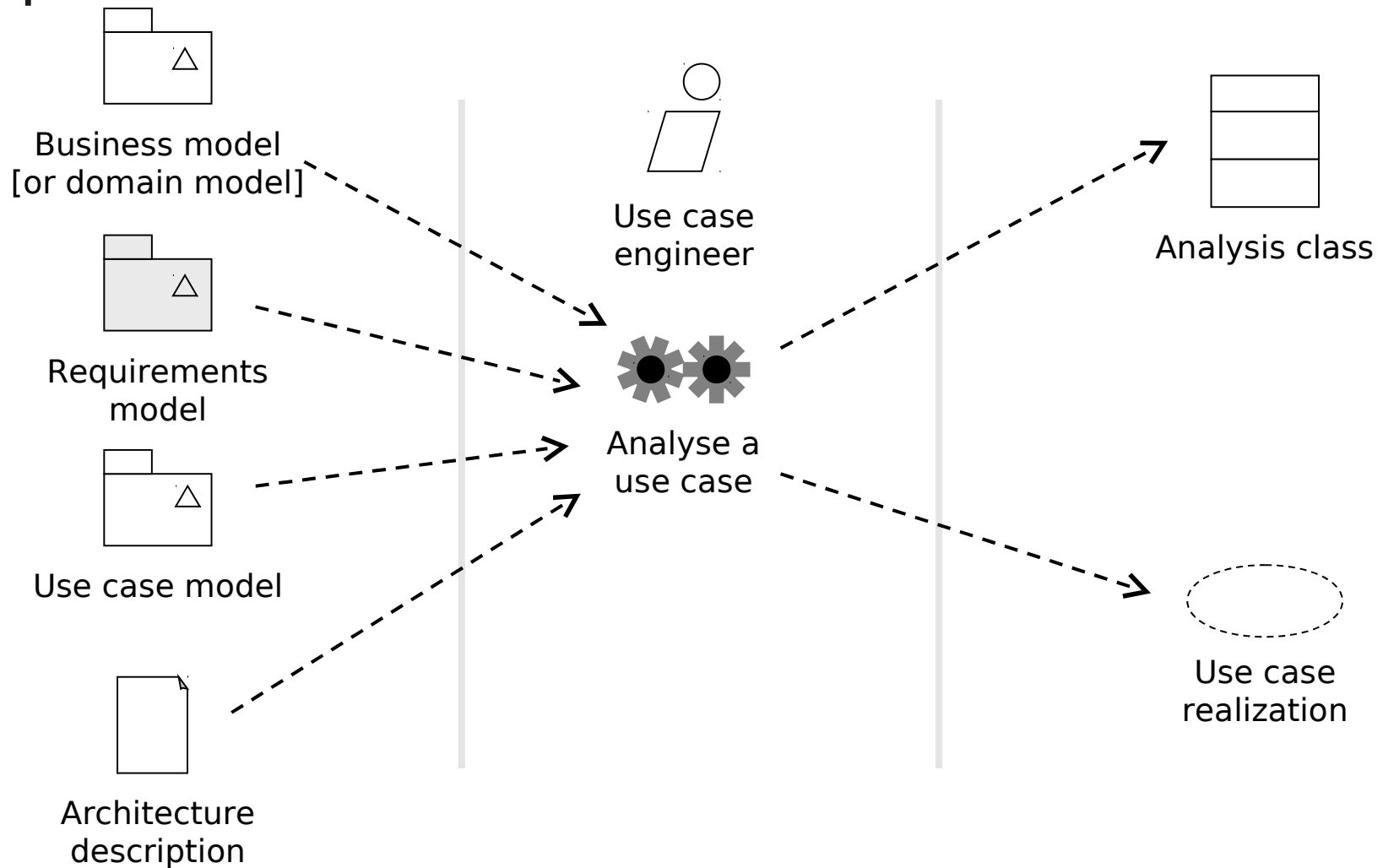
Summary

- Packages are the UML way of grouping modeling elements
- There are dependency and generalisation relationships between packages
- The package structure of the analysis model defines the logical system architecture

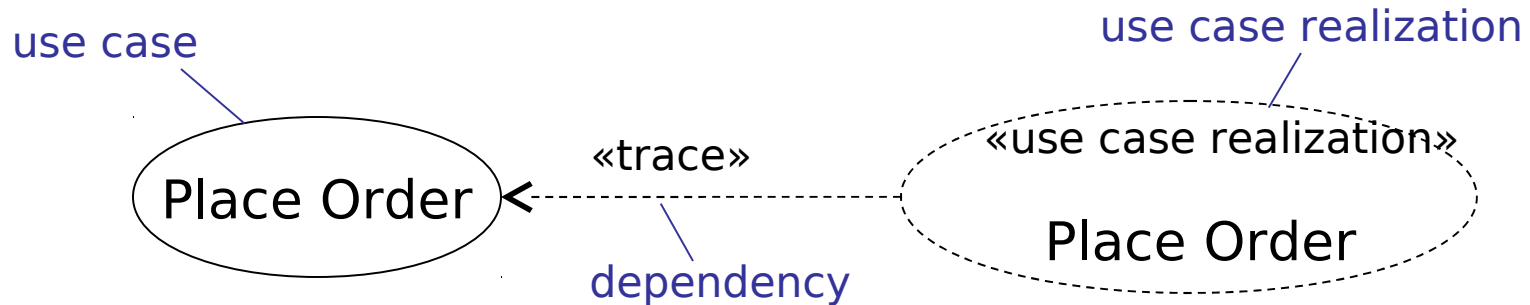
Analysis - use case realization



Analyse a use case



What are use case realizations?



