Chapter 1

Overview of the course

Course: Advanced Computer Science Lab

Course Number: 320201

Time: Monday, Tuesday,15.30-19.30

Place: R3 Lecture Hall (15.30-17.00, Monday)
R2 Lecture Hall (15.30-17.00, Tuesday) CLAMV (Other times)

Instructor:
Dr. Holger Kenn, Tel: 3112,
E-mail: h.kenn@iu-bremen.de

Web page: http://www.faculty.iu-bremen.de/course/AdvCSLab/

Lecture Materials:

Stevens,Pooley: Using UML
Beck: Extreme Programming Explained
Gamma et al.: Design Patterns
Sommerville: Software Engineering
Jalote: An integrated approach to software engineering
Ghezzi et al.: Fundamentals of software eingineering
Szyperski: Component Software: Beyond object oriented programming

1.1 Introduction

1.1.1 What is a good system?

A good system is:

- useful and usable
- reliable
- flexible
- cheap
- available

The fundamental problem of software engineering is that a single programmer can only focus on a limited amount of issues at any given time.

In order to build software systems that go beyond this limit, we need to separate the sytem into modules. If modules often depend on eachother, we have nothing gained.

What we are looking for is a way to split up systems so that the coupling between the modules is as low as possible.

Good systems have low coupling.

Encapsulation results in low coupling. Coupling only occurs at well-defined places, the interfaces.

As long as the interface does not change, the module can be changed internally without affecting other modules.

Abstraction leads to high coupling.

Abstraction: The client of a module does not need to know more than what is specified in the interface. (A student is a person. A person has a name. So we can treat the student as a person without knowing that he's a student, e.g. ask for his name.)

Encapsulation: The client of a module cannot know more than what is specified in the interface. (Hiding the details so nobody can rely on them.)

1.1.2 Objects and OOP

Objects are things that can interavt via messages. An object can send a message to another object or receive a message from another object.

An Object has

state An object can store data in attributes of which each has a value. attributes can be constant or mutable. Normally, the attributes of an object cannot be changed during its lifetime but the value of these attributes can be changed.

behaviour An object can react to messages, e.g. by changing its state or by sending other messages. The code executed when an object receives a message is called a method.

identity Objects have an identity beyond the value of their attributes, i.e. there can be two different objects with the same attribute values. However, the same object may be adressed under different names.

Interfaces are the messages that objects can receive. Messages that are defined as public can be sent to the object from anywhere.

A Class describes a set of objects in the system with similar function.

Creating an object of a class is called instanciation. The object created is an instance of the class. A part of the system that creates objects is often called an object factory.

Organising objects in classes has the advantage that it helps to fulfil one of the principles of software engineering that is:

Only write code once!

Inheritance is the process of deriving one class from another class by adding attributes and adding or modifying methods. Modifying methods is called overriding. The derived class is called a subclass, the class derived from is called a superclass.

A subclass is an extended, more specialized version of its superclass. It contains the methods and attributes of the superclass but can contain more.

A subclass inherits from a superclass. The subclass is a specialisation of its superclass. The subclass is more specialized than its superclass. A superclass is a generalisation of a subclass. A subclass can also be called derived class. A superclass can also be called base class.

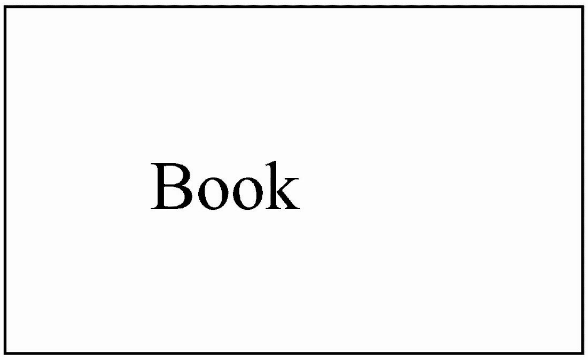
Polymorphism: an object that is polymorph can be used as an instance of several classes, e.g. a subclass can be used as if it's of the type of its superclass.

Dynamic binding: Although a subclass is used as if it's of the type of its superclass, sending methods to this object can result in different methods beeing called if the methors have been overridden in the subclass.

