Object Oriented Programming in C++

Principles of OOP. Classes. Objects.

2018

Rosana Matuk
Facultad de Ingeniería y Tecnología Informática
Universidad de Belgrano, Argentina

Contents

1. Modelling concepts
2. Class
3. Object
4. Class contracts
5. Some advantages of defining classes
6. Relationships between classes
7. Conclusions

Contents

1 Modelling concepts

2 Class

3 Object

4 Class contracts

5 Some advantages of defining classes

6 Relationships between classes

7 Conclusions

Which is the Problem?

One important problem that a software engineer must face when he/she is to develop a software application is the following:

- The application domain has high-level concepts and is human-oriented (e.g., student, academic subject, professor, etc.).
- The constructs offered by traditional programming languages are usually quite low-level and machine-oriented (array, integer, pointer...).

The immediate consequence of this abstraction gap between what is needed by the software developer and what is offered by the programming language is that any piece of software, ranging from medium-size programs to huge applications, becomes increasingly difficult to construct, test, maintain and reuse.

Object-oriented programming (OOP) paradigm

Goal of the OOP Paradigm

To bridge the abstraction gap, so that, we can construct a software application using directly entities that represent high-level domain concepts (student, subject, date, etc.).

How does OOP achieve its goal?

Allowing the definition of new kinds of types.

In the same way as predefined types are defined in terms of a set of operations that can be applied to the entities of those types (e.g., the predefined type integer has the operations +, -, *, / and remainder associated to; these operations can be applied to integers) it makes perfectly sense to define new high-level types giving the list of services they offer to their prospective users.

Abstraction

Abstraction is the property by which we describe an object focusing only in some aspects of it and forgetting deliberately other aspects which are not interesting for us for the time being.

Two different abstractions are involved in the definition of a new type:

1. (what) We do not consider all its aspects, we just focus on those issues which are of interest for its users and forget the rest.
2. (how) When we define a type as a set of operations which describe the behaviour of its entities, we forget about how that behaviour will be internally implemented.

Abstract type: what

Abstract type: Definition

An abstract type denotes a set of entities characterized by a list of operations that may be applied to them together with a precise specification of each one of these operations.
Usually, the list of operations that define a type and their specification are referred to as the type behaviour, type specification or the type contract.
An abstract type is also called interface.
The set of entities which share the operations defined for a type are called instances of that type.

Contents

1. Modelling concepts
2. Class
3. Object
4. Class contracts
5. Some advantages of defining classes
6. Relationships between classes
7. Conclusions

Abstract type: how to make them software?

We have defined an abstract type in terms of its behaviour (the list of the operations it provides along with their specification).

However, in order to execute a software application that uses instances of an abstract type it is still necessary to choose a structure or representation for these instances (e.g., the way in which they will be stored, probably in the computer memory) and the implementation of the operations of the type in terms of that representation.

Type representation

Type representation: Definition

The representation of the instances of a type is the way chosen by the designer to store these instances internally (usually in the computer memory), so that the applications that use those instances can be executed.

This representation is performed in terms of lower-level elements defined previously by the type designer or provided by the programming language in which the type is being designed.

Type representation: examples

- An instance of the type `Date` can be represented in terms of three integers: day (1..31), month (1..12) and year (1900-2100).
- An instance of the type `String`, which models a sequence of characters, can be represented internally in terms of an array in which the characters of the string are stored and an integer to state how many characters a specific instance of `String` has. An alternative representation for this type would be an array of characters finished with a special character mark.
- An instance of the type `Student` can be represented in terms of:
  - Two instances of the type `String` (to represent his/her name and id.).
  - An instance of the type `Date` (to represent his/her birth date).
  - An array of instances of type `String` (to represent the list of subjects in which he/she has enrolled).

Implementation of the type operations

Implementation of the type operations: Definition

The implementation of the type operations is the implementation of the operations defined for the type in terms of the type representation.

OOP languages: novel artifact

Most object-oriented programming languages provide a construct that allows:

- the abstract definition of types in terms of their operations
- the representation of the type in terms of lower-level elements and
- the implementation of those operations.

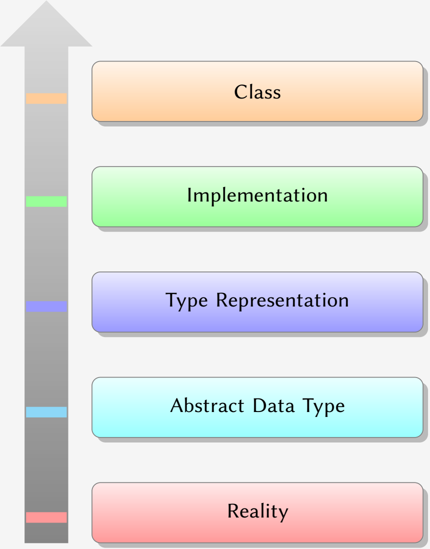
This construct is called class.

