6. UML 2.0: 13 sublanguages

UML presents the state-of-the-art in SE
with respect to combining

specification
understanding
visualization
analysis

of (software) systems

and with respect to additionally combining
support for

construction
documentation

of software systems

to that aim, already UML 1.4 / 1.5 had

9 diagram languages
as visual sublanguages:

divided into

4 static diagram languages

5 dynamic diagram languages

UML 2.0 even has

13 diagram languages
as visual sublanguages

divided into

6 static diagram languages

7 dynamic diagram languages

the 6 static diagram languages are

- class diagram
  logical structure unit, on type level

- object diagram
  logical structure unit, on instance level

NB

some people prefer to consider
the class diagram and the object diagram
as two manifestations of
the same “class/object diagram”

furthermore there are

- component diagram
  physical structure:
    specifying a software (sub)system

  is now one component in isolation

  can now have ports:
    physical interfaces for
    connecting physical links (channels) to

- deployment diagram
  physical structure:

    specifying component allocation
    on hardware

    together with machine connections
    (eg. LANs, busses)

the following two ar new diagram languages

- package diagram
  - grouping of whatever model fragments

  - more or less like old package

  - nevertheless,
    rather class-like or component-like

  - can now have ports too

- composite structure diagram
  this is a separate diagram for
    physically interconnected elements

  the (new) collaboration diagram is seen as
  a particular composite structure diagram

the 7 dynamic (behavioural)
diagram languages are

- use case diagram
  functionality as declarative behaviour,
    without any time or step ordering
  (this is the only dynamic diagram without
    such ordering; so it is
    uniquely declarative in dynamics)

  roughly, globally indicating
  - the main behavioural units
  - where such units are used

  can “now” be located
    in whatever structural item
    ie. inside a certain class

  can “now” be handled as class-like
    ie. can participate in other relationships
    than a uses relationship

- statemachine diagram
  (former statechart diagram)
  detailed behaviour,

  usually local to a class/object),

  not necessarily sequential (one thread)
  but often more or less so

- activity diagram
  global behaviour,

  often concurrent,
    nearly data-flow-process-like

  usually organised in swim lanes

  although non-local, its being global
    is comonly restricted
    to a concrete collaboration

note:

whereas the use case diagram is declarative,

the statemachine and activity diagrams are
really behavioural, specified in terms of
consecutive steps

even commucative steps can be visible
in the latter two,

but interaction still remains (rather) hidden

interaction is far more explicitly addressed
in the remaining 4 diagram languages

- communication diagram
  (former collaboration diagram)
  interaction between objects,

  scenario-wise, by enumerating the steps

  one scenarion per diagram

a communication diagram can be viewed
as an specialized collaboration diagram:

the structure of the physically connected
objects (or roles thereof)

is enhanced with

one collaborative scenario,
the communicative (interaction) steps located
at the link via which the sending is done

- sequence diagram (sd)
  interaction between objects,

  scenario-wise, by placing the steps along
  scale-less time axes, one axe per object

  now all scenario's together:
  "one" per sd-frame,
  sd-frames composed within a
  generalized while-structure frame

note;
this is a substantial enhancement compared to
the 1.4 version,
where only one scenario was allowed

there are two new dynamic diagrams:

- timing diagram:
  sequence diagram
  (mostly simple ones only)
  with state changes per lifeline and with time

- interaction overview diagram
  while-structured composition of
  sequence diagram fragments

  structure of an activity diagram
  with lifelines instead of swim lanes
  and with interactions
  instead of activities

note:

out of 7 dynamic diagrams
4 address interaction explicitly

but these 4 do not address other behaviour

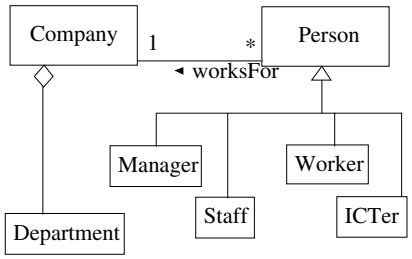
the 2 diagrams addressing
the other behaviour
do not address interaction (so much)

the 1 declarative dynamic diagram
does not address
step-wise behaviour
nor interaction

rough impression of the 13 diagrams:

(see Fowler for more details - but not all)

class diagram:



rectangles are the classes

edges between the classes are the relationships

triangle (in edge) refers to is-a relationship:
inheritance

diamond (in edge) refers to part-of relationship:
aggregation / composition

black triangle at edge:
read direction of relationship name

a more elaborate class description:

name compartment
attribute compartment
operation compartment
other, e.g. responsibility
or signal compartment

example attributes of Person:

address,
Sofi-number,
age,
salary,
function

together with type indication

example operations of Staff:

test,
review,
inform,
study

together with parameters

attribute and operation compartment are
nearly always present

example responsibilities of ICTer:

requirements-engineering,
designing,
coding

responsibilities might be combined
with contracts
such responsibilities might correspond
to (the) operations specified

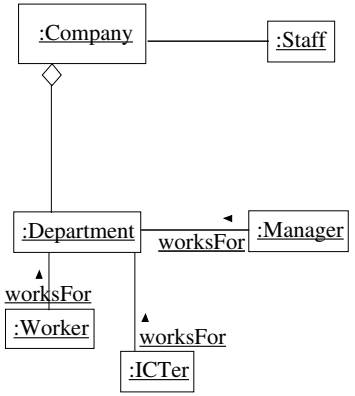
the signals originate from real-time situations:
they consist of name and type of
incoming signals as received,
outgoing signals as sent;

sending / receiving usually goes via ports
ports are rather specialised classes,
on their instance level linked via connector
representing the physical channel for
sending / receiving between ports

ports can serve as physical interface of a group
of objects, eg, a package or component

ideas are from ROOM, via UML-RT, now in
UML 2.0

object diagram:

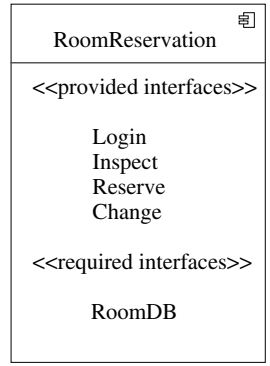


underlined text indicates an instantiation

rectangles are the objects

edges between the classes are
the links / relations

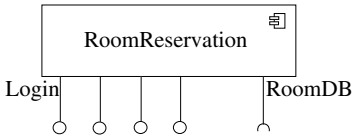
component diagram: now iconized



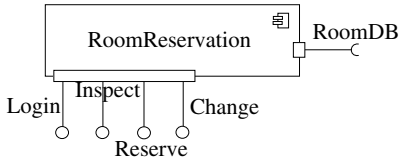
components are
units of physical implementation

like a software system
or (!) an organisational working unit or artefact

can also be visualized as:

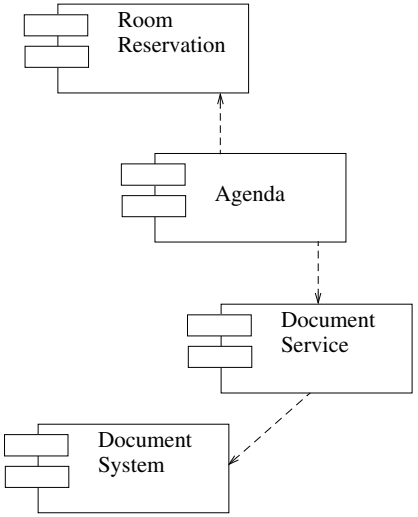


or as: with ports at the component border



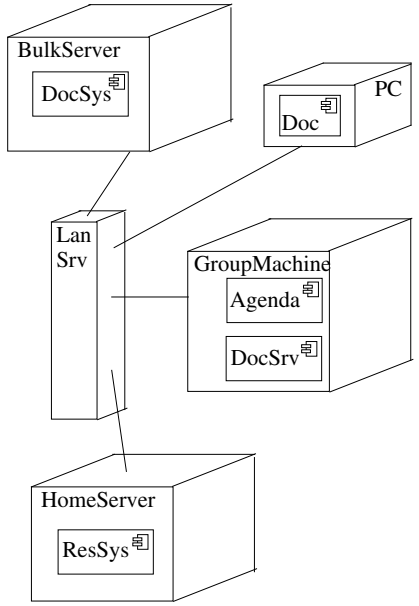
in the latter case,
the interfaces name the functionality provided
and the actor required, where
the ports are to have suitable protocol roles for
realizing the corresponding interactions

compare to old component diagram (UML1.4):



the above is now expressed through
the new composite structure diagram

deployment diagram:



a deployment diagram visualizes
the allocation - called deployment - of compo-
nents to physical hardware - called nodes

