

Modélisation des Systèmes Informatiques.

C. Choppy
Informatique S6

UML - OCL - Problem Frames -

↳ Unified Modeling Language.

Methodologie for object-oriented languages (1989-94: more than 50).

- Grady Boock → Boock
- Jim Rumbaugh → OMT
- Ivar Jacobs → COSE
- some other

} UML 0.9 (1996).

1997: UML 1.0, adoption par l'OMG (Object Management Group)

2003: UML 1.5, puis UML 2.0

2015: UML 2.5.

differences
↔

UML 1.5 (2003)

- 2 use case diagram
- 4 class diagram
- X object diagram
- 21 statechar diagram
- 34 activity diagram
- 10 sequence diagram
- 18 collaboration diagram
- 32 component diagram
- 35 deployment diagram
- X
- X
- X
- and other modifications
(like acts in diagrams).

UML 2.0 (2003)

- //
- //
- //
- state machine diagram
- //
- //
- communication diagram
- //
- //
- X composite structure diagram
- 19 interaction overview diagram
- 20 timing diagram
- 33 package diagram
- 36 OCL

In UML 2.0, these diagrams are classified into:

- Structured diagrams: class, component, deployment, composite structure, package.
- Behavior diagrams: use case, state machine, activity, interaction (sequence, collaboration, interaction overview, timing).

Use Case.

A use case is a typical interaction between a user and a computer system.

A use case captures some user-visible functions.

A use case achieves a discrete goal for the user.

A use case may be small or large.

user goal vs system interactions:

user goal:

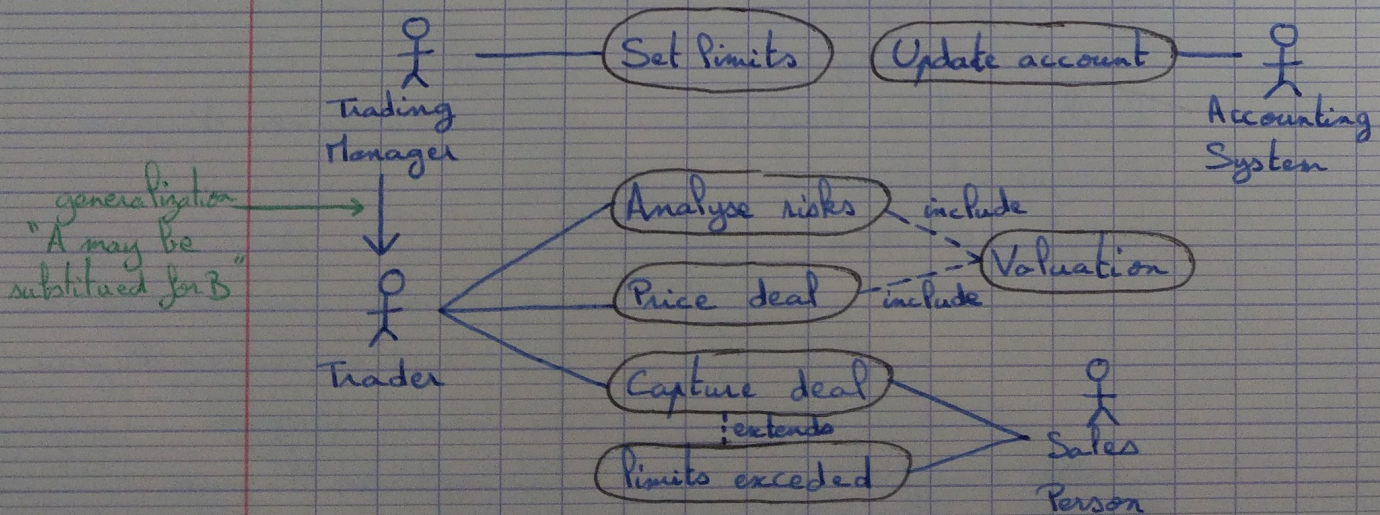
"ensure consistent
formatting for
a document"

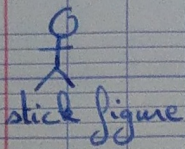
Word Processing

system interactions:

- define a style
- change a style
- move a style from one document to another

example of use case:





stick figure

This is an actor:

- represent a role
- is not necessary a person
- a given person may play several roles

primaire / secondaire

* un acteur
primaire est celui pour qui la fonctionnalité est produite.
* l'acteur
secondaire est utilisé par le système pour réaliser la fonctionnalité.

Stylistic variations among use case diagrams:

- all interactions with remote systems.
- only when the other system enters the contact.
- only system actors who need use case.
- system actor is a wrong focus.

Use case are all about externally required functionalities.

Thinks about security policy.

Some use cases do not have clear links to specific actors.

ex: "Send the bill?" Billing Department: gets value from the use case.

A good source for use case is external events.

Name the
Use Case

use case: it is a verb phrase.

EXTENDS relationship: one use case is similar to another, but does a bit more.

- Look for all that could go wrong.
- plot all variations as extensions.

INCLUDE relationship: a chunk of behavior is similar across more than one use case and you do not want to copy its description.

Both extends and include imply factoring out common behavior from several use case.

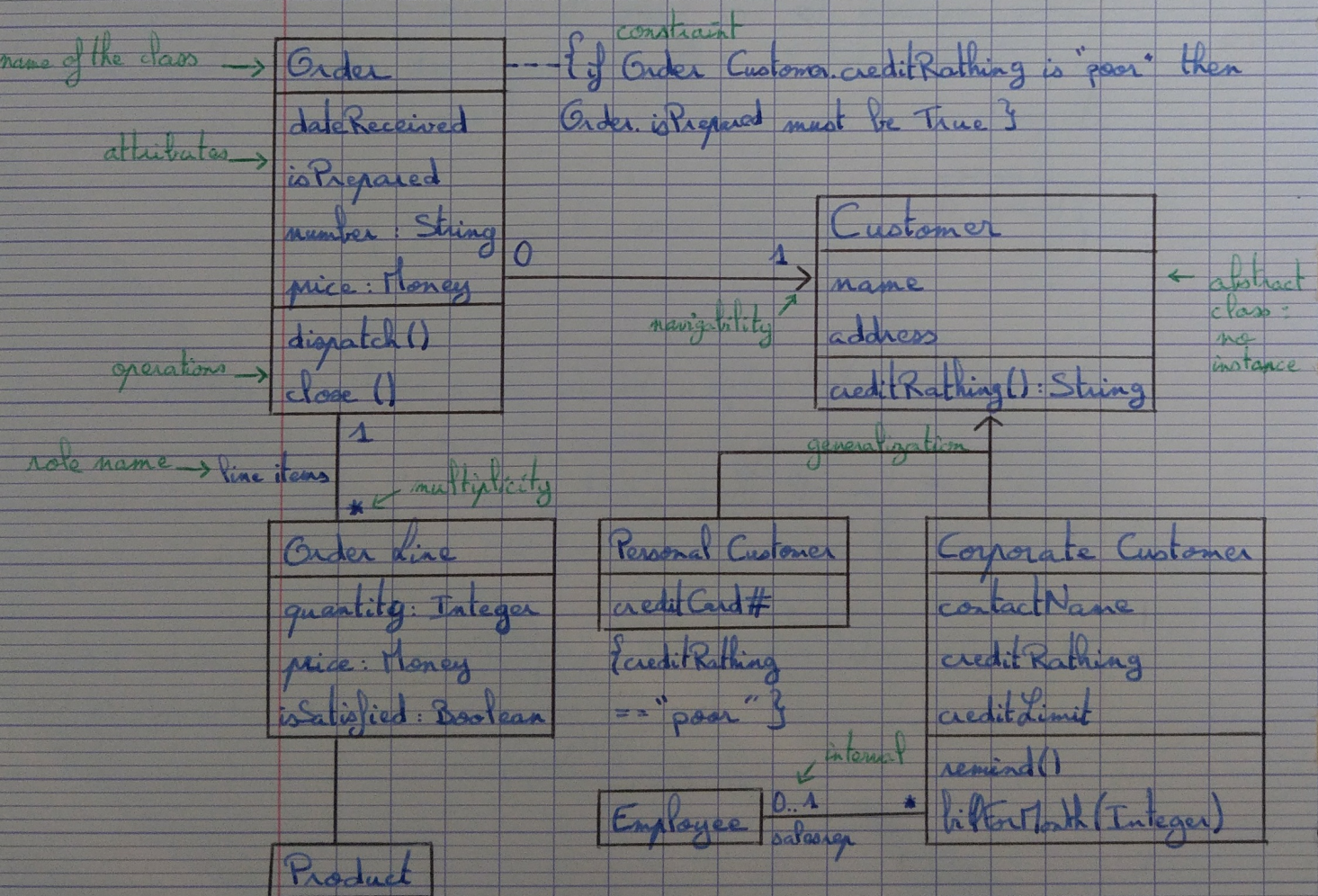
- use extends when you are describing a variation on a normal behavior.
- use include when you want to avoid repetition over several use cases.

Class Diagram

A class diagram describes the type of objects in the system and the various kinds of relationships among them:

- association: "has a" ex: a customer may rent a video.
- subtypes/generalization: "is a" a nurse is a kind of person.

example of class diagram:



* Attributes (similar to associations):

optional / compulsory

visibility name : type = default

attributes are always single values.

Visibility:

+ public

protected

- private

* Operations:

visibility name (parameters-list) : return-ty-expression { property string }

ex: - getting methods

- setting methods

* Constraints:

{ constraint }

can be expressed in any language (natural language, or any preferred)

Like OCL: Object Constraint Language (adopted by OMG).

(constraints)

* Design by contract:

- pre-condition

ex: { input >= 0 }

- post-condition

ex: { input >= 0 }

- invariant

ex: { balance == sum(entries.amount) }

they are properties expressed by assertions.

(constraints)

* Assertions requires some special care: they can play a unique role in subclassing.

One of the dangers of polymorphism is that you could redefine a subclass operation to be inconsistent (incoherent) with the superclass operation.

The invariant and post conditions of a class must apply to all subclasses. The subclasses can choose to strengthen (reinforce) these assertions but they cannot weaken (afflict) them.

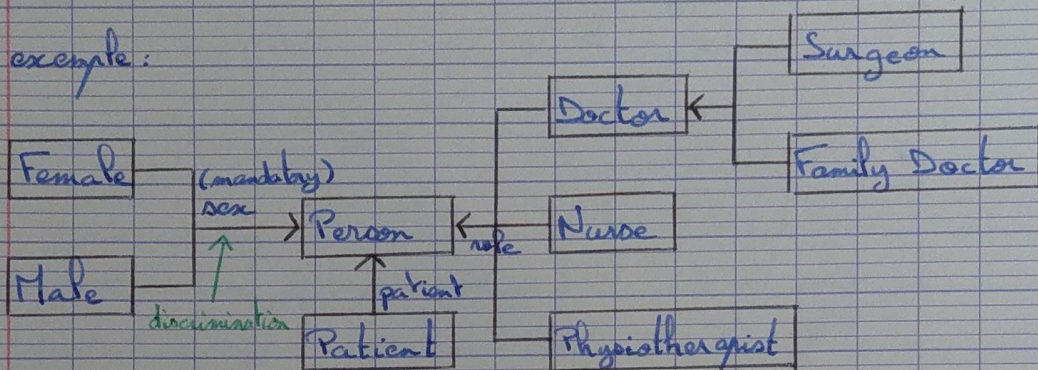
The pre-condition, on the other hand, cannot be strengthened but may be weakened.

* Stereotypes:

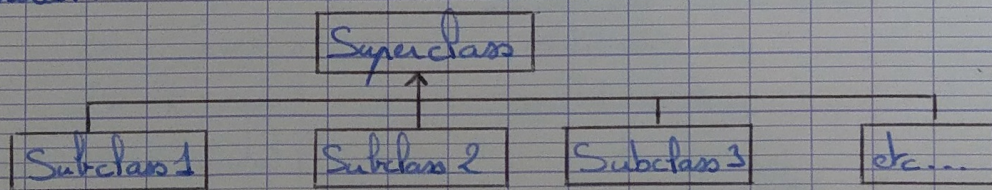
They are used to introduce a high-level classification of an object that gives an indication of the kind of object it is: « name of the stereotype »

Multiple and dynamic classification / generalization:

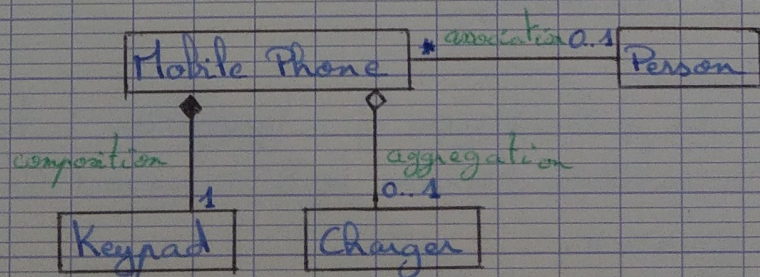
example:



general:



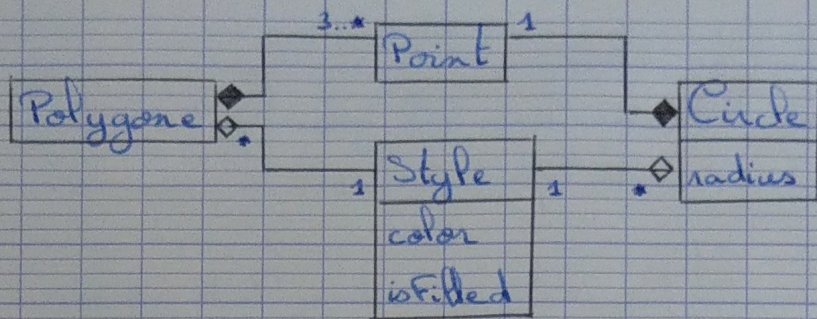
Association, aggregation and composition:



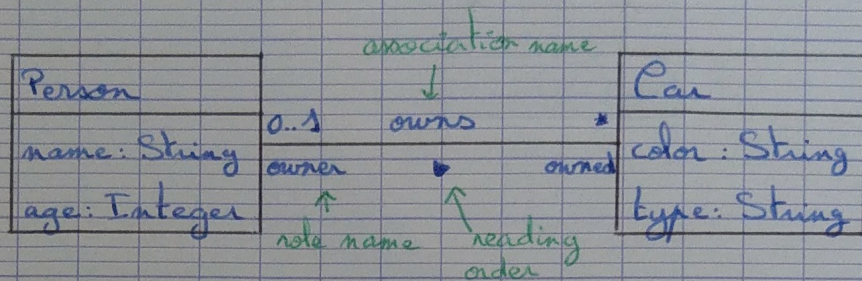
The keypad compose the mobile phone.

The charger is needed by the mobile phone.

zero or one person have a certain number of mobile phone.



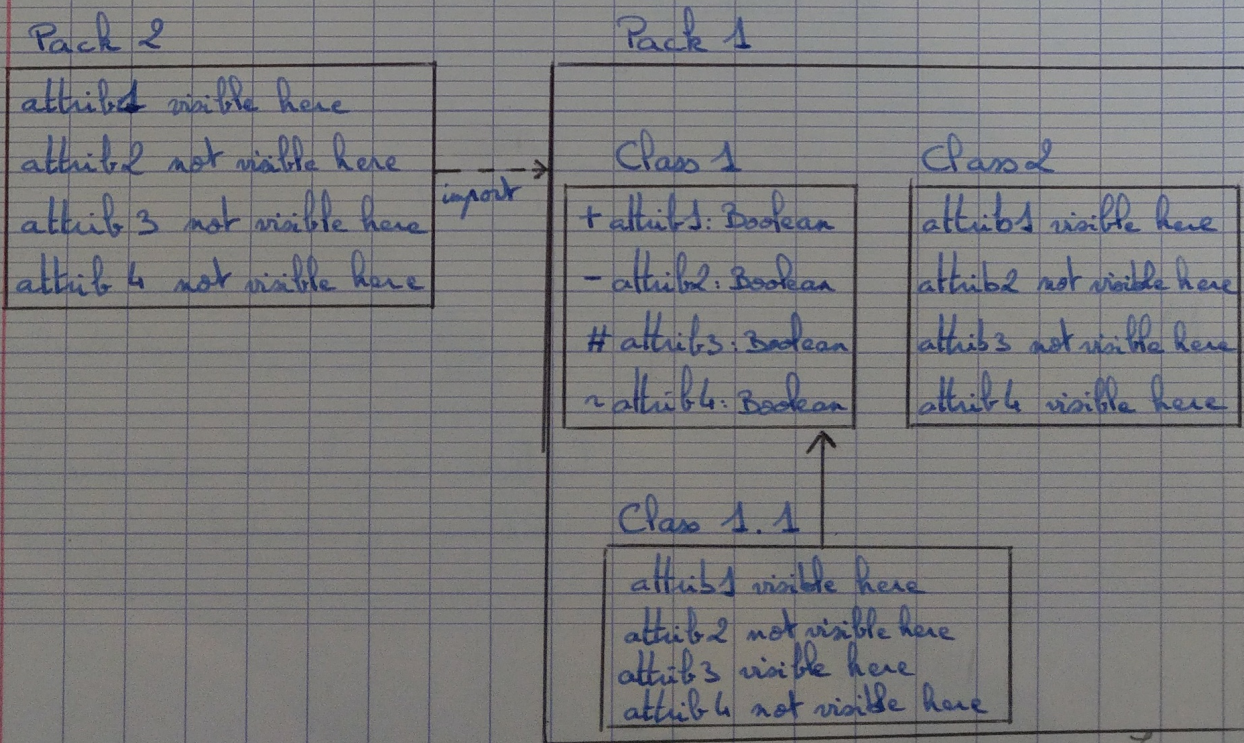
* More on associations:



Class features visibility:

- + public: visible from anywhere.
- private: not visible outside the objects instances of the class.
- # protected: visible only from the class and from its subclasses.
- ~ package: not visible outside the package containing the class.