- Each use case has exactly one *use case realization*
 - parts of the model that show how analysis classes collaborate together to realise the behaviour specified by the use case
 - they model *how* the use case is realised by the analysis classes we have identified
- They are rarely modelled explicitly
 - they form an implicit part of the backplane of the model
 - they can be drawn as a stereotyped collaboration

UC realization - elements

- 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



Interactions

- Interactions are units of behavior of a context classifier
- In use case realization, the context classifier is a use case
 - The interaction shows how the behavior specified by the use case is realized by instances of classifiers
- Interaction diagrams capture an interaction as:
 - Lifelines - participants in the interaction
 - Messages - communications between lifelines

Lifelines

```
jimsAccount [ id = "1234" ] : Account
```

name


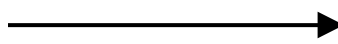

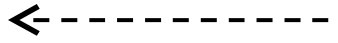

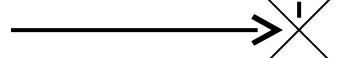
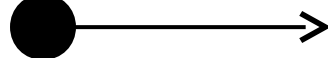
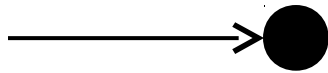
selector

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
 - The lifeline jimsAccount represents an instance of the Account class
 - The selector [id = "1234"] selects a specific Account instance with the id "1234"

Messages

- A message represents a communication between two lifelines

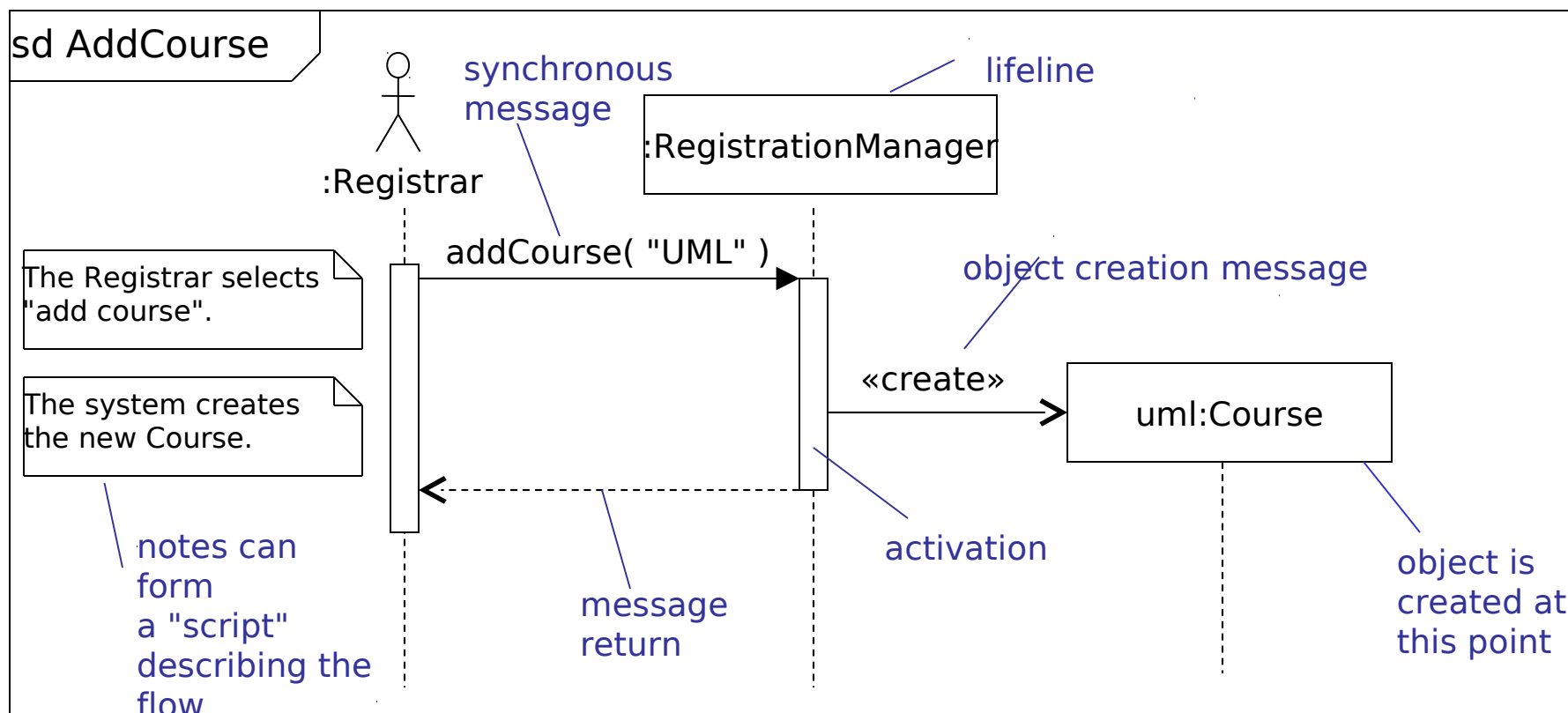
sender  receiver/target	type of message	semantics
	synchronous message	calling an operation synchronously the sender waits for the receiver to complete
	asynchronous send	calling an operation asynchronously, sending a signal the sender <i>does not</i> wait for the receiver to complete
	message return	returning from a synchronous operation call the receiver returns focus of control to the sender
	creation	the sender creates the target
	destruction	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



