

More on UML Class Models, Interaction Diagrams, State & Activity Diagrams

N.B. See module web-page for accompanying link to text book diagrams.

More on UML Class Models, Interaction Diagrams, State & Activity Diagrams	1
6. More on Class Models	2
6.1 More about associations	2
<i>Recall of • 5.2 Associations & 5.3 Attributes and operations</i>	2
6.1.1 Aggregation and composition [Figures 6.1, 6.2]	3
6.1.2 Roles [Figure 6.3]	3
6.1.3 Navigability [Figures 6.4, 6.5]	4
6.1.4 Qualified associations [Figures 6.6, 6.7, 6.8]	5
6.1.5 Derived associations [Figure 6.9]	5
6.1.6 Constraints [Figures 6.10, 6.11]	6
Panel 6.1 OCL, The Object Constraint Language, OCL - Omitted	6
6.1.7 Association classes [Figures 6.12, 6.13]	7
6.2 More about classes	8
Note on Stereotypes (Panel 6.2)	8
3 variants on idea of a class i.e. 3 ways of classifying objects	8
6.2.1 Interfaces [Figures 6.14, 6.15]	9
6.2.2 Abstract classes	10
Note on “Properties and tagged values” (Panel 6.3)	11
6.3 Parameterized classes [Figure 6.16]	12
6.4 Dependency [covered already]	12
6.5 Components and packages [see later]	12
6.6 Visibility, protection [see later; note symbols]	12
10. More on Interaction Diagrams	13
10.1 Generic interaction diagrams	13
10.1.1 Conditional behaviour [Figures 10.1, 10.2]	13
10.1.2 Iteration [Figures 10.3, 10.4]	13
10.2 Concurrency	14
10.2.1 Modeling several threads of control [Figure 10.5, 10.6]	14
12. More on State & Activity Diagrams	16
12.1 Other kinds of events	16
12.2 Other kinds of actions [Figure 12.1]	16
12.3 Looking inside states [Figures 12.2, 12.3]	17
12.4 Concurrency inside states [Figure 12.4]	17

6. More on Class Models

6.1 More about associations

Recall of • 5.2 Associations & 5.3 Attributes and operations

- We saw that classes correspond to nouns and, similarly, associations correspond to verbs.

- Definition: Class A and class B are associated if some object of class A *has to know about* some object of class B (or subclasses of B for which substitutivity holds), that is if one of following holds:

- An object of class A sends a message to an object of class B
- An object of class A creates an object of class B
- An object of class A has an attribute whose values are objects of class B or collections of objects of class B
- Object of class A receives message with object of class B as argument

- Note (see section 6.4 of text): Recall that "A depends on B if a change in B may force a change in A. Notice the difference between a dependency between two classes and an association between them. An association between two classes represents the fact that objects of these classes are associated. A dependency is between the classes themselves, not between the objects of those classes ... for example, a class always depends on a class from which it inherits." The UML terminology can be a bit confusing (dependency, relationship, association) and we will try to make it clear in the rest of this part of the notes.

6.1.1 Aggregation and composition [Figures 6.1, 6.2]

Note: In this and following sections, the summary notes on text figures appear hand-written in the lecture overheads.

Figure 6.1:

Aggregation & composition: Both record that an object of one class is part of an object of another.

Name of the association is "is a part of" but not essential to include it.

Diamond is at the "whole" side.

An object could be part of several objects at same time.

Figure 6.2:

Composition is a special kind of aggregation, which does impose restrictions.

The "whole" "strongly owns its parts".

If a whole object is deleted or copied then so are its parts.

Multiplicity (near whole) must be "1" or "0..1", in general. A part cannot be part of more than one whole by composition.

In above example (board game), each square is part of exactly one board. Here it makes sense to delete/copy each square as the board is deleted/copied.

6.1.2 Roles [Figure 6.3]

Figure 6.3:

Often one naturally reads an association both ways e.g. "is taking" and "is taken by".

May be more readable to have separate names for the roles the objects take in an association.

