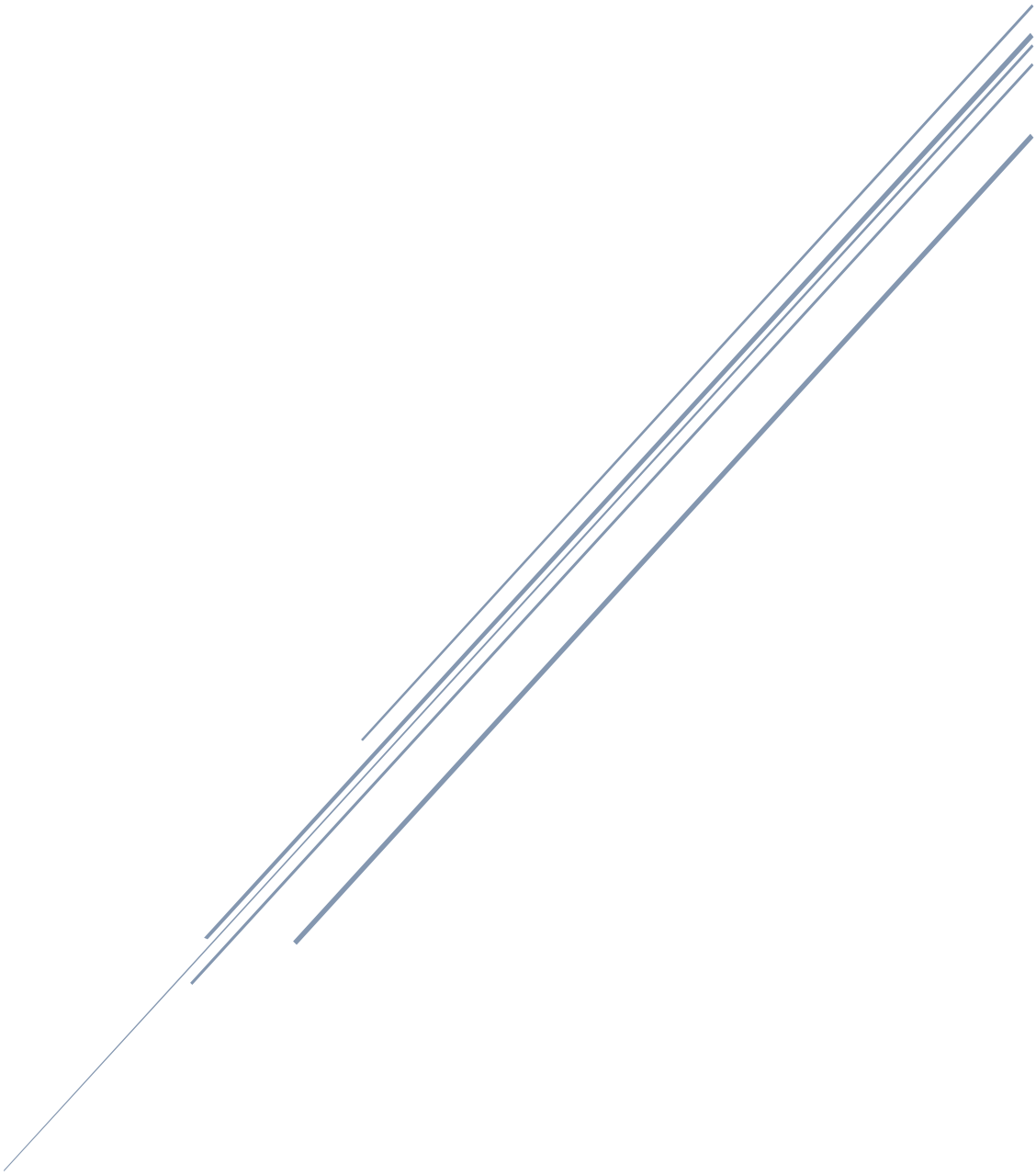
OBJECT ORIENTED PROGRAMMING

FUNDAMENTAL

MAXIME KELTSMA



www.smaltek.fr

Table des matières

1 Introduction

This document is for those wishing to understand the basic concepts of Object Oriented Programming (OOP), or clarify certain aspects.

The concepts, presented in a very simple way, are common to all object-oriented languages. No code example is given. The reader can move towards other publications concerning language in particular.

2 The OOP in two words

Try an OOP definition: it is a computer programming technique based on basic components called objects.

A program designed in this way will be materialized at run time, by a set of objects in the computer central memory. The program operations will then be based on the exchange of messages between these objects.

We will see that these objects have a life cycle, internal structure, and that they are designed to perform tasks that has been entrusted to them.

When talking about creation of a computer program, the programming phase is always preceded by an analysis phase in which the designer strives to define the characteristics of objects necessary for the functioning of the future software.

We will see that the principles of OOP are simply common sense and have nothing really complex.

3 A bit of history

The basic of OOP concepts emerged in the 1960s with the language Simula-67, then were completed in the 1970s with SmallTalk 71 and SmallTalk 80, themselves largely inspired by Simula-67.

During the years 80 and 90 more new languages natively object appear: C++, Lisp, Objective C, Java.

From the 1990s, OOP has become the reference in the whole of the computer programming industry.

During this era, some languages have appeared being natively object oriented. Others were not and are now with more or less success and sometimes with certain peculiarities.

