Software Engineering Course

Chapter 5

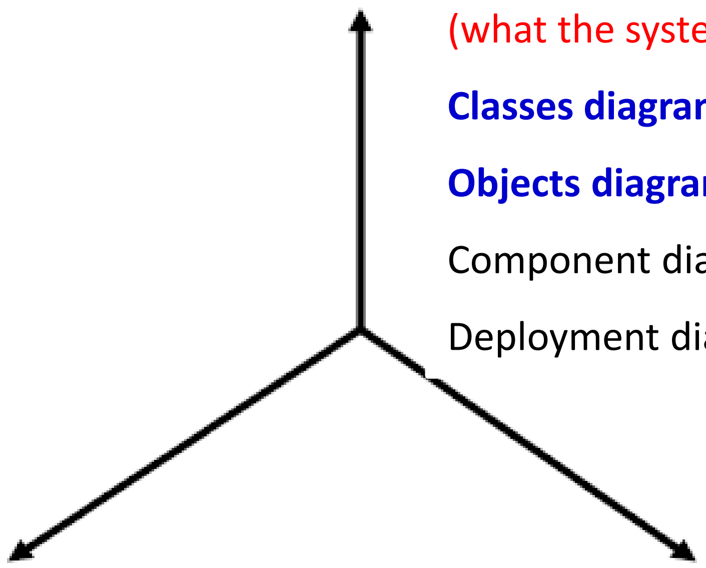
UML Diagrams: Dynamic View

2022-2023

Dr. Sofia Kouah

✓Three modeling axes:

Modeling axes



Static

(what the system is)

Classes diagram

Objects diagram

Component diagram

Deployment diagram

Fonctionnel

Functional

(what the system is doing)

Use cases diagram

Dynamique

Dynamic

(how the system evolves)

State / transition diagram
Activity diagram
Sequence diagram
Collaboration diagram

In this chapter, we are interested in:

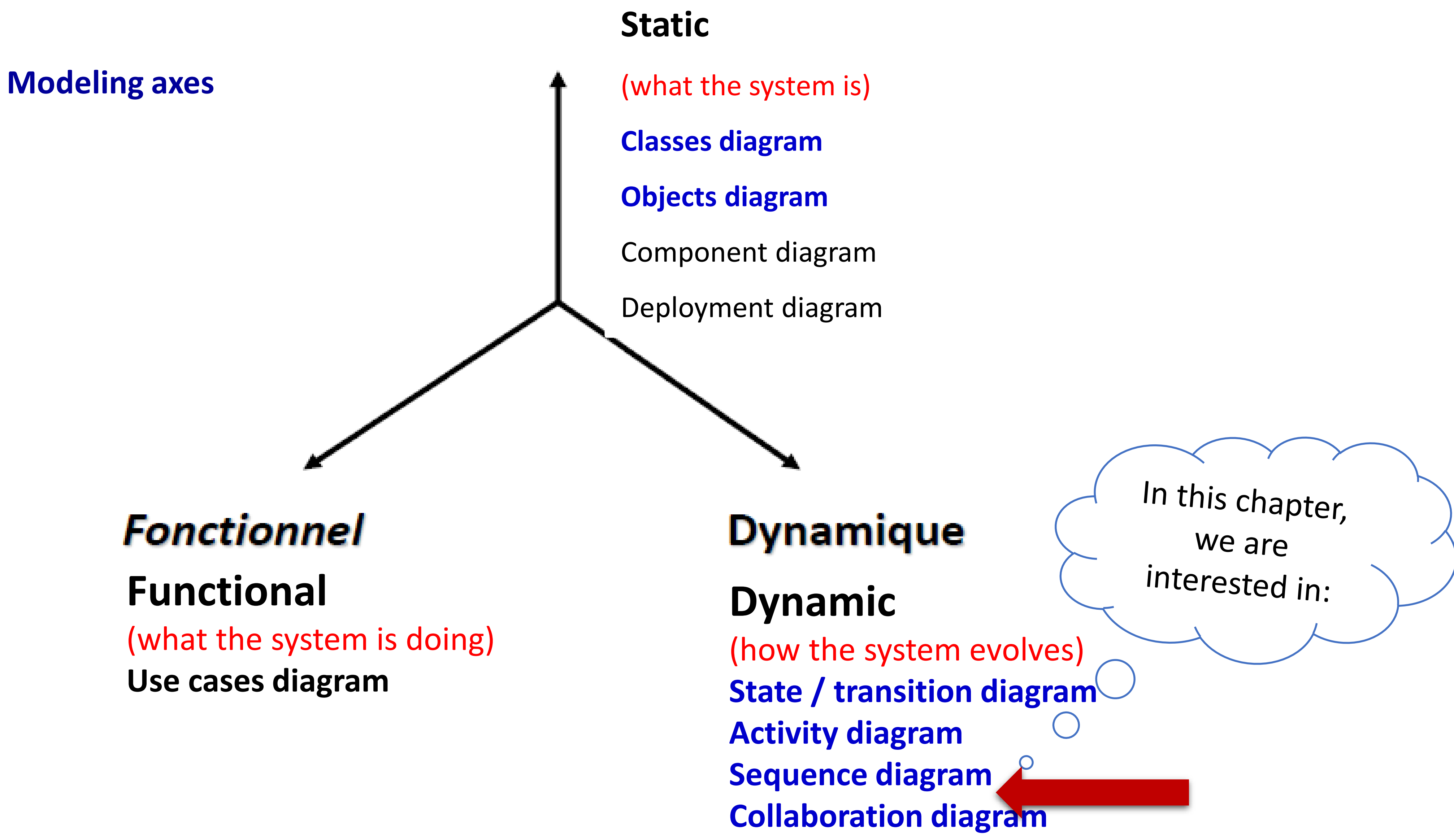
Plan

1. Interaction diagrams (sequence and collaboration)
2. State/transition diagram
3. Activity Diagram

Part 1: Interaction diagrams

- Sequence diagram
- Collaboration (Communication) Diagram

✓Three modeling axes:



Sequence Diagram: Role

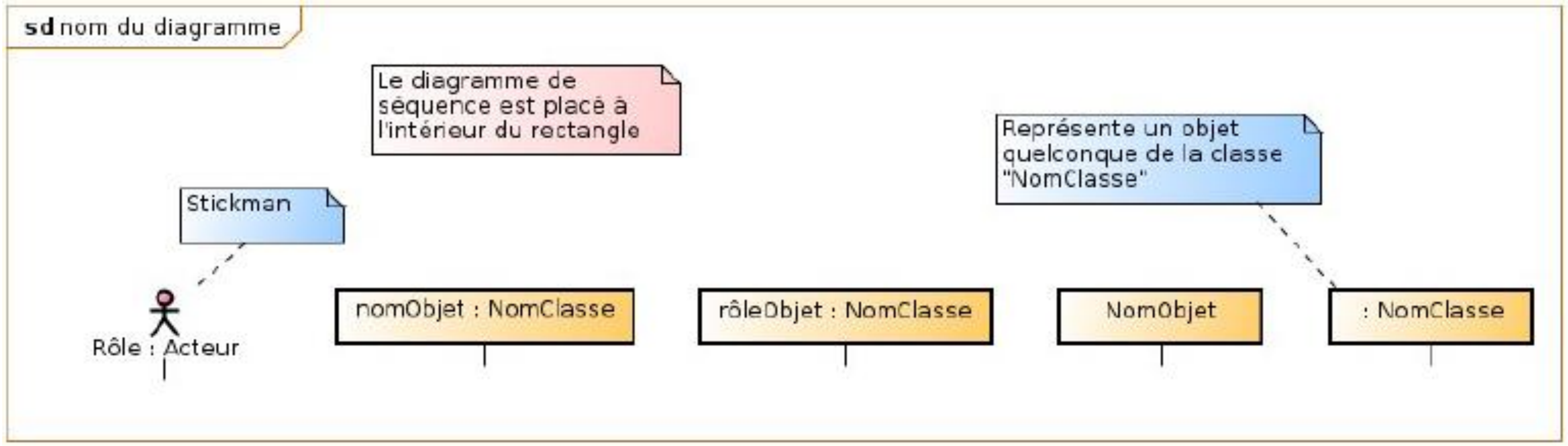
- ✓It is a **behavioral (dynamic) diagram** and more precisely it **is an interaction diagram**.
- ✓Allows to represent **exchanges between the various objects and actors** of the system according to time.
- ✓ Unless the system to be modeled is extremely simple, **we cannot model the overall dynamics** of the system in a single diagram.
- ✓We will therefore **use a set of sequence diagrams,** each corresponding to a sub-function of the system, generally to illustrate a use case..

Sequence Diagram: Components

- ✓**Delimitation of the sequence diagram**
- ✓**The object**
- ✓**The lifeline**
- ✓**The messages**

Sequence Diagram: Components

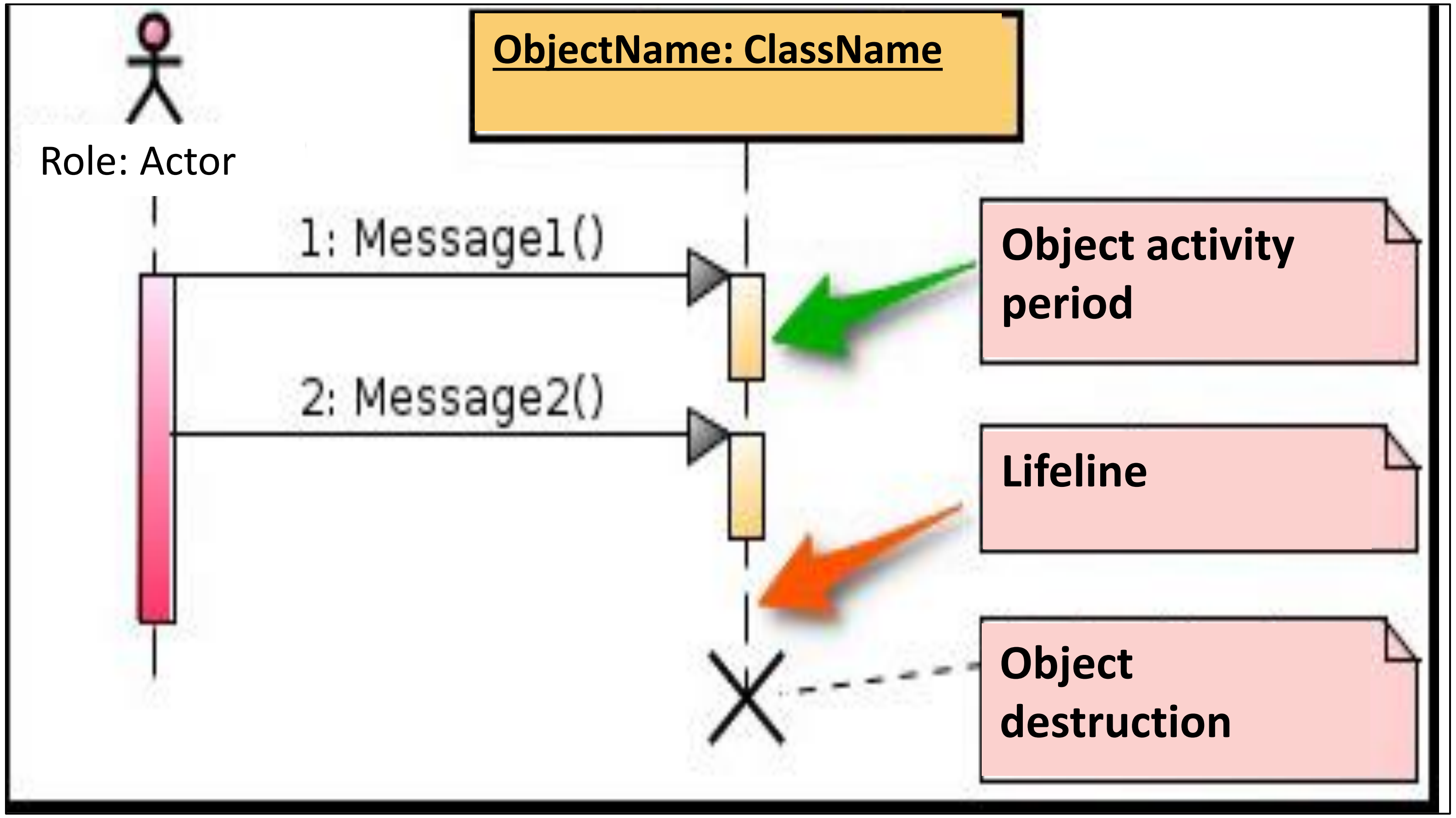
- Delimitation of the sequence diagram: The sequence diagram is placed in a **rectangle which has an sd label at the top left** (sequence diagramm) followed by the name of the diagram.
- The object: In a sequence diagram, the object has the same representation as in the object diagram. The object name is usually underlined and can take one of four forms:


sd nom du diagramme
Le diagramme de séquence est placé à l'intérieur du rectangle
Stickman
Représente un objet quelconque de la classe "NomClasse"
Rôle : Acteur
nomObjet : NomClasse
rôleObjet : NomClasse
NomObjet
: NomClasse

Sequence Diagram: Components

The lifeline: As it represents the dynamics of the system, the sequence diagram brings into action the instances of classes involved in the realization of a particular use case.

Exchanges **between the objects** but also **the exchanges with the actors**, we also find the representation of the stickman (which can be considered as an object).

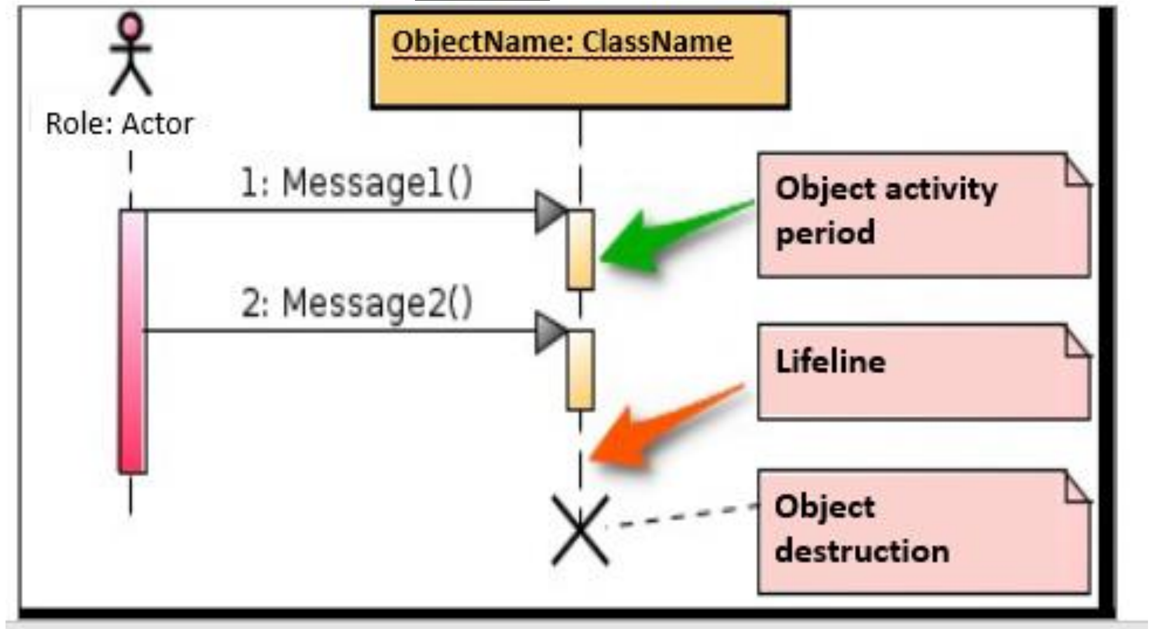


Sequence Diagram: Components

✓**Each object is associated with a lifeline** (in dotted lines vertical to the object) which can be considered as a temporal axis (time flows from top to bottom).

✓The **lifeline** indicates the **periods of activity** of the object (generally, the times when the object performs one of these methods).

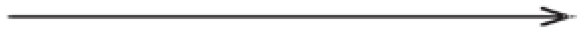
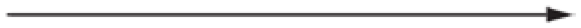
✓When **the object is <u>destroyed</u>**, the **lifeline ends with a <u>cross</u>**.



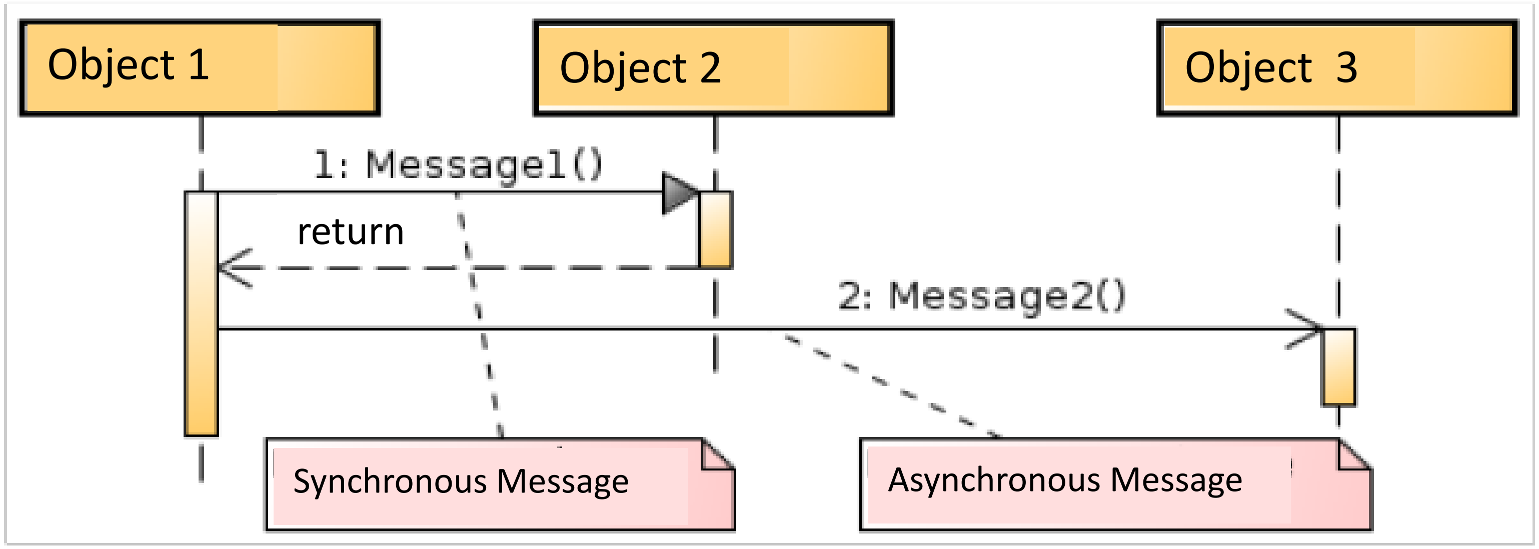
Sequence Diagram: Components

- ✓**Message**: defines **a particular communication between lifelines**.
- ✓A **message** is a communication from one object to another object.
- ✓The reception of a message is considered by the receiving object a**s an event that must be processed** (or not). There are several types of message, the most common are:
  - ✓**The invocation of an operation: synchronous message** (call of a method of the target object).
  - ✓**The sending of a signal: asynchronous message** (used for event management).
  - ✓**The creation or destruction** of a class instance during the main cycle.