1.1.3 A simple case stuty

1.2 An introduction into UML

UML represents a class with a rectangle which contains the name of the class:



The objectives for our class models are:

- We want to build a system as fast and as cheap as possible that fulfils our current requirements.
- We want to build a system that is easy to service and easy to extend

To reach the first objective: *Every behaviour that we expect of our system must be provided by the available objects in a reasonable way.*

And for the second opbective, we have seen that we need a system with weak coupling, additionally:

A good class model consists of classes that represent classes of the problem domain that hardly change. These do not depend on special functionality that is required today.

How to design a class model:

DDD: Data driven Design: identify all data in the system, put the data into classes and then think about the responsibilities of the classes. Noun identification is a core aspect of DDD.

RDD: responsibility driven design: Identify all responsibilities, put responsibilities into classes and then identify all data for the classes. CRC-Cards are a part of RDD.

Noun identification: find all nouns from a user specification. (use singular). Remove everyting that ”does not fit”, rename the rest if necessary.

Examples of reasons not to create a class:

- redundant: Two things are essentially the same, or one is such a slight generalisation that it is not really necessary to create a separate class.
- vague: Something that is not propperly defined should not be a class. Clarify before you decide.
- an Event or an Operation: Normally, you don’t need classes, although in some cases, the events might be represented through a class.
- Metalanguage: ”requirement”, ”constraint”, ”system” probably do not need classes
- not belonging to the system: examples are ”week”, ”place” etc.

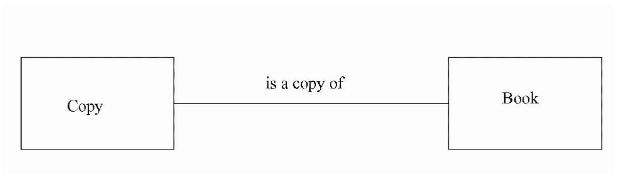
- nouns being an attribute: the ”name” of a ”person” normally does not lead to the creation of a ”name”-object.

What are objects and classes? From strong to weak arguments for a class:

1. physical things: a person, a book, a copy
2. roles: library member, student, teacher
3. events: arrival, departure, request
4. interactions: meetings, seminar

If we are talking about objects, we talk about their representation within a computer system. Therefore, we do not want to record irrelevant things in our system and remember that the objects are the system. For example, we do not want to implement a ”system” object that implements all functions and uses the objects only as datastructures.

An association in UML looks like this:

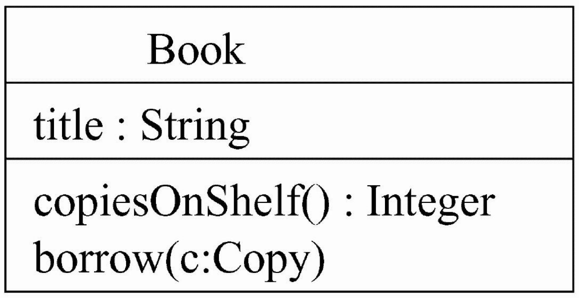


This association does not describe any relationship or navigability of the relation. If we want to indicate that there are a certain number of objects involved in the relation, we can use the `a..b`-notation. `1..*`, `0..5` or `7` are possible examples. These are written on both sides of the association line.

Classes A and B are associated if:

- 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 with values that are single or multiple objects of class B
- An object of class A receives a message that contains an object of class B as argument.

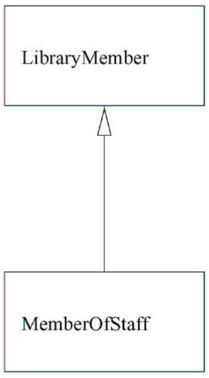
Classes can have attributes and operations.



Operations are executed if an object receives a message. Depending on the type of message, different methods may be executed to execute the operation, but normally, an operation is implemented by one method so we can use both terms. The signature of an operation specifies names and types of parameters and the type of the data returned.

Attributes have a name and a type.

A generalisation is a special relationship between classes. It is usually implemented through inheritance.



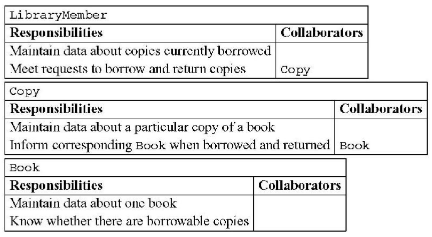
An object of a specialized class can replace an object of a more general class in every context that expects an object of the more general class but not vice versa.

There should be no conceptual difference between the reaction of both objects to the same message. (Example: object.GetSomething() should always return the same thing for all derived classes.)

Plan english test for generalisation: The class ”Foo” is a generalisation of the class ”Bar” if every ”Foo” is a ”Bar”.