Interaction diagrams

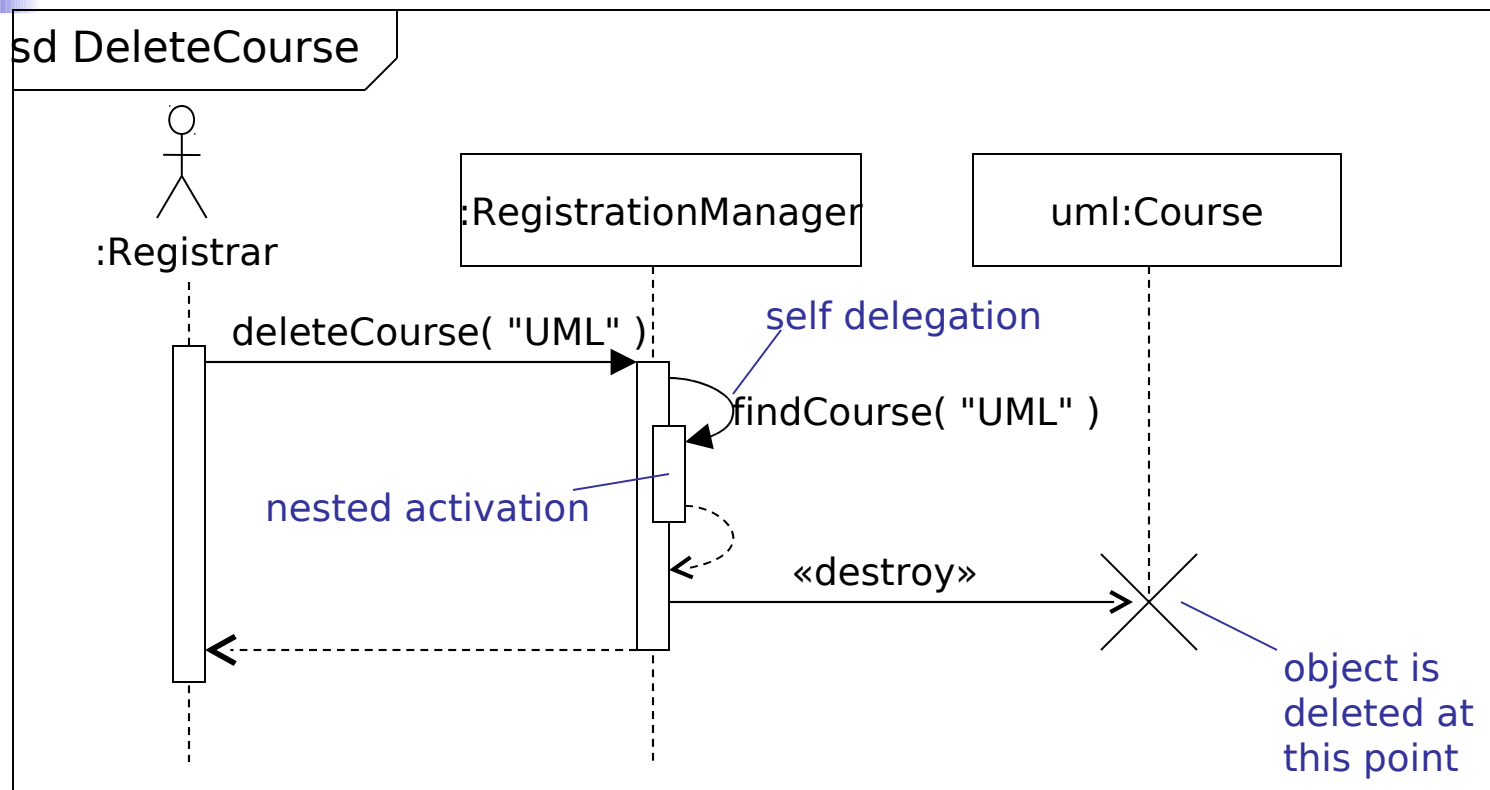
- Sequence diagrams
 - Emphasize time-ordered sequence of message sends
 - Show interactions arranged in a time sequence
 - Are the richest and most expressive interaction diagram
 - Do not show object relationships explicitly - these can be inferred from message sends
- Communication diagrams
 - Emphasize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

