Module 7

Interaction Diagrams

Vasco Amaral
vma@fct.unl.pt

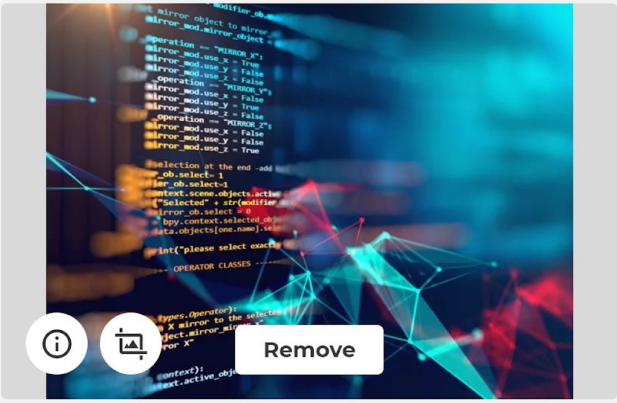
Quiz time

Fred Brooks em “No Silver Bullet" menciona balas de prata para se referir a:

30 sec

Points

1000



- à única solução para a complexidade do desenvolvimento de SW
- ao peso da complexidade do software actual sobre o programador
- não menciona nem balas de prata nem lobisomens
- à ausência de um único modo de atacar complexidade no desenvolvimento de SW

Fred Brooks em “No Silver Bullet" menciona balas de prata para se referir a:

30 sec

Points

1000

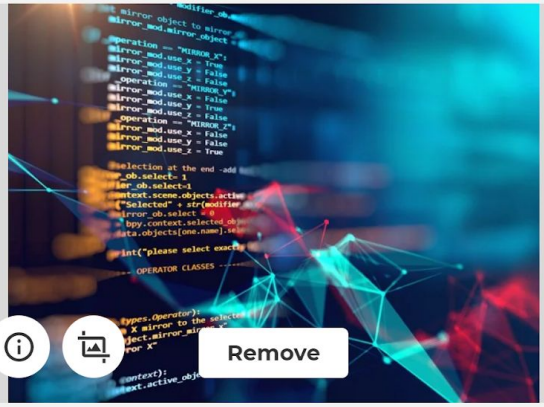
- à única solução para a complexidade do desenvolvimento de SW
- ao peso da complexidade do software actual sobre o programador
- não menciona nem balas de prata nem lobisomens
- à ausência de um único modo de atacar complexidade no desenvolvimento de SW

O que contribui para o sucesso de projectos de Engenharia de Software?

30 sec

Points

1000



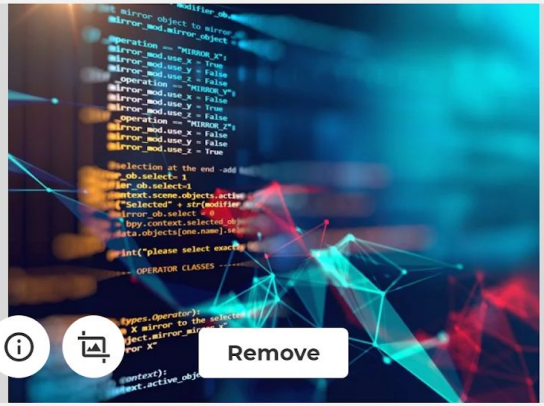
- metodologias e ferramentas como IDEs de desenvolvimento
- Falha no entendimento dos requisitos
- Falta de eficiência na comunicação e tamanh dos projectos
- Alta taxa de rotatividade do pessoal nos projectos

O que contribui para o sucesso de projectos de Engenharia de Software?

30 sec

Points

1000



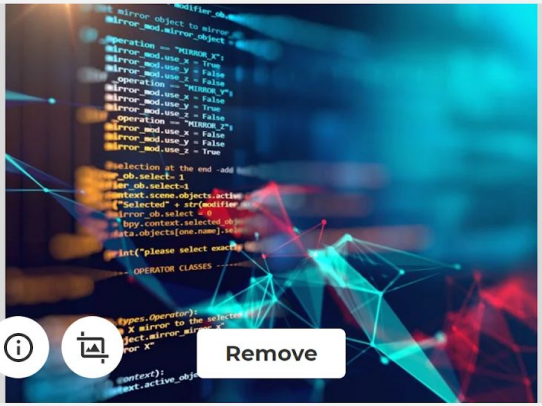
- metodologias e ferramentas como IDEs de desenvolvimento
- Falha no entendimento dos requisitos
- Falta de eficiência na comunicação e tamanho dos projectos
- Alta taxa de rotatividade do pessoal nos projectos

O que não é parte do Software?

30 sec

Points

1000



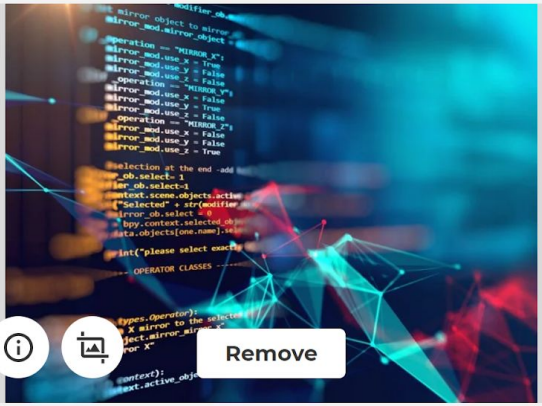
- Código, especificação, configuração manuais de sistema e utilização
- Bibliotecas
- Relatório de todo o passado na empresa
- Websites de suporte e manutenção

O que não é parte do Software?

30 sec

Points

1000



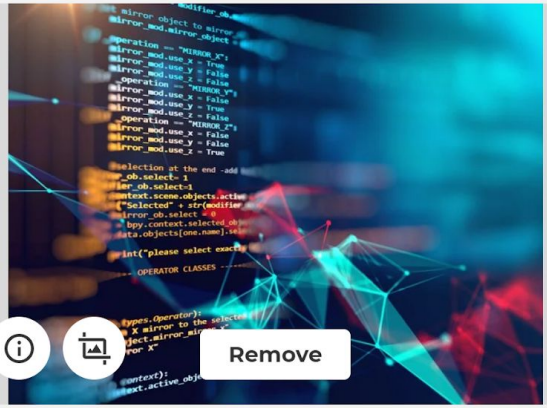
- Código, especificação, configuração manuais de sistema e utilização ✓
- Bibliotecas ✓
- Relatório de todo o passado na empresa ✓
- Websites de suporte e manutenção ✓

O que faz a deontologia profissional ?

30 sec

Points

1000



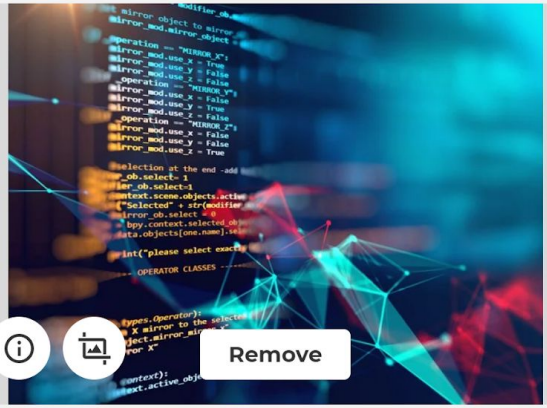
- Regula relações e comportamento dos profissionais com Sociedade e Clientes
- Regula relações e comportamento dos profissionais com Empregadores e Colega
- Regula relações e comportamento dos profissionais com família e amigos
- Estabelece um código moral ao profissional

O que faz a deontologia profissional ?

30 sec

Points

1000



- Regula relações e comportamento dos profissionais com Sociedade e Clientes ✓
- Regula relações e comportamento dos profissionais com Empregadores e Colega ✓
- Regula relações e comportamento dos profissionais com família e amigos ✓
- Estabelece um código moral ao profissional ✓

Let us go back to the analysis process for a while...

Previously...

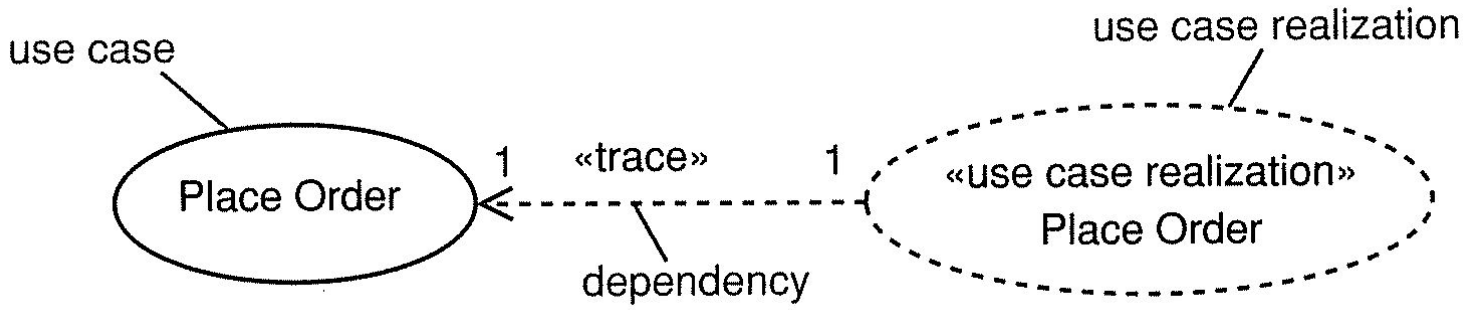
- We have been studying:
  - Use Cases
  - Activity Diagrams
  - Class Diagrams
- We saw how activity diagrams can be used to detail use cases
- Today, we revisit and analyze the use cases in greater detail (refine)
  - **Realizing Use Cases**

Why is use case realization important?

- To understand which analysis classes interact with each other in order to realize the behavior described in the Use Case
- To understand which message instances of those classes have to be exchanged in between classes in order to produce a certain behavior
  - Which are the main **operations** of the analysis Classes?
  - Which are the main **attributes** of the analysis classes?
  - Which are the main **relations** between analysis classes?
- If necessary, update the Use Case model, requirements, and analysis class model
  - **Essential to keep consistency** !

What is Use Case Realization?

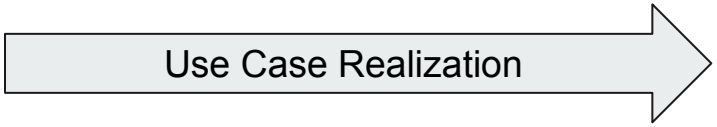
- A use case realization shows how classes collaborate to realize system functionality
- Use case realizations are an implicit part of the model backplane
  - There is exactly one use case realization per use case
  - You typically do not show them in diagrams



Use case realization

- Set of Classes that interact in order to realize the Use Case Behavior
  - e.g. In the *"Borrow Book"* Use Case, the librarian in that Use Case can interact with the *"Book"*, *"Lending Registration"* and *"User"* analysis classes to be able to realize the Use Case

Functional Requirements Specification	High Level System Specification
Use case	Analysis Class Diagram
	Interaction Diagram
	Special Requirements
	Refined Use Case



We are using the iterative method

- Analysis Class Diagrams tell a “story” about how the classes interact so that their instances work together in building the Use Case behavior
- The Interaction Diagrams show how class instances cooperate to realize that behavior
- We can find new requirements in the process of refining them
- The Use Cases can be refined even further

Interaction Diagrams

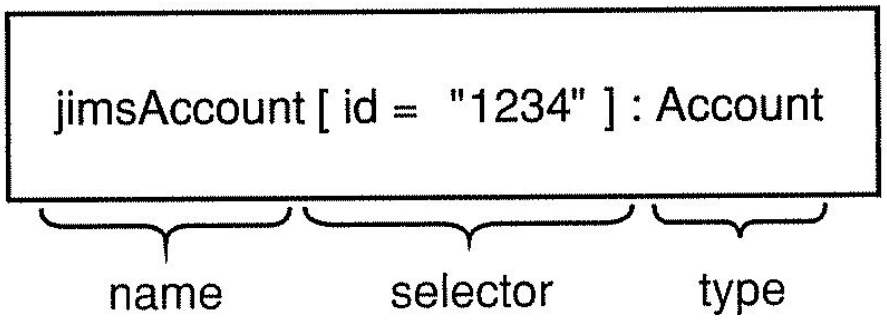
(Previously known as Sequence Diagrams in UML 1.*)

What is an interaction?

- Interaction are units of behavior of a context classifier
  - The classifier sets the context to an interaction
  - In the Use Case realization, the Use Case is the classifier that sets the context
  - When we detail an interaction, it is common to find new operations and attributes in the analysis classes
    - Therefore, we have to update and synchronize the class diagrams!

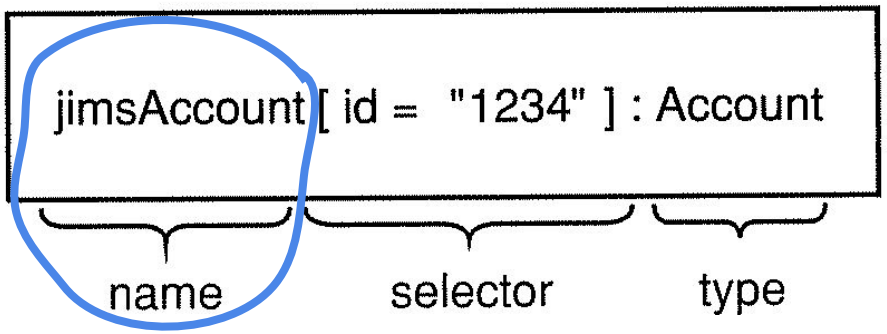
Main Elements in an Interaction: Lifeline

- Lifeline – a single participant in an interaction



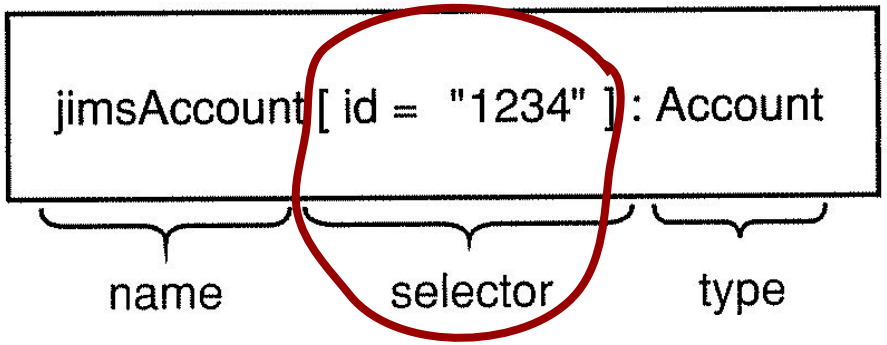
Main Elements in an Interaction: Lifeline

- Lifeline – a single participant in an interaction
  - **Name** – used to refer to the lifeline



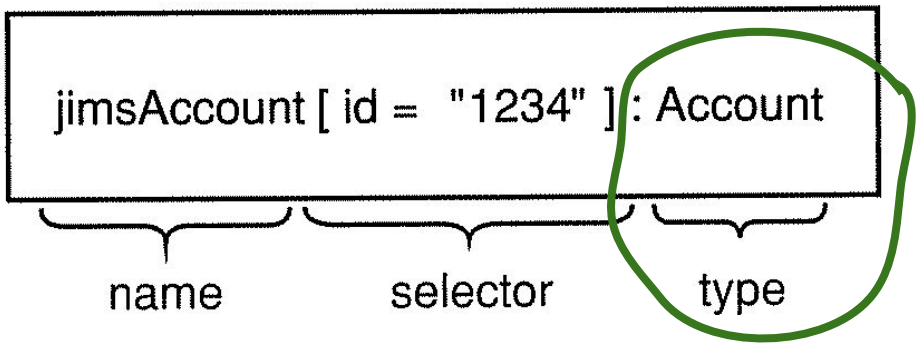
Main Elements in an Interaction: Lifeline

- Lifeline – a single participant in an interaction
  - **Name** – used to refer to the lifeline
  - **Selector** – Boolean expression to specify a particular participant instance (if it does not exist, the participant can be any instance of the corresponding type)



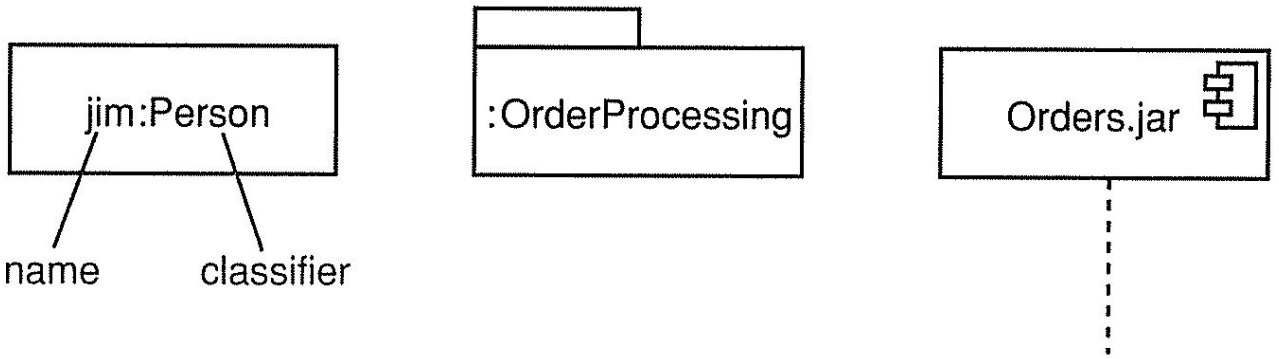
Main Elements in an Interaction: Lifeline

- Lifeline – a single participant in an interaction
  - **Name** – used to refer to the lifeline
  - **Selector** – Boolean expression to specify a particular participant instance (if it does not exist, the participant can be any instance of the corresponding type)
  - **Type** – Classifier name that the lifeline represents



Main Elements in the interaction: Lifeline

- A lifeline represents how a classifier instance participates in the interaction
- Lifelines are represented with the same icon as their type and have a vertical dashed “tail” when used in sequence diagrams



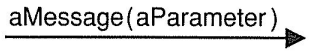
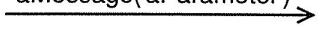
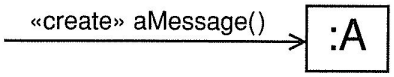
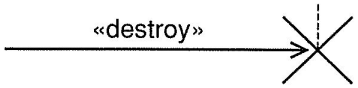
Main Elements in an Interaction: Message

- A message represents a specific communication between two lifelines in an interaction
- This may include:
  - Calling an operation - a call message
  - Creating and Destroying instances
  - Sending a signal

Messages kind

- Synchronous (sender waits for answer)
- Asynchronous (sender does not wait for answer)
- Return (returns focus of control)
- Create (new lifeline)
- Destroy (lifeline)
- Found (unknown origin)
- Lost (unknown destiny)

Message kinds

Syntax	Name	Semantics
	Synchronous message	The sender waits for the receiver to return from executing the message
	Asynchronous message	The sender sends the message and continues executing – it does *not* wait for a return from the receiver
	Message return	The receiver of an earlier message returns focus of control to the sender of that message
	Object creation	The sender creates an instance of the classifier specified by the receiver
	Object destruction	The sender destroys the receiver If its lifeline has a tail, this is terminated with an X
	Found message	The sender of the message is outside the scope of the interaction Use this when you want to show a message receipt, but don't want to show where it came from
	Lost message	The message never reaches its destination May be used to indicate error conditions in which messages are lost

Interaction Diagrams

- Show communication among objects
- Goal:
  - **Specify the Use Case realization**
  - Specify how to realize an operation

There are four types of interaction diagrams

- **Sequence diagrams**
- **Communication diagrams**
- Interaction overview diagrams
- Timing diagrams

Sequence diagrams vs Communication diagrams

- Both specify the same information
- Each focuses in different aspect
- Sequence Diagram (**time oriented**)
  - Shows how messages are organized
  - Do not show how to get the reception object
- Communication Diagram (**space oriented**)
  - Show the static and dynamic relations in between objects
  - The sequence of messages is explicitly shown
  - Time is not one dimension

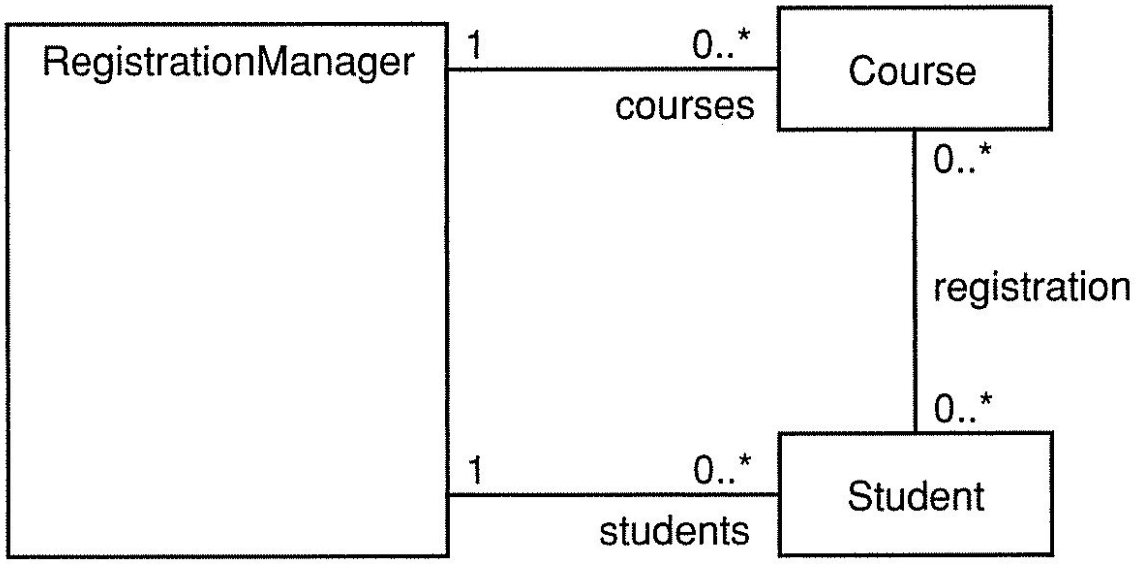
Sequence diagrams

Show interactions between lifelines as a time-oriented sequence of events

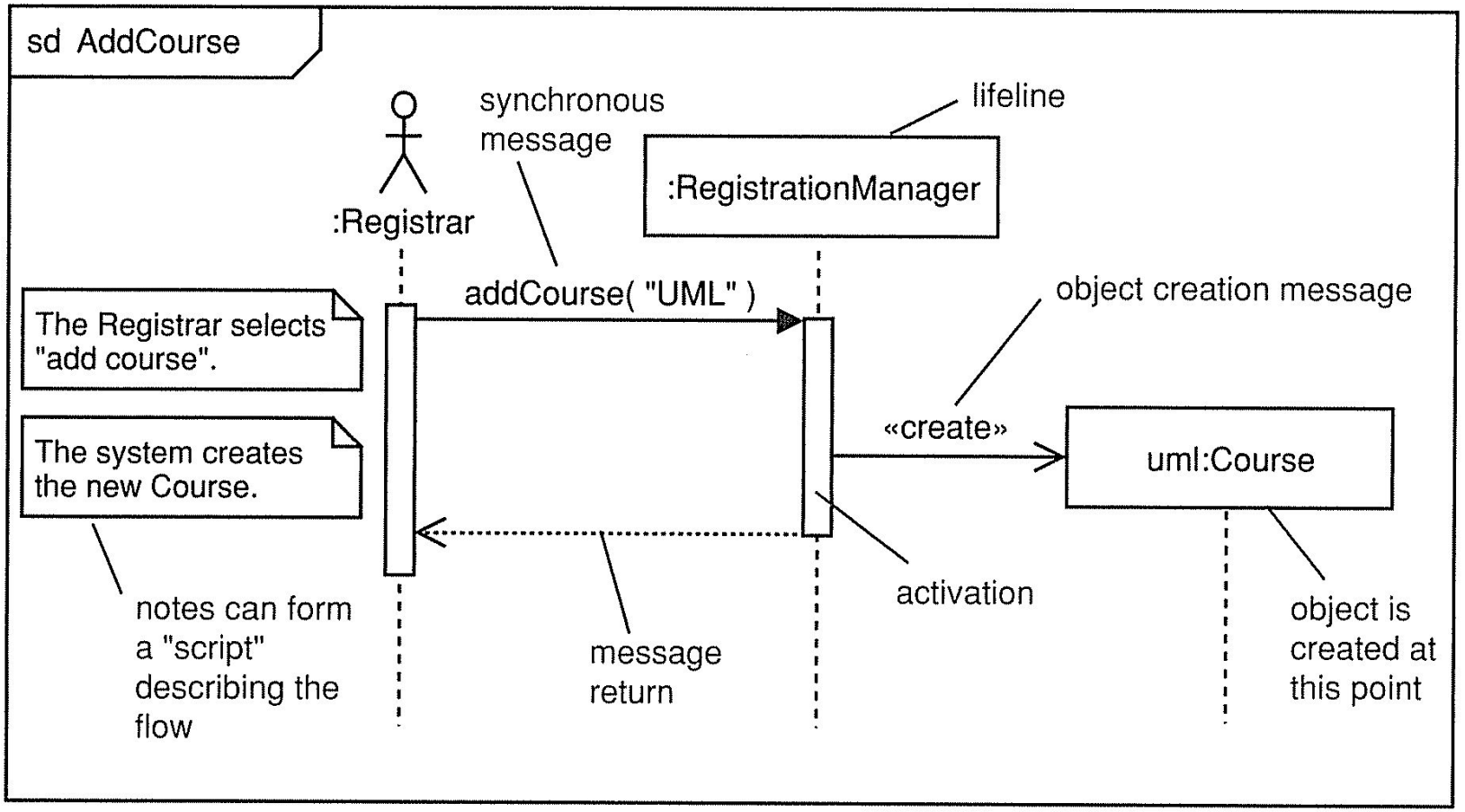
Consider the AddCourse use case

Use case: AddCourse
ID: 8
Brief description: Add details of a new course to the system.
Primary actors: Registrar
Secondary actors: None.
Preconditions: 1. The Registrar has logged on to the system.
Main flow: 1. The Registrar selects "add course". 2. The Registrar enters the name of the new course. 3. The system creates the new course.
Postconditions: 1. A new course has been added to the system.
Alternative flows: CourseAlreadyExists

Analysis class diagram corresponding to the use case AddCourse

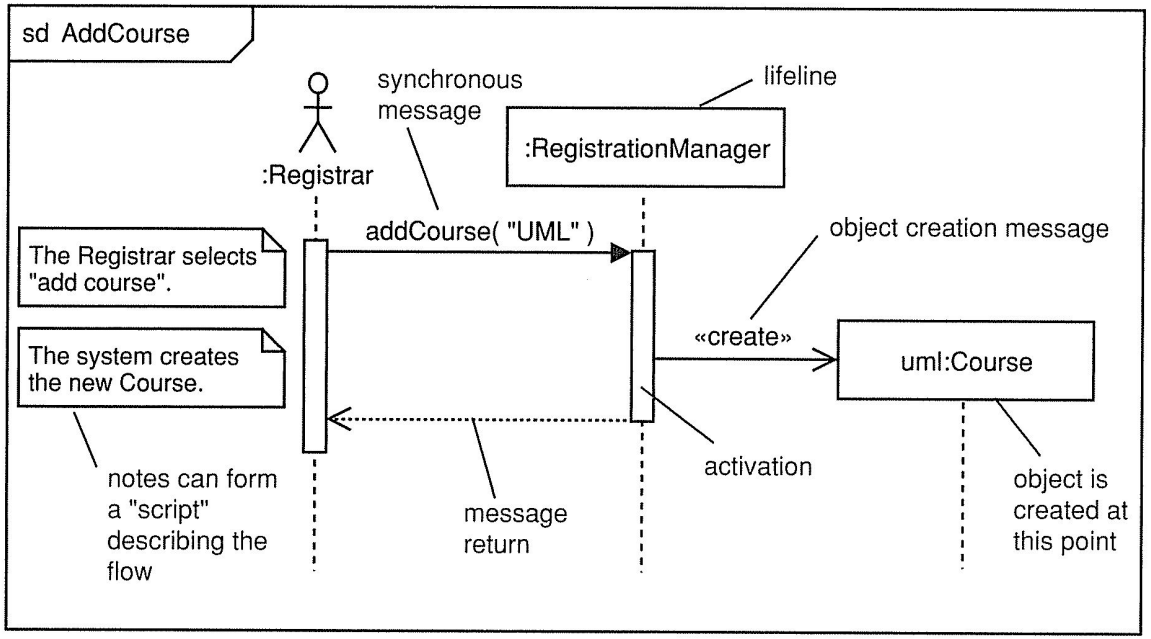


The Sequence Diagram (sd) AddCourse realizes the behavior specified in the AddCourse use case



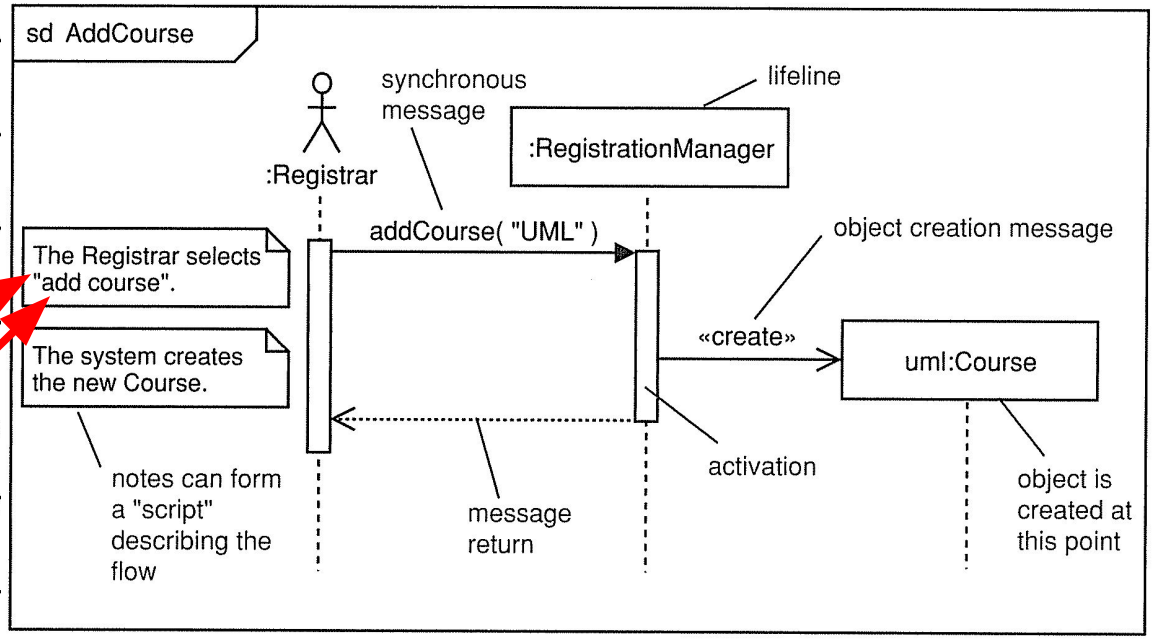
Interaction diagrams are not verbatim transcriptions of the use case

Use case: AddCourse
ID: 8
Brief description: Add details of a new course to the system.
Primary actors: Registrar
Secondary actors: None.
Preconditions: 1. The Registrar has logged on to the system.
Main flow: 1. The Registrar selects "add course". 2. The Registrar enters the name of the new course 3. The system creates the new course.
Postconditions: 1. A new course has been added to the system.
Alternative flows: CourseAlreadyExists



The first two steps of the flow are not relevant at analysis - user interface is a design matter

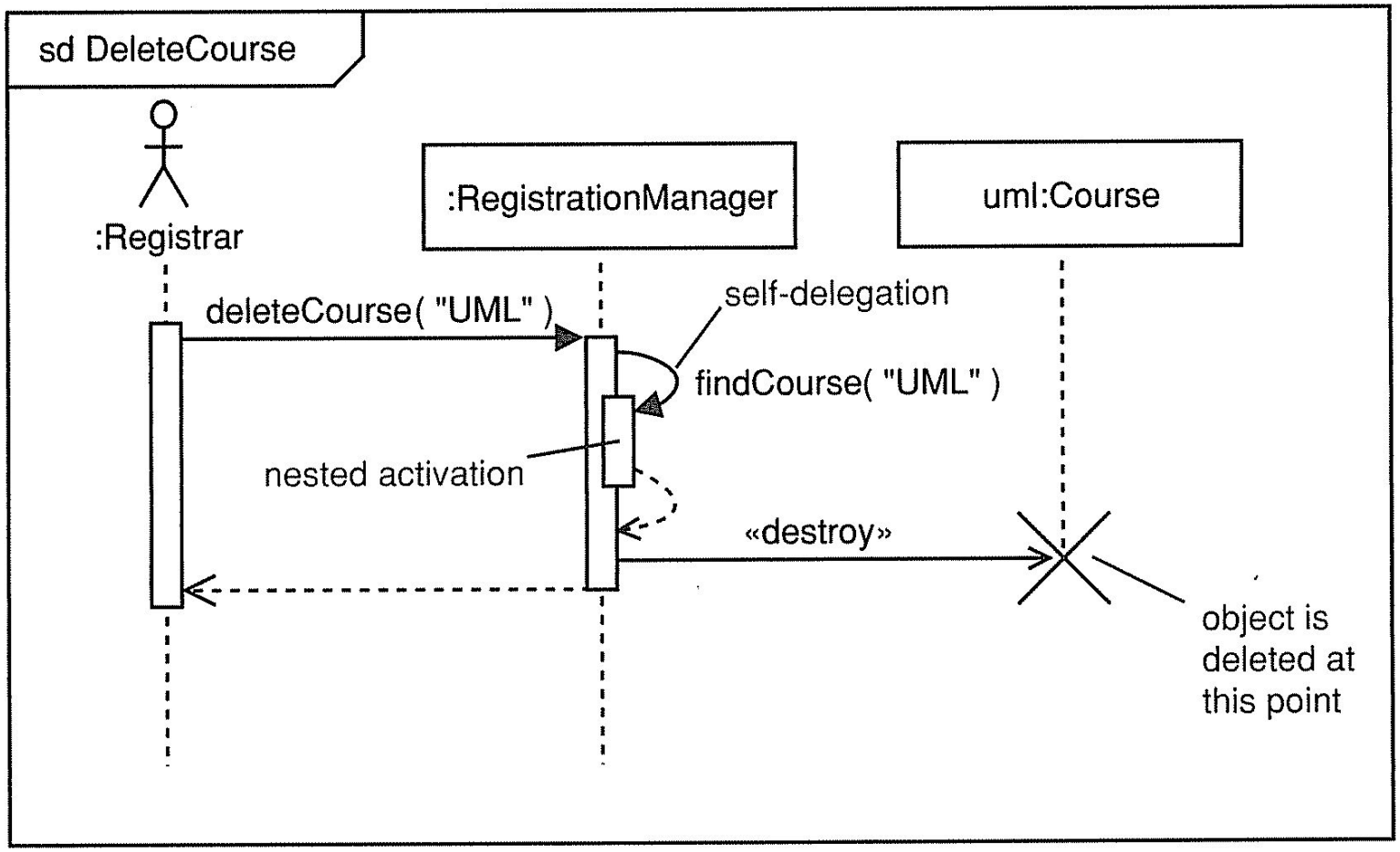
Use case: AddCourse
ID: 8
Brief description: Add details of a new course to the system.
Primary actors: Registrar
Secondary actors: None.
Preconditions: 1. The Registrar has logged on to the system.
Main flow: 1. The Registrar selects "add course". 2. The Registrar enters the name of the new course 3. The system creates the new course.
Postconditions: 1. A new course has been added to the system.
Alternative flows: CourseAlreadyExists