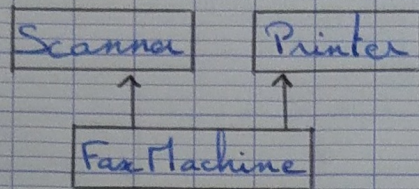
example:



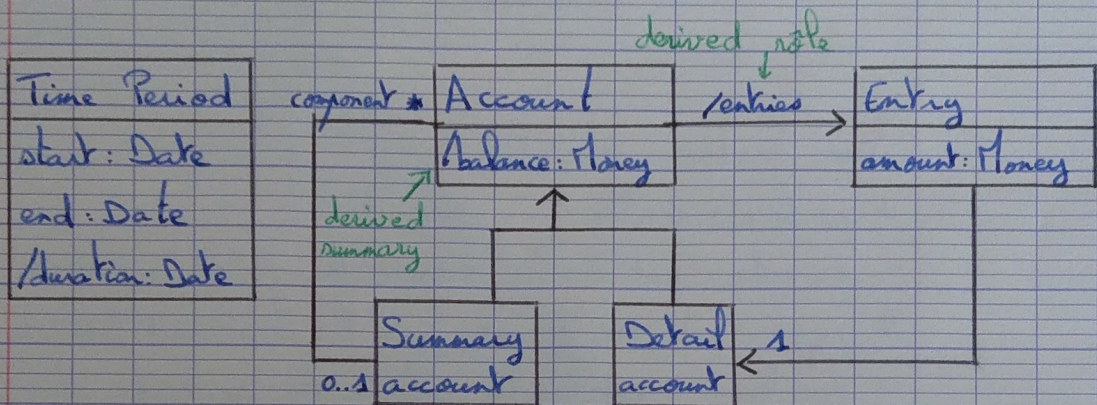
* Active class:

A state machine will generally be associated to it to describe its behaviors.

Generalization: multiple inheritance (handle with care).

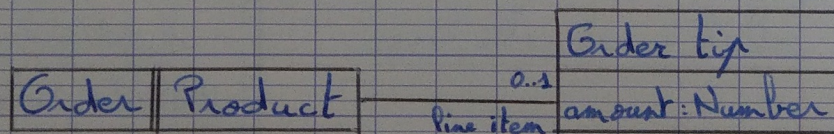


Derived associations and attributes.



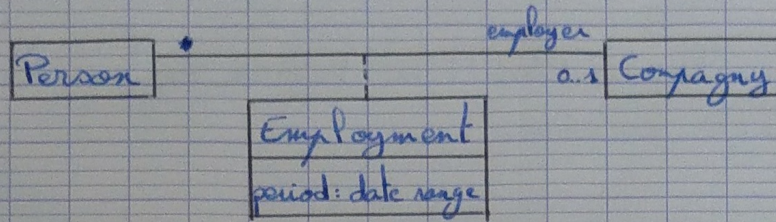
entries role is derived using component entries

Qualified association



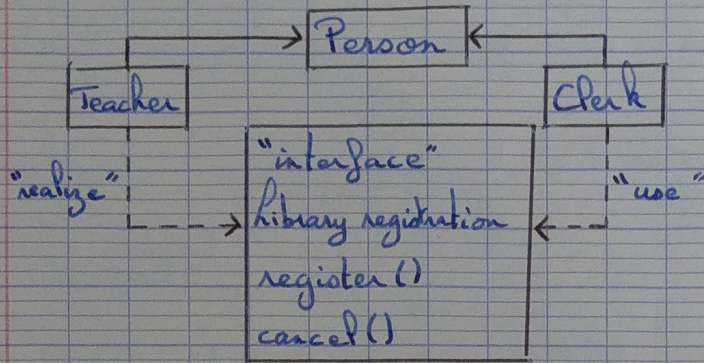
↑ there may be an order line for each instance of Product.

Association class

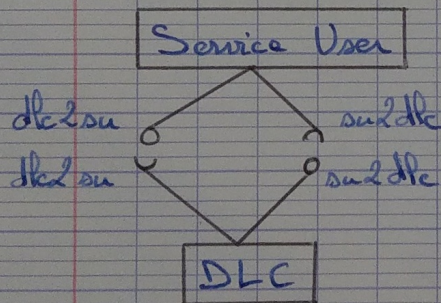


→ This association between a person and a company is an employment.

Interfaces.



Ports, interfaces and signals.



h required interface

o provided interface

"interface" su2dlc
"signal" d. EstabReg(DLCident)
"signal" d. EstabResp()
"signal" d. ReleaseReg(DLCident)
"signal" d. DataReg(DLCident, UData, Integer)

"interface" dlc2su
⋮

Interaction diagrams.

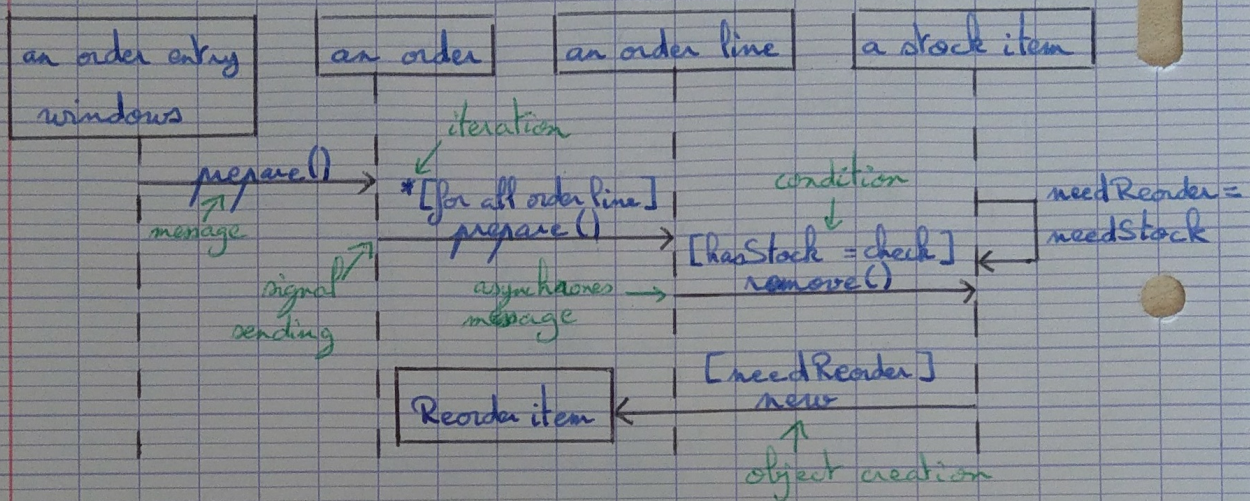
Typically, an interaction diagram captures the behavior of a single use case.

Interaction diagram types:

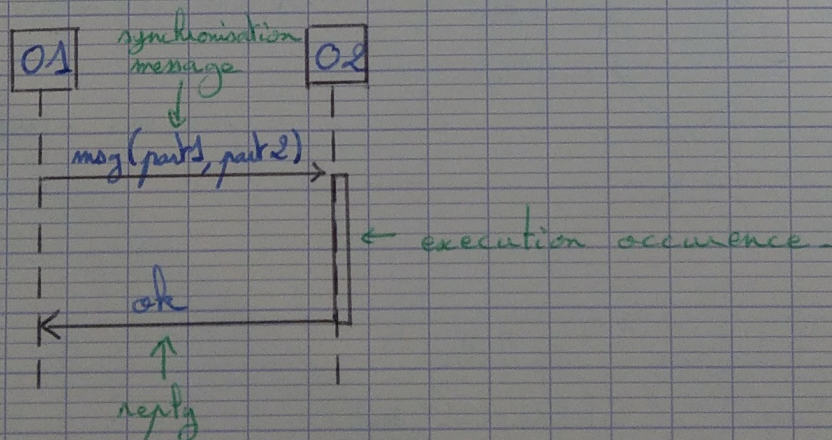
- sequence diagrams
- communication diagrams.

Sequence diagrams:

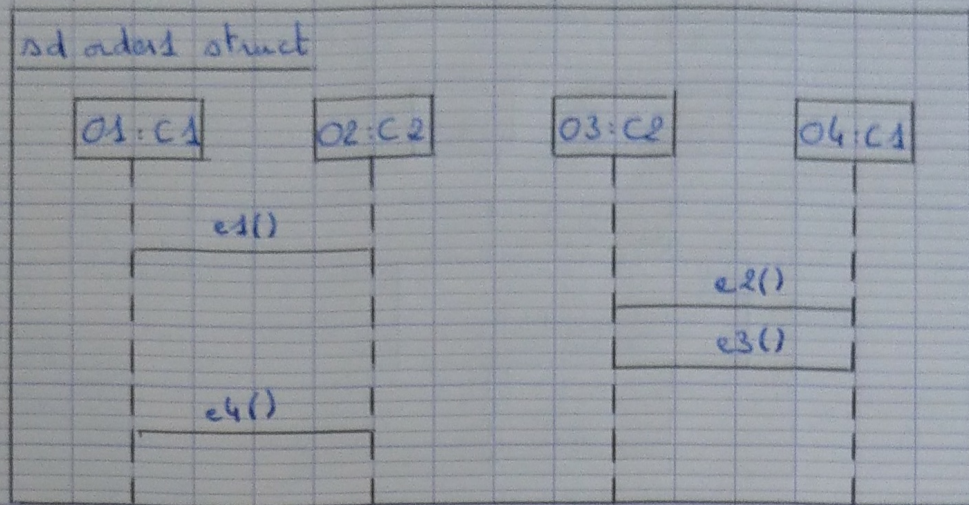
example:



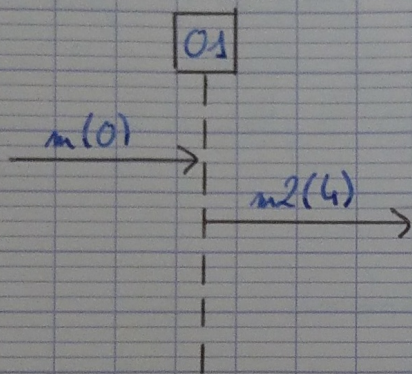
general:



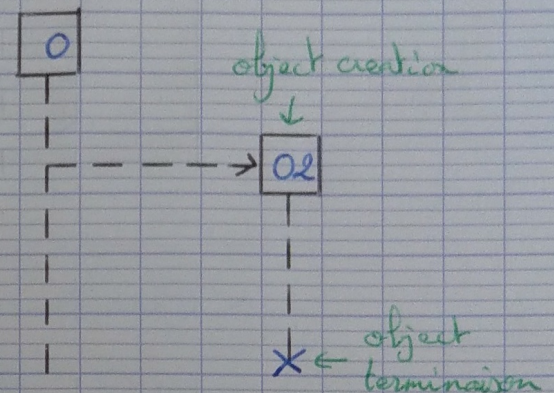
* Struct or weak events sequencing:



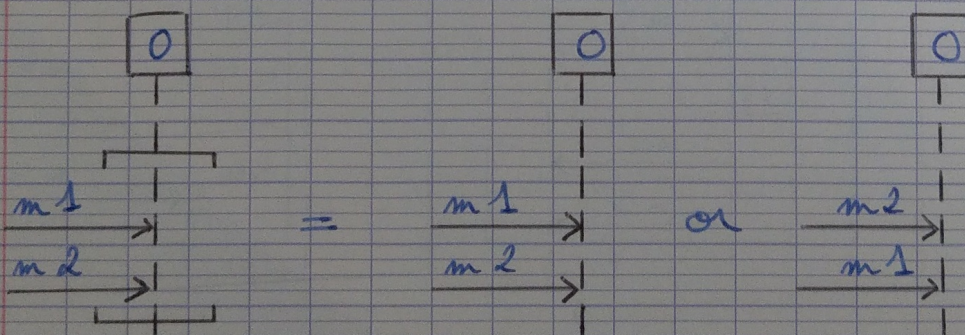
* Messages found or lost:



* Object creation and termination:

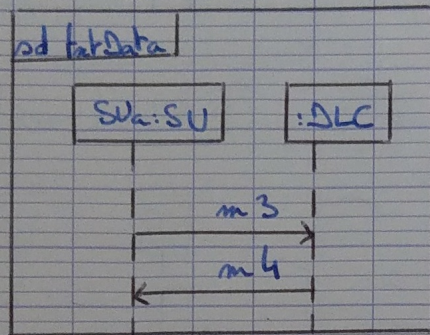
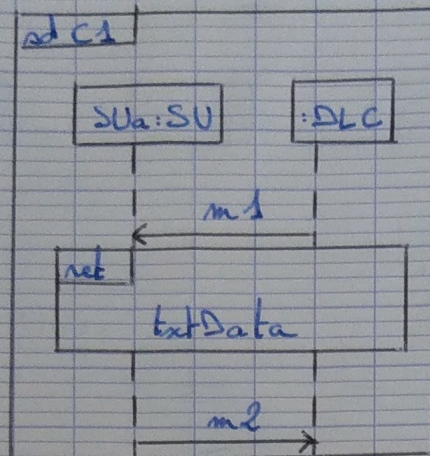


* Coregion:

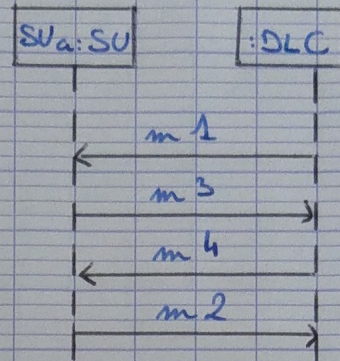


La coregion correspond à "l'ordre importe peu".

* Interaction occurrence :

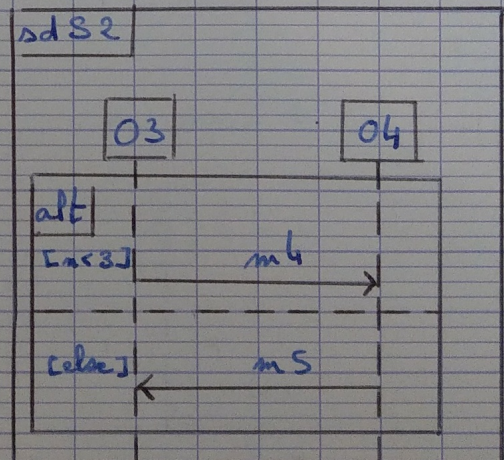
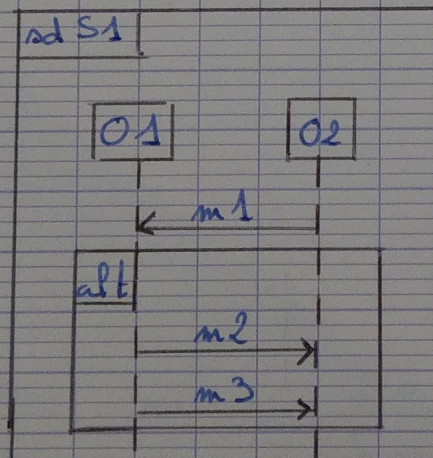


=



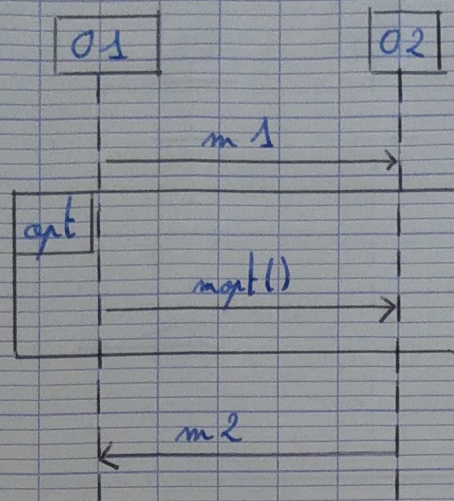
→ c'est un appel d'une boîte à l'autre, rend le schéma plus clair si beaucoup de boîtes.