Class

Class: Definition [Meyer1997]

A class is an abstract type together with its representation and implementation.

Recap: Solving the abstraction gap


Class

Implementation

Type Representation

Abstract Data Type

Reality

Class definition in C++: example

```
class Date{
public:
    void create(int pday, int pmonth, int pyear, bool& err);
    int getDay();
    int getMonth();
    int getYear();
    void setDay(int pday, bool& err);
    void setMonth(int pmonth, bool& err);
    void setYear(int pyear, boole& err);
    void copy(Date d);
    void addDays(int ndays);
    unsigned int daysinBetween(Date d);
private:
    unsigned int day;
    unsigned int month;
    unsigned int year;
};
```

Class definition in C++: example (cont.)

```
void Date::create(int pday, int pmonth, int pyear, bool& err){
    if (correctDate(pday,pmonth,pyear){
        day=pday;
        month=pmonth;
        year=pyear;
        err=false;
    }
    else{
        err=true;
        ....
    }
}
void Date::setDay(int pday, bool& err){
    if (correctDate(pday,month,year)){
        day=pday; err= false;
    }
    else err=true;
}
....
```

Definition of the class `Date`: analysis

Class specification (or class behaviour)

It is stated in the part of the class definition in terms of a list of public operation headers. These are the operations that any user of this class can apply to its instances.

Class implementation

- Class representation: It follows the `private` label. The items `day`, `month` and `year` are the *class* attributes and they constitute the class representation. Class users cannot access the elements defined in the private part of the class.
- Operation implementation: This implementation is usually given out of the class block (often in another file). The name of each operation is preceded by the name of the class to which it belongs.

Class definition in two layers

Interface or specification layer

- Enumeration of the services offered by the class (i.e., the class behaviour). The services offered by the class are called methods or operations and are enumerated by means of their header or signature. In C++, this enumeration of method headers comes up following the label `public:` in the class definition.
- Detailed specification of the class behaviour. This specification is also called class contract and includes the class invariant and the precondition and postcondition for each operation. Class contracts are usually contained in a separated file.

Class definition in two layers (cont.)

Implementation layer

- Class structure (or class representation): All the attributes that compose a class instance. These attributes constitute the internal representation of a class, that is, the data that any instance of the class should store in order to offer the intended class behaviour. These attributes may be either values of predefined types (e.g., int) or instances of other classes. The user programs cannot access these attributes. In C++, attributes appear following the label `private:` in the class definition.
- Operation implementation: It is constituted by the implementations of the operations that form the class interface (and, hence, provide the class behaviour). This implementation is carried out in terms of the attributes that constitute the class representation (and, optionally, in terms of other auxiliary class operations).

Class members

Attributes and operations are the building blocks that are used to define the structure and behaviour of a class and are called class members.

All class instances have the same attributes and can be accessed by means of the same set of operations.

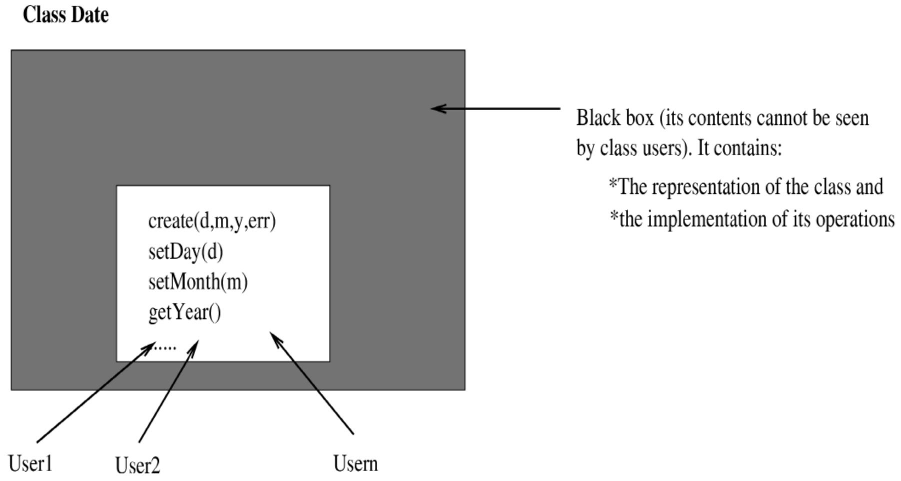
Class user vs. class implementer

- The class user (or class client) is only interested in the interface layer. He/she will use the class operations without caring about the implementation layer. The class construct, as it is defined in most programming languages, enforces this idea by forbidding the user any access to the implementation layer. This policy is called encapsulation or information hiding.
- The class implementer provides the implementation layer for the class.