Sequence Diagram: Components

- ✓A **synchronous message** blocks the sender until the receiver replies. The flow of control passes from the sender to the receiver.
  - ✓**Typically**: method call
  - ✓If an object A invokes a method of an object B, A remains blocked until B has finished.
  - ✓**Representation**: 
  - ✓You can associate to the method call messages a return message (in dotted lines) marking the resumption of control by the object sending the synchronous message.
- ✓An **asynchronous message** does not block the sender. The sent message can be taken into account by the receiver at any time or ignored.
  - ✓**Typically**: sending signal (see "signal" class stereotype
  - ✓**Representation**: 

Sequence Diagram: Components

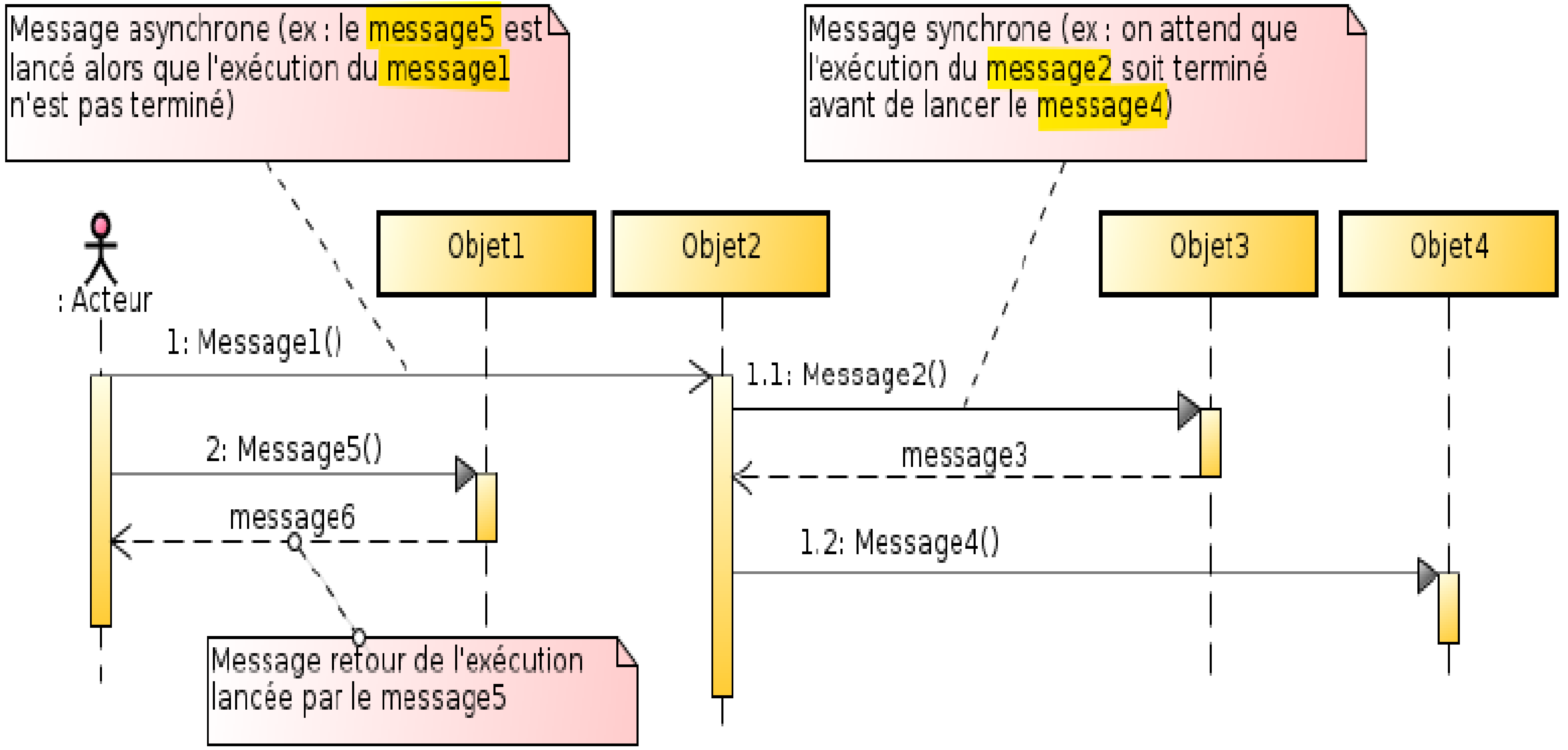


- If a **method** that has been activated (by a message) must return values at the end of its activation, this is done by a **return message** (Dashed Arrow).

- The **return message** is therefore not a method call (it therefore does not cause the activation of an object).

- The **return message** often bears the name of the returned item.

Sequence Diagram: Example

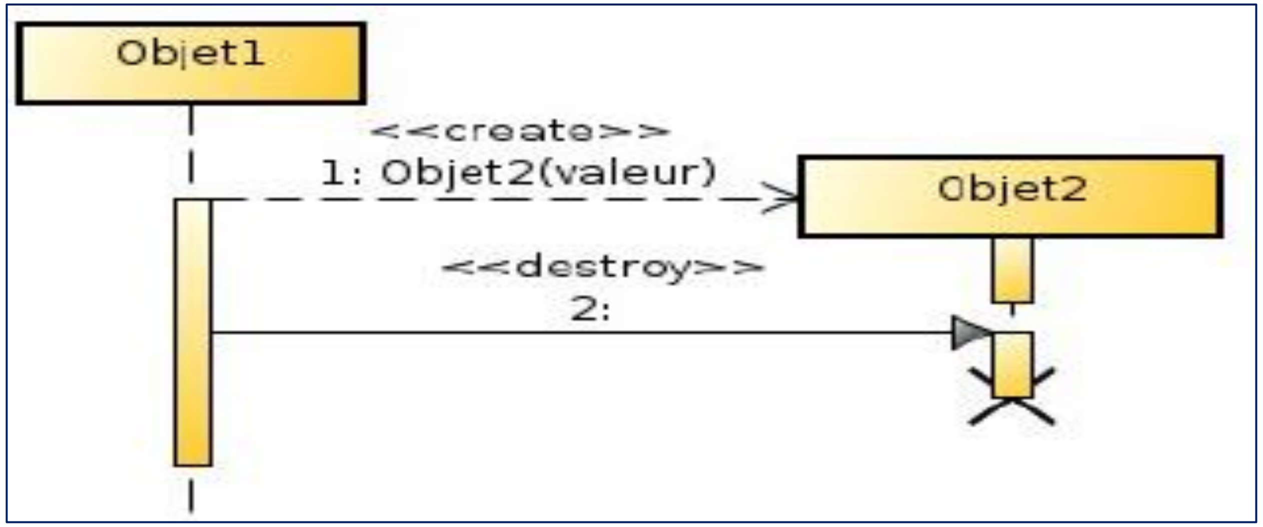

Message asynchrone (ex : le message5 est lancé alors que l'exécution du message1 n'est pas terminé)
Message synchrone (ex : on attend que l'exécution du message2 soit terminé avant de lancer le message4)
: Acteur
Objet1
Objet2
Objet3
Objet4
1: Message1()
1.1: Message2()
message3
2: Message5()
message6
1.2: Message4()
Message retour de l'exécution lancée par le message5

Sequence diagram: Creation and destruction of objects

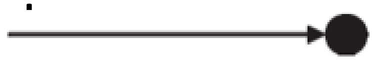
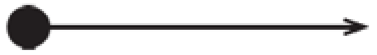
A sequence can also contain the **creation** or **destruction** of an object:

- The **creation of an object** is materialized by a **specific message**, call of a constructor, generally accompanied by the "**create**" **stereotype** which points to the beginning of the lifeline of the created object.

- The **destruction of an object** is represented by a **cross** at the end of its lifeline. Often the object is destroyed following the reception of a message, but this is not mandatory. In this case, it carries the stereotype "**destroy**".



Sequence diagram: Found message & Lost message

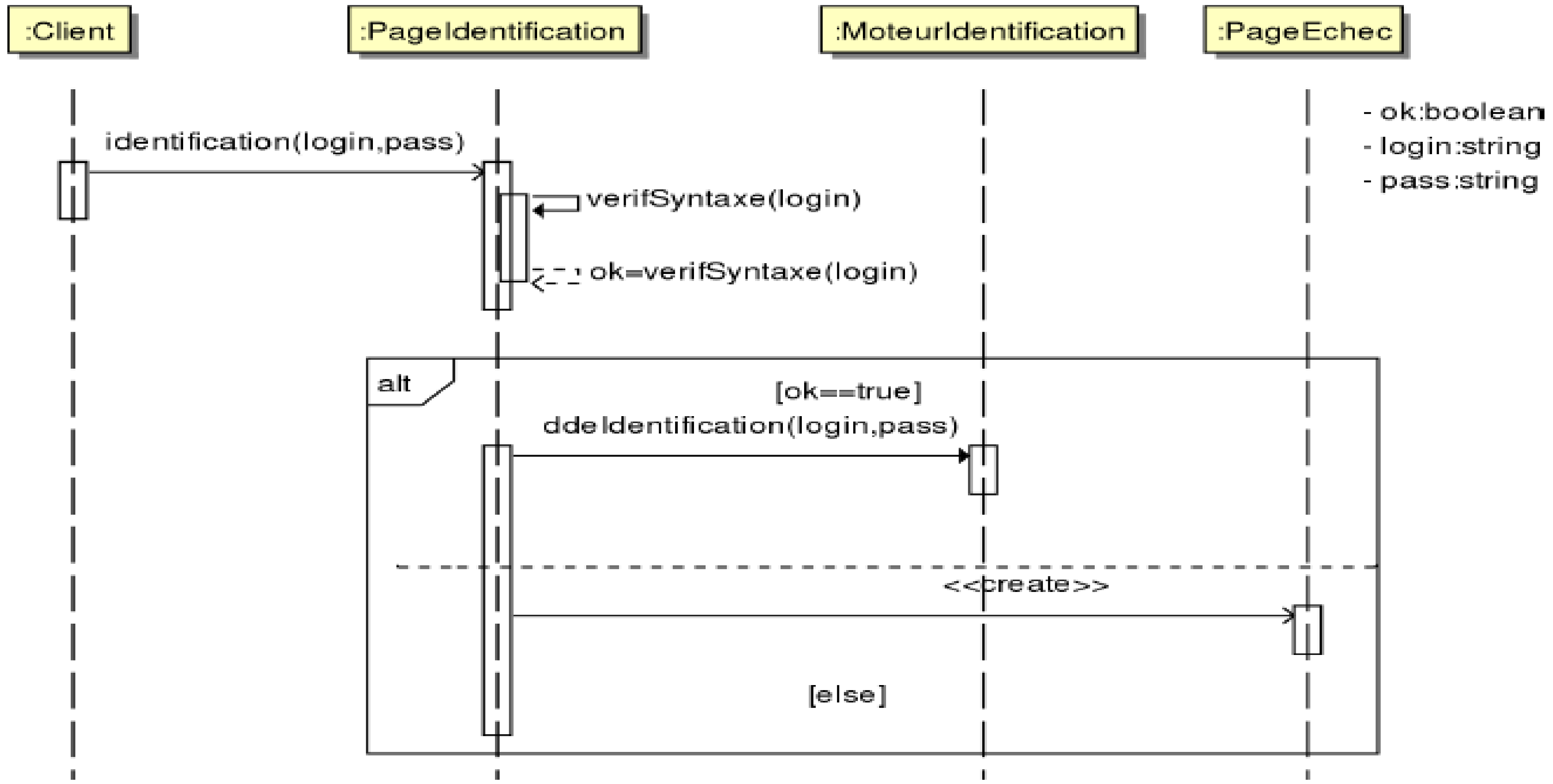
- ✓ A **complete message** is such that the send and receive events are known.
  - ✓ A **complete message** is represented by an arrow starting from one lifeline and arriving at another lifeline.
- ✓ A **lost message** is such that the sending events are known, but not the receiving events.
  - ✓ The arrow starts from a life line but arrives on an independent circle marking the ignorance of the recipie 
  - ✓ Example: broadcast
- ✓ A **message found** is such that the reception event is known but not the transmission event 

Sequence Diagram: Combined Fragment

- ✓A **combined fragment** breaks down a complex interaction into fragments that are <u>simple enough to understand</u>.
- ✓Recombining the fragments **restores the complexity**.
- ✓A **combined fragment** is represented in the same way as an interaction. It is represented by a **rectangle** whose <u>upper left corner contains a pentagon</u>.
- ✓Within the pentagon is **the type of the combination** (called the interaction operator).

Sequence Diagram: Combined Fragment

- Combined Fragment Example



Sequence Diagram: Combined Fragment

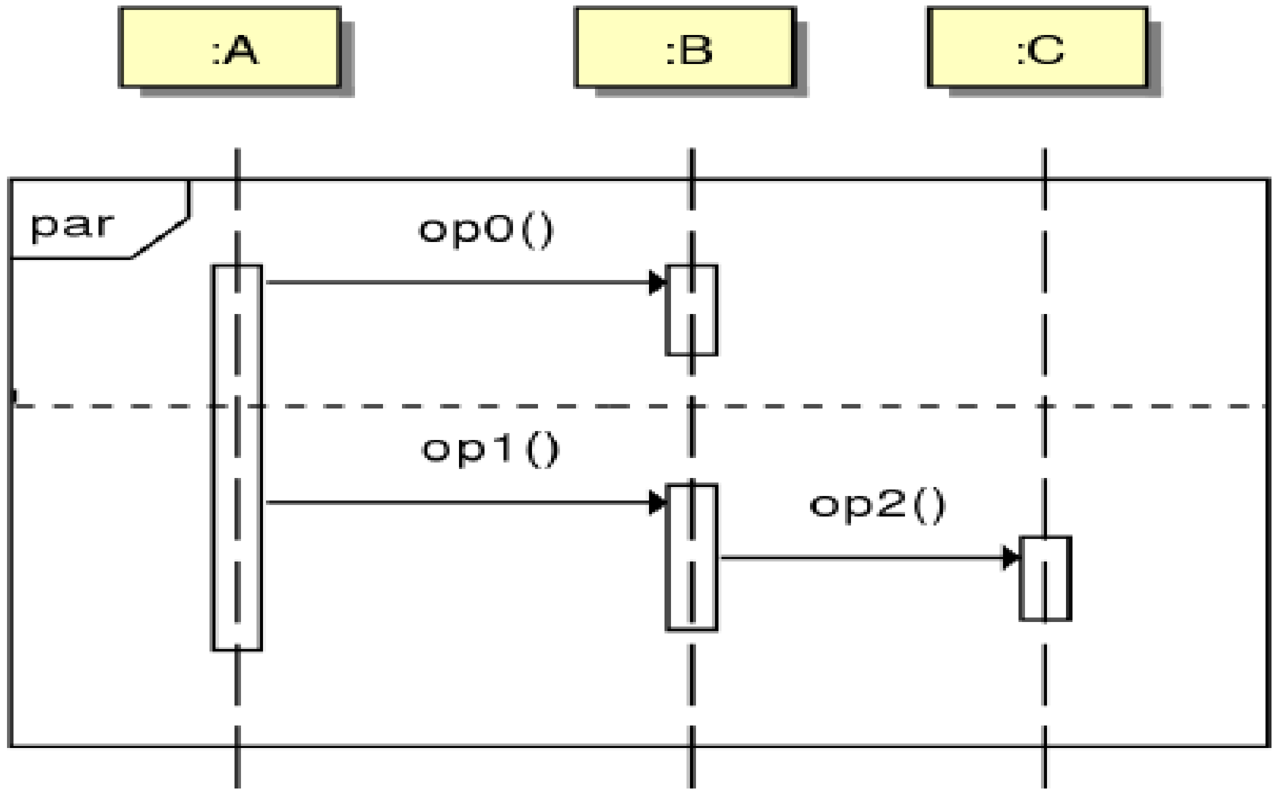
Type of interaction operators:

- ✓**Branching operators** (choices and loops): alternative, option, break and loop;
- ✓**Operators controlling the parallel sending of messages**: parallel and critical region;
- ✓**Operators controlling the sending of messages**: ignore, consider, assertion and negative;
- ✓**Operators determining the order in which messages are sent**: weak sequencing and strict sequencing.

Sequence Diagram: Combined Fragment

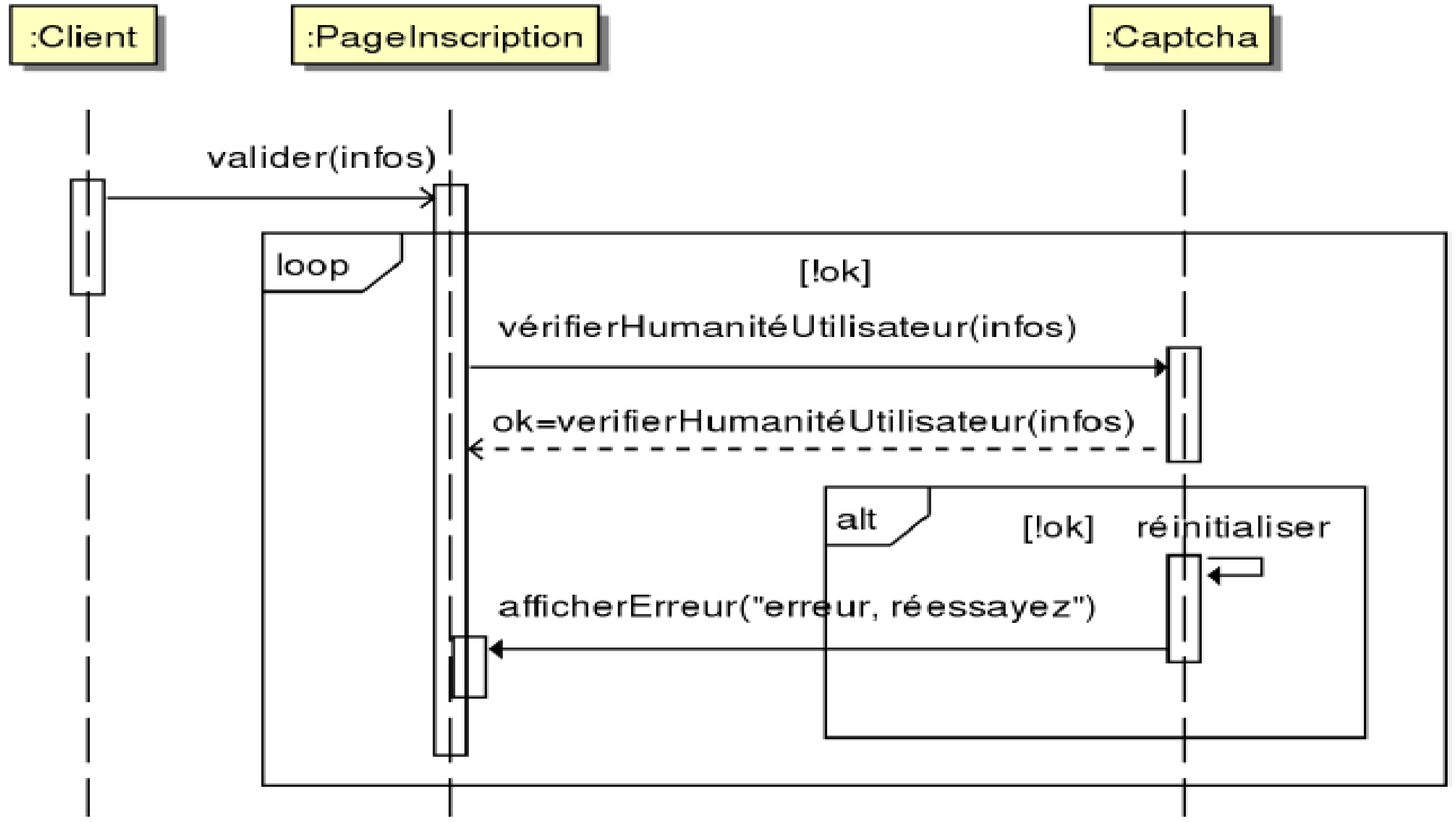
✓**Parallel operator**: The par operator allows sending messages in parallel.

✓What happens on both sides of the dotted line is independent.