* Operator alt (alternative):



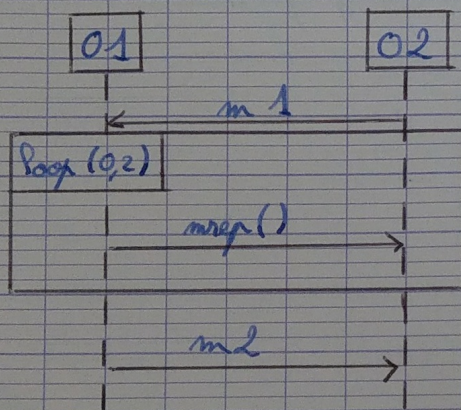
↳ j'ai le choix entre les différentes option, ou alors chaque option correspond à un cas particulier.

* Operator opt (optional):



→ je choisis si j'entre dans le opt ou non.

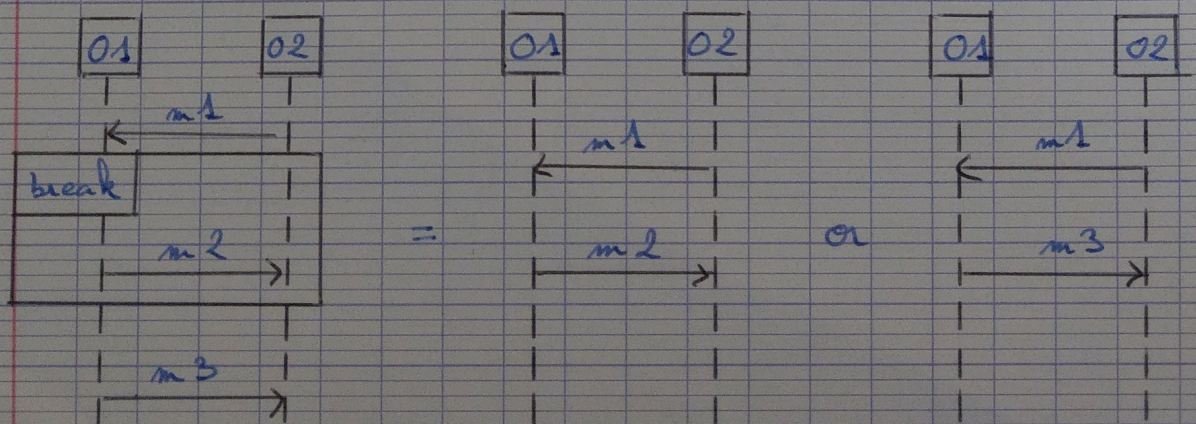
* Operator loop:



number of repetition:

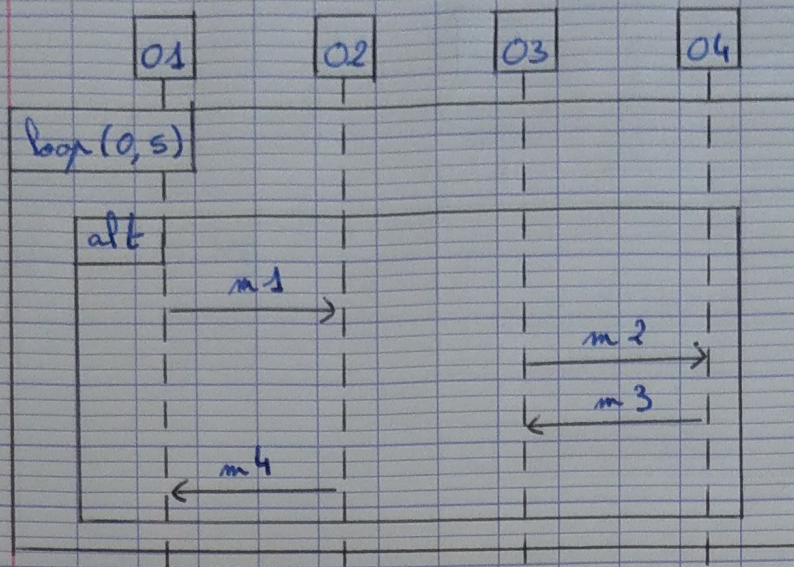
	minimum	maximum
loop(0,2)	0	2
loop(3)	3	3
loop(0,*)	0	∞
loop	0	∞

* Operator break:

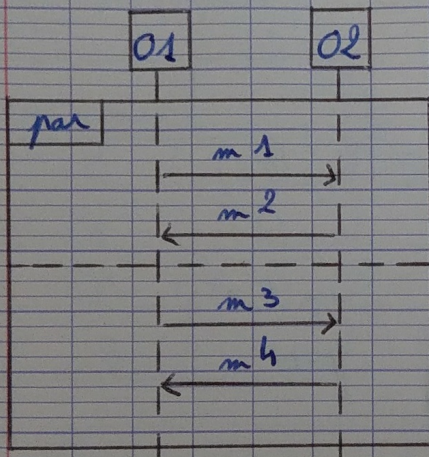


↳ soit j'évite le break, soit je rentre dedans et j'ai fini en bas du break.

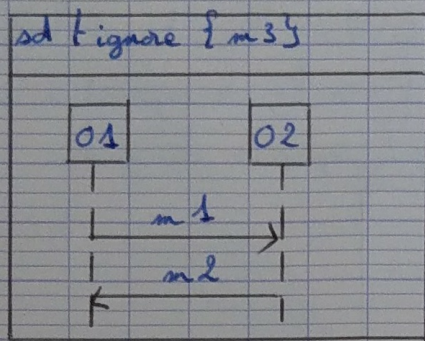
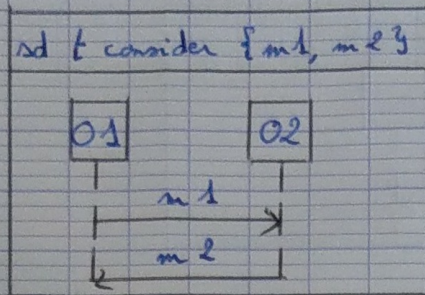
* Nested Operators:



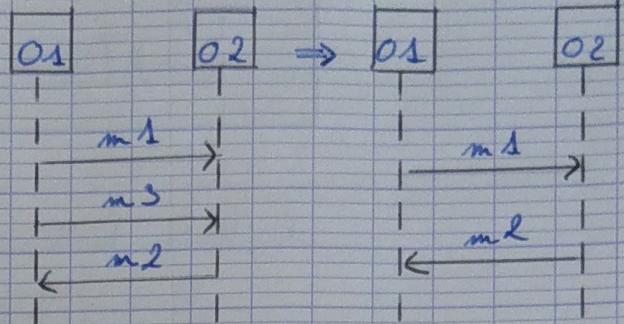
* Operator par: describe the parallel merge between the behavior of operands.



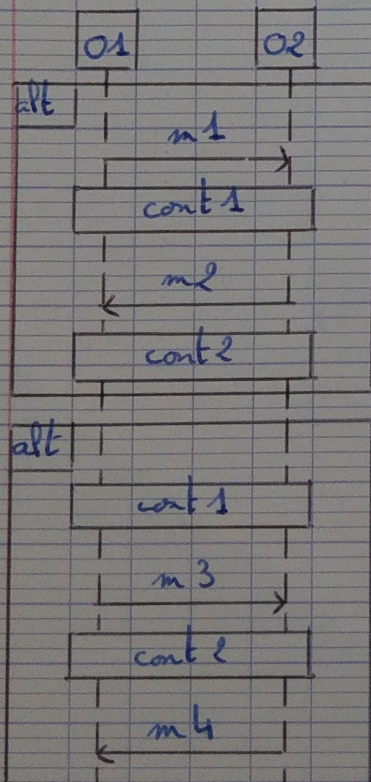
* Operator consider : any other messages are ignored / operator ignore



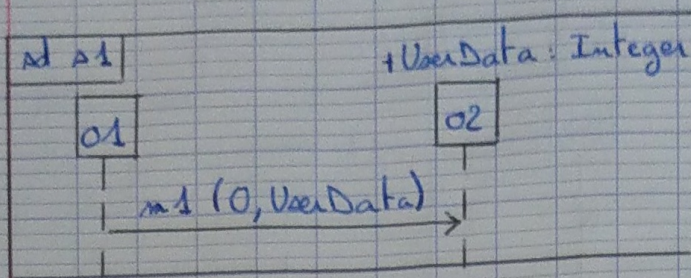
execution trace:



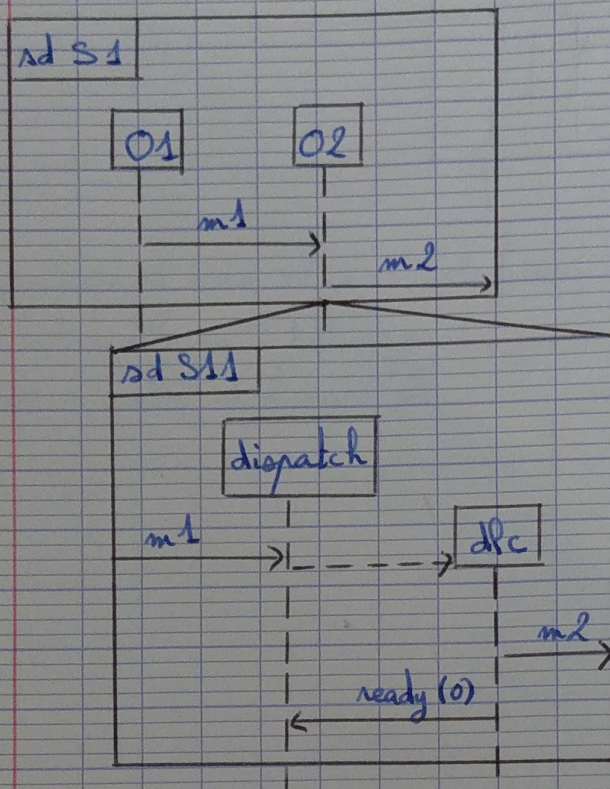
* Continuation symbols : can be used to memorize the branch executed in an alternative in order to select the branch to follow in the next alternative.



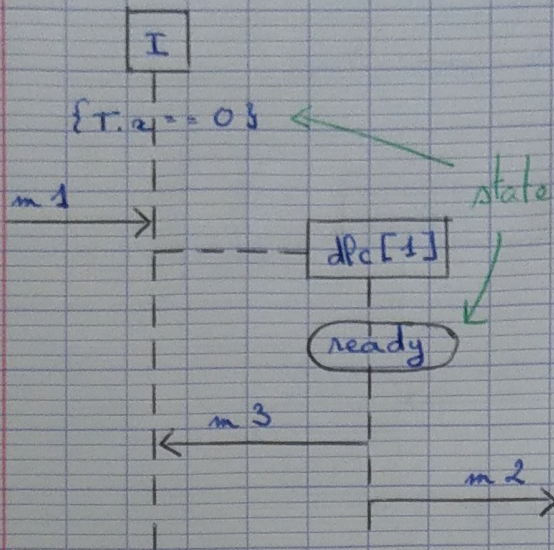
* local attitudes:



* Parts decomposition: to show the internal behavior of a lifeline, it can be decomposed in a kind of sub-sequence diagram.

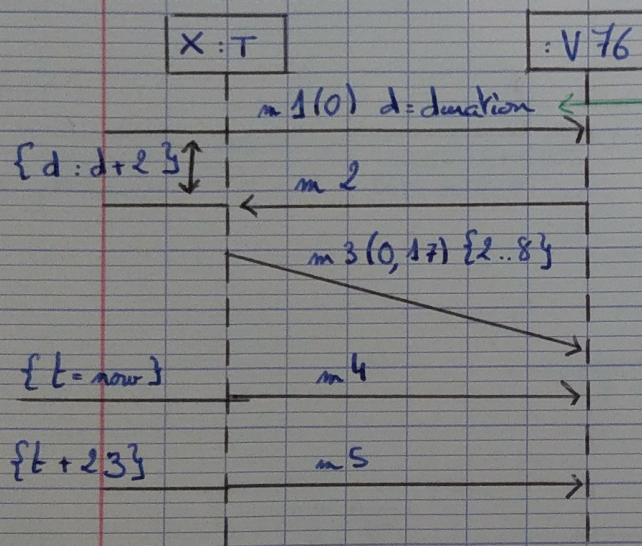


* State invariants: Derived values can be specified in state invariant.



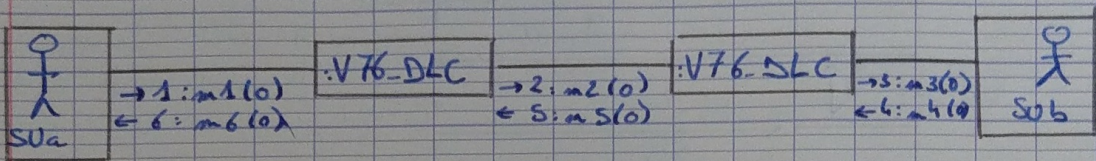
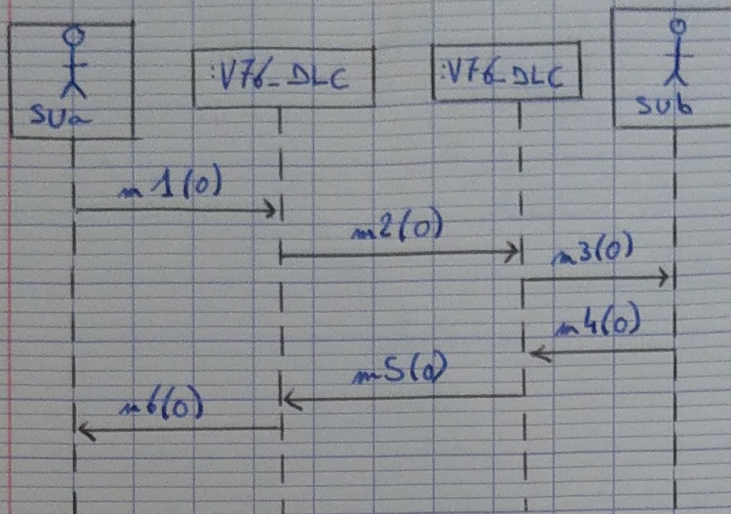
dpc: Data Link Control

* Time constraint:

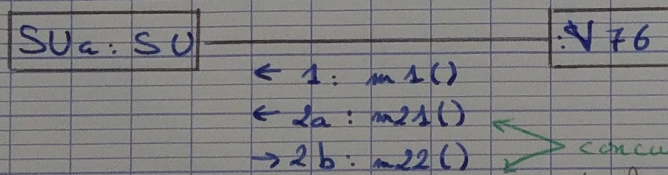


Communication diagrams: (collaboration diagram before UML 2.0).

example from sequence to communication:



* With concurrent messages:

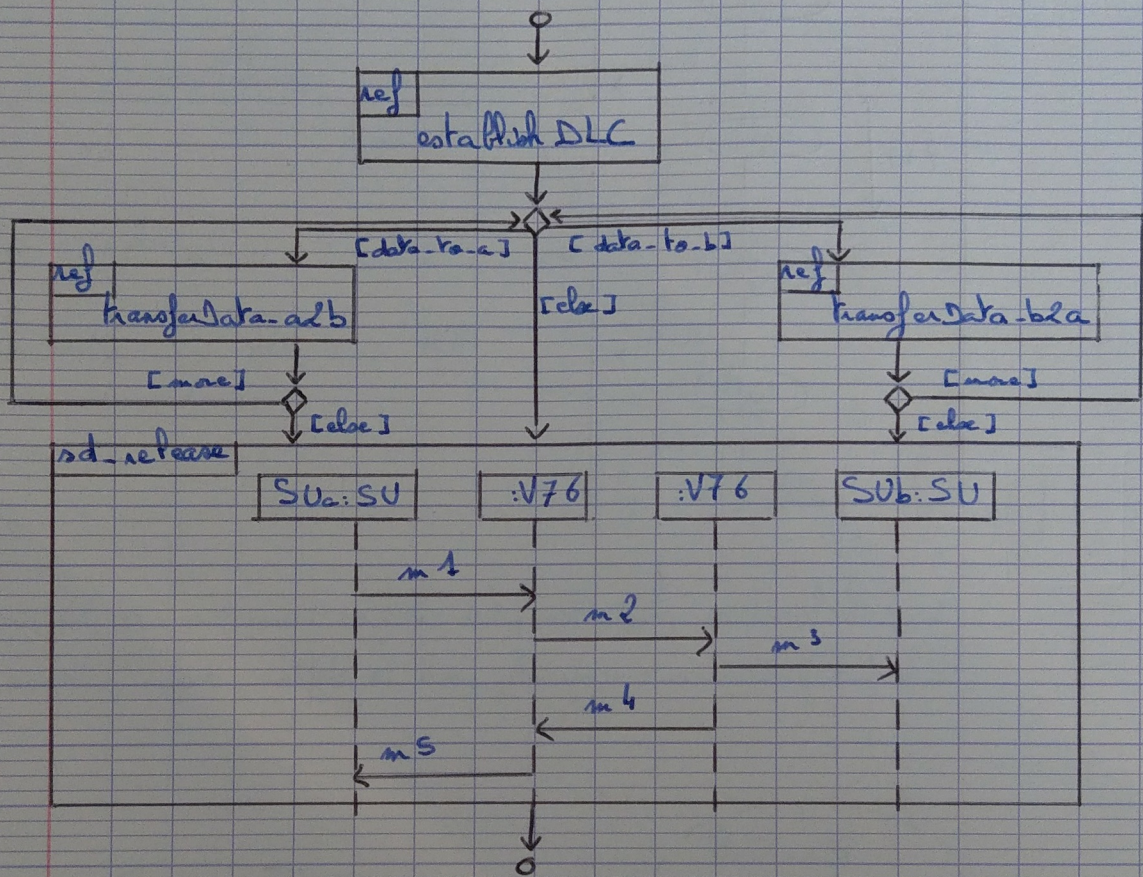


concurrent messages
↳ il n'y a pas d'ordre précis entre ces deux là.

