

CA314 – Object Oriented Analysis & Design - 8

File name: CA314_Section_08_Ver01

Author: L Tuohey

No. of pages: 20

Table of Contents

8. UML Class Models - More (see ref 1, Chap. 6)	3
8.1 More about associations.....	3
8.1.1 Aggregation and composition	3
8.1.2 Roles	4
8.1.3 Navigability.....	4
8.1.4 Qualified associations	5
8.1.5 Derived associations	6
8.1.6 Constraints	7
8.1.7 Association classes.....	9
8.2 More about classes.....	10
8.2.1 Stereotypes.....	10
8.2.2 Three variants on the idea of a class	11
8.2.3 UML Interface notation	11
8.2.4 Notes on some general design issues, especially Interfaces	13
8.2.4.1 Context.....	13
8.2.4.1.1 What are good systems like?	13
8.2.4.1.2 Encapsulation/Low coupling	13
8.2.4.1.3 Abstraction – Good i.e. High Cohesion.....	14
8.2.4.1.4 Outline of issues of architecture & component-based design.....	14
8.2.4.2 A Java Interface example.....	15
8.2.4.2.1 Background.....	15
8.2.4.2.2 Implementing interfaces in Java - Example.....	16
8.2.4.3 Recap of benefits of modularity with well-defined interfaces.....	18
8.2.5 Abstract classes.....	18
8.2.6 Properties and tagged values.....	18
8.3 Parameterized classes.....	19
8.4 Visibility, protection	20

8. UML Class Models - More (see ref 1, Chap. 6)

8.1 More about associations

8.1.1 Aggregation and composition

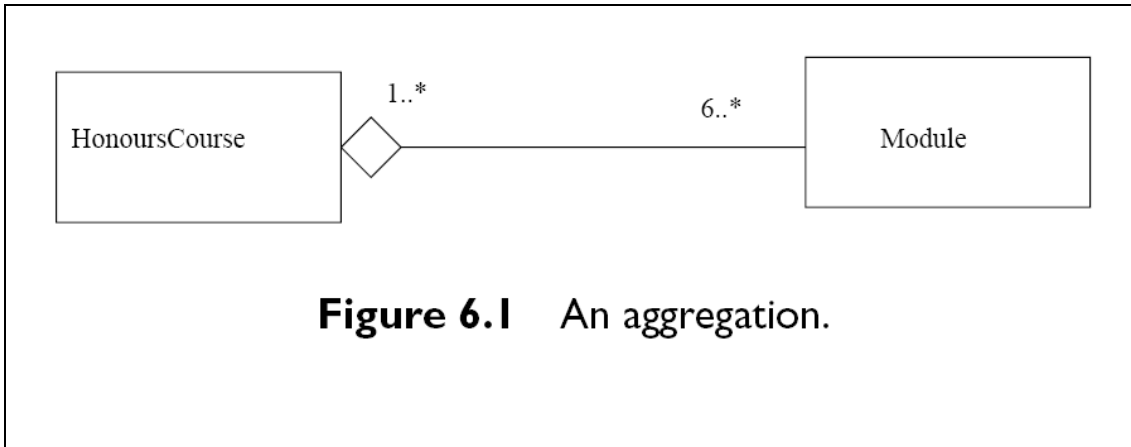


Figure 6.1 An aggregation.

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 it is not essential to include it.

Diamond is at the "whole" side.

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

Aggregation indicates the structural inclusion of objects of one class by another, and is usually implemented by a class having an attribute whose type is the included class. Roughly, one can think of the aggregator as the “whole” and the class aggregated as the “part”.

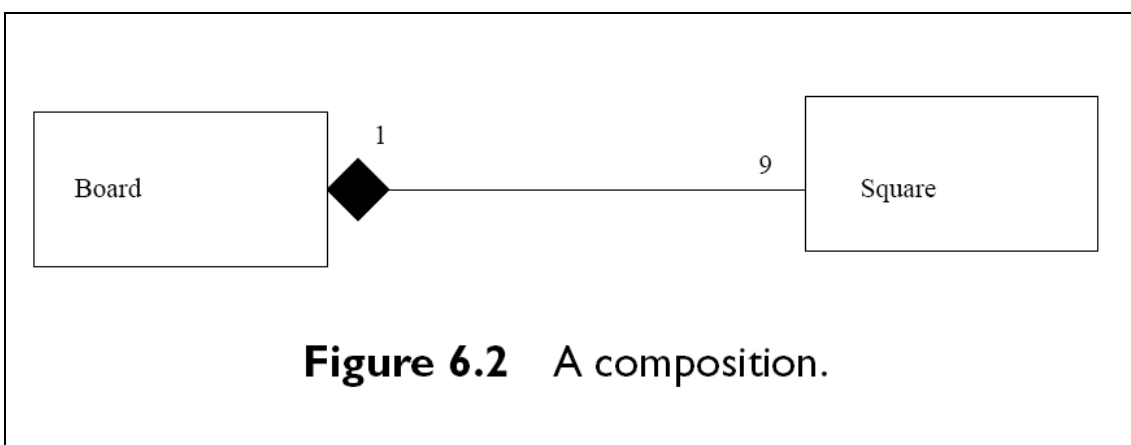


Figure 6.2 A composition.

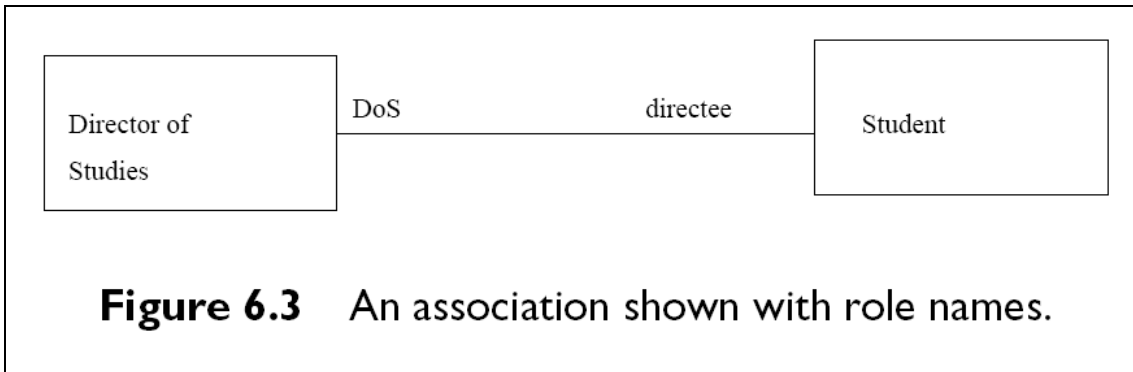
Composition is a special kind of aggregation, which does impose restrictions.