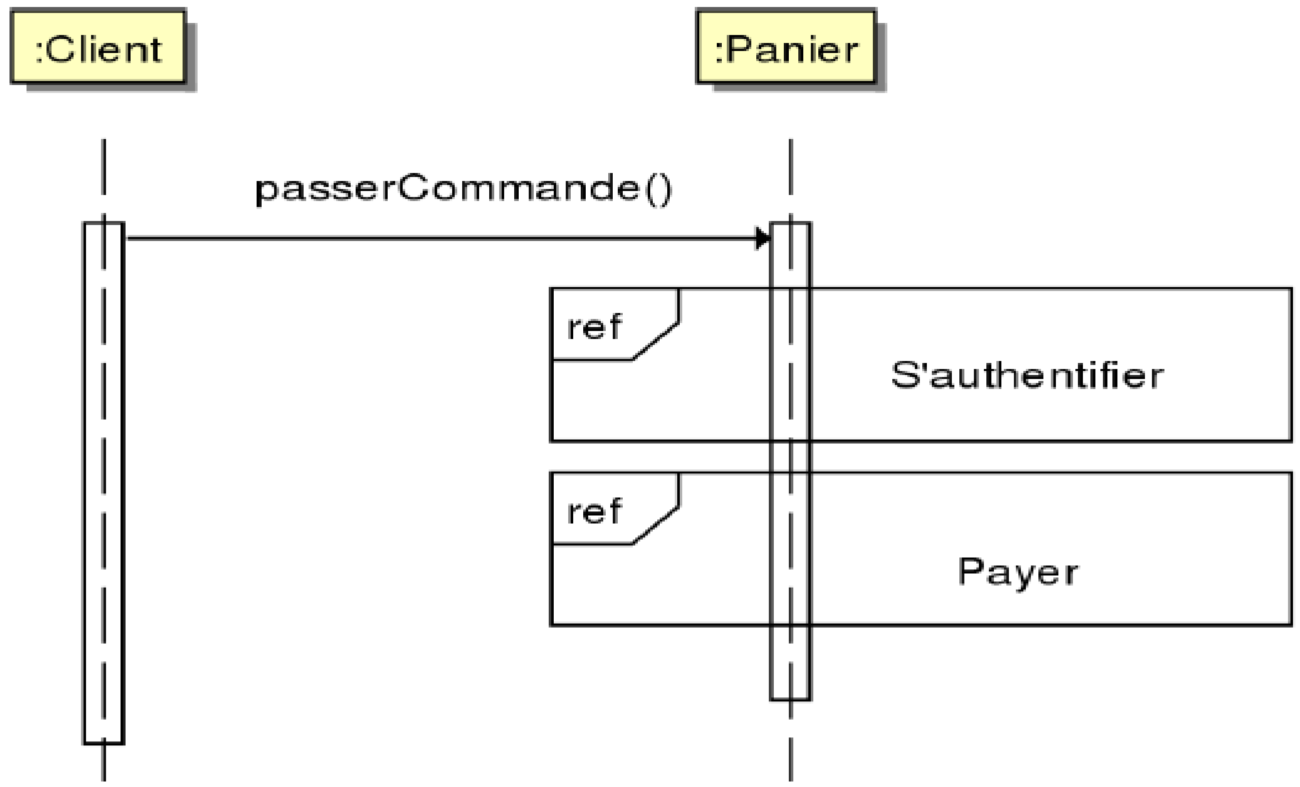
Sequence Diagram: Combined Fragment

- **Loop and alt fragments:**


:Client
:PageInscription
:Captcha
valider(infos)
loop
[!ok]
vérifierHumanitéUtilisateur(infos)
ok=verifierHumanitéUtilisateur(infos)
alt
[!ok]
réinitialiser
afficherErreur("erreur, réessayez")

Sequence Diagram: Combined Fragment

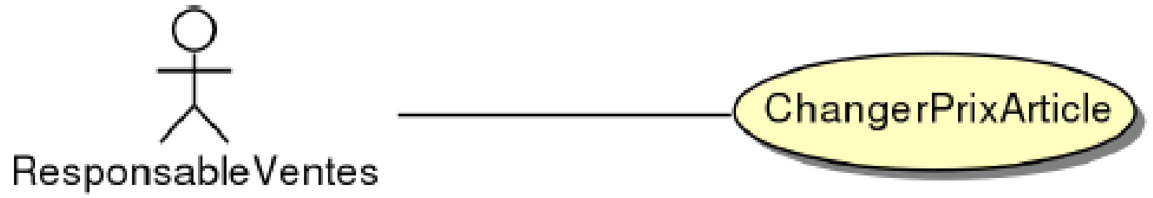
- **Reusing**: Reusing an interaction consists in **placing a fragment carrying the reference ref** where the interaction is useful.
- The **name of the interaction** is specified in the fragment.



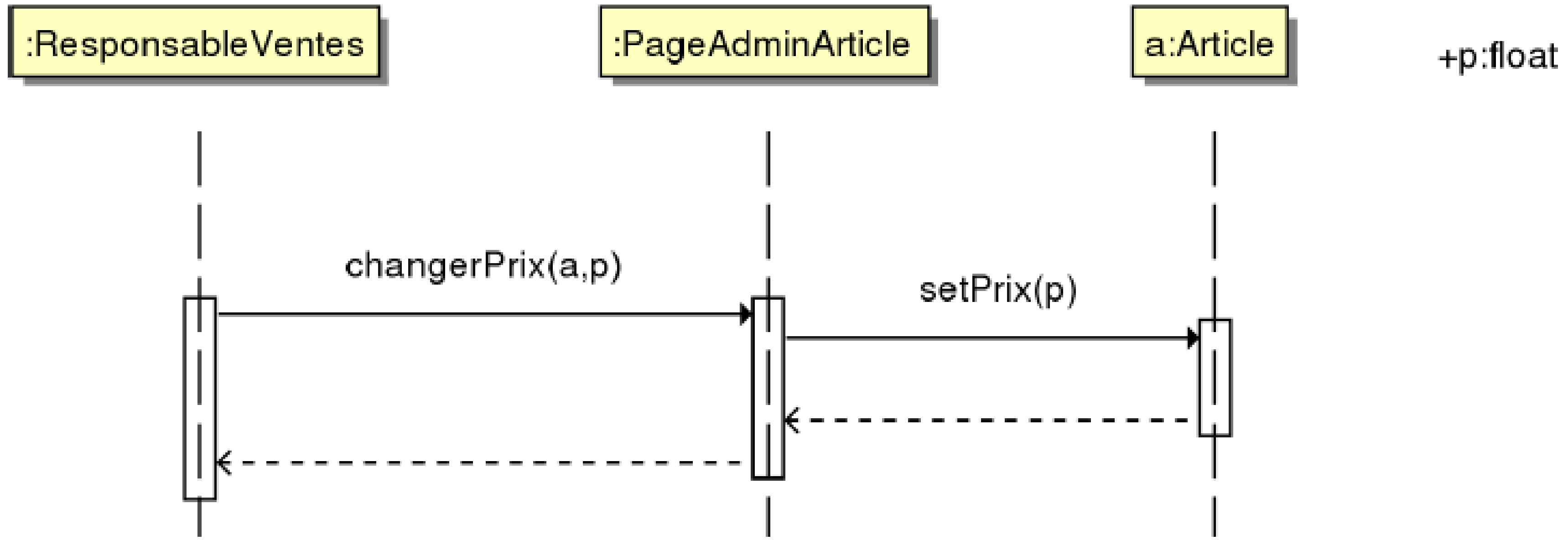
Sequence diagram: Example 1

Specifying a use case

Use Case:



Corresponding sequence diagram



Sequence diagram: Example 2

Specifying a complete sd for calculating the total of a an order

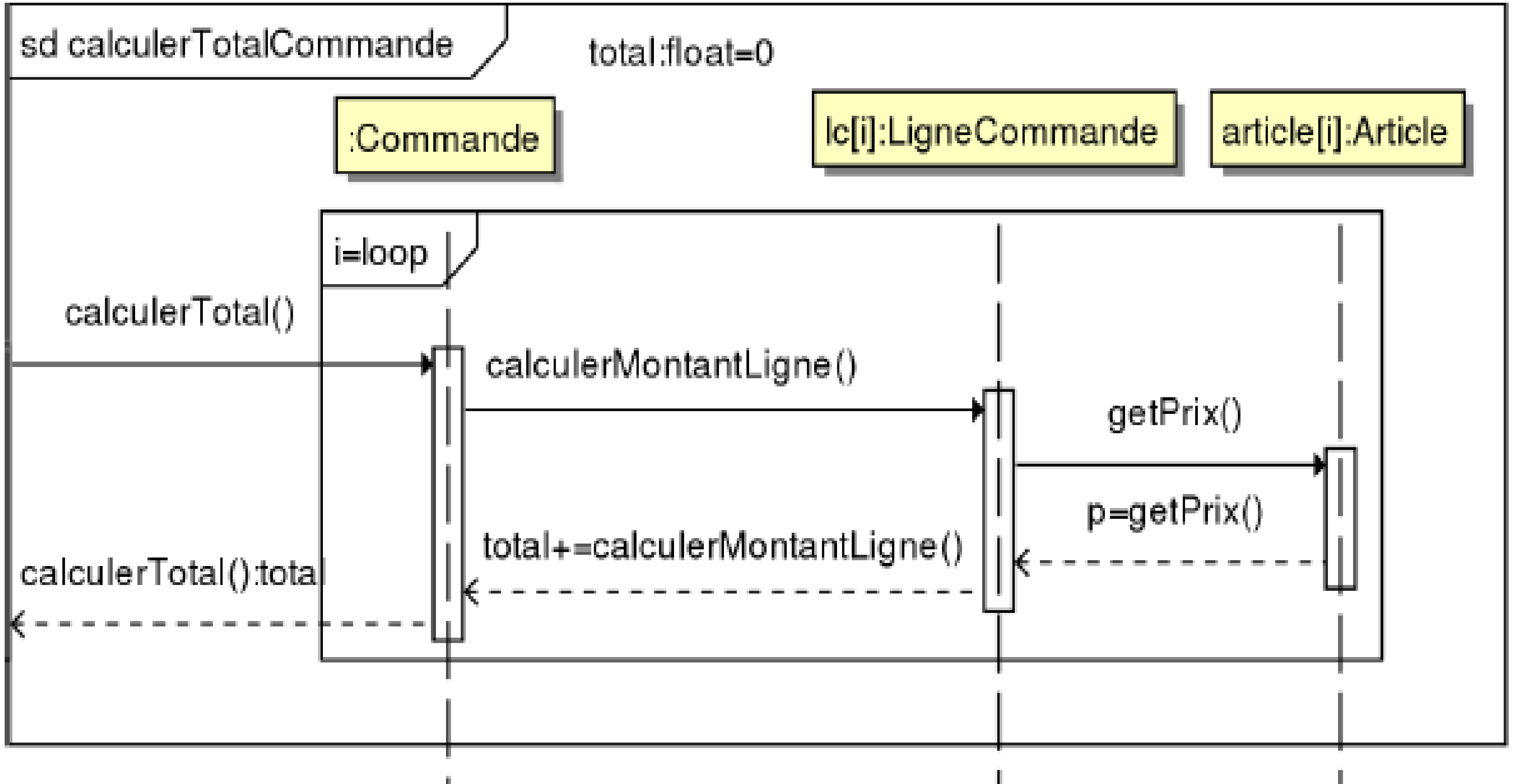
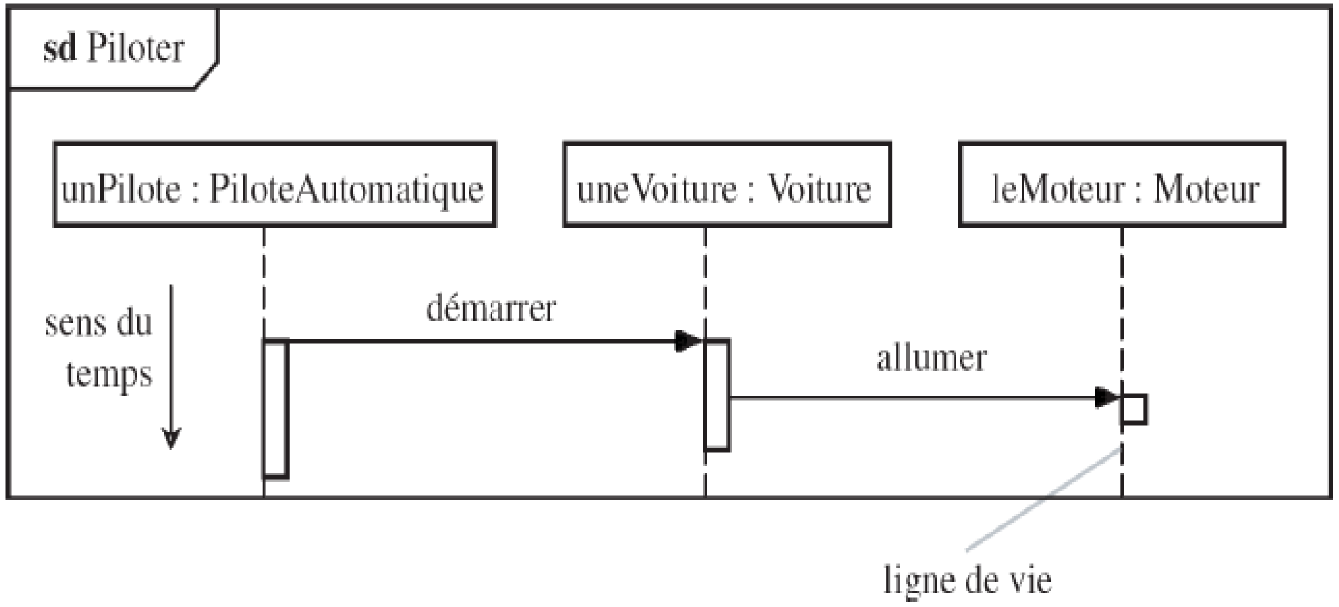
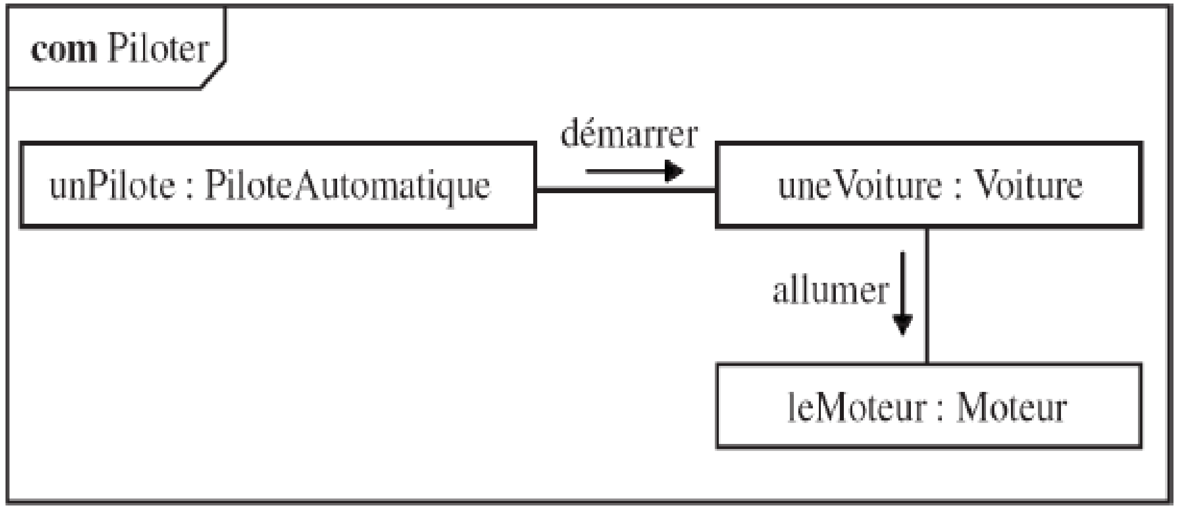


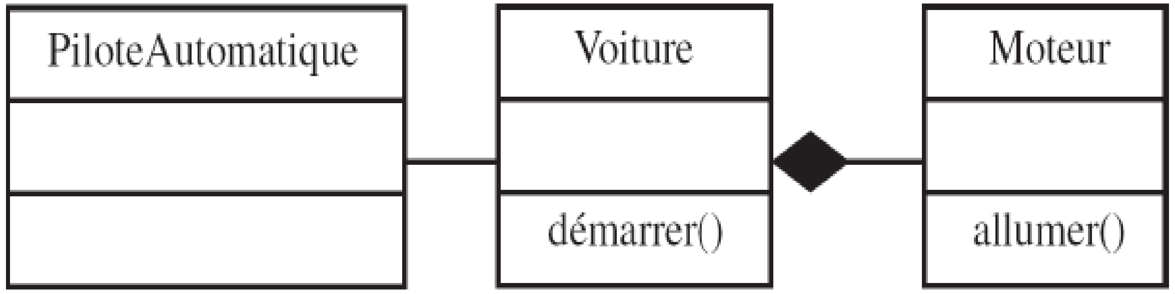
Diagramme de communication

- ✓**Communication diagrams** and **sequence diagrams** are two kinds of interaction diagram.
- ✓A **sequence diagram** shows interactions from **a temporal perspective**, emphasizing the temporal sequencing of messages exchanged between lifelines.
- ✓A **communication diagram** shows a **spatial representation of lifelines**.
- ✓They **represent the same thing**, but in different forms.

Sequence Diagram & Communication Diagram







Communication Diagram: Components

- ✓**Role**
- ✓**Connector**
- ✓**Message**

Communication Diagram: Components (Role & Connector)

✓The **role** defines the context of use of the interaction.

✓A **role** in a communication diagram corresponds to a lifeline in a sequence diagram.

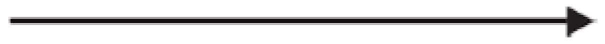
✓**Relationships between lifelines** are called **connectors**.

✓A **connector** is represented in the same way as an association but the semantics is broader: **a connector is often a transient association**.

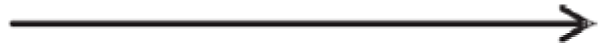
Communication Diagram: Components (Message)

As in the sequence diagram, two types of messages are distinguished:

✓A synchronous message blocks the sender until the receiver replies. The control flow passes from the sender to the receiver.

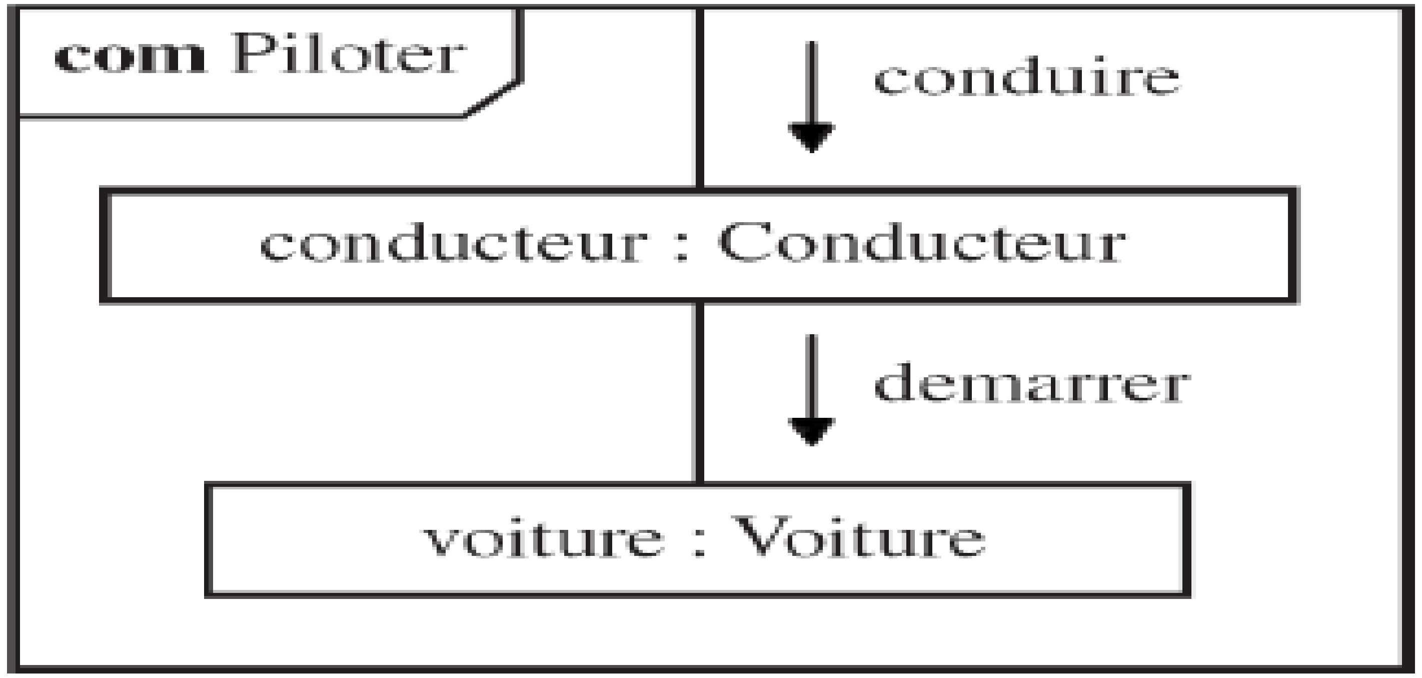


✓An asynchronous message is not blocking for the sender. The sent message can be taken into account by the receiver at any time or ignored.