Interaction Overview diagram.

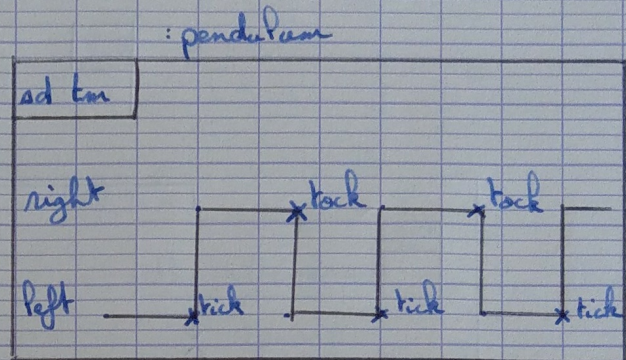
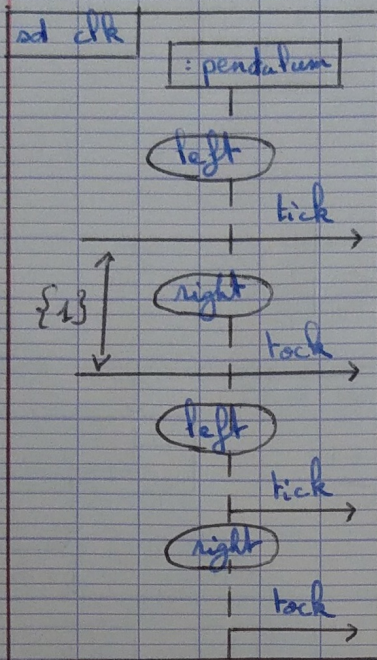
An interaction overview diagram provides a high-level synthetic view of interactions.

example:

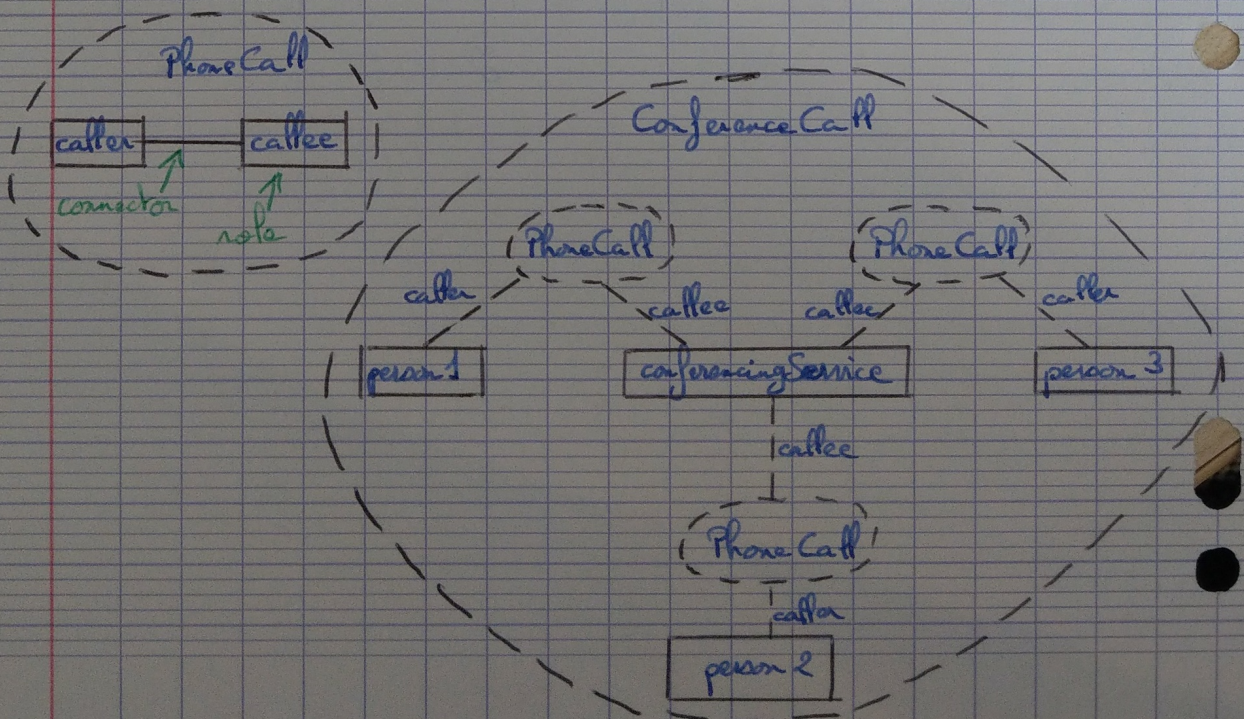


Timing diagram.

Similar to sequence diagram rotated 90 degrees left, show the state of an UML element in function of time.



A collaboration illustrates how entities communicate to perform a certain task. Collaborations are generally used to define design patterns (that can be re-used).



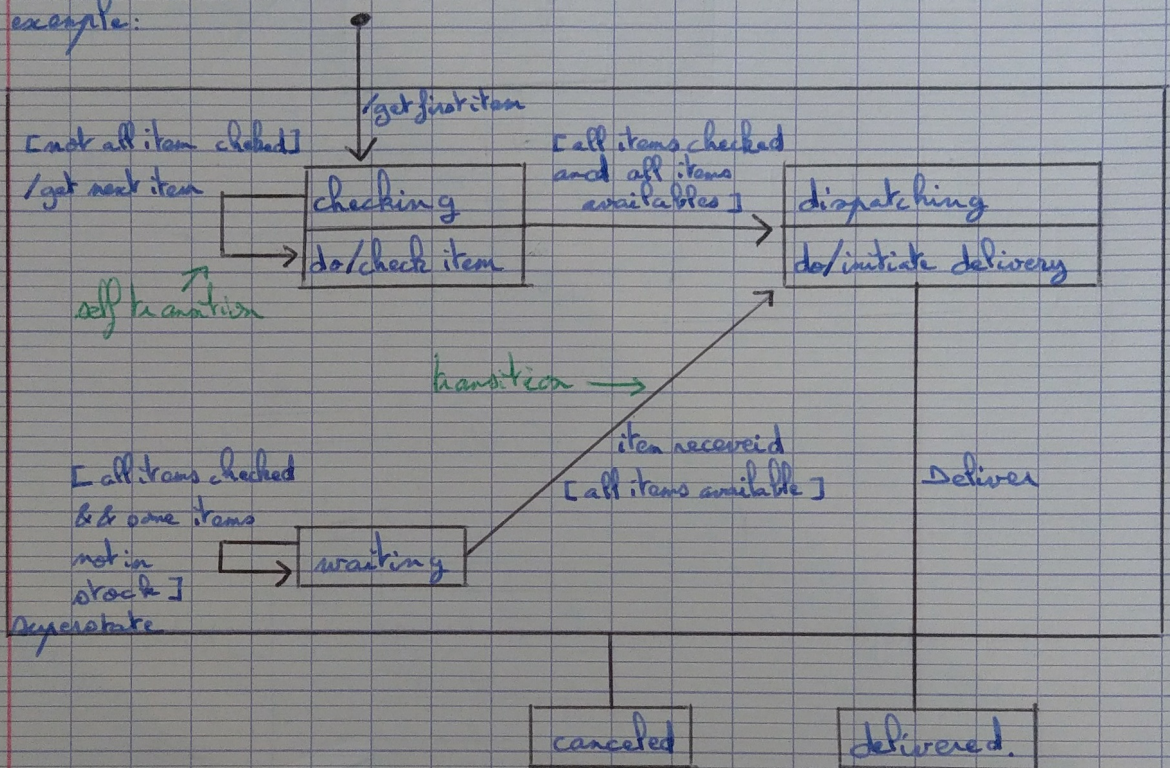
State Machine diagrams (statechart diagram before UML 2.0).

↳ David Harel

Focus: states and transition between states.

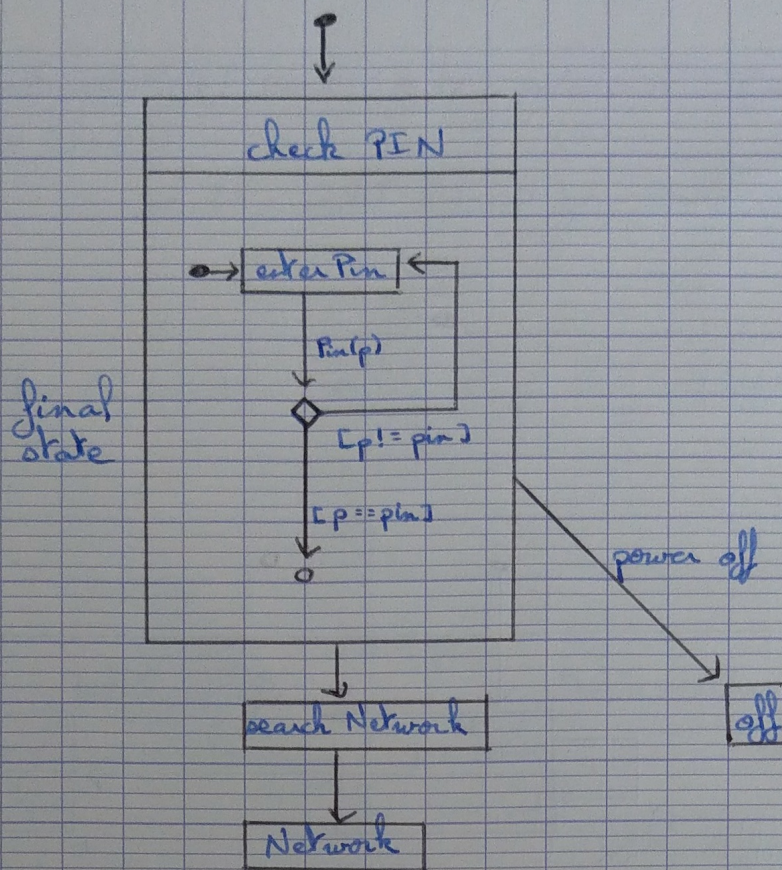
The syntax for a transition: Event [Guard] / Action.

example:

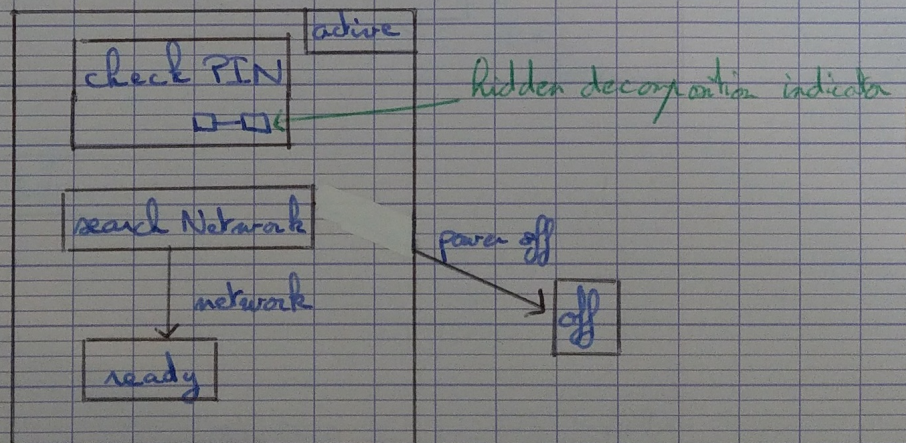


NB: action $\neq$ activity:

- action is immediate, not interrupted and associated with transition.
- activity take some time and performed while in a state.



OR

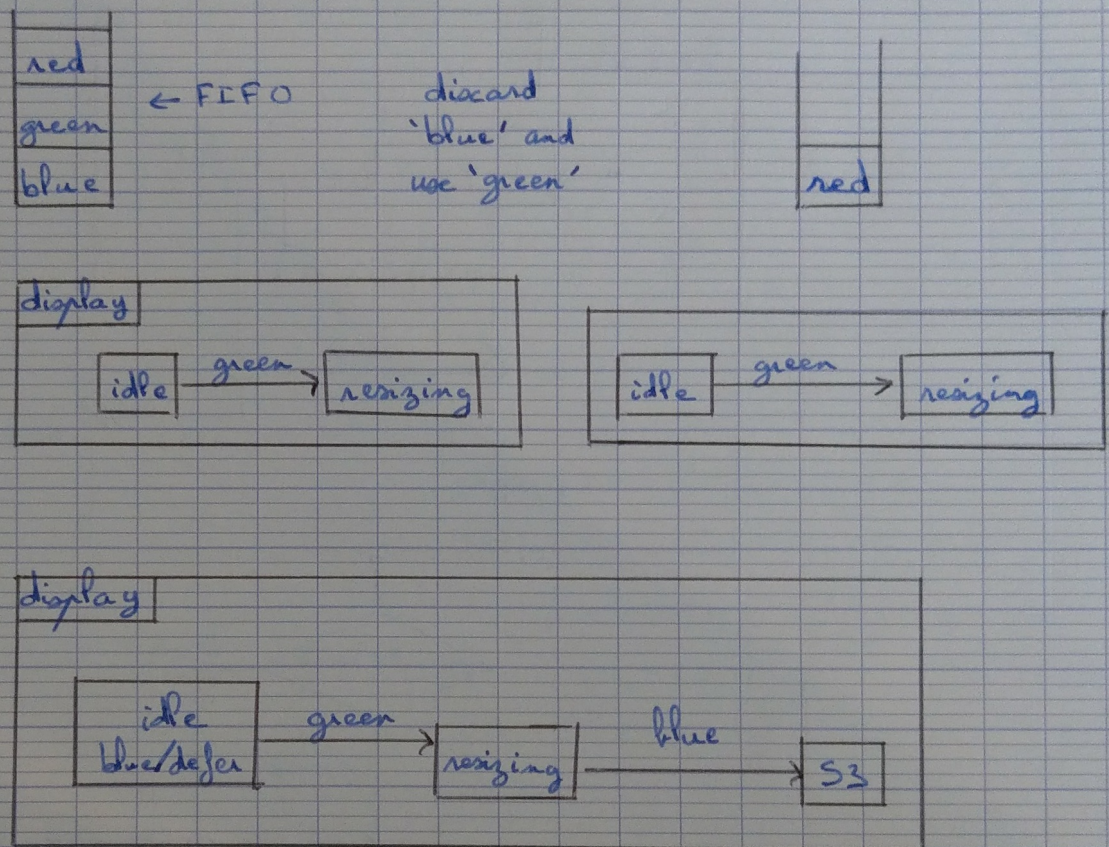


* Event queuing, discarding and deferring.