Class user vs. class implementer: class contract

The link between the class user and the class implementer is provided by the class contract: The class user commits him/herself to using the class operations in the precise way stated by the contract. On the other hand, the class implementer commits him/herself to implementing the class so that it provides the services stated by the contract in the exact way in which they have been stated.

Information hiding



Class Date

create(d,m,y,err)
setDay(d)
setMonth(m)
getYear()
.....

User1

User2

Usern

Black box (its contents cannot be seen by class users). It contains:

*The representation of the class and

*the implementation of its operations

Information hiding: Advantages

- *Complexity reduction in application development.* The user of the class `Date` can use the high-level services offered by this class without having to bother, at the same time, about how those services are implemented. In other words, abstraction of the low-level implementation details helps in managing complexity.
- *Representation independence and easier reuse.* The user of the class `Date` will not notice those changes that are made in the class definition, as long as they do not modify the services offered by the class. The class designer may change the class structure or the implementation of the class services in order to find a more efficient alternative or in order to comply with new class requirements. However, if these changes conform with the services previously offered by the class, the old users will be able to use the new class definition without having to change their applications.

Contents

1. Modelling concepts
2. Class
3. Object
4. Class contracts
5. Some advantages of defining classes
6. Relationships between classes
7. Conclusions

Object

Object: Definition

An object is a run-time instance of some class.

Object: Characteristics

Objects may be characterized by three aspects [Booch1991]: state, behaviour and identity.

Object: state, behaviour and identity

Object: state

The state of an object is constituted by the object attributes together with the information stored in them at a given run time instant.

Object: behaviour

The behaviour of an object describes how other objects may interact with it. The interaction with an object `obj` is always carried out by means of the operations defined by the class of which `obj` is an instance.

Object: identity

Identity is the property of an object which makes it different from any other object.

Object behaviour: Types of operations

There may be considered three different kinds of operations according to what they do to the state of the class instance to which they are applied:

- **Creators or constructors.** They create and initialize the instance to which they are applied.
- **Modifiers.** They modify the state of the instance to which they are applied.
- **Consultors or queries.** They return some piece of the state of the instance to which they are applied but they do not modify this state.

Types of operations: example

The operations of the class `Date` may be classified in:

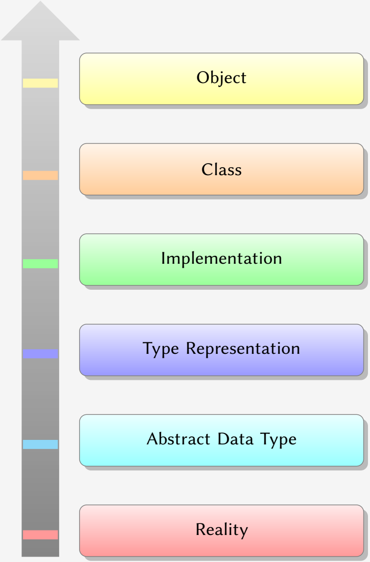
- *Constructor:* `create`
- *Modifiers:* `setDay, setMonth, setYear, copy, addDays`
- *Queries:* `getDay, getMonth, getYear, daysInBetween`

Classes vs. objects

A class is the description of the structure and behaviour that are shared by all the objects which are instances of that class. The objects are the real entities (allocated in the computer memory) which have the structure described by the class and to which class operations can be applied. Each object has a separated identity.

Example: car model vs. real car.

Recap: Solving the abstraction gap



Object

Class

Implementation

Type Representation

Abstract Data Type

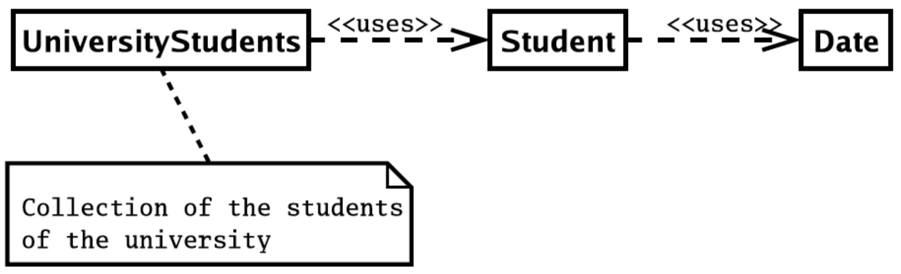
Reality

The object orientation programming paradigm

A program designed using the object orientation programming paradigm can be described as a collection of objects, instances of different classes, that collaborate among them by calling the operations defined in the classes of which they are instances.