Normally, inheritance is the method to implement a generalisation. But there are other options such as a composition.

CRC-Cards (CRC = Class Responsibilities, Collaborations)



LibraryMember

Responsibilities	Collaborators
Maintain data about copies currently borrowed	
Meet requests to borrow and return copies	Copy

Copy

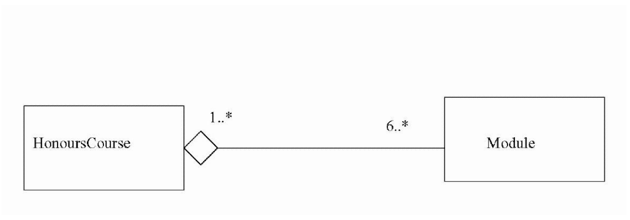
Responsibilities	Collaborators
Maintain data about a particular copy of a book	
Inform corresponding Book when borrowed and returned	Book

Book

Responsibilities	Collaborators
Maintain data about one book	
Know whether there are borrowable copies	

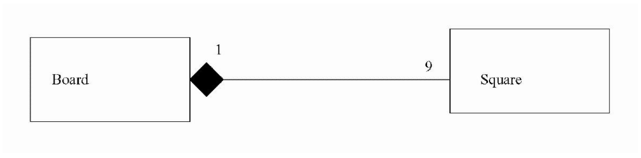
Refactoring is the process of changing (and improving) the object-oriented design of a system.

An aggregation is the UML way of expressing that one thing is a part of another.



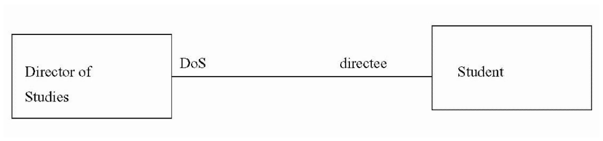
Note that the ”aggregation diamond” is on the side of the ”whole”, not on the side of the ”part”. Again, we can use the multiplicity notation to express numerical relationships.

If two things are always part of eachother, we can use a composition to express this:

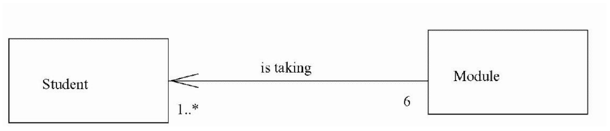


An aggregation is a composition if the parts do not exist without the whole.

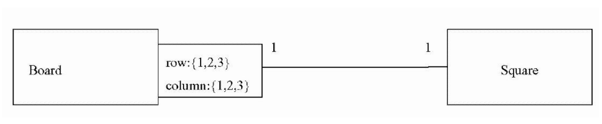
In an association, two classes can have specific roles for eachother. These can be written down on the association line.



Associations can be drawn with an explicit navigability. This means that one class can send messages to another class, but not vice versa.

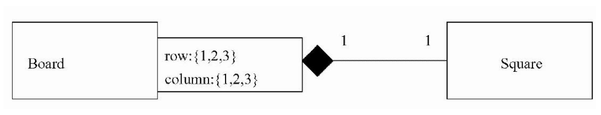


Associations can be qualified, i.e. there can be specific ”slots” in one object for the other objects.

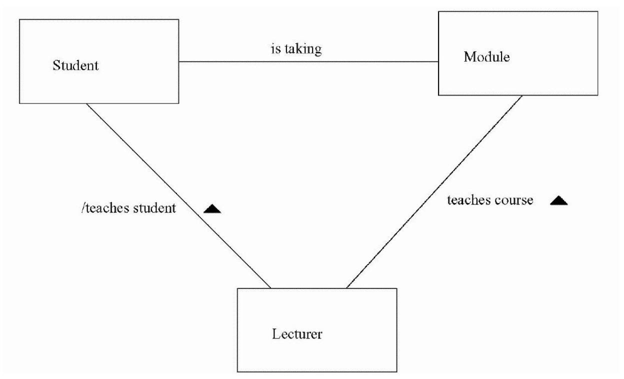


Qualification is similar to having specific roles for the objects of the association, in the example with the boardgame, there is one specific (1,1)-field on the board.