(see DocSys, Agenda, DocSrv, ResSys
inspired by transp. 6.22)

3-dimensional-boxes represent the nodes

the links between nodes are
communication paths:

physical connections
via which nodes communicate

even UML 2.0 uses the deployment diagram
only for deployment to computer-like hardware

so there is no counterpart-interpretation for organisations

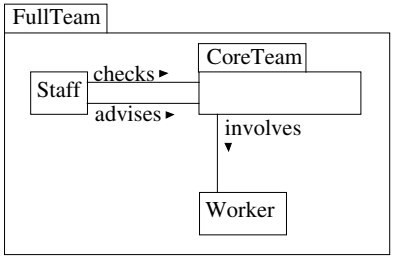
apart from being stored in a database, deployed on some central node,
artifacts of eg current interest are deployed, ie physically present, on one or more local nodes

(see eg. document Doc on the PC)

package diagram:

it is considered to be a new diagram,
but it is rather like it was

a package is a grouping
(of whatever UML fragments
but very often it is class-like:
just a class / object diagram fragment):



a package can have ports, like a component

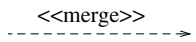
- it has the “tab” at the left upper corner

naming of its members is somewhat strict:

each package has a “name space” for its members, the elements it “owns”

therefore, a package can import members from other packages
(possibly via qualification or via aliasing)

also, a package can “merge” with another:



where the source package is merging with the target package, the merged one

composite structure diagram:

a really new diagram, constituting

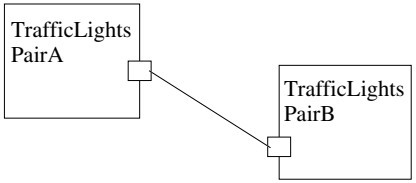
a composition of interconnected elements modelling how instances cooperate at run time

there are two manners of visual representation:

- composition via ports and connectors
connected elements are
classes, packages, components, ...

- composition as collaboration
connected elements are
restricted - as far as relevant - views of
classes (or whatever) denoted as roles

via ports and connectors:



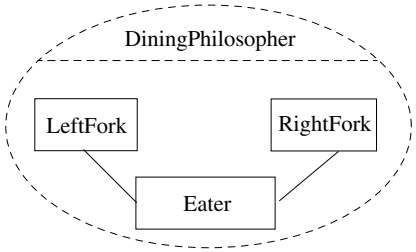
communication of an element - a pair of German traffic lights - with its environment

takes place via its port(s) only

a port specifies either the services provided / required or more detailed role behaviour

a connector of the right type - conforming to the ports it connects - transmits the communication, possibly after some extra manipulation

as collaboration:



the collaboration expresses:

3 elements exist being the roles of:
- Eater connected to a Philosopher,
- LeftFork and RightFork, each connected to a different Fork

the idea is:

5 objects instances of class Philosopher exist

also 5 objects instances of class Fork exist

the 5 Phils are seated at a round table, where they can eat after having thought long enough

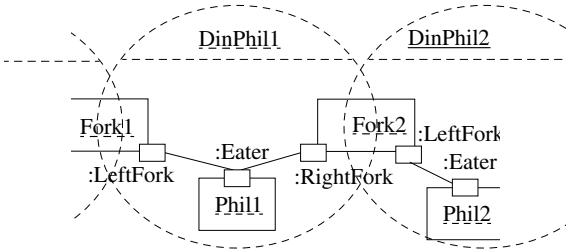
for eating the need the 2 Forks each of which they share with their respective two neighbour Phils

they may take a Fork only if it is not in use by the other Phil

(elegant illustrations of deadlock, starvation
and also of a good solution)

note:
a collaboration expresses the structural information of UML 1.4’s collaboration diagram