An object o1 of class C1 that calls some operation of another object o2 of class C2 is a client or user of the class C2 (we may also say that class C1 is a client or user of class C2). On the other hand, class C2 is a provider of some services for class C1.

Client vs. provider: example



The class `Student` is a "client" of the class `Date` because it uses the class `Date` in order to store/access the birthdate of a student.
In its turn, the class `Student` is a provider for the class `UniversityStudents`.
This latter class is intended to store all the students enrolled in the university.

Contents

1 Modelling concepts

2 Class

3 Object

4 Class contracts

5 Some advantages of defining classes

6 Relationships between classes

7 Conclusions

Class contracts

Class contract or class specification: Definition

The class contract (or class specification) states precisely the operations offered by a class and their precise behaviour.

The class user commits to use the class operations in the way stated by the class contract. On the other hand, the class implementer commits to implementing the class so that it provides the services stated by the contract in the exact way in which they have been stated.

Class contracts

The specification of class behaviour by means of a contract is essential in O.O. programming.

Every class must have a contract.

The class contract must state **what** does the class do instead of **how** does it do it. That is, *the class contract should not be written in terms of the class representation*, nor should it describe the details of the implementation of the class operations.

Defining a class contract

Structure of class contracts: Definition

Class contracts are defined by means of the following elements:

- Precondition and postcondition assertions, associated to each class operation, which state the operation behaviour precisely.
- Class invariants associated to the class, which states the global properties of the class.

Precondition

Precondition: Definition

A precondition is an assertion attached to a class operation which establishes those constraints which should be true at the beginning of the execution of that operation.
This assertion must be guaranteed by the client of the class, not by the operation to which that operation has been attached.

Precondition: example

We may choose between two different preconditions for the operation create(d,m,y):

- Precondition: void. This means that the client may produce a call today.create(d,m,a) without any special checking of the three parameters. In this case, the implementation of create(d,m,y) will be responsible for ensuring that the three parameters constitute a correct date. If this is not the case, it should manage the error in some way:
  - By means of a (boolean) error parameter (err) which becomes true if d-m-y are not a correct date: today.create(d,m,y,err) or
  - Throwing an exception object.
- Precondition: d-m-a represents a correct date later than 1-01-1900. This means that the client, before calling today.create(d,m,y) should make sure that d-m-y constitutes a correct date. In this case, the operation implementation will not check d-m-y to ensure that it is a correct date. Hence, no error will be produced by the operation.

Precondition: criterion

Assign the responsibility for checking a condition to the part (user or operation implementation) which can take it in a more natural way.
In most occasions, a void precondition is preferable since it leads to a more robust solution. However, sometimes, non-void preconditions are more natural.

Precondition: example

The class IntegerStack models a collection of elements of type integer, so that the insertions and deletions of integers in the collection are performed at the top of it (i.e., the last element to get into the stack is the first to get out). One of the operations of the class IntegerStack is the following: void pop(); This operation removes from the stack its uppermost element. It is an error to try to apply the pop() operation to an empty stack. How should we deal with this error?

- A void precondition and the error is managed by the implementation of the operation by returning a boolean (output parameter) which is true if it has been attempted to remove an element from an empty stack (i.e., void pop(bool err);).
- A precondition which states that the stack is not empty.

In this case, the second alternative is more natural because most of the algorithms that work with stacks never try to pop an element from an empty stack.

Postcondition

Postcondition: Definition

A postcondition is an assertion associated to a class operation which establishes those properties that should hold at the end of the execution of that operation, provided that the precondition held at the beginning of its execution.

This assertion must be guaranteed by the class implementer.

Postcondition

The postcondition of an operation will show usually two different forms according to the type of the operation:

- Creators and modifiers: The postcondition should contain the way in which the object to which the operation has been applied has been modified as a result of such operation.
  It is important to notice that the postcondition should focus on *which* changes have taken place on the object rather than on *how* these changes have been implemented.
- Consultors: The postcondition should indicate which part of the object state is obtained by the operation and where it is obtained. In addition, it should indicate that the state of the object to which the operation has been applied has not changed with respect to its precondition.