Today object oriented languages are very numerous. Amongst the most famous: Java, C++, C#, PHP, Objective C, Delphi (Pascal), and even the venerable Cobol, appeared in 1959, became object oriented in 2002.

4 Classes and objects

4.1 A small analogy

Need us a theme to introduce the concepts. So choose an area that lends itself well to the game of OOP: boats.

In the world of shipyards were built boats of all sizes and for various uses.

First of all, a plan is prepared. The plan defines the features necessary for the future vessel construction:

- The number of bridges.
- The measures.
- The number of rooms
- The number of frames
- The shell thickness.
- …

Through the plan, it is possible to build as many boats as needed. They will all conform to the plan in terms of structure, but will have custom values:

- The color of the hull.
- The name of the boat.
- The registration.
- …

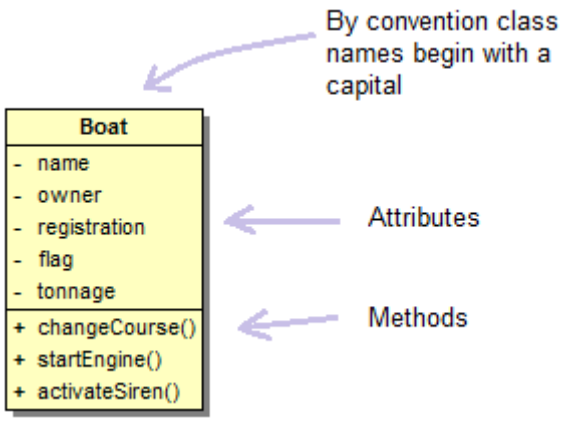
In OOP, the construction plan is called a class, and the boats are objects created from this class.

4.2 Class

The class describes the structure of the objects that will be instantiated. This structure is composed:

- Messages that the object will be able to treat. We call them methods. For example, a boat object must be able to process the following messages:
    - start_engine
    - change_course
    - activate_siren
- Attributes: characteristics that possess the object and whose values may be common to all objects, or the more often defined for each object. Imagine for example a boat object must have attributes such as:
    - A name
    - An owner
    - A registration
    - A Pavilion
    - A tonnage

Classes are represented this way:



4.3 Object

An object is created (instantiated) in central memory, by copy of a class. Therefore the object becomes an autonomous, structured element according to its original class, but whose attributes have eigenvalues. Also any object at its creation, was automatically given by the language, an identification number called reference. This reference is used internally by the language itself to manage the disposal of end-of-life objects.

The reference also allows that a message arrives to the object to which it is addressed. The object that emits the message must therefore have the reference of the recipient object.

One must distinguish on one hand, the creation of the reference, and on the other hand the assignment of an object to this reference. They are two separate operations. In the line of code following it creates a reference called B1 and we indicate that it is a Boat type. At this point the reference exists but references nothing:

```
B1 is a Boat
```

In the following line of code, we instantiate a boat object that we assign to our reference B1 (the instantiation operator 'new' is used here because existing in several major object languages):

```
B1 = new Boat
```

The two lines can be combined into one:

```
B1 is a Boat = new Boat
```

A class can generate as many objects as necessary. The only limit is the size of the computer central memory.

Some programming techniques allows to save objects in a database or in the form of a file. It's called **objects persistence** or **Serialization**. The goal is to put the object aside in order to remove it from the central memory and retrieve it later.

5 Life cycle of an object

Objects will be born, live and die in the computer central memory, during the execution of the program. All of these objects truly form the program.

The activity of the objects is to exchange messages. Typically an object receives a message from another object, performs the action corresponding to this message and return possibly a response to the message sender.

If object A wants to send a message to object B, it must know its reference. Language, which manages internally all references used by the program, can determine for example that an object is no longer referenced by any other. It cannot receive messages. This object is somehow out of the program. The only solution will be to eliminate it from the central memory. Many languages perform this automatic cleaning.

5.1 Instantiation

Instantiate verb refers to the action of creating an object from a class. You can meet the expression 'class instance' which means simply an object.

During its creation the object executes such a special method called a **constructor** that can contain some actions to perform by the object. The constructor() method can also be simply empty.

It is of course the class designer that defined the content of the constructor. Once the constructor method executed, the object is ready for receiving his first messages.

5.2 Active phase

During its active phase, the object only answer to messages sent to him. Each message received has to match a method that contains a treatment to be performed by the object. It can be extremely simple or more complex treatment.

If the object receives a message that he does not know, then a run-time error is generated.

5.3 Disposal

There comes a time when the object is no longer needed. There are at least 3 reasons for this:

- The program ends.
- The program has completed a certain task, and objects which he needed are no longer necessary.
- The object is no longer referenced.

In these cases, the objects must be removed from the central memory. The purpose is obviously to recover memory space.

In languages of older generations, the deletion of objects was made by the developer. The latter should ensure that program well removes objects after use. If this task was not accomplished, some objects could remain indefinitely in main memory while they were no longer in use.