Sequence diagram syntax



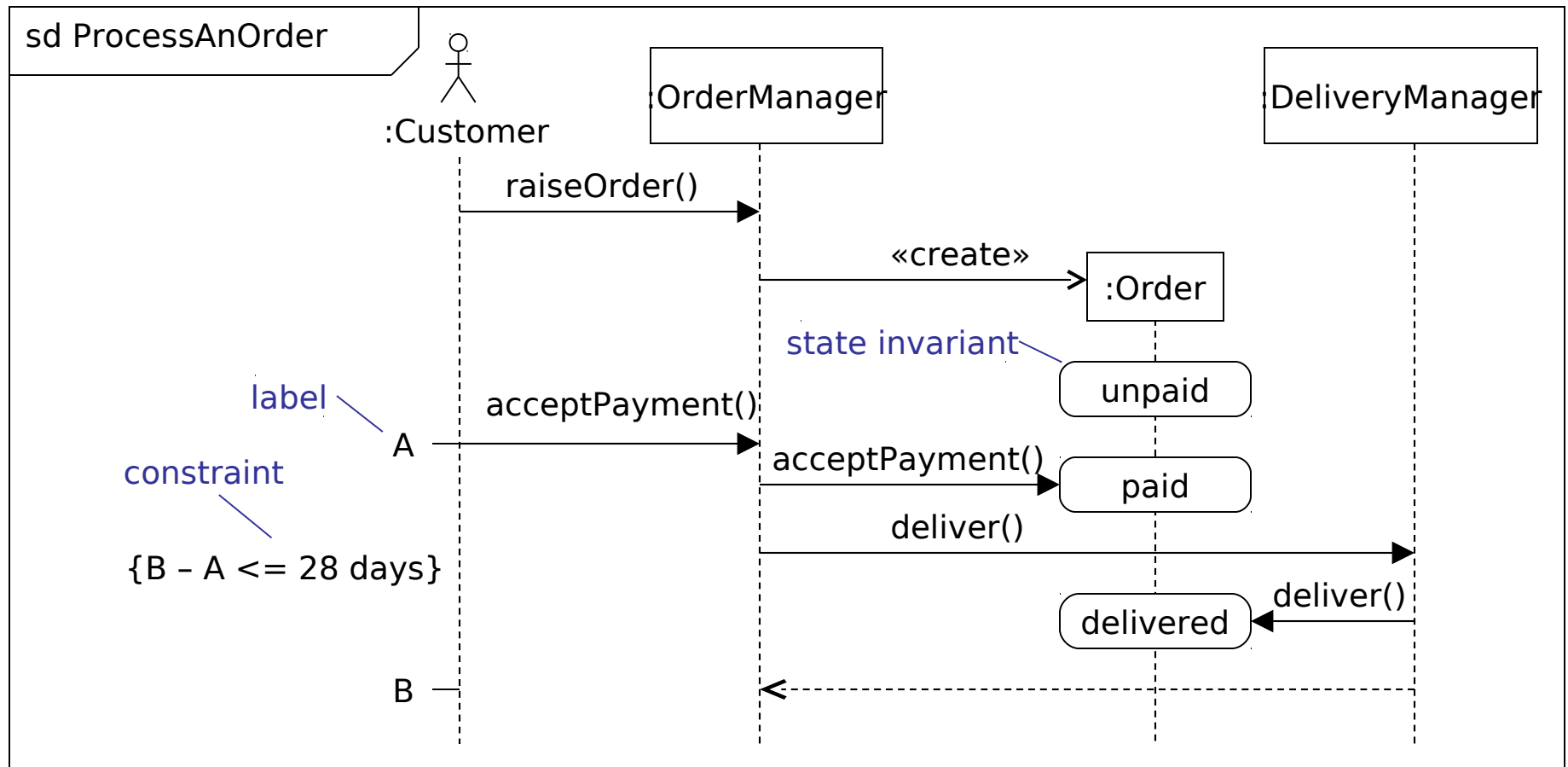
- All interaction diagrams may be prefixed **sd** to indicate their type
 - You can generally infer diagram types from diagram syntax
- Activations indicate when a lifeline has focus of control - they are often omitted from sequence diagrams