Example

```
void addDays(int ndays);
```

Call: `x.addDays(ndays)`

Pre: void

Post: The resulting date `x` is the date `x@pre` plus `ndays` days.

Remember: any use of the private members of a class in either the postcondition or the precondition is explicitly and strictly forbidden!!!

Class invariant

A class invariant is an assertion that expresses certain constraints on the state of class objects. These constraints should hold:

- After the creation of an object of that class (i.e., after the application of a creator/constructor operation to a class object).
- Before and after the application to a class object (which has already been created) of any operation which belongs to the class interface with arguments which satisfy the class invariants of their respective classes.

In the case of creator (or constructor) operations, the class invariant is only required after their execution.
That is, the class invariant should hold after the creation of an object and should be maintained by any operation which is applied afterwards to that object (provided that the operation has valid arguments).

Contents

1 Modelling concepts

2 Class

3 Object

4 Class contracts

5 Some advantages of defining classes

6 Relationships between classes

7 Conclusions

Advantages of using Classes

Classes as abstract definition of new types

Data abstraction is the main way proposed by the OOP paradigm to deal with the complexity inherent to the construction of software applications.
The addition of a new type in a programming language involve two conceptually different things: the definition of the services offered by the type and the implementation of those services in terms of the language constructs. Keeping both issues separated contributes to reduce the complexity of the management of those types.
OOP languages provide the notion of class as a way to define new types by separating the services offered from their implementation. As a consequence, the user of these classes only has to care about what services are offered by them. By abstracting the implementation issues, he/she reduces the complexity of its management.

Advantages of using Classes

Encapsulation and information hiding

To ensure that the class user does only access the class services, its implementation aspects are considered private. The user of a class can only access its public issues, which are constituted by the services offered by the class. That is, the implementation is hidden from the user. As a consequence, we get independence of a class from its implementation. Since the programs that use a class cannot access its implementation, this implementation may change (for instance, to make it more efficient) without affecting those programs. **The only requirement is that the class contract does not change.**

Advantages of using Classes

Modularity and reuse

The goal of reuse is to be able to construct software modularly by reusing already constructed software modules, so that we do not have to design from the scratch every new piece of software. The class, in O.O. programming, constitutes a very complete and natural reusable module, which encapsulate data abstractions, but also functional abstractions. Class reuse is easy through its interface, which provides a contract. Information hiding ensures the robustness of the user applications that reuse a class with respect to implementation improvements in it.

It is difficult to be able to reuse a module in many different contexts without any adaptation. A specific unit will make a good reusable module only if it provides mechanisms to be easily adapted to different situations. Classes provide those mechanisms: inheritance, aggregation, polymorphism and genericity make it possible to adapt a class so that it can be reused in different contexts.

Advantages of using Classes

Relationships between classes

Concepts in real life are linked between them by means of different types of relationships. For instance, the concept “person” is more general than the concept “woman”. An instance of concept “car” should contain four instances of the concept “wheel”. The concept “collection” can be constituted by “persons”, “cats”, “documents” or whatever.

Since one main purpose of O.O. programming is to model reality and we use classes in order to do so, it is necessary that O.O. programming languages provide tools to model these kinds of relationships between classes. At least, they should offer: class specialization-generalization, class aggregation, and genericity.

Advantages of using Classes

Relationships between classes (cont.)

- Class aggregation: An object of a class can be part of the structure of an object of another class (e.g. an object of class `Wheel` can be a part of an object of class `Car`).
- Class specialization-generalization: We can define hierarchies of classes. The higher a class comes up, the more general the concept it models is (e.g., the class `Person` is more general than the class `Man`).
- Genericity: The definition of some classes will accept other classes as parameters. For instance we may define a class `CollectionOfItems` as a collection of objects of a specific class `Item`. However, `Item` will be a parameter of CollectionOfItems. When an object of `CollectionOfItems` is created at execution time, an actual parameter will be bound to `Item` stating the specific type of the items that will compose the collection. In this way, we may create collections of students, of persons, etc.

Contents

1 Modelling concepts

2 Class

3 Object

4 Class contracts

5 Some advantages of defining classes

6 Relationships between classes

7 Conclusions

Classes: Relationships between classes and reusing

Relationships between classes

- Genericity: Templates
- Class association:
  - Aggregation
  - Composition (strong aggregation)
- Class specialization-generalization: Inheritance, Abstract classes

Reuse: Polymorphism

- Over-Riding
- Over-Loading

Genericity

Generic class: Definition

A class A is generic if its definition depends on one or more types (usually, classes) that act as parameters of A.
A generic class is *instantiated* when the type parameter is substituted by an actual type.