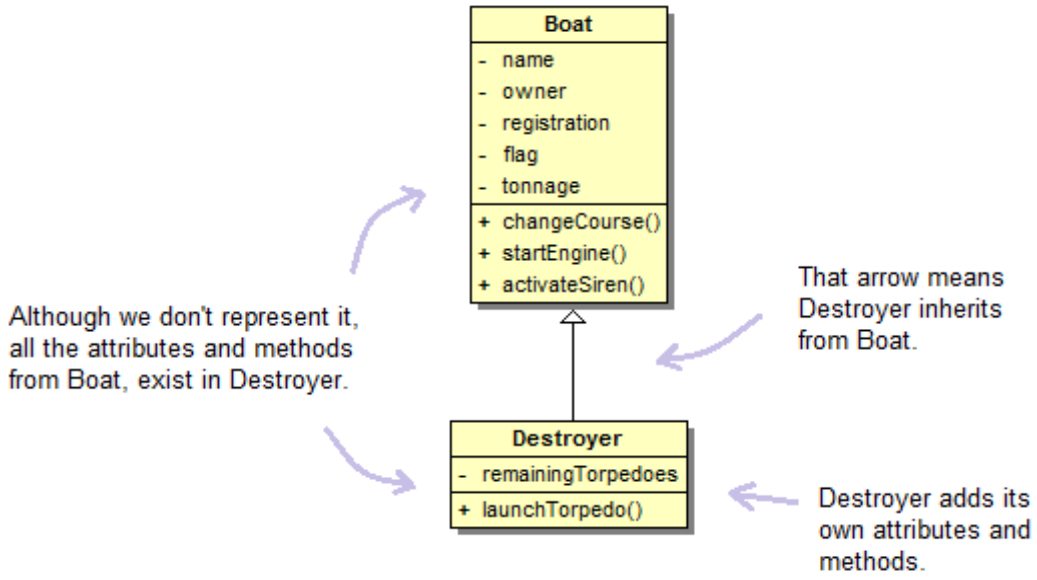
To remedy this and unload the developer of this thankless task, an automatic objects deletion system appeared in modern object languages. The principle of this system, called garbage collector, is simple: any object that is no longer referenced is removed from main memory. However, the developer always has the possibility to explicitly delete an object.

6 Inheritance

In OOP, inheritance is an automatism to recover the attributes and methods of a class.

Who inherits?

An object instantiated from a class, inherits the attributes of its class. But the power of inheritance is found in the inheritance between classes. Let's take our Boat class and create a new class inheriting from Boat:



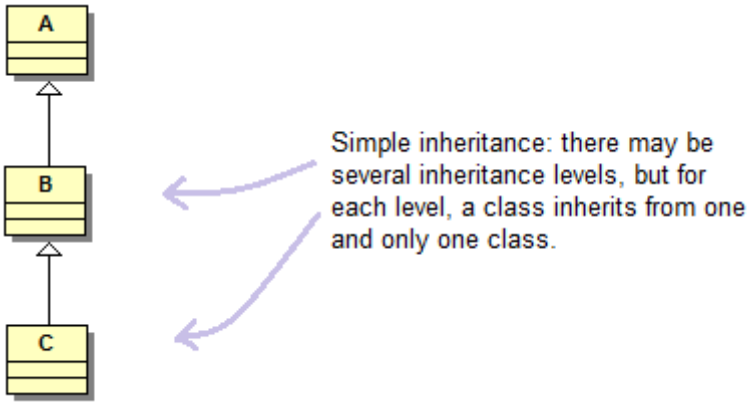
The Boat class is the **parent class** or **super class**.

The Destroyer class is called **derived class** or **subclass.**

A derived class adds attributes and methods to its legacy heritage. It becomes a more specialized class that does its parent class. Thus the parent classes are designed to be general: they have attributes and methods that relate to all subclasses. For example startEngine() method has no place in Boat class, because it would not have sense for a subclass "Sailboat" (assuming that Sailboat has no motor).

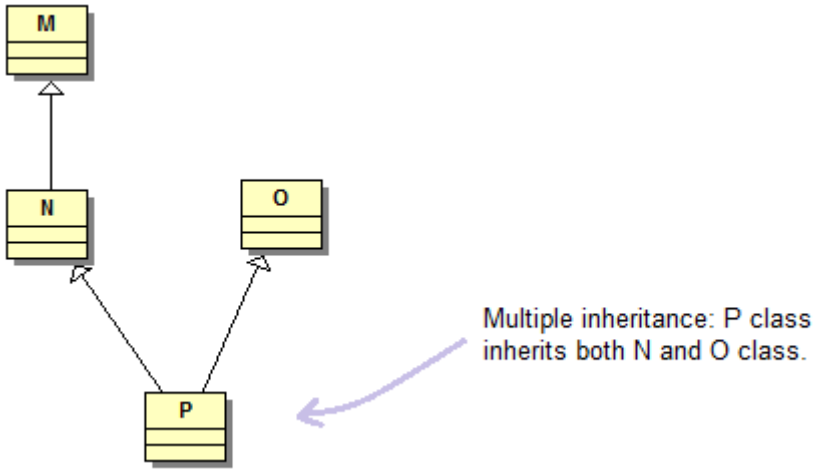
6.1 Single-inheritance

With simple inheritance, a class inherits directly from a single class.