Role of Director of Studies is "DoS" (person who directs)

Role of student is "directee" (person being directed)

6.1.3 Navigability [Figures 6.4, 6.5]

Figure 6.4:

There are some (unspecified) number of student objects associated with "module"

6 objects of class "module" are associated with the "student" object

Navigability: Should a "student" object be able to send a message to its associated "module" objects, or the other way round, or both ways?

Figure 6.5:

This arrow-head indicates that a "module" object knows about "student" and can send messages to "student".

Possible use: "module" retrieves a list of student names by sending a message to each associated "student" object.

Point of caution: If class A "knows about" class B then we cannot reuse A without B.

Ambiguity: If navigability arrows are not shown does it mean "non-navigability" or "navigability not specified"? In text book, means the latter.

6.1.4 Qualified associations [Figures 6.6, 6.7, 6.8]

Figure 6.6:

"Qualified" associations -> means of providing fine detail

"Plain" = "unqualified"

Board game is "noughts & crosses"

Fact that square is part of board by decomposition is not shown here (but see Fig 6.8)

Figure 6.7:

Want to capture that the 9 squares are found by giving the 9 possible pairs values to attributes "row" and "column".

The "1" next to Square specifies that if we take a board object, call it "b", and specify values for both "row" and "column", then there is exactly one "square" object associated with "b".

Diagram does not specify which class "row" and "column" belong to. They could belong to "board" though formally they are attributes of the link.

Figure 6.8:

Here, we are just including the additional information that this particular association is a composition.

6.1.5 Derived associations [Figure 6.9]

Figure 6.9:

Question: Do we always need to show an association when its existence can be deduced from something else on the diagram?

Answer: We may choose to show the implied association or not. A third option in UML is to show that association as a derived association by putting a "slash" in front of its name.

The associations "is taking" and "teaches course" are known or given. What we are asking is whether we really need to show "teaches student".

Note: The solid black triangles (◻) have nothing to do with derived associations. They may be used for any association name to indicate the direction of the association described by the name.

6.1.6 Constraints [Figures 6.10, 6.11]

Figure 6.10:

Constraint = Condition to be satisfied by any correct implementation of a design.

Example 1: Use of a constraint to express a "class invariant":-

```
{self.noOfStudents > 50 implies (not(self.room=3317))~
```

[Above constraint example is written in OCL (object constraint language) but could well be in English or other 'notation']

In the diagram each "Copy" object is supposed to represent either a copy of a book or a copy of a journal. The diagram does not rule out (*which, however, it should*) that a copy is associated with both a book and a journal.

Example 2 (of a constraint) is to indicate that there's an "exclusive or" between two associations - this is shown in Figure 6.11 where the deficiency in Figure 6.10 is remedied.

Figure 6.11:

This concludes Example 2 of a constraint use. The constraint "{xor}" and associated "dashed line" depict the required "exclusive or".

"xor" is a predefined constant & is part of UML.

Caution: Text warns that constraints should be used sparingly, otherwise

- get too complex
- hamper maintenance & re-use

Panel 6.1 OCL, The Object Constraint Language, OCL - Omitted

Note: We will not cover "OCL" beyond what we have in examples in the notes.

6.1.7 Association classes [Figures 6.12, 6.13]

Figure 6.12:

Question: Where (in the example shown) should the system record the student's mark in this module? The point at issue is that the marks are really connected with the pair {Student, Module}.

Solution 1: Treat the association between 'Module' and 'Student' as a class. Such a class is called an 'association class' as it is both a class and an association.

In the figure, the association and class "is taking" must have the same name as they are the same thing. This runs against our general notion that class and association correspond to noun and verb, respectively.

Figure 6.13:

Solution 2 (of question posed under Figure 6.12): Invent a new class (say, 'Mark') and associate it with both 'Student' and 'Module' in the standard way.

The class 'Mark' will probably have operations (methods) as well as attribute(s) as will the association class (of Figure 6.12) (e.g. getMark).