The "whole" "strongly owns its parts". In particular, if a whole object is deleted or copied then so are its parts.

In general, multiplicity (near whole) must be "1" or "0..1". 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.

8.1.2 Roles



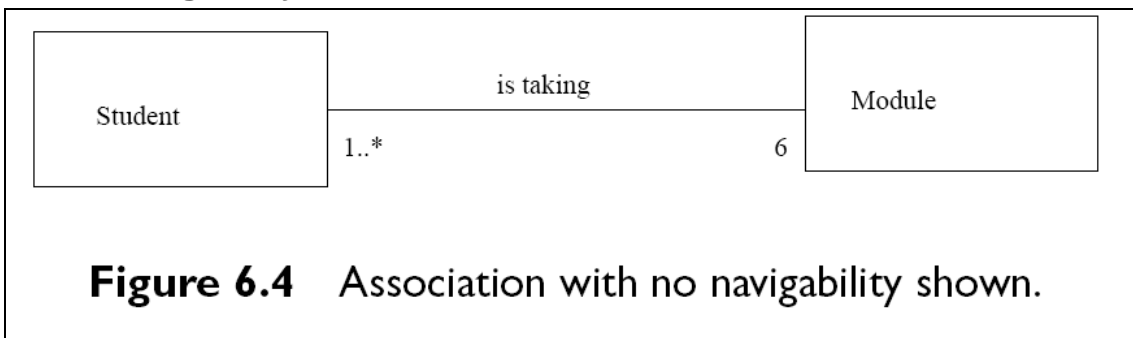
Often one naturally reads an association both ways e.g. "is taking" and "is taken by".

It may be more readable to have separate names for the roles the objects take in an association. For example, in the figure,

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

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

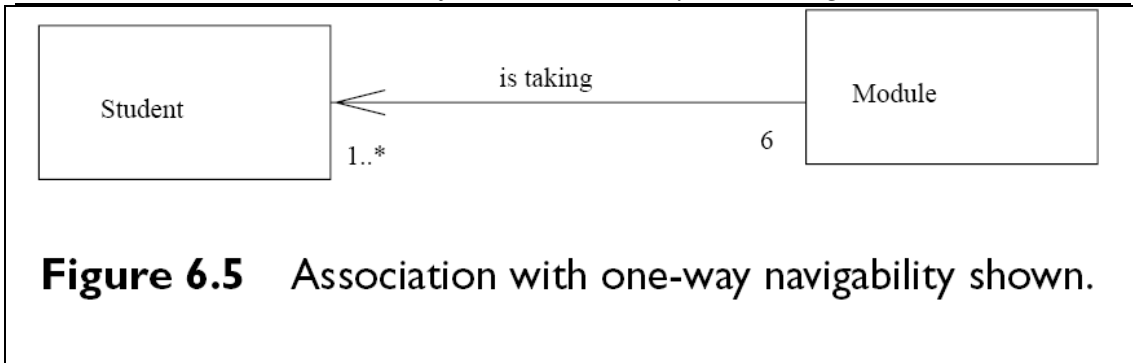
8.1.3 Navigability



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? More generally, should a "student" object know about its associated "module" objects, or the other way round, or both ways?



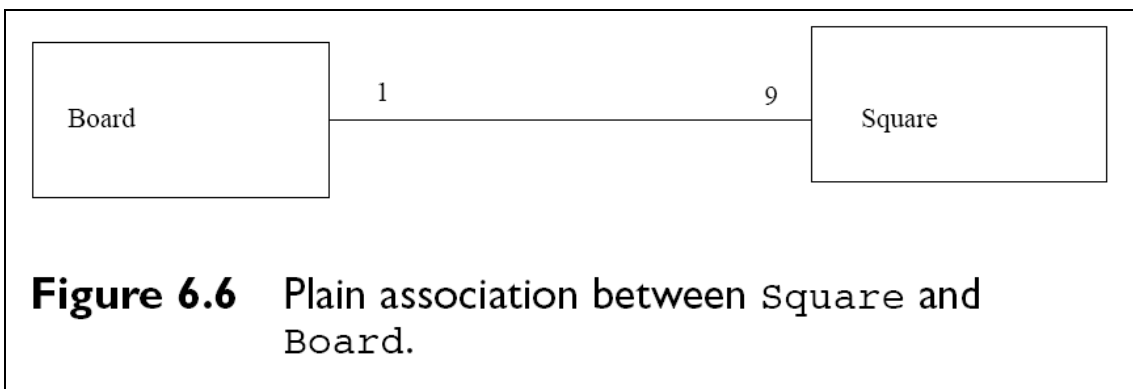
The arrow-head indicates that a "module" object knows about "student" and can send messages to "student".

A 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 Reference 1, the latter is meant.

8.1.4 Qualified associations

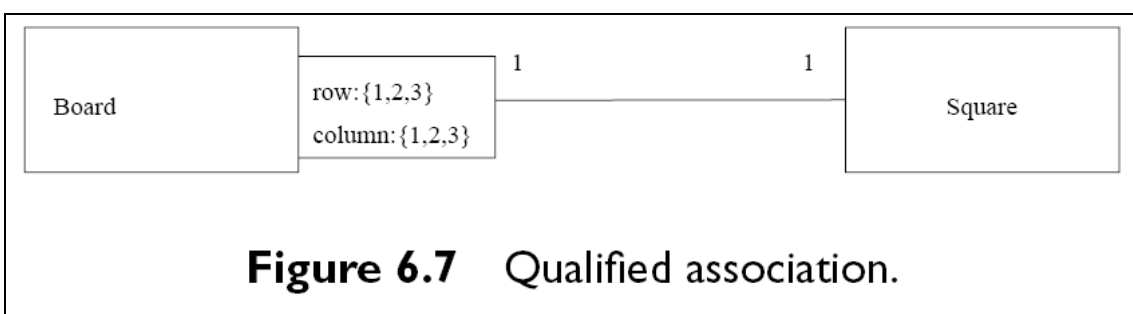


"Qualified" associations are a means of providing fine detail

"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)



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.