Note: the distinction between messages and connectors: we could have a connector without a message, but never the reverse.

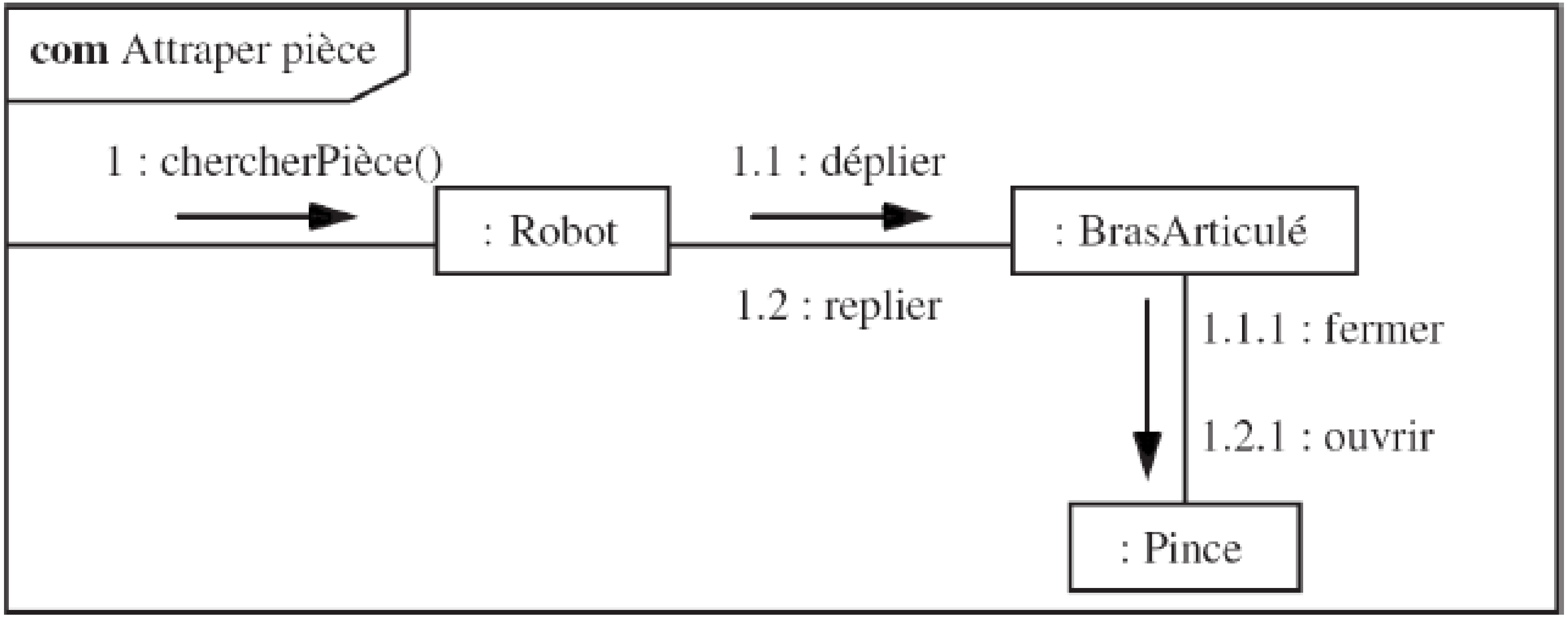
Communication Diagram: Components (Message)



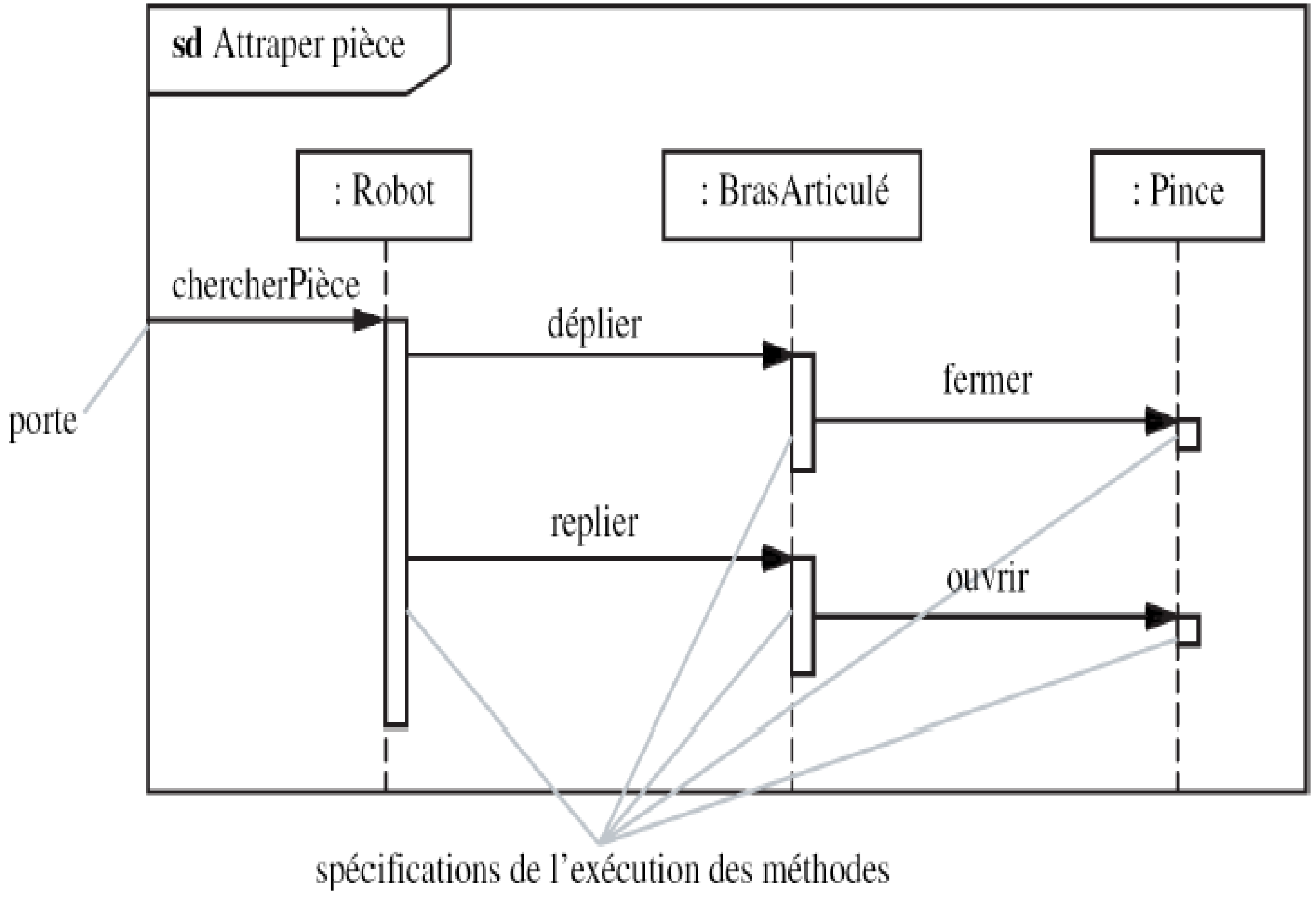
Communication Diagram: Components (Message)

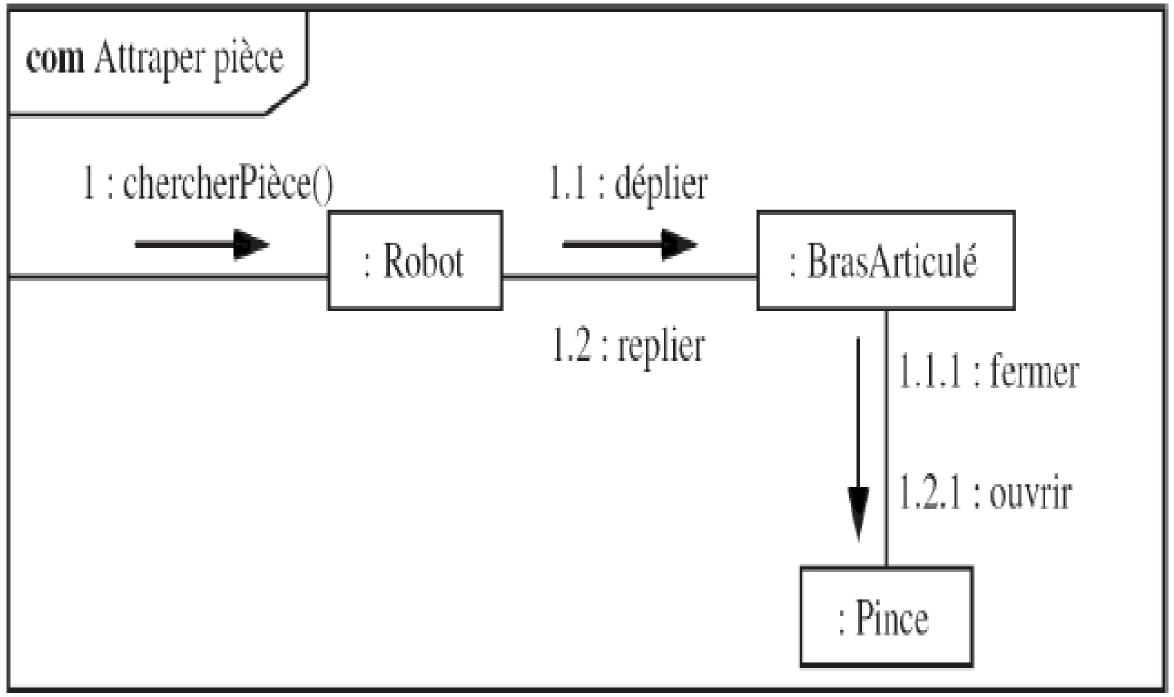
- ✓To **represent the temporal aspects**, <u>sequence numbers</u> are added to the messages.
- ✓**Successive messages** are ordered according to an <u>ascending sequence number</u> (1, 2, 3, ... or 3.1, 3.2, 3.3, ...).
- ✓**Messages sent in cascade** (ex: method call inside a method) <u>bear a nesting number with a dot notation</u>:
  - ✓1.1, 1.2, ... for messages called by a method whose call was number 1
  - ✓2.a.1, 2.a.2, ... for messages called by a method whose call was number 2.a

Communication Diagram: Example



Equivalence between Sequence Diagram and Communication Diagram

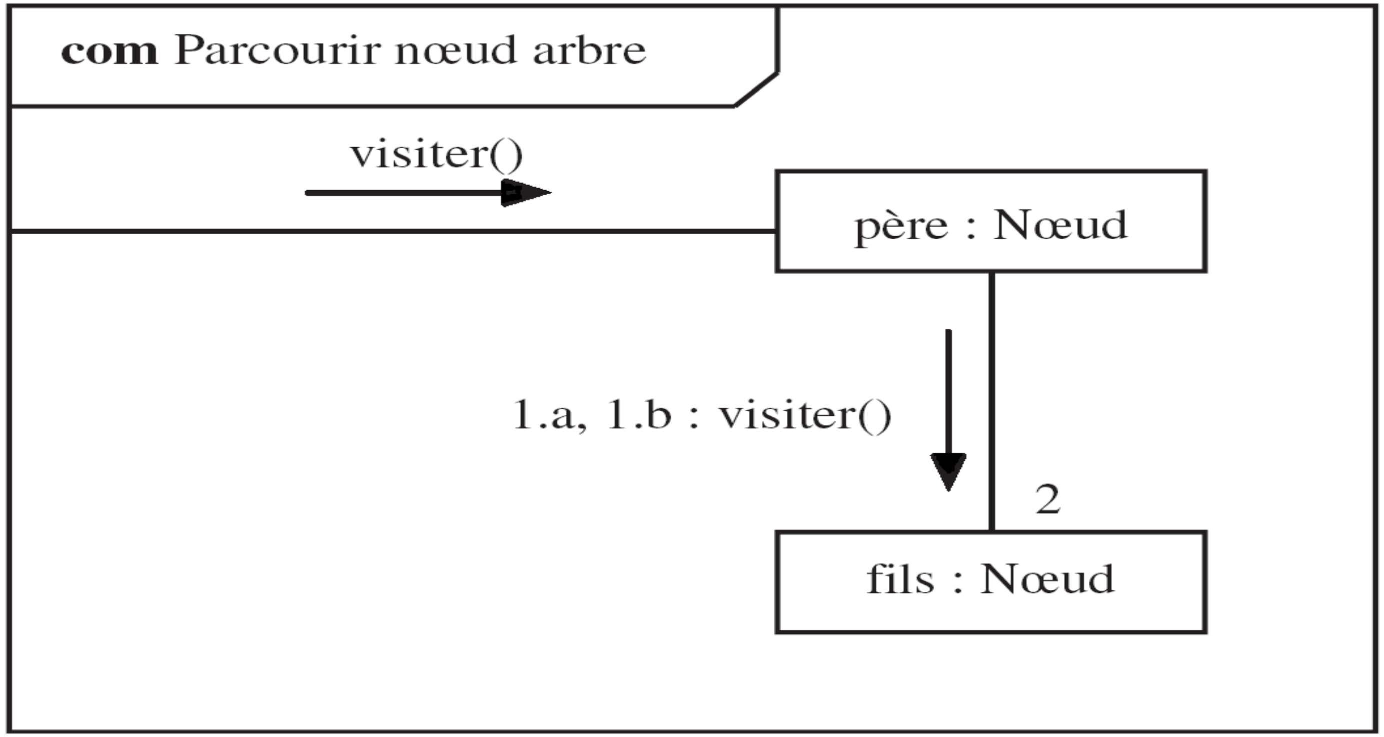




Communication Diagram: the components (Parallel Messages)

When messages are sent in parallel, they are numbered with letters

1.a, 1.b,... for simultaneous messages sent in response to a message whose sending was number 1



Communication diagram: the components (Choice and loop operators)

No combined operators in communication diagrams:

- *[<iterationclause>] represents **an iteration**.

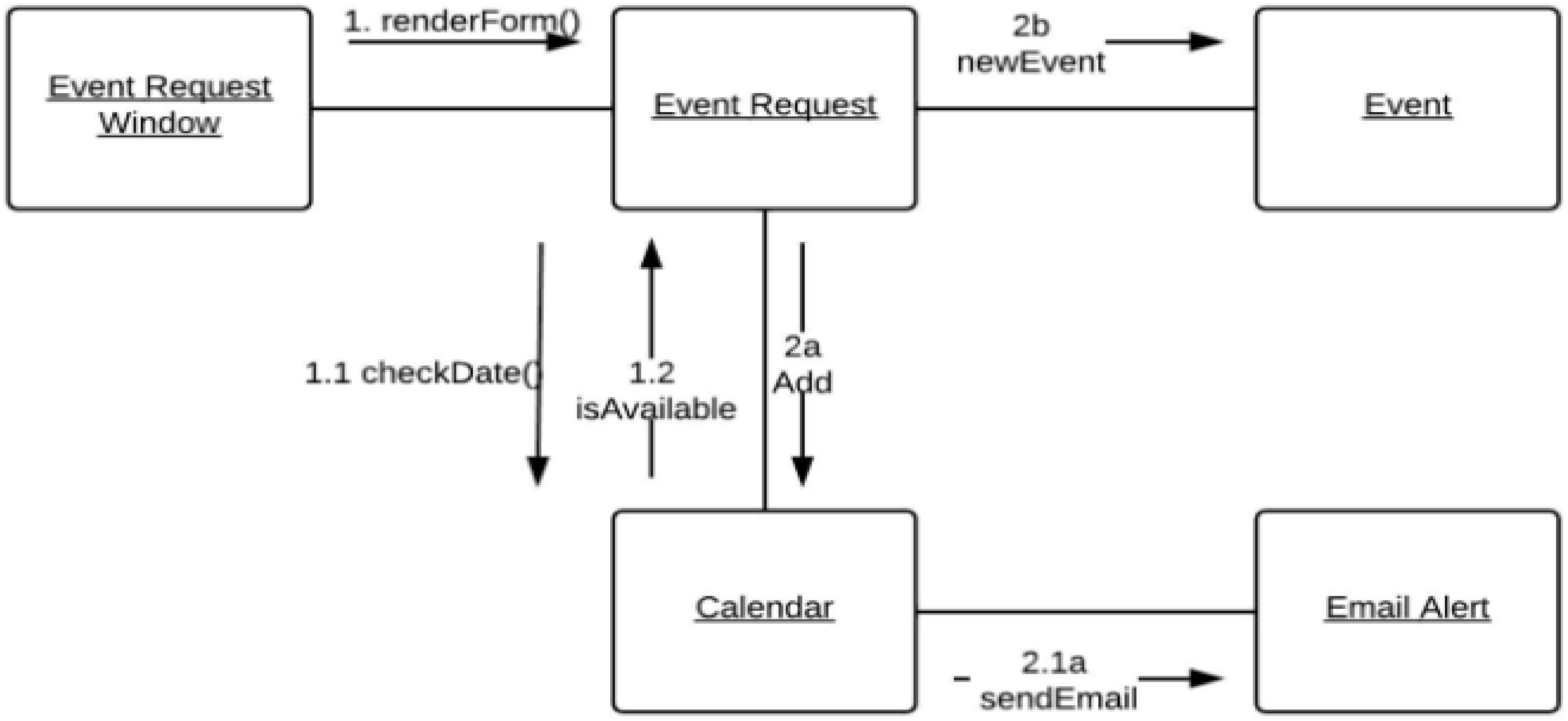
The iteration clause can be expressed in the format i:=1..n

If it is a boolean condition, this represents the stop condition

- *[<iterationclause>] represents a **choice**. The condition clause is a boolean condition.