6.2 More about classes

Note on Stereotypes (Panel 6.2)

- In UML stereotyping is a way of attaching extra classifications to model items.
- Stereotype is placed close to corresponding model element
- Some stereotypes are pre-defined, for example
 <<interface>>, <<use>>, <<type>>, <<implementation class>>
- Also possible to define one's own stereotypes. For example, <<Lan>>, <<TCP/IP>>, <<persistent>>.
- In a project, the definition of non-predefined stereotypes must be documented in an agreed place and manner.

3 variants on idea of a class i.e. 3 ways of classifying objects

<i>VARIANT</i>	<i>How much info. about attributes, operations, implementation?</i>
<<interface>>	Specifies a list of operations which anything matching the interface must provide No implementations associated with the operations. Nothing specified about object <u>state</u> => has no attributes
<<type>>	Like an <<interface>> except that it can have <u>state</u> => has attributes as well as operations. No implementation
<<implementation class>>	Can realize a <<type>>. Defines the physical implementation of its operations and attributes

6.2.1 Interfaces [Figures 6.14, 6.15]

Figure 6.14 (and 6.15 partly):

An interface specifies some operations of some model element that are visible outside the element. In figure 'Stringifiable' is an interface - it is satisfied by anything (here, by 'Module') that understands the message 'stringify' and returns a string.

Two ways are presented for representing the fact that 'Module' matches (or realizes or supports) the interface 'Stringifiable'. These ways are

- dashed arrow from 'Module' to 'Stringifiable'
- small circle labeled "Stringifiable" together with the solid line connecting it to 'Module'.

Note: dashed arrow ('realization arrow') makes it clear that 'Module' depends on 'Stringifiable'. This is a weak form of inheritance.

Two ways are presented for representing the fact that 'Printer' depends on the interface 'Stringifiable' only. These ways are

- dashed arrow labeled with the stereotype <<use>>
- dashed arrow from 'Printer' to small circle labeled 'Stringifiable'

We are being given the information that 'Printer' does not care about any other feature of 'Module'. Nevertheless, there is an association between 'Printer' and 'Module', as it is 'Module' that realizes 'Stringifiable'.

Figure 6.15:

To be compared with Figure 6.14 - leaves out 'realization' arrow and the dependency arrow labeled <<use>>

Note: It is possible that 'Module' could have other interfaces, supported by other operations besides 'stringify'. On the other hand, 'Stringifiable' could be realized by others besides 'Module'.

6.2.2 Abstract classes

A class is abstract if, for at least one of its operations, no implementation is defined
=> cannot instantiate an abstract class.

Remark: An abstract class in which implementation is not defined for any operation, and in which there are no attributes, is effectively the same as an interface.

Text notes that abstract classes are often used in C++ like 'Java interfaces' are used in Java. Thus, a C++ abstract class could well form the code corresponding to a UML interface.

Note on “Properties and tagged values” (Panel 6.3)a) Properties

{abstract} is an example of a property and is depicted as illustrated (opposite), using brackets.

Just as objects have values for their attributes, model elements have values for their properties. For example, {isAbstract=true} or {abstract} for short

StaffMember
{abstract}
staffName
...
Calculatebonus
....

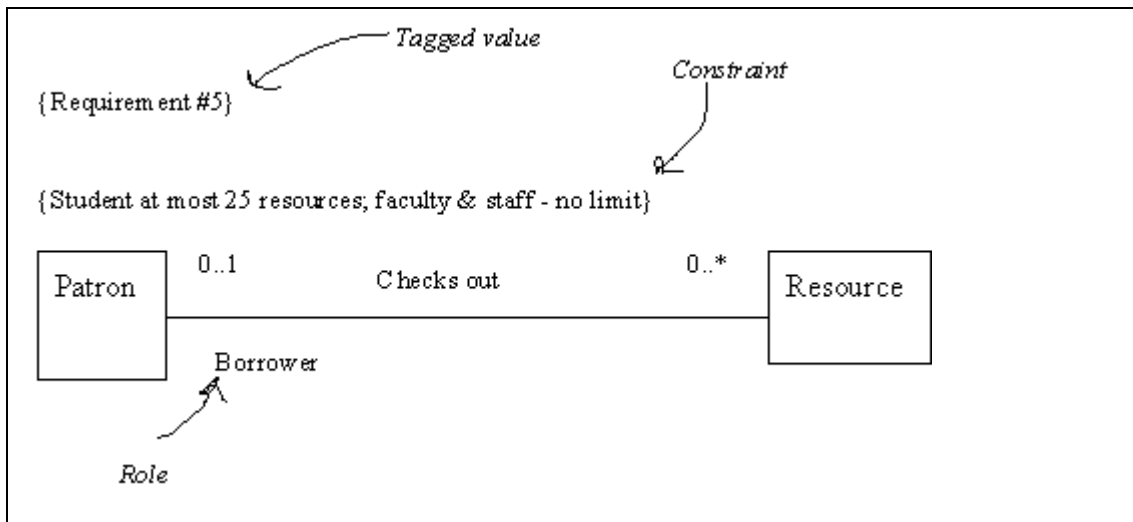
{isQuery} (property of an operation - means the operation does not affect state)