Deletion and self-delegation

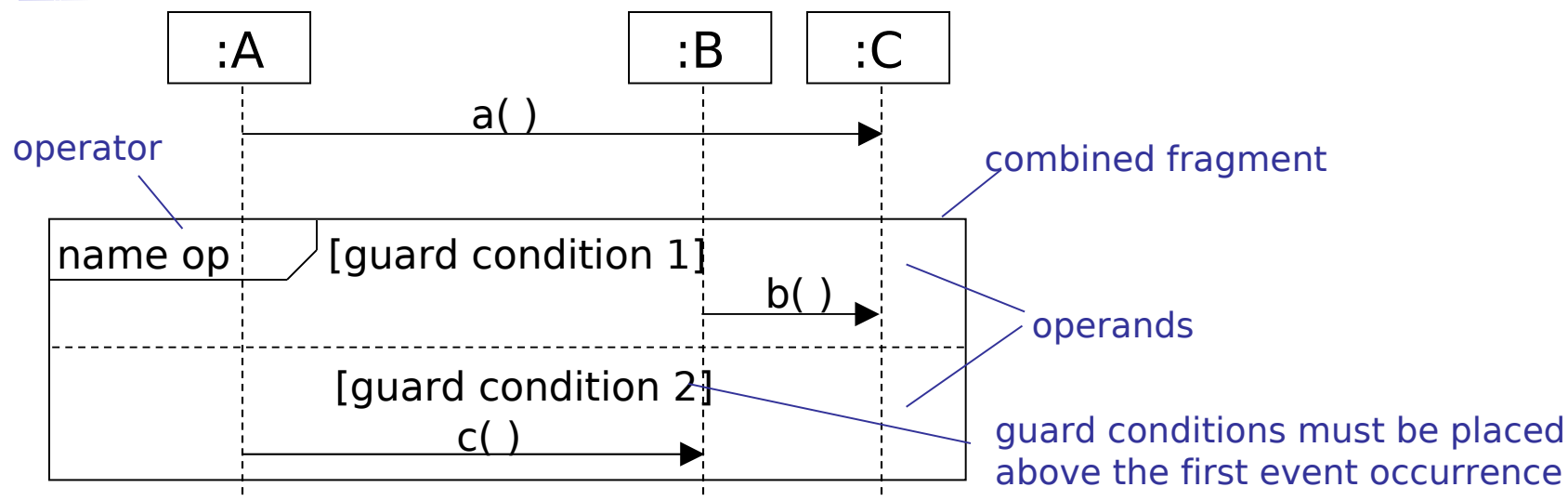


- Self delegation is when a lifeline sends a message to itself
 - Generates a nested activation
- Object deletion is shown by terminating the lifeline's tail at the point of deletion by a large X

State invariants and constraints



Combined fragments



- Sequence diagrams may be divided into areas called *combined fragments*
- Combined fragments have one or more *operands*
- *Operators* determine **how** the operands are executed
- *Guard conditions* determine **whether** operands execute. Execution occurs if the guard condition evaluates to true
 - A single condition may apply to all operands OR
 - Each operand may be protected by its own condition

Common operators

operator	long name	semantics
opt	Option	There is a single operand that executes if the condition is true (like if ... then)
alt	Alternatives	The operand whose condition is true is executed. The keyword else may be used in place of a Boolean expression (like select... case)
loop	Loop	This has a special syntax: loop min, max [condition] Iterate min times and then up to max times while condition is true
break	Break	The combined fragment is executed rather than the rest of the enclosing interaction
ref	Reference	The combined fragment refers to another interaction

ref

```
findStudent(name):Student
```

ref has a single operand that is a reference to another interaction.

This is an *interaction use*.