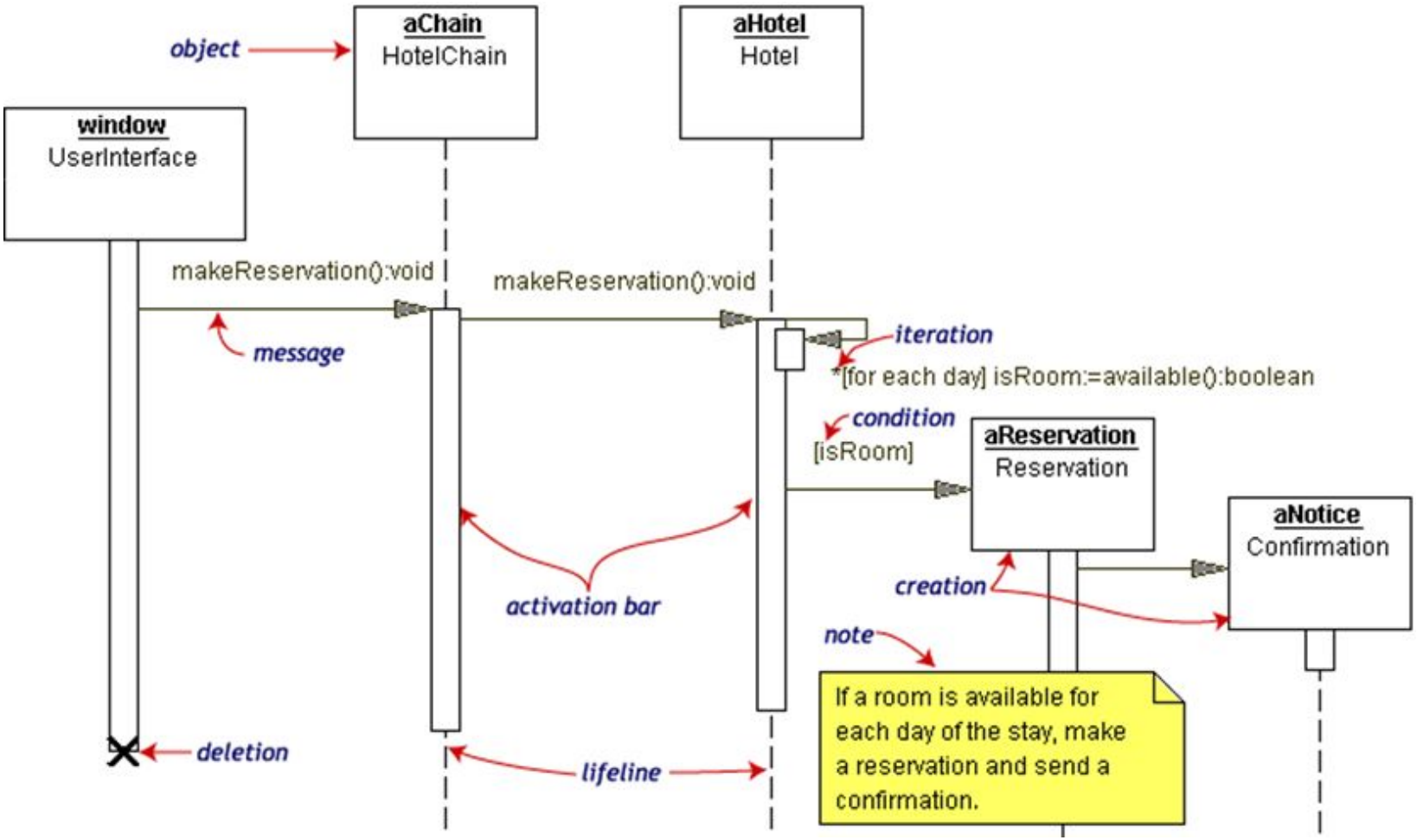
The DeleteCourse use case

Use case: DeleteCourse
ID: 8
Brief description: Remove a course from the system.
Primary actors: Registrar
Secondary actors: None.
Preconditions: 1. The Registrar has logged on to the system.
Main flow: 1. The Registrar selects "delete course". 2. The Registrar enters the name of the course. 3. The system deletes the course.
Postconditions: 1. A course has been removed from the system.
Alternative flows: CourseDoesNotExist

This second example illustrates self-delegation and object destruction

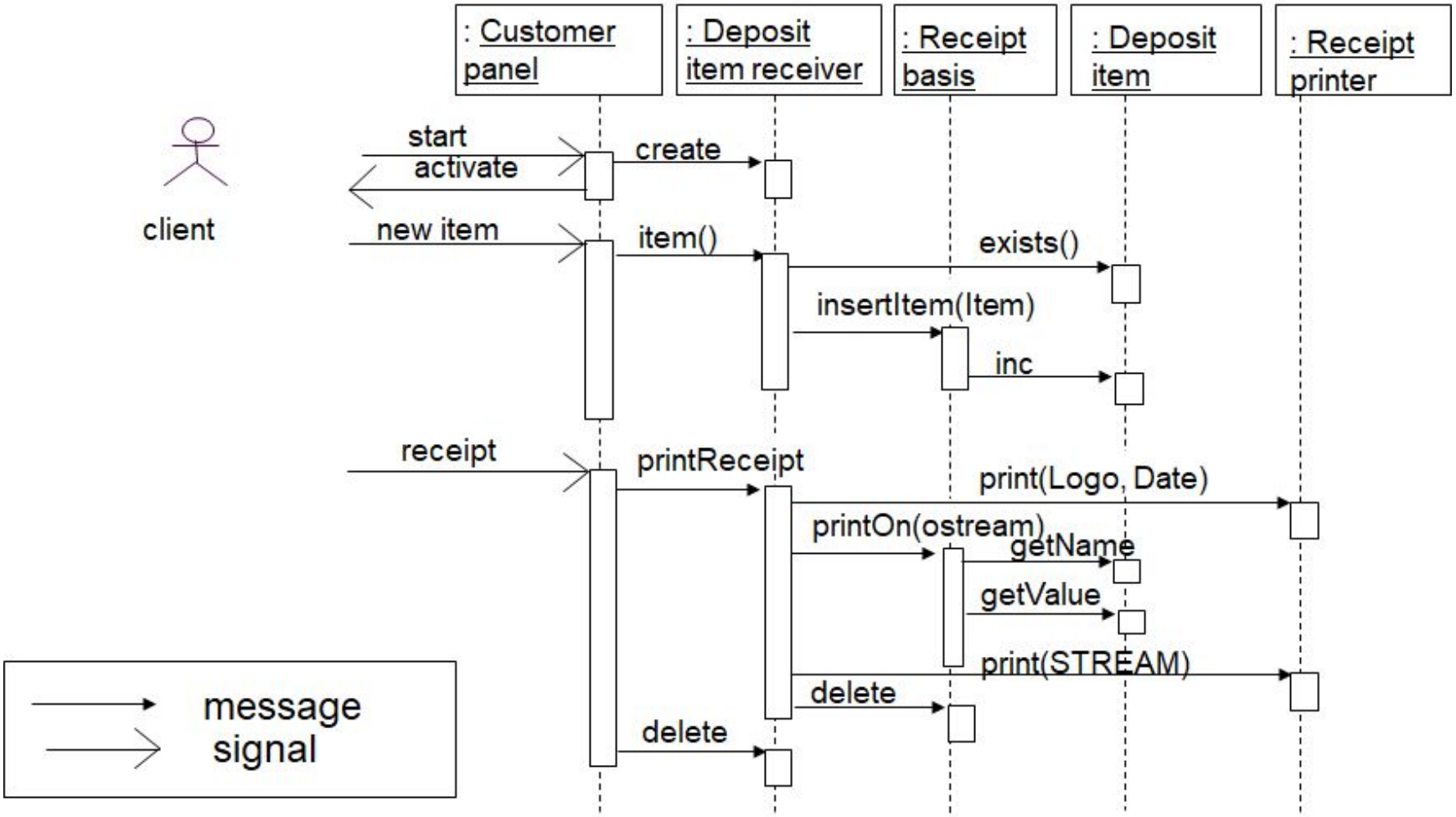


Sequence diagrams basic syntax summary

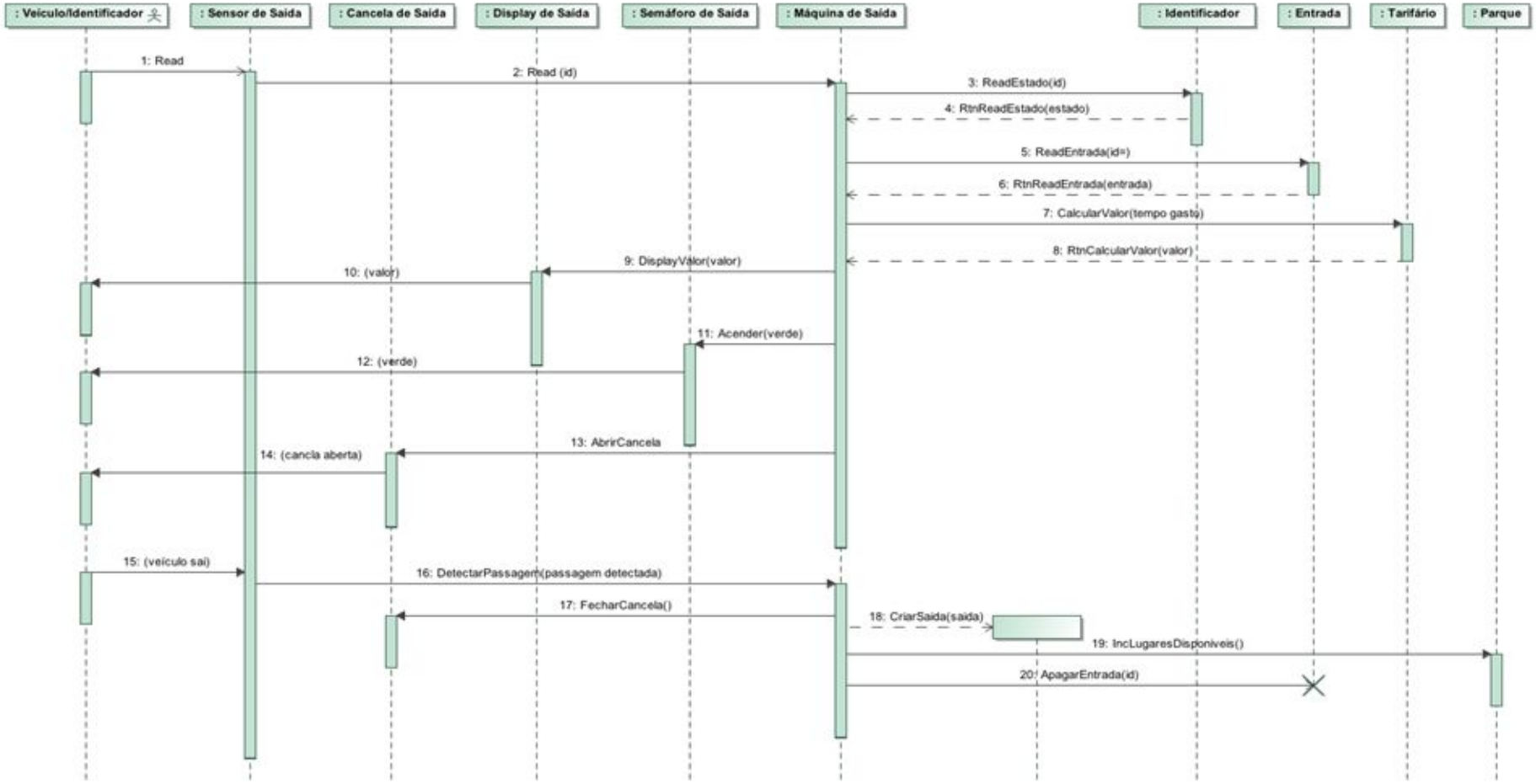


Interaction diagrams design

Interaction diagrams can become fairly complex



Sequence diagram for the scenario “authorized car to leave the parking garage”



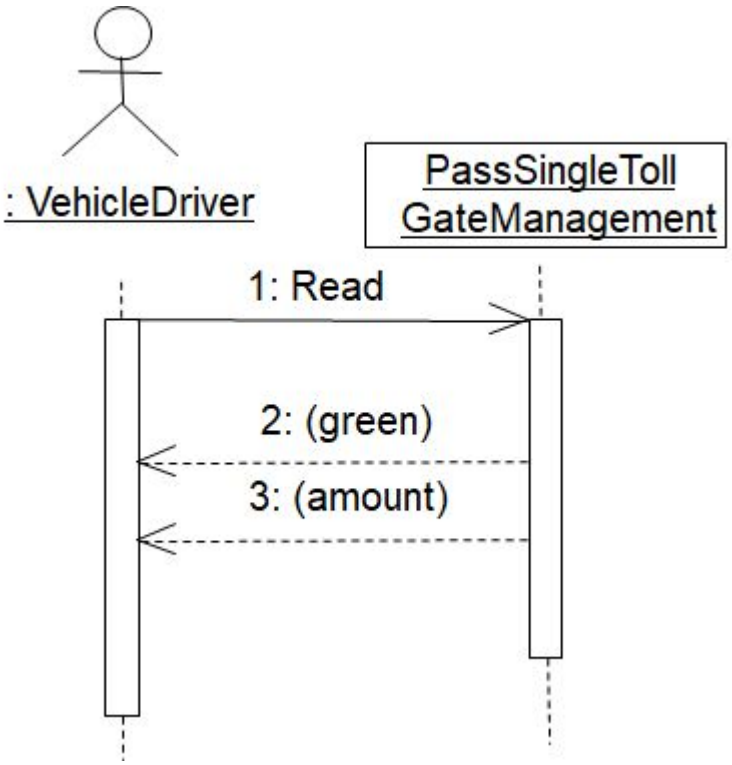
Constructing a sequence diagram can be challenging - we need guidelines

- It is not always obvious how to build the SD
  - Control objects are involved when we detect that the interface object leads/controls the execution
  - The SD can start to be built by using the three types of objects as guides.
- Some construction rules:
  - The first message is always sent by an actor
  - The first message is always received by and interface object
  - Add a control object when the interface object becomes a decision maker

Building Design Sequence Diagrams: a step-by-step guide

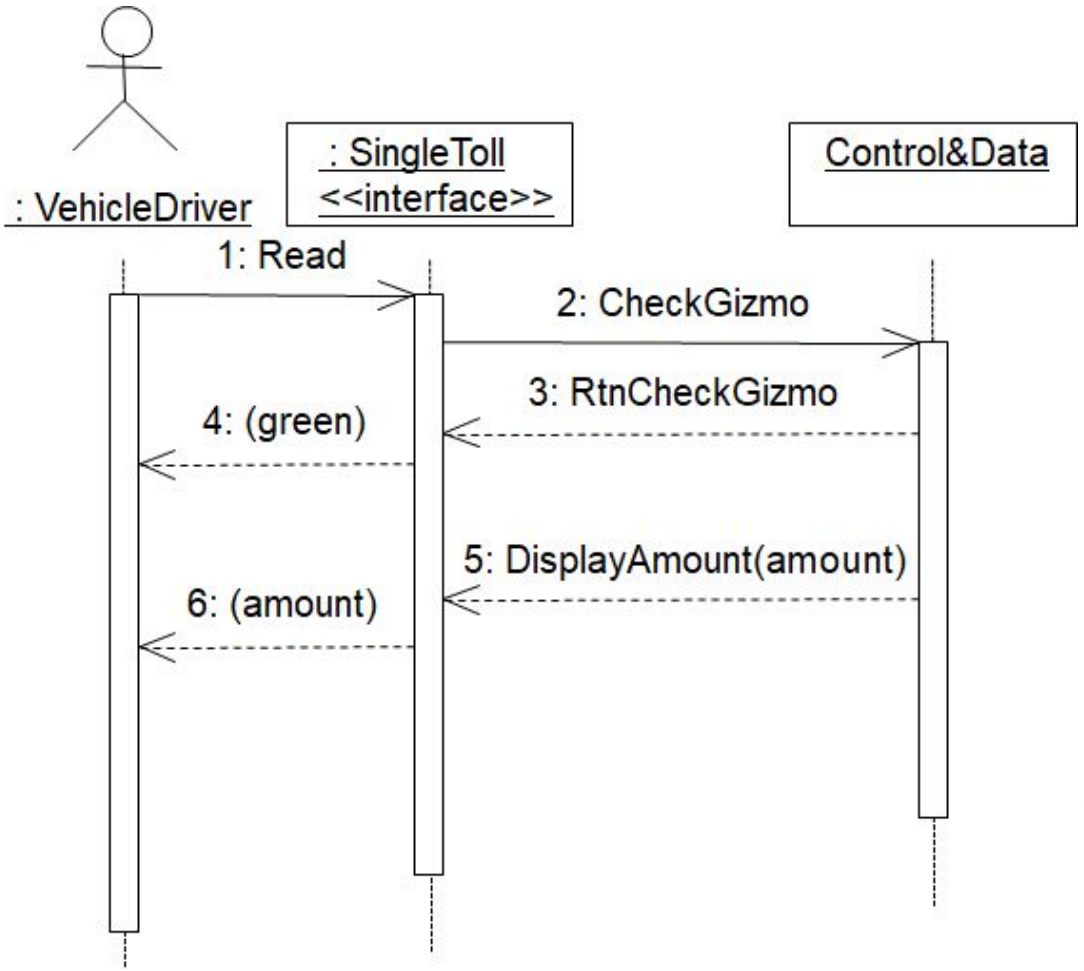
The “25 de Abril” Bridge Toll case study

Rule #1: when starting, the system is seen as a black box



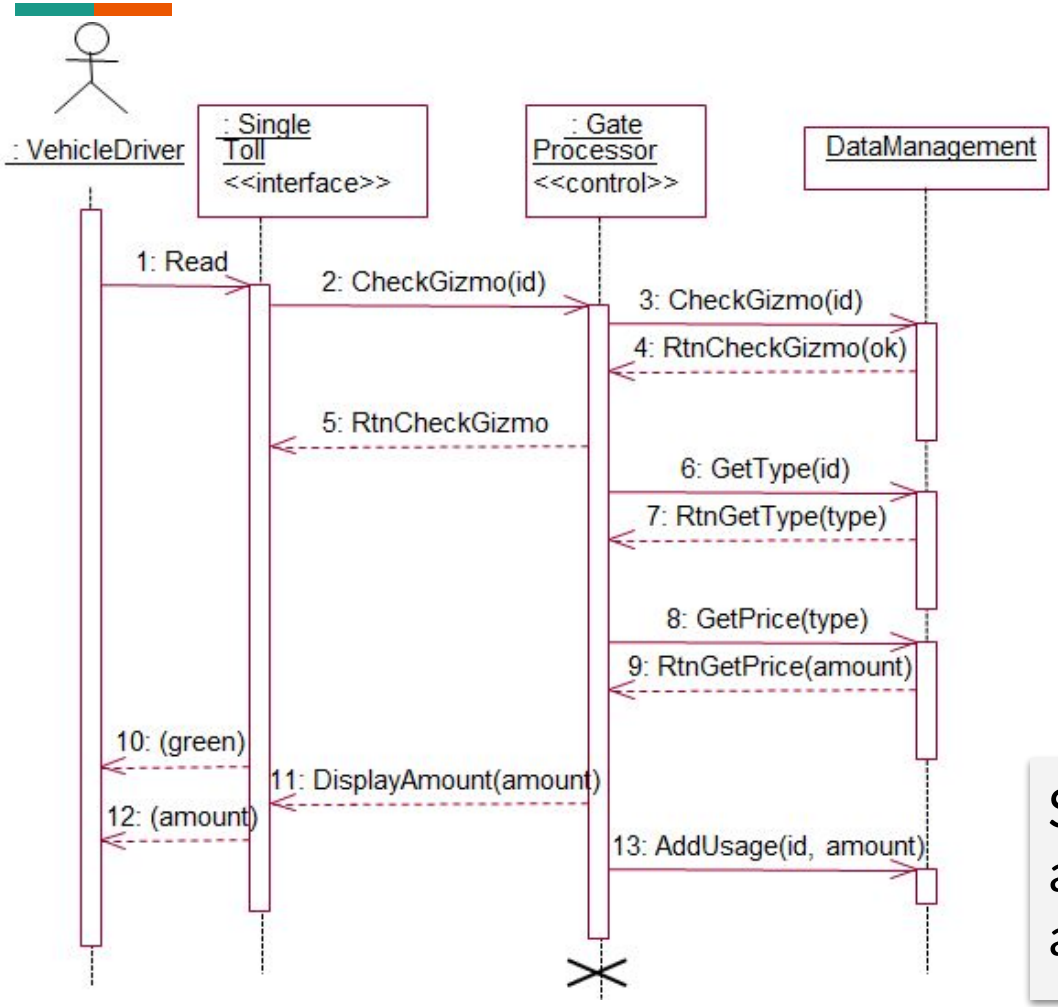
Sequence Diagram with the system represented as a black box

Rule #2: All messages from/to actors pass by an interface object



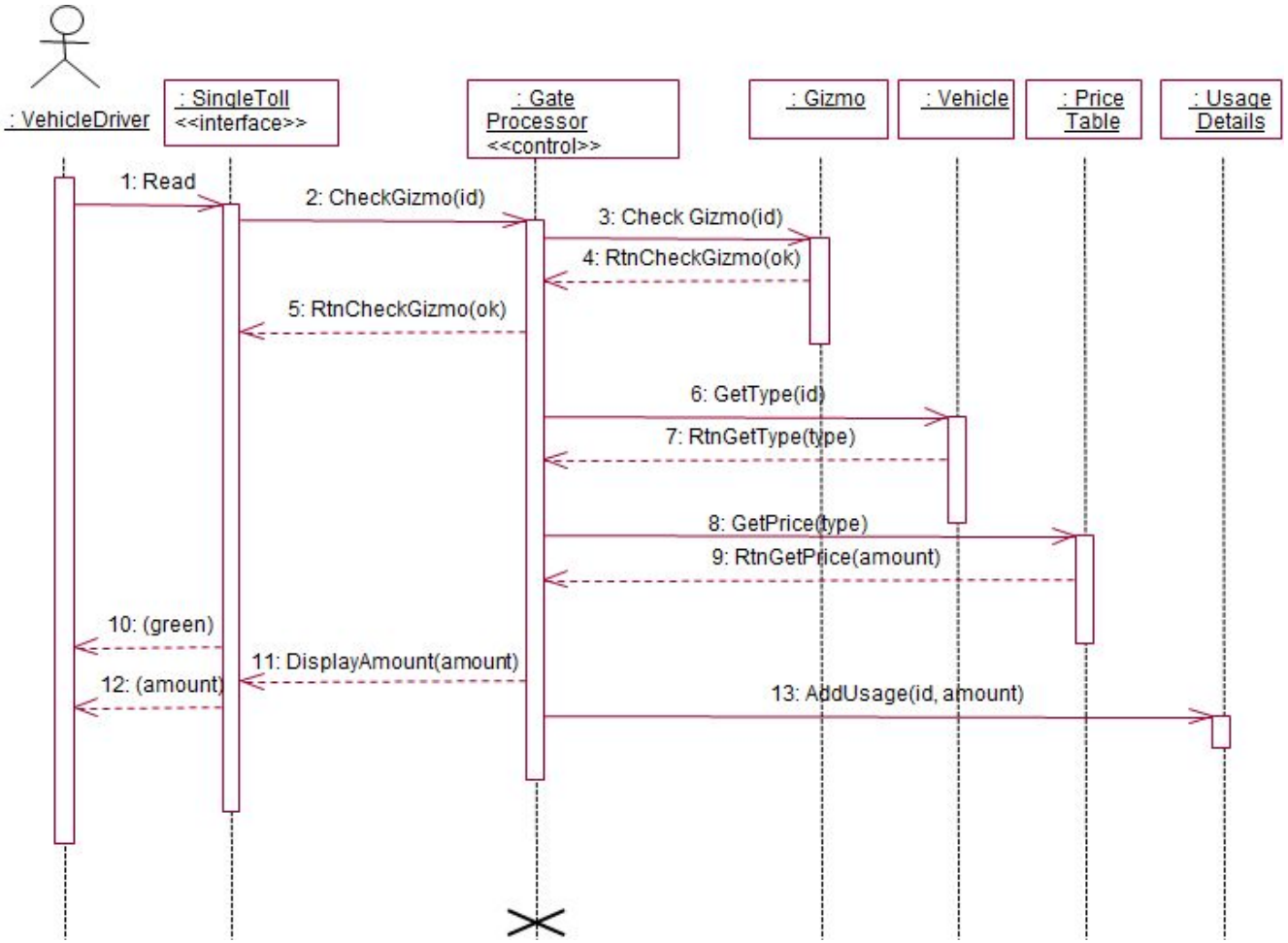
Sequence Diagram with an interface object

Rule #3: we need control objects for complex functionalities



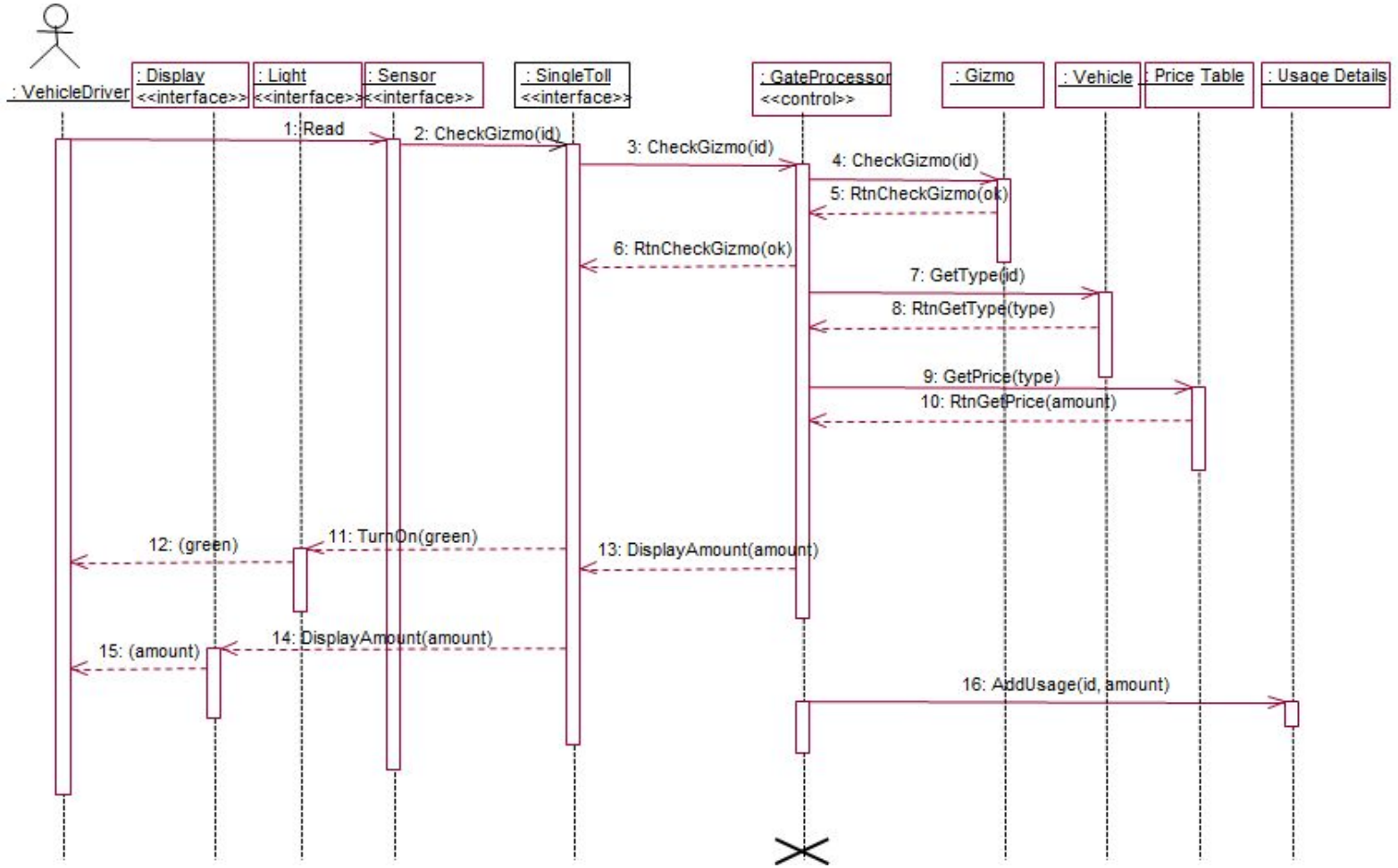
Sequence Diagram with an interface object and a control object