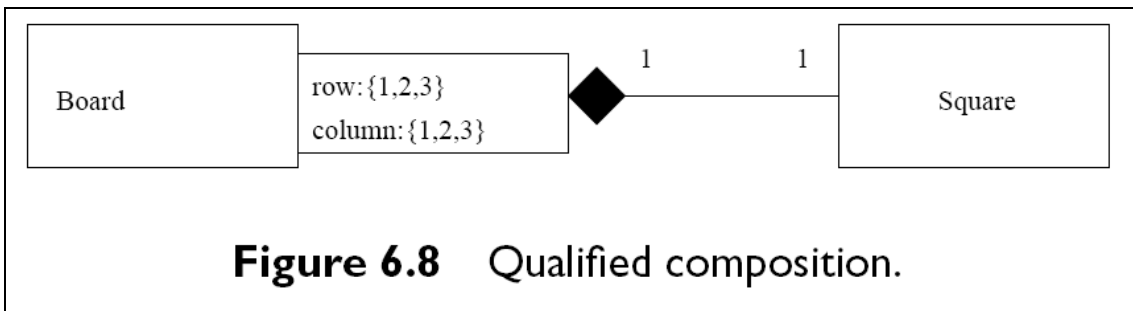


Figure 6.8 Qualified composition.

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

8.1.5 Derived associations

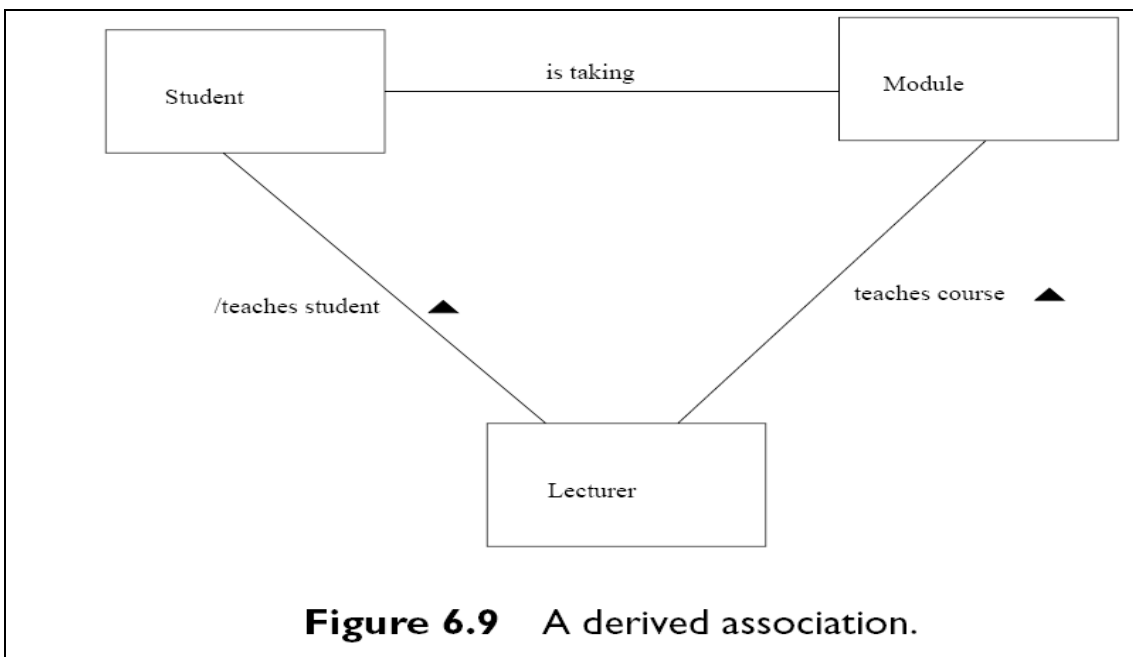


Figure 6.9 A derived association.

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 is 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.

8.1.6 Constraints

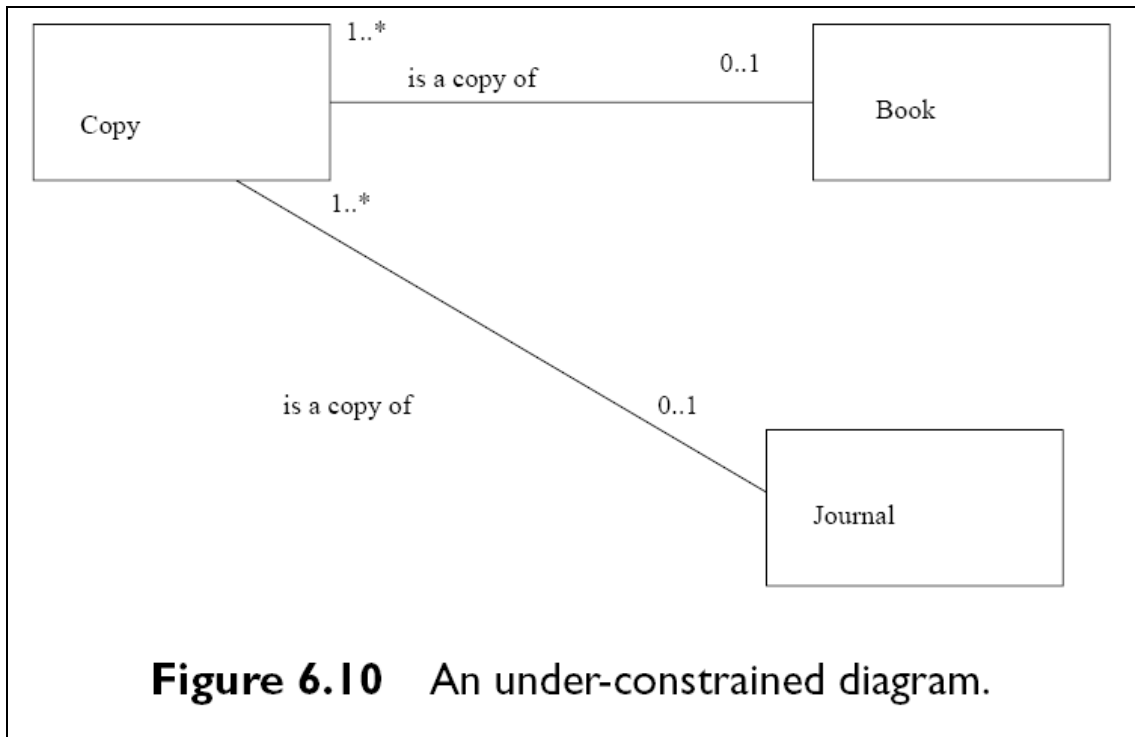
Definition: Constraint = Condition to be satisfied by any correct implementation of a design.