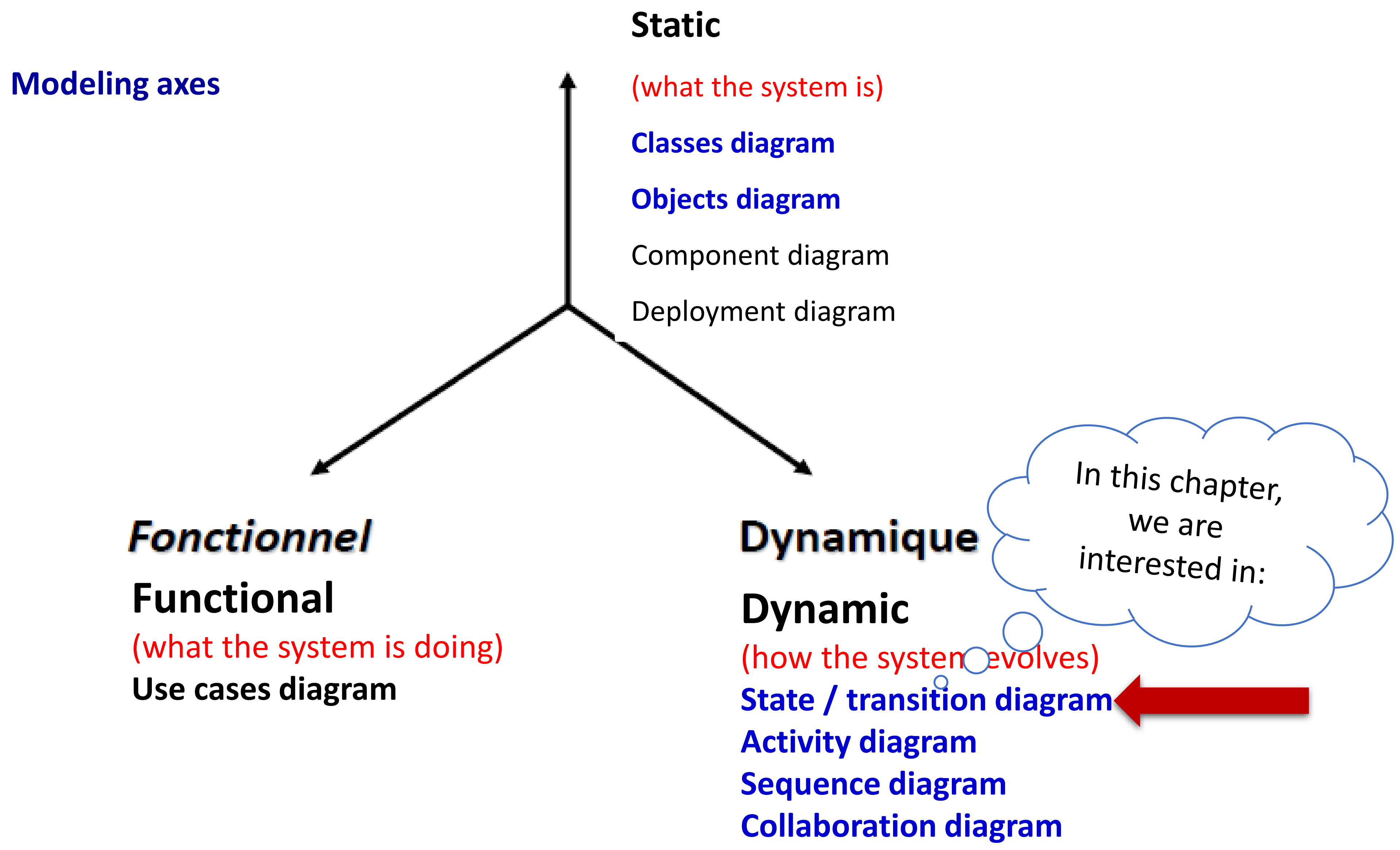
Example

- ✓This communication diagram explains the process to add an event to a calendar.
- ✓The numbers on each line represent the order and options in which they are activated.
- ✓Some actions happen concurrently because of the use of letters.



Part 2: State / transition diagram

✓Three modeling axes:



State/Transition Diagram

Un automate à états finis est la **spécification de la séquence d'états** que subira un **objet** au cours de son cycle de vie.

Un tel automate représente le **comportement** d'un **classeur** dont les sorties:

- ne dépendent pas seulement de ses entrées,

- mais aussi d'un **historique** des sollicitations passées.

Cet **historique** est caractérisé par un état.

Les objets <u>**changent d'état en réponse à des événements extérieurs**</u> donnant lieu à des transitions entre états.

State/Transition Diagram

✓A **finite state automaton** is the specification of the **sequence of states** that an object will undergo during its life cycle.

✓Such an automaton **represents the behavior of a classifier** whose <u>outputs</u>:

- <u>do not depend only on its inputs</u>,

- but also a history of past solicitations.

- This history is characterized <u>by a state</u>.

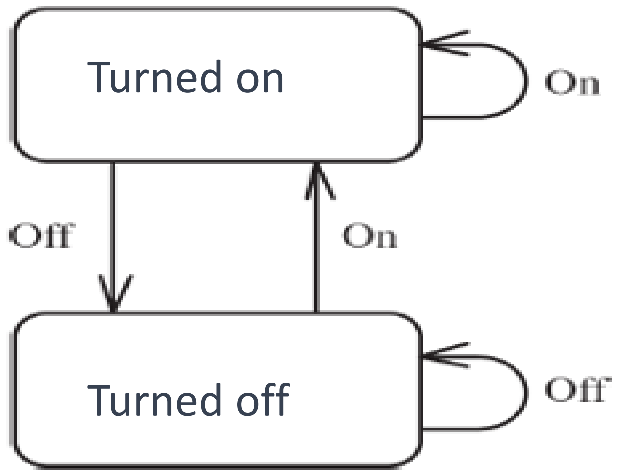
✓**Objects change state** <u>in response to external events</u> **giving rise to <u>transitions</u>** between states.

State/Transition Diagram

- ✓State Diagram are used to capture **the behavior** of a software system.
- ✓UML State machine diagrams can be used to model the behavior of a class, a subsystem, a package, or even an entire system.
- ✓ It is also called a **Statechart** or **State Transition diagram**.

State/Transition Diagram

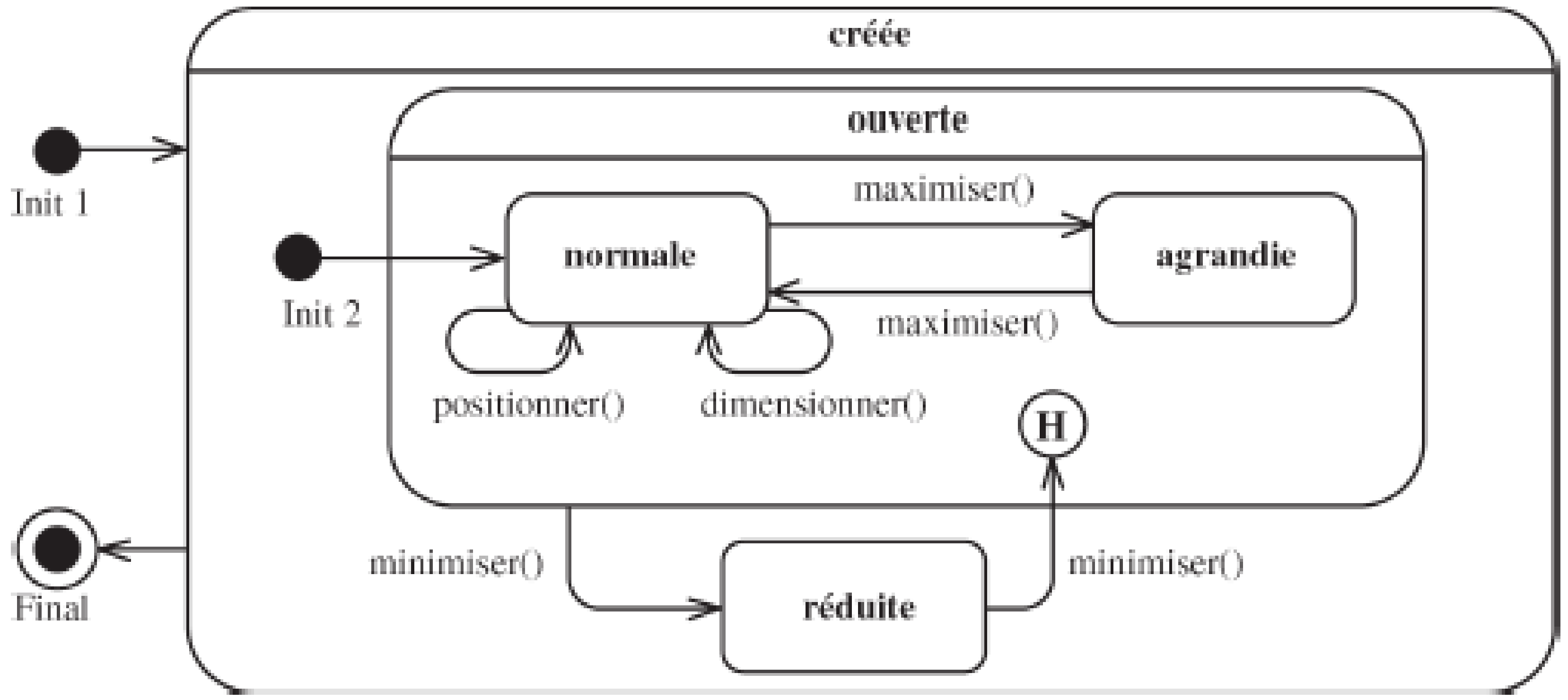
- ✓The states are represented by rectangles with rounded corners
- ✓The transitions are represented by oriented arcs linking the states together.
- ✓Some states, called composites, can contain sub-diagrams.
- ✓**Example**:



State/Transition Diagram

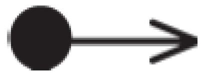
- ✓Organization of states and transitions for a given classifier is represented in a statechart diagram.
- ✓The **dynamic model** includes <u>several state diagrams</u>.
- ✓Each **state diagram** concerns <u>only one class</u>.
- ✓Each **automaton has finite states**, <u>runs concurrently</u>, and <u>can change state independently of the others</u>.

State/Transition Diagram (Example)

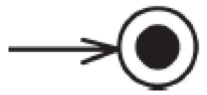


State/Transition Diagram

- The **initial state** defines the default starting point for the automaton or substate.
- When an object is created, it goes into the initial state:
- It is represented by:



- The final state indicates that the execution of the automaton or substate has ended:
- It is represented by:



State/Transition Diagram

Transitions in a Statechart diagram are triggered by trigger events.

- ✓A method call on the current object generates an event of type "call".
- ✓Changing from false to true the truth value of a Boolean condition implicitly generates a "change" type event.
- ✓The reception of an asynchronous signal, explicitly emitted by another object, generates an event of type "signal".
- ✓The passage of a fixed duration after a given event generates an event of type "after". By default, time begins to elapse upon entering the current state.

The trigger event is indicated next to the arrow representing the transition

State/Transition Diagram

Simple transition:

- ✓A transition between two states is represented by an arc that links them to each other.
- ✓It indicates that an instance can change state and perform certain activities, if a trigger event occurs and the guard conditions are met.

State/Transition Diagram

Decision Point:

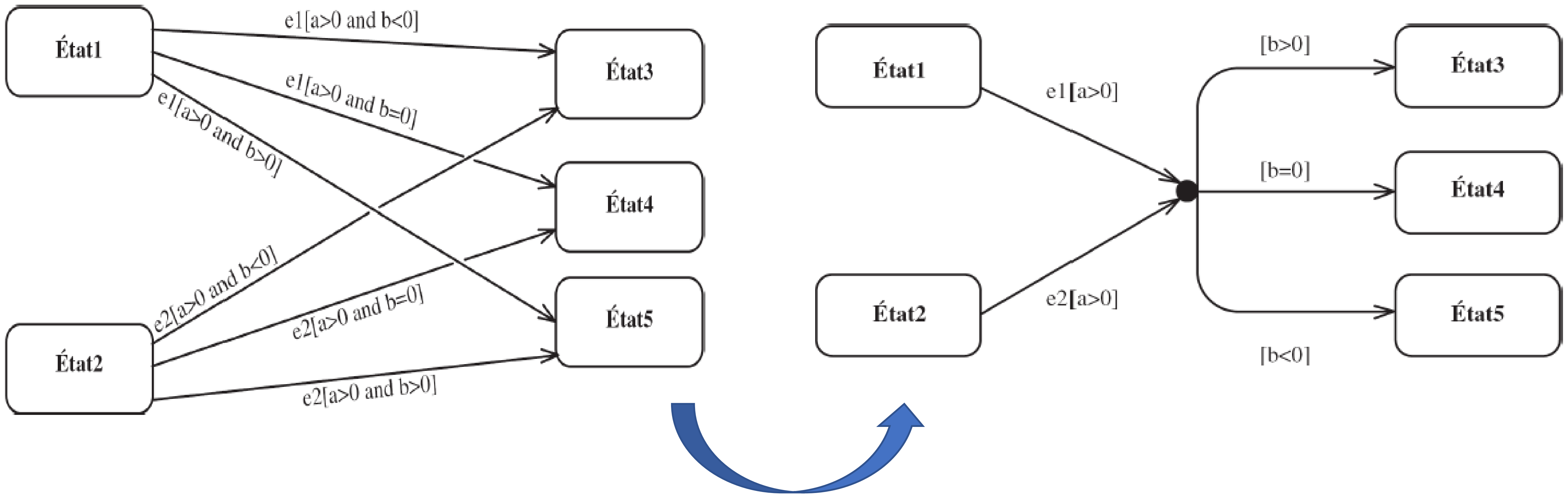
✓Alternatives can be represented for the crossing of a transition.

✓**For this, we use special pseudo-states:**

- ✓**The junction points** (small solid circle) allow to share transition segments.
  - They are just a writing shortcut.
  - They allow more compact representations.
- ✓**Choice points (diamond)** are more than writing shortcuts.

State/Transition Diagram

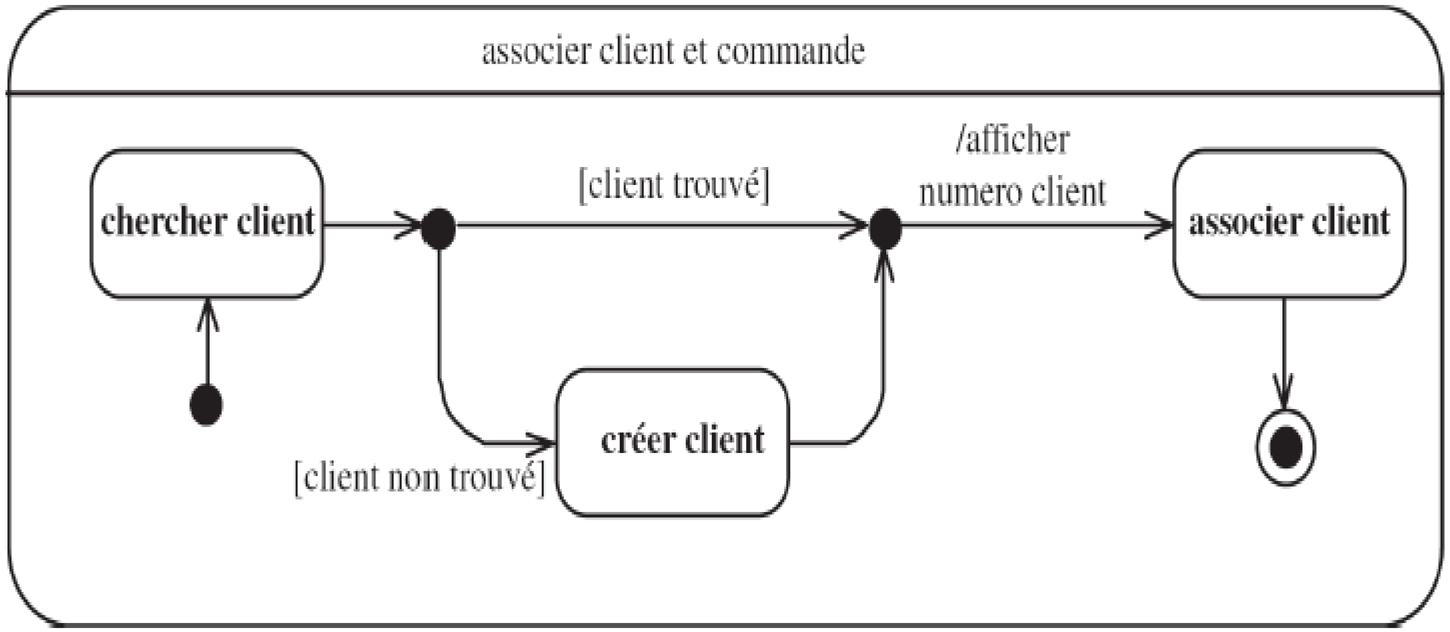
Junction Bridge



To take a path, all guards along that path must evaluate to true upon crossing the first segment.

State/Transition Diagram

Example of a Junction Point



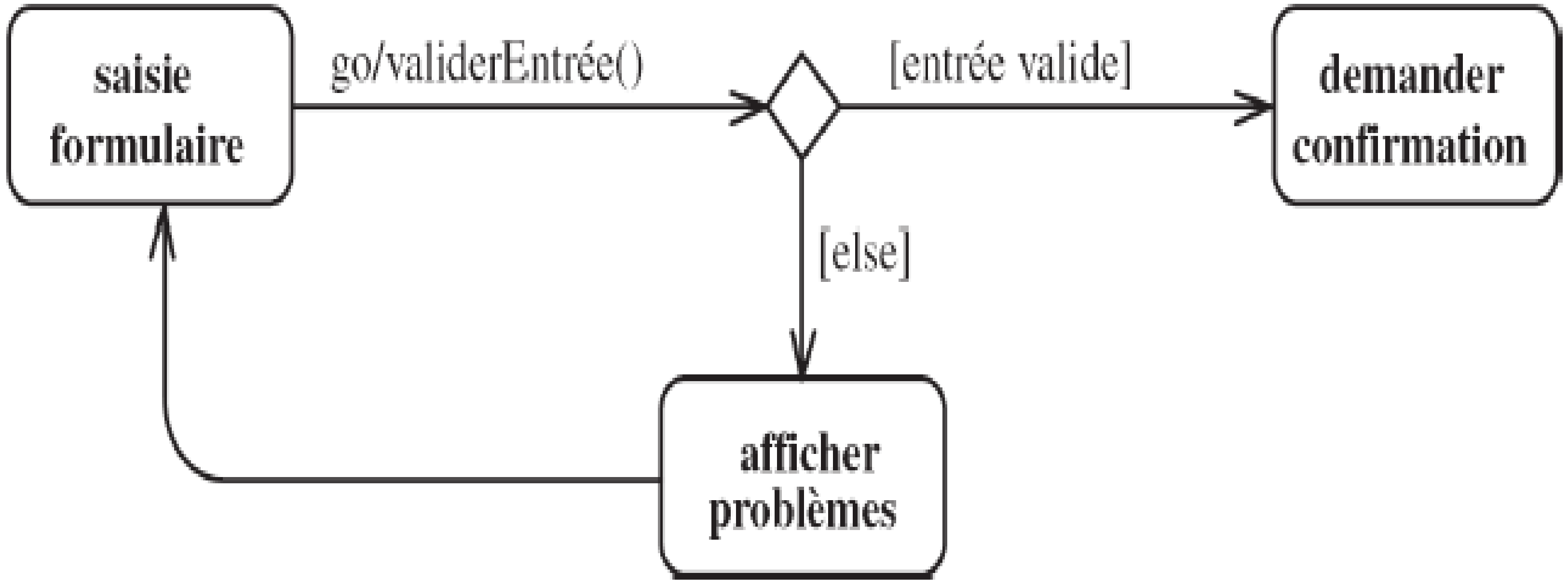
State/Transition Diagram

choice point

- ✓The **guards after the choice point** are evaluated when they are reached.
- ✓This makes it possible to **base the choice on the results obtained** by crossing the segment before the choice point.
- ✓If, when the choice point is reached, no downstream segment is passable, the **model is poorly formed**.
- ✓Unlike junction points, choice points are not simple writing shortcuts.

State/Transition Diagram

Example of a Choice Point



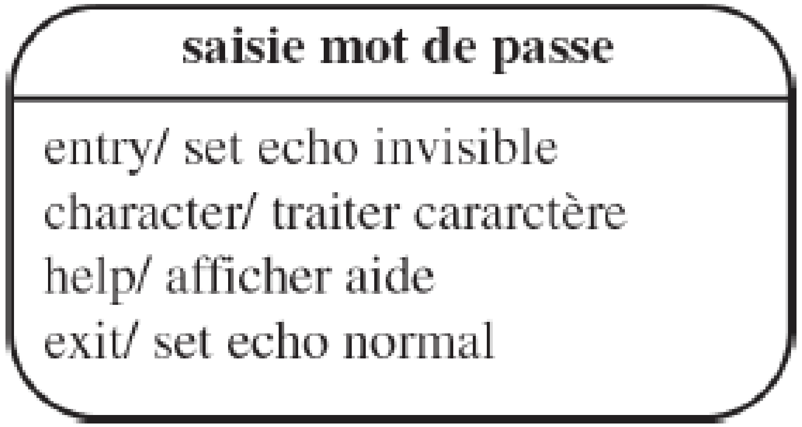
State/Transition Diagram

Internal Transition

✓An object **remains in a state for a certain period** of time and internal transitions may occur.

