Encapsulation (zapouzdření)

In object-oriented programming, **encapsulation** refers to the **bundling of data with the methods that operate on that data, or the restricting of direct access to some of an object's components.** Encapsulation is used to hide the values or state of a structured data object inside a class, preventing unauthorized parties' direct access to them. Publicly accessible methods are generally provided in the class (so-called *"getters"* and *"setters"*) to access the values, and other client classes call these methods to retrieve and modify the values within the object.

This mechanism is not unique to OOP. Implementations of abstract data types, e.g., modules, offer a similar form of encapsulation.

Virtual methods, virtual method table (VMT)

early-binding, late-binding methods

In object-oriented programming, when a derived class inherits from a base class, an object of the derived class may be referred to via a pointer or reference of the base class type instead of the derived class type. If there are base class methods overridden by the derived class, the method actually called by such a reference or pointer can be bound either **'early'** (by the compiler), according to the declared type of the pointer or reference, or **'late'** (i.e., by the runtime system of the language), according to the actual type of the object referred to.

Virtual functions are resolved **'late'**. If the function in question is 'virtual' in the base class, the most-derived class's implementation of the function is called according to the actual type of the object referred to, regardless of the declared type of the pointer or reference. If it is not 'virtual', the method is resolved **'early'** and the function called is selected according to the declared type of the pointer or reference.

Virtual methods (virtuální metody)

A virtual method is a method that **can be redefined in derived classes.** A virtual method has an implementation in a base class as well as derived the class. **It is used when a method's basic functionality is the same but sometimes more functionality is needed in the derived class.** A virtual method is created in the base class that can be overriden in the derived class. We create a virtual method in the base class using the virtual keyword and that method is overriden in the derived class using the override keyword.

When a method is declared as a virtual method in a base class then that method can be defined in a base class and it is optional for the derived class to override that method. The overriding method also provides more than one form for a method. Hence it is also an example for polymorphism.

When a method is declared as a virtual method in a base class and that method

has the same definition in a derived class then there is no need to override it in the derived class. But when a virtual method has a different definition in the base class and the derived class then there is a need to override it in the derived class.

When a virtual method is invoked, the **run-time type** of the object is checked for an overriding member. The overriding member in the most derived class is called, which might be the original member, if no derived class has overridden the member.

Virtual method table (virtuální tabulka metod)

Whenever a class defines a virtual function (or method), most compilers **add a hidden member variable** to the class that points to an **array of pointers to (virtual) functions** called the *virtual method table*. **These pointers are used at runtime to invoke the appropriate function implementations, because at compile time it may not yet be known if the base function is to be called or a derived one implemented by a class that inherits from the base class.**

There are many different ways to implement such dynamic dispatch, but use of virtual method tables is especially common among C++ and related languages (such as D and **C#**).

> Suppose a program contains three classes in an inheritance hierarchy: a superclass, `Cat`, and two subclasses, `HouseCat` and `Lion`. Class Cat defines a virtual function named `speak`, so its subclasses may provide an appropriate implementation (e.g. either `meow` or `roar`). When the program calls the speak function on a `Cat` reference (which can refer to an instance of `Cat`, or an instance of `HouseCat` or `Lion`), the code must be able to determine which implementation of the function the call should be dispatched to. This depends on the actual class of the object, not the class of the reference to it (`Cat`). The class **cannot generally be determined statically** (that is, at compile time), so neither can the compiler decide which function to call at that time. The call must be dispatched to the right function dynamically (that is, at run time) instead.

Abstract class

Abstract classes are classes that contain one or more abstract methods. An abstract method is a method that is declared, but contains no implementation. Abstract classes may not be instantiated, and require subclasses to provide implementations for the abstract methods.

Polymorphism

Oolymorphism is the **provision of a single interface to entities of different types** or the **use of a single symbol to represent multiple different types.**

The most commonly recognized major classes of polymorphism are:

- **Subtyping** (also called subtype