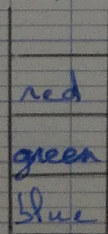
UML 2.0: "Event are dispatched and processed by the state machine, one at a time. The order of de-queuing is not defined."

"It is a semantic variation whether an event is discarded if there is no appropriate trigger defined for it."

→ FIFO and events which cannot be received are discarded.

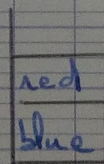


→ Step 1



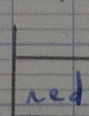
idle
consume green

→ Step 2



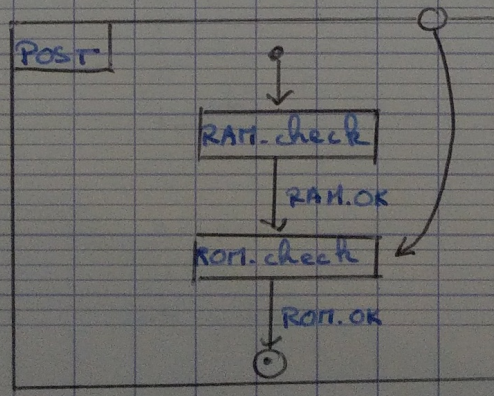
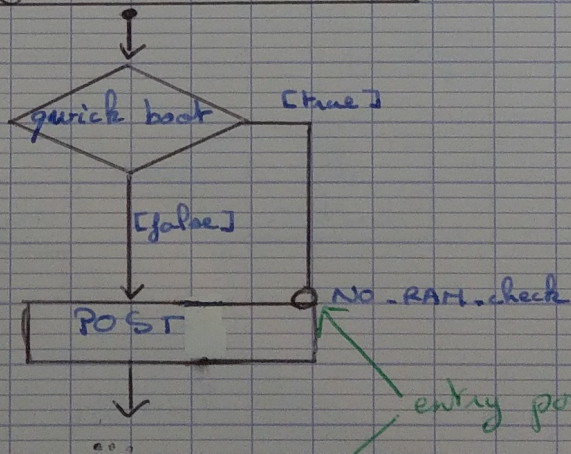
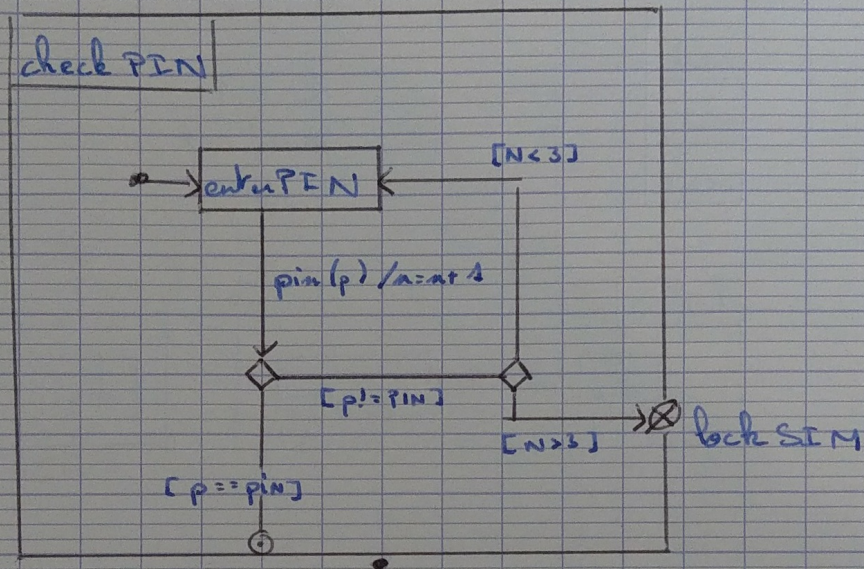
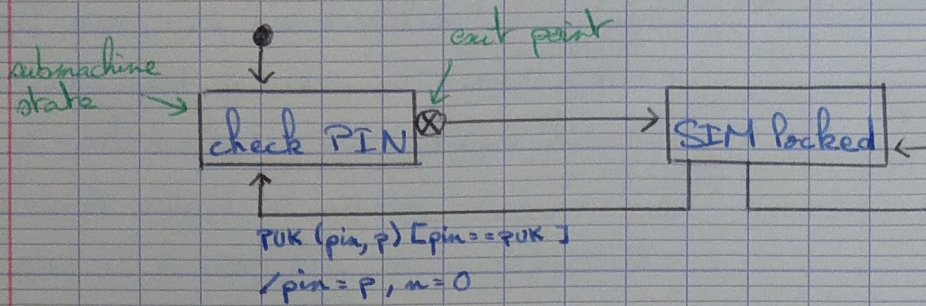
resizing
consume blue

→ Step 3

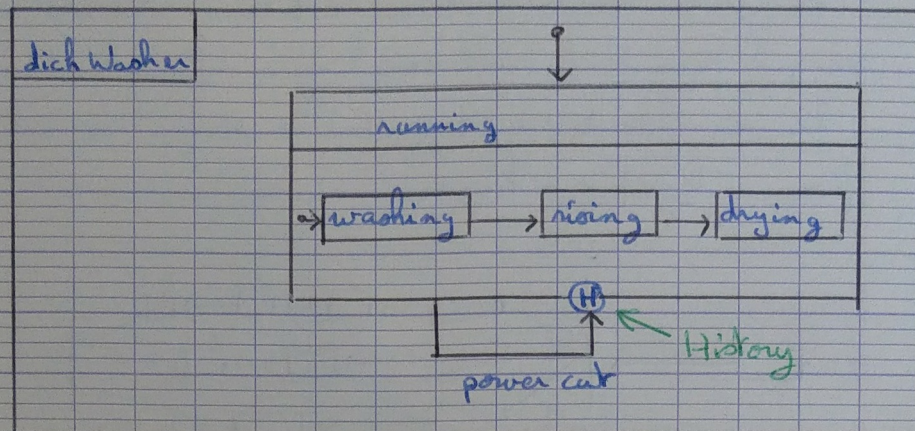


current state action.

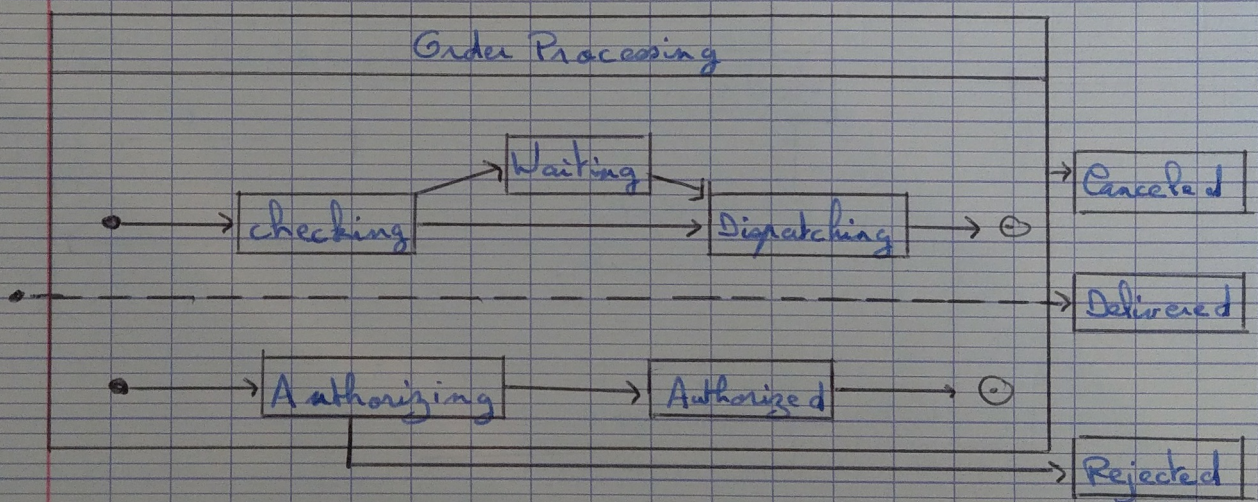
* Submachine states, entry and exit points.



- * **History**: The history pseudo-state is used to memorize the current substate when leaving a state.

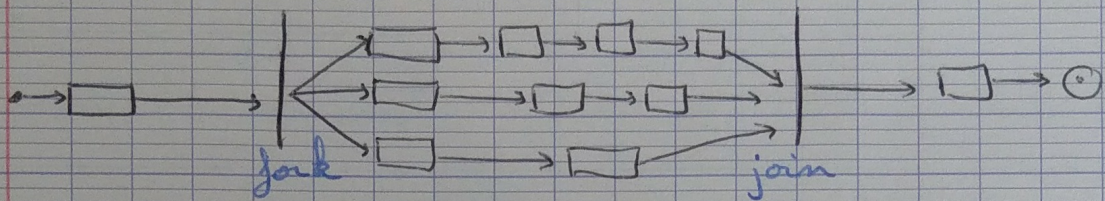


- * **Region**: A state may be split into several regions, separated by dashed lines. A region may have an optional name.

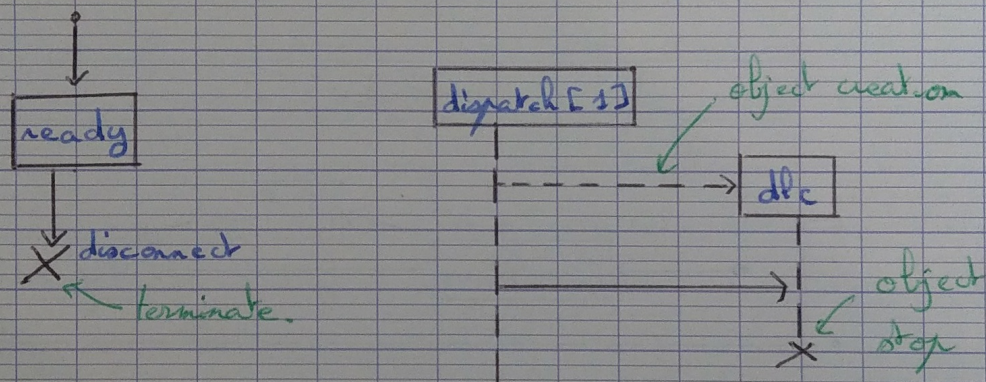


* Fork and Join

The fork pseudo-state is used to split a transition into several transitions each reaching a different orthogonal region. The join pseudo-state is used to merge several transitions coming from a different orthogonal region.



* Terminate: executing the terminate pseudo-state terminates the execution of the state-machine sequence diagram.

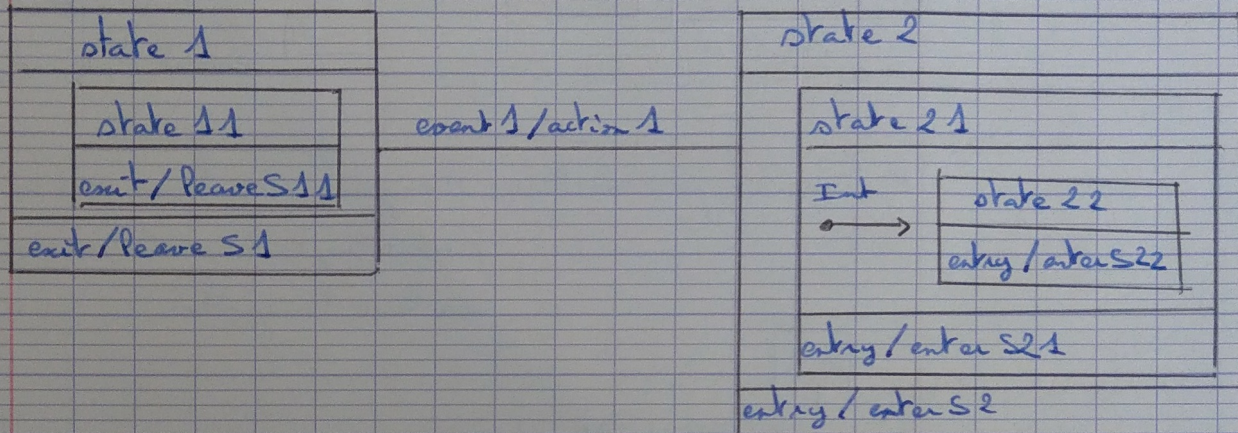
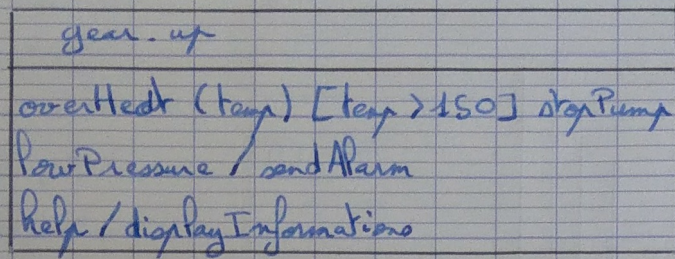


Internal Activities:

- entry: the activity specified is performed when entering the state.
- exit: the activity specified is performed when leaving the state.
- do: the activity specified is performed as long as the state machine is in the state or until the activity is completed ← state activities.

gear-up
entry / turn-on-gear-up-indicator
exit / turn-off-gear-up-indicator
do / monitor-pressure.

* Internal transition

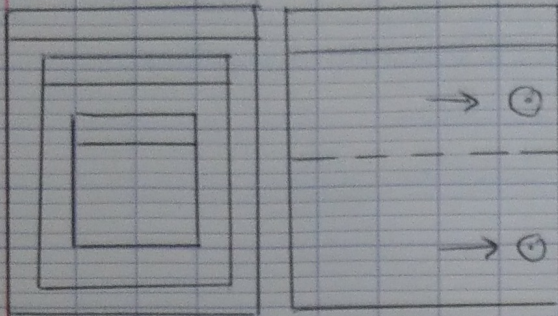


event 1 occurs: leave S11, leave S1, action 1, enter S2, enter S21, enter S22.

A composite state either:

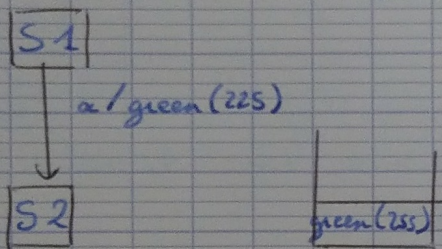
- * contains one region (non-orthogonal composite state)
- * is decomposed into two or more orthogonal regions (orthogonal composite state)
- * if the composite state is active; each of its regions can have at most one immediately contained action substate. If a state is active, all its transitively contained states are also active.
- * a transition going directly to a substate activates it explicitly and all other orthogonal regions (if any) are activated in their initial pseudo-states.
- * a transition to a final state of the region represents the completion of the behavior in the enclosing region.

- * compilation of the behavior in all orthogonal regions represents compilation of the behavior by the enclosing.

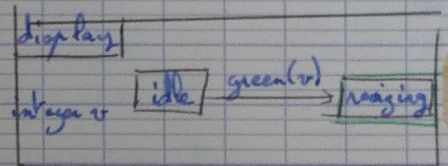
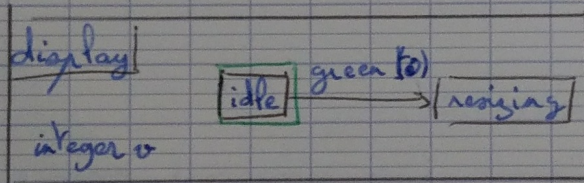


- * when exiting from a composite state, the active substates are exited recursively. The exit behaviors are executed in sequence starting with the innermost active state in the current state configuration.