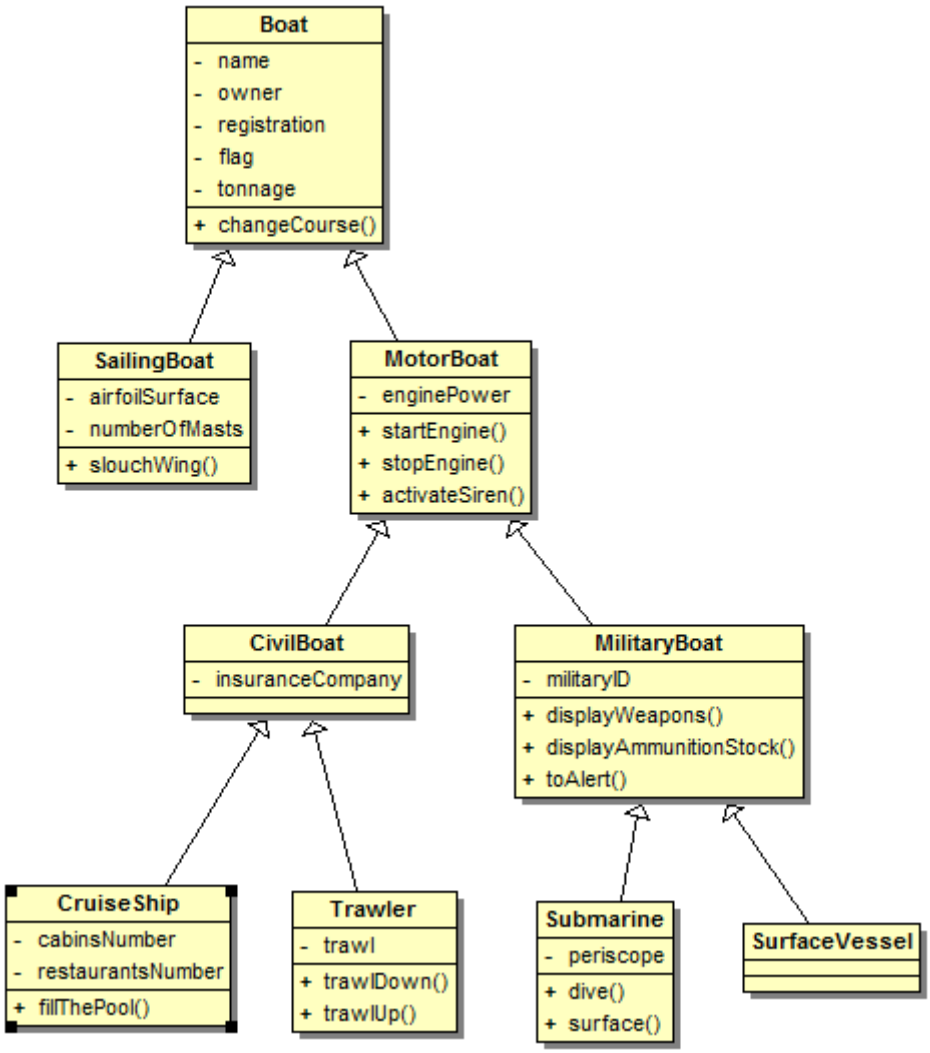
6.2 Multiple inheritance

With multiple inheritance, a class can directly inherit from multiple classes. This feature tends to disappear in recent object-oriented languages, as source of confusion and difficulty of maintenance. Instead is proposed a mechanism based on simple inheritance, associated with the technique of **interfaces** (see relevant chapter).



7 Class model

Now let's fully exploit inheritance. The following diagram is an example of a class diagram. The analyst develops this model to meet the specifications of the future software. A class diagram is not an exact science. There are always several approaches, several possible variations.



Some remarks:

- A class can have several direct subclasses.
- There may be several levels of inheritance (class, subclass, sub-subclass, etc...)

- A class implicitly has the attributes and methods of all of its parent classes. The Trawler class for example, has its own attributes and methods, plus those of CivilBoat, plus those of MotorBoat, plus those of Boat.
- The more we go down the inheritance tree, the most classes specialize and enrich themselves.
- The SurfaceVessel class doesn't have enrichment: it adds neither attributes nor methods to its legacy. This may be the designer choice at a moment, and poses no technical problem.
- It is theoretically possible to instantiate objects from any of these classes, but we will see later the notion of **abstract class** whose particularity is to prohibit the creation of objects, and allows only the creation of subclasses.

8 Encapsulation

Objects have attributes and methods, and in their generosity put these resources at the disposal of other objects. But some resources deserve to be protected. If our Submarine class had an attribute containing the code of missiles firing, it would be better that it is not freely accessible to all objects.

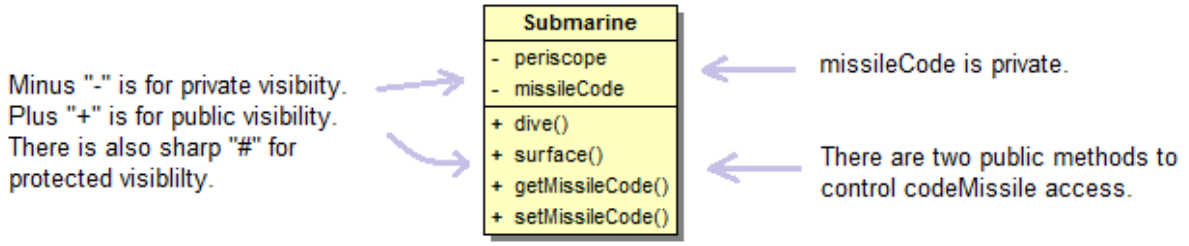
Protecting the attributes and methods of a class, is called encapsulation. These resources access will be controlled or even prohibited. In the latter case, this means that the resource is for the exclusive use of the object itself.