Genericity

Template class definition

A template class definition is a class definition which depends on one or more parameter types (called template parameters). This template class definition is used in order to create an actual class by instantiating the template parameter with a specific type when an object instance of that class is created.
An actual class A does not exist until its template parameter has been instantiated.
The following notation is used:

- Definition:
  template <class T> class A
- Instantiation: `A<int> a;`

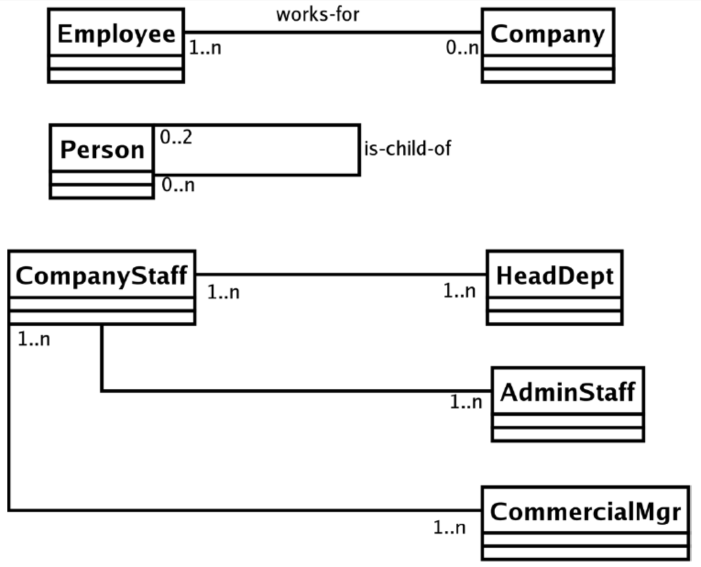
Asociation

Asociation: Definition

An association between the classes A and B establishes a semantic relationship between these classes, according to which, a specific instance of one class (e.g., A), is linked with 0, 1 or various instances of the other one (B).

The extension of an association is constituted by the pairs (a,b) of linked instances (a is an instance of A and b is an instance of B).

Representation of associations using UML



Agregation

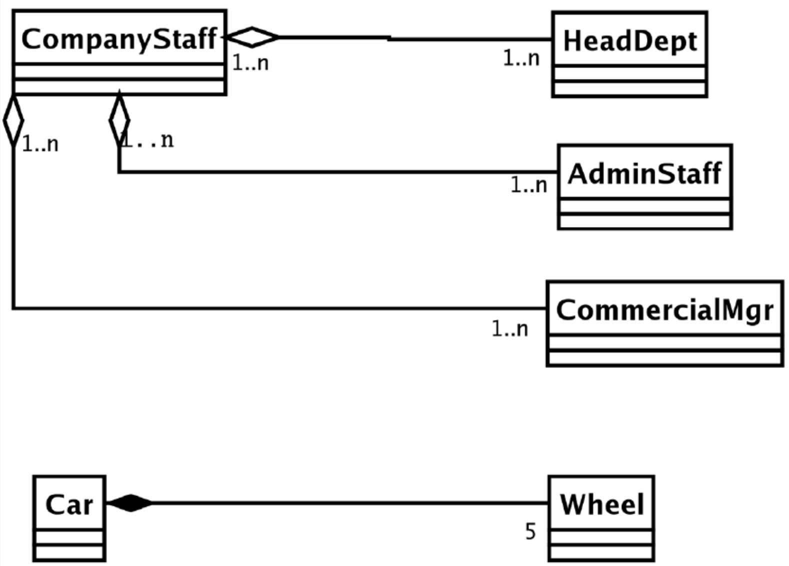
Agregation: Definition

An aggregation is a special kind of association between exactly two classes which represents a whole-part relationship between a (whole) class and another (part) class. The instances of the former are constituted by instances of the latter.

We say that an instance of the whole class has one or several instances of the part class.

Aggregations can be organized in a hierarchical way (i.e., forming trees of concepts: A is an aggregation of Bs, which, in their turn, are aggregations of Cs). However, aggregations cannot generate loops (e.g., C cannot be, at the same time, an aggregation of As).