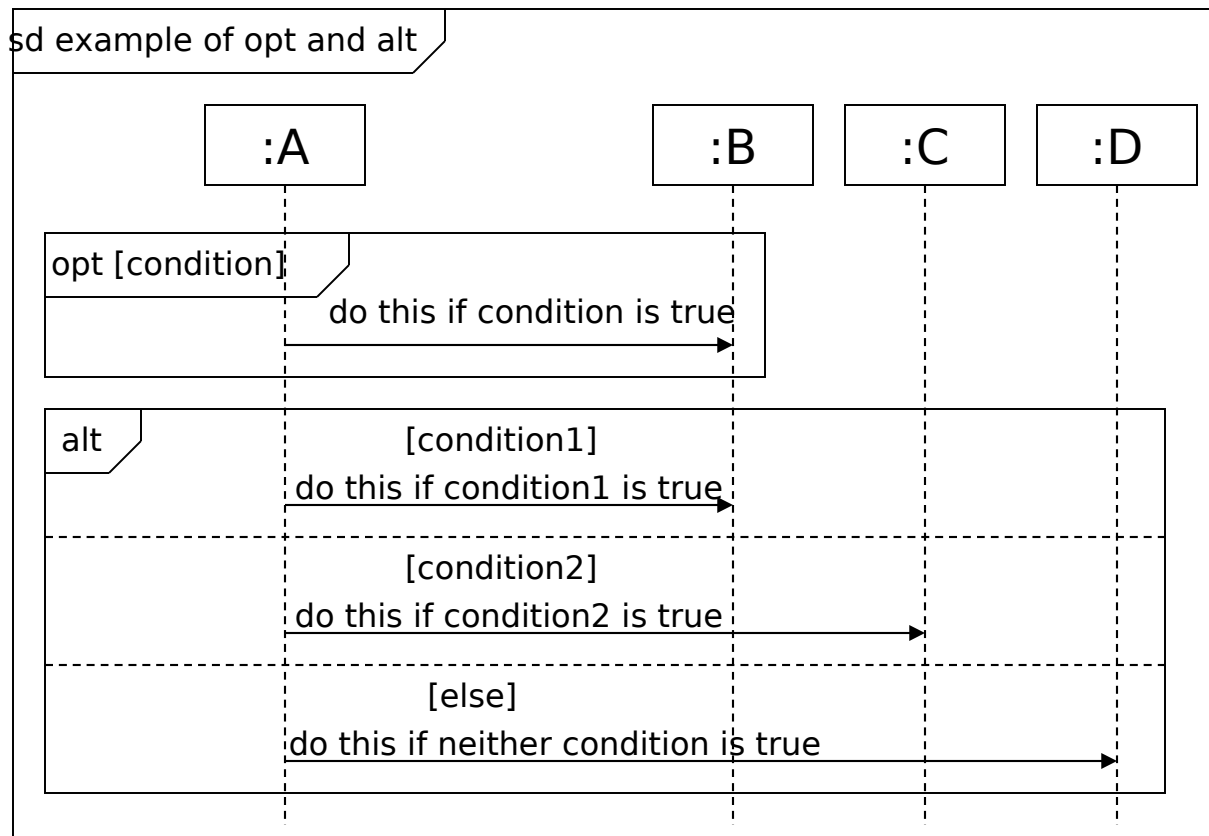
The rest of the operators

- These operators are less common

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 <i>same</i> lifeline from <i>different</i> operands must happen in the same sequence as the operands
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 message types are intentionally ignored in the interaction
consider	consider	Lists the message types that are considered in the interaction
assert	assertion	The operands of the combined fragments are the only valid continuations of the interaction

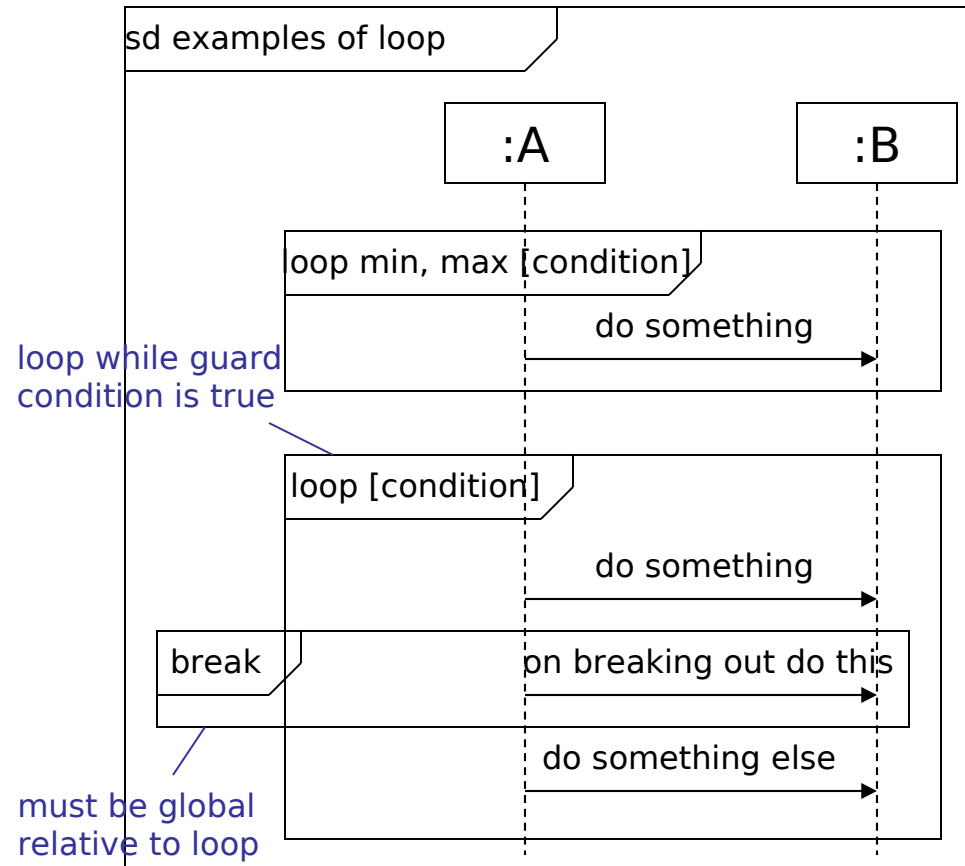
branching with opt and alt

- 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



Iteration with 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
 - If only min is specified then max = min
 - 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