Encapsulation is based on the visibility given by the designer to the class attributes and methods. In OOP there is always at least these 3 visibility levels:

- **Public** : attribute or method will be accessible to all classes and objects of the program.
- **Protected** : the attribute or method will be available to objects of this class as well as objects of all its subclasses.
- **Private** : attribute or method will not be inherited by the sub classes. Object instantiated from this class will have the private resource, but it cannot be directly accessed by other objects.

A common approach to protect an attribute or a method, is to make private while providing public methods to control the access.

Take our Submarine class and give him the codeMissile attribute required for missiles firing. This sensitive data will have a **private** visibility, and so will be invisible from other objects. But it is necessary that this data can be read or modified under certain conditions. So we add 2 methods with **public** visibility:



- getMissileCode(): allows you to read the missile code. For example, this method may require entry of a password.
- setMissileCode(): method used to assign or modify the missile code. Here again the method may require entry of a password, and check also that the missile code provided, meets certain criteria.

These two methods are called **accessors**. View from the outside of the object, they are the only way to access an encapsulated data.

9 Abstract classes and interfaces

9.1 Abstract classes

A class is defined as abstract using the **abstract** modifier in its definition. By convention, and to facilitate their identification, the name of the abstract classes appears in italics in classes models. Classes that are not abstract are often called **concrete classes**.

Abstract classes contain attributes and methods. But we can also find **abstract methods**. They are "empty" (unimplemented) methods. Only the name of the method, its input parameters, and the type of its return value, are defined.

Also a class that contains at least one abstract method, must necessarily be too abstract.

The particularity of abstract classes is that they cannot create objects. What's the use in this case?

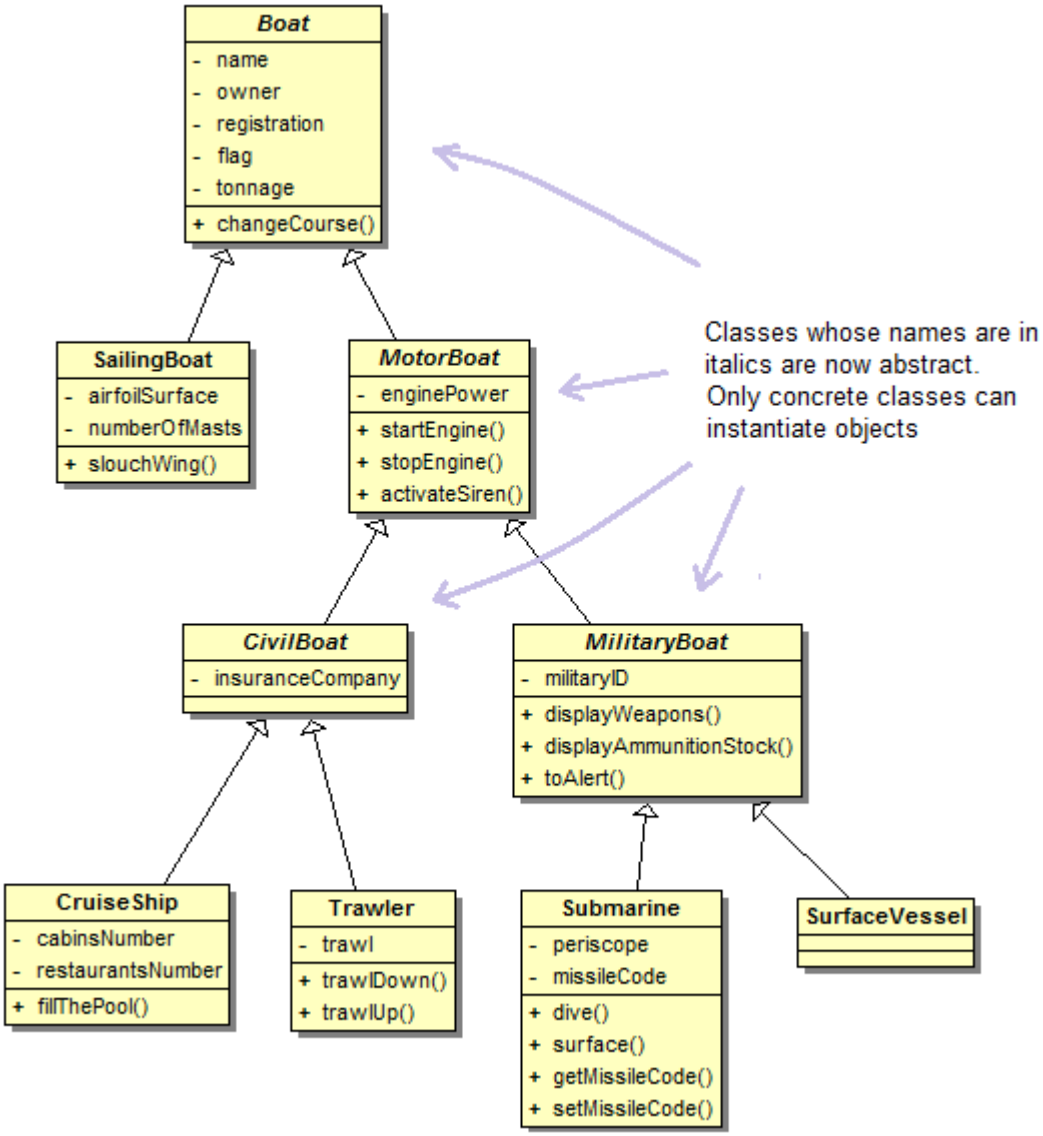
They are used to delegate the implementation of its abstract methods, to subclasses. The abstract class then is like a prototype, partially implemented, and sub classes will have to finish the job of implementation. We understand then why abstract classes should not create objects.

