Oriented Object Programming Concepts

La programació orientada a objectes està basada en objectes, com el seu nom indica, un objecte és una representació informàtica d'un element del món real. Tot pot ser un objecte, i aquests objectes tenen característiques i comportaments. Un exemple del món real podria ser un llapis, un llapis té les seves característiques, com el color, duresa o llargària, a més també té comportaments, com escriure, i així podríem passar un munt d'exemples del món real a objectes.

Tots els objectes comencen des d'una plantilla, aquesta plantilla se'n diu classe. En una classe es defineixen les característiques i comportaments d'un objecte, les característiques o atributs són variables i els comportaments o mètodes funcions.

L'objecte que creem a través d'una classe se'n diu instància. Totes les instàncies d'una mateixa classe comparteixen els mateixos atributs i mètodes que la classe. El que varia entre instàncies és el contingut dels atributs, ja que en una classe els atributs no estan inicialitzats amb valors, sinó que són les instàncies les que assignen valors propis als atributs de la classe.

Per a crear una classe primer hem de tenir clar que és l'encapsulament, l'encapsulament consisteix en l'ocultació de les decisions de disseny, de manera que els canvis en una classe afectin el mínim possible el programari ja desenvolupat.

Els modificadors d'accés permet modificar l'accessibilitat a variables, mètodes i classes, hi ha quatre tipus de modificadors d'accés, privat, públic, protegit i paquet. Privat fa que els membres siguin visibles només en aquesta classe. Públic que els membres siguin accessibles també des de fora de la classe. Protegit que els membres siguin visibles per aquesta classe i les seves subclasses. Paquet que els membres siguin visibles des de dins del mateix paquet.

Els modificadors de no accés no canvien l'accessibilitat d'atributs i mètodes, sinó que poden alterar el seu comportament i els hi proporcionen propietats especials, d'aquests hi ha tres tipus, estàtic, abstracte i final. Estàtic fa que els atributs i mètodes pertanyin a la classe i no a la instància. Abstracte defineix classes o mètodes com a abstractes, estan declarats, però no definits. Final modifica un membre perquè aquest no pugui ser canviat.

Una altra cosa important que hem de tenir en compte sobre la creació de classes és que una classe ha de tenir un constructor. Un constructor és un mètode sense retorn dins de la classe que serveix per a poder crear instàncies. El constructor d'una classe ha de tenir el mateix nom que la classe, i al no retornar res no escriurem res com a retorn.

Seguin l'exemple del llapis, aquest és un exemple amb C#. És important saber que és bona pràctica tenir cada classe en un arxiu propi de la classe, si haguéssim de fer una nova classe anomenada goma hauríem de crear un arxiu nou que contingui la nova classe. Seguin amb el llapis, aquest té els seus atributs, creats, però no inicialitzats, i els seus mètodes. Els atributs d'una classe per defecte són privats, i això és el que volem, que no es pugui accedir als atributs des de fora de la classe, sinó que haguí d'utilitzar el constructor o altres mètodes per a accedir als atributs de la classe. Com que no podem modificar els atributs d'una classe hem de crear mètodes per a poder modificar-los, hem de crear una funció Set i una de Get per a cada atribut, així podrem canviar i veure el valor dels atributs. Aquests mètodes al contrari dels atributs volem que siguin accessibles des de fora de la classe, així que són públics. Per a poder crear instàncies de la classe és necessari el constructor, aquest cridarà als mètodes Set per a modificar els atributs de la classe i poder crear una instància.

Per a crear una instància de la classe llapis ho hem de fer amb la paraula reservada new. En utilitzar new creem una instància de la classe i cridem al constructor de la classe, i així se'ns crea un nou objecte amb la plantilla.

L'herència també és un concepte fonamental en la programació orientada a objectes, aquesta s'utilitza per a crear classes a través de classes ja definides. En el cas del llapis podríem fer una herència en la qual el pare o superclasse seria llapis i els seus fills o subclasses podrien ser llapis de fusta o llapis portamines, aquestes subclasses comparteixen els mateixos atributs i mètodes que el seu pare, a més poden tenir atributs o mètodes únics de la seva classe, com la duresa de la goma d'esborrar que tenen els llapis portamines a l'altre costat. D'herències també hi ha dos tipus, simples i múltiples. En l'herència simple una classe només pot heretar d'una altra classe, mentre que en la múltiple, una classe pot heretar de múltiples classes a la vegada.

Un altre concepte important és el polimorfisme, aquest permet implementar amb el mateix nom en diverses classes, cadascun amb una funcionalitat diferent. Tenim dos tipus de polimorfisme, Overloading i Overriding. L'Overloading consisteix a tenir diversos mètodes amb el mateix nom, però amb diferents paràmetres, això permet diferenciar un mètode d'un altre en funció dels paràmetres que li passem. L'Overriding consisteix a tenir un mètode en la classe pare que s'hereta a la classe fill, aquest mètode pot ser sobreescrit per a la classe fill i ser canviat completament, mantenint el tipus de retorn, ja que el fill hereta tots els atributs i mètodes del seu pare.