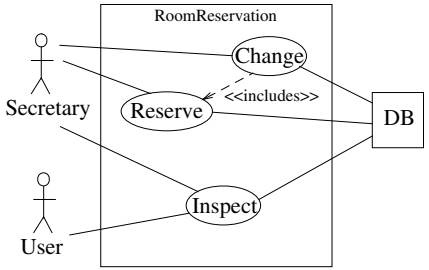
the same collaboration
in an unusual and instantiated form:



so far we have presented
the 6 structural diagram sublanguages
of UML 2.0

hereafter we present its 7 dynamical
diagram sublanguages

use case diagram:



ovals are the use cases: each use case has one or more scenario's, describing the interaction between the use case and the actor(s)

actors are persons or things outside the system

use cases and actors are "class-like"

the box containing the use case is called
system boundary

each use case refers to behaviour (scenario's)

but nothing in the diagram refers to
explicit or implicit chronological ordering

<<includes>> and also <<extends>> are
dependencies indicating structural connections

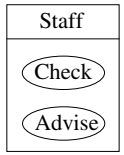
relationships between actors and use cases are
uses dependencies
in case of a human actor:
usually directed to the use case
in case of a non human actor:
often directed to the actor, but not always

although a use case diagram is about behaviour,
it only specifies its structural aspect:
being there; for whom / what; connections
this is the declarativity mentioned above

so, use case diagrams remained unchanged

"but"

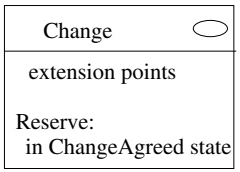
- any classifier can contain ("own") its use cases



(inspired by above FullTeam package)

main "change" here is:
it was already allowed
but now it is
more explicitly advocated / put forward

- a class-like notation for use cases, with icon



(inspired by above RoomReservation)

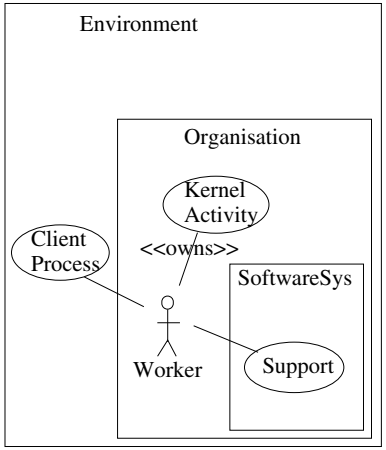
note how the above allows for

refining a use case in terms of smaller use cases,
still without any ordering

(although up to now this seems highly unusual
in UML)

a very welcome / interesting / intriguing
consequence of the above

is the following unusual appearance of
a general use case diagram:



this representation opens
new insight in modelling

even during early phases of SE
ie before designing the software system to-be

it makes sense **to model**
what, how, where, when
of the **organisation** and **environment** systems

in order to investigate **relevance / impact**
of the software system (to-be)

question:

is it reasonable to expect
world outside software system
can be sufficiently well modelled by UML

at this point:
a tentative **yes**, see role of model in ch 2, 4, 5

remaining chapters will,
among other things, consolidate the "yes"

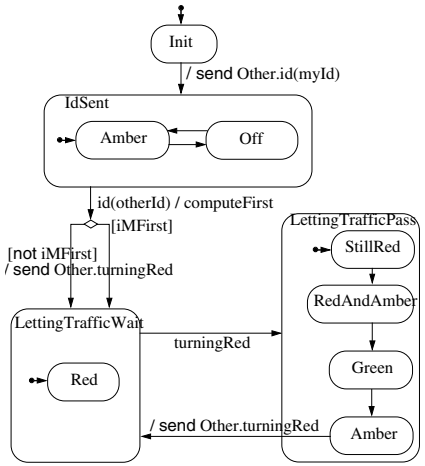
above ideas are referred to as:

Integration-Orientation

(see the CoOrg05 paper)

statemachine diagram:

very much inspired by Harel's statecharts



this is a rather technical example:

take two "equal" statemachines as above,

each statemachine specifies

one identically behaving pair of traffic lights

(German traffic lights actually,
because of state RedAndAmber

a statemachine (diagram) specifies
local behaviour, usually thread-like

rounded squares are (super / sub)states

directed edges are transitions / steps

small diamonds are pseudo-states for testing

a statemachine always is in a state,
so a transition is in one go

transition labels:

- between [and] : a guard / condition for the transition

- after / : an action list; eg. sending: "send event"; actions are atomic: "run to completion"

- before / , or without any / : "event", which corresponds to receiving: " accept event"

in a state:

- entry and exit actions can be specified

- activities can be performed (interruptable)

- refinement of a state by a substatemachine

- refinement of a state by concurrent substatemachines, separated by dashed lines

statemacines in UML also have some Petri net features:

a parallel continuation of a transition,
called fork;
a sequential continuation of two transitions,
called join:

fork: join:

very often a statemachine specifies the possible behaviours of one class / object

or the possible behaviours of an "interface" of a package or of a component

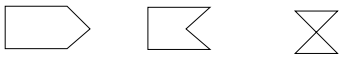
any instantiated statemachine has, at any moment, "one of its states" as current state

compared to the old statechart diagram from UML 1.4,
the statemachine diagram has remained

essentially unchanged

but

some new notation exists for the actions:



send accept accept time

accept action is the old receive, corresponding to some send elsewhere

accept time action is receiving an event that has been sent at a certain time instance, eg. at the end of each month
the visualization suggests an hour-glass / clock

statemachine is still meant for local behaviour, such as for a class, object, component

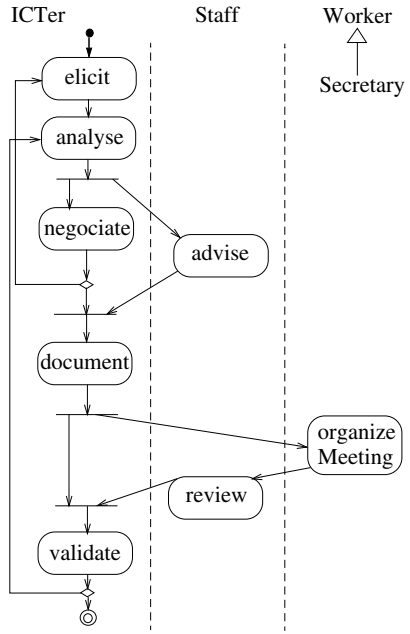
but now much mre explicitly also for

port	protocol role
connector	protocol
interface	external / visible behaviour

a superstate,
being a state containing other state(s)
is called a **composite state**
if the state is refined sequentially only;

a superstate is said to consist of **regions**,
if the state is refined into at least two parallel refinements, which are the regions

activity diagram:



more or less like statemachine diagrams, but
- states now reflect one **activity**
(commonly, a state is called an activity)

- transitions now only can have guards
(omitted above)

so:

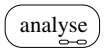
neither sends nor accepts (receives)
are to be specified
thus interaction is (usually) left implicit

activity diagram specifies non local behaviour,
across classes / objects

often, the various classes / objects involved
have their own “swim lanes”

an activity from an activity diagram
often represents a large activity only;

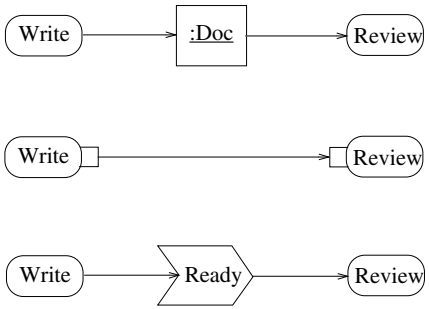
however, a state from an activity diagram
can be refined as follows:



the activity name “analyse” then refers to
a separate activity diagram with this name
the new diagram may have various swim lanes

can be combined with “object flow”,
reflecting how some passive object
like a document or a product or an item
subsequently is being processed
by the various classes / objects involved

the (new) notation for such object flow is:



in the third case of the object flow,
the object is a “signal object”

this makes interaction in an activity diagram
- unusual, although not strictly forbidden -
visually more explicit

note:

a passive class / object (that is flowing)
may also be present via its own swim lane

activities in such a passive swim lane
usually correspond to
rather “passive activities”,
such as
registration, update, transformation of shape,
transportation

