
Here, we review Chapter 6 of text (More on Class Models)

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

The detailed sub-sections of Chapter 6 are

6.1 More about associations

6.1.1 Aggregation and composition [Figures 6.1, 6.2]

6.1.2 Roles [Figure 6.3]

6.1.3 Navigability [Figures 6.4, 6.5]

6.1.4 Qualified associations [Figures 6.6, 6.7, 6.8]

6.1.5 Derived associations [Figure 6.9]

6.1.6 Constraints [Figures 6.10, 6.11]

Panel 6.1 OCL, The Object Constraint Language

[Will skip this panel but will come back to OCL later]

6.1.7 Association classes [Figures 6.12, 6.13]

6.2 More about classes

Panel 6.2 Stereotypes

6.2.1 Interfaces [Figures 6.14, 6.15]

[Will come back to Interfaces with examples later]

6.2.2 Abstract classes

Panel 6.3 Properties and tagged values

6.3 Parameterized classes [Figure 6.16]

6.4 Dependency [*mentioned earlier*]

6.5 Components and packages [*covered already*]

6.6 Visibility, protection

Summary notes on text figures (hand-written in 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.

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)

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.

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.

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.

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

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).

Notes:1) Stereotypes

- In UML 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.

2) Three variants on the idea of a class - 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

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'.

Notes:1) 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.

2) 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

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

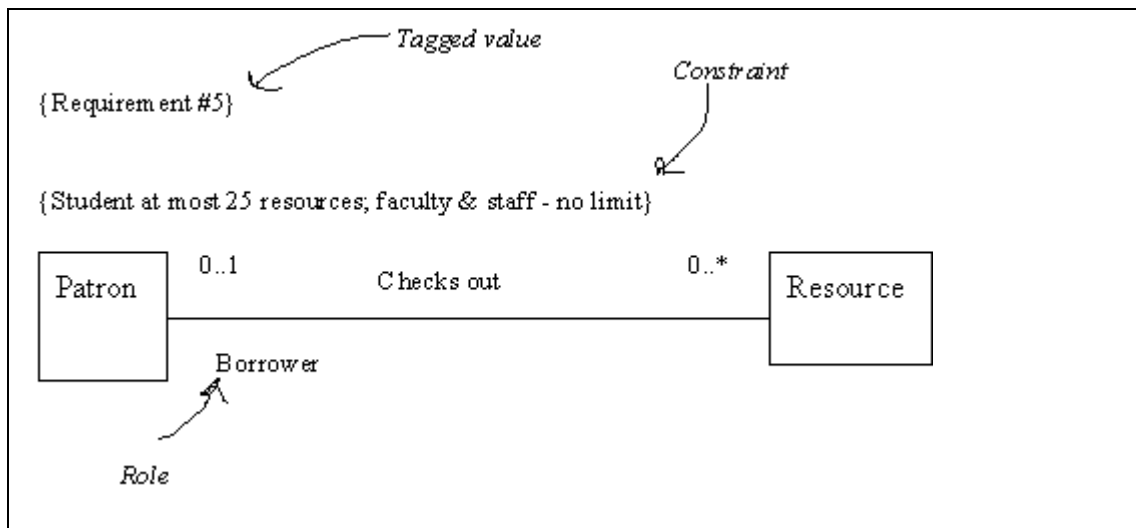
General form: {propertyName = value}

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.

StaffMember { abstract }
staffName ...
Calculatebonus

3) 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.

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)*).

Notes:

1) "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

3) Text has sub-sections on

6.4 Dependency – see earlier notes

6.5 Components and Packages – see later notes

6.6 Visibility, Protection: We note that UML has the following symbols to distinguish

public: +

protected: #

package: ~

private: -

members of a class.