Seguint amb l'exemple del llapis, ara aquest és una superclasse de les classes llapis de fusta i llapis portamines. Per a crear una classe que és fill d'una altra en la declaració de la classe afegim dos punts i el nom de la classe pare, i en el constructor afegim dos punts i la paraula reservada base i dins dels parèntesis tots els atributs de la classe pare. La paraula reservada base cridarà el constructor del pare per a crear les instàncies del fill. Un exemple d'Overloading el podem aplicar al mètode escriure, creem un altre mètode igual, però en aquest afegim un paràmetre que serà la superfície on està escrivint el llapis. Per a fer l'overriding també podem utilitzar escriure, a l'afegir la paraula reservada virtual al pare podem sobreescriure aquest mètode des d'un fill amb la paraula reservada override.

Una helper class és una classe que conté mètodes que assisteixen o ajuden a altres classes principals en la realització d'altres tasques. La helper class normalment no té atributs, sinó que només té mètodes, aquests mètodes són estàtics i poden ser utilitzats sense haver d'instanciar la classe, s'utilitza per a agrupar lògica relacionada que no pertany a cap altra classe.

En el cas del nostre llapis podem fer una helper class amb un mètode que ens digui quin tipus de llapis és el que tenim o un mètode que ens digui si el llapis necessita ser canviat perquè s'ha utilitzat moltes vegades.

En la programació orientada a objectes també hi ha relacions, aquestes representen connexions entre classes. Les relacions descriuen com els objectes de diferents classes interactuen entre ells. Hi ha dos tipus de relacions, IS-A i HAS-A.

La relació IS-A és un concepte de les herències, i és unidireccional. Es representa mitjançant una fletxa buida que surt de la classe filla i que acaba a la classe pare. Seguint amb el nostre llapis, el llapis és material d'oficina, però el material d'oficina no és un llapis.

La relació HAS-A és un concepte en el qual una classe té una altra classe com a component o membre. En el cas del nostre llapis, aquest té una mina de grafit dins seu, i això implica que conté la mina com a part de la seva estructura. La relació HAS-A pot ser de tres tipus, agregació, composició o associació.

L'agregació tracte que un objecte pot estar format per altres objectes, i aquests objectes poden existir sense aquest objecte. Es representa mitjançant una línia contínua que finalitza en un dels extrems per un rombe buit, sense omplir. El rombe buit s'ubicarà a la part de l'objecte base. Ara tenim un estoig, l'estoig conté llapis, i els llapis poden seguir existint sense l'estoig.

La composició al contrari que l'agregació, els objectes no poden existir sense formar part d'un altre objecte. Es representa mitjançant una línia contínua que finalitza en un dels extrems per un rombe ple. El rombe ple s'ubicarà a la part de l'objecte base. El nostre llapis està format per una mina de grafit i una carcassa de fusta, la mina i la carcassa per si soles no poden existir, necessiten a llapis.

L'associació defineix les connexions entre dos o més objectes, la qual cosa permet associar objectes que instancien classes que col·laboren entre si. Es representen amb una línia contínua on es defineix la multiplicitat, representada amb el format [n...m]. Aquí un exemple, l'estudiant utilitza el llapis, l'estudiant i el llapis són entitats independents i no tenen cap relació de tipus conté tal cosa o està fet de tal cosa.

Oriented Object Programming Concepts

Object-oriented programming is based on objects, as its name suggests. An object is a computational representation of a real-world element. Anything can be an object, and these objects have states and behaviors. A real-world example could be a pencil. A pencil has its own states, such as color, hardness, or length, and it also has behaviors, like writing. In this way, we could translate many real-world examples into objects.

All objects start from a template, and this template is called a class. In a class, the states and behaviors of an object are defined. The states, or attributes, are variables, and the behaviors, or methods, are functions.

The object we create from a class is called an instance. All instances of the same class share the same attributes and methods defined by the class. What varies between instances is the content of the attributes, as attributes in a class are not initialized with specific values. Instead, it is the instances that assign their own values to the class's attributes.

To create a class, we first need to understand encapsulation. Encapsulation involves hiding design decisions so that changes within a class have the least possible impact on the software that has already been developed.

Access modifiers allow us to modify the accessibility of variables, methods, and classes. There are four types of access modifiers: private, public, protected, and package. Private: Members are visible only within the same class. Public: Members are accessible from outside the class. Protected: Members are visible to the class itself and its subclasses. Package: Members are visible within the same package.

Non-access modifiers do not change the accessibility of attributes and methods but instead alter their behavior and provide them with special properties. There are three types of non-access modifiers: static, abstract, and final. Static: Makes attributes and methods belong to the class rather than the instance. Abstract: Defines classes or methods as abstract, meaning they are declared but not implemented. Final: Modifies a member so that it cannot be changed.