Rule #4: Control objects centralize processing involving entity objects

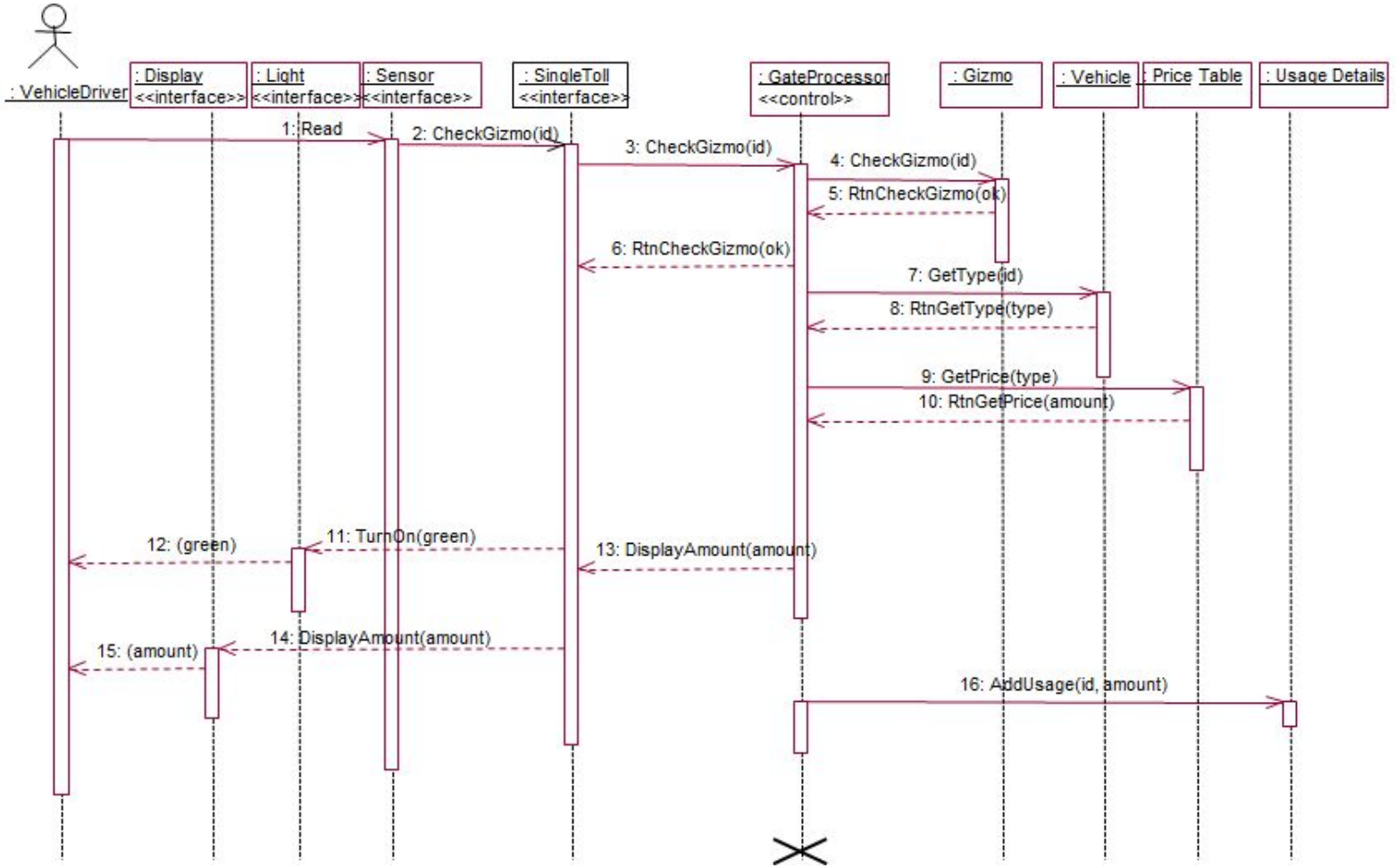


Sequence Diagram with an interface object and a control object

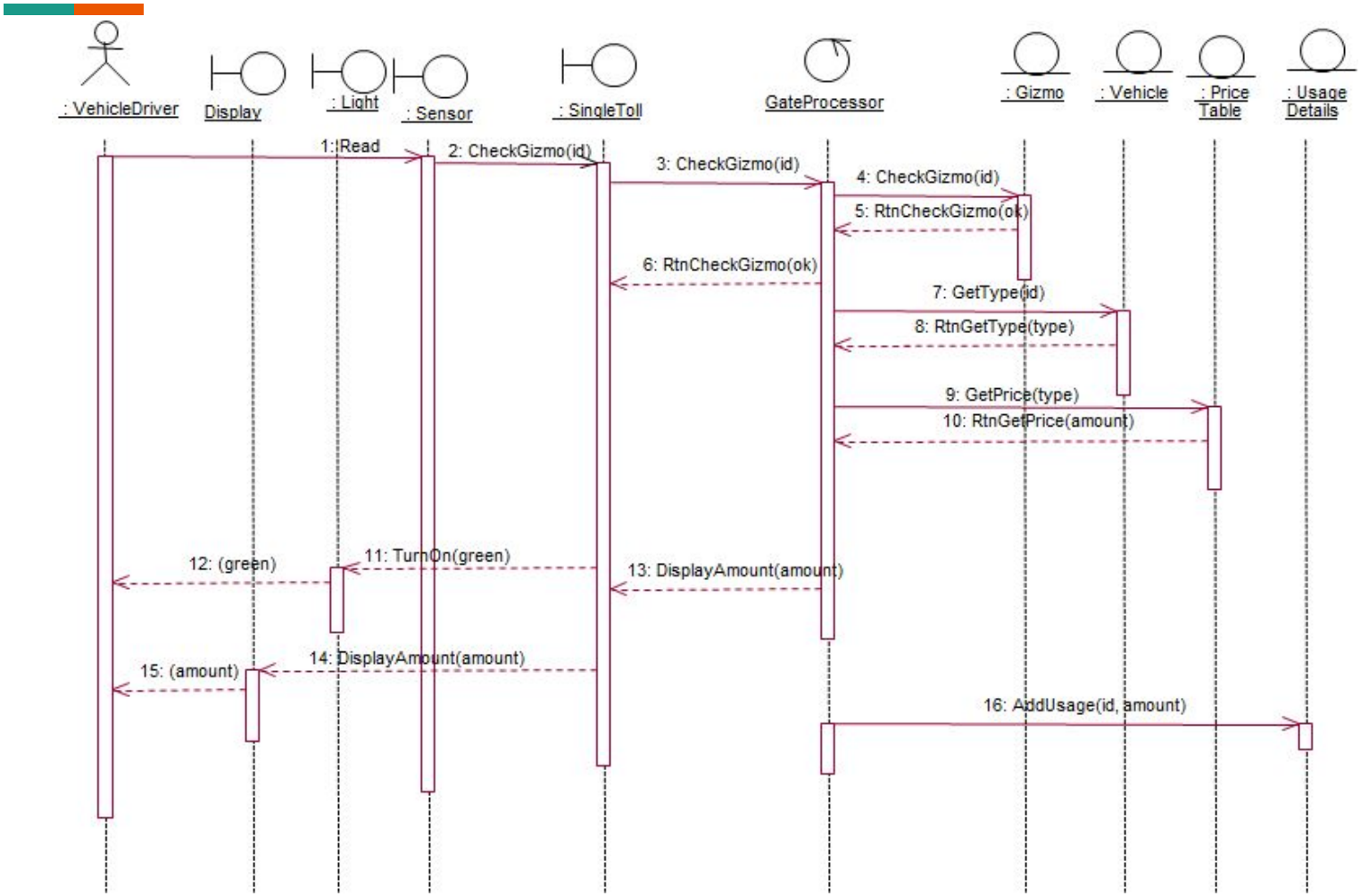
Rule #5: the interface objects can be several at the same time



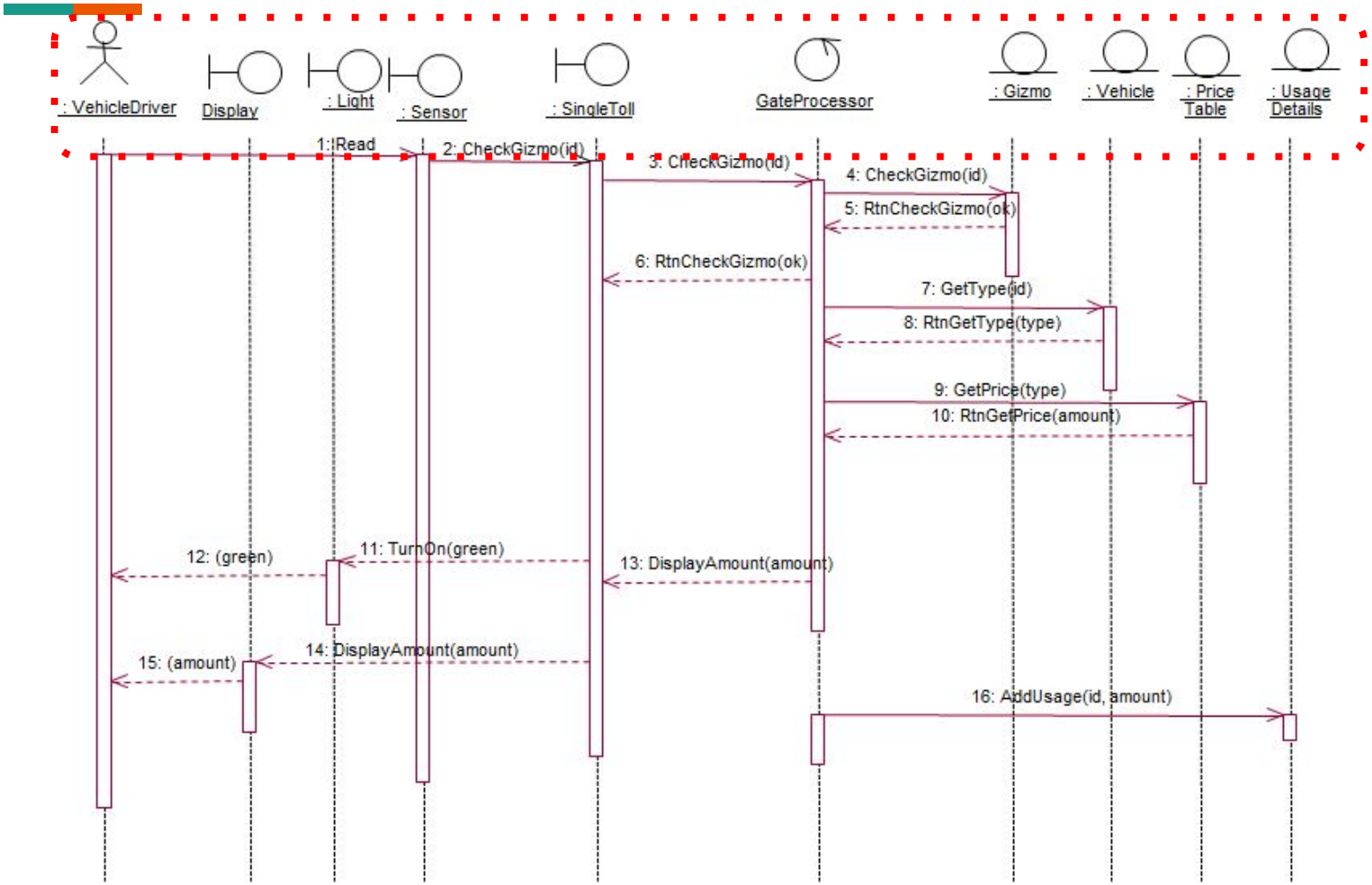
Rule #6: the interface objects can be hierarchically connected



You can also use an alternative notation for stereotyping your lifelines (remember these decorations from class diagrams?)



This is also a strategy for identifying relevant design classes

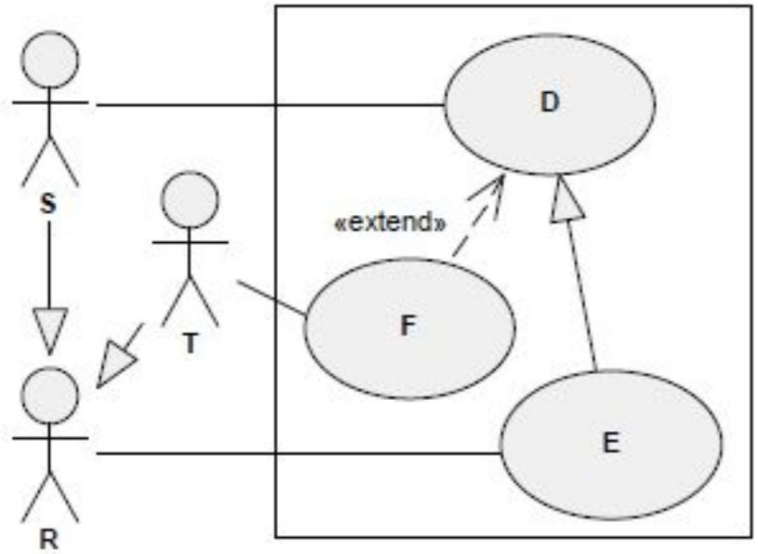


Let's make a break for a small Quizz

Qual das seguintes frases **não caracteriza** um ator num diagrama de casos de uso?

A. Um ator não comunica directamente com outros actores.
B. Um ator pode especializar outro ator.
C. Um ator pode representar o papel de um utilizador no sistema.
D. Um determinado utilizador de um sistema deve ser sempre representado pelo mesmo ator.

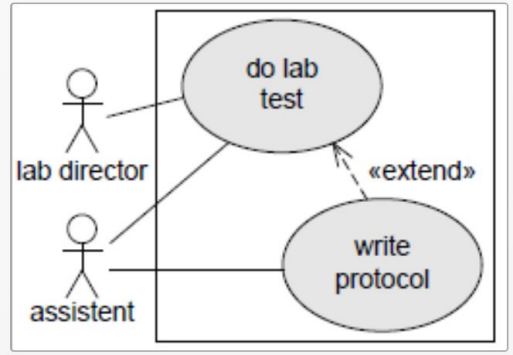
Que combinação de actores comunica com o Use Case D: (escolha a resposta **correcta**) *[uma resposta]*:



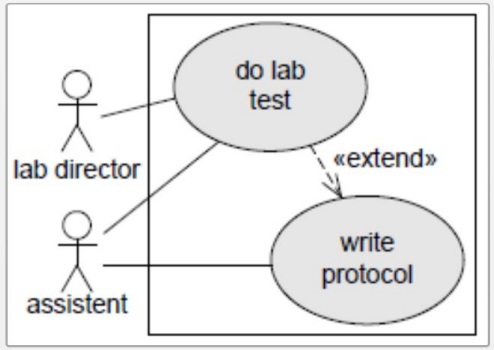
A. R ∧ T
B. T ∧ S
C. T ∧ S ∧ R
D. S
E. R ∧ S

Como é que modelaria a seguinte situação em UML 2.0:
O director do laboratório faz um teste ao laboratório junto com o seu assistente. O assistente tem sempre de escrever o protocolo durante o teste do laboratório.
(escolha a opção **mais correcta**) *[uma resposta]*:

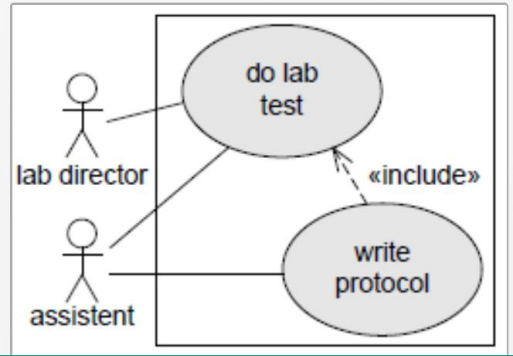
A



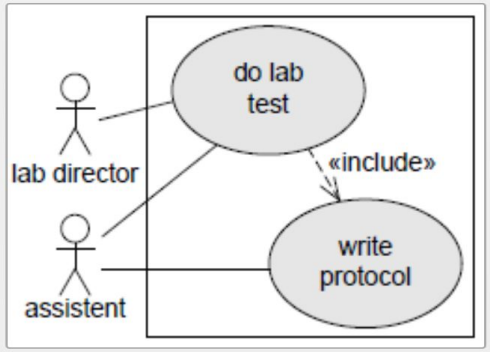
B

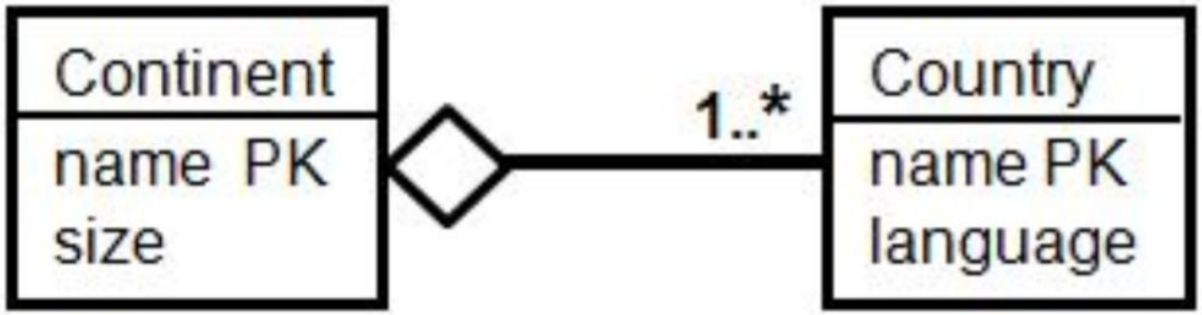


C



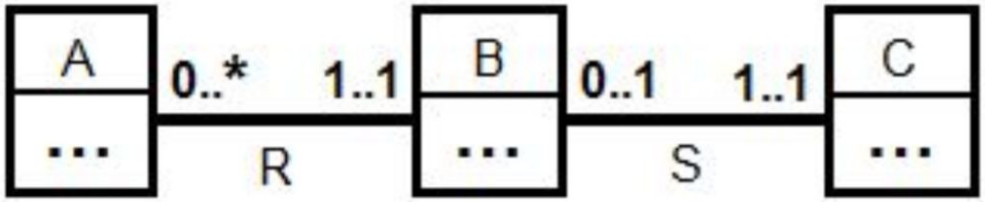
D





A. Cada país deve pertencer a um continente.
B. Um país pode estar em nenhum continente.
C. Um país pode falar dois idiomas.
D. Dois continentes diferentes não podem ter o mesmo tamanho.
E. Nenhuma das anteriores.

[B3] O diagrama UML abaixo apresenta algumas restrições nas cardinalidades das classes A,B e C. Qual das seguintes combinações é possível? Note que a cardinalidade da classe C, denotada por |C|, indica o número de objetos da classe.



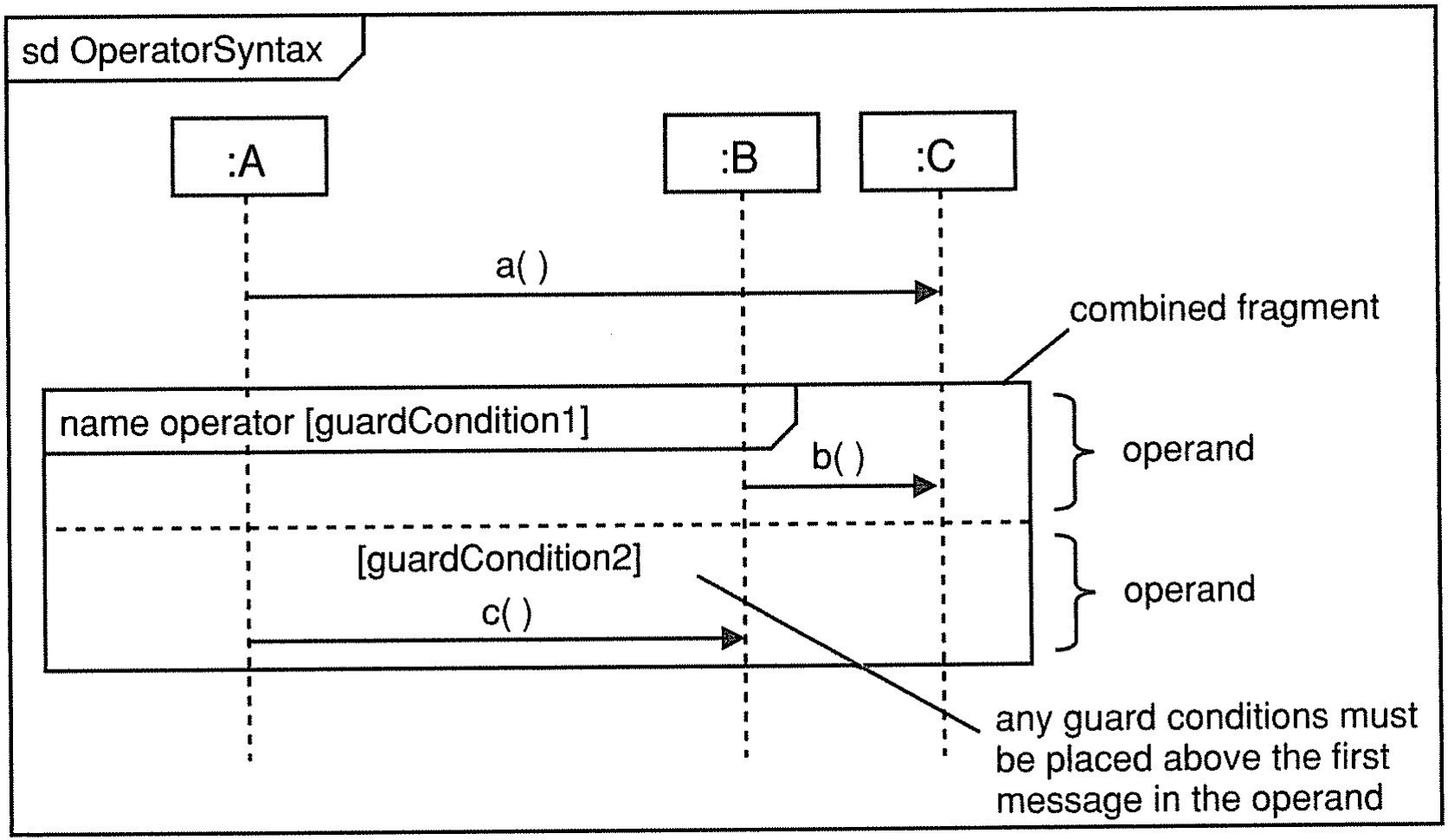
A. |A| = 10; |B| = 20; |C| = 10
B. |A| = 0; |B| = 10; |C| = 1
C. |A| = 0; |B| = 0; |C| = 1
D. |A| = 0; |B| = 10; |C| = 0
E. Nenhum das anteriores

Combined fragments and operators

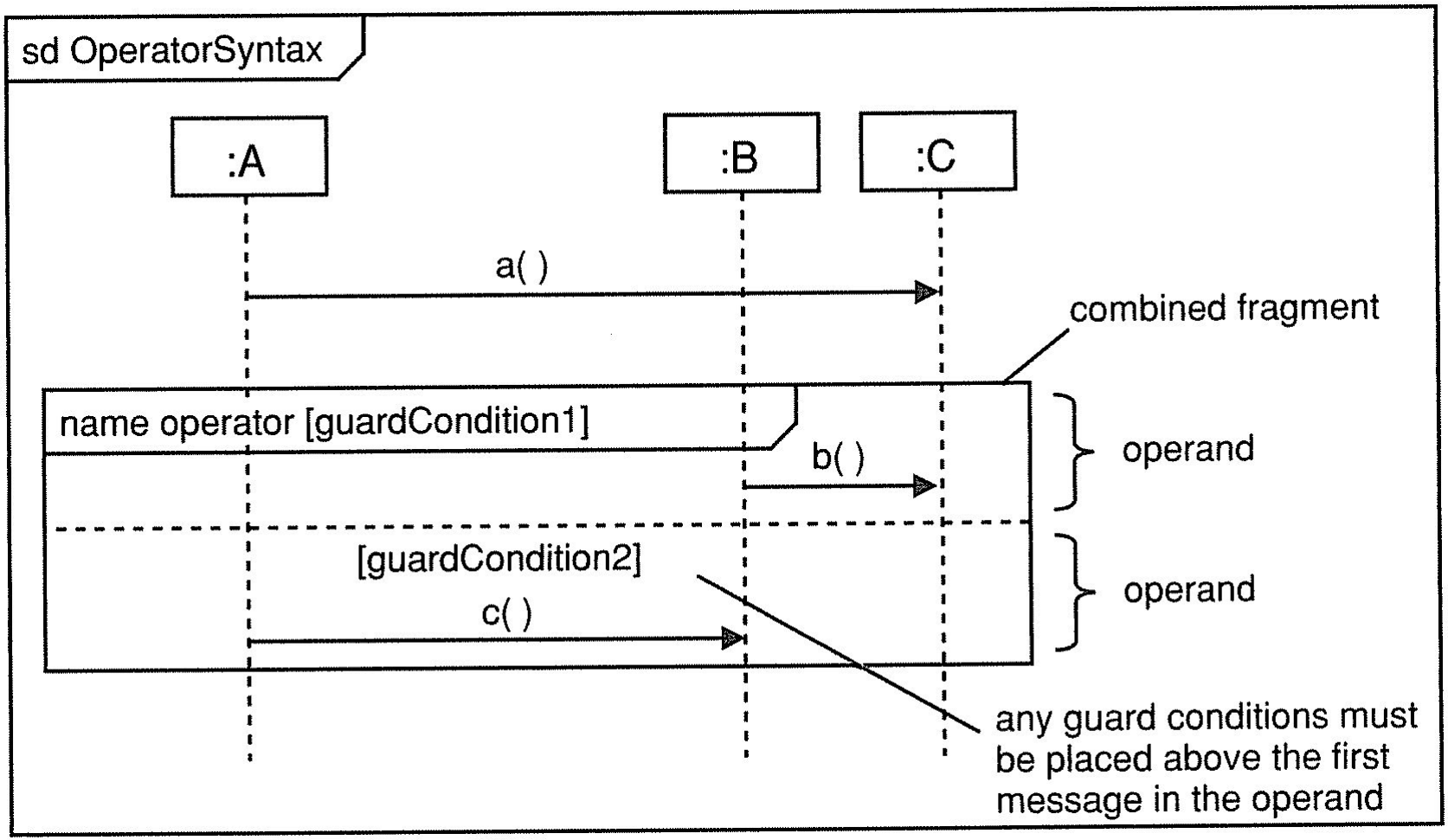
Combined fragments and operators

- Combined fragments divide a sequence diagram into different regions with different behaviors
- Fragments are composed of:
  - An operator
    - Determines **how** the operands execute
  - One or two operands
  - Zero or more guard conditions
    - Determine **if** operands are executed, or not

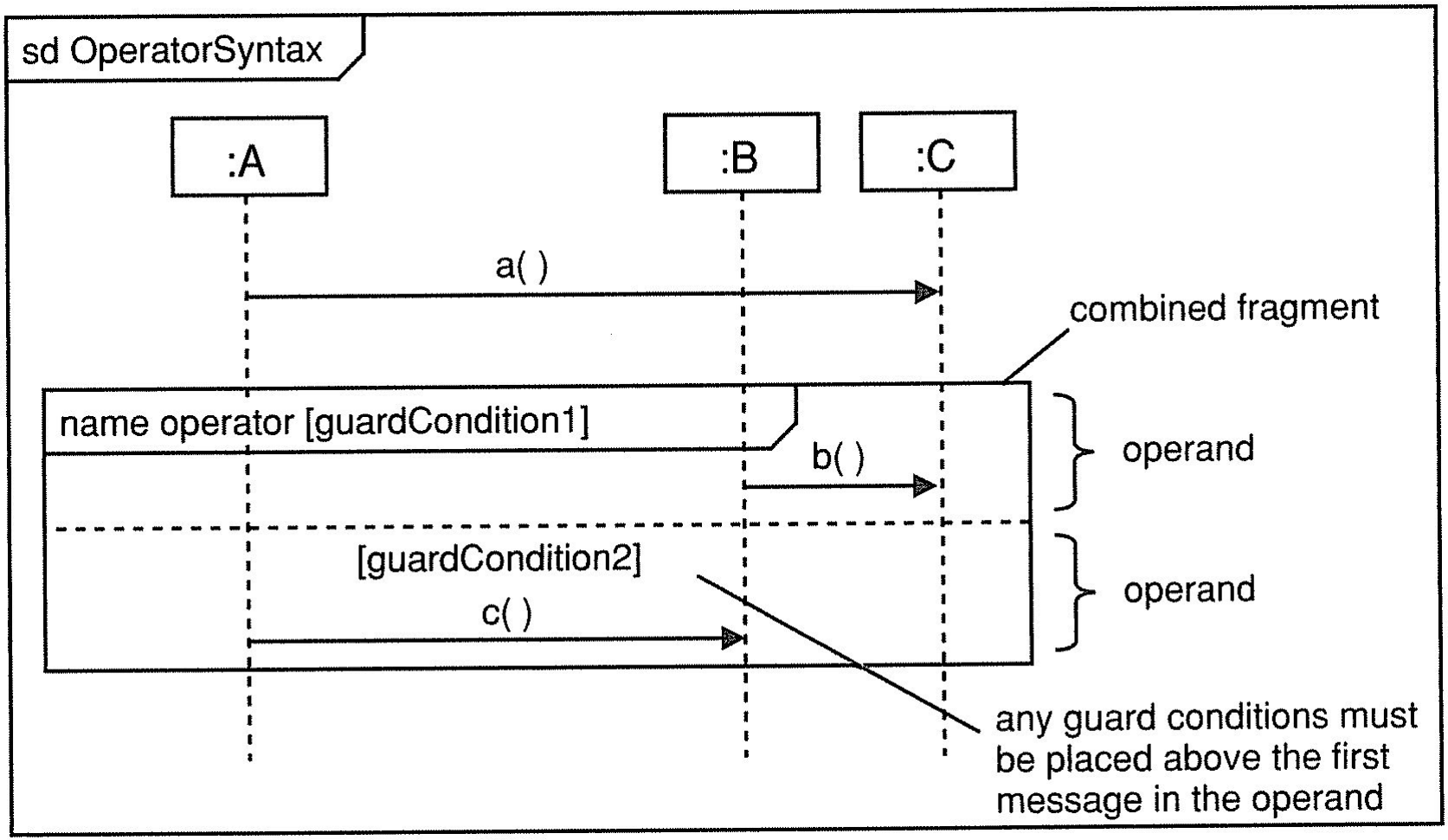
Combined fragments and operators



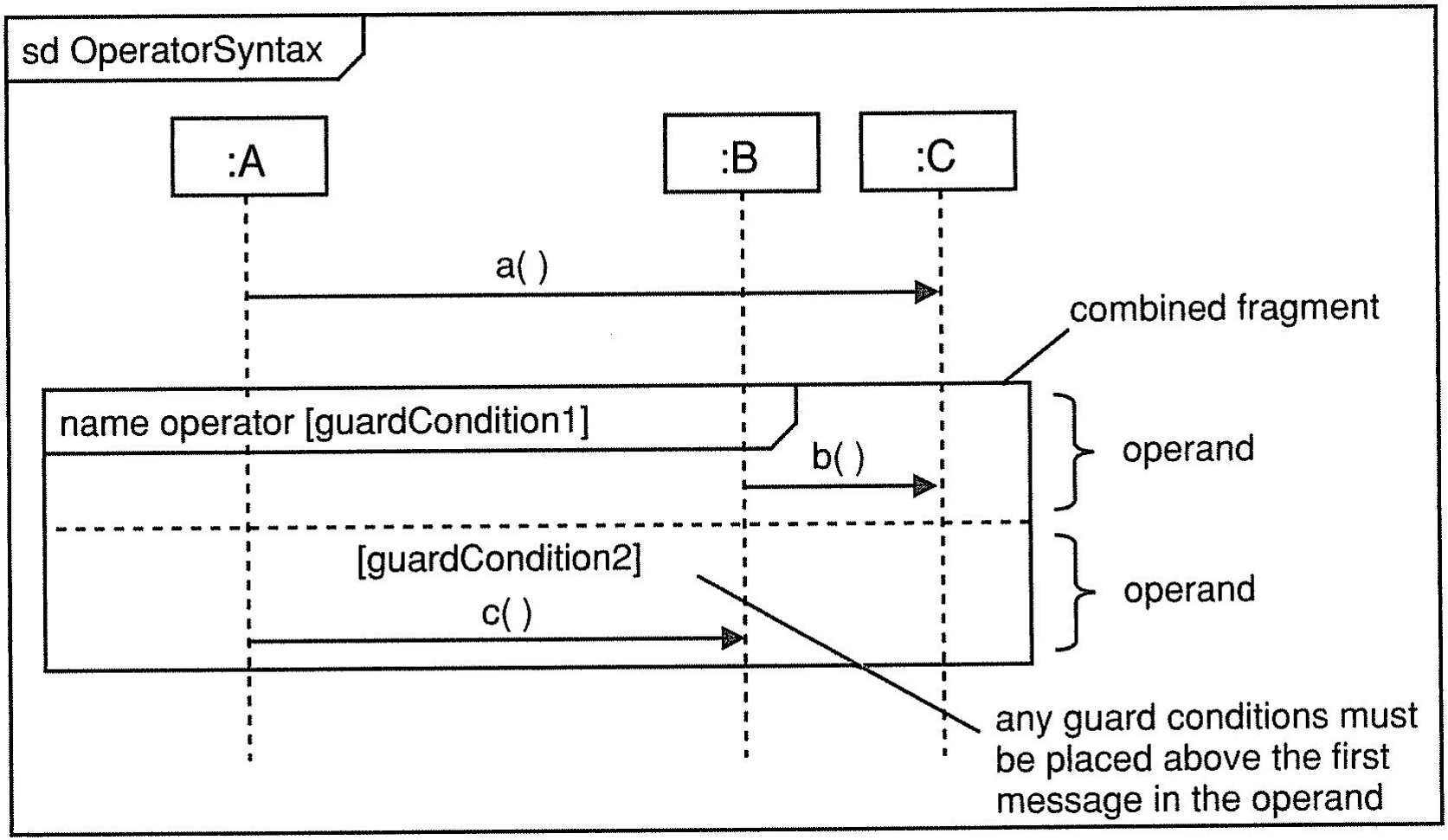
Combined fragments divide a sequence diagram into different areas with different behavior



An operator determines how its operands will execute



Guard conditions determine whether their operands execute

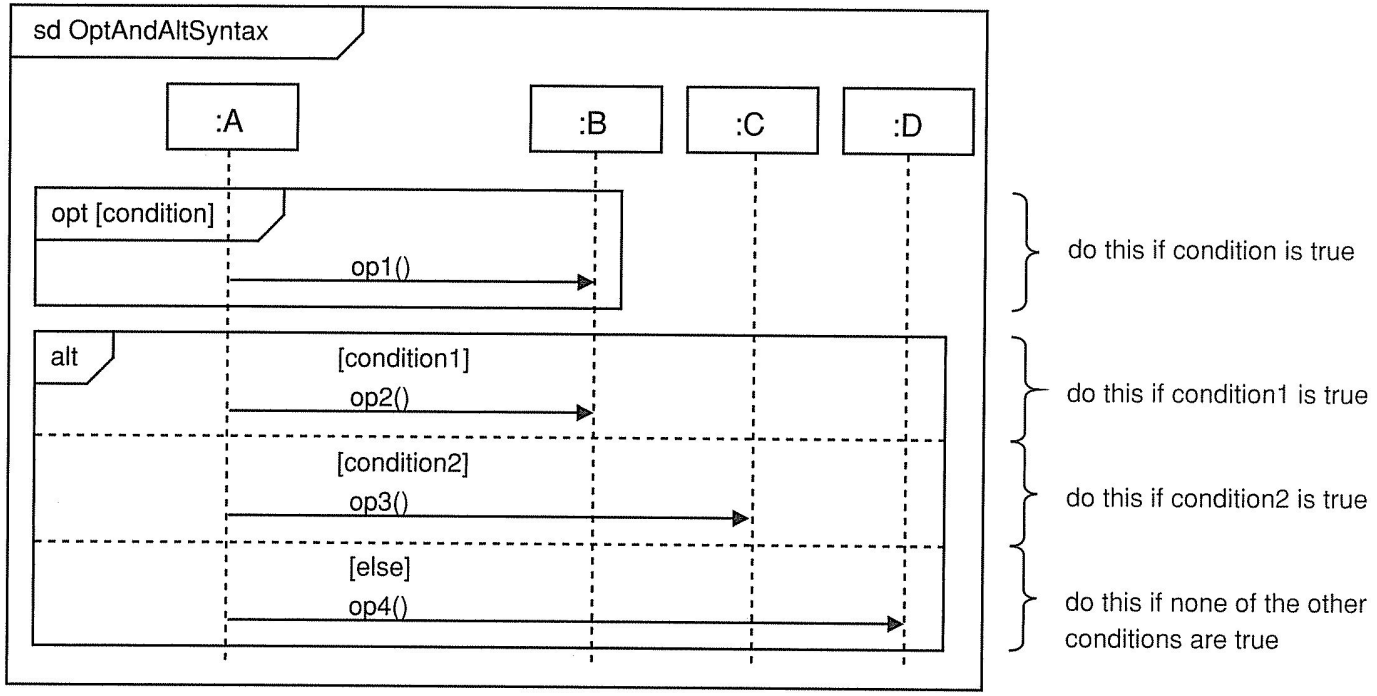


Types of combined fragments

- Alternative (**alt**)
- Option (**opt**)
- Iterate (**loop**)
- Interruption (**break**)
- Reference (**ref**)
- Parallel (**par**)
- Region (**region**)
- Weak sequence (**seq**)
- Strong sequence (**strict**)
- Negative (**neg**)
- Ignore (**ignore**)
- Consider (**consider**)
- Assertion (**assert**)