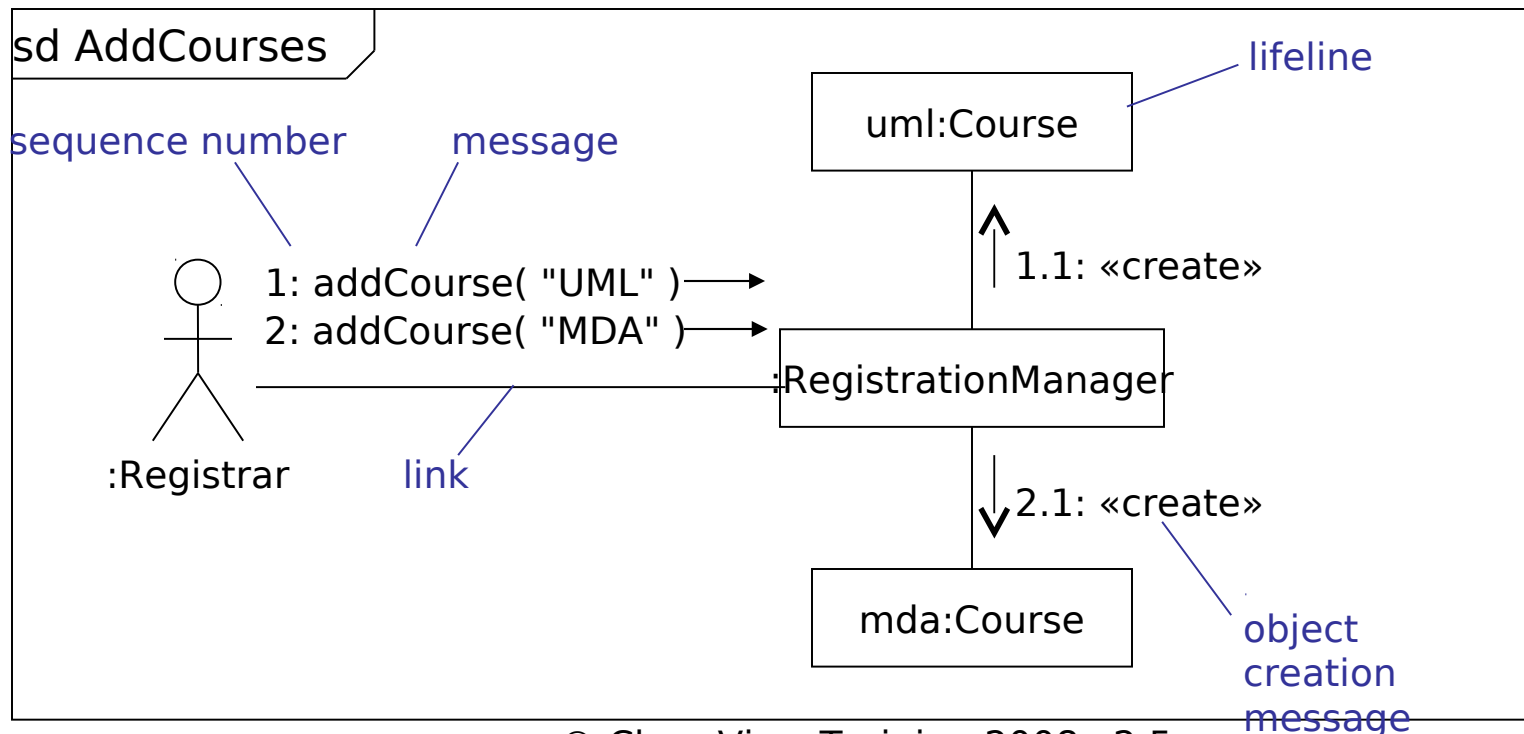
loop idioms

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 n) {body}	repeat while booleanExpression is true	loop [booleanExpression]
repeat {body} while(booleanExpression n)	execute once then repeat while booleanExpression is true	loop 1, * [booleanExpression]
forEach object in set {body}	Execute the loop once for each object in a set	loop [for each object in objectType]

- To specify a forEach loop over a set of objects:
 - use a for loop with an index (see later)
 - use the idiom [for each object in ObjectType](e.g. [for each student in :Student])