Modelling of aggregations using UML



Compositions

Composition: Definition

A composition is a strong form of aggregation. It has two additional properties:

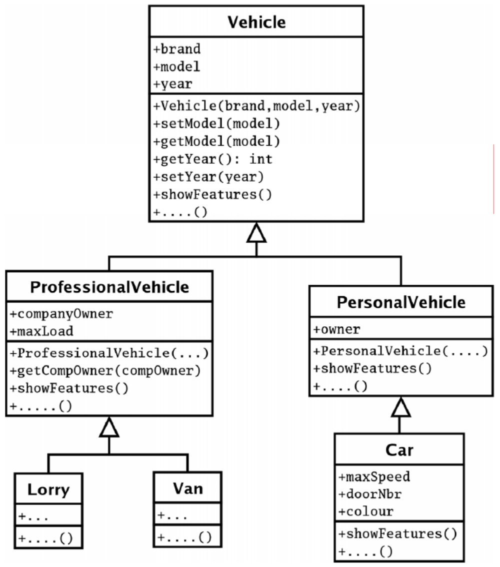
- Any instance of the part class can participate in, at most, one composition at a given instant. (However, it can be deleted from one composition and added into another).
- When the composite object is deleted, all its parts are (usually) deleted with it.

Inheritance

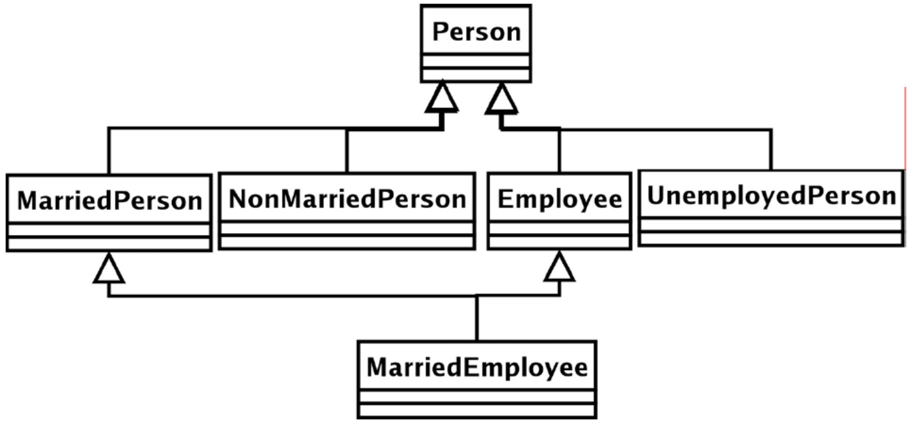
Subclass: Definition

A subclass B is a class that keeps the same structure (attributes) and behaviour (operations) that another class A (which is called B's *superclass*). We say that B inherits the structure and operations of A.
In addition, the subclass B may add new attributes or operations to those inherited from A.

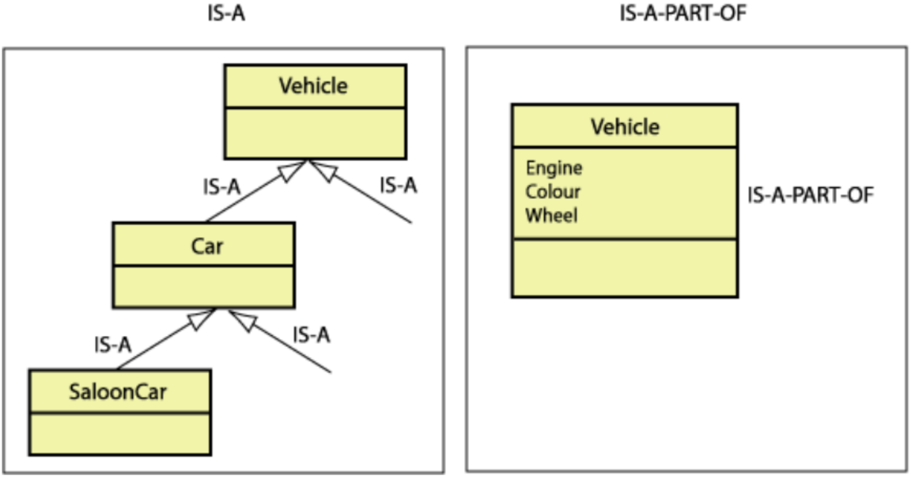
Inheritance: example



Multiple inheritance: example



Inheritance vs. Composition: test

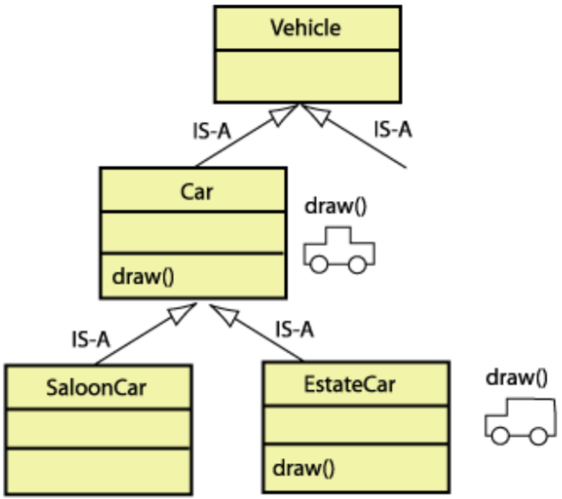


Polymorphism

Polymorphism means "multiple forms". In OOP these multiple forms refer to multiple forms of the same method, where the exact same method name can be used in different classes, or the same method name can be used in the same class with slightly different paramaters. There are two forms of polymorphism, over-riding and over-loading.