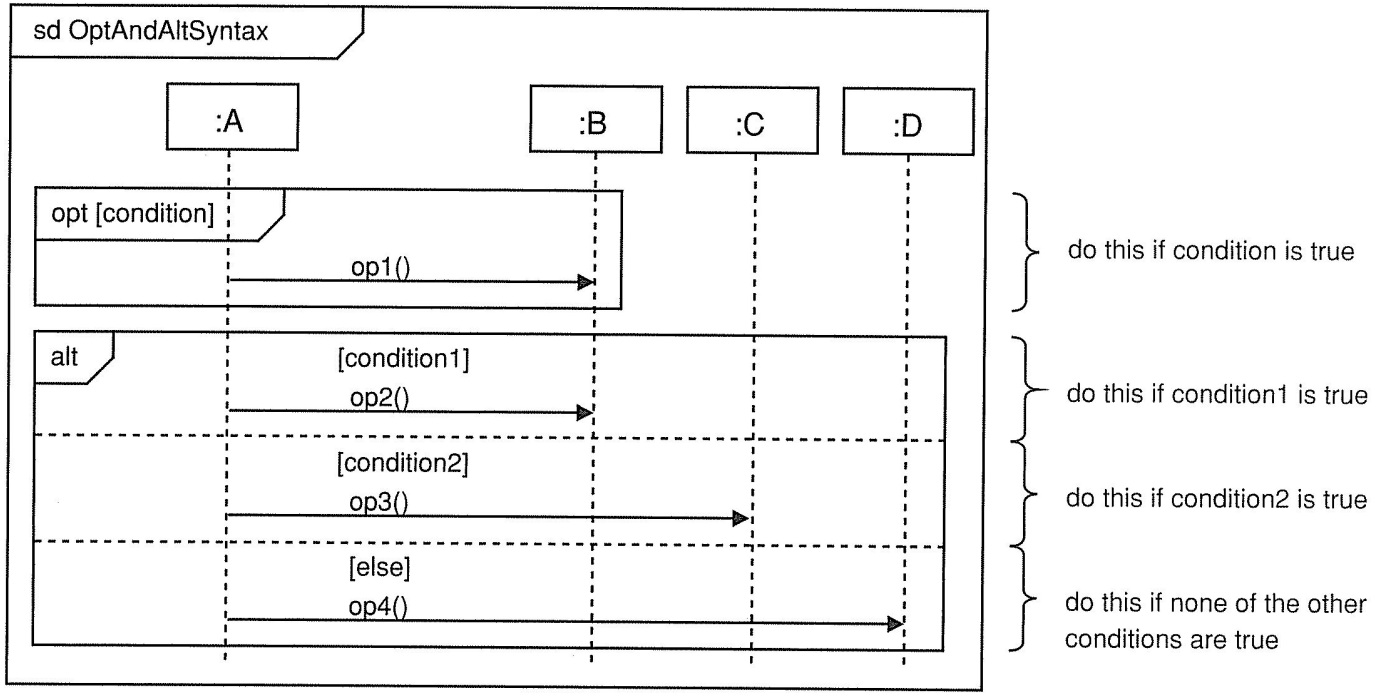
Alternative (alt) creates multiple mutually exclusive branches (as in a switch)

- The alternative to execute depends on the value of the guard
  - The guard "else" is supported - its operand executes only if none of the other guard conditions is true



Option (opt) creates a single branch

- Special case of an alternative where only one of the operands may execute
  - Similar to an if… then, without any else

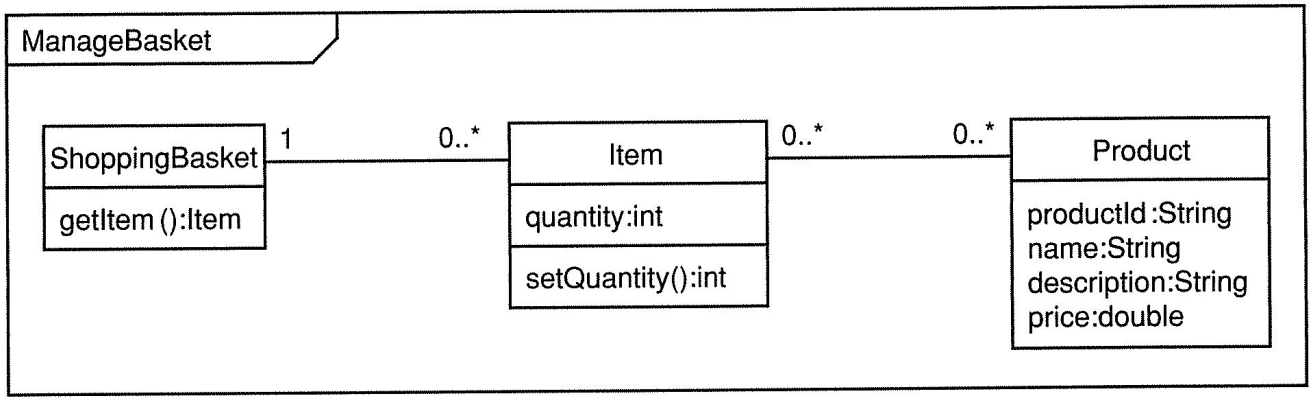


Example of using (opt) and (alt)

Use case: ManageBasket
ID: 2
Brief description: The Customer changes the quantity of an item in the basket.
Primary actors: Customer
Secondary actors: None.
Preconditions: 1. The shopping basket contents are visible.
Main flow: 1. The use case starts when the Customer selects an item in the basket. 2. If the Customer selects "delete item" 2.1 The system removes the item from the basket. 3. If the Customer types in a new quantity 3.1 The system updates the quantity of the item in the basket.
Postconditions: None.
Alternative flows: None.

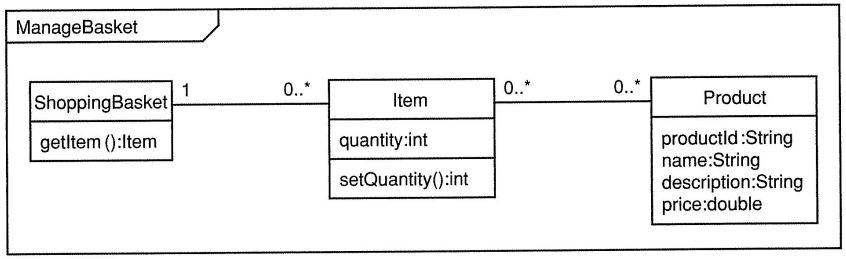
Analysis class diagram for this use case

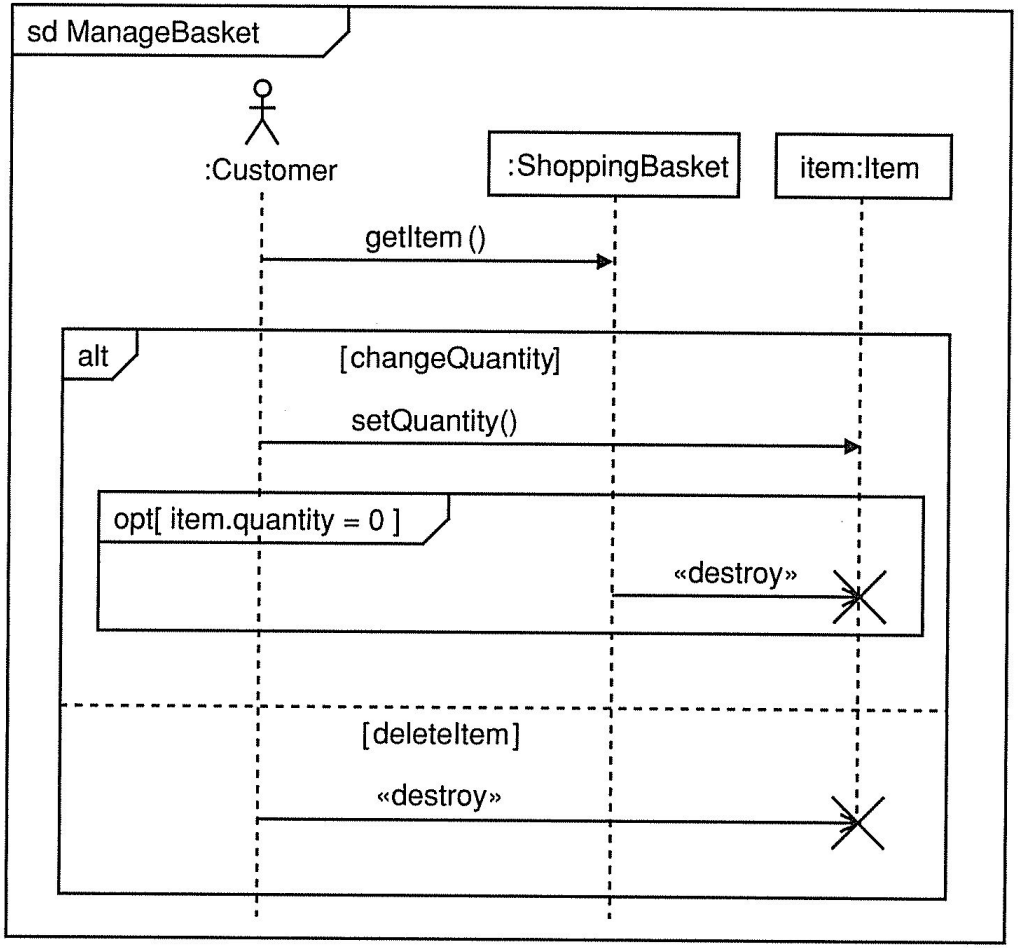
Use case: ManageBasket
ID: 2
Brief description: The Customer changes the quantity of an item in the basket.
Primary actors: Customer
Secondary actors: None.
Preconditions: 1. The shopping basket contents are vis
Main flow: 1. The use case starts when the Custor basket. 2. If the Customer selects "delete item" 2.1 The system removes the item from the basket. 3. If the Customer types in a new quantity 3.1 The system updates the quantity of the item in the basket.
Postconditions: None.
Alternative flows: None.



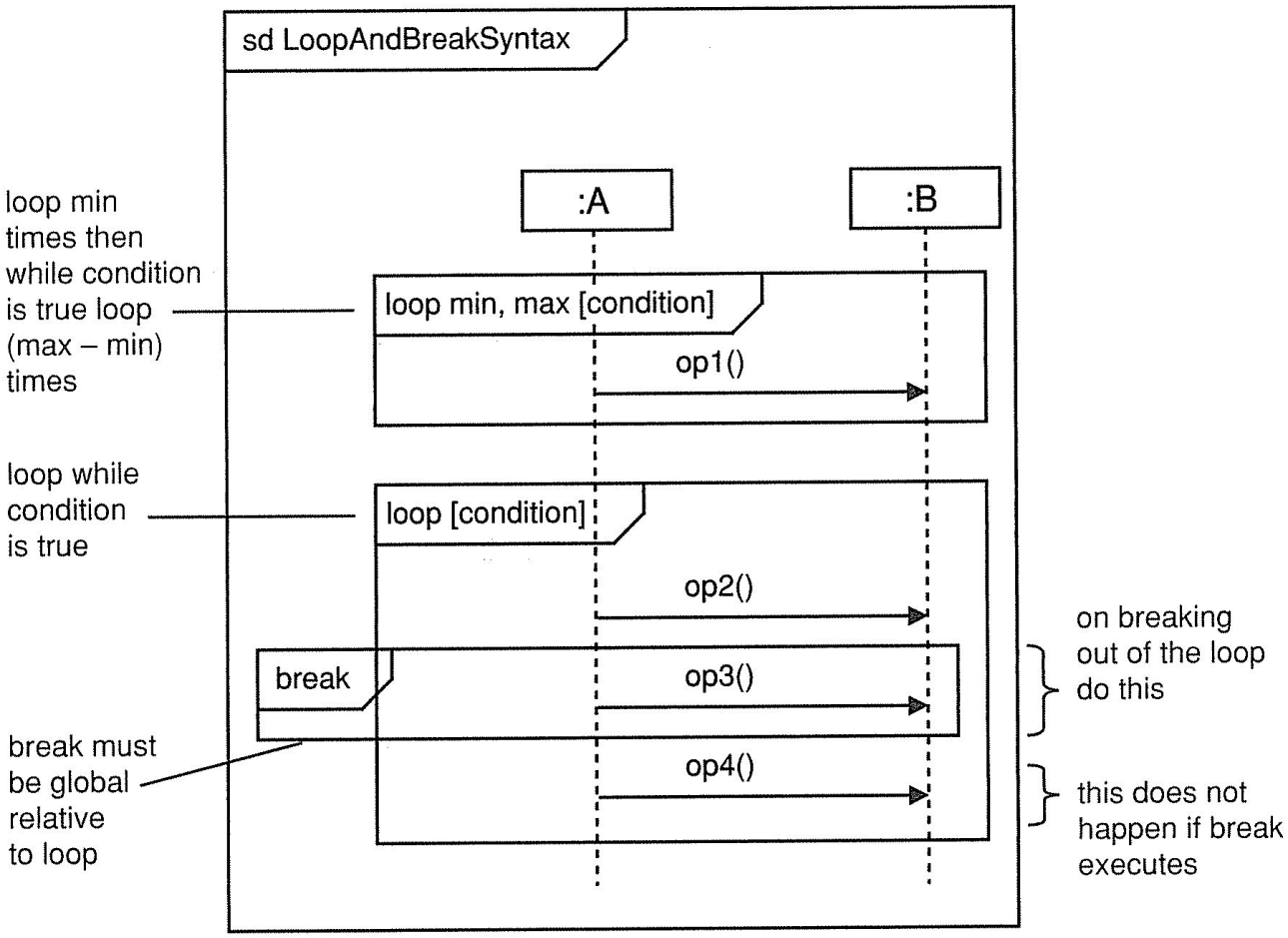
Sequence diagram for this use case

Use case: ManageBasket
ID: 2
Brief description: The Customer changes the quantity of an item in the basket.
Primary actors: Customer
Secondary actors: None.
Preconditions: 1. The shopping basket contents are visible.
Main flow: 1. The use case starts when the Customer selects an item in the basket. 2. If the Customer selects "delete item" 2.1 The system removes the item from the basket. 3. If the Customer types in a new quantity 3.1 The system updates the quantity of the item in the basket.
Postconditions: None.
Alternative flows: None.





loop & break syntax



Iteration (loop)

- Optional guard: [<min>, <max>, <Boolean-expression>]
  - **loop min, max [condition]** works as follows

    loop min times

    while the condition is true

    loop (max – min) times
- A loop without max, min, or a condition is an infinite loop
- If only min is given, then max = min
- The condition is usually a Boolean expression, but may also be arbitrary text, provided the intent is clear
  - In this case, automatic code generation will be harder… :-(

Common loop idioms

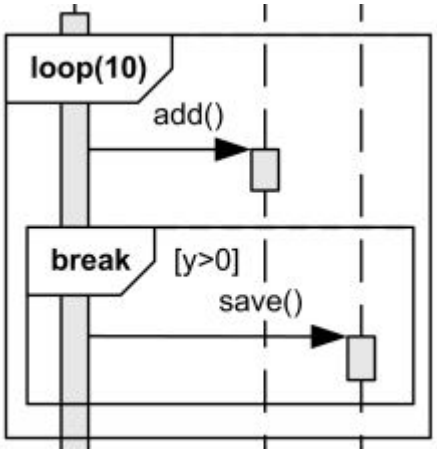
Type of loop	Semantics	Loop expression
while(true) { body }	Keep looping forever	loop or loop *
for i = n to m { body }	Repeat (m – n) times	loop n, m
while(booleanExpression) { body }	Repeat while booleanExpression is true	loop [booleanExpression]

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

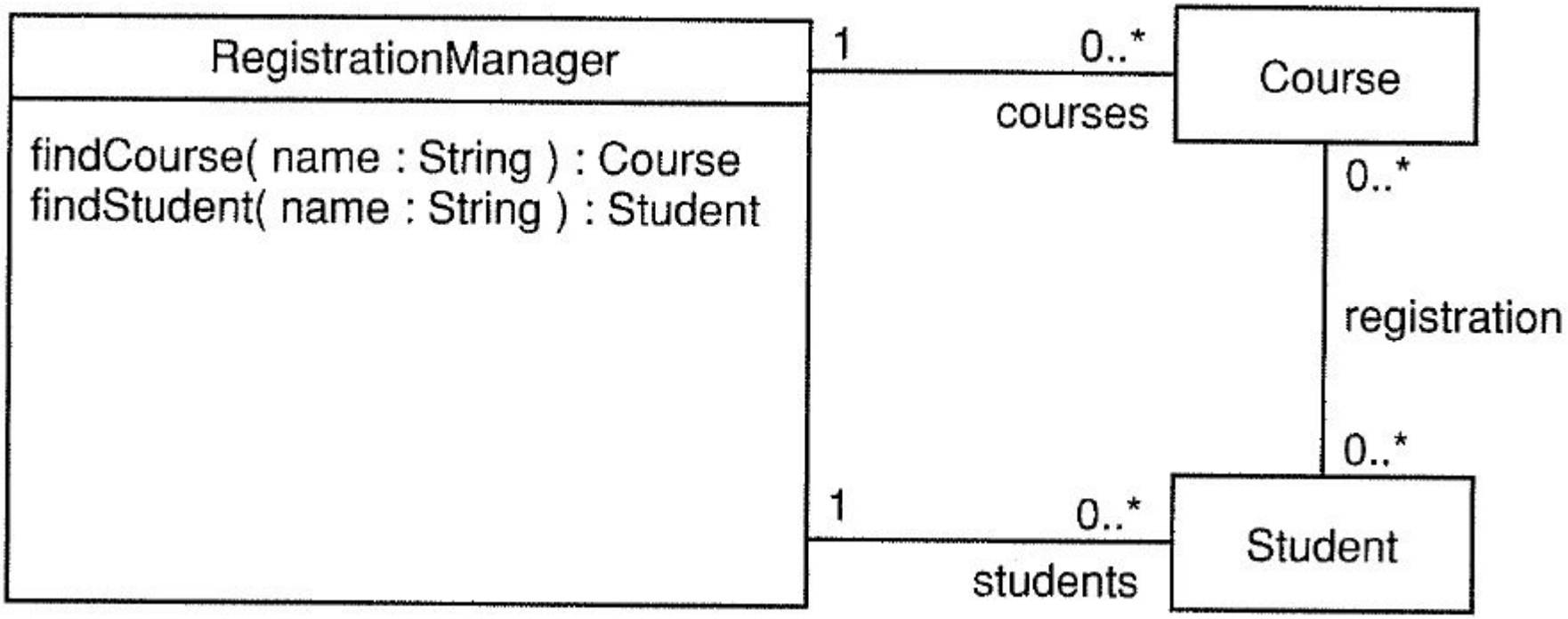
forEach object in collection { body }	Execute the body of the loop once for each object in a collection of objects	loop [for each object in collectionOfObjects]
forEach object of class { body }	Execute the body of the loop once for each object of a particular class	loop [for each object in ClassName]

Interruption (break)

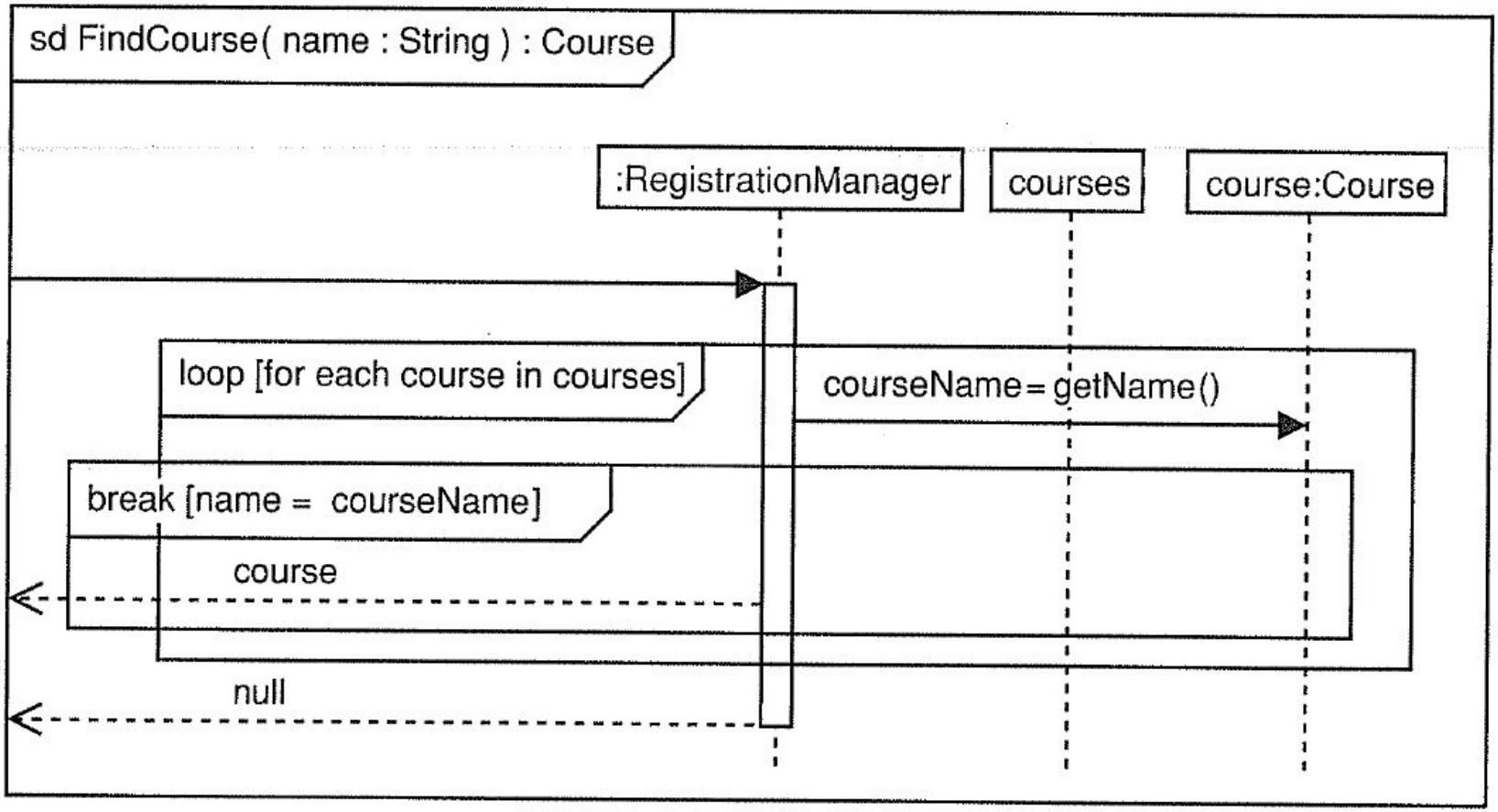
- When the break guard condition evaluates to true, the break operand executes and the loop terminates
- The break represents an alternative which is executed in turn of the rest of the fragment
  - Similar to a break in a cycle
  - **The rest of the loop after the break does not execute!**
    - Example: break enclosing loop if y > 0



loop and break example



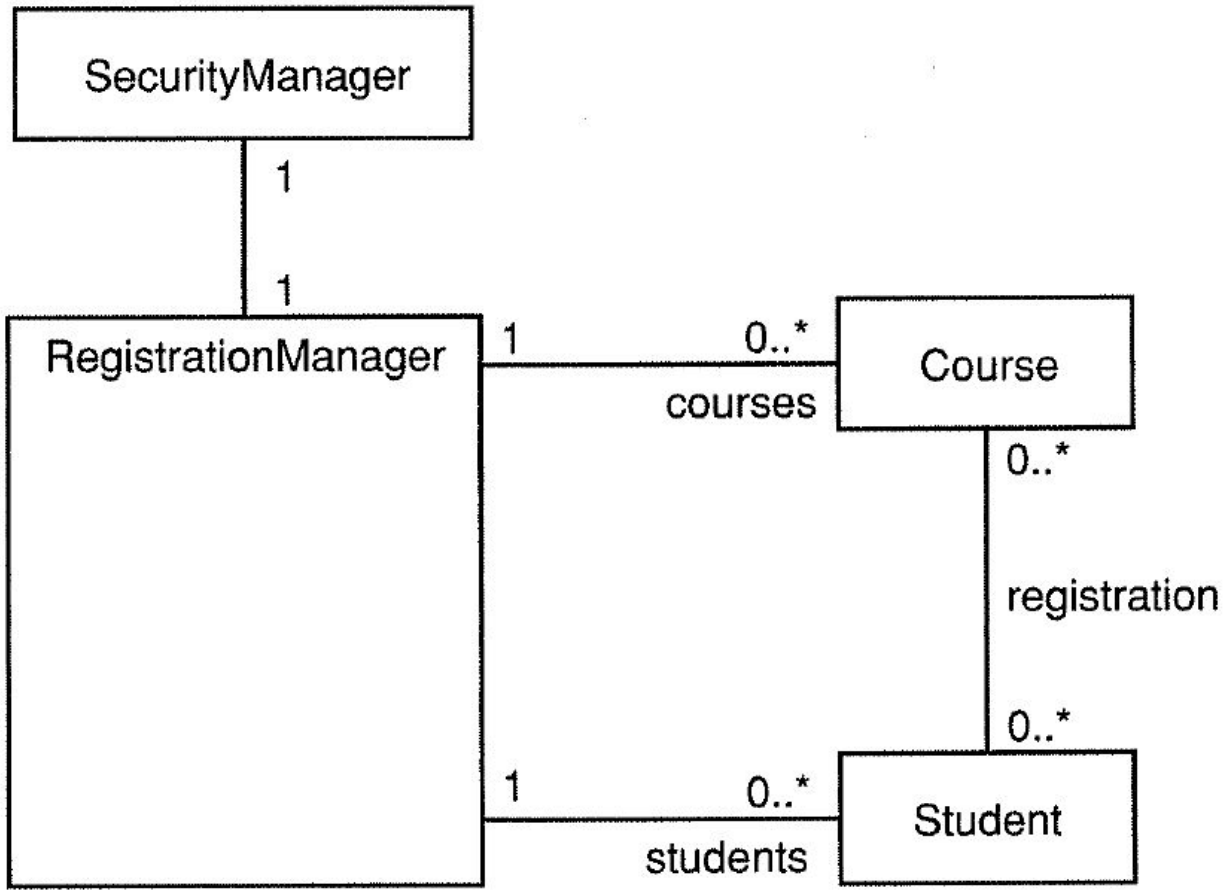
loop over a collection to find a specific element



Interaction occurrences

- Sometimes, a particular sequence of sent message occurs very often
- Instead of drawing that sequence over and over again, we should reuse it
  - drawing the same sequence over and over again is tiresome and error-prone
- Interaction occurrences are **references to another interaction**
- When an interaction occurrence is placed into an interaction, the flow of the interaction it references is included at that point

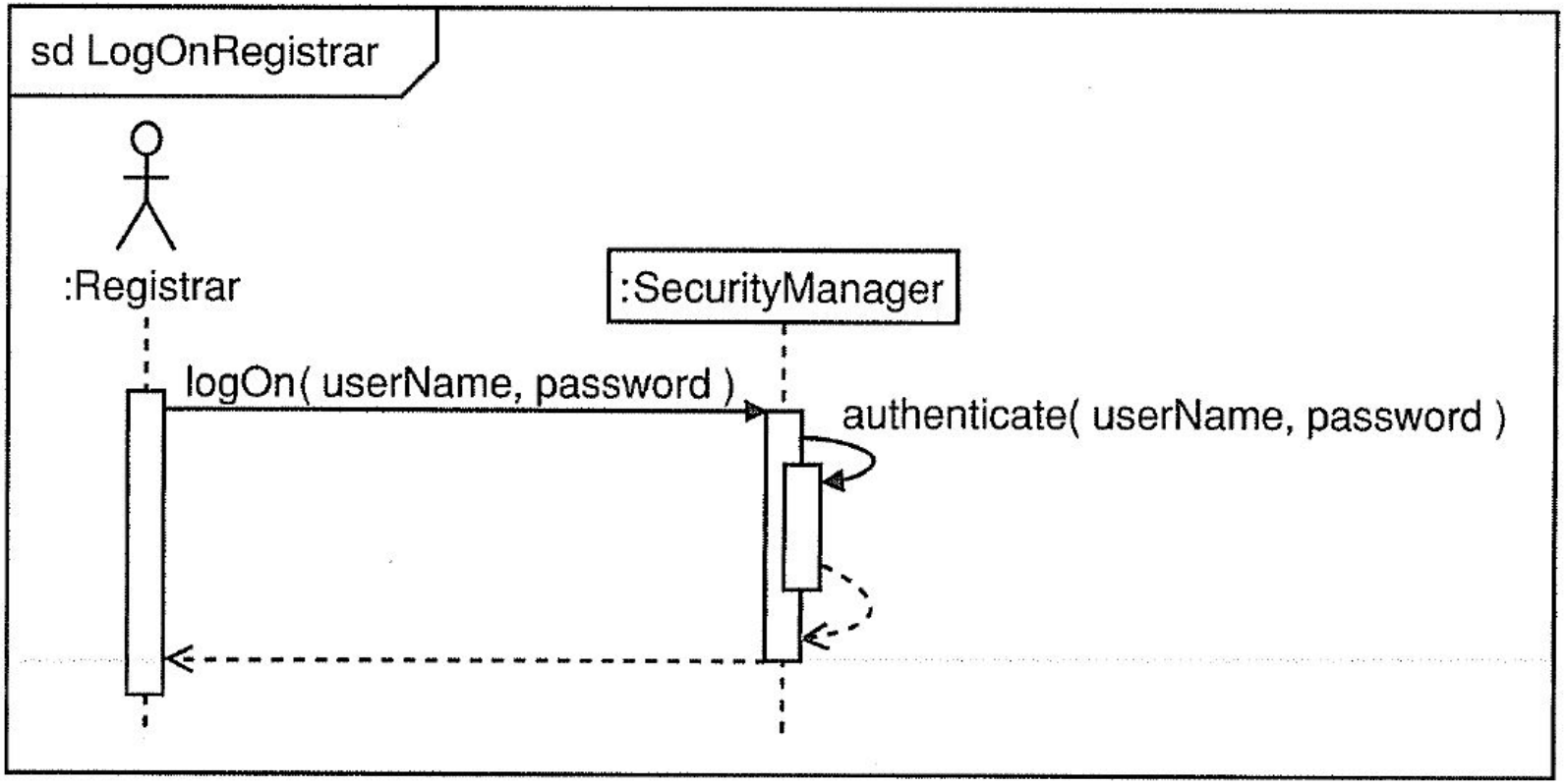
Consider the following analysis class diagram



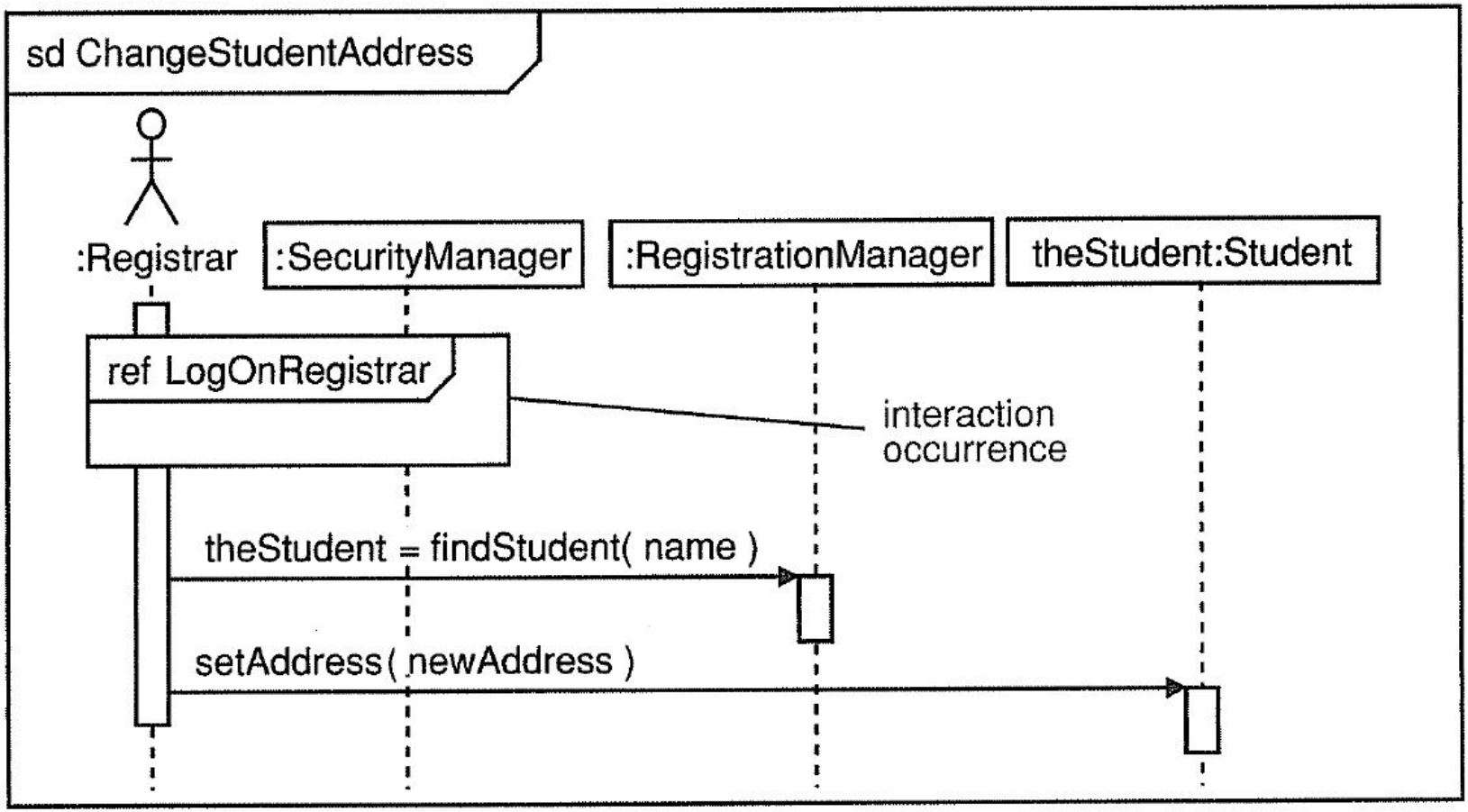
… and the LogOnRegistrar use case

Use case: LogOnRegistrar
ID: 4
Brief description: The Registrar logs on to the system.
Primary actors: Registrar
Secondary actors: None.
Preconditions: 1. The Registrar is not logged on to the system.
Main flow: 1. The use case starts when the Registrar selects "log on". 2. The system asks the Registrar for a user name and password. 3. The Registrar enters a user name and password. 4. The system accepts the user name and password as valid.
Postconditions: 1. The Registrar is logged on to the system.
Alternative flows: InvalidUserNameAndPassword RegistrarAlreadyLoggedOn