✓An **internal transition** does not modify the current state, but generally follows the rules of a simple transition between two states.

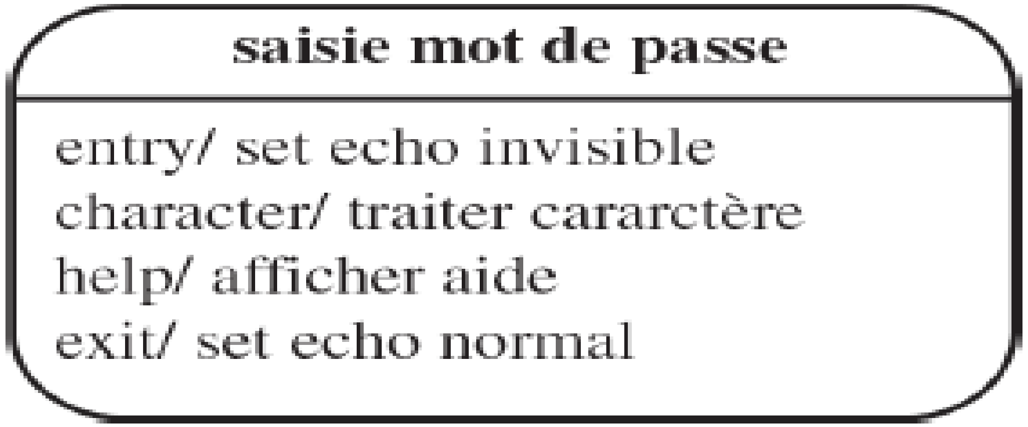
✓**Three special triggers** are introduced allowing the firing of internal transitions: **entry/**, **do/**, and **exit/**.



State/Transition Diagram

Internal transition

- **entry:** an activity to be carried out **<u>each time you enter the state</u>** in question.
- **exit:** activity to execute **<u>when exiting the state</u>**.
- **do:** continuous activity which is carried out **as long as we are in the state**, or <u>until the associated calculation is finished</u>.

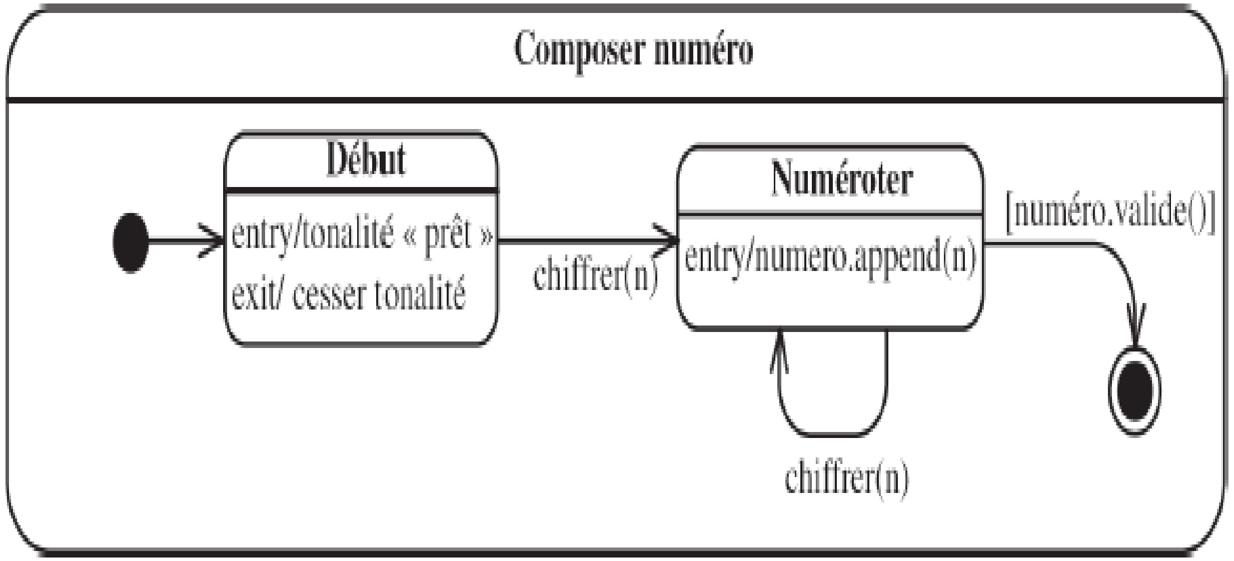


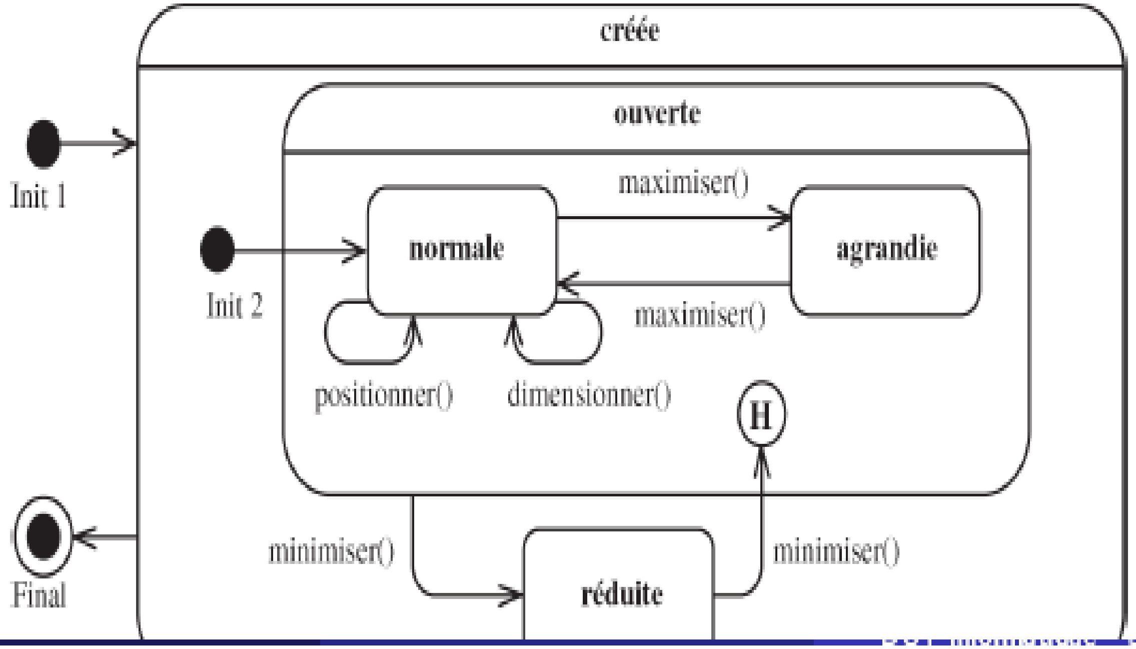
State/Transition Diagram

Composite state

✓**A composite state**, as opposed to a so-called simple state, is <u>decomposed into two or more sub-states</u>.

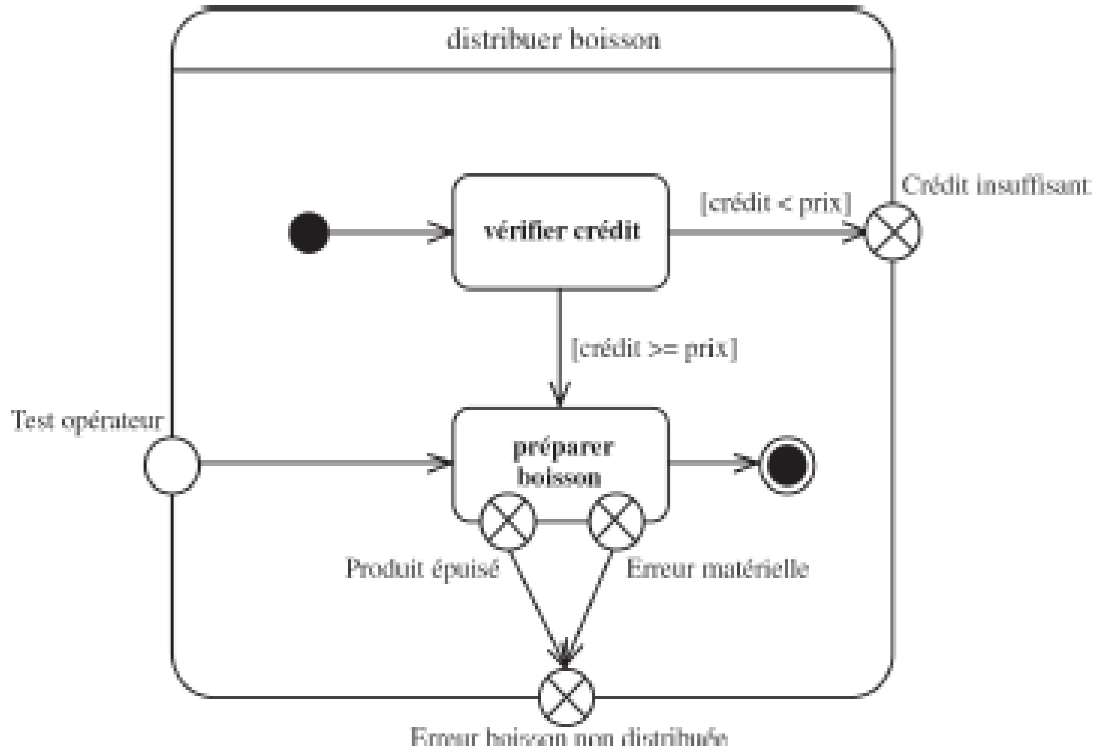
✓A **composite state** is represented as follows:





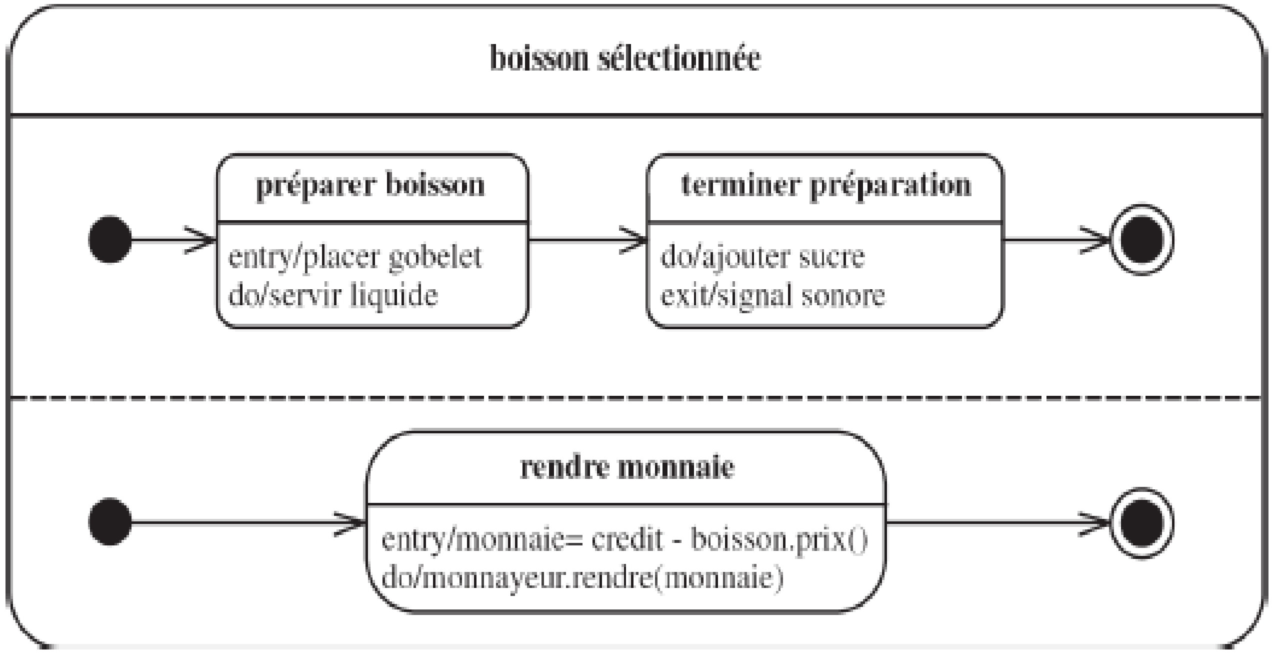
State/Transition Diagram

To be able to represent a sub-state independently of a macro-state, we use **<u>connection points</u>**.
With an **X** for the **<u>exit</u>** **<u>points</u>**.
empty for entry points
These interfaces make it possible to abstract the sub-states from the macro-states (reusability).



State/Transition Diagram

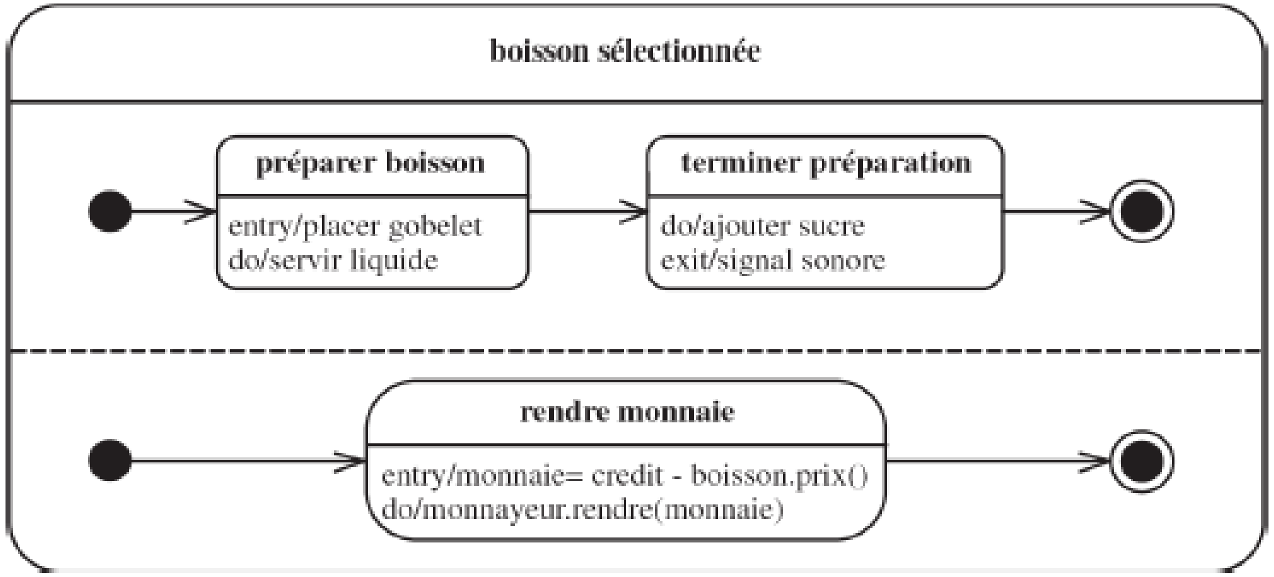
Concurrent State:
- With a dotted separator, we can represent several automata executing independently.
- An object can then be simultaneously in several concurrent states.



State/Transition Diagram

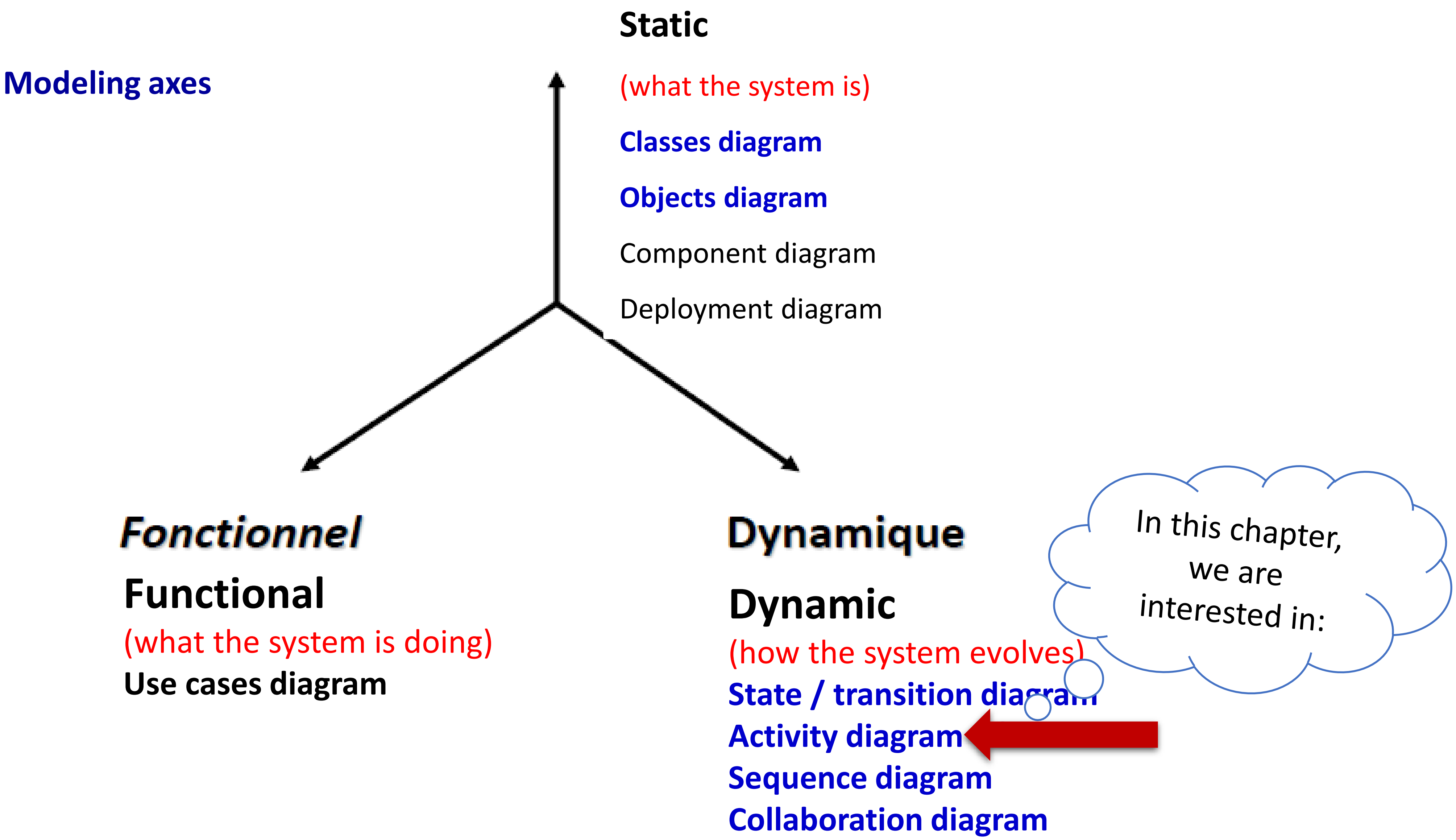
Concurrent Transition:

- ✓ A fork transition corresponds to the creation of two concurrent states.
- ✓ A join transition is a synchronization barrier that removes concurrency.
- ✓ To be able to continue their execution, all the concurrent tasks must first be ready to cross the transition.



Part 3: Activity Diagram

✓Three modeling axes:



Activity Diagram

- ✓Activity diagrams provide a graphical and unambiguous way to model processes.
  - ✓ Method behavior
  - ✓Progression of a use case
- ✓An activity represents an execution of a mechanism, a sequence of sequential steps. The passage from one activity to another other is materialized by a transition.
- ✓These diagrams are quite similar to state-transitions but with a different interpretation.

Activity Diagram

- ✓Statechart diagrams are defined for each classifier and do not involve more than one.
- ✓Conversely, activity diagrams allow a description that (partially) overcomes the structuring of the application in classifiers.
- ✓The vision of activity diagrams is close to imperative programming languages (C, C++, Java)
    - ✓States represent calculations
    - ✓There are no external events but waiting for the end of calculations.
    - ✓There may be concurrency between activities.