A class derived from an abstract class is not required to implement all the abstract methods. It can even implement none, but the sub classes remain abstract until there is at least an abstract method.

This delegation technique is one of the forms of polymorphism (see relevant chapter). In fact the sub classes can each implement an abstract method according to their specific need. So this method will have different behaviors according to the subclass: it is polymorphism.

For example, if we consider our classes model, the designer could declare as abstract all high level classes. Only sufficiently enriched classes will be able to create objects.

Note that in this example, the abstract classes contain no abstract methods (their names would be in italics). Technically, it is not mandatory.



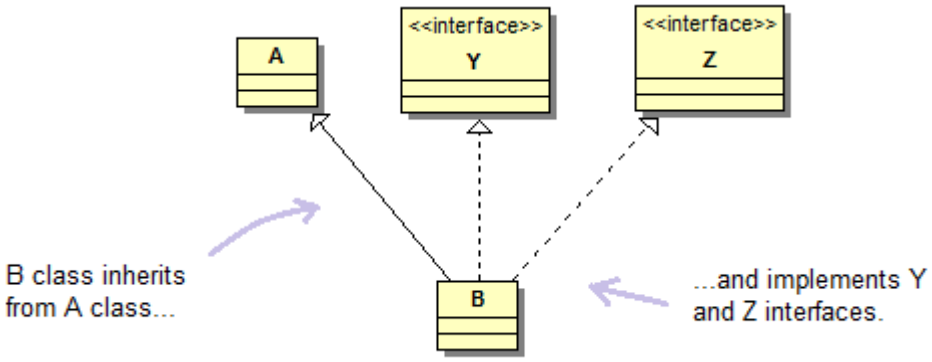
9.2 Interfaces

Push the abstract class concept to the limit: suppose an abstract class that has no attribute and has only abstract methods: it gets an interface.

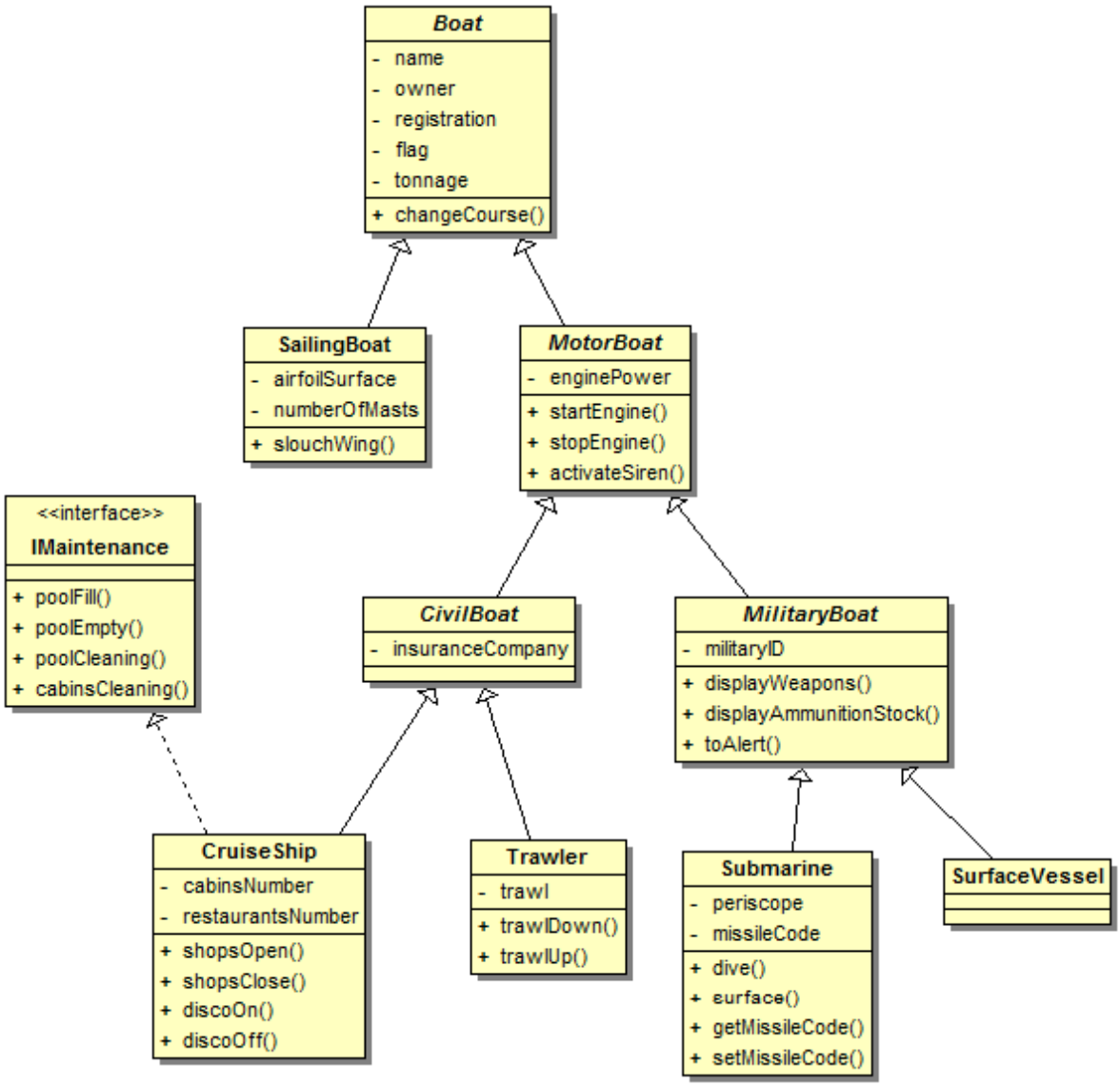
Interfaces are declared with the **interface** keyword. It is therefore not a class, but they offer the following possibilities:

- It contains only methods definitions (method name, input parameters type, and return value type).
- If a class implements an interface, then this class will have to define all the methods of the interface. We can consider the interface as a contract that the class must respect.
- As we have seen with simple inheritance (the most common), a class can only derive from a single parent class. However a class can implement multiple interfaces (there is no limit). This is a powerful tool that replaces multiple inheritance.
- It is possible to derive some interfaces as we do with classes.

The interfaces are marked with the stereotype < < interface > >. Note that inheritance arrows are continuous while interfaces implementing arrows, are dashed.



Below is a more concrete example: the CruiseShip class inherits from CivilBoat but at the same time implements the IMaintenance interface, which it will have to define the methods. The class will be able to receive and process new specific messages relevant to boat maintenance.



9.3 Other types of classes

We have seen that the **abstract** modifier allowed to define a class as abstract.

There are 3 other class modifiers:

- **Final** : A class declared **final** cannot have sub class, i.e. forbidden legacy. This class will only be able to create objects. It is in some way the opposite of abstract classes.
- **Private** : this class is accessible only from the file where it is defined. It will not be visible to other classes and objects.
- **Public** : (default modifier): the class is visible by all others.

Note that the **abstract, final, public** and **private**, modifiers are mutually exclusive.

10 Modifiers for methods

Methods also have their modifiers:

Final : any method defined with the **final** modifier cannot be overwritten in the sub classes.

Private : the method will be visible by the class, and will be invisible to other classes, including sub classes.

Protected : the method will only be visible by the class and its subclasses.

Static : the method is not copied into the objects created by the class. It will therefore exist only in a single copy, in the class itself, and is used commonly by all objects created by this class.

Public : the method will be visible by all classes (default modifier).

11 Polymorphism techniques

In OOP, methods can have several forms (they are polymorphic). It means that the same method name can exist in multiple classes, with different implementations, that is, they perform different treatments or at least with some variants. Polymorphism, which is one of the basis of OOP and especially **design patterns**, object programming models, covers several aspects.

11.1 Methods overload and overwrite

Before entering the heart of the matter, define the two ways to create multiple methods with the same name

11.1.1 Overload

In the same class, it is possible to create several methods with the same name, provided their signatures are different. The signature is defined by the number and type of input parameters, and the type of the return value. This is the **method overload** technique

At the time the method is called, the language will automatically choose the method that corresponds to the number and type of the parameters supplied in the call.

11.1.2 Overwrite

Method overwriting takes place in an inheritance relationship. The subclass can overwrite a method from his parent class, giving him a new implementation that will replace that of the parent class. The overwritten method keeps the same signature.

The overwritten method can optionally not replace, but acts in addition to the treatment of the parent class method. In this case the overwritten method starts by calling the parent class method and then run the complementary treatment.

11.2 Ad hoc polymorphism

It is a form of automatic polymorphism in which language, at run time or compile time, will decide on the action to be taken depending on the type of data involved in the treatment.

Consider for example the operator '+' in some languages can have two interpretations:

- Add values.
- Concatenate character strings.

According to the following lines of code, implicit adaptation of treatment, or even data conversion, will be done automatically by the language:

- **23 + 7** : here the program must add two integers. The data being of the same type, no conversion is necessary.
- **16.3 + 8** : here the program must add real value with an integer. The integer must be converted into real value and the result will be a real value. The developer does not have to worry about these conversions, they will be carried out by ad hoc polymorphism.
- **'It's midnight ' + 'doctor Schweitzer.'** : here the program have to concatenate two strings. Ad-hoc polymorphism will automatically adapt the treatment to the type of the arguments.

11.3 Parametric polymorphism

Parametric polymorphism is method overloading indicated above: a method is defined more than once in the same class with the same method name, but different signatures.

Consider our Submarine class. It has a dive() method. We indicate that it takes one parameter: an integer that gives depth to achieve:

dive(depth : integer)

Now we overload this method by adding a second dive() method taking as parameters, depth and a boolean indicating if it's a fast dive:

dive(depth : integer, fastDive : boolean)

We now have two dive() methods in the same class. They have different signatures, and obviously also different implementations.
When calling a dive() method, the language will be able to automatically choose the right one, based on parameters past in the call: it is parametric polymorphism.