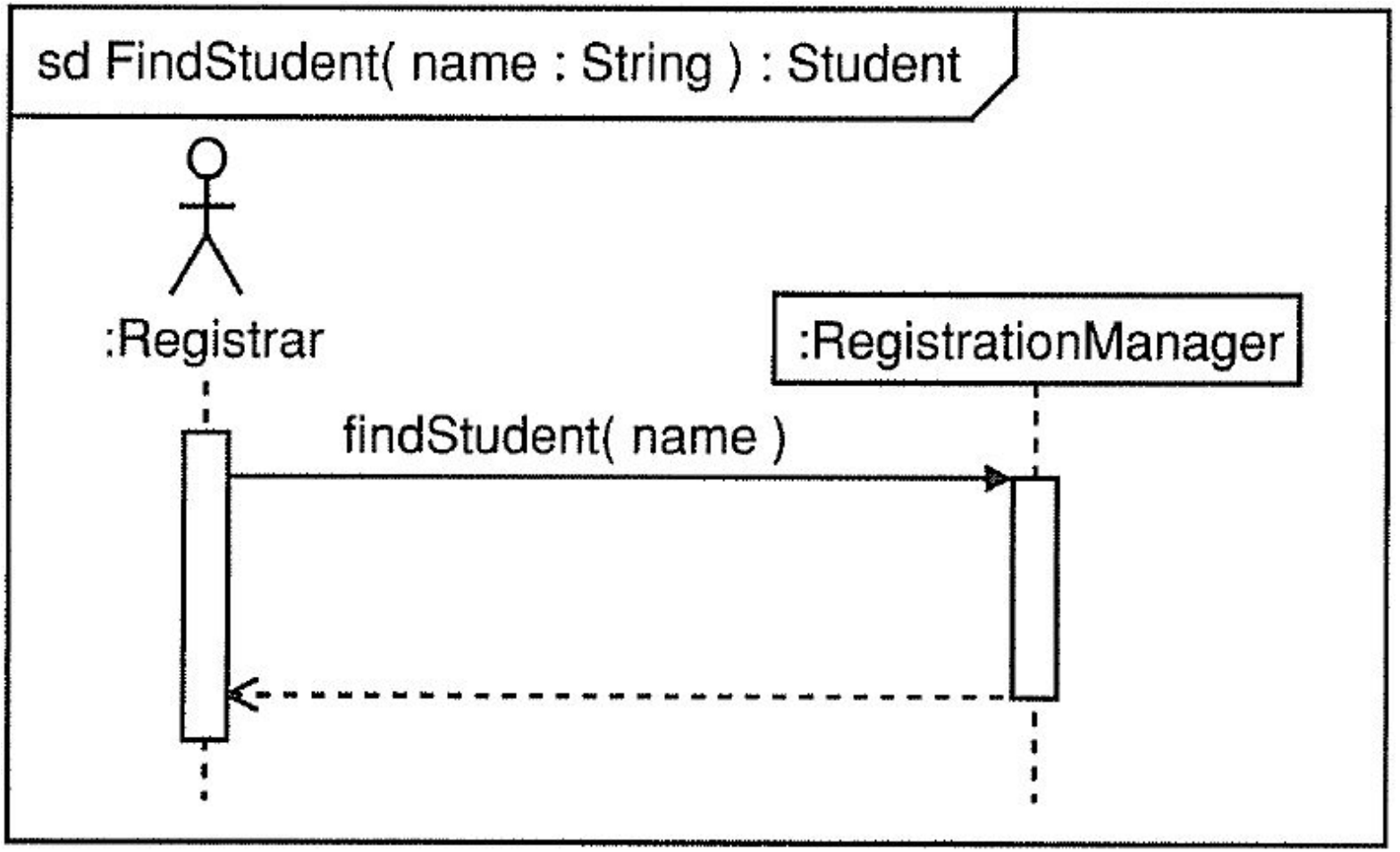
The use case could be refined into the LogOnRegistrar sequence diagram



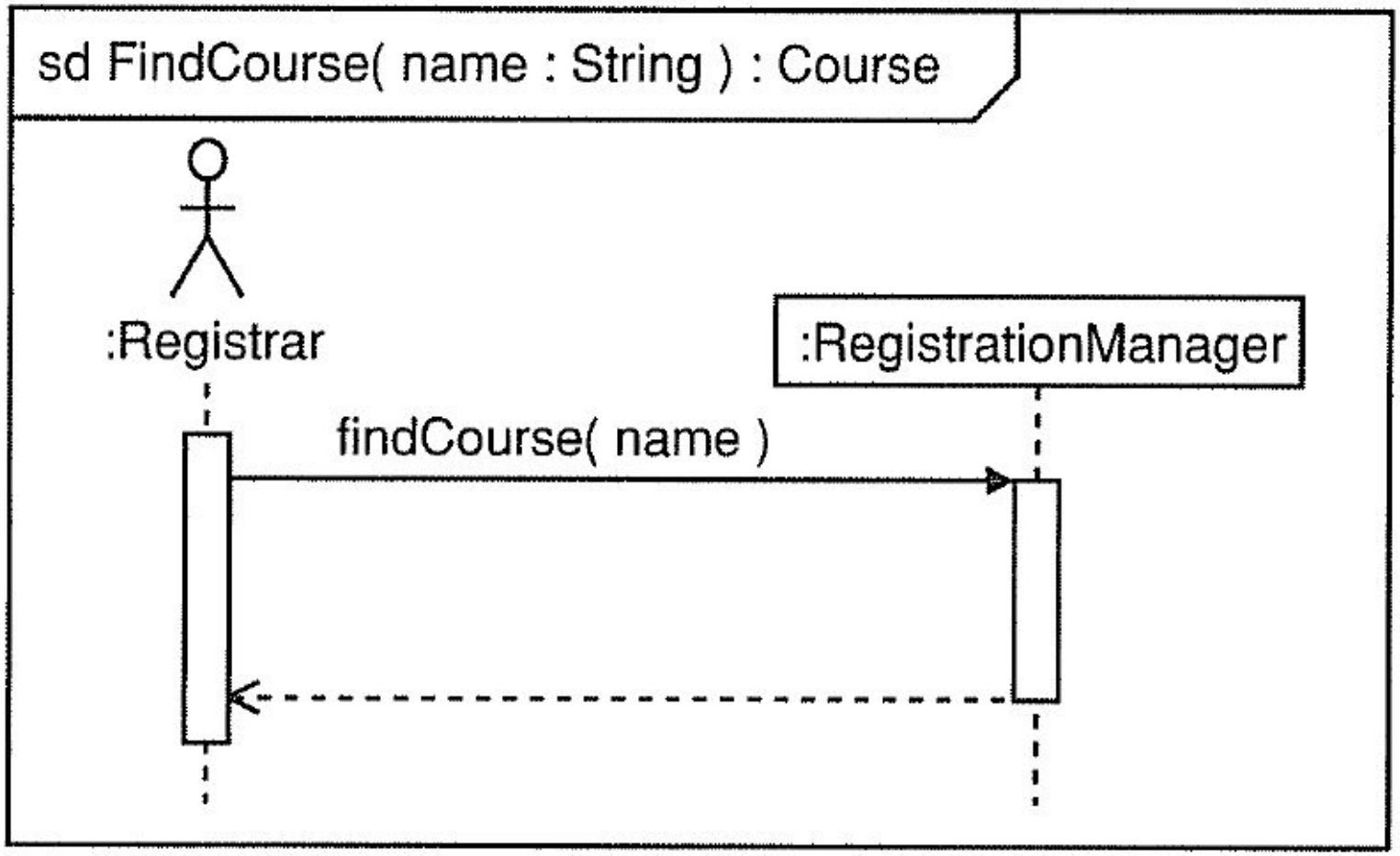
For a lot of much more interesting use cases, we can now reference the LogOnRegistrar



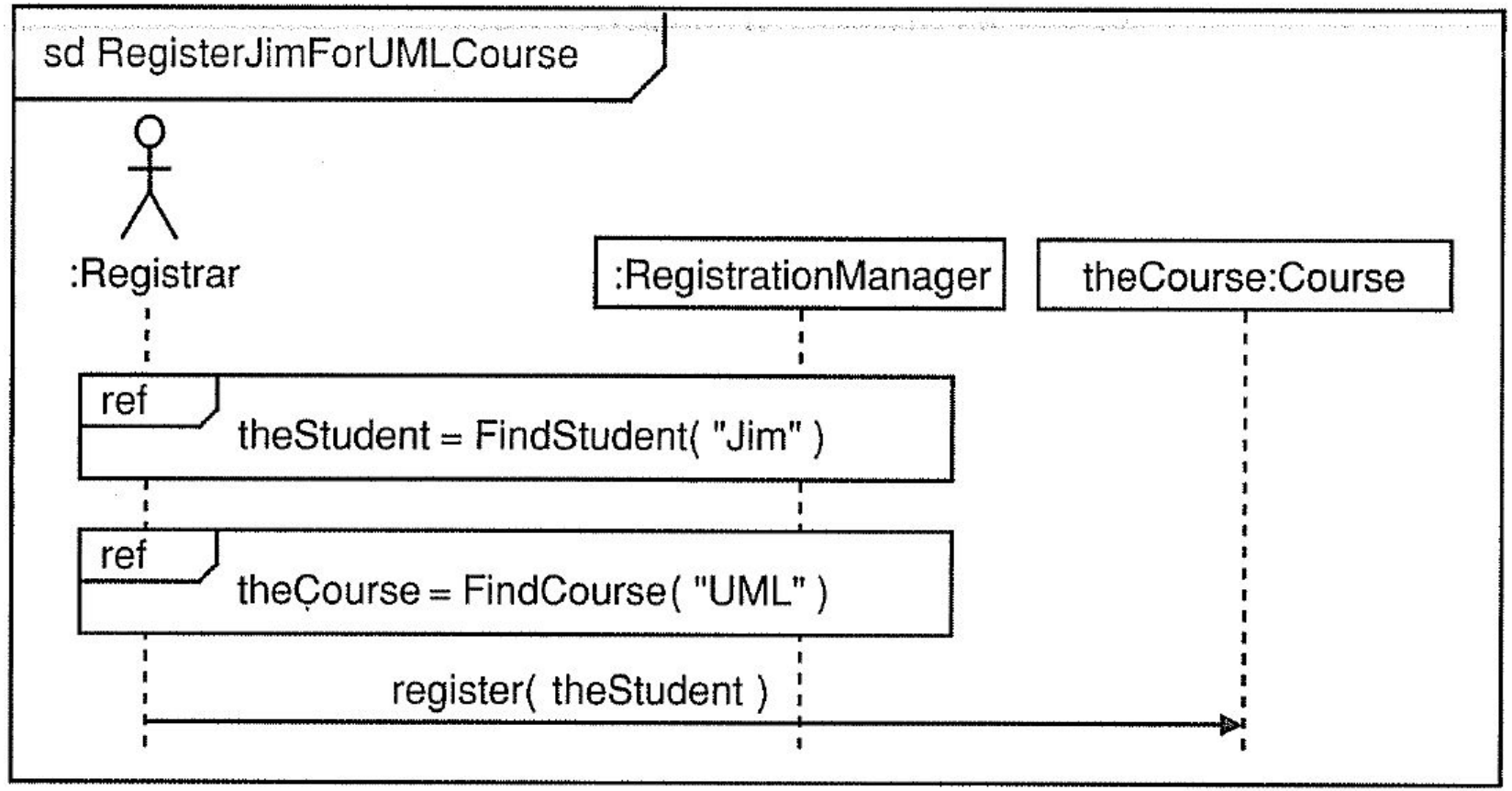
We can use this to create reusable sequence diagrams for common interactions



We can use this to create reusable sequence diagrams for common interactions

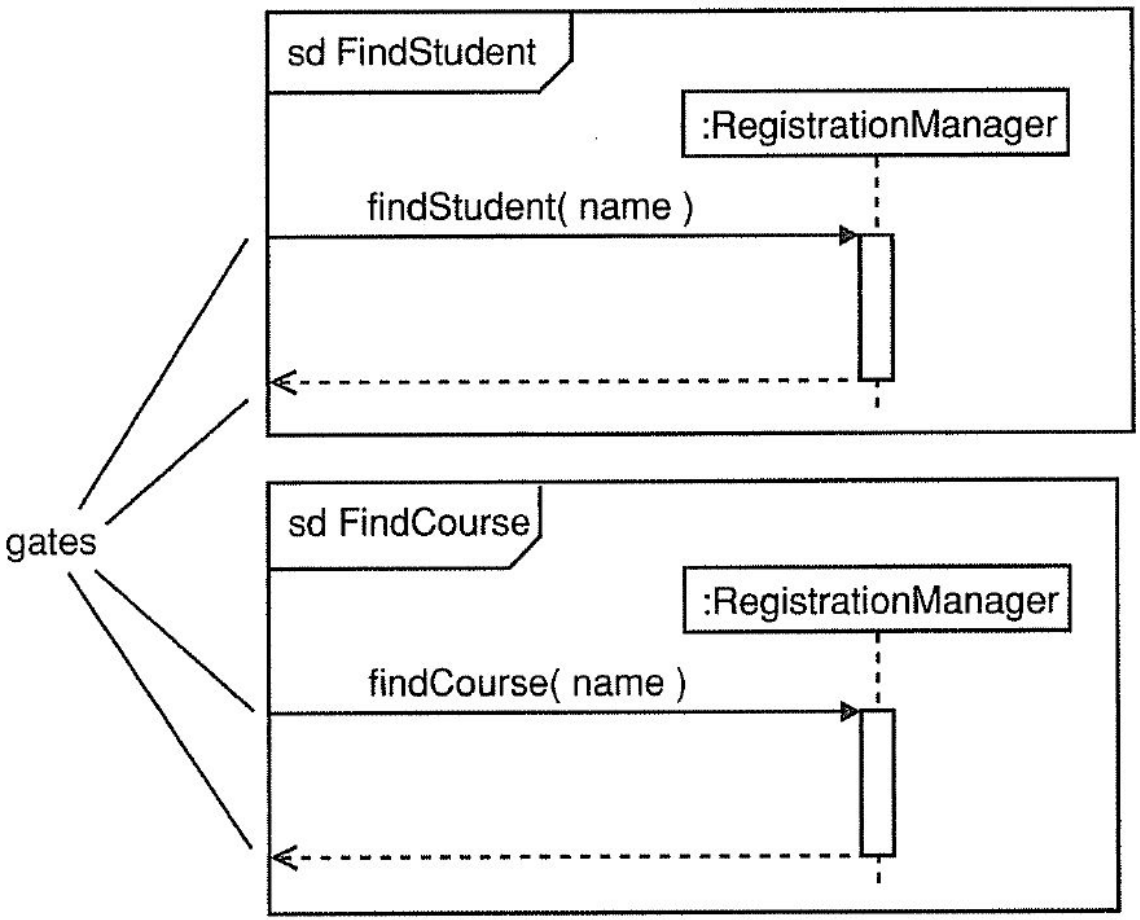


We can use this to create reusable sequence diagrams for common interactions

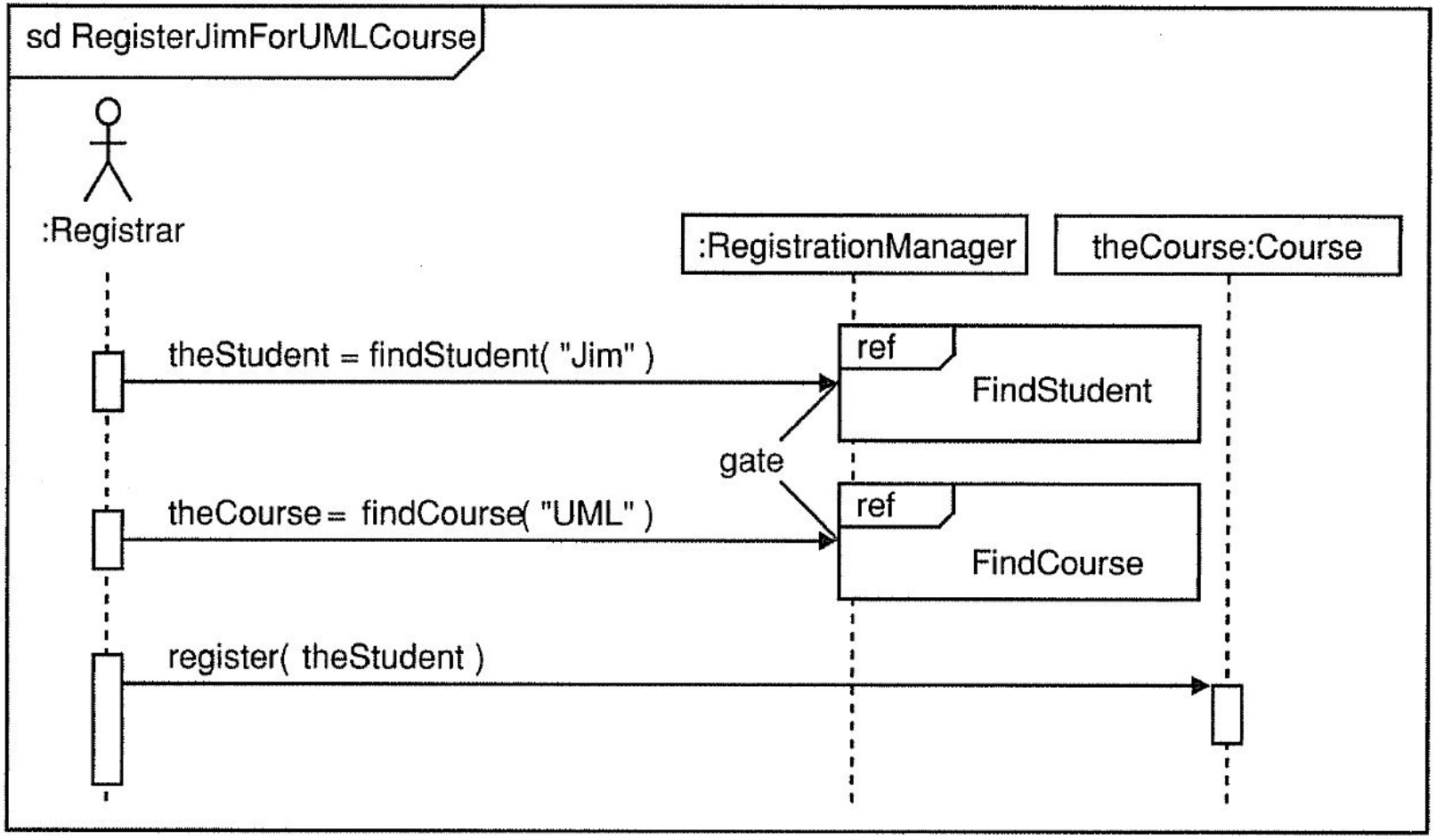


Gates are inputs and outputs of interactions

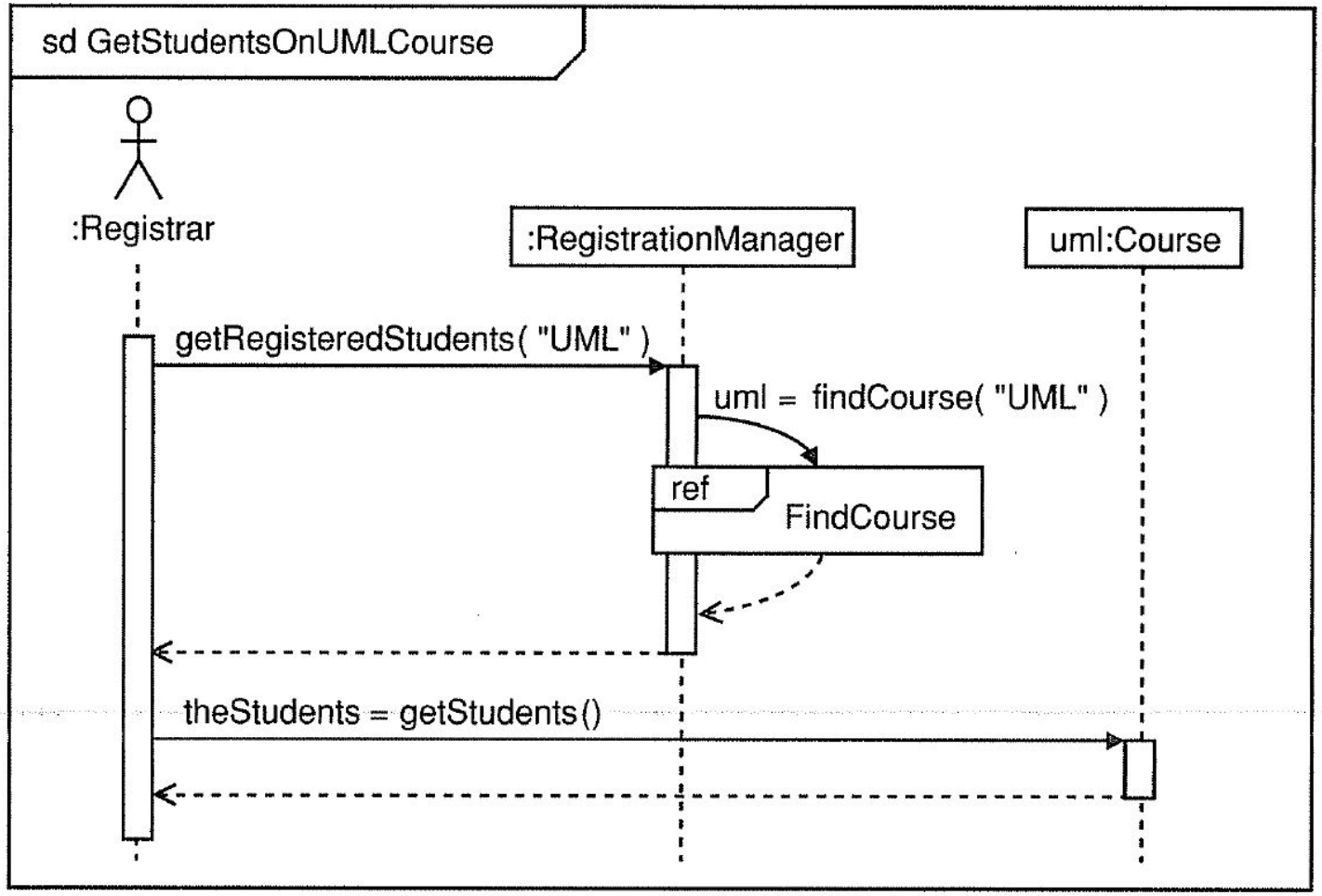
Use a gate when you want an interaction to be triggered by a lifeline that is not part of the interaction



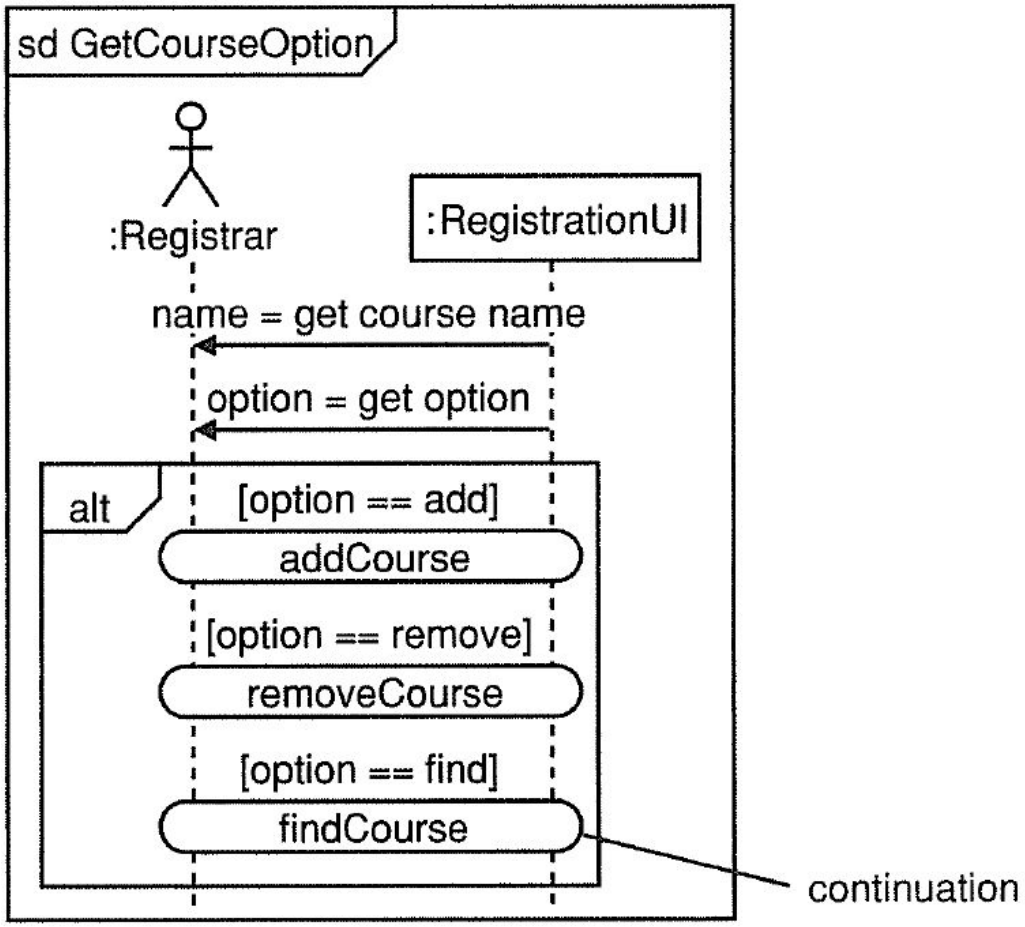
Gates provide added flexibility in the reuse of interactions: use them when some of the messages arrive from the output and you don't know in advance where they might come from



Gates can also be used in messages to the same lifeline

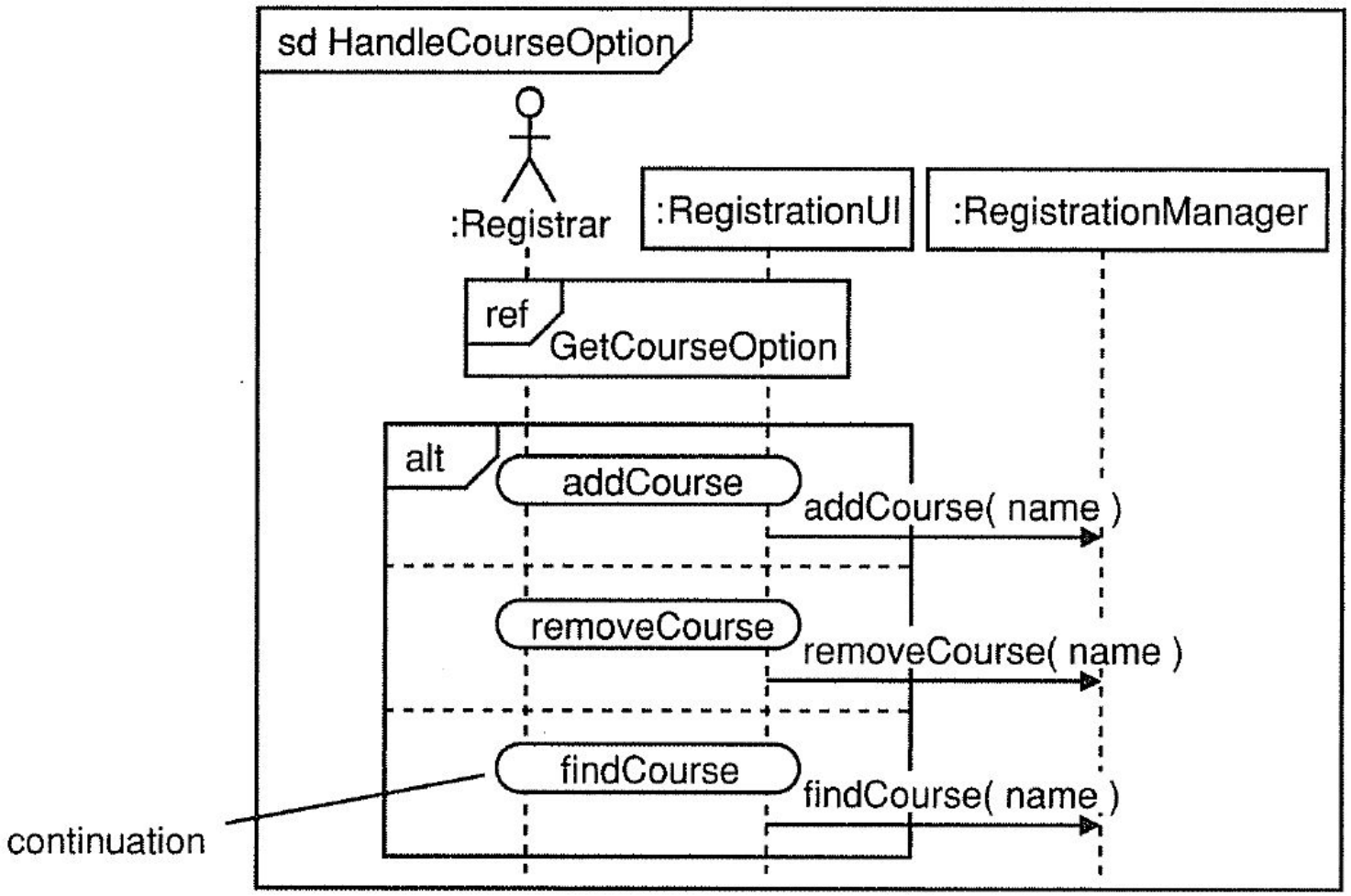


Continuations allow an interaction fragment to terminate in such a way it can be continued by another fragment



Depending on which option is selected, the interaction terminates at one of the three continuations

You can use continuations to decouple interactions (here, GetCourseOption from HandleCourseOption)



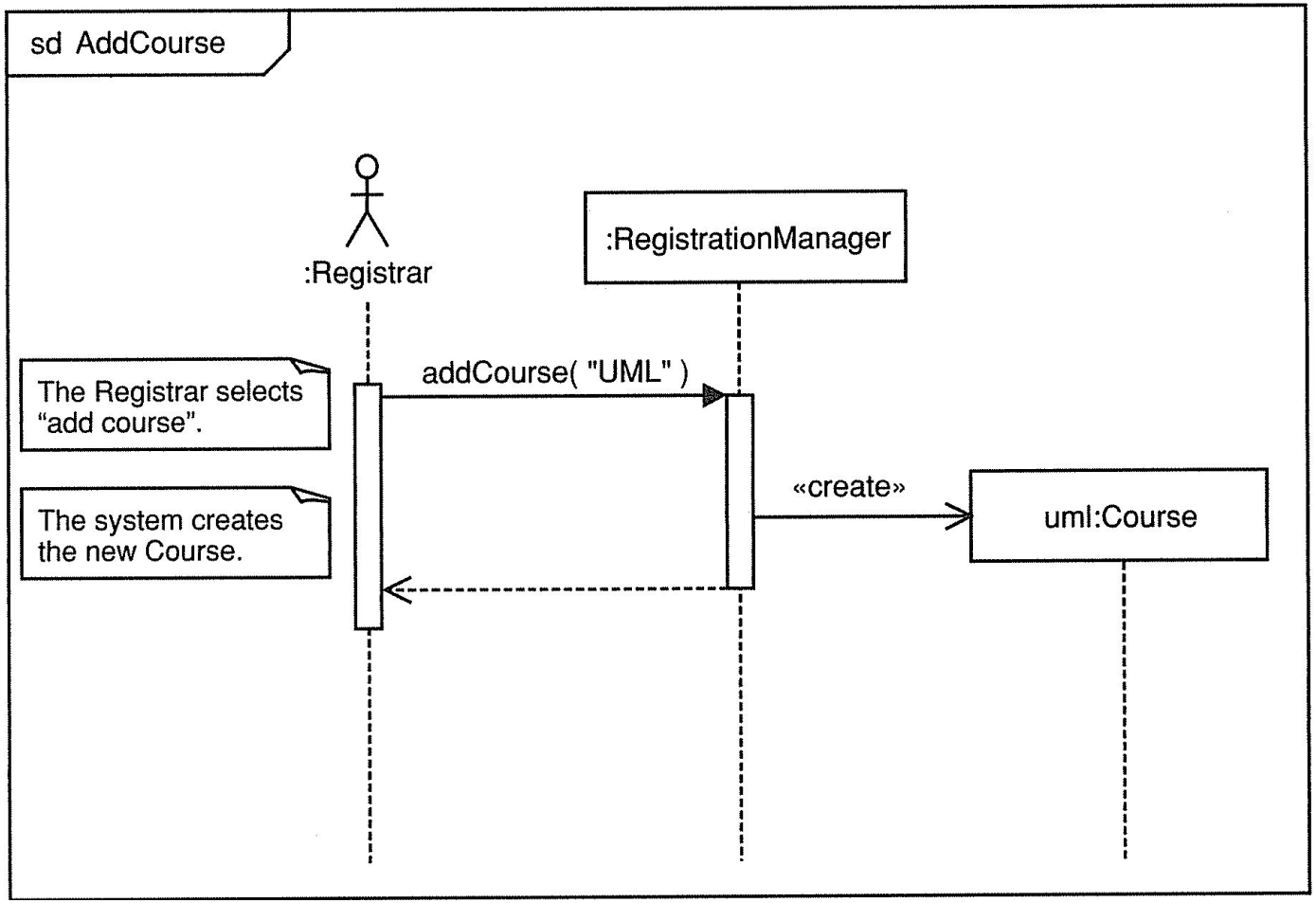
Interaction diagrams at the design level

- In design we may refine key analysis interaction diagrams or create new ones to illustrate central mechanisms such as object persistence
- You may introduce new mechanisms such as
  - Object persistence
  - Object distribution
  - Transactions
  - ...