Another important aspect to consider when creating classes is that a class must have a constructor. A constructor is a non-returning method within the class that is used to create instances. The constructor of a class must have the same name as the class, and since it does not return anything, no return type is specified.

Following the pencil example, here's an explanation using C#. It’s important to note that best practice is to have each class in its own file. If we were to create a new class called "Eraser," we would need to create a new file to contain the new class.

Continuing with the pencil, it has its attributes, which are created but not initialized, and its methods. By default, a class's attributes are private, which is what we want—to prevent access to the attributes from outside the class. Instead, the constructor or other methods must be used to access the class's attributes.

Since we cannot directly modify a class's attributes, we need to create methods to manage them. For each attribute, we create a **Set** method to modify its value and a **Get** method to retrieve it. Unlike the attributes, these methods should be accessible from outside the class, so they are public.

To create instances of the class, a constructor is necessary. This constructor will call the **Set** methods to modify the class's attributes and enable the creation of an instance.

To create an instance of the **Pencil** class, we use the reserved keyword **new**. By using **new**, we create an instance of the class and call the class constructor, which creates a new object based on the class template.

Inheritance is also a fundamental concept in object-oriented programming. It is used to create classes based on already defined classes. In the case of the pencil, we could have an inheritance where the parent or superclass is **Pencil**, and its children or subclasses could be **Wooden Pencil** or **Mechanical Pencil**. These subclasses share the same attributes and methods as their parent, but they can also have unique attributes or methods specific to their class, such as the hardness of the eraser in mechanical pencils.

There are also two types of inheritance: **single** and **multiple**. In **single inheritance**, a class can inherit from only one other class, while in **multiple inheritance**, a class can inherit from multiple classes at the same time.

Another important concept is **polymorphism**, which allows methods with the same name to be implemented in different classes, each with a different functionality. There are two types of polymorphism: **Overloading** and **Overriding**. **Overloading** involves having multiple methods with the same name but different parameters. This allows us to differentiate between methods based on the parameters we pass to them. **Overriding** occurs when a method in the parent class is inherited by the child class. This method can be overridden in the child class and completely changed, while still maintaining the return type, since the child class inherits all the attributes and methods from the parent.

Continuing with the **Pencil** example, now it serves as a superclass for the **Wooden Pencil** and **Mechanical Pencil** classes. To create a class that is a child of another, we add a colon and the name of the parent class in the class declaration. In the constructor, we add a colon and the reserved word **base**, followed by all the attributes of the parent class within the parentheses. The **base** keyword will call the parent class's constructor to create the child class instances.

An example of **Overloading** can be applied to the **write** method. We create another method with the same name but add a parameter, such as the surface where the pencil is writing. This allows us to differentiate between two **write** methods based on the parameters passed.

For **Overriding**, we can also use the **write** method. By adding the reserved keyword **virtual** to the parent class, we can override this method in a child class using the reserved word **override**. This allows the child class to change the behavior of the method while maintaining the same method signature.

A **helper class** is a class that contains methods which assist or help other main classes in performing various tasks. The helper class typically does not have attributes but only methods. These methods are static and can be used without having to instantiate the class. It is used to group related logic that doesn't belong to any other class.

In the case of our **Pencil**, we could create a helper class with a method that tells us what type of pencil we have, or a method that tells us if the pencil needs to be replaced because it has been used too many times.

In object-oriented programming, there are also relationships, which represent connections between classes. These relationships describe how objects from different classes interact with each other. There are two types of relationships: **IS-A** and **HAS-A**.

The **IS-A** relationship is a concept of inheritance, and it is unidirectional. It is represented by a hollow arrow pointing from the child class to the parent class. Continuing with our **Pencil** example, a pencil **IS-A** office supply, but an office supply is not necessarily a pencil. This illustrates the hierarchical structure where the child class (the pencil) is a specific type of the parent class (office supply), but the reverse is not true.

The **HAS-A** relationship is a concept where one class has another class as a component or member. In the case of our **Pencil**, it **HAS-A** graphite mine inside it, meaning the pencil contains the mine as part of its structure. The **HAS-A** relationship can be of three types: aggregation, composition, or association.

Aggregation refers to a relationship where one object is made up of other objects, but these objects can exist independently of the containing object. It is represented by a solid line ending with an empty diamond shape. The empty diamond is placed on the side of the base object. For example, a **Pencil Case** contains **Pencils**, and the pencils can still exist without the pencil case.

Composition refers to a relationship where the contained objects cannot exist without being part of the containing object. It is represented by a solid line ending with a filled diamond shape. The filled diamond is placed on the side of the base object. For example, a **Pencil** is made up of a **Graphite Mine** and a **Wooden Casing**, and the mine and casing cannot exist by themselves; they need the pencil to exist.

Association defines the connections between two or more objects, allowing the association of objects that instantiate classes which collaborate with each other. It is represented by a solid line where multiplicity is defined, shown as [n...m]. For example, a **Student** uses a **Pencil**. The **Student** and the **Pencil** are independent entities and do not have a relationship of "contains" or "is made of."