“things that happen to” such a class / object,
as if an administrator is taking notes thereof

furthermore,

more dimensional swim lanes can help

to differentiate eg
not only between actors in a certain function
but also, and simultaneously, between locations

and

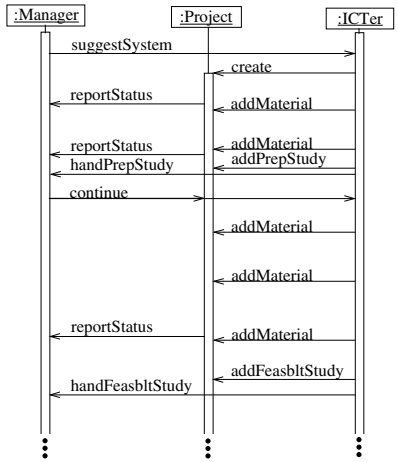
whole regions can be indicated, selcted for
rather general handling:

- for possible interruption

- for expansion of the (inner) activity handling,
involving various collections of inputs/outputs

sequence diagram (sd):

basically, any sequence diagram consists of a
nested frame containing one or more (usually
rather many) sd-lifeline fragments, like eg. the
following sdll-fragment



any sdll-fragment,
like any communication diagram as we shall
see,

restricts its behavioural representation to
the interaction steps

such as
sends, receives, signals, remote calls, triggers

in addition, any sdll-fragment,
like any communication diagram,

only presents one scenario of the interaction

ie. 1 example interaction sequence realization,
out of many possibilities of such realizations

time is implicitly present:
top down, as an invisible vertical axis

at the top of a sdll-fragment,
one finds the (human / non human) actors,
usually objects;

they are
the participants of the particular interaction

the vertical line / thick bar
under each participant is
its “lifeline”
being thick when (/ where) the participant is
actively available for the interaction

many technical details exist
about
synchronous vs asynchronous
time dependency
kind of communication (message, trigger)
initiating vs resulting

sdll-fragments are often used as
specification of the various use case scenario's

so they are suited for
modelling example interactions
partly occurring outside software system

this is another indication for the more general
suitability of OO / UML for
modelling the world outside software system

for instance,

there is nothing against incorporating
in an sdll-fragment
direct communication between actors outside software system

(in above sdll-fragment:
from ICTer to Manager)

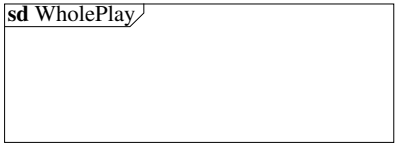
now we come to the actual sequence diagram:

it still is about interaction, expressed on the basis of lifelines,

but instead of specifying one scenario,
it gives as much scenario's as wanted

to that aim, an sd is composed from sd-fragments,
like a (main) program is composed from structured programming statements:

the main sd-fragment is called "frame":

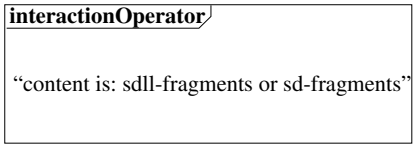


inside one either finds one or more sdll-fragments or further sd-fragments

sd-fragments inside a frame are either nested or non-overlapping

they cover a subset of the relevant lifelines, visualized by graphically containing parts of them, corresponding to a time interval

an sd-fragment looks as follows:



the interaction operator can be:

alt: alternatives in separate compartments,
excluding one another

opt: content is an option

loop: content is to be repeated

ref: has a name as content,
refers to an **sd** of that name
comparable to procedure call

par: parallel threads in separate compartments

these present the basic structuring facilities,

similar to

if then else
if then
while do
call

that concatenation is missing here, is compensated

by the "sequencing" of the sd itself

but some others are also nice to have:

break: as alternative to remainder of enclosing

seq: so-called weak sequencing, only the ordering per lifeline is relevant

strict: so-called strict sequencing, all ordering is relevant

neg: negative, expressing what is forbidden

critical: critical region, atomic execution

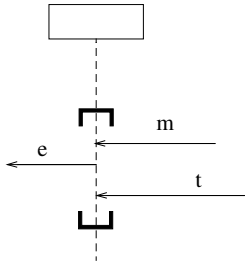
ignore signals: apart from the signals indicated

consider signals: only for the signals indicated

assert: explicitly specifies the only valid continuations

furthermore:

"coregion" in a lifeline
allows any order for the signals there
between the thick brackets, vertically lined:



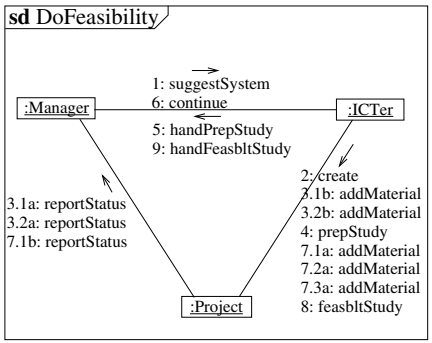
this is shorthand for a **par** fragment

communication diagram:

the new name for a collaboration diagram

it only covers simple sd's, without further fragmenting

the notation places the essential part of the diagram inside an **sd** frame:



note:

that essentail part is

the old UML 1.4 collaboration diagram

compared to the new collaboration diagram,
the essential part
of the communication diagram
annotates the links of a (new) UML 2.0
collaboration diagram with:

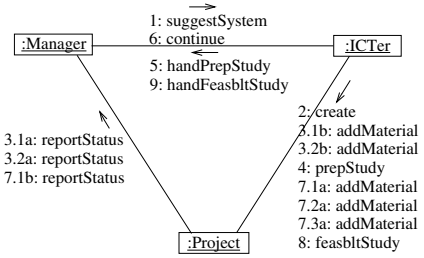
the annotation is as follows:

the signals being transmitted over these links

the signals have a sequence numbering according to their chronological ordering

moreover, the signals are grouped per transmission direction

essental part of collaboration diagram:



such essential part is claimed to be
semantically equivalent to one sdll-fragment
(not completely true!)

instead of a(n implicit) time axis, it orders the communication by enumeration

addition of symbol “a” or “b” to such order number, refers to parallel subthreads

interaction overview diagram:

new diagram, combining

- activity diagram structure (without swim lanes)
- each activity is replaced by an sd whose life-lines correspond to these swim lanes

ie. all “activities” are formulated in terms of communication/interaction only

interaction overview diagram is used for “complicated” sd’s with many fragments:

it gives these sd’s a more activity-diagram-like presentation

translation of sd to interaction overview:

alt, **opt**, **break** are translated by pair of decision / merge

par is translated by pair of fork / join

loop is translated as (visual) cycle

where every branching / connecting is properly nested

the other fragments refer to
exactly one continuation

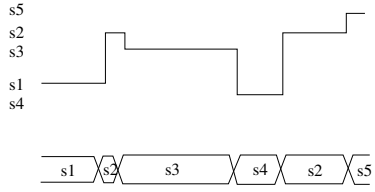
timing diagram

new diagram, combining

sequence diagram and explicit time
and instead of each lifeline:
a scenario of (local) state change sequences

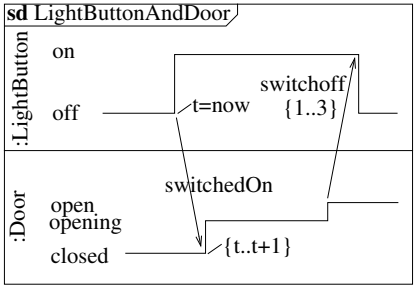
to that aim, a sequence diagram has a horizontal time axis - from left to right - instead of a vertical time axis

a state change sequence can have two appearances:



sends and receives can then be explicitly coupled with state changes

often such sends and receives are visualized by a somewhat oblique arrow, annotated by time:



note: sd’s allow
many interaction scenario’s in one diagram,
but only one state change sequence on a lifeline