Polymorphism: Over-Riding

Over-Riding: redefinition of an inherited method to provide specific behaviour for a derived class



Polymorphism: Over-Loading

Over-Loading: The same method name can be used, but the number of parameters or the types of parameters can differ, allowing the correct method to be chosen by the compiler.

Example:
add (int x, int y)
add (String x, String y)

Abstract Classes

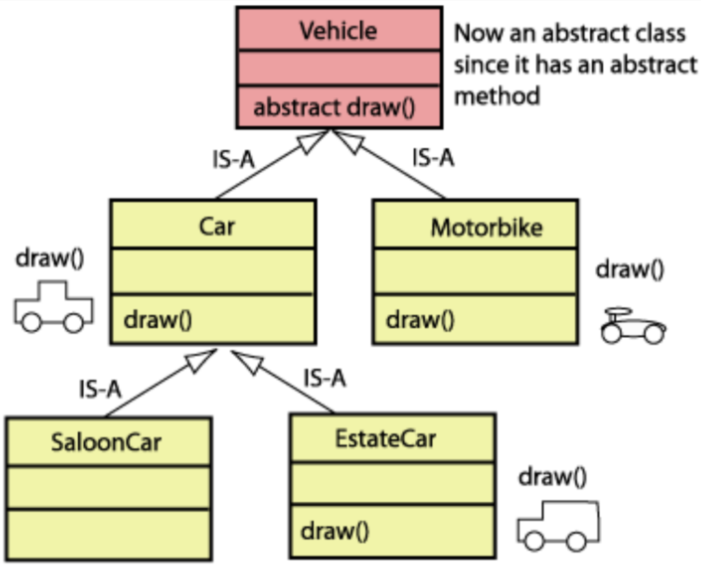
Abstract Class: Definition

An *abstract class* is a class that is incomplete, in that it describes a set of operations, but is missing the actual implementation of these operations. Abstract classes:

- Cannot be instantiated.
- So, can only be used through inheritance.

Example: In the Vehicle class example previously the draw() method may be defined as abstract as it is not really possible to draw a generic vehicle. By doing this we are forcing all derived classes to write a draw() method if they are to be instantiated.

Abstract Class: Example



UML Notation: Recap

- Classes are represented with blocks. The name of the class is written at the top of the block. The representation (atributes) and operations of the class are written inside the block.
- Inheritance is represented by arrows.
- Associations are represented by lines with cardinality.
- Aggregations are represented by white diamonds.
- Compositions are represented by black diamonds.

Contents

1 Modelling concepts

2 Class

3 Object

4 Class contracts

5 Some advantages of defining classes

6 Relationships between classes

7 Conclusions

Conclusions

The essence of structured programming is to reduce a program into smaller parts and then code these elements more or less independently from each other. Although structured programming has yielded excellent results when applied to moderately complex programs, it fails when a program reaches a certain size. To allow for more complex programs to be written, the new approach of OOP was invented. OOP, while allowing you to use the best ideas from structured programming, encourages you to decompose a problem into related subgroups, where each subgroup becomes a self-contained object that contains its own instructions and data that relate to that object. In this way complexity is reduced, reusability is increased and the programmer can manage larger programs more easily.

Conclusions

All OOP languages, share the following three capabilities:

- Encapsulation: the ability to define a new type and a set of operations on that type, without revealing the representation of the type.
- Inheritance: the ability to create new types that inherit properties from existing types.
- Polymorphism: the ability to use one name for two or more related but technically different purposes ("one interface, multiple methods").

References

- Josep Maria Ribó, Ismet Maksumić and Siniša Čehajić, "Introduction to OOP with C++", Edicions de la Universitat de Leida, 2005.
- Robert Nürnberg, "Object Oriented Programming in C++". Imperial College London. 2016.
- Notes of "EE402 - Object-oriented Programming with Embedded Systems", School of Electronic Engineering, Dublin City University, 2018.