11.4 Inclusion polymorphism

This type of polymorphism is based on inheritance. In the chapter about objects, we have seen that to create a reference we should indicate its type. At the moment to give a reference an object, several cases can occur:

1. The type of the instantiated object is different from that of the reference, and there is no inheritance link between both. In this case an error will be launched at compile time or run time depending on the used language.
2. The type of the instantiated object is the same as that of the reference. In this case of course the reference and the object are compatible:

   B1 is a SailingBoat = new SailingBoat

3. Type of the instantiated object, is a subclass of the reference type:

   B2 is a MotorBoat = new Trawler

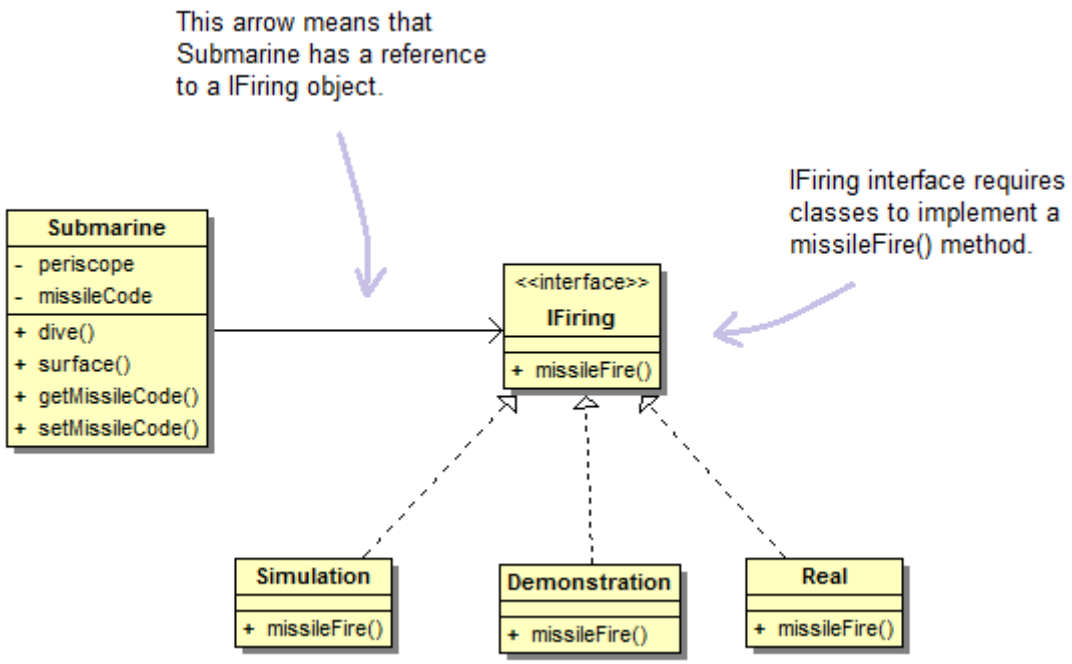
   The line of code is correct because the Trawler class indirectly is a subclass of MotorBoat class. B2 believes manipulate a MotorBoat object. In fact it is a Trawler, and indeed this is not absurd because a Trawler is a MotorBoat because it has inherited all the features.

   Thanks to this, it will be possible to introduce an inclusion polymorphism: for example, consider the startEngine() method defined in MotorBoat class, and suppose it has been overwritten in Trawler class to take into account some specifics of how to start the engine of a trawler. And when the startEngine() message will be sent to the B2 reference, this is the startEngine() treatment of Trawler object that will run without B2 knows nothing.

We can summarize saying that from B2 point of view:

- The apparent type is MotorBoat.
- The (hidden) actual type is Trawler

And that is not all: inclusion polymorphism also works well with interfaces. Imagine the following situation:



Generally in this kind of design, this is not the Submarine class that instantiates one of the 3 objects of missile firing, but a high-level object that controls precisely this kind of things. In the end, Submarine object has a reference to an IFiring object, without knowing what is the concrete object that was instantiated, among the 3 possible.

To launch its missile, Submarine object must invoke missileFire() method on the IFiring object. Obviously the implementations of missileFire() differ in 3 classes. The submarine itself, is even not aware of the existence of the three classes. He only sees an object of type IFiring.

This missileFire() method which can have multiple implementations, it is still the inclusion polymorphism.

12 Lexicon

Class : grouping of methods and attributes, acting as a template to instantiate objects.

Abstract class : class that cannot instantiate objects and can only serve as a parent class for sub classes.

Concrete class : as opposed to abstract classes, concrete classes can instantiate objects.

Constructor : method existing in any class, and executed at the time of object creation. The developer usually placed preparatory operations in it.

Design patterns : there are some twenty patterns. These are programming object models, each addressing a particular problem.

Destructor : method existing in any class and executed just prior to the destruction of an object. The developer usually placed cleanup operations in it.

Encapsulation : object programming technique to check or even prohibit access to certain attributes or methods.

Inheritance: relationship between two classes. The inheriting class, retrieves methods and attributes from its parent class.

Instantiate : for a class: creating an object.

Interface : set of methods not implemented, defined in a file. A class that implements an interface is required to define all the interface methods.

Abstract method : unimplemented method defined in a class, and whose the implementation is delegated to the sub classes.

Object : A class instantiates (creates) objects. Objects are copies of the class in computer central memory.

Polymorphism : literally: which has several forms. Said a method appearing in several classes of the same family, with the same name but with different implementations.

Persistence : storing information on a medium other than the computer central memory, allowing you to retrieve this information even after the computer restarted.

Overwrite : A method is overwritten when a new implementation is given in a class.

Signature of a method : all the information that distinguish methods between them: name of the method, name and type of input parameters, type of the return value.

Overload : A method is overloaded when, in the same class, a method with same name is created but with a different signature.

Serialization of an object: recording of an object in the form of a file, allowing you to reload this object later, or transmit this object via the network.

Visibility : more or less restricted perimeter that is given to an attribute, method, or a class, using key words (public, private, protected) and that determines who can access or not to this element.

End of document