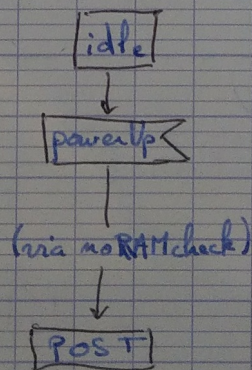
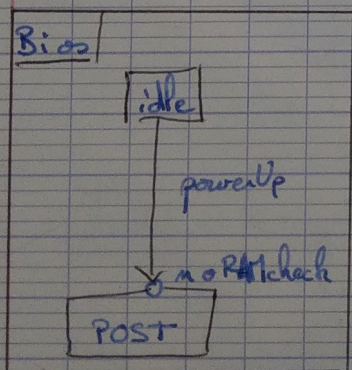
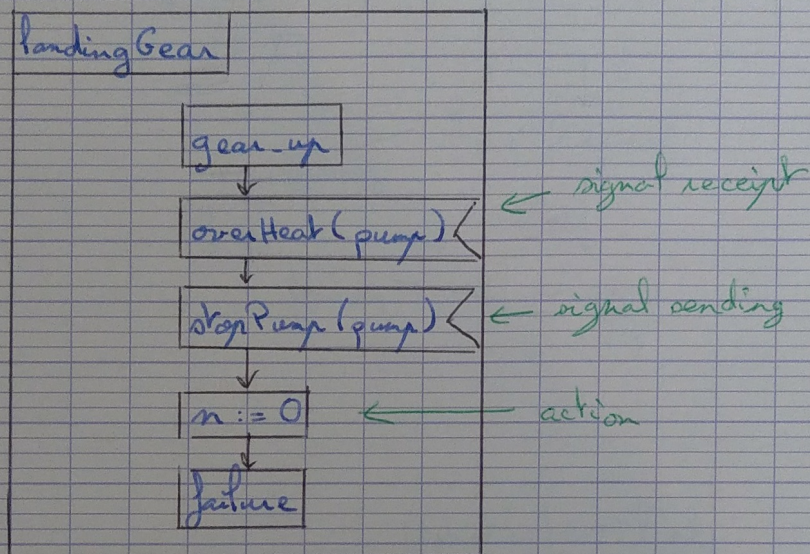
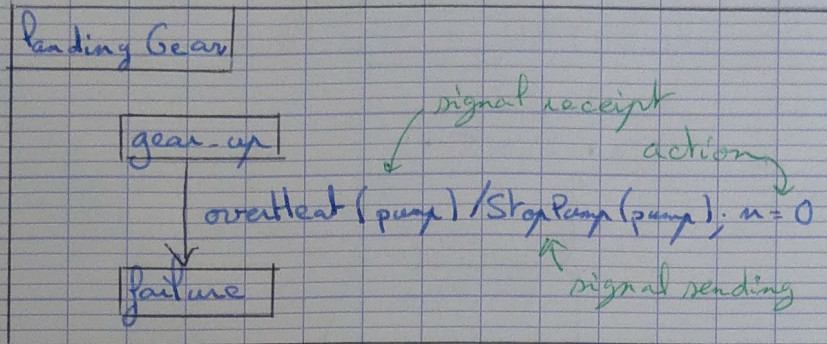
- * **Signal parameters:** A signal may have one or more parameters, to transmit values from one state machine to another.



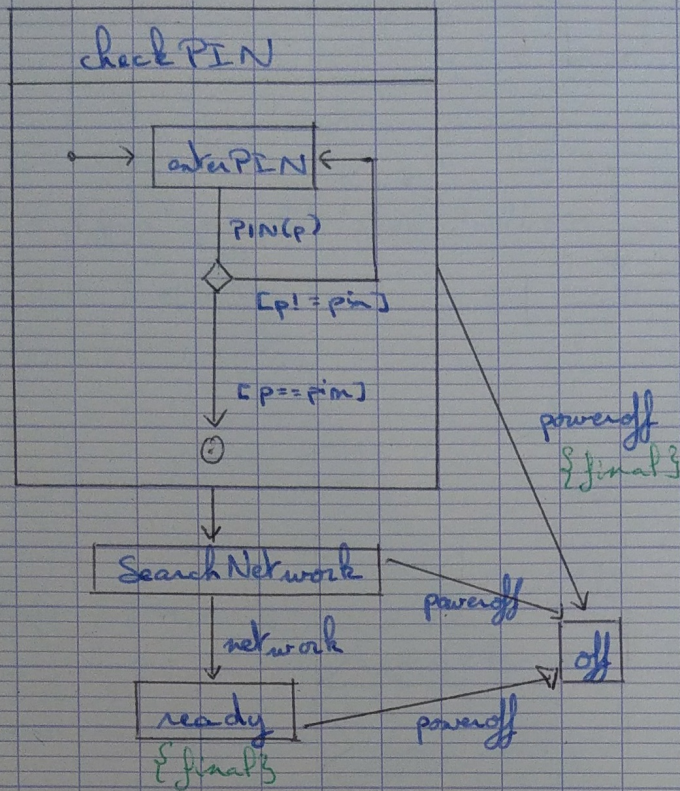
current state



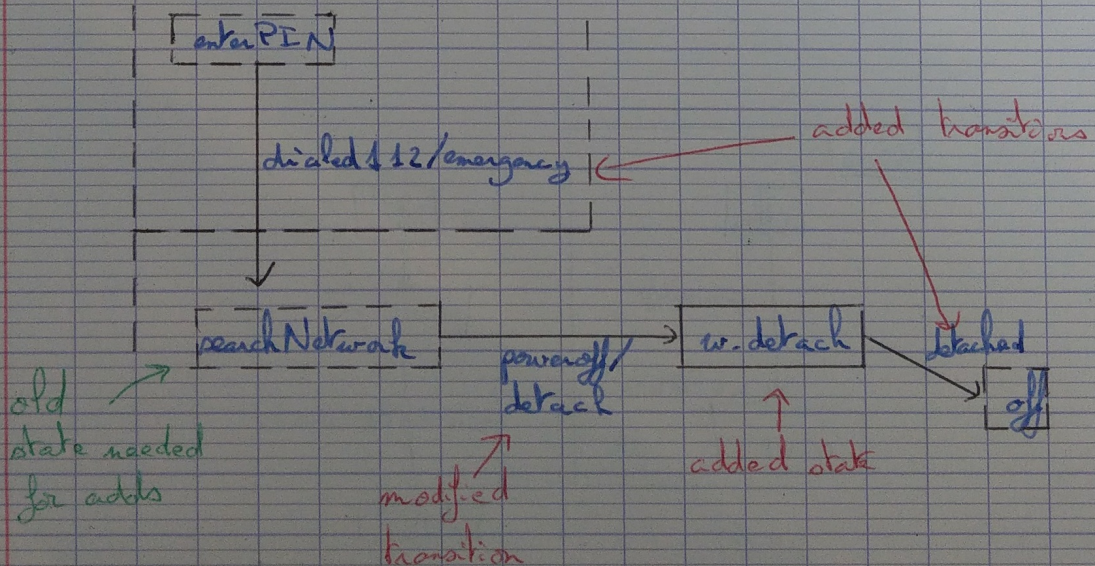
* Graphical Notations



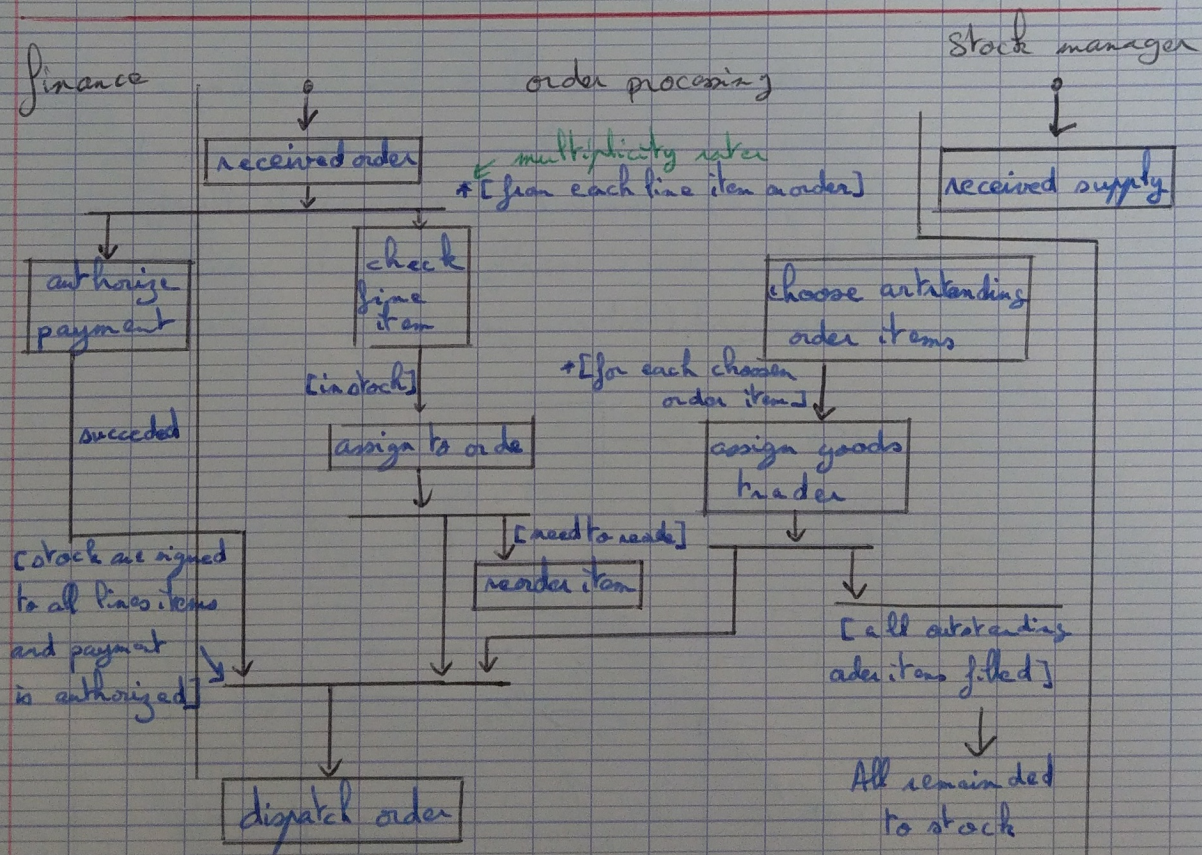
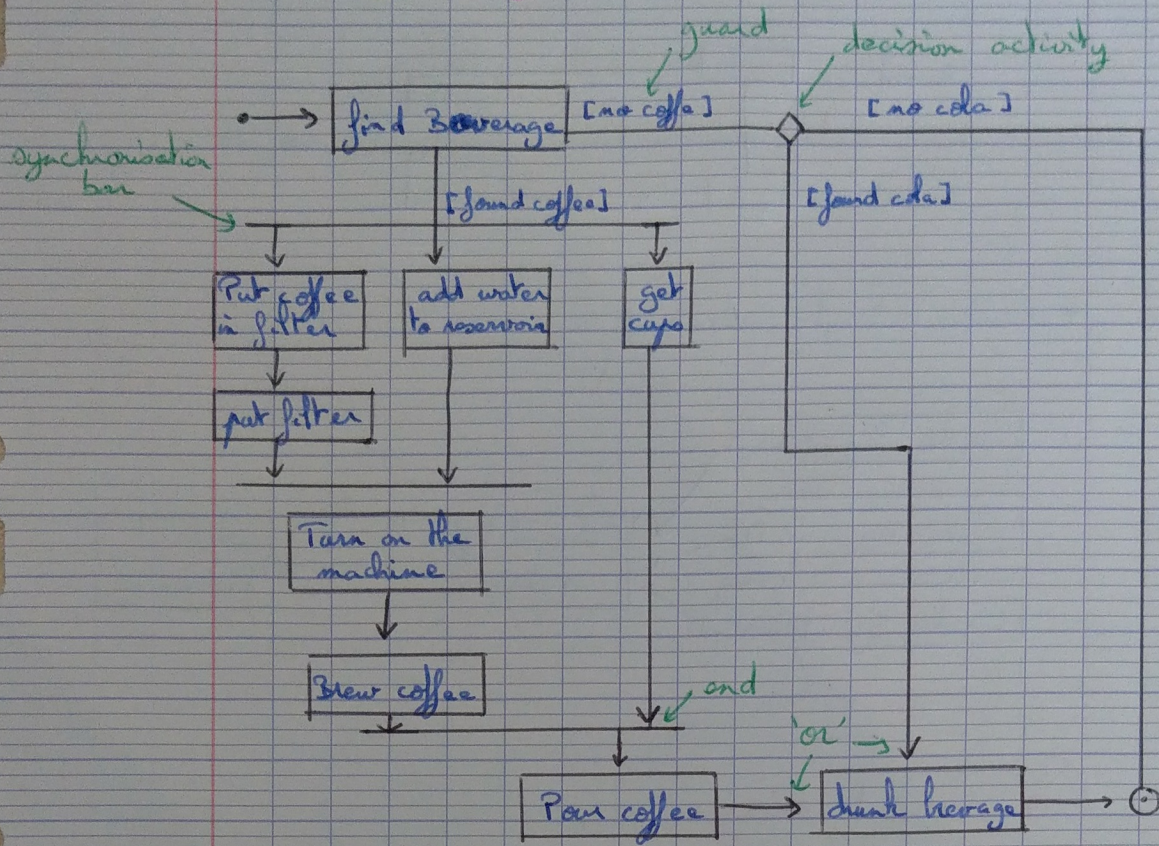
* state machine extensions



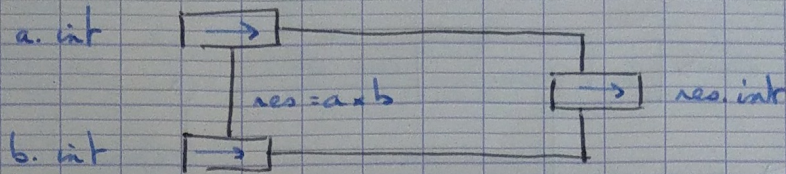
check PIN (extended)



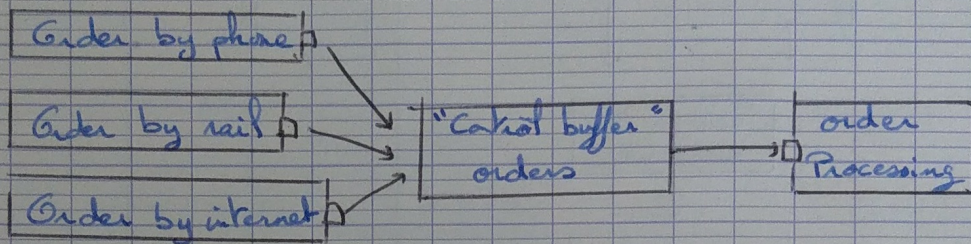
Activity diagrams.



* Input/Output pin (data flow)



* Central buffer mode



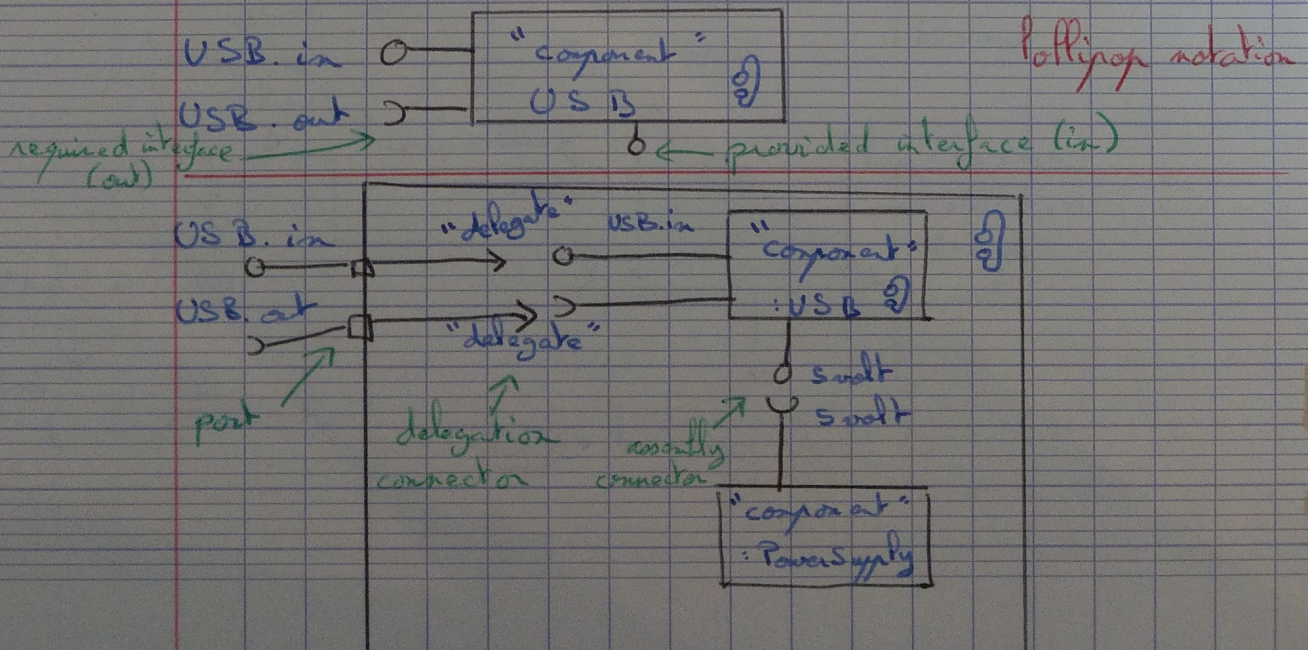
Component diagrams.