Example 1 (non-diagrammatic): Use of a constraint to express a "class invariant":-

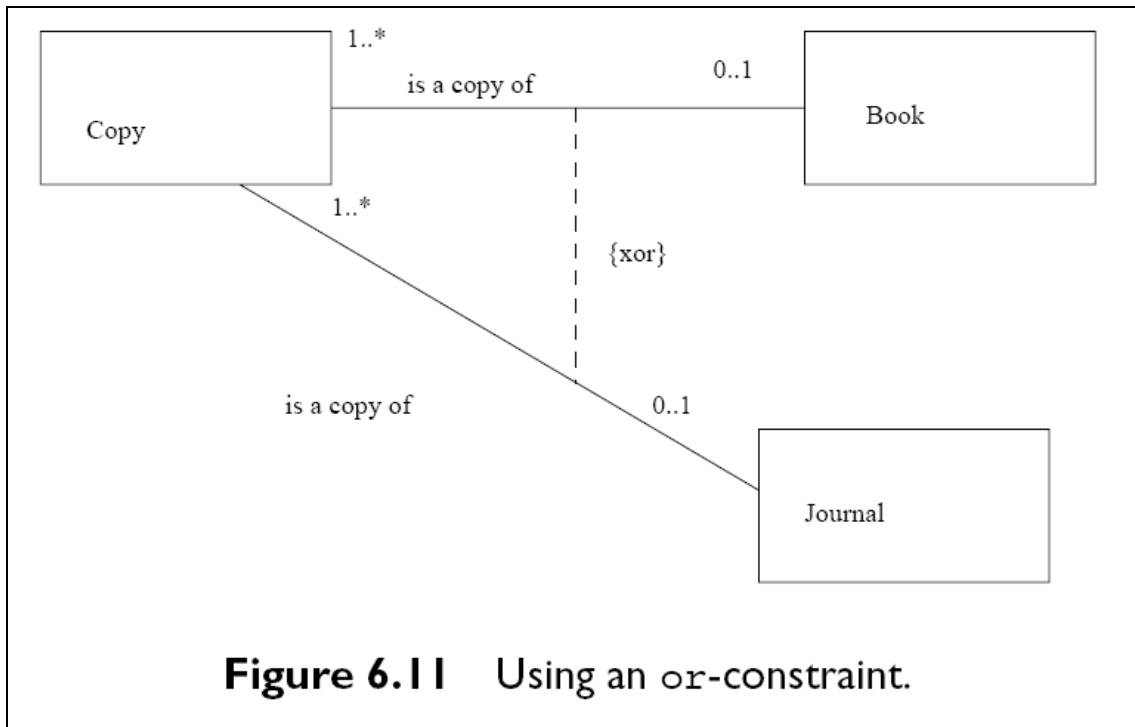
`{ self.noOfStudents > 50 implies (not(self.room=3317)) }`

This constraint example is written in OCL (object constraint language) but could well be in English or other 'notation'.

Example 2 (diagrammatic):



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. The defect of Figure 6.10 is remedied in the following figure:

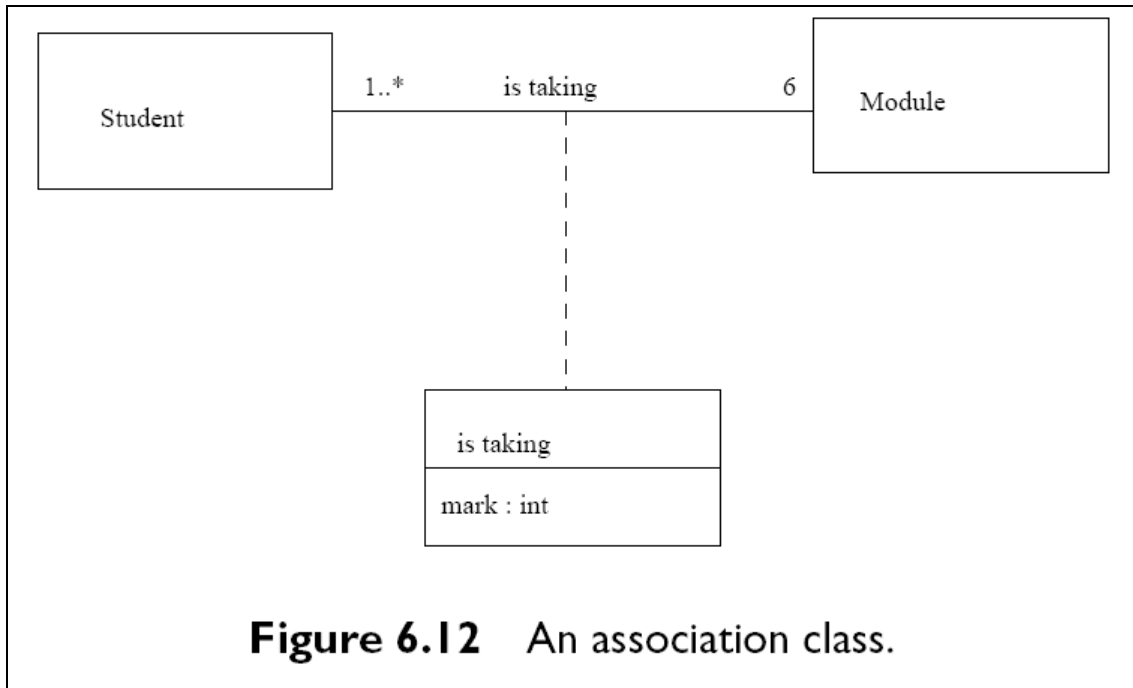


The constraint "{xor}" and associated "dashed line" depict the required "exclusive or". "{xor}" is a predefined constant & is part of UML.