General form: {propertyName = value}

Note: properties have to do with the model rather than the implemented system. For example, there might not be an attribute "isQuery" in an implemented system.

b) Tagged values

One can define one's own 'tagged values' as well as using pre-defined UML properties. For example, one could have



See text (p. 89) for more 'tagged values' examples.

Note: Stereotypes provide a more powerful option than 'tagged' values. The latter should be used to attach a little more information to an element.

6.3 Parameterized classes [Figure 6.16]

Figure 6.16:

'Parameterized class' also called a 'template' - not actually a class at all!

Idea is to take advantage of common properties. For example, regardless of what kinds of things are in a list, we need to do standard things such as 'add to list' and 'delete from list'.

- Parameterized class (template) = List<T>
- Let C = class of students. Then Class of lists of students = List<C>
- An object of class List<C> is a particular list of students

Notation for a parameterized class is similar to the icon for a normal class with the addition of a small, dashed rectangular box containing the formal parameter(s).

The figure illustrates two equivalent ways of showing that a class is the result of giving an argument to a template (i.e. parameterized class). One way is to use an ordinary class icon whose name contains the parameterized class with argument (*here, List<Game> where Game is assumed to be a class. List<Game> is a class of lists of games*). The second way is to give a class an 'ordinary' name (*here, StudentList*) but also to show that it results from the template by connecting it to the template icon with a dependency arrow containing the stereotype <<bind>> and the name of a class (*here <<bind>>(Student)*).

Note: **genericity** is a term often used to describe construction of classes by use of templates.

Similarly, in some programming languages, one can define generic procedures, methods, etc

6.4 Dependency [covered already]

6.5 Components and packages [see later]

6.6 Visibility, protection [see later; note symbols]

We note here that UML has symbols to distinguish 'public' (+), 'protected' (#), package (~) and private (-).

10. More on Interaction Diagrams

10.1 Generic interaction diagrams

Note: Section 10.1 brings up the question - raised by some of our examples - as to whether and how to show all possible sequences of messages that occur or whether to show just one scenario. 'Generic' strictly requires all possible sequences to be shown but text is looser.

10.1.1 Conditional behaviour [Figures 10.1, 10.2]

Figure 10.1:

Remember: [...] is used to denote a condition (guard) in front of a message.

The 2 fragments depict 2 possible solution to the same problem. The *branching* solution (left) does not imply a time sequence - only one of the alternatives is followed during any one execution.

Figure 10.2:

General consideration: Avoid over-complex diagrams even if UML does provide syntax.

This diagram depicts how UML can show where different outcomes result from different conditional messages.

10.1.2 Iteration [Figures 10.3, 10.4]

Figure 10.3:

The * in "3.1:[i:=1..2]a()" indicates that message "a" is sent repeatedly. The element "[i:=1..2]" is an *iteration clause* telling how many times "a" is to be sent; such a clause is optional.

Various forms of iteration clause are possible. For example,

"[item not found]" would indicate that the corresponding message be sent repeatedly until item is found.

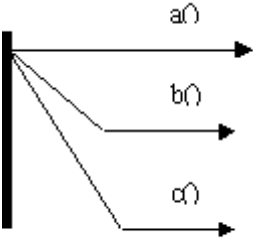
Figure 10.4:

Example of nested iteration.