A component is a black-box modelling a replaceable part of a system.

A component may contain or not other components.

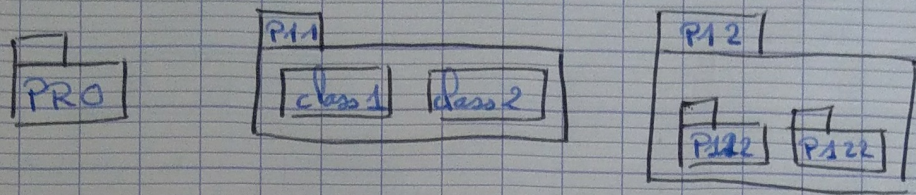
A component communicates with outside through interfaces.

Interface define exactly the signals provided (that can be received) or required (that can go out) by a component.



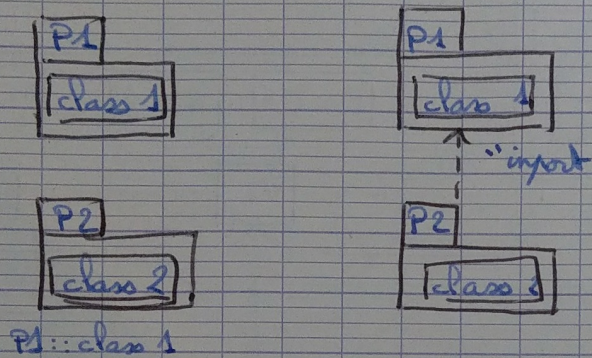
lollipop notation

Package diagram

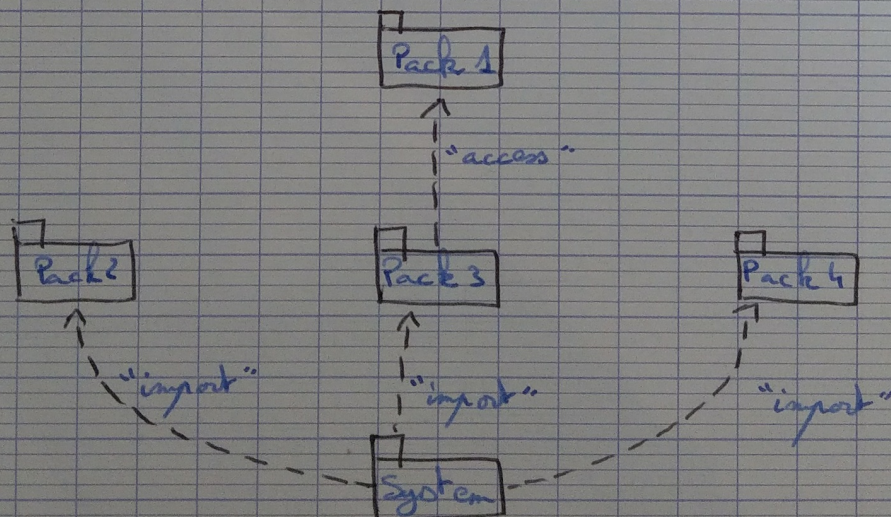


Package import relationship:

- "import" provides public visibility.
- "access" provides private visibility.

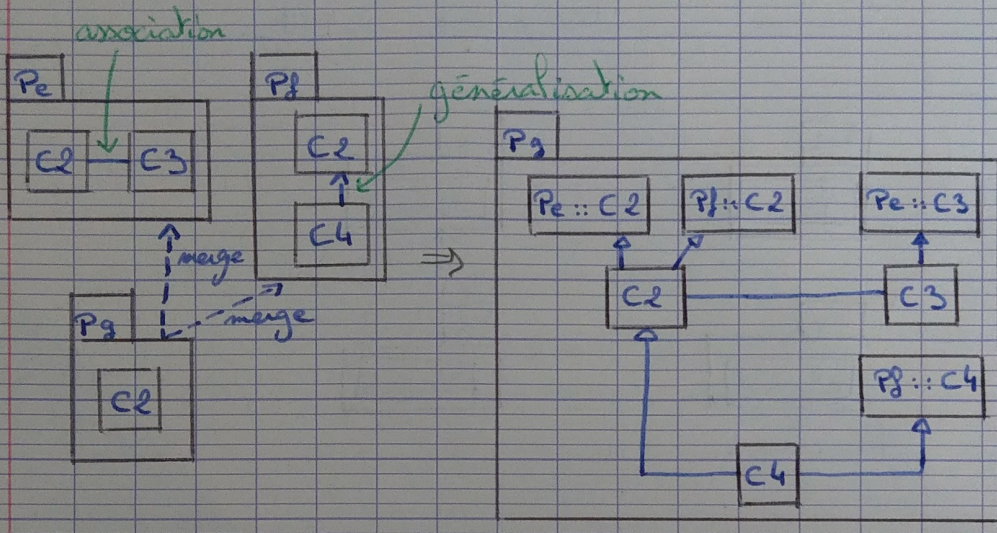
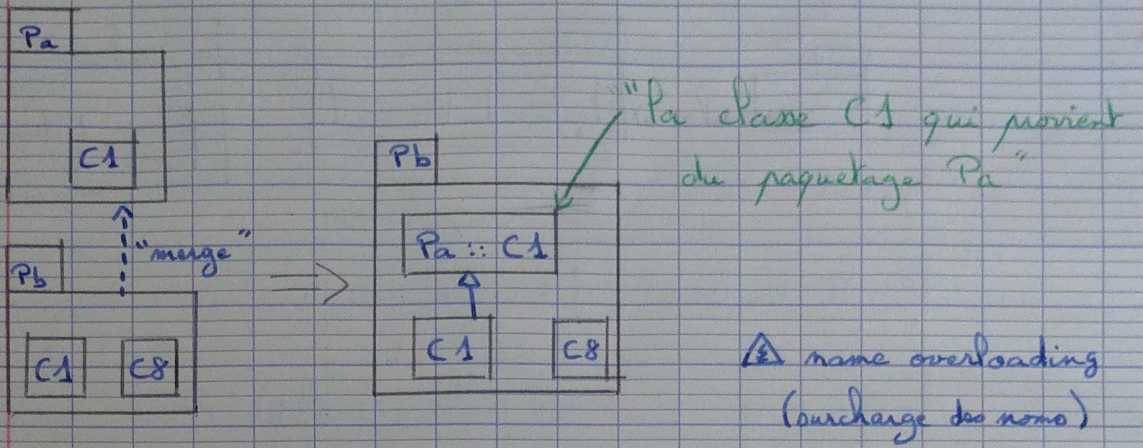


class 4 is visible from class 2



Elements imported from pack 1 are not visible from system.

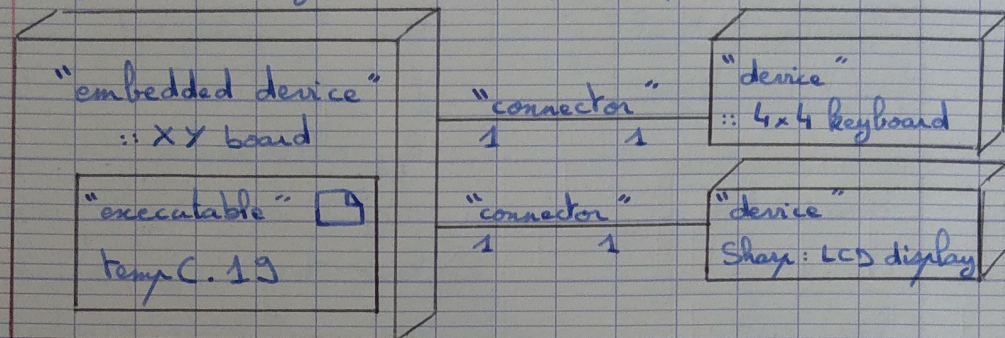
Package merge relationship



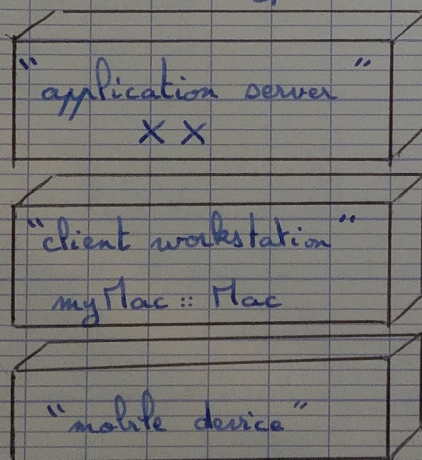
Deployment Diagram

A deployment diagram models the run-time architecture of a system, showing the mapping of software artifacts to nodes. Nodes are either hardware or software elements.

Artifact result from the development process:



Nodes Stereotypes



etc ...

examples of artifacts:

" source "

" file "

" library "

" device "

The Object Constraint Language (OCL)

{ ~ }
 ↑ any language.

Media
kind: MediaKind
capacity: Natural
format(): Integer

"enumeration"
MediaKind
floppyDisk
hardDisk
C.D

Context Media inv

if self.kind = MediaKind::floppyDisk
 then self.capacity < 2 endif

if self.kind = MediaKind::CD

then self.capacity < 750 endif

-- 750 actually ← commentaire

Context Media :: format(): Integer

pre. self.kind <> MediaKind::CD

post. self.capacity <= self.capacity @pre

valeur de capacity au moment
 du pre → avant d'avoir
 exécuté l'opération.

A constraint is a packagable element which represents some condition, restriction or assertion related to some element or several elements.

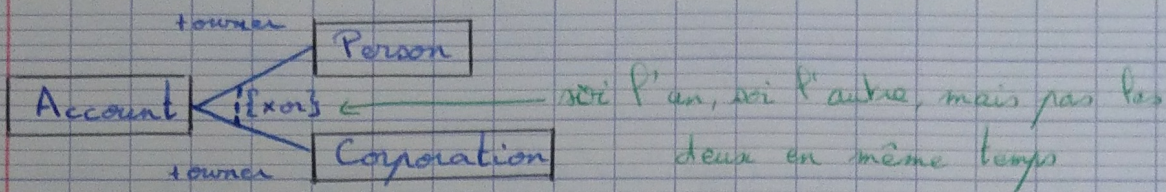
It is usually specified by a boolean expression which must evaluate to true or false.

There are many possible kind of owners for a constraint. The owner of a constraint determines when the constraint is evaluated. (ex: operation → pre/post).

Constraints could have an optional name, though usually it is anonymous.

'obligatoire'
 [optionnel]
 remplaceur

constraint ::= '{' [name ':'] boolean-expression '}'



Different kinds of Constraints

* Invariants (over a class):

a constraint that must be met by all instance of the class.

ex: context Class 1

inv attrib1 > 100

context Class 2

inv secondInvariant: attrib2 < 100

* Pre and Post conditions (over a method):

must be true before (pre) or after (post) the method.

ex: context Class1:: method(r1: Integer): Integer

pre valueIsLargeEnough: r1 > 100

post attrib1 >= attrib1 @ pre + 100 and result > r1/10

* Query definition

ex: context Class1:: query1(r: Integer): Integer

body: r + 100 + attrib1.

context Typename inv

context Packagename :: Typename inv

context Packagename1 :: Packagename2 :: Typename inv

Basic types in ACL.

boolean / not / or / and / xor / implies / if ^{boolean} then ^{anything} else $\leftrightarrow$
 3 values different: null, false, true (three-valued logic).

b	not b
null	null
false	true
true	false

b1 or b2	b2
	null false true
b1 null	null null true
false	null false true
true	true true true

b1 and b2	b2
	null false true
b1 null	null false null
false	false false false
true	null false true

Integer, Real: = / $\leftrightarrow$ / < / <= / > / >=
 + / - / * / / / abs / max / min

Integer: mod / div

Real: round / floor

String: concat / size / tolower / toupper / = / $\leftrightarrow$

examples:

'New York'. tolower $\rightarrow$ 'new york'

'Ohio'. size $\rightarrow$ 4

1+2 $\rightarrow$ 3

(7, 8). floor $\rightarrow$ 7

7. mod(3) $\rightarrow$ 1

'Washington'. concat(' State') $\rightarrow$ 'Washington State'

* Collection types

	ordered	uniqueness
Set	no	yes
(multi-element) (multi-set) Bag	no	no
Sequence	yes	no

* Operation select.

Operation select picks out a subcollection of elements that meet a particular condition.

collection $\rightarrow$ select ($v \mid$ boolean-expression-with- v)

ex: bag {7, 3, 5} $\rightarrow$ select ($x \mid x \geq 5$) $\Rightarrow$ bag {7, 5}

* Operation reject.

Operation reject picks out a subcollection of elements that fail a particular condition.

collection $\rightarrow$ reject ($v \mid$ boolean-expression-with- v)

ex: bag {7, 3, 5} $\rightarrow$ reject ($x \mid x \geq 5$) $\Rightarrow$ bag {3}

union: set $\times$ set $\rightarrow$ set
set $\times$ bag $\rightarrow$ bag

intersection: set $\times$ set $\rightarrow$ set
set $\times$ bag $\rightarrow$ set

including: set $\times$ any $\rightarrow$ set

$a \rightarrow$ including (b): "all elements of a with b added if it was not in the set a "

excluding: set $\times$ any $\rightarrow$ set

$a \rightarrow$ excluding (b): "all elements of a without b , it is excluding if it was in the set a "

- set $\times$ set $\rightarrow$ set

= set $\times$ set $\rightarrow$ boolean

asBag
asSequence
asSet

set $\rightarrow$ bag
set $\rightarrow$ sequence
bag $\rightarrow$ set

bag: union, intersection
sequence: first, last.

at
append
prepend

sequence $\times$ integer $\rightarrow$ type of elements in the sequence
sequence $\times$ any $\rightarrow$ sequence
sequence $\times$ any $\rightarrow$ sequence.

Collection: size, count(e), isEmpty, notEmpty
include: collection $\times$ any $\rightarrow$ boolean
includesAll: collection $\times$ collection $\rightarrow$ boolean
sum (if numerical elements)

ofType

the type of a

isTypeOf

true if a is type of b

isKindOf

true if a is of type b or any subtype of b.

* Operation collect.

it creates a new collection by applying an expression to each element of a collection.

ex: bag {2, 4, 3} $\rightarrow$ collect(x | x * x) $\Rightarrow$ bag {4, 16, 9}

* Operation forAll

it results in true if expression is true for all elements of the collection.

ex: bag {10, 5, 4} $\rightarrow$ forAll(x | x > 2) $\Rightarrow$ true

* Operation Exists.