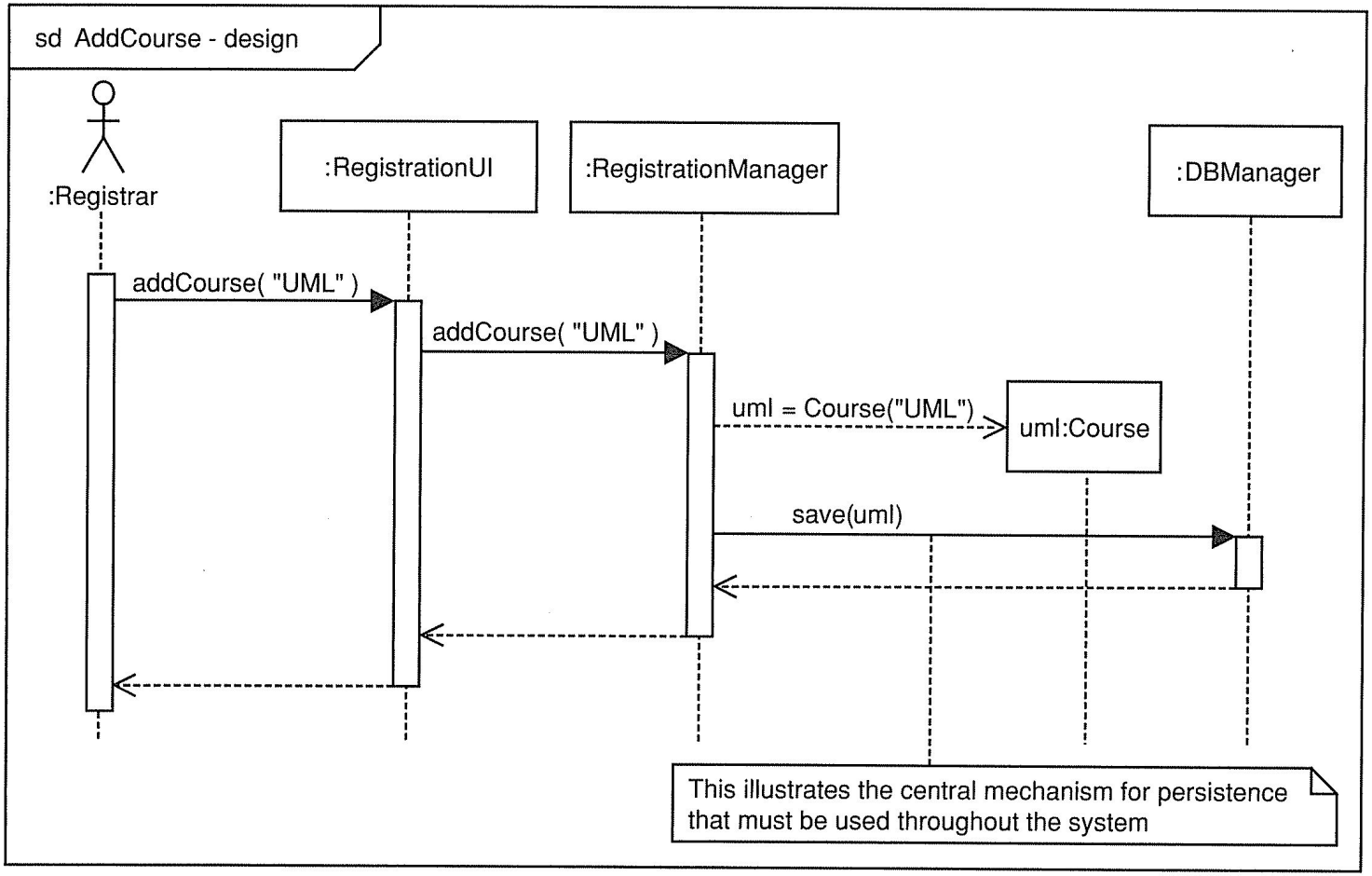
Consider again the addCourse use case

Use case: AddCourse
ID: 8
Brief description: Add details of a new course to the system.
Primary actors: Registrar
Secondary actors: None.
Preconditions: 1. The Registrar has logged on to the system.
Main flow: 1. The Registrar selects "add course". 2. The Registrar enters the name of the new course. 3. The system creates the new course.
Postconditions: 1. A new course has been added to the system.
Alternative flows: CourseAlreadyExists

We could create an analysis sequence diagram for it

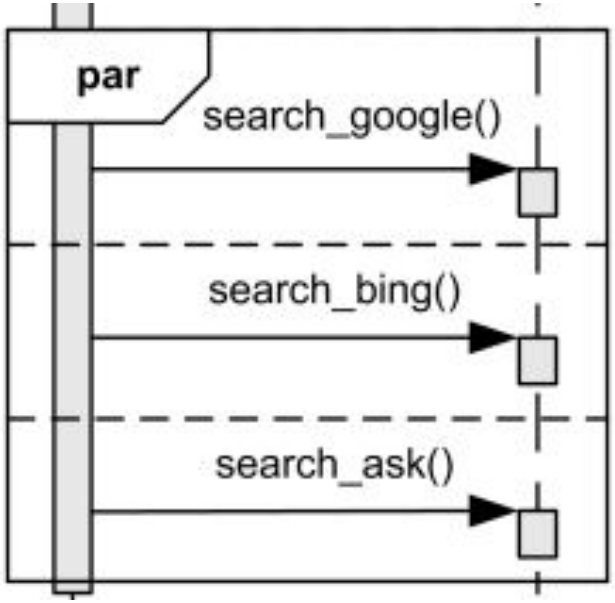


We can refine it to include a UI lifeline (the interface) and a controller lifeline (RegistrationManager)



Parallel (par)

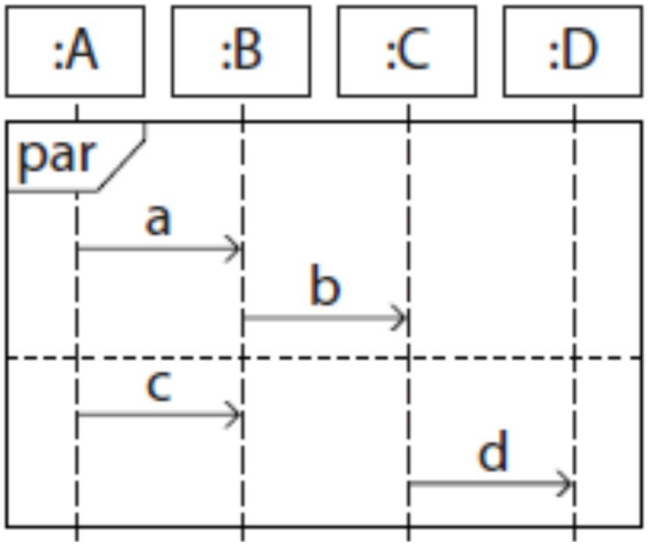
- All the operands execute concurrently (interleaved)
  - Search Google, Bing and Ask in any order, possibly in parallel



Parallel (par)

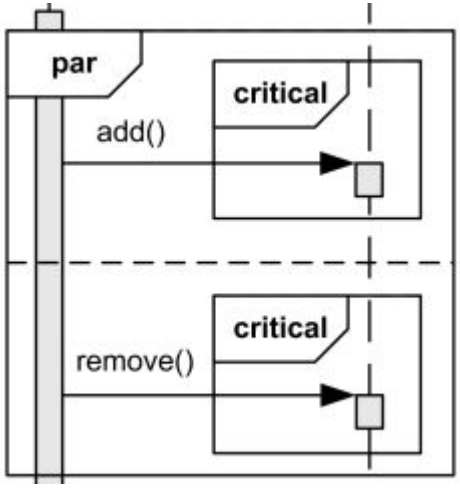
- All the operands execute concurrently (interleaved)
  - a->b->c->d
  - a->b->d->c
  - a->c->b->d
  - a->c->d->b
  - a->d->b->c
  - a->d->c->b
  - c->a->b->d
  - c->a->d->b
  - c->d->a->b
  - d->a->b->c
  - d->a->c->b
  - d->c->a->b

  (any sequence where b is **after** a, really)



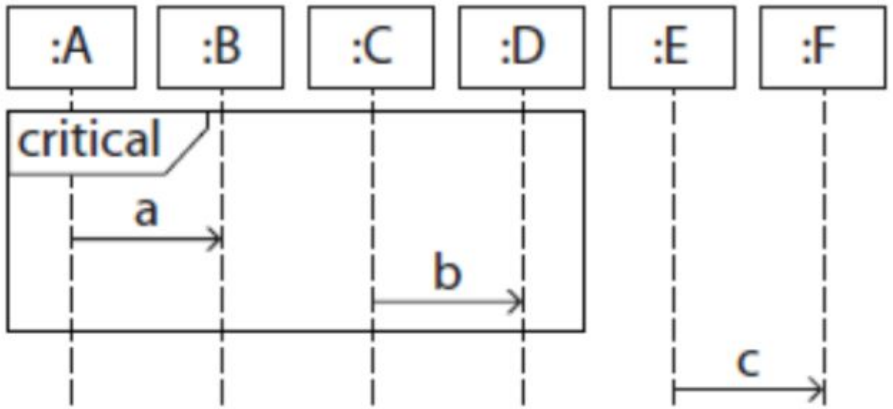
Critical region (critical)

- The execution traces may not be interleaved with events of other lifelines
  - add() or remove() could be called in parallel, but each should be run as a critical region



Critical region (critical)

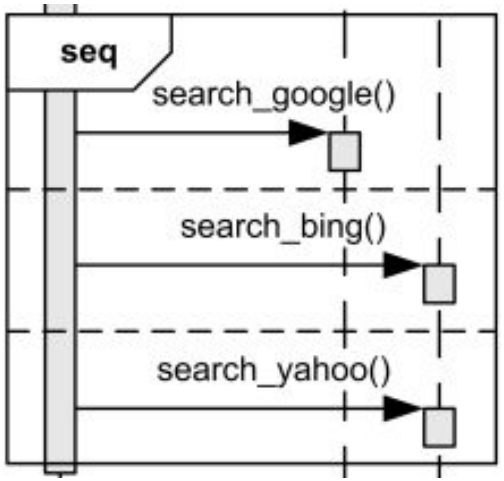
- The execution traces may not be interleaved with events of other lifelines
  - The execution is performed without any interruption
    - a->b->c
    - b->a->c
    - c->a->b
    - c->b->a



(any sequence where the sub-sequences a->b, or b->a, are not interrupted)

Weak sequence (seq)

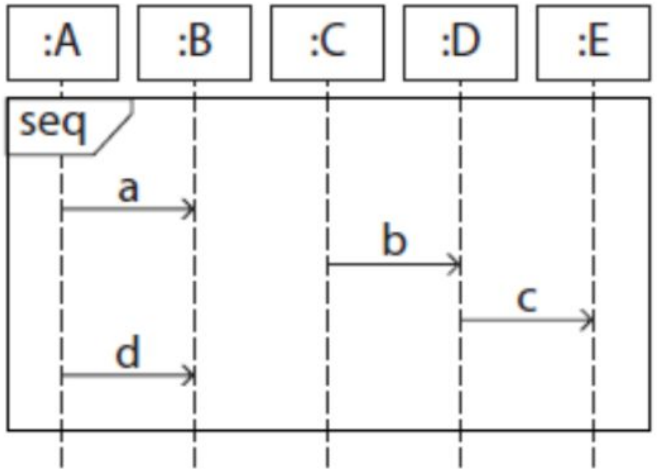
- The operands execute in parallel in the different lifelines, with a restriction: events received in the same lifeline, originated by different operands, occur in the same sequence as the one of the operands
  - Search google, possibly in parallel with Bing and Yahoo, but search bing before yahoo (because they are in the same lifeline)



Weak sequence (seq)

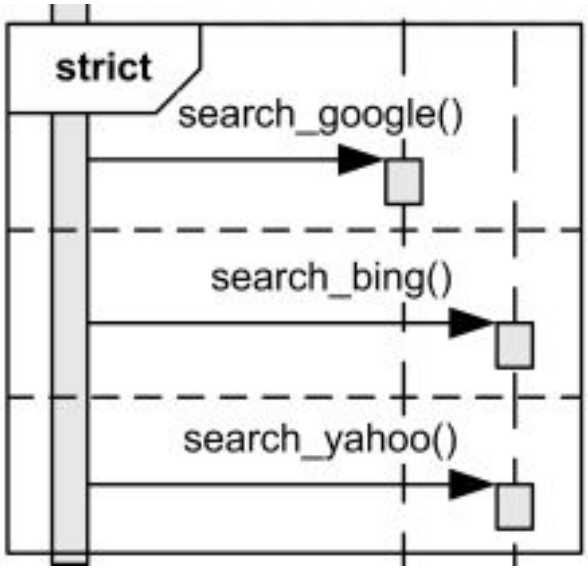
- The operands execute in parallel in the different lifelines, with a restriction: events received in the same lifeline, originated by different operands, occur in the same sequence as the one of the operands
  - a->b->c->d
  - a->b->d->c
  - a->d->b->c
  - b->c->a->d
  - b->a->c->d
  - b->a->d->c

    (a before d, b before c)



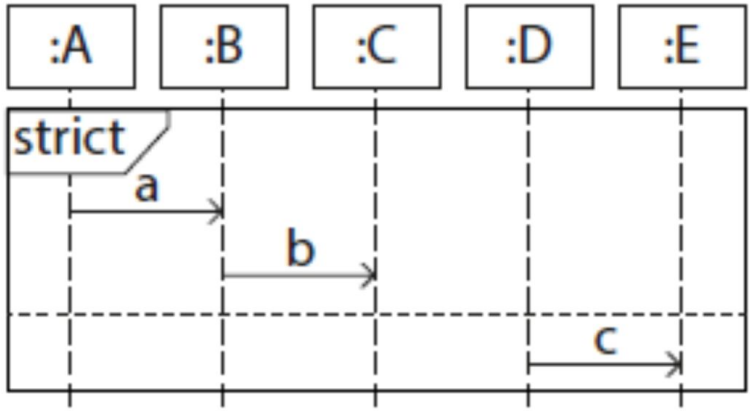
Strong sequence (strict)

- The operands execute in strict sequence
  - Search Google, Bing, and Yahoo, in strict sequential order



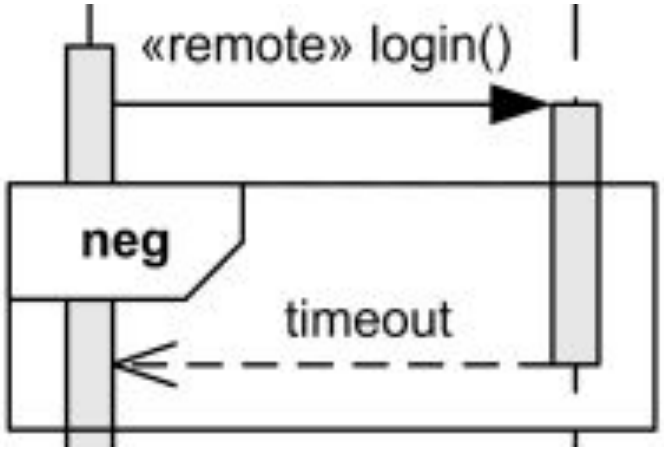
Strong sequence (strict)

- The operands execute in strict sequence
  - a -> b -> c



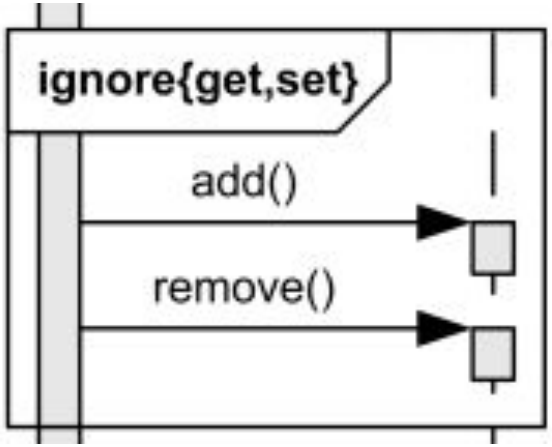
Negative (neg)

- Identifies sequences that may not occur
  - If we receive back a timeout message, this means the system has failed



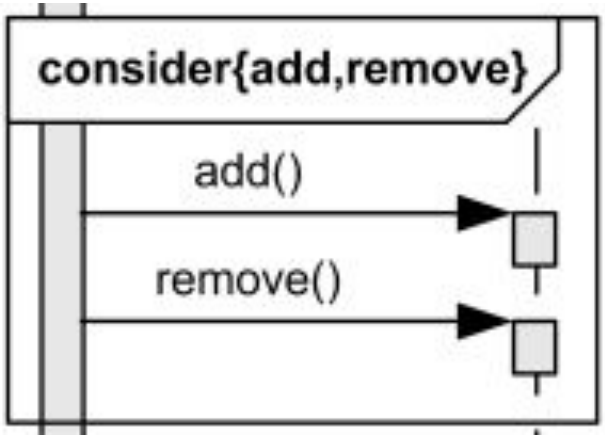
Ignore (ignore)

- List messages that are deliberately omitted
  - Ignore get() and set() messages, if any



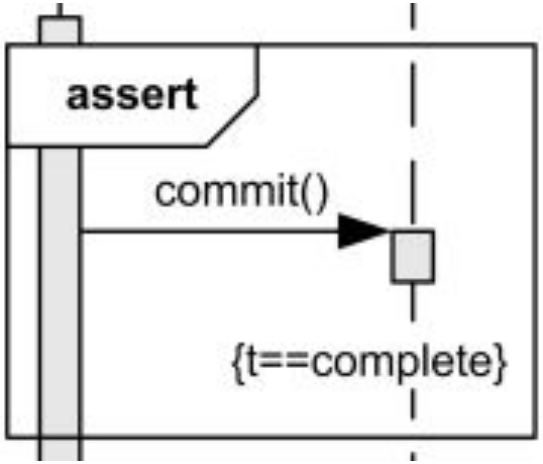
Consider (consider)

- List messages that were intentionally included
  - Consider only add() or remove() messages, ignore any other



Assertion (assert)

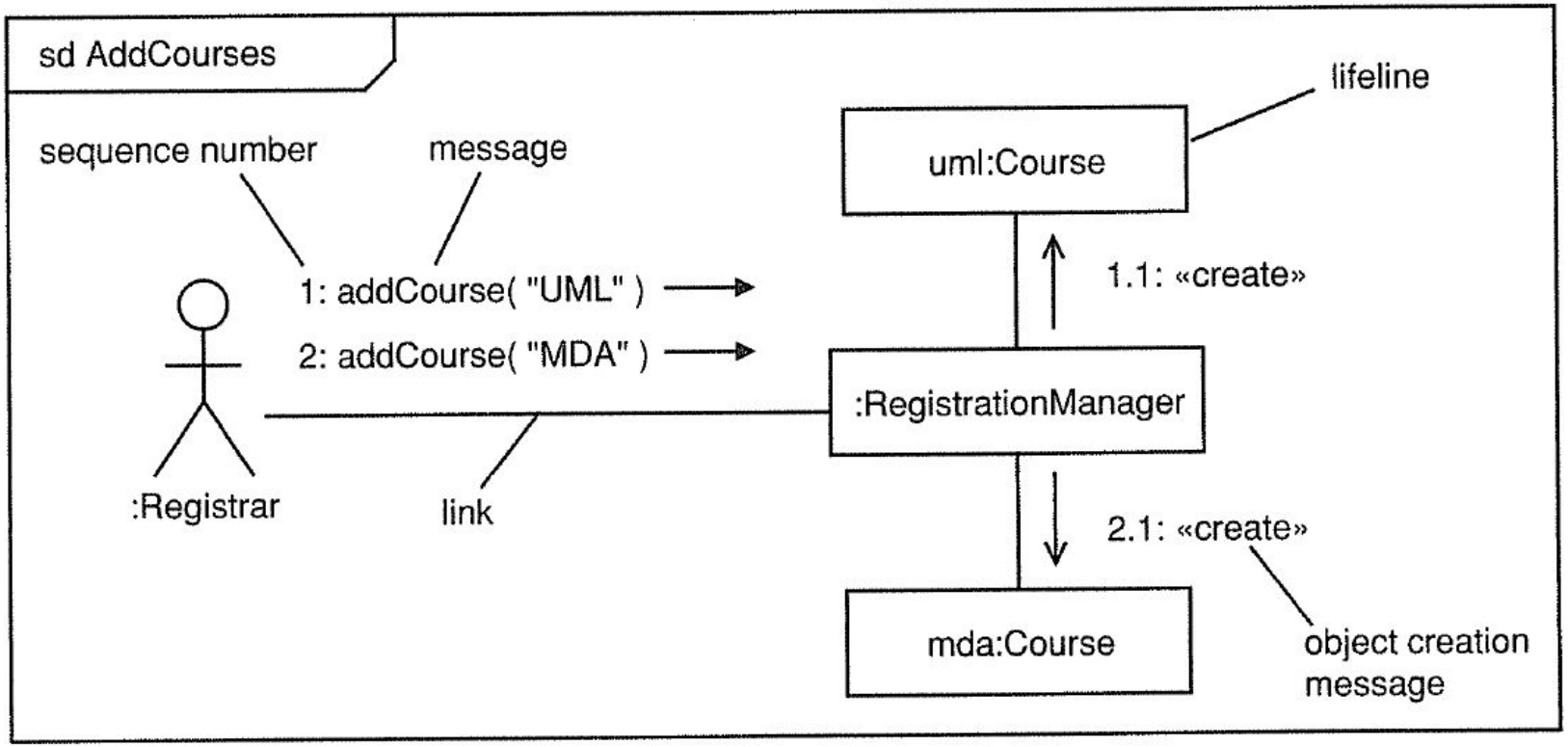
- Represents the only valid behavior in a given point in the interaction
  - commit() message should occur at this point, following with evaluation of state invariant
  - any other continuation would be wrong



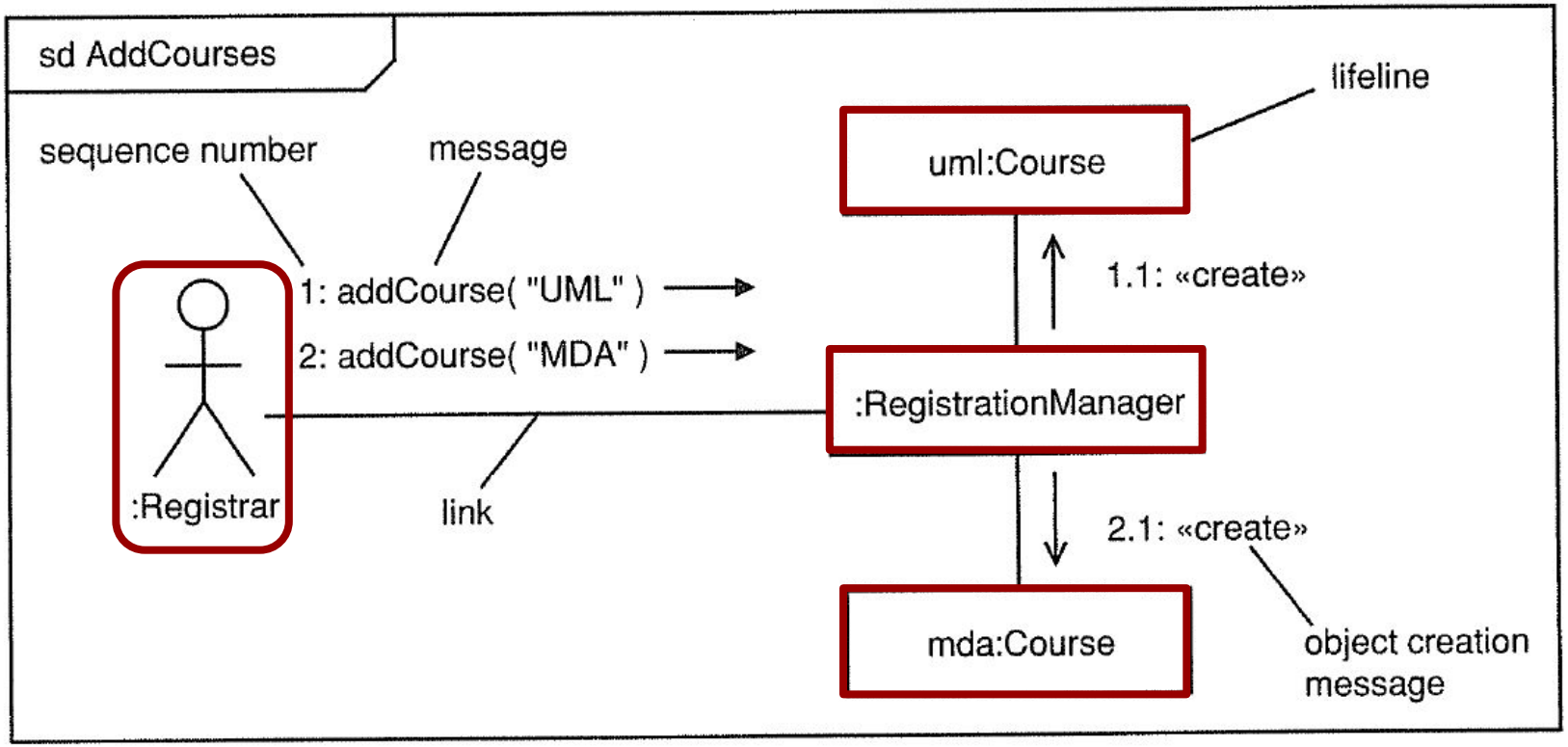
Communication diagrams

Emphasize the structural aspects of an interaction

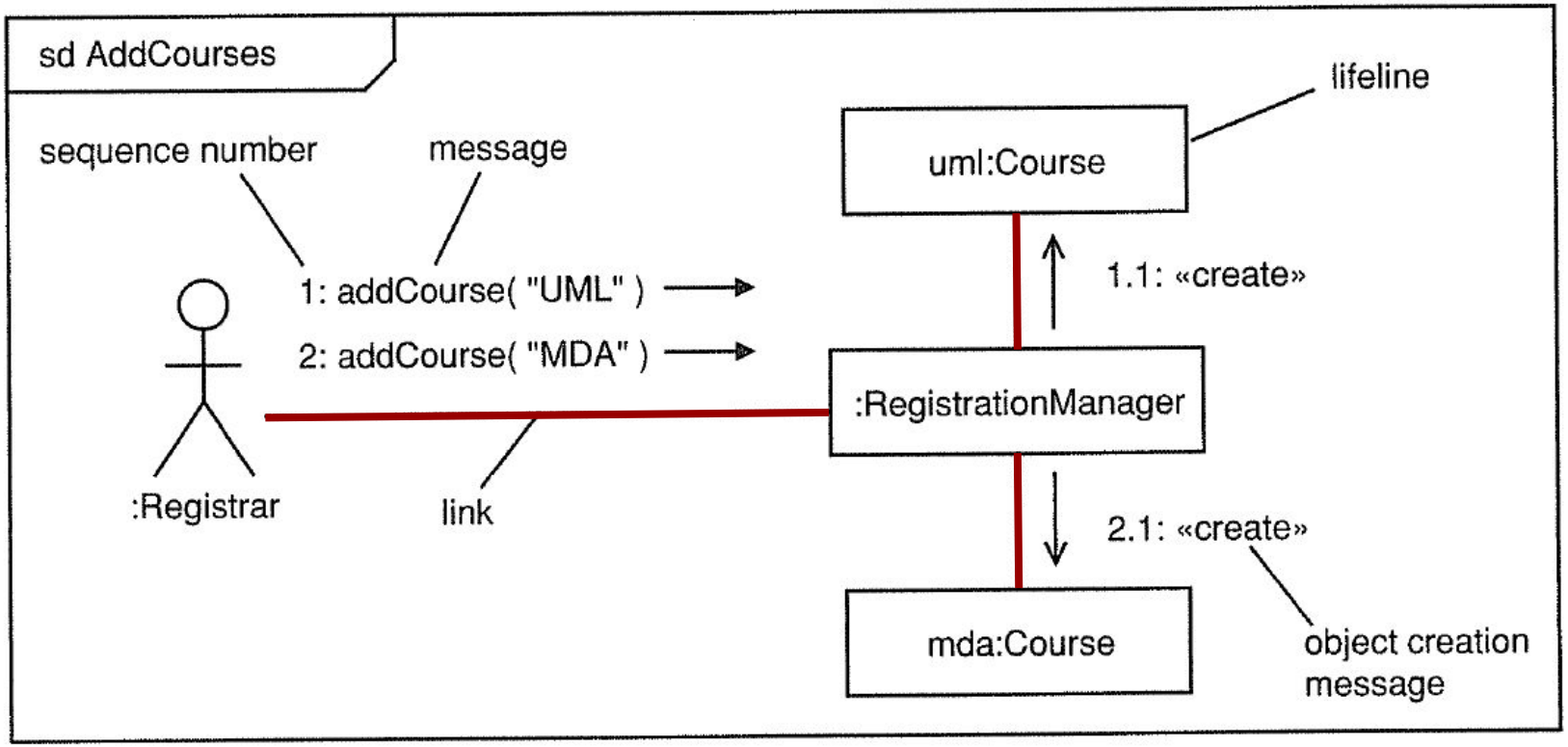
Communication diagrams syntax



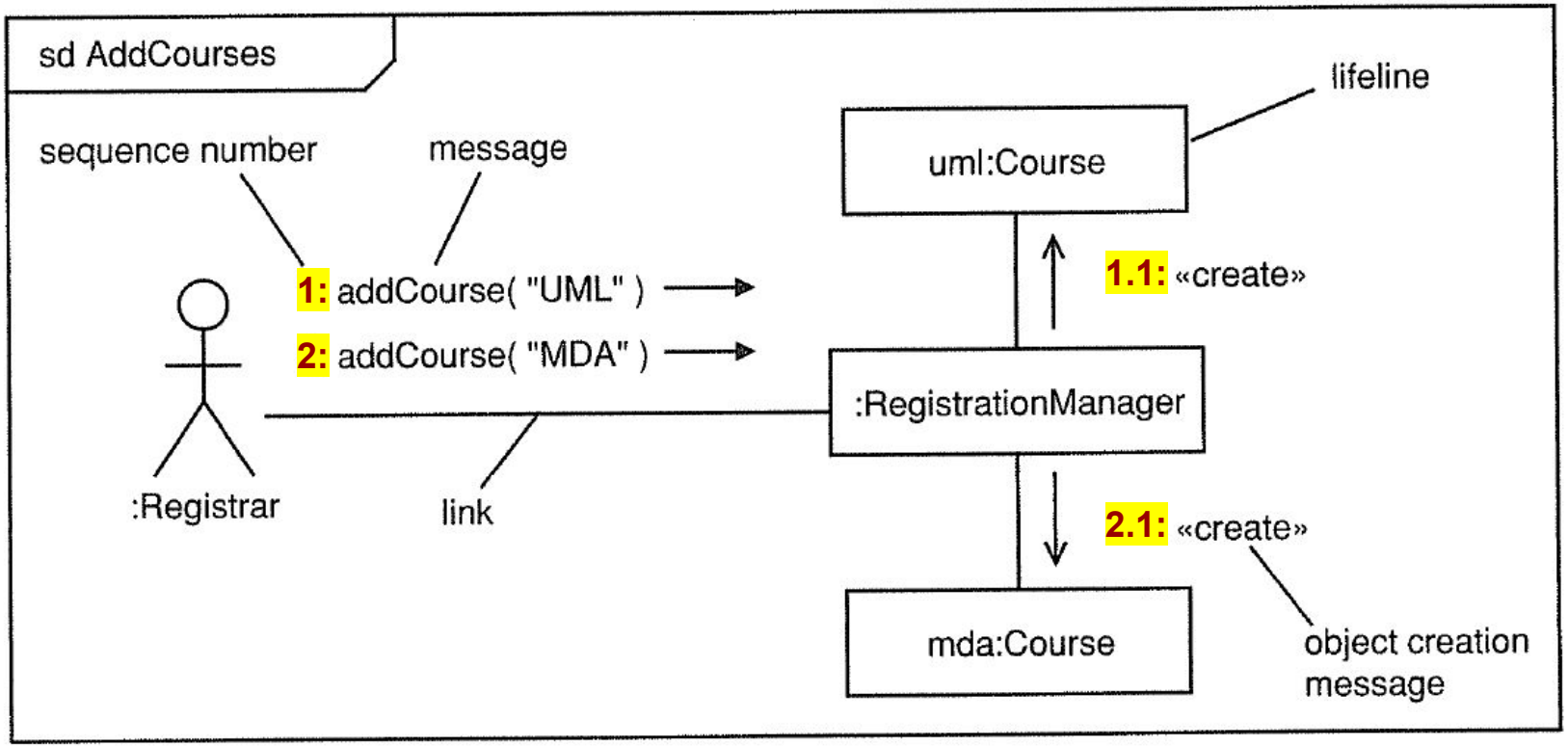
Communication diagrams syntax is similar to that of sequence diagrams, but the lifelines have no "tails"



lifelines are connected by links that provide communication channels for the messages