10.2 Concurrency

10.2.1 Modeling several threads of control [Figure 10.5, 10.6]

Three ways a new "thread of execution" may start up are:

<p>1. Single thread splits into several</p>	<p>For example, an object which is computing sends 3 messages at same time</p>  <p>Notice there are no conditions on these branches. On a collaboration diagram, can use a numbering convention (e.g. 2.13. A, 2.13.B, etc) to distinguish simultaneous messages</p>
<p>2. An active object is one that owns its own thread of control</p>	<p>An active object sends a message while there is computation going on elsewhere Slightly heavier borders on active objects</p>
<p>3. Object sends an asynchronous message to another object</p>	<p>Can cause another object to become active without the sender having to stop computing</p>

Note: Avoid possible confusion between an active object and an object that becomes "activated" on receipt of a message. Text is a bit loose on this.

Figure 10.5:

Note corrections (should be as in text p.135):

- Delete row labeled "Flat".
- In row labeled "Asynchronous" change arrow to "->" and text to *"The sender does not lose control; it sends the message and may continue immediately. The recipient of the message may also become active [i.e. activated], if it wasn't already."*

Concurrency:

Up to section 10.2 in text we have considered case that

- one object (at most) is computing at a time
- if an object sends a message it awaits a response
- have a single processor (though can have multi-tasking with a single processor)

In section 10.2 are considering concurrent systems. For example,

- Distributed systems with several processors running in parallel
- Multi-threaded or multi-scheduling
- Reactive systems that receive inputs from environment in various ways

Figure 10.6:

This sequence diagram models the following scenario:

A student visits a web-site & chooses a selection of modules. Not necessarily immediately (asynchronous) the system does one of three things of which one is as in the diagram. Thus, if choices are not pre-approved for this student (but are legal), send an e-mail to this student's Director of Studies. System does not wait for a reply.

First ("choose...") & last ("email") messages are asynchronous, while second ("confirm...") is synchronous with its corresponding "return" displayed.

Note: Be advised that special issues arise in concurrent programming.

 12. More on State & Activity Diagrams

12.1 Other kinds of events

Brief note on types of events:

Event Type	Description
Call event	Most usual and already discussed An object receives a call for one of its operations Annotated by 'signature' of event (See below)
Signal event	An object receives a signal (or asynchronous communication) Annotated by 'signature' of event (See below)
Change event	Occurs when a condition becomes true Annotated by when keyword with a condition (usually Boolean).
Elapsed-time event	Caused by the passage of a designated period of time after a specified event (often the entry to current state) Annotated by after(time expression).

Note Basic syntax for 'call' or 'signal' event:

event-name parameter-list where parameter-list is optional.

12.2 Other kinds of actions [Figure 12.1]

Figure 12.1:

"Average" object waits for "update" messages from other objects. Such messages contain a parameter "val" to be added to current total.

"update" is put within the icon so that "entry" and "exit" events are not triggered by its receipt.

Two (external) events cause a self-transition ("report", "reset")

Action sequences contain "/" to separate actions.

12.3 Looking inside states [Figures 12.2, 12.3]

Figure 12.2:

Contains an example of an "elapse time" (or simply "time") event.

There is an implicit "completion" event corresponding to when the internal state diagram (figure 12.3) reaches its end state.

"include/activeDetail" means that 'active' is a compound state and that there is a detailed state diagram called 'activeDetail' nested within it.

Figure 12.3:

This example is drawn from the case study in Chapter 17 of text book. In that case study the first, second and third sub-states represent "waiting to join a queue", "in queue, waiting for service" and "being served, waiting for service to be completed", respectively.

12.4 Concurrency inside states [Figure 12.4]

Figure 12.4:

Diagrams (a) and (b) are equivalent ways of expressing the same thing

In (a): - At entry, start states of all (both) regions are reached at the same time.

 - Transitions leaving only "fire" when all regions have reached end state

In (b), use of "synchronization bars" in a state diagram is depicted.