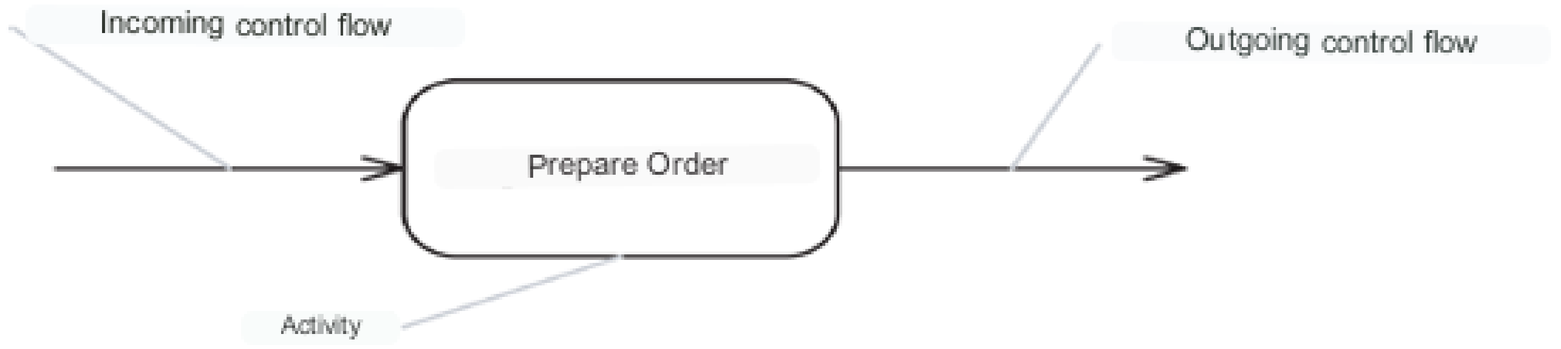
Activity Diagram

Activities describe a treatment.

✓The control flow remains in the activity until the processing is finished.

✓You can define variables local to an activity and manipulate the variables accessible from the context of the activity (containing class in particular).

✓The activities can be hierarchically nested, we then speak of composite activities.



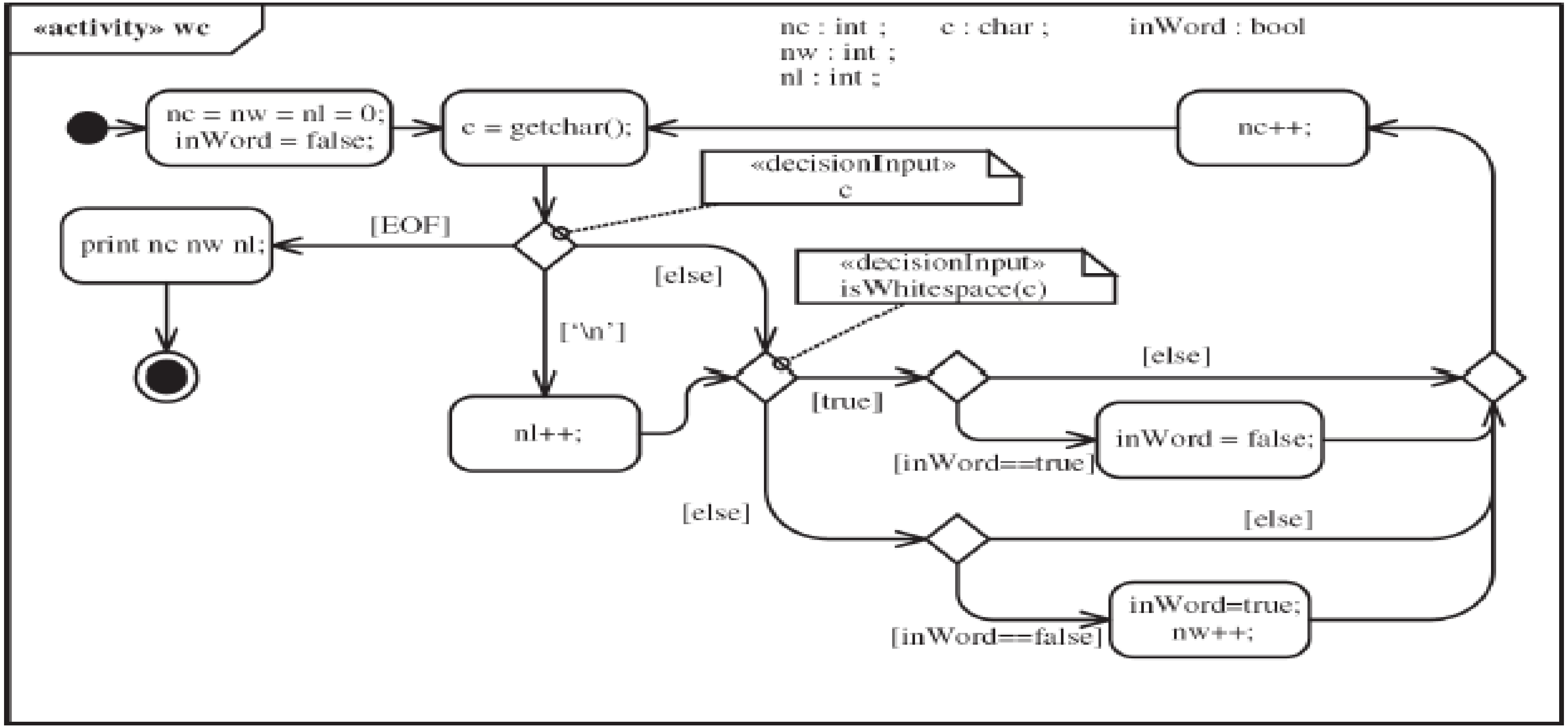
Activity Diagram (Transition)

- ✓Transitions specify the sequence of processing and define the flow of control.
- ✓They are represented by solid arrows that connect the activities together.
- ✓They are triggered as soon as the source activity is completed.
- ✓They automatically cause the immediate start of the next activity to be triggered (the target activity).
- ✓Unlike activities, transitions are crossed atomically, in principle without perceptible duration.

Activity Diagram (Conditional Control Structure)

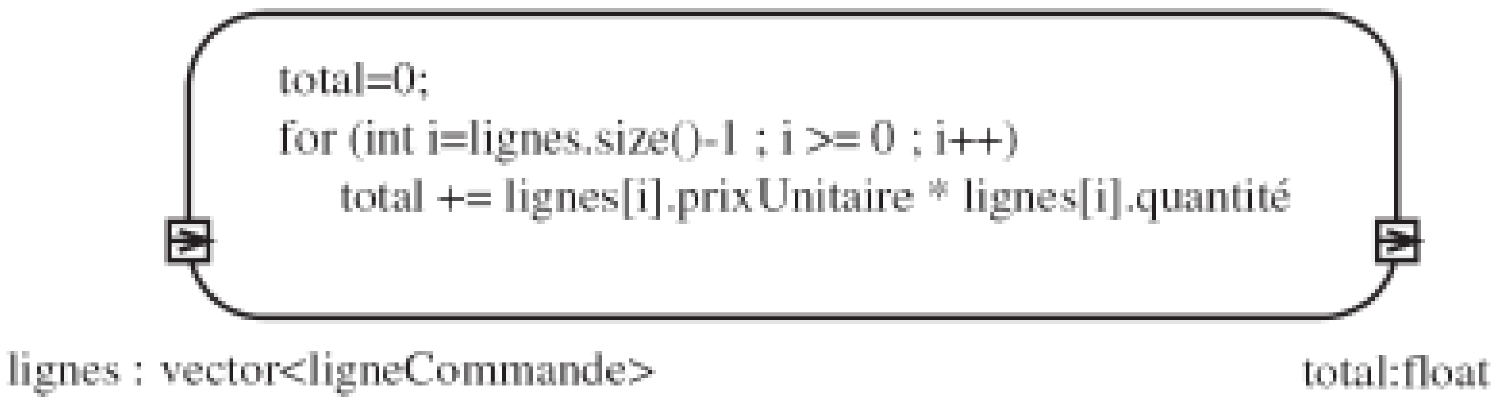
- ✓Expressing conditions using transitions with conditional guards
- ✓These transitions can only be taken if the guard is true.
- ✓We have an [else] clause which is validated if and only if all the other guards of the transitions having the same source are false.
- ✓Conditions are noted in square brackets.
- ✓To better highlight a conditional branch, we can use the choice points (diamonds).
- ✓Choice points show a switch of the control flow.

Activity Diagram (Conditional Control Structure -Example)

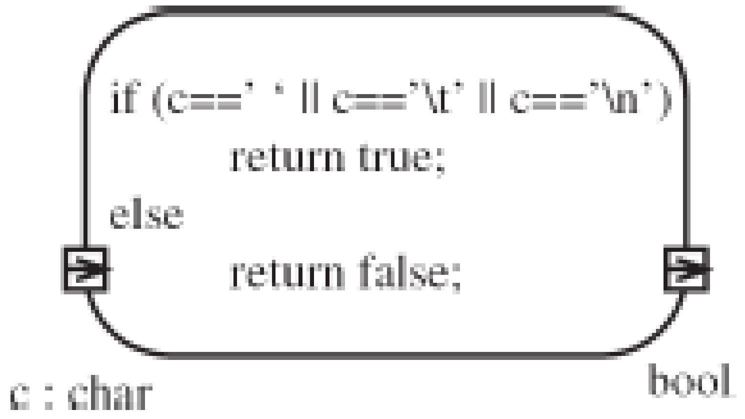


Activity diagram (structured activities)

- **Structured activities** use the usual control structures (conditional and loop) through a syntax that depends on the tool.
- The precise syntax of these annotations is not defined in the UML standard.
- In a structured activity, the input arguments and the outputs are defined **by boxed arrows**.
- **Example**:



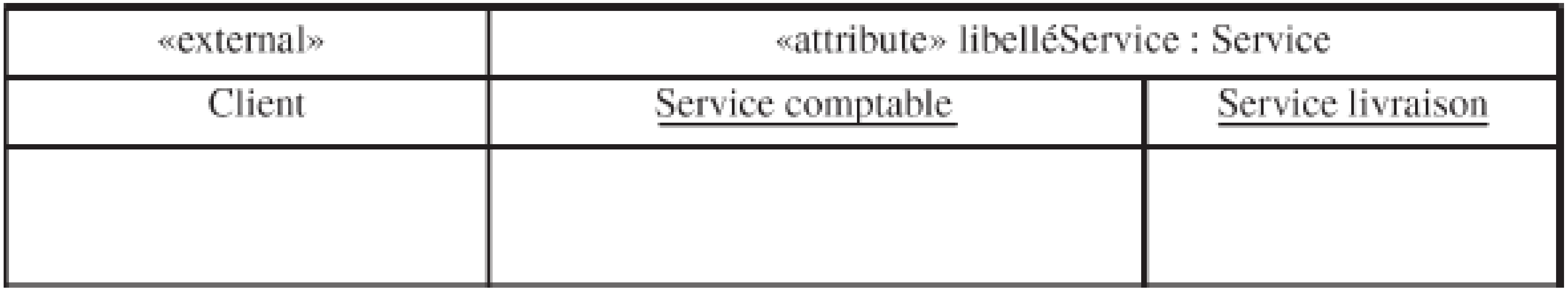
```
total=0;
for (int i=lignes.size()-1 ; i >= 0 ; i++)
    total += lignes[i].prixUnitaire * lignes[i].quantité
```



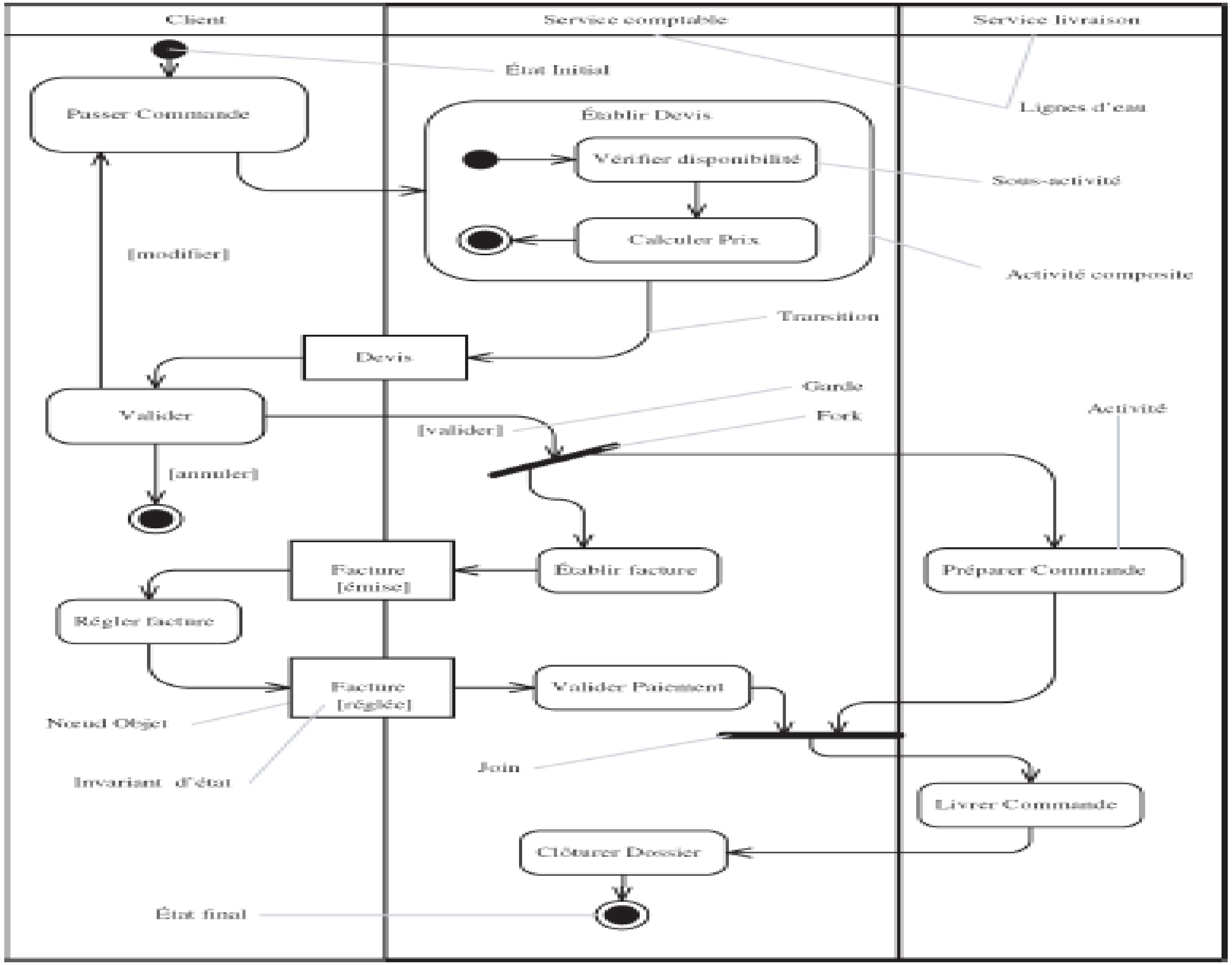
```
if (c==' ' || c=='\t' || c=='\n')
        return true;
else
        return false;
```

Activity diagram (**partitions**)

- To model a process implementing several classifiers, the classifier responsible for each activity can be specified.
- Partitions allow activities to be assigned to specific elements of the model.
- A partition can itself be broken down into sub-partitions.
- To specify that an activity is carried out by a particular classifier, it is positioned in the corresponding partition.

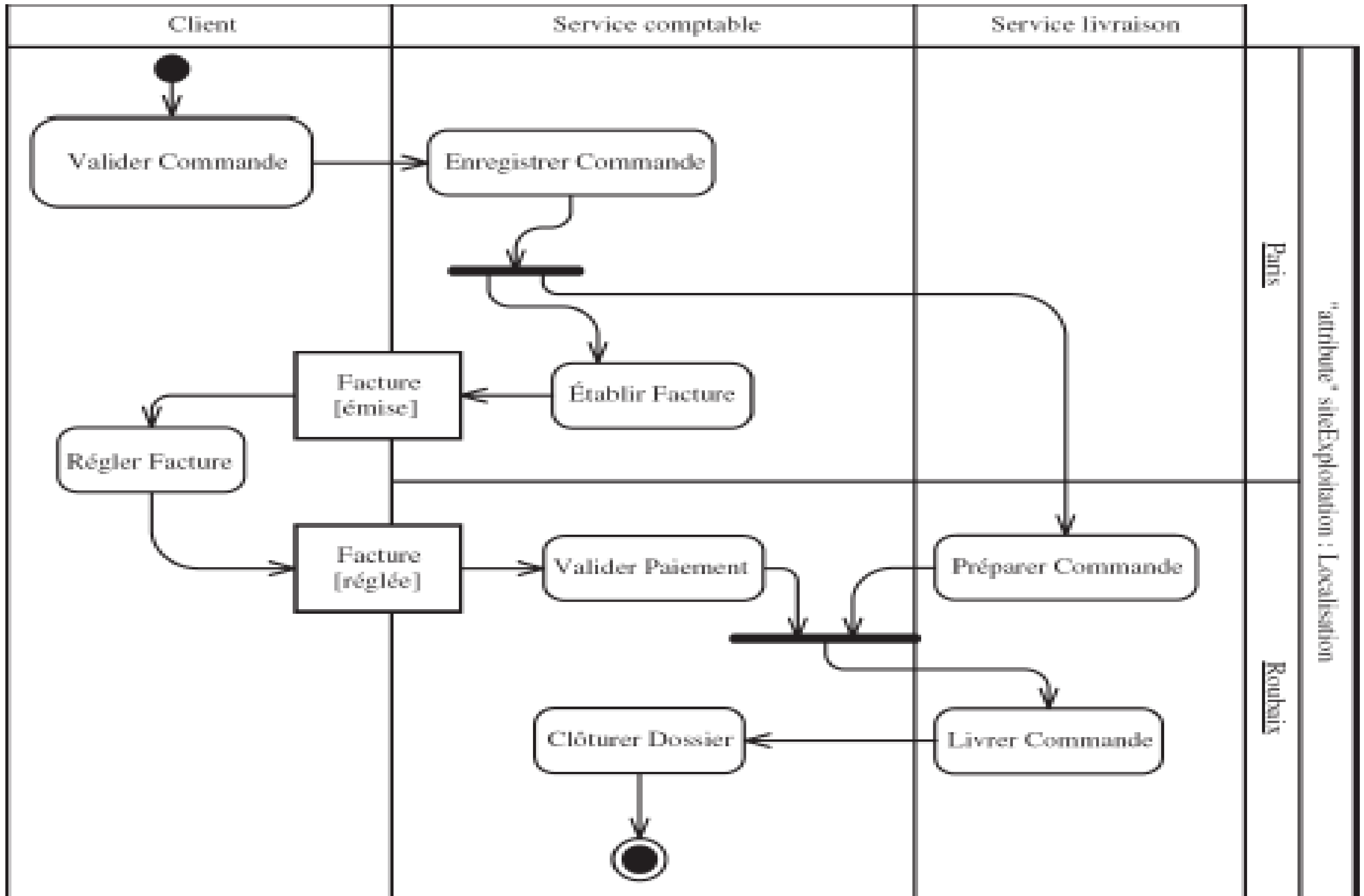


Activity diagram (Partitions –Example 1)



Activity diagram (Partitions –Example 2)

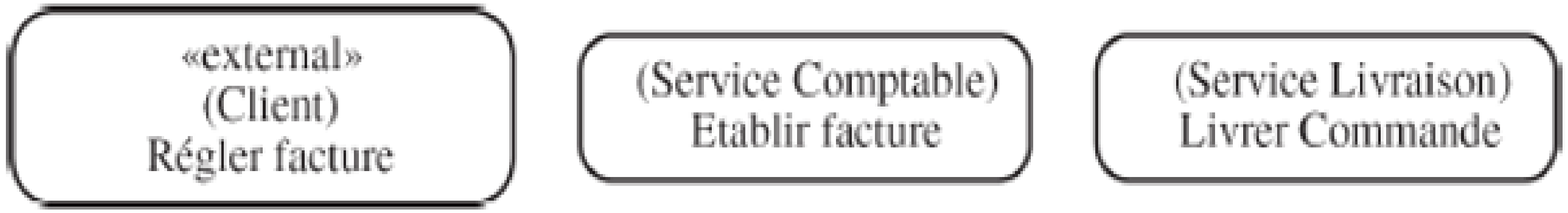
- Multidimensional partitions



Activity Diagram (Explicit Partition)

✓This notation is less graphically cumbersome.