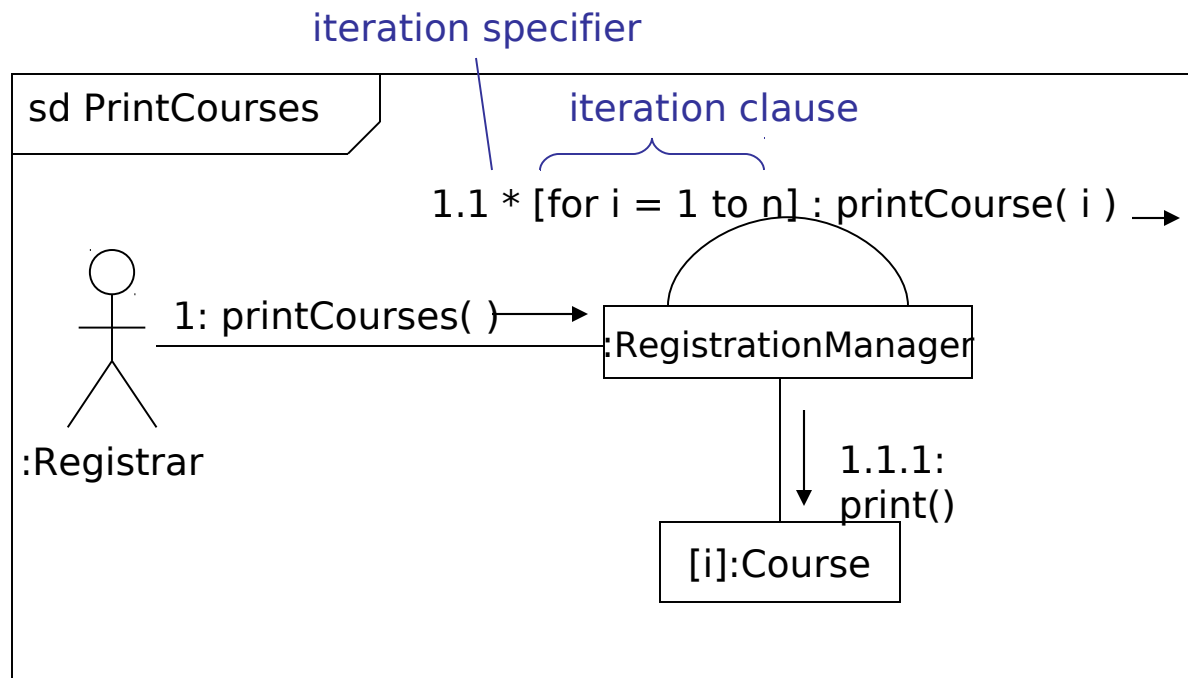
Communication diagram syntax

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

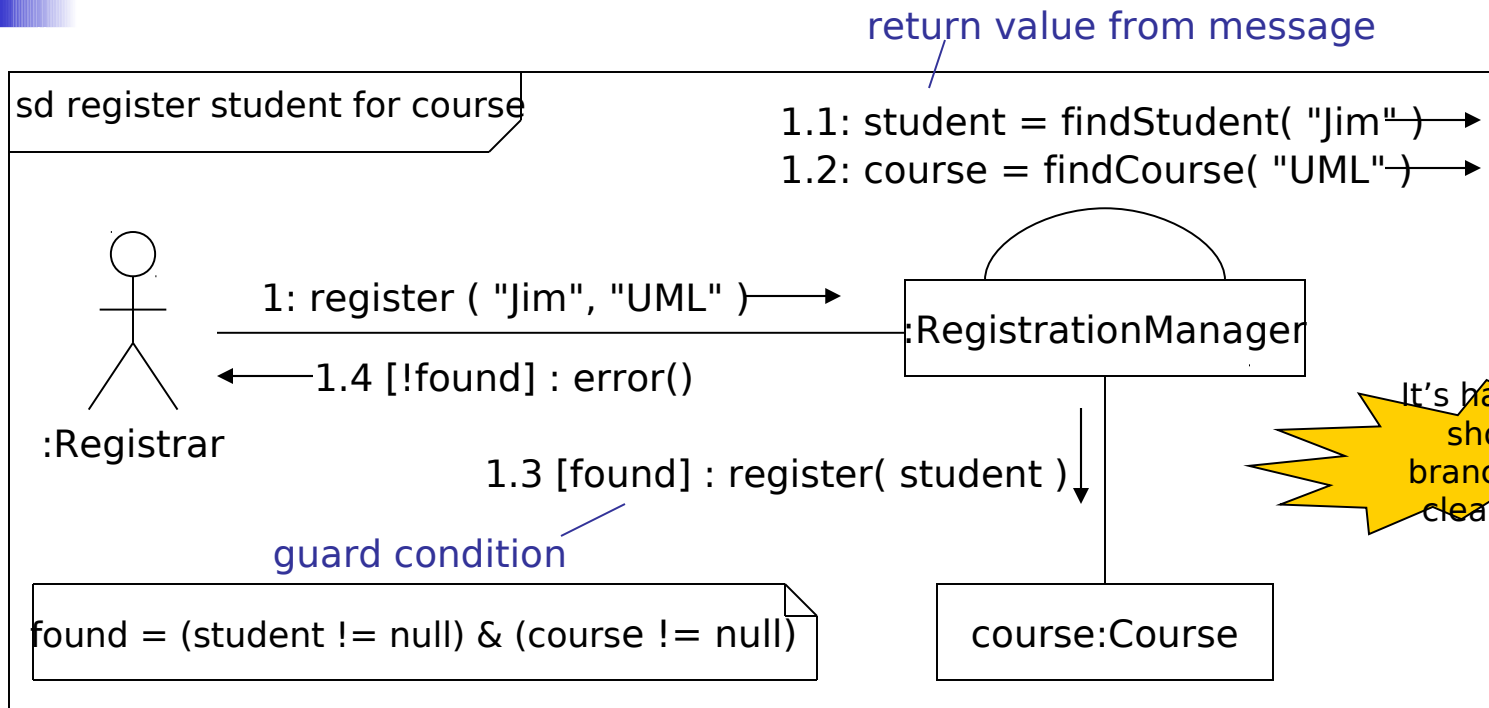


Iteration

- 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



Summary

- 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
 - Interaction overview diagrams - show how complex behavior is realized by a set of simpler interactions
 - Timing diagrams - emphasize the real-time aspects of an interaction
- We have looked at sequence diagrams and communication diagrams in this section - we will look at the other types of diagram later