Compositions can be qualified too:

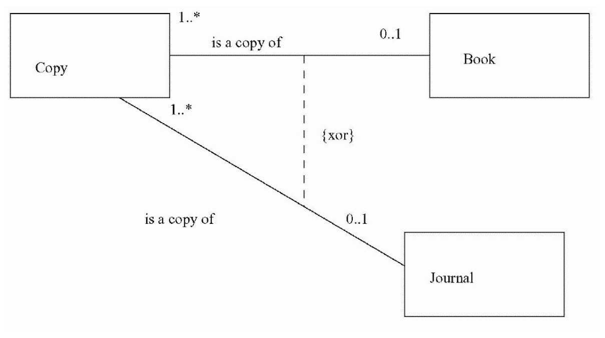


There can be derrived associations.



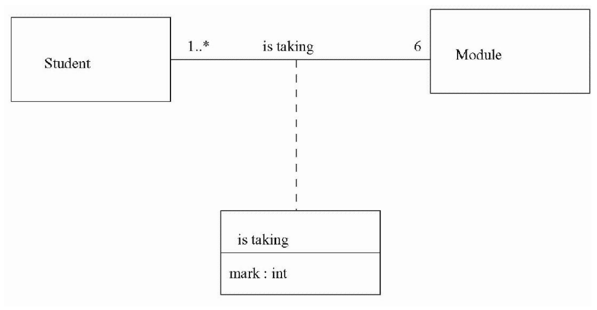
This is the case if an association between two objects leads to an association between each of the two objects and a third (and vice versa). In this example, if the student takes a course and the lecturer gives the course, then the lecturer and the student get an association.

Constraints are a possibility to express relations between associations.



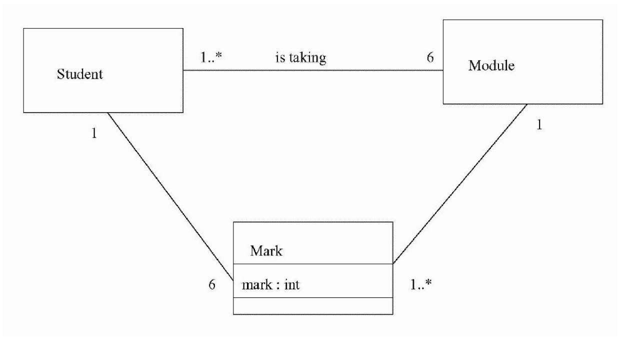
In this case, the constraint is that a copy is either a copy of a journal or a copy of a book.

Sometimes, it is necessary to record additional information with an association, such as the grade for a course (which is an attribute of the association of a student and a course).



In this case, we obtain an association class that carries the name of the association.

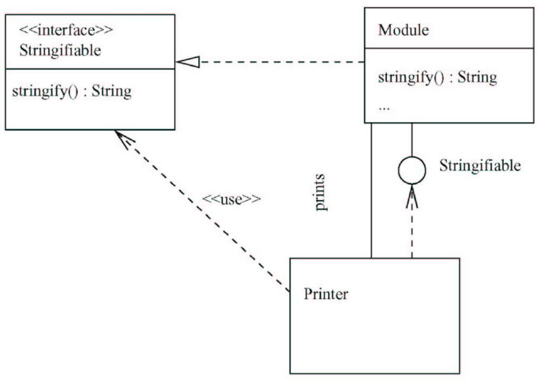
Instead, we could create an explicit class with a new name:



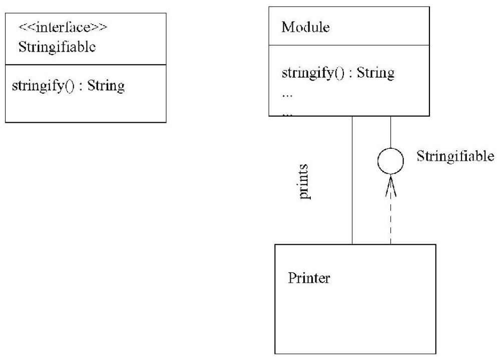
UML can be extended by using stereotypes. A stereotype is an added feature of a model element. Stereotypes are using the following notation: « `sterotype name` » . Examples for stereotypes are:

- « `interface` » defines a number of operations that all objects have to implement if they have this interface. It does not define an implementation for the operations.
- « `type` » is a class that defines an interface and has a state. It does not define an implementation or the attributes of a class.
- « `implementation class` » defines a class that contains the implementation for a type or for an interface.