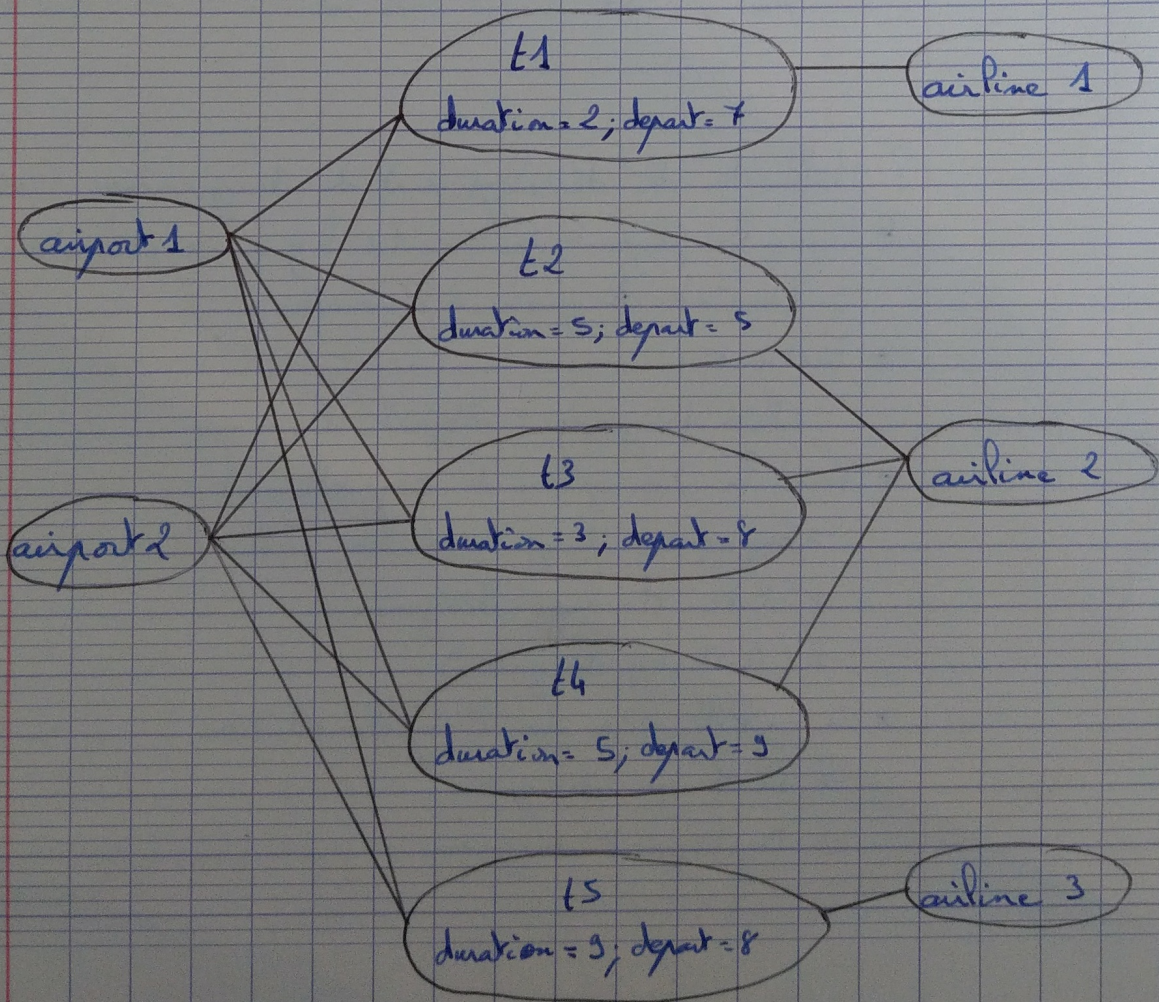
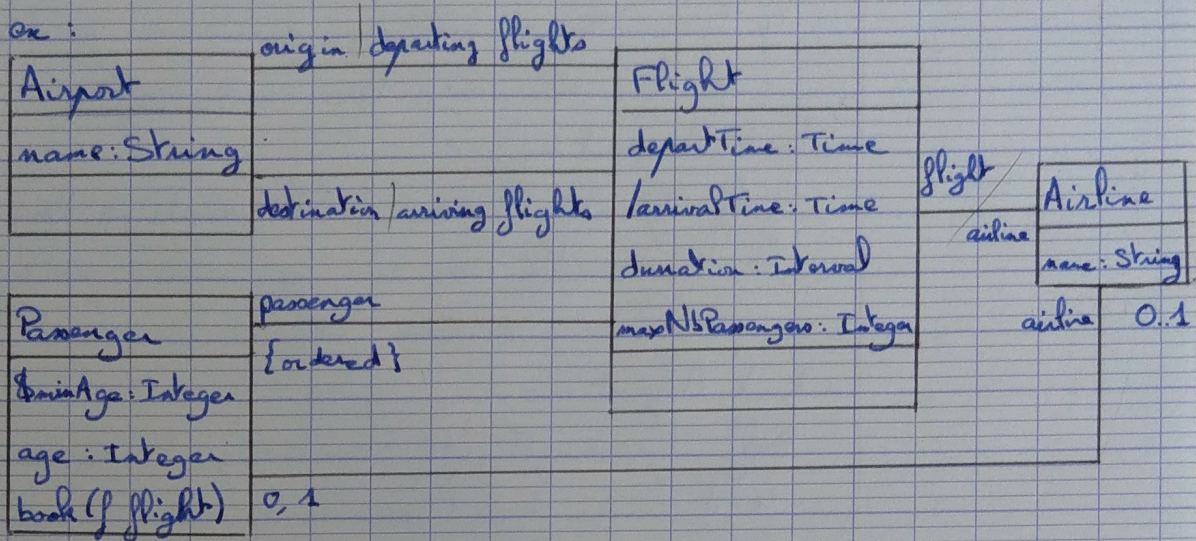
it results in true if there is at least one element in the collection for which the expression is true.

ex: bag {10, 5, 4} $\rightarrow$ Exists(x | x < 5) $\Rightarrow$ true

Local variables: `let var: Type = <expression 1> in <expression 2> .`

- * Operation iterate.

* Operation iterate.
collection → iterate (elem: Type, answer: Type = <initial value> |
<expression-with-elem-and-answer>).



context Airport inv;

Self. arrivingFlights $\rightarrow$ collect(airline) $\rightarrow$ not Empty

$\hookrightarrow$ l'ensemble des compagnies d'aviation qui gèrent les vols à l'arrivée n'est pas vide. $\Rightarrow$ vrai

context Airport inv;

Self. departingFlights $\rightarrow$ select(duration < 4) $\rightarrow$ not Empty

$\hookrightarrow$ l'ensemble des vols au départ dont la durée est inférieure à 4 n'est pas vide. $\Rightarrow$ vrai

context Airport inv;

Self. departingFlights $\rightarrow$ forall(departTime.hour > 6)

$\hookrightarrow$ l'ensemble des vols au départ partant après six heures $\Rightarrow$ faux (t2)

context Airport inv;

Self. departingFlights $\rightarrow$ exists(departTime.hour < 6)

$\hookrightarrow$ il existe, dans l'ensemble des vols au départ, certains qui partent avant six heures $\Rightarrow$ vrai. (t2)

? Méthode de développement de diagrammes OML

* Dans la plupart des cas (s'il y a un utilisateur):
Use Case diagram + comment.

* Partout:

Class diagram + Object diagram

* Pour la description de comportement:

- $\rightarrow$ Sequence diagram (also additional comment for Use Case diagram)
- $\rightarrow$ State Machine diagram
- $\rightarrow$ (Activity diagram)

* Si considération de l'implémentation:

- $\rightarrow$ Package diagram
- $\rightarrow$ Deployment diagram (if distributed system).

* Problem Frames

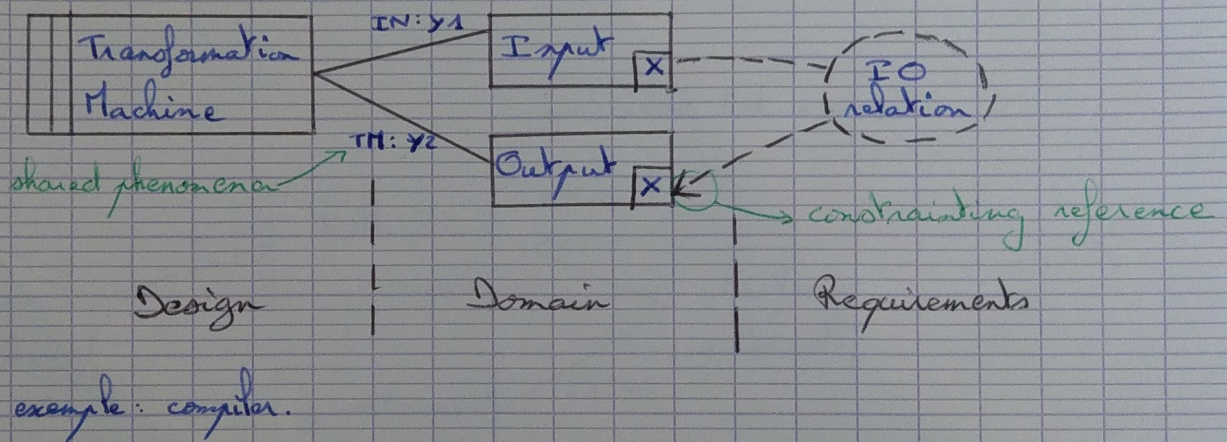
Some basic problem frames which can combined (and adapted)

- ↳ transformation frame (user) / work pieces (user)
- ↳ commanded behaviour (user) / required behaviour (user)
- ↳ commanded information (user) / required information (user)

- Domain Model
- Requirements
- Design

Domain $\begin{cases} X & \text{lexical (data, files, ..., inert)} \\ C & \text{causal (with a behaviour)} \\ B & \text{biddable (people).} \end{cases}$

* Transformation Frame



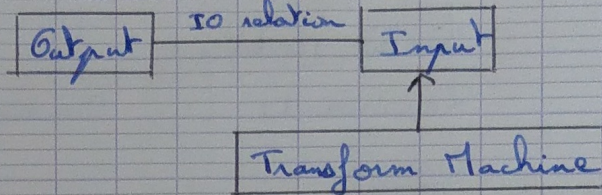
UML development

→ Domain: Class d.

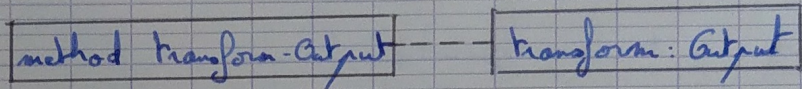
Input

Output

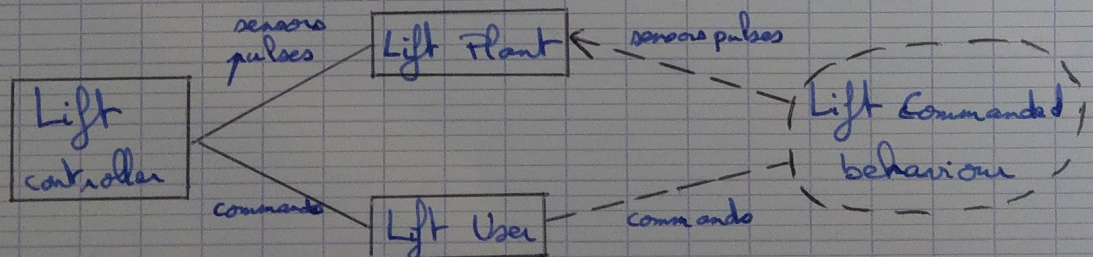
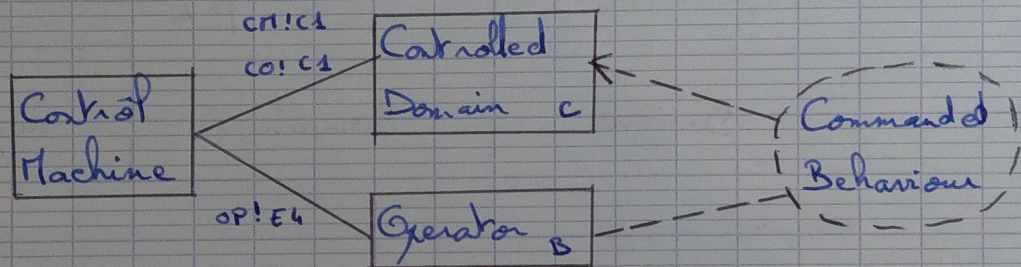
→ Requirements:



→ Design:

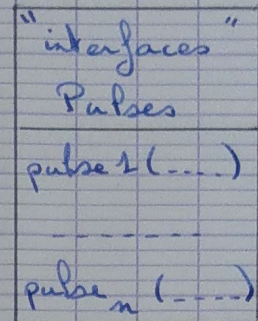
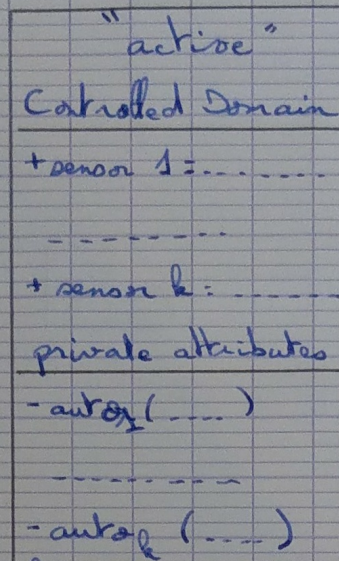


Commanded Behaviour:



*UML model

→ Domain:

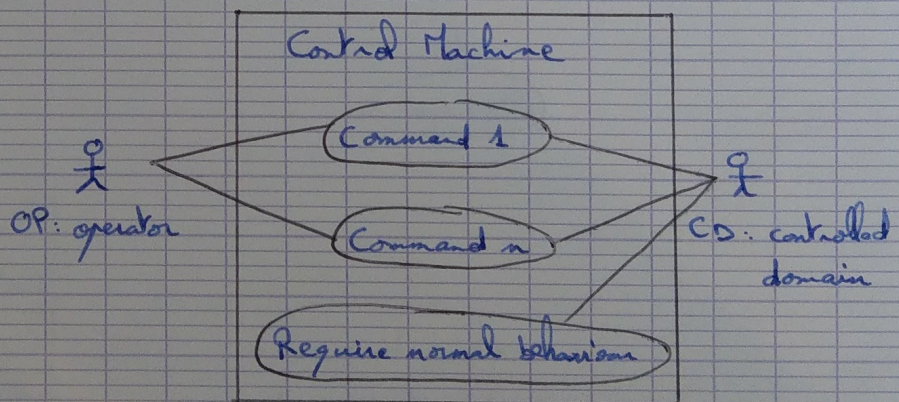


State Machine describing the behaviour.

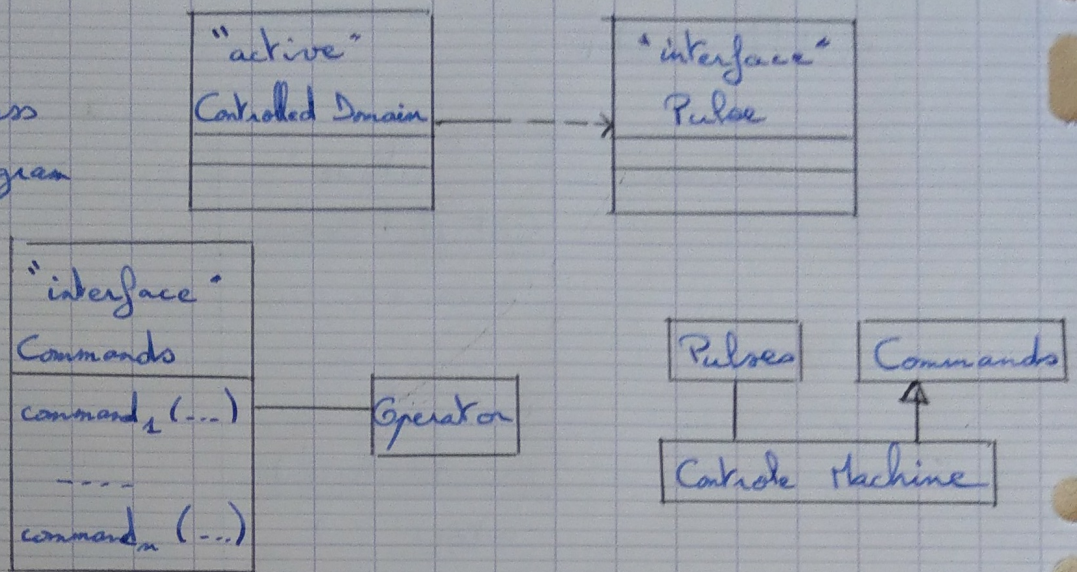
*Commanded Behaviour

→ Requirements:

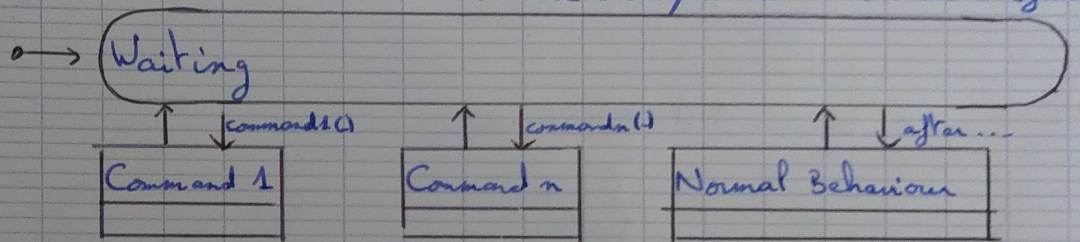
Use Case
diagram



Class
diagram



use case description state machine diagram



Design

- Class diagram
- State Machine diagram