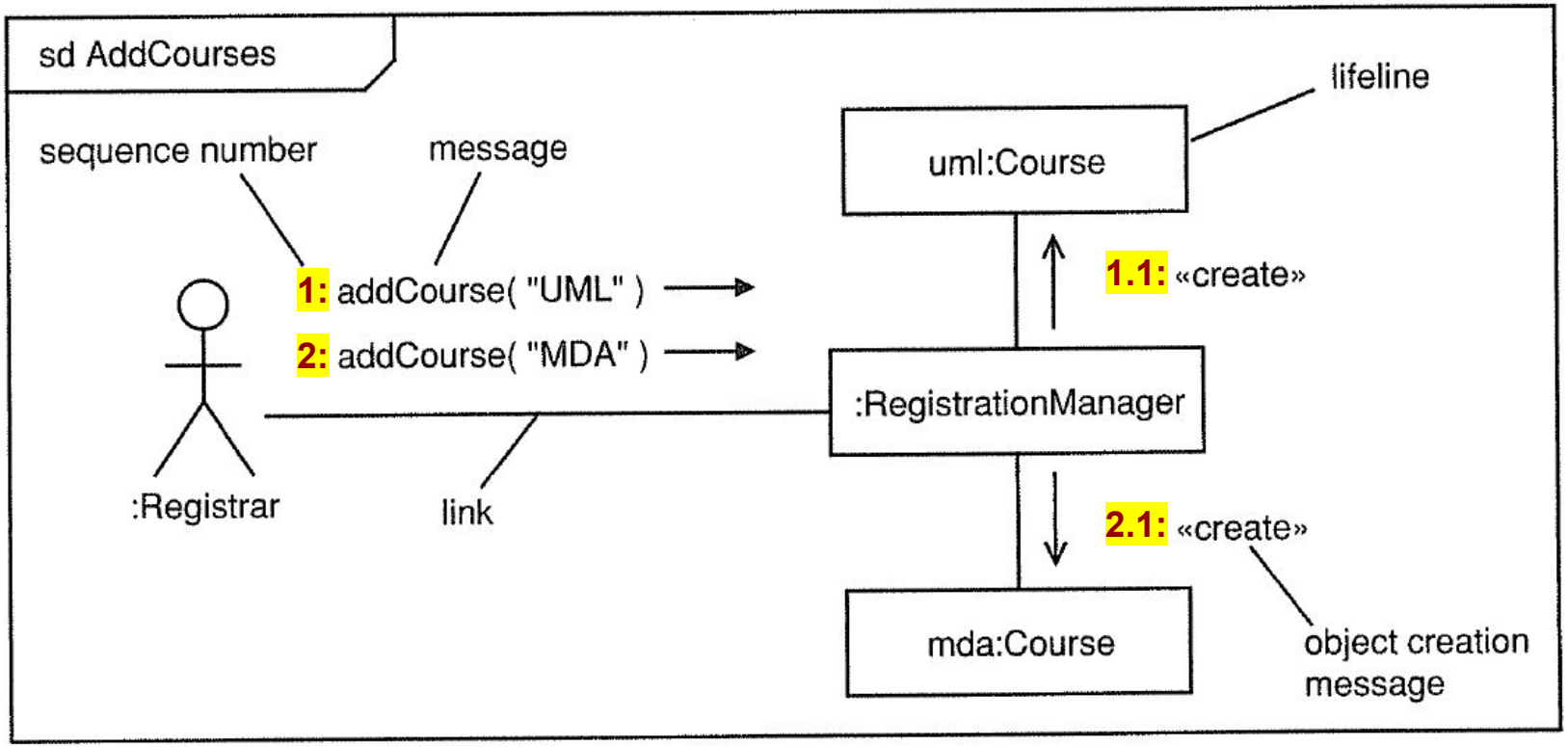


Sequencing and nesting are indicated by numbering each message hierarchically

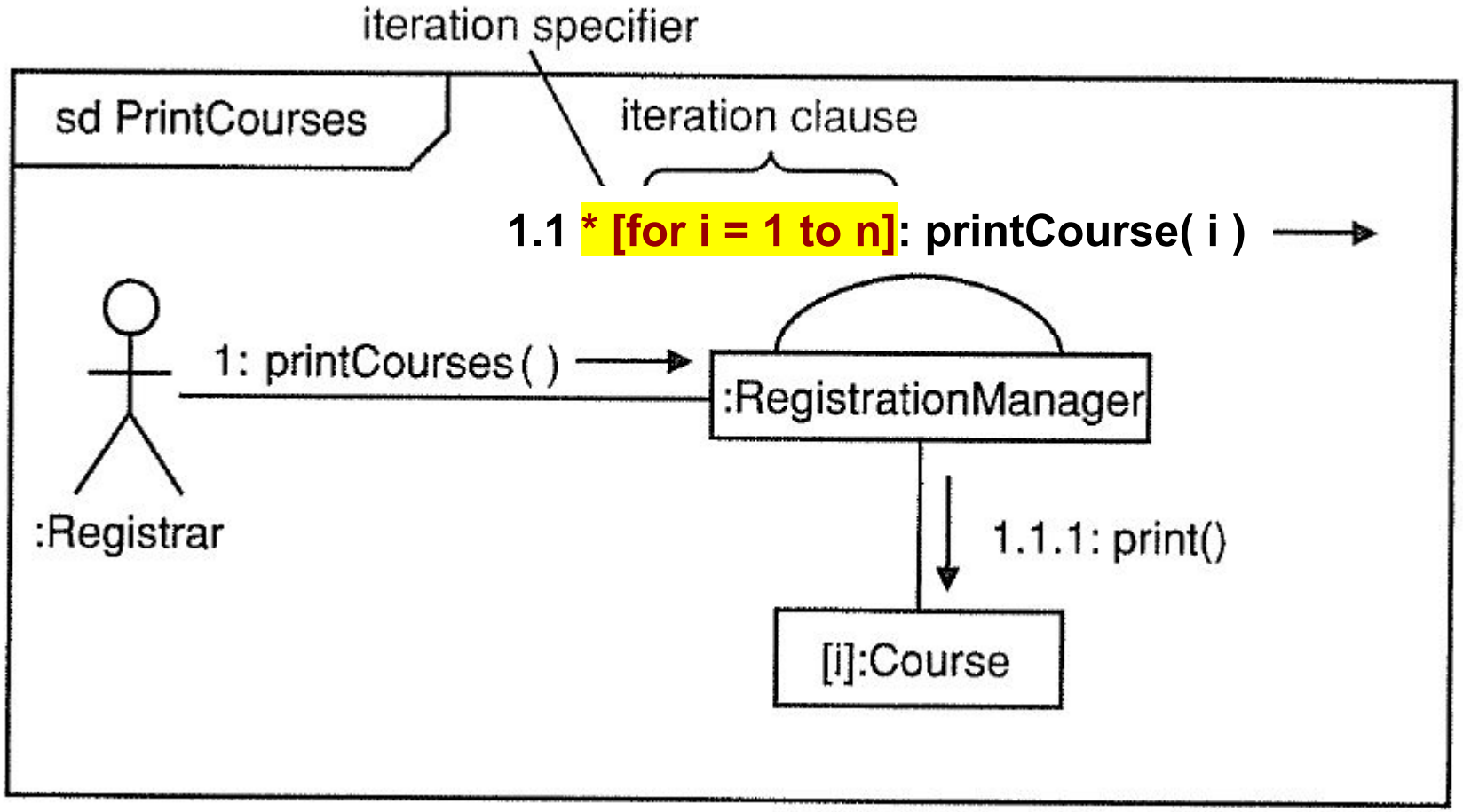


Sequencing and nesting are indicated by numbering each message hierarchically

Message sequence:
1. addCourse("UML")
   1.1. <<create>>
2. addCourse("MDA")
   2.1. <<create>>

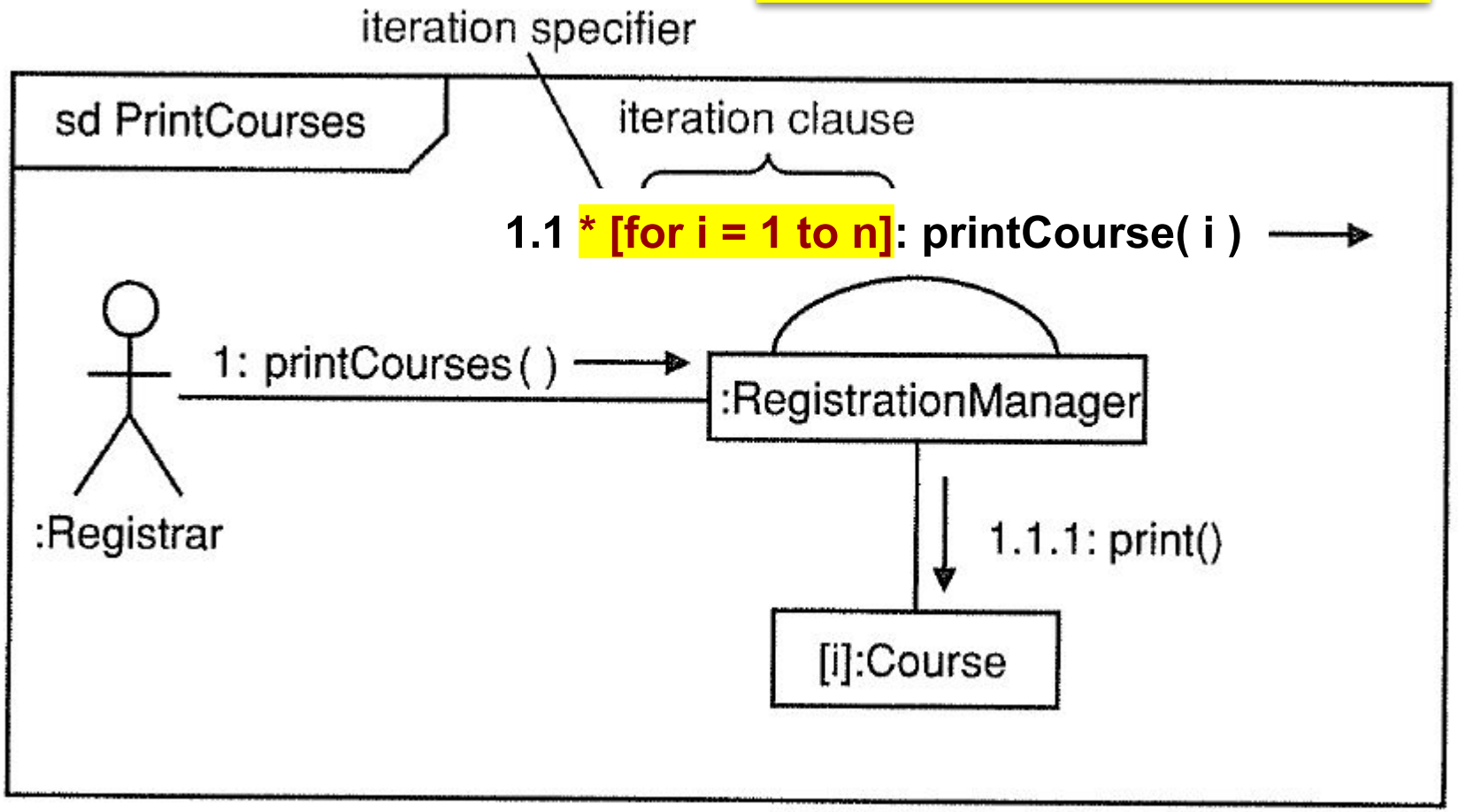


Iteration - an iteration clause specifies the number of times to repeat a message to send

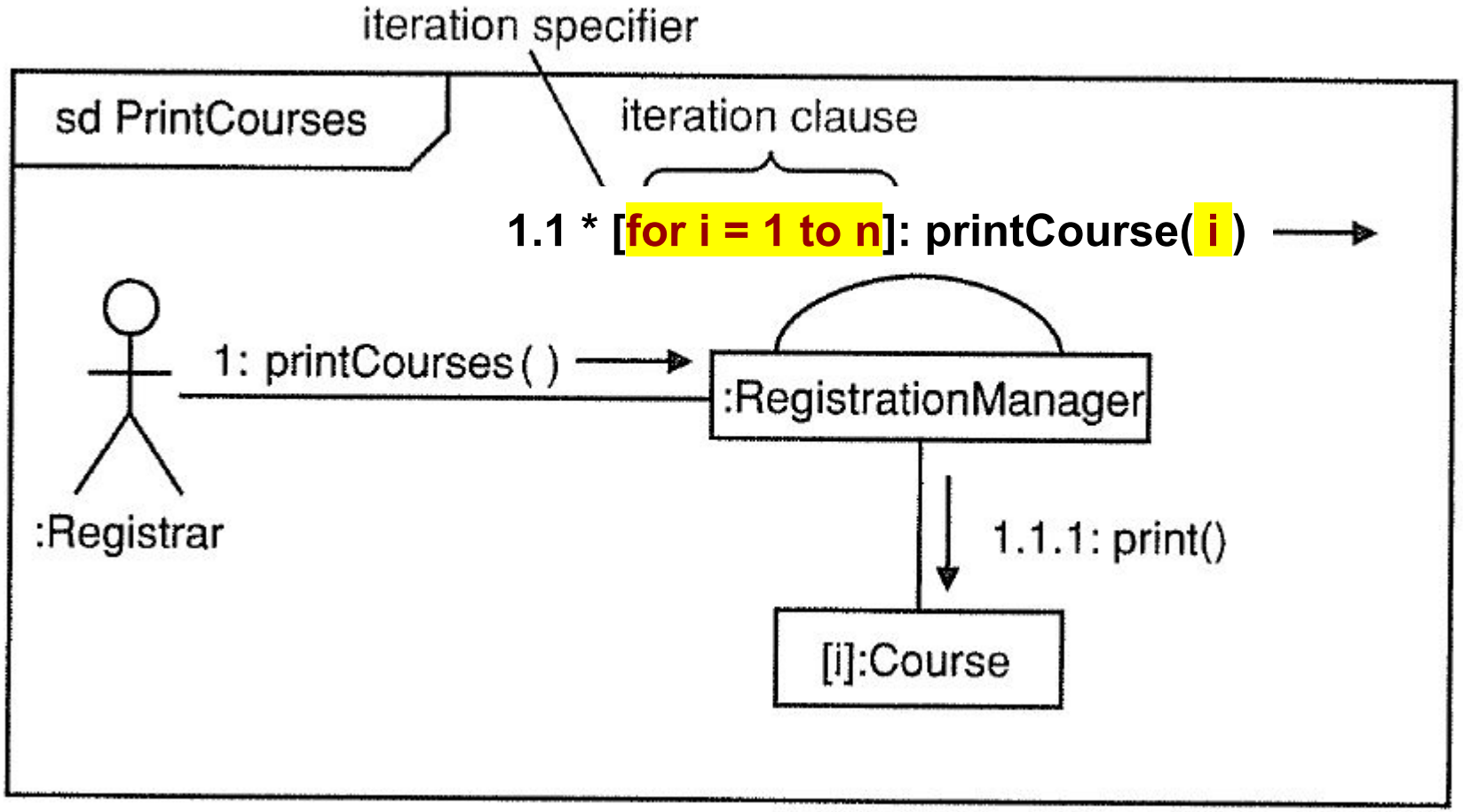


Although the UML specification does not enforce it, we will use the same syntax as for sequence diagrams:

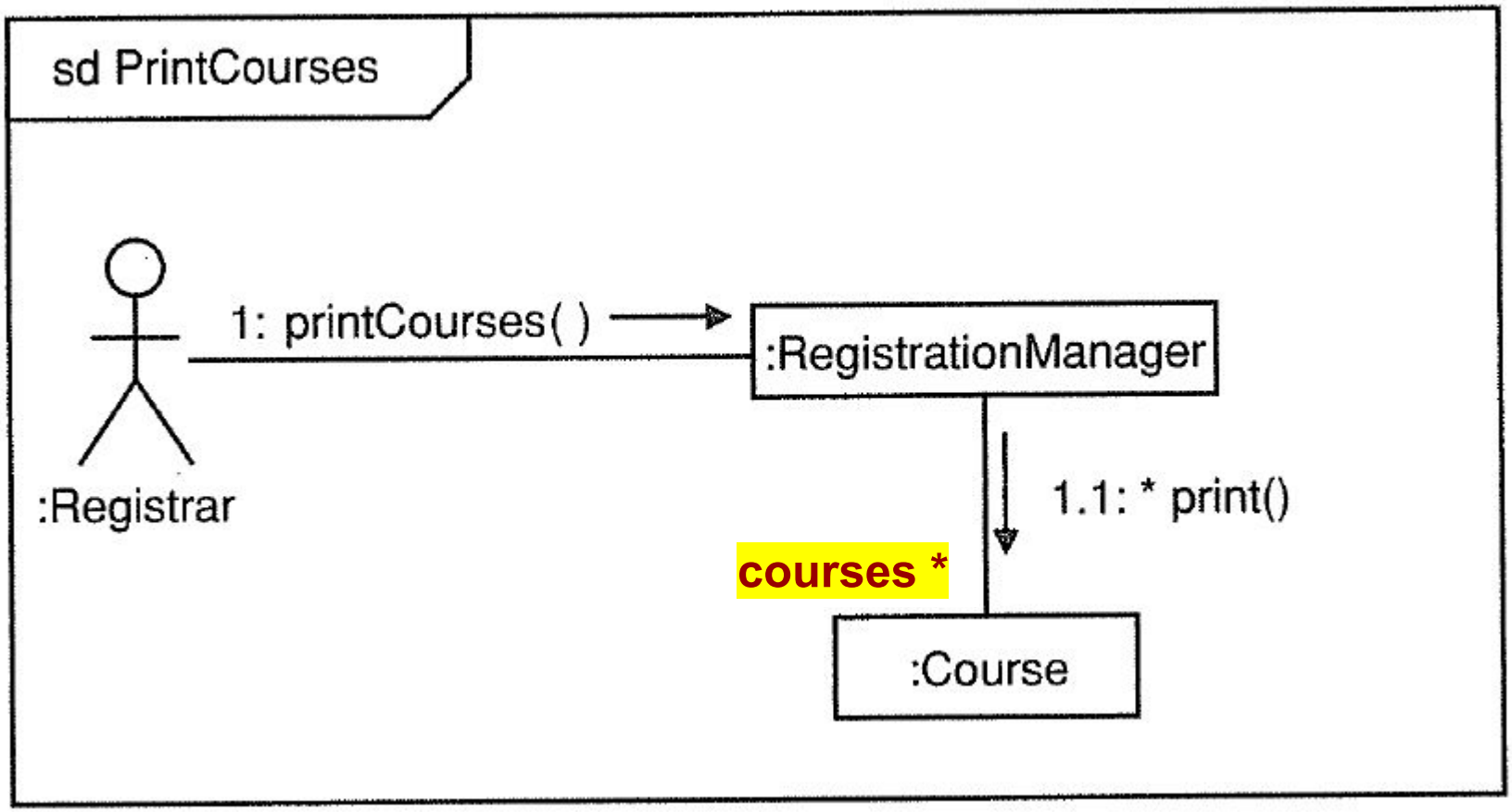
***[loop min, max [condition]]**



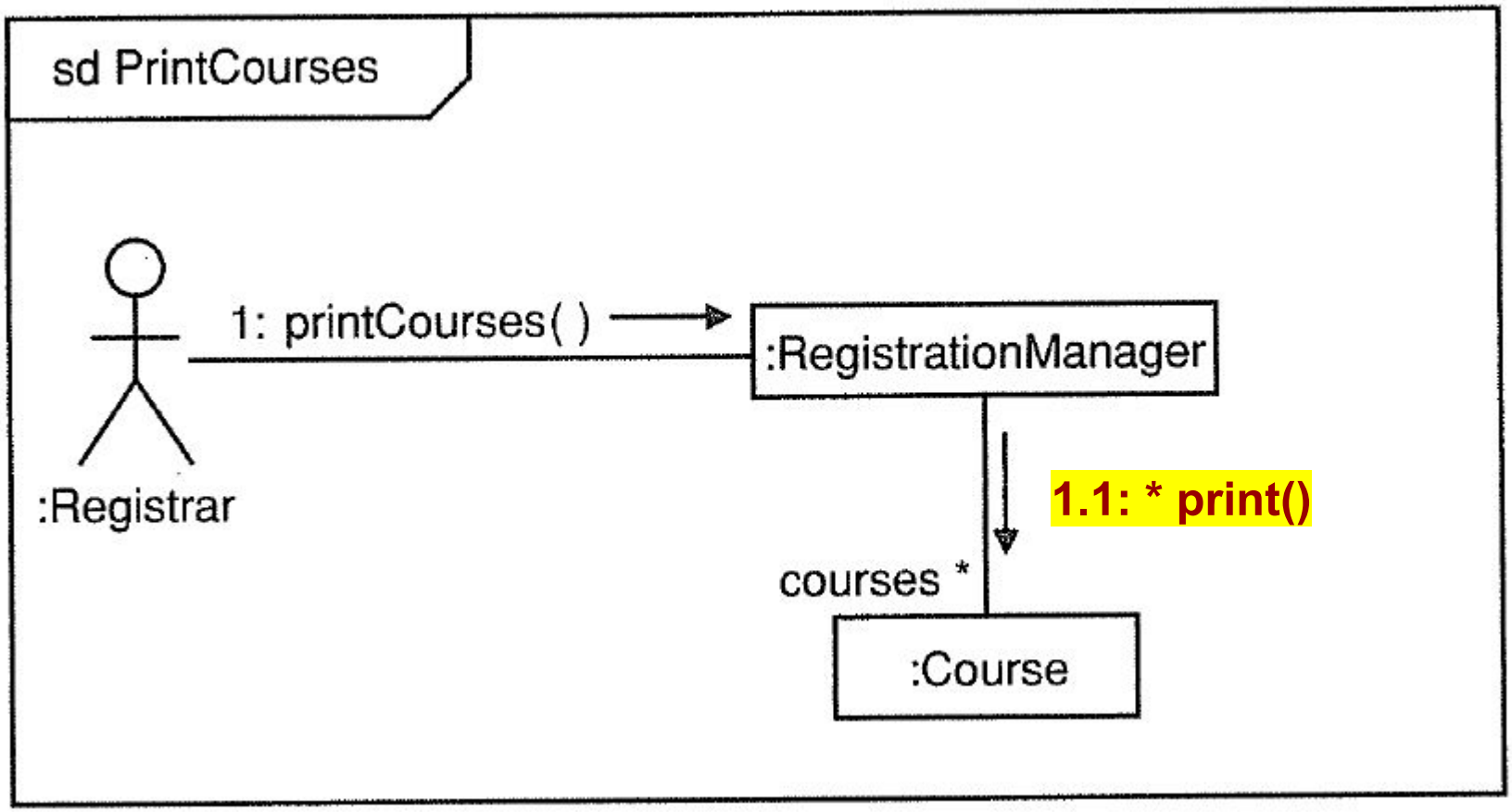
What if the stored courses are *not* in some kind of indexed collection? This solution assumes they are!



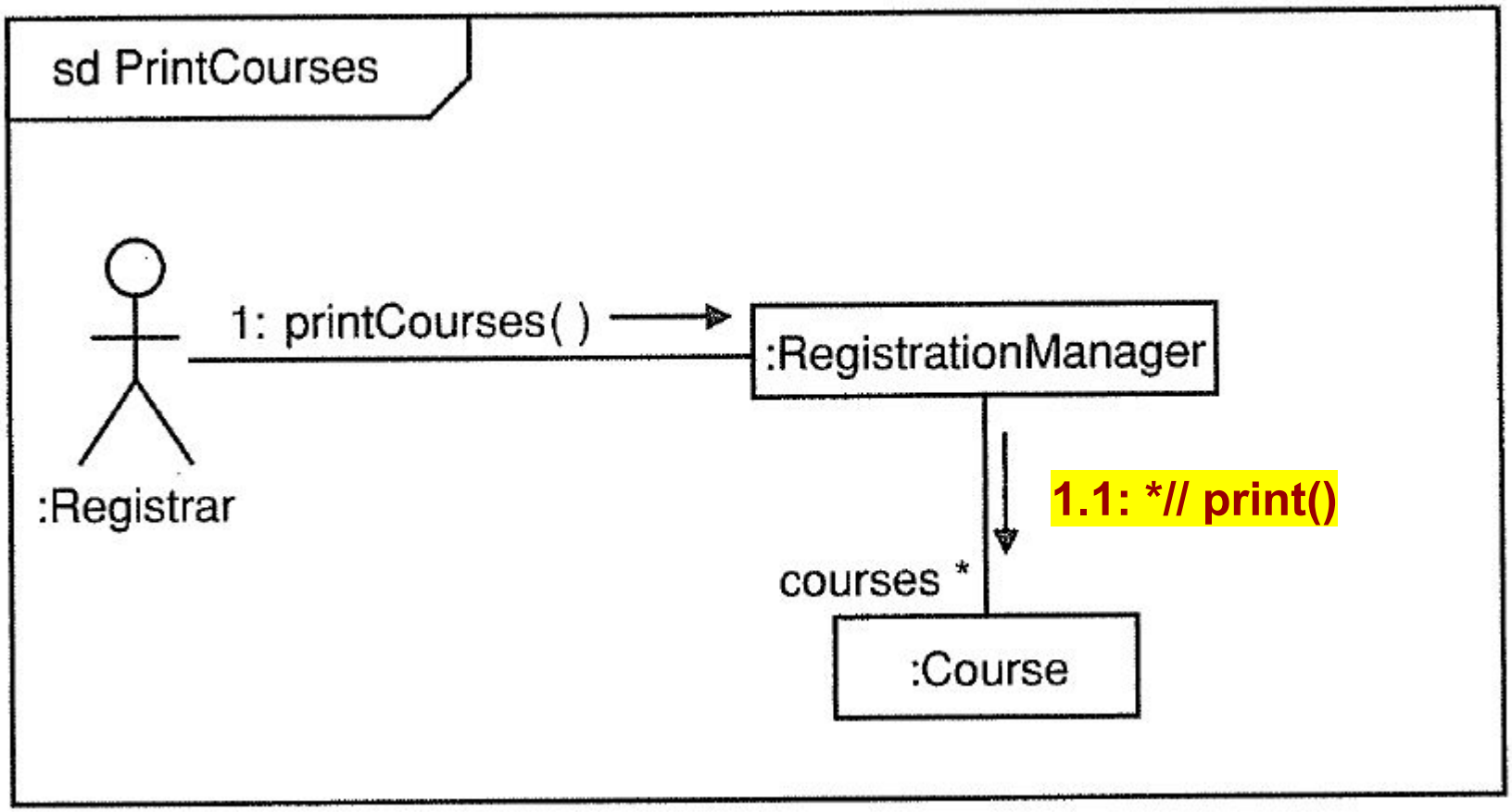
If we remove the indexation assumption, we can still show the role and multiplicity



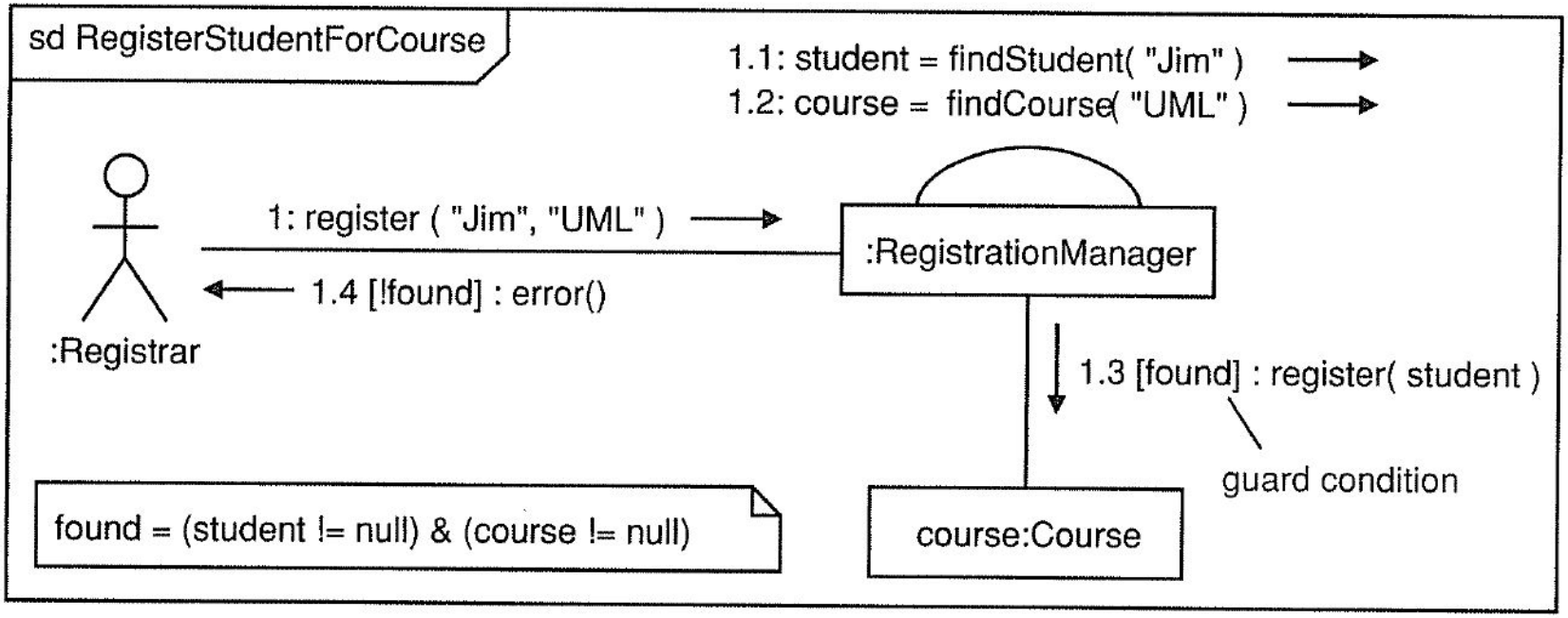
The * in 1.1: * print() denotes that the print() message is sent to each object in the collection



We can also send the messages in parallel to all the elements of the collection with the *// parallel iteration specifier



Branching sends the message only if the condition is true



1: register (“Jim”, “UML”)
    1.1: student = findStudent(“Jim”)
    1.2: course = findCourse(“UML”)
    1.3: [found] : register(student)
    1.4: [!found] : error()

Shortcomings of Communication diagrams

- Branching is hard to represent clearly
- Communication diagrams tend to become cluttered fast
- It is better to show complex branches on sequence diagrams

Sequence vs communication diagrams

Sequence diagrams

- Easy to read and understand
- The message order is very clear and intuitive
- Show proper structures for cycles, concurrency, alternatives, etc
- Require discipline while being constructed

Communication diagrams

- Less demanding regarding discipline while constructing
  - We don't know the exact place where the object will stay
- The message order can be added later
- … but are poorer

Modelling concurrency

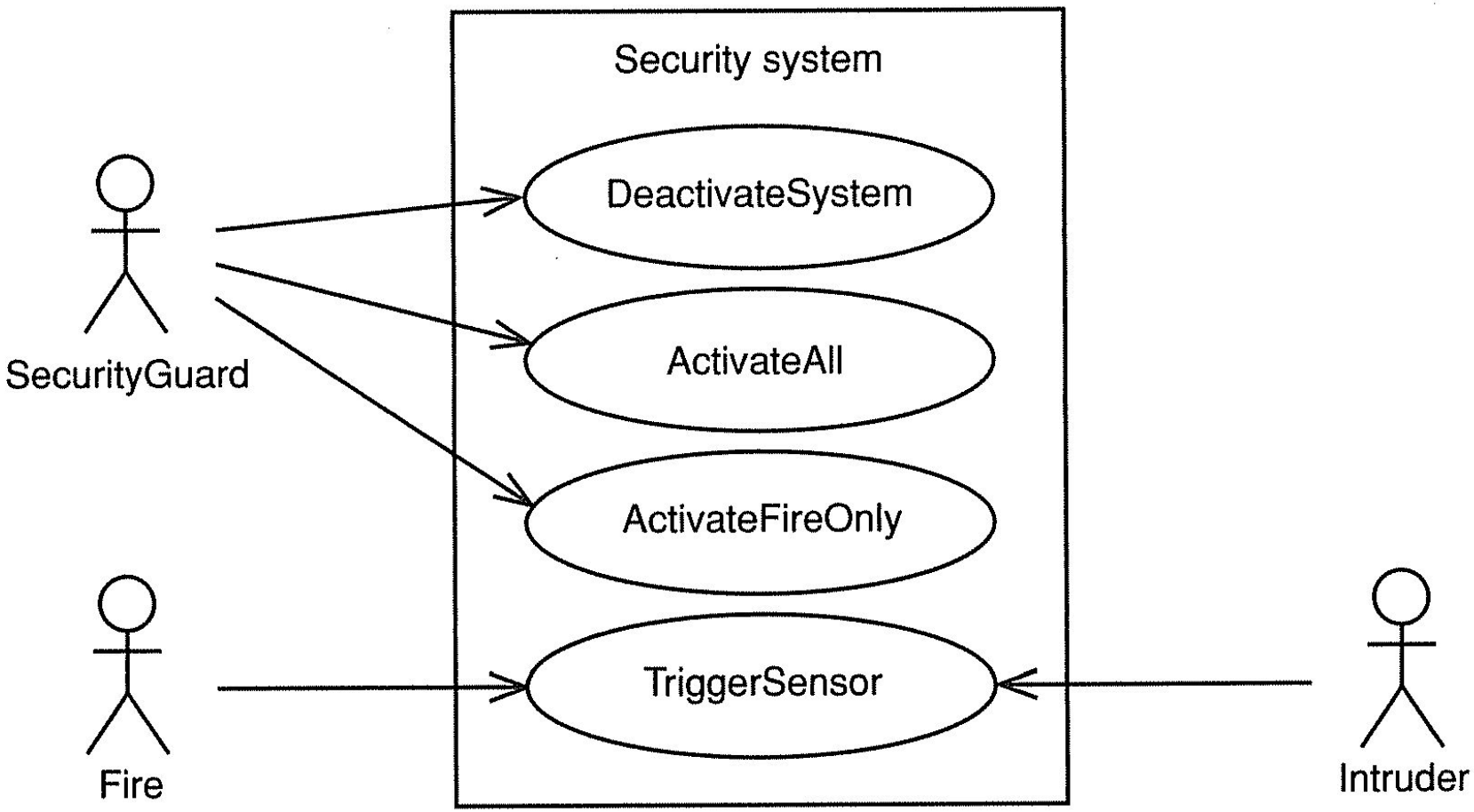
Concurrency is a key consideration in design

- UML 2 supports concurrency in several ways:
  - Active classes
  - Forks and joins (activity diagrams)
  - par operator (sequence diagrams)
  - sequence number prefixes (communication diagrams)
  - multiple traces (timing diagrams)
  - orthogonal composite states (state machines)

Active classes

- In concurrency, each active object has its own thread of execution
- Active objects are instances of active classes

Consider a security system embedded system



Use case DeactivateSystem

Use case: DeactivateSystem
ID: 1
Brief description: Deactivate the system.
Primary actors: SecurityGuard
Secondary actors: None.
Preconditions: 1. The SecurityGuard has the activation key.
Main flow: 1. The SecurityGuard uses the activation key to switch the system off. 2. The system stops monitoring security sensors and fire sensors.
Postconditions: 1. The security system is deactivated. 2. The security system is not monitoring the sensors.
Alternative flows: None.