An example for an interface definition is this:

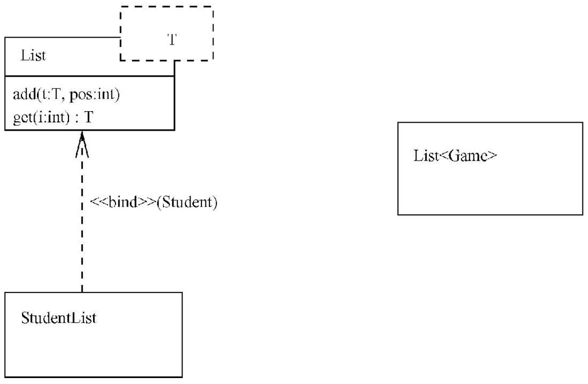


Normally, we do not need to draw the association between the interface and its user, so we can also use the following notation:



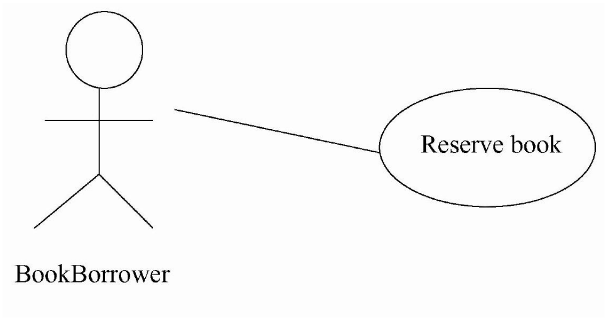
A mix between an interface and a class is an abstract class,indicated by the word `{ abstract }` in the class description. Such a class implements some but not all of its operations.

It is also possible to express parameterized classes in UML. These are the equivalent to templates in C++.



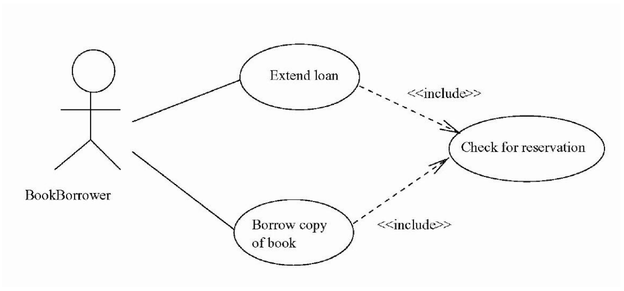
UML uses the so-called use cases to illustrate interactions with the system. One can imagine an use case as one of the things that users do with the system, i.e. borrowing a book from a library would be a use case for the library system.

The users in a UML use case diagram are drawn as so-called actors. This illustrates the fact that the same person may have different roles in different use cases, therefore they are acting according to their use case role.



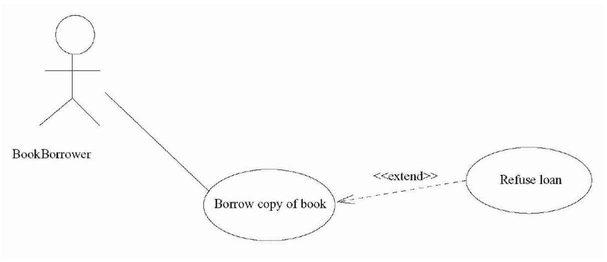
Every use case illustrates a task within the system, usually the actor has some kind of benefit from executing the task. Actors are named after the use case, not after the actual person interacting with the system. Often non-human actors such as external systems are added in use case diagrams too.

Use cases can include other use cases:



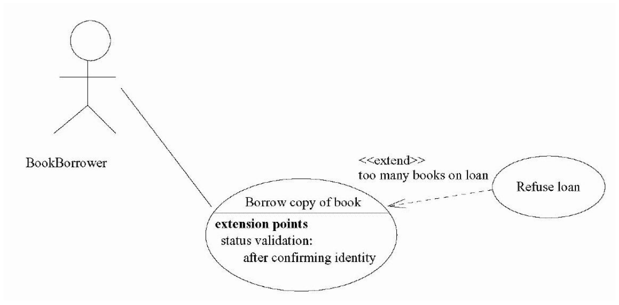
Include are used to illustrate a possible reuse.

In case that there are different possible outcomes in a use case, we use the extension of an use case.

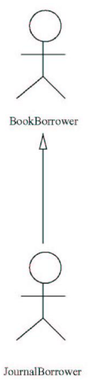


In this case, the extension is only used in some cases while the include directive defines an inclusion in all cases.

Like classes, actors can be generalized as well.



Instead of drawing the actor figure, the actor can also be drawn as a stereotype of a class:


BookBorrower
JournalBorrower