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

- get too complex
- hamper maintenance & re-use

8.1.7 Association classes

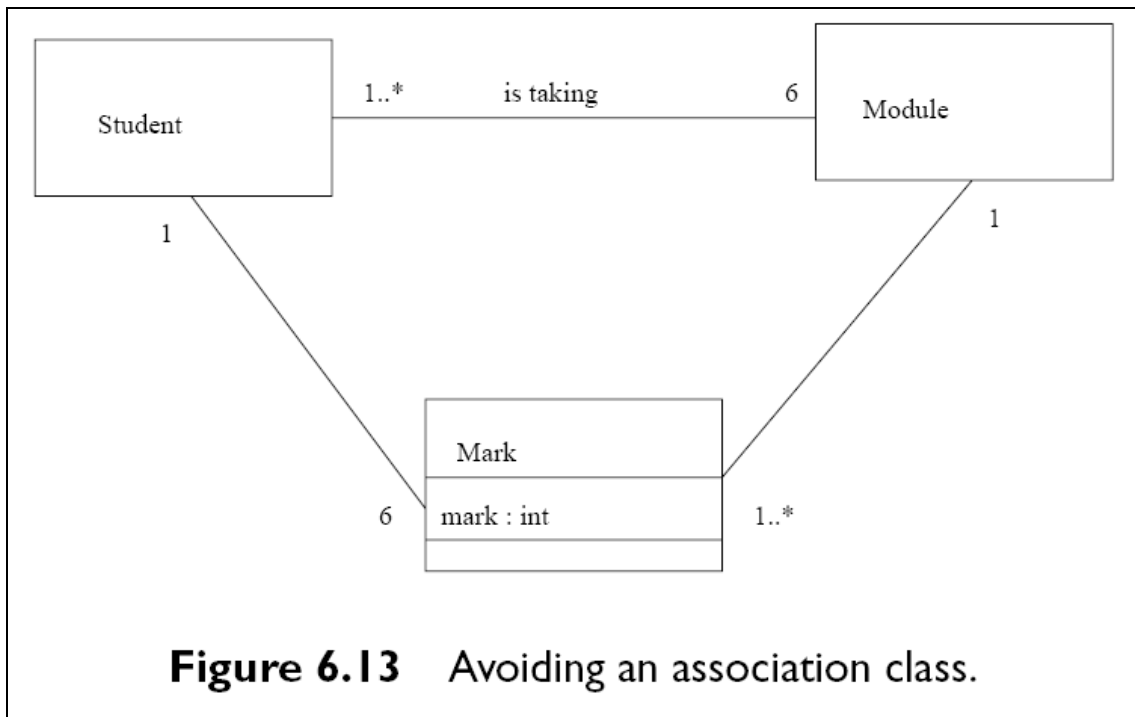


Question: Where, in Figure 6.12, should the system record the student's mark in this module? The point at issue is that the marks are really connected with the pair {Student, Module}.

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.

Solution 2: Invent a new class (say, 'Mark') and associate it with both 'Student' and 'Module' in the standard way. Thus,



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).

8.2 More about classes

8.2.1 Stereotypes

In UML use of stereotypes 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>>.

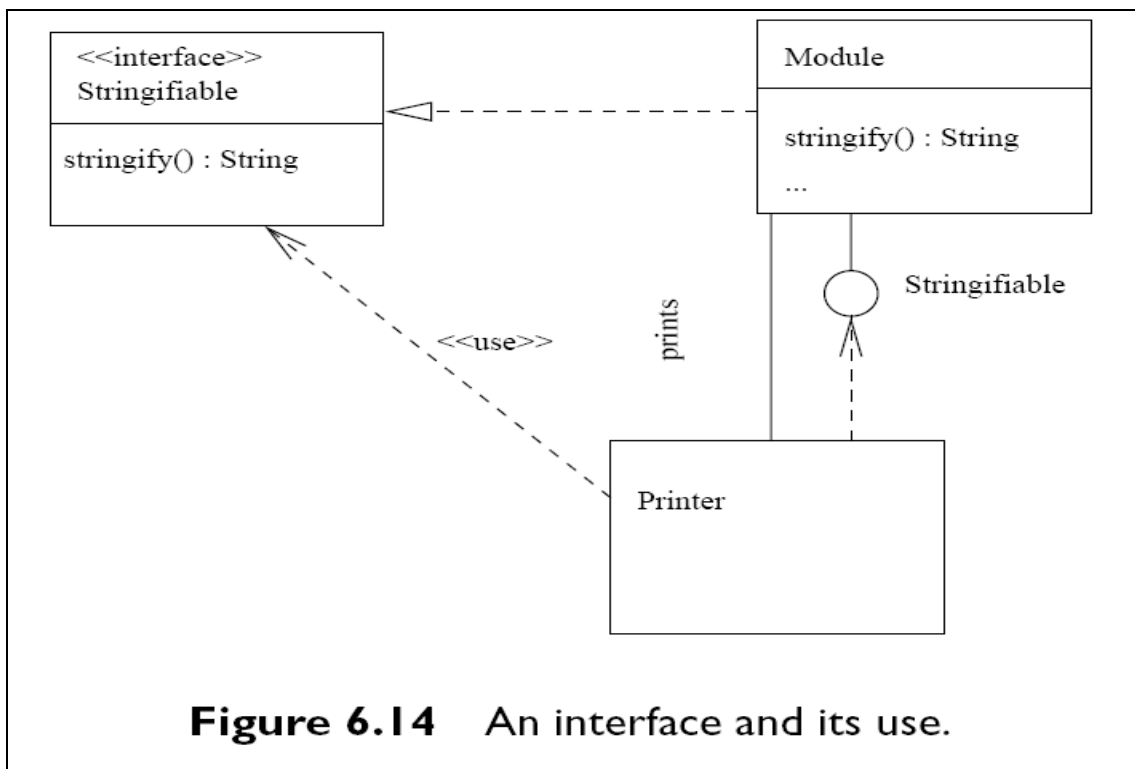
- In a project, the definition of non-predefined stereotypes must be documented in an agreed place and manner.

8.2.2 Three variants on the idea of a class

- 3 ways of classifying objects:

VARIANT	How much info. about attributes, operations, implementation?
<<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

8.2.3 UML Interface notation



An interface specifies some operations of some model element that are visible outside the element. In Figure 6.14, '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 (*note arrow head*) from 'Module' to 'Stringifiable'
- small circle labeled "Stringifiable" together with the solid line connecting it to 'Module'.

Note: The 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'.

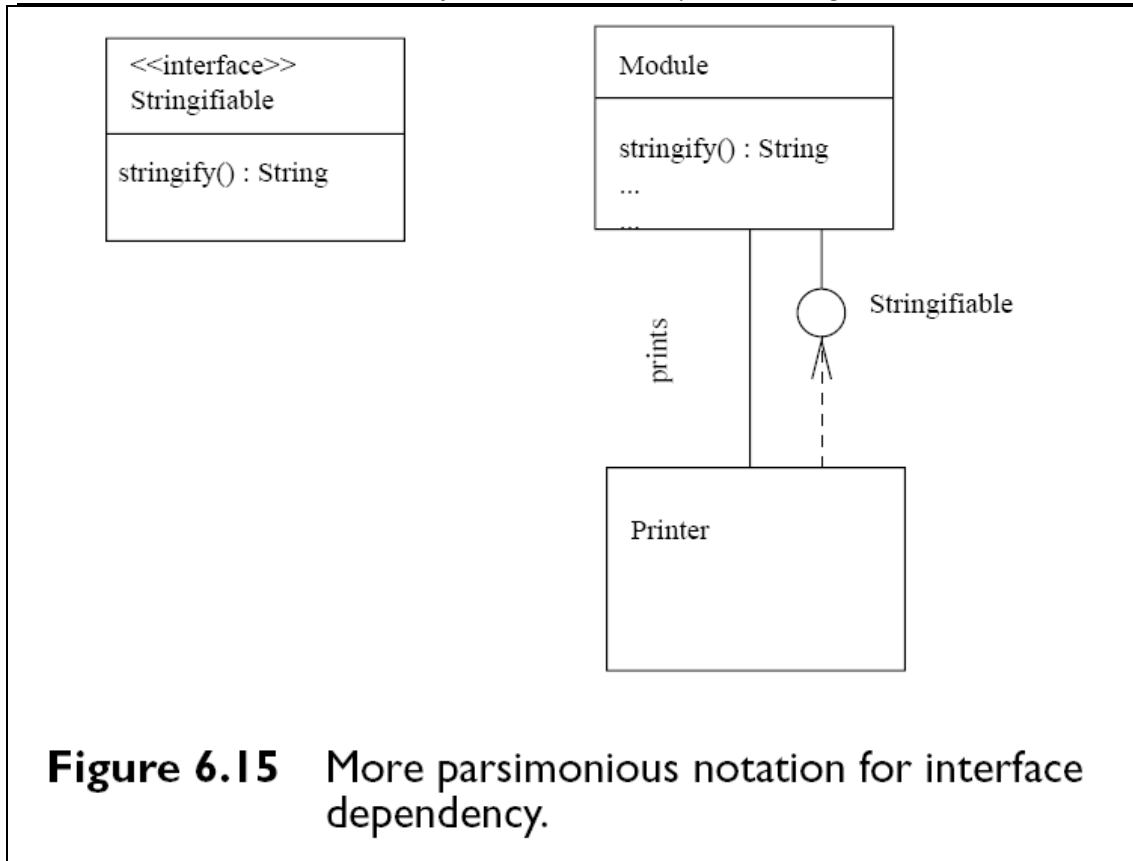


Figure 6.15 should be compared with Figure 6.14 - leaves out 'realization' arrow and the dependency arrow labeled <<use>>

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'.

8.2.4 Notes on some general design issues, especially Interfaces

8.2.4.1 Context

8.2.4.1.1 What are good systems like?

- A key point is the need to *localise* related material in design and code.
- Modularity is concerned with splitting of a system into manageable "chunks", with *interfaces* between them.

8.2.4.1.2 Encapsulation/Low coupling

- Desirable to "*Minimise the number of cases in which a change in one module necessitates a change to another module*". If we achieve this will have *low coupling*.
- Three pieces of information are pertinent when considering coupling:

A. Which modules are "clients" of, that is depend on, a given module? *This information identifies the "client" modules that may need to be changed or re-tested if the given module is changed.*

B. What changes inside a module may affect its "clients" (and thereby the rest of the system)? *If we know the assumptions "clients" of a given module may make about it we should be able to tell which changes to it may impact outside itself.* See our previous discussion on "Design by Contract" (section 3.1). Pertains to the module's **provided interface**.

C. What does the given module itself depend on? (**required interface**)

- Tools are available to help establish the dependencies mentioned above (*including UML, but there are also verification tools that measure in some sense the degree of coupling of a system*). Also, some design methods (e.g. UML) support definition of both provided and required interfaces.

8.2.4.1.3 Abstraction – Good i.e. High Cohesion

- "Cohesive" means we want a module to represent some intuitively related things rather than a collection of unconnected material. A non-cohesive module is bad for coupling also as one is likely to get more disparate modules depending on it.

- Given that a module has "cohesive responsibilities", the next thing is to provide it with an interface that "abstracts away from" implementation details. These are details that a "client" module neither needs to nor should know about.

- Terms "**abstraction**", "**information hiding**" and "**encapsulation**" are all used, sometimes with slightly different meanings. Reference 1 uses:

Abstraction: when a "client" does not need to know more than is in the interface

Encapsulation: when a "client" is not able to know more than is in the interface.

8.2.4.1.4 Outline of issues of architecture & component-based design

If a module has high cohesion and low coupling (*at least in terms of its required interface*) it should be a good candidate for **re-use** and/or **replacement** (*by a more advanced version of itself, say*). Ideally, it should be a "pluggable component", for re-use in similar software products or for replacement in later versions of current software.

How practical or feasible this use as a component is will depend also on the architecture of which the module is a part. Briefly, and roughly,

- Often, in the past, the architecture of a piece of software was considered in isolation from other pieces of software.

- But, it is more fruitful in many cases to consider a common architecture (architectural “**framework**”) for a range of products or, at the extreme, throughout a software development organisation.

A key benefit, one hopes, of such an architectural approach would be greater use of modules as pluggable components.

Notes:

1) We will touch on “pluggable components” again when presenting **UML support** for documenting such. However, be aware that the whole area of component-based software engineering is very wide.

2) Another aspect of reuse in SW development is in identifying and making use of **design patterns**.

8.2.4.2 A Java Interface example

8.2.4.2.1 Background

We are mainly concerned with design rather than programming in this module. Still, it is interesting to look at an example of how provided interfaces may be implemented in Java, say. The example is taken from "*Data structures and algorithms in JAVA*" by Goodrich and Tamassia.

As background, the following quotation repeats, to some extent, the previous notes but it is interesting to have other writers’ perspectives.

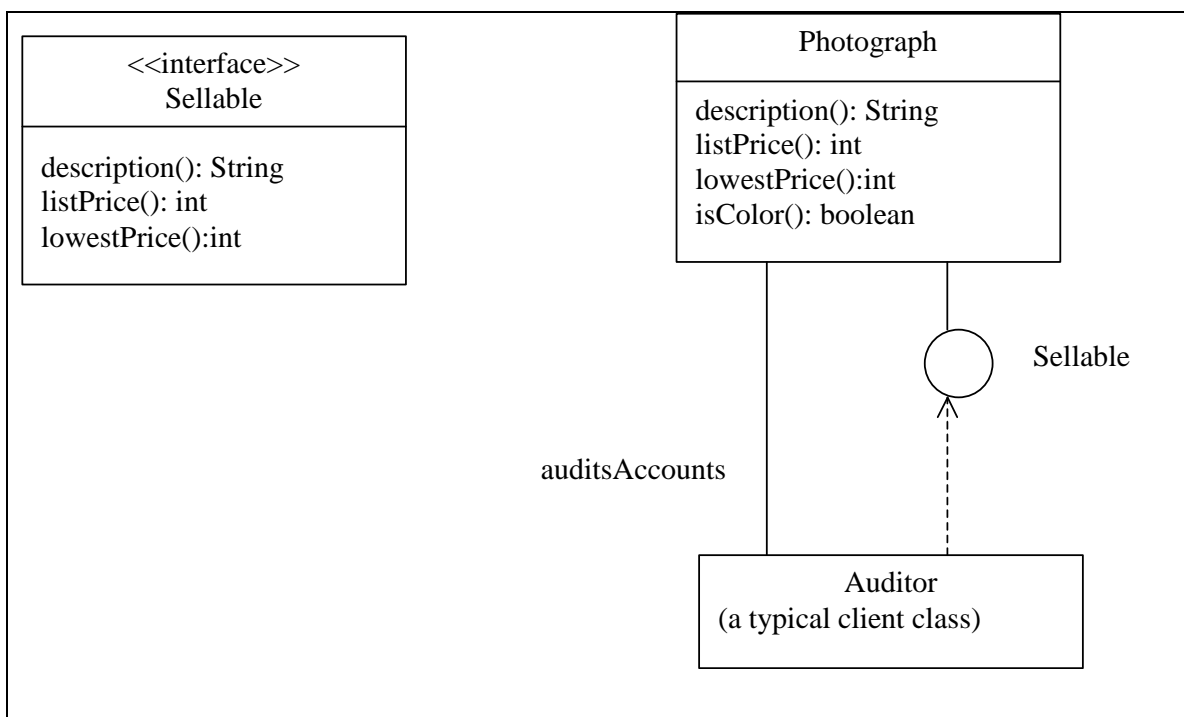
“In order for two objects to interact, they must "know" about the various messages that each will accept, that is, the methods each object supports. To enforce this "knowledge", the object-oriented paradigm asks that classes specify the **application programming interface** (API), or simply **interface**, that their objects present to other objects. In the **ADT-based** approach to data structures ..., an interface defining an ADT is specified as a type definition and a collection of methods for this type, with the arguments for each method being of specified types. This specification is, in turn, enforced by the compiler or run-time system, which requires that the types of parameters that are actually passed to methods rigidly conform with the type specified in the interface. This requirement is known as **strong typing**. Having to define interfaces and then having those definitions enforced by strong typing admittedly

places a burden on the programmer, but this burden is offset by the rewards it provides, for it enforces the encapsulation principle and often catches programming errors that would otherwise go unnoticed.”

8.2.4.2.2 Implementing interfaces in Java - Example

The main structural element in Java that enforces an API is the *interface*. An interface is a collection of method declarations with no data and no bodies. That is, the methods of an interface are always empty. When a class implements an interface, it must implement all the methods declared in the interface. In this way, interfaces enforce a kind of inheritance called *specification*, where we require that each method inherited be specified in full.

Suppose, for example, that we want create an inventory of antiques we own, categorized as objects of various types and with various properties. We might, for instance, wish to identify some of our objects as sellable, in which case they could implement the `Sellable` interface according to the UML diagram below, and afterwards in Java code fragments:



```
/** Interface for objects that can be sold. */
```

```
public interface Sellable{  
    /** description of the object */  
    public String description();  
  
    /** list price in cents */  
    public int listPrice();  
  
    /** lowest price in cents we will accept */  
    public int lowestPrice();}
```

We can then define a concrete class, `Photograph`, shown in the following code fragment, that implements the `Sellable` interface, indicating that we are willing to sell any of our `Photograph` objects: This class defines an object that implements each of the methods of the `Sellable` interface, as required. In addition, it adds a method, `isColor`, which is specialized for `Photograph` objects.

```
/** Class for photographs that can be sold. */  
public class Photograph implements Sellable{  
    private String descript; // description of this photo  
    private int price; // the price we are setting  
    private boolean color; // true if photo is in colour  
  
    public Photograph(String desc, int p, boolean c){//constructor  
        descript = desc;  
        price = p;  
        color = c;  
    }  
  
    public String description(){ return descript;}  
    public int listPrice(){ return price;}  
    public int lowestPrice(){ return price/2;}  
    public boolean isColor(){ return color;}  
}
```

8.2.4.3 Recap of benefits of modularity with well-defined interfaces

- Makes multi-person teams more productive
- Allows developers to focus on a particular part of the system, knowing the extent to which other parts can be ignored
- Should make "bugs" easier to find
- Helps in testing (especially integration and re-testing)
- Makes re-use more feasible
- The requirement that interfaces be clearly defined is an essential starting point. However, a more interesting question, which we have already looked at to some extent, perhaps is how to set about choosing modules with the "right" content and then ensuring that they have the "right things" in their interfaces.

8.2.5 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. 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.

8.2.6 Properties and tagged values

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
....

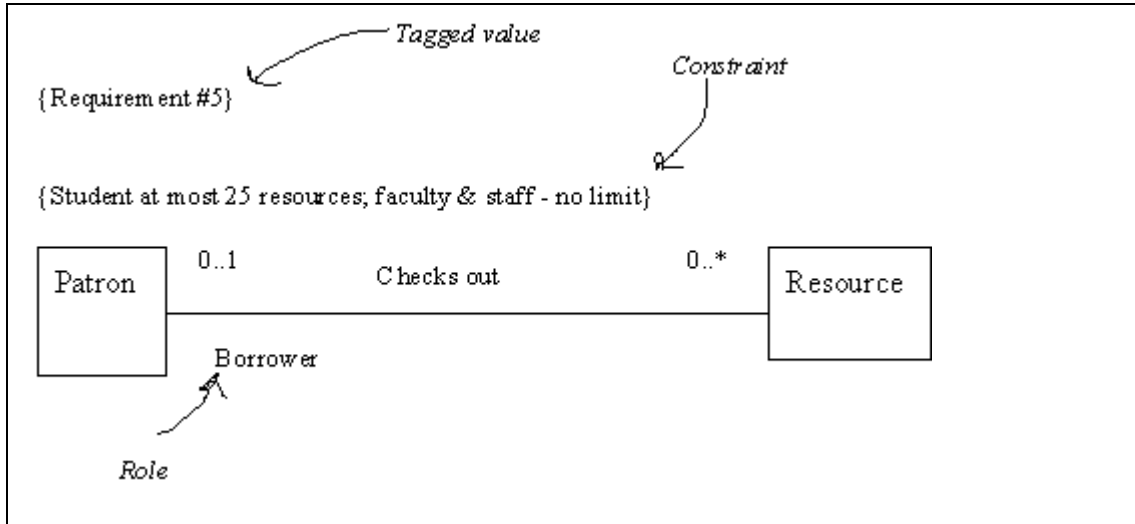
{isQuery} (property of an operation - means the operation does not affect state)

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.

Tagged values

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



See reference 1 (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.

8.3 Parameterized classes

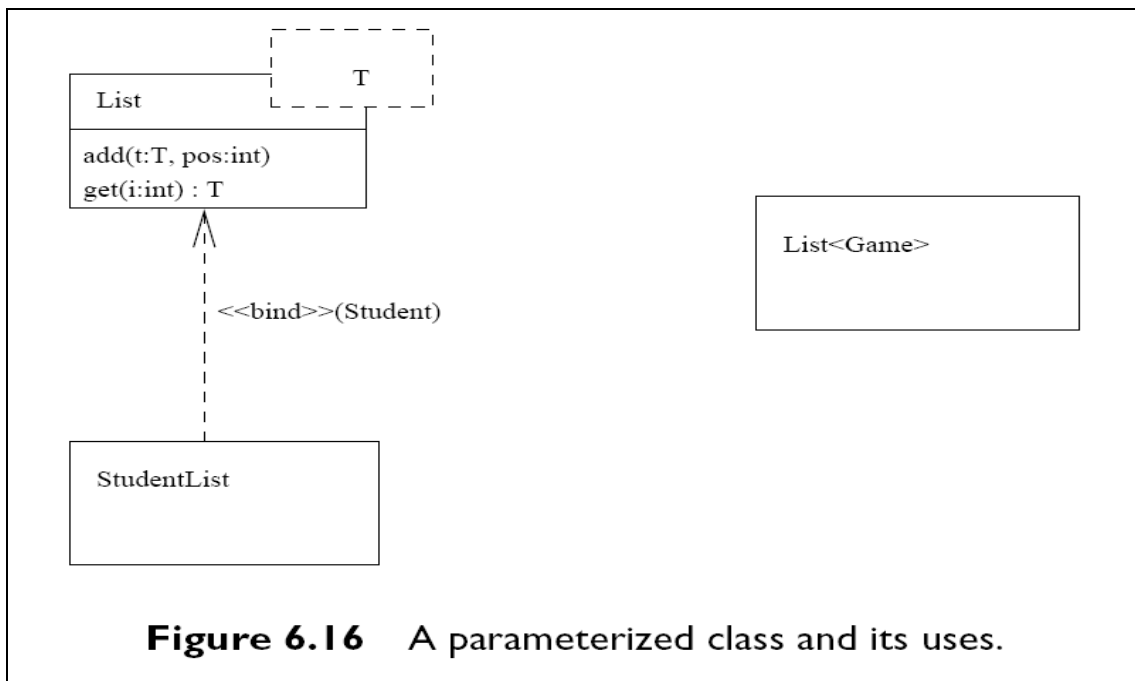


Figure 6.16 A parameterized class and its uses.

'Parameterized class' is 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).

Figure 6.16 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

8.4 Visibility, protection

We note that UML has the following symbols to distinguish

- | | |
|------------|---|
| public: | + |
| protected: | # |
| package: | ~ |
| private: | - |

members of a class.