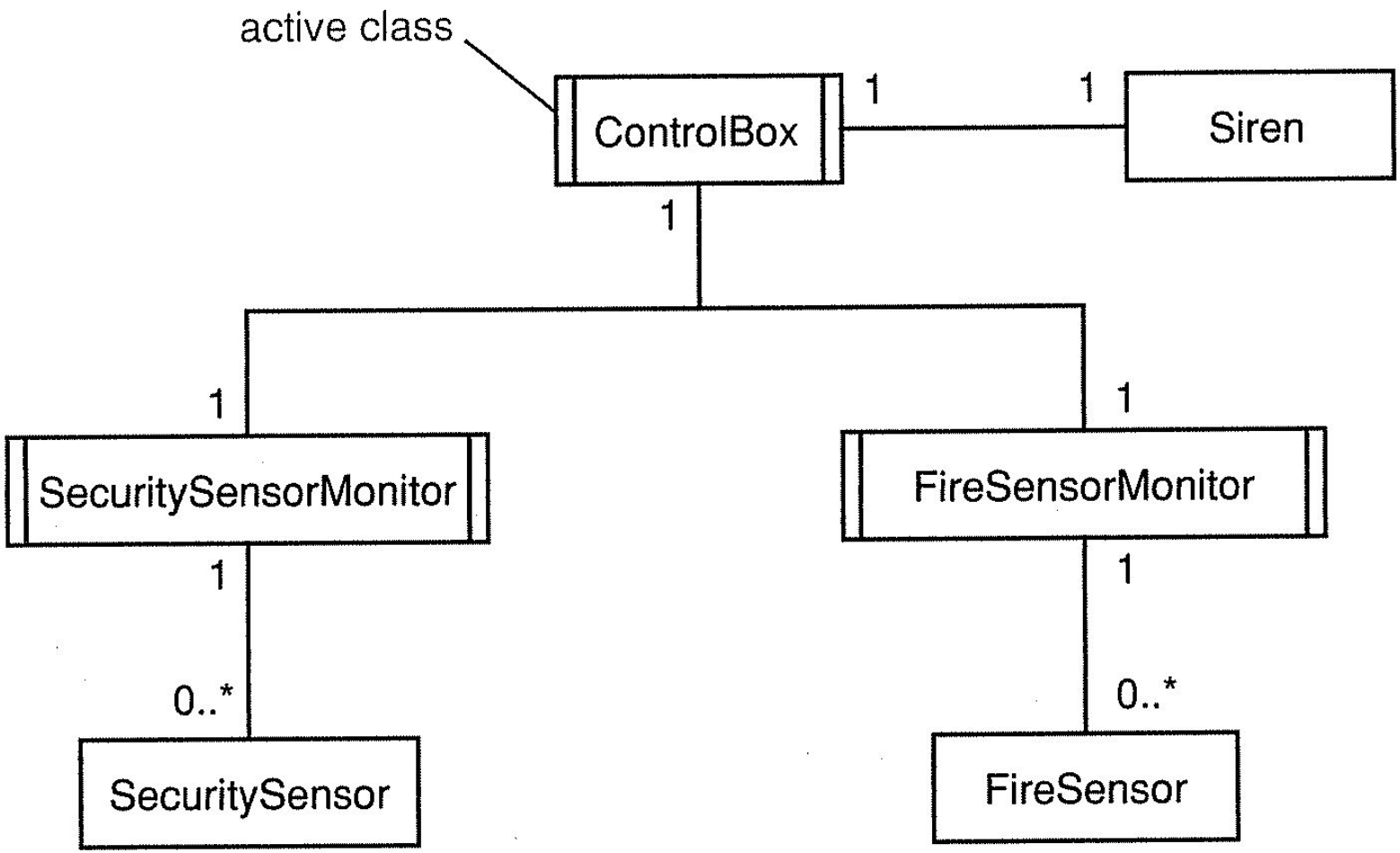
Use case ActivateAll

Use case: DeactivateSystem
ID: 1
Brief description: Deactivate the system.
Primary actors: SecurityGuard
Secondary actors: None.
Preconditions: 1. The SecurityGuard has the activation key.
Main flow: 1. The SecurityGuard uses the activation key to switch the system off. 2. The system stops monitoring security sensors and fire sensors.
Postconditions: 1. The security system is deactivated. 2. The security system is not monitoring the sensors.
Alternative flows: None.

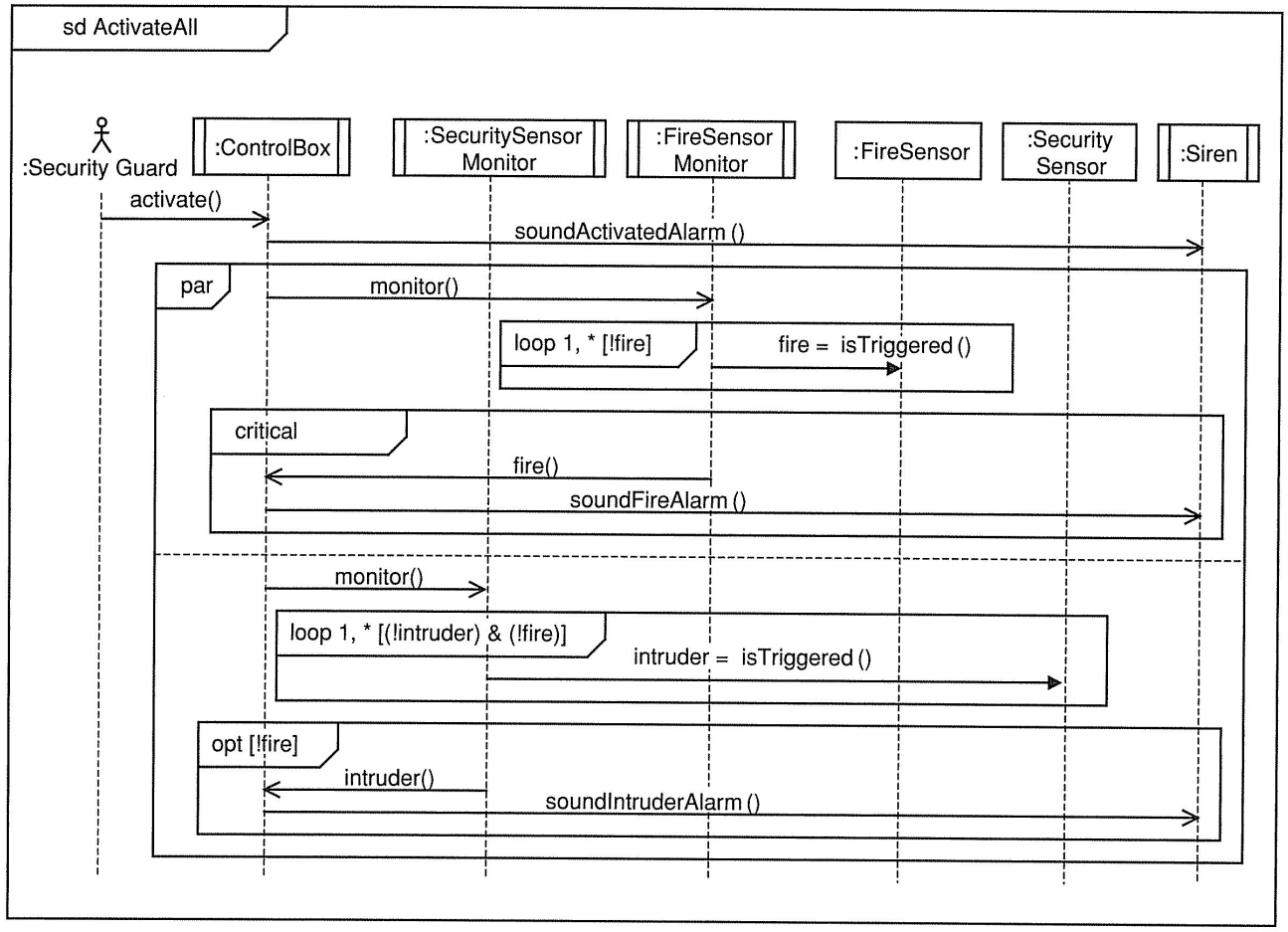
Use case TriggerSensor

Use case: TriggerSensor
ID: 3
Brief description: A sensor is triggered.
Primary actors: Fire Intruder
Secondary actors: None.
Preconditions: 1. The security system is activated.
Main flow: 1. If the Fire actor triggers a FireSensor 1.1 The Siren sounds a fire alarm. 2. If the Security actor triggers a SecuritySensor 2.1 The Siren sounds a security alarm.
Postconditions: 1. The Siren sounds.
Alternative flows: None.

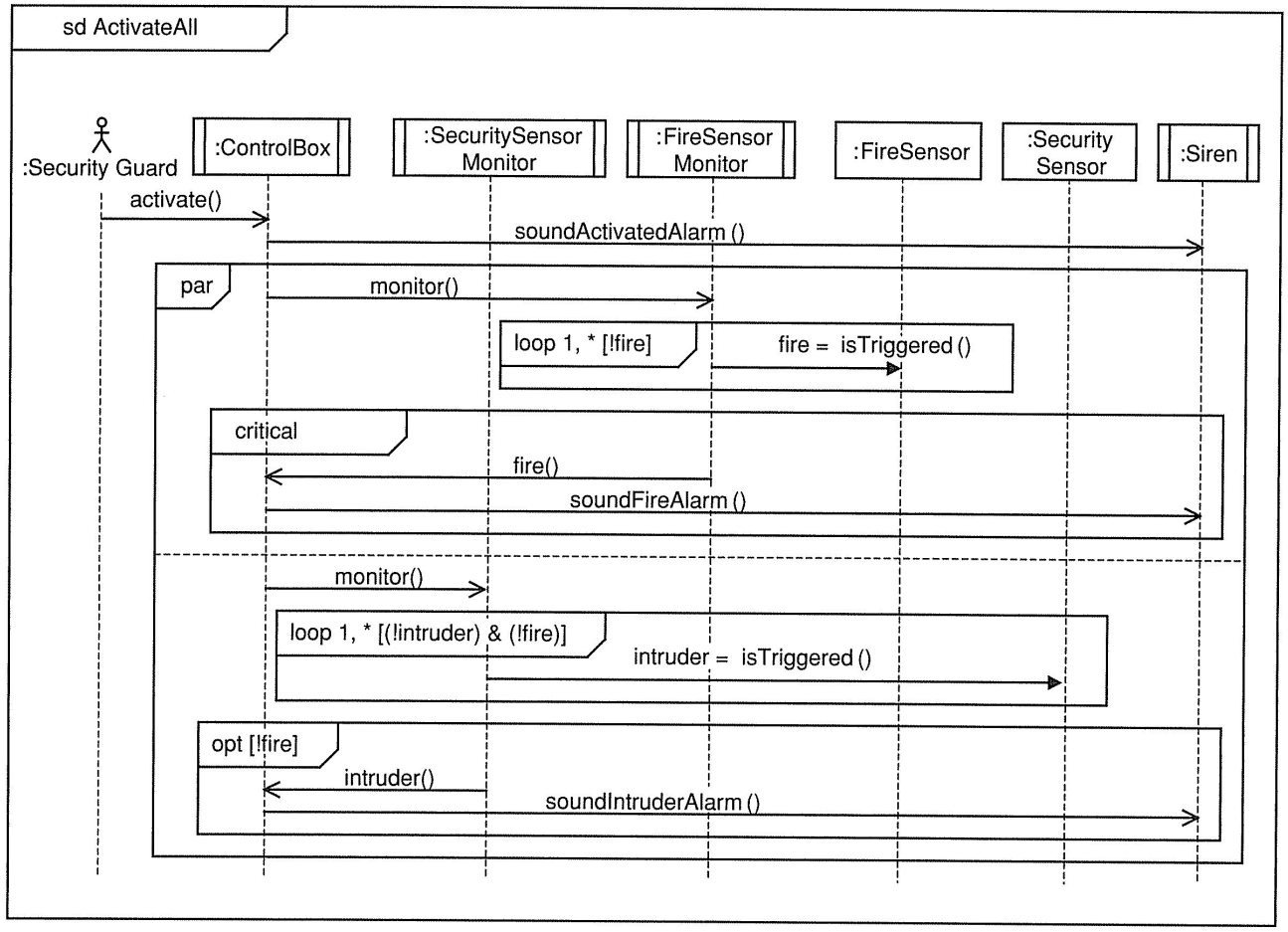
Active classes have instances that are active objects



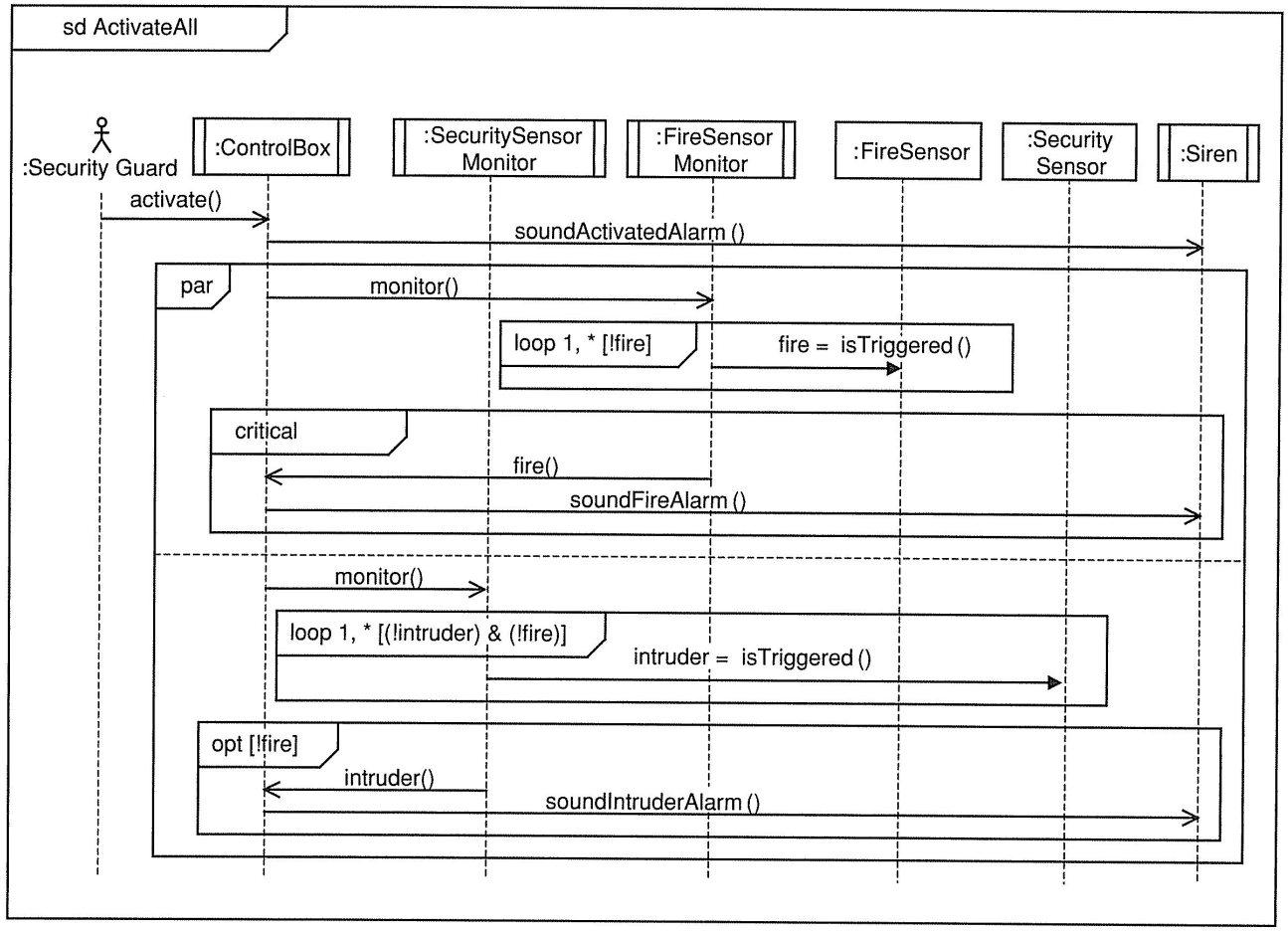
Sequence diagram for ActivateAll



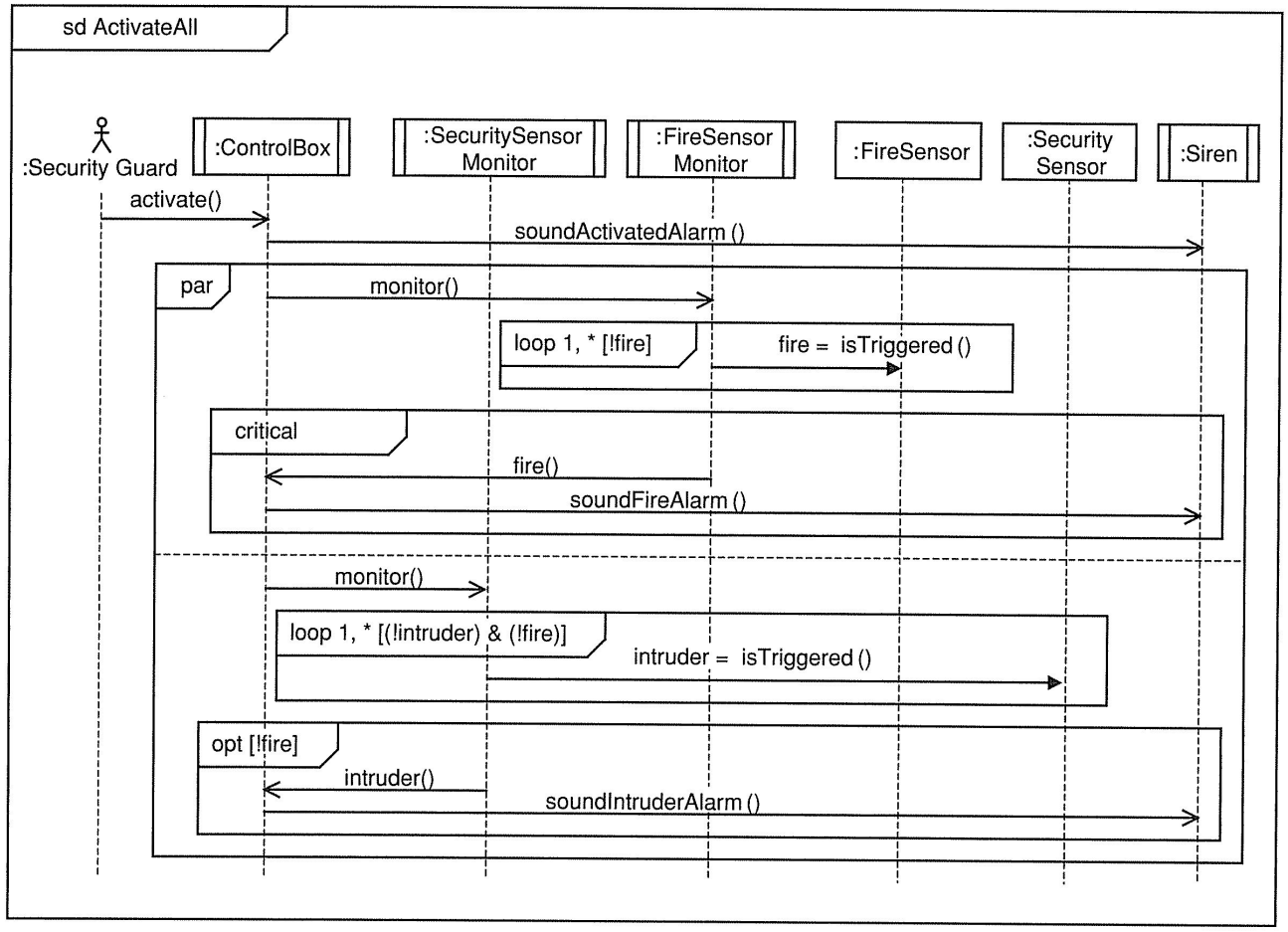
Both operands of par execute in parallel



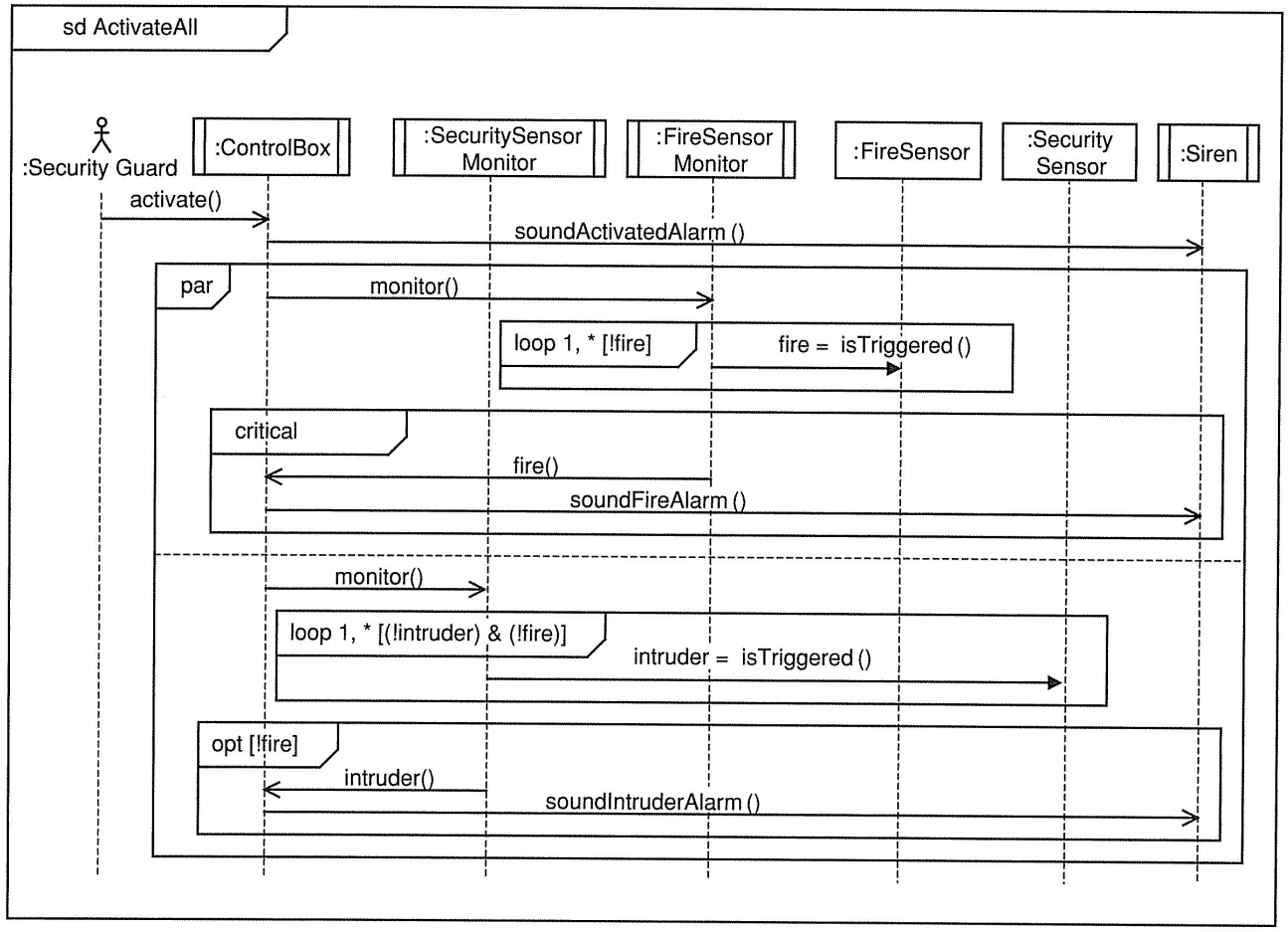
A critical section represents an atomic behavior that can't be interrupted



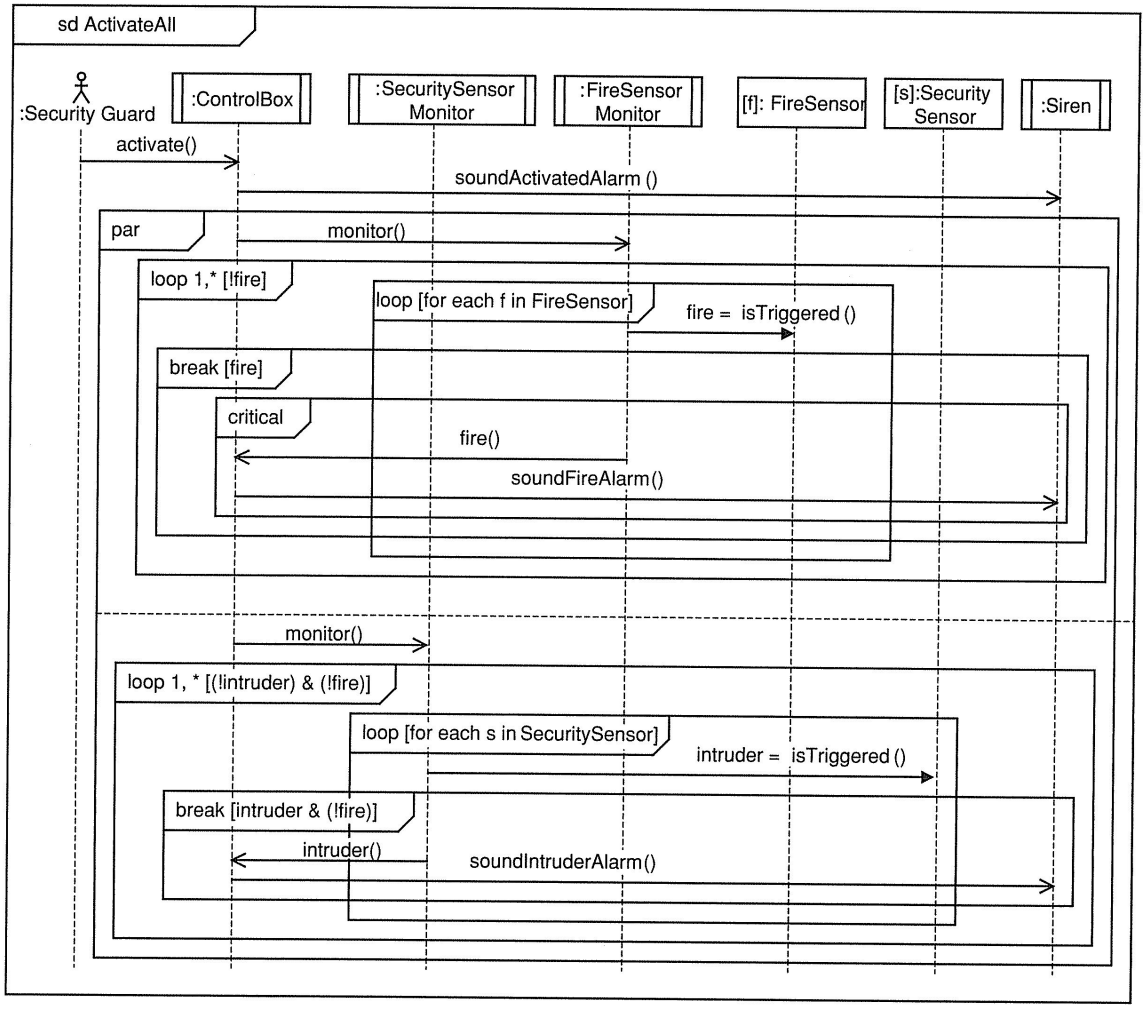
Both loops have a semantics of a repeat... until - they execute once, set the variable used in their condition, and then repeat while it is true



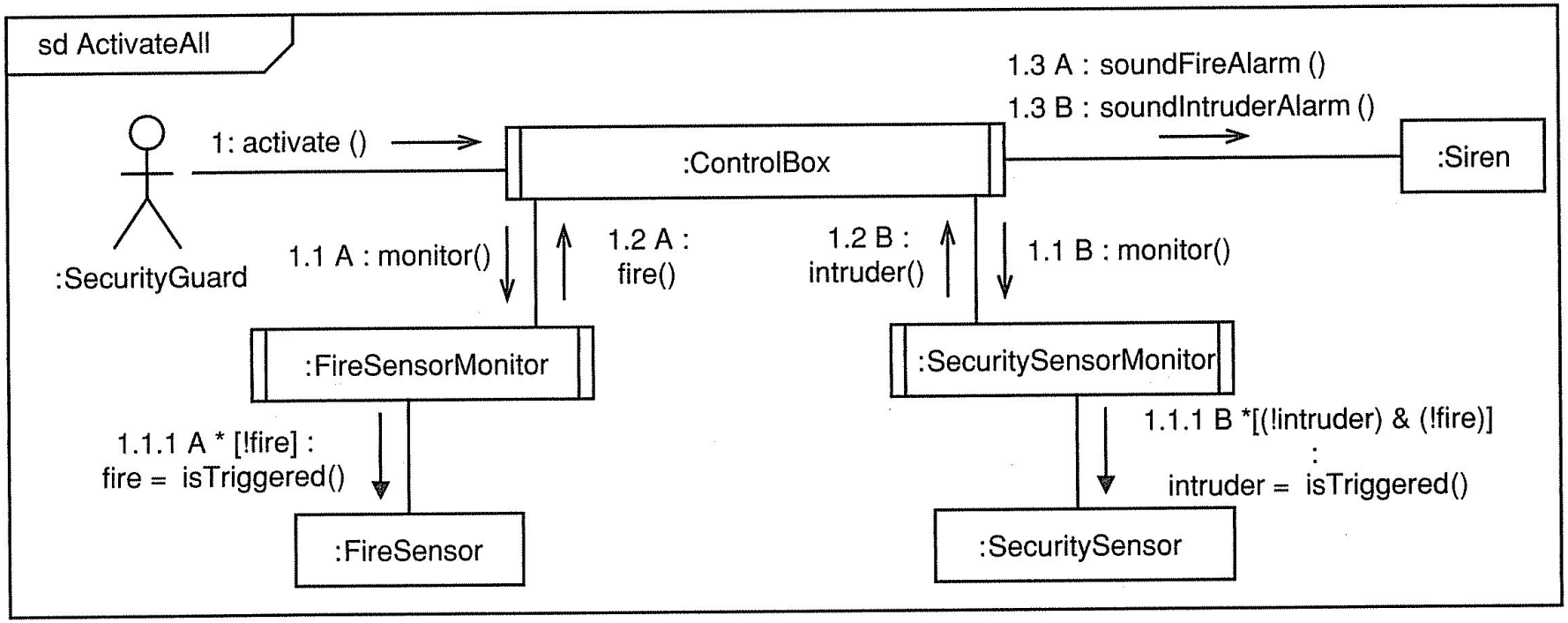
The fire alarm must always have precedence over the intruder alarm



If we have several fire sensors, we have to loop and check each of them



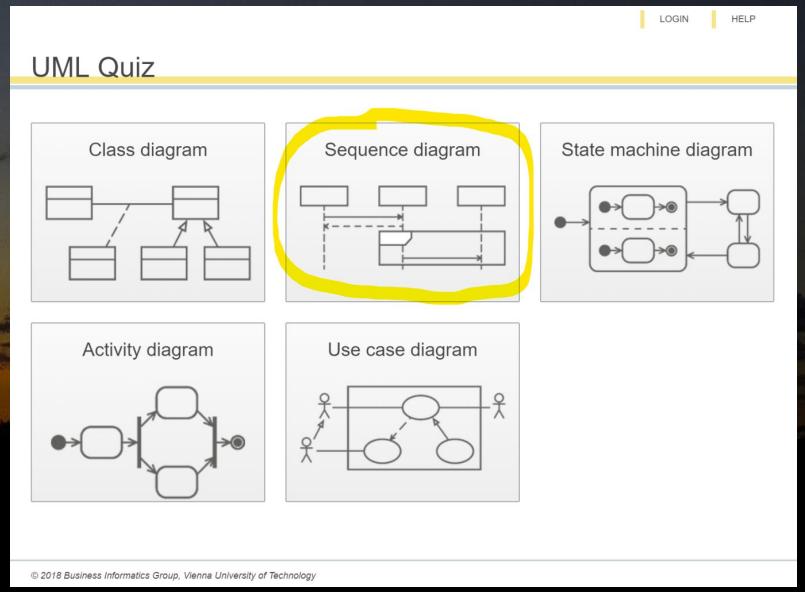
Concurrency in communication diagrams



What if we have multiple fire sensors?

- It is possible to show nested loops, but the diagram gets too cluttered
- Go back to sequence diagrams, where this can be better represented

Did you really understand Sequence Diagrams? Test yourself at: http://elearning.uml.ac.at/



Bibliography

Jim Arlow and Ila Neustadt, “UML 2 and the Unified Process”, Second Edition, Addison-Wesley 2006

- Chapters 12, 20