✓However, it less highlights the **membership of activities groups** in the same container. .

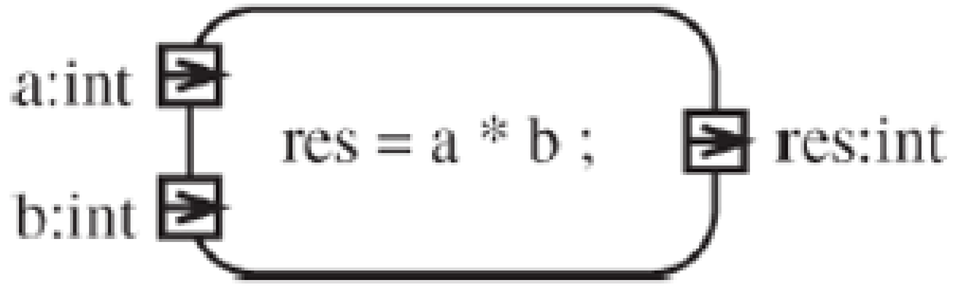


Activity diagram (the arguments and return values of the operation)

- ✓The activity diagrams presented so far clearly show the behavior of the control flow.
- ✓However, the data stream does not appear.
- ✓If an activity is well suited to describing an operation of a classifier, it needs a way to specify the arguments and return values of the operation. This is the role of:
  - ✓Pins,
  - ✓nodes,
  - ✓Flows of associated objects.

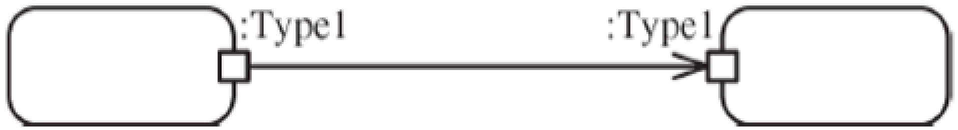
Activity Diagram (Pin)

- A **Pin** represents a **connection point** for an action.
- The **action** can only **begin if a value is assigned** to each of its input pins.
- The **values** are passed in by copy.
- When the activity ends, a value must be assigned to **each of the output pins.**

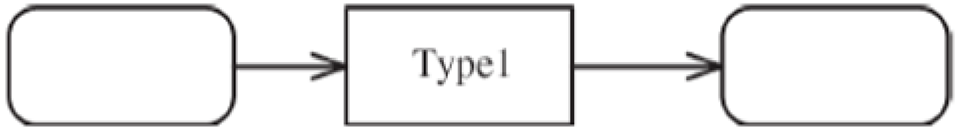


Activity Diagram (Data Flow)

- A **flow of objects** is used to **pass data** from one activity to another.
- In fact, an **arc whose origin and destination is a pin** corresponds to a **data flow**.

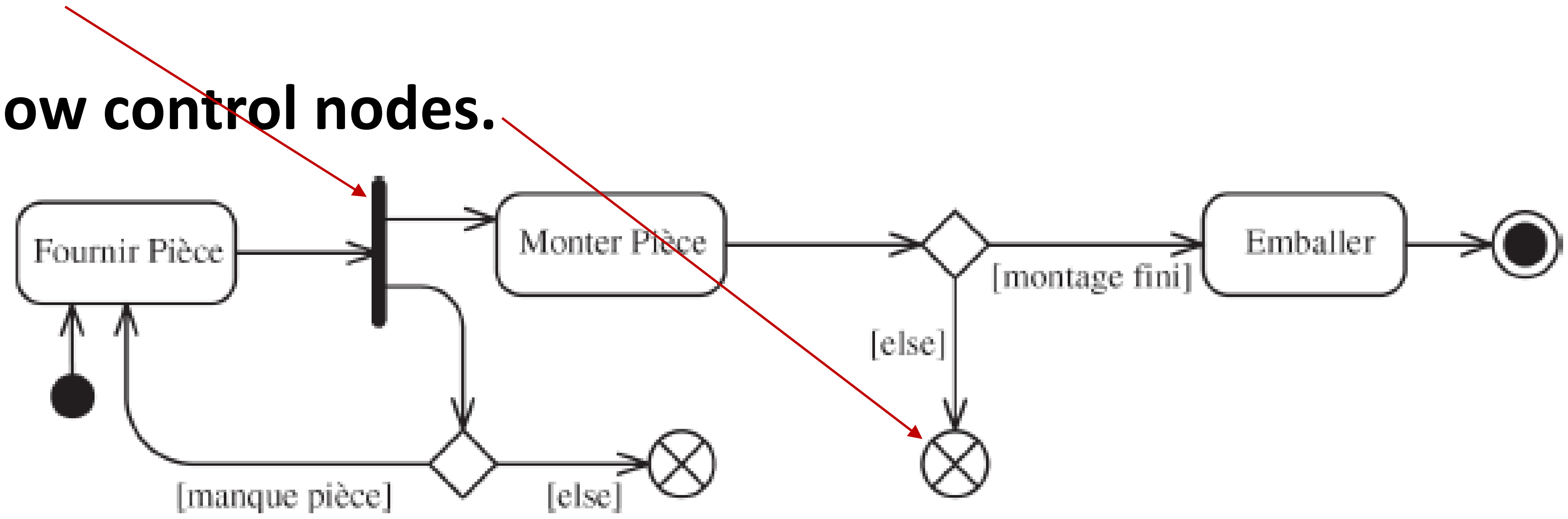


- A **node of objects** makes it possible to **better highlight the data**.
- It is a **typed container** that allows the transit of data.



Activity Diagram (Concurrent Activity)

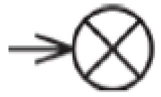
- ✓Activity diagrams are basically used to represent **sequential activities**.
- ✓Nevertheless, we can represent <u>**concurrent activities**</u> with:
- ✓**Synchronization bars,**
- ✓**The final flow control nodes.**

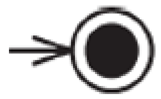


Activity Diagram (Synchronization Bar)

- ✓Several transitions can have a synchronization bar as their source or as their target.
- ✓When the synchronization bar has several output transitions, we speak of a fork type transition which corresponds to a duplication of the control flow into several independent flows.
- ✓When the synchronization bar has several input transitions, we speak of a join type transition which corresponds to a rendezvous between control flows.
- ✓For convenience, it is possible to merge join and fork type synchronization bars.
- ✓There are then several incoming and outgoing transitions on the same bar.

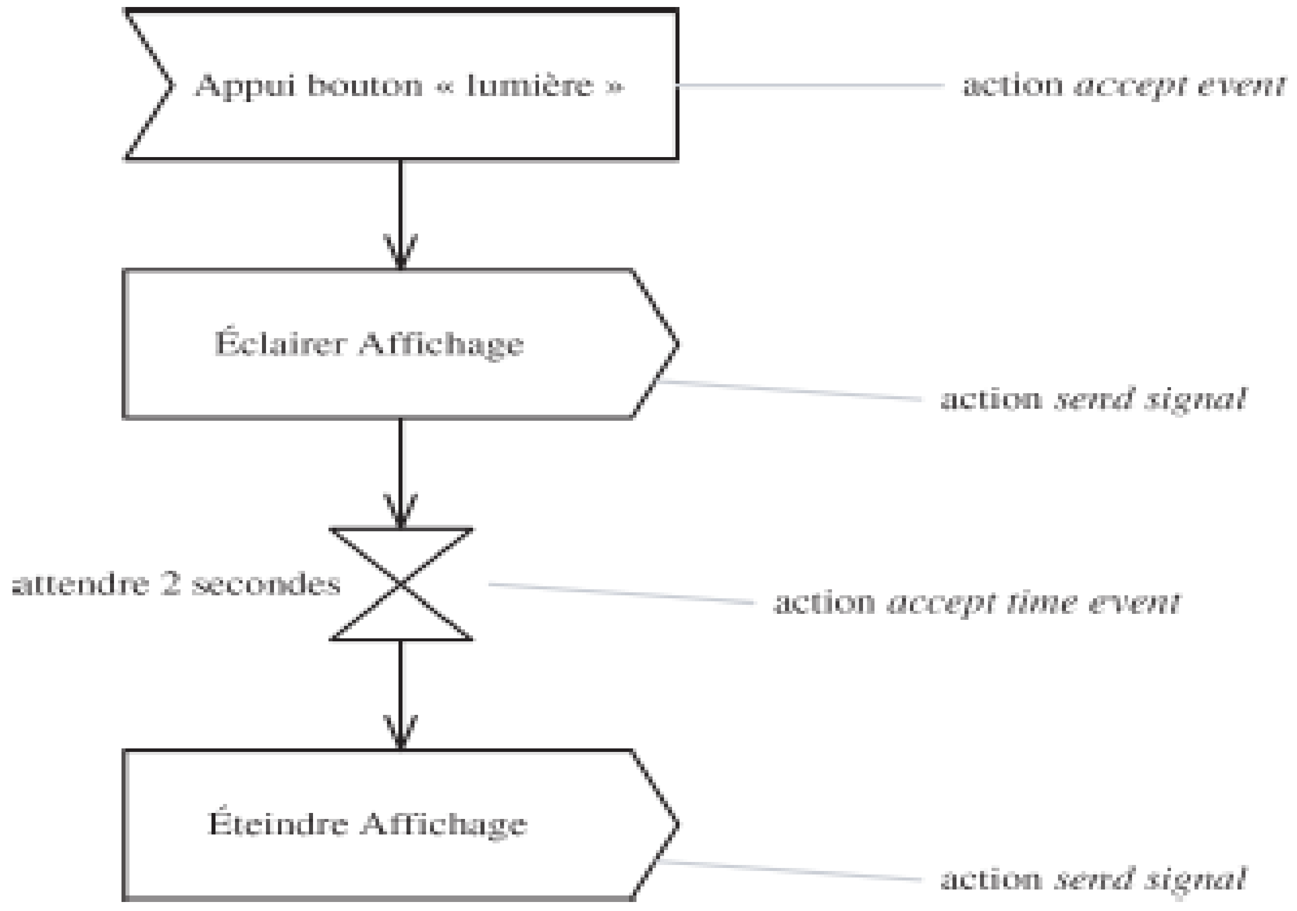
Activity Diagram (final flow)

- A control flow that reaches a control node of type "final flow" is destroyed.
- Other control flows are not affected.
- A **final flow** is represented by: 

- This type of node is **less** "**strong**" than a **final control node**.
- In this case (**final control node)**, **all other activity control flows** are **interrupted** and **destroyed**.
- A **final control node** is represented by: 

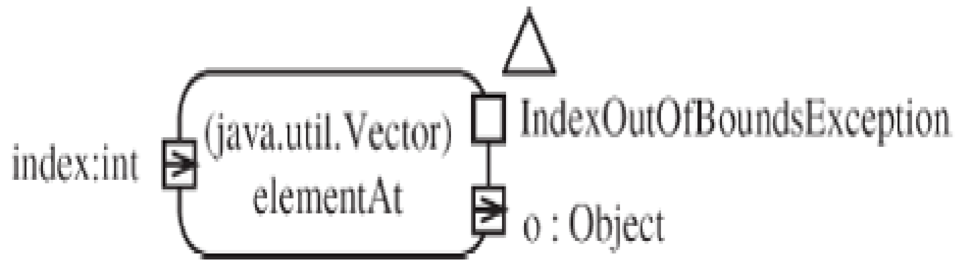
Activity Diagram (Communication Actions)

- **Communication actions** manage parameter passing and return, synchronous and asynchronous operation call mechanisms.
- **Communication actions** can be used as activities in activity diagrams.



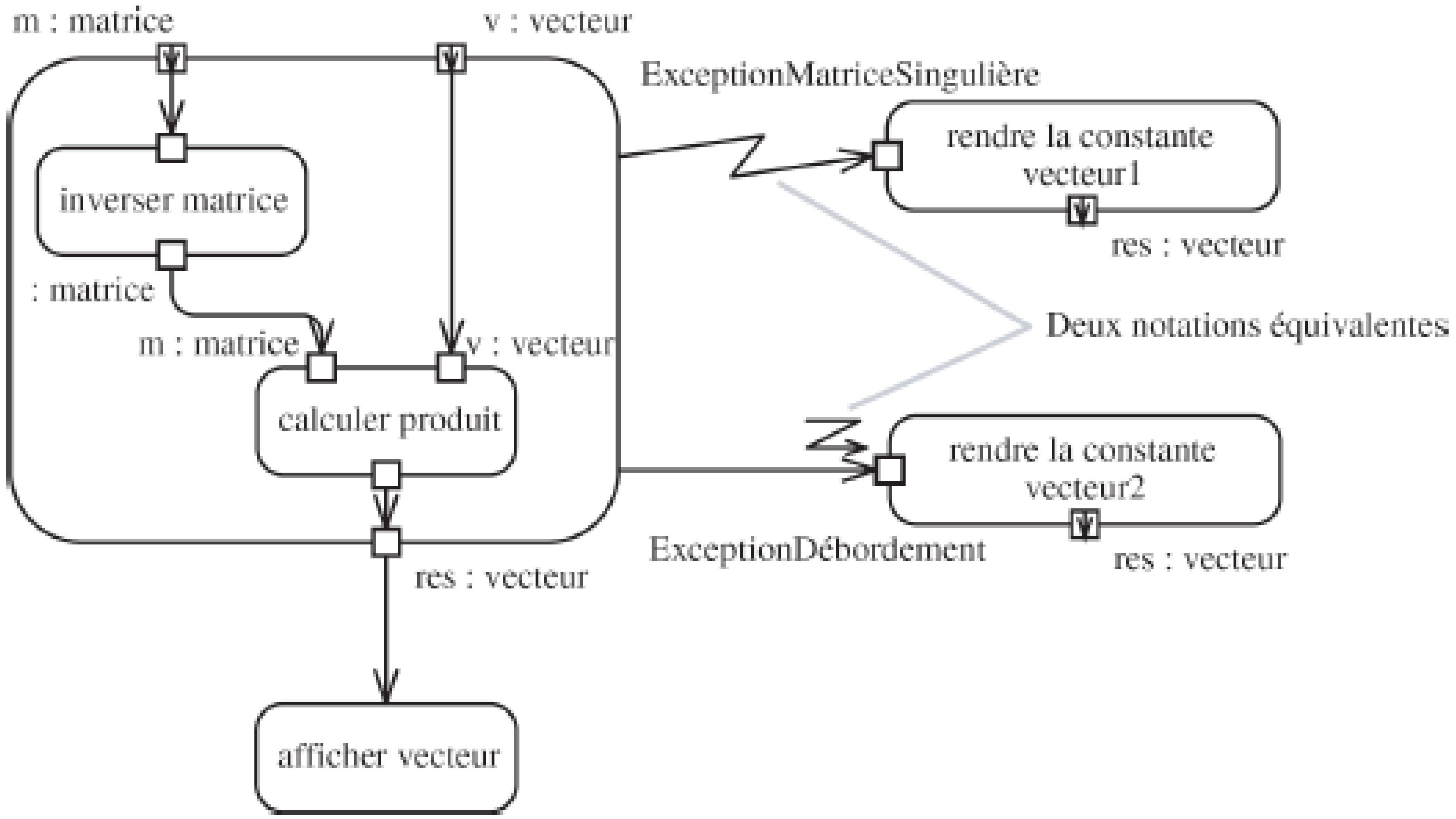
Activity Diagram (Exception)

- ✓**Exceptions** allow processing to be interrupted when a situation that derives from normal processing occurs. They provide cleaner management of errors that may occur during processing.
- ✓We use **exception pins** (with a triangle) one to handle exception throwing and exception catching.



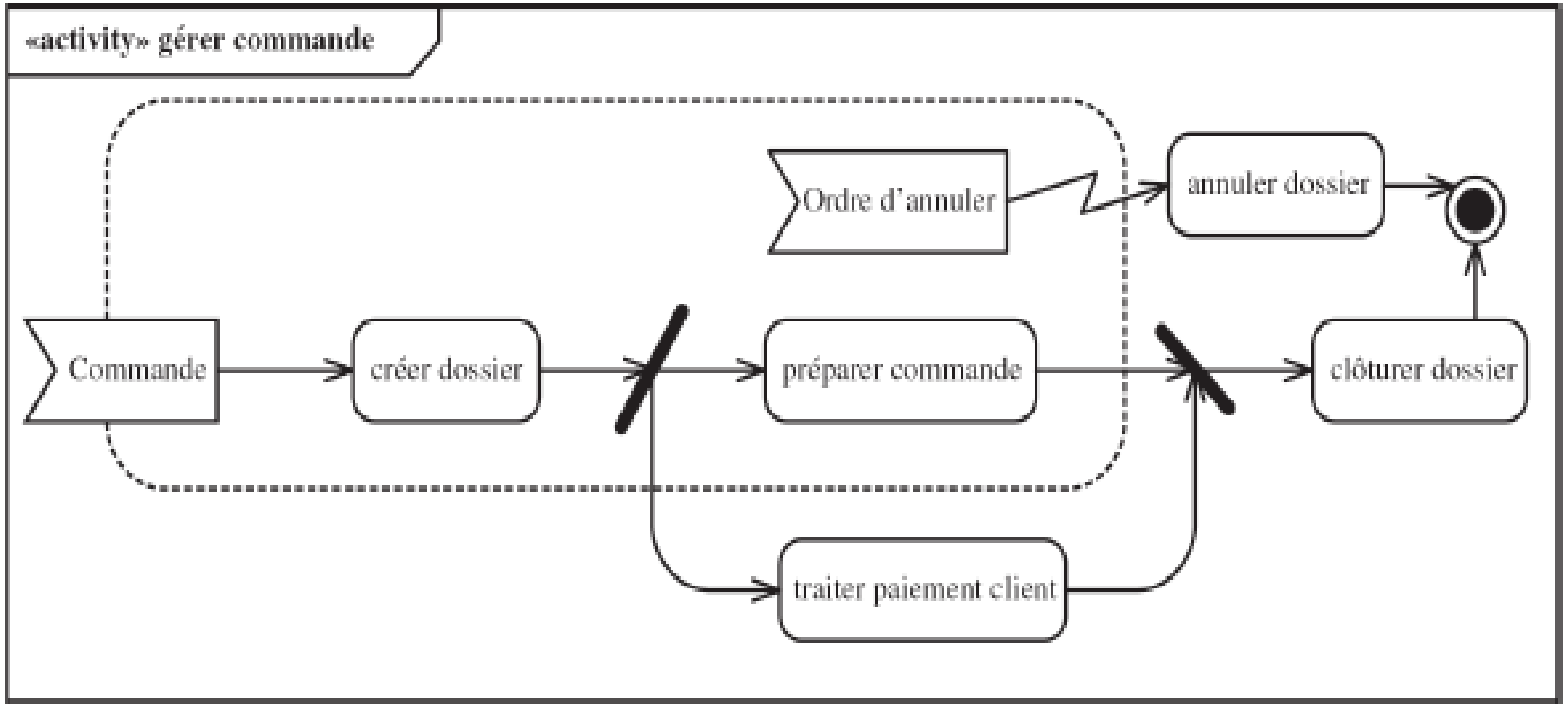
Activity diagram (Exception processing)

- A data flow corresponding to an exception is materialized by a "zigzag" arrow.



Activity Diagram (Interruptible Region)

- ✓An **interruptible region** is represented by a **dotted rounded frame**.
- ✓If the **interrupt event occurs**, all activities in progress in the interruptible region are **stopped** and the **control flow follows the zigzag arrow** that leaves the region.


«activity» gérer commande
Ordre d'annuler
annuler dossier
Commande
créer dossier
préparer commande
clôturer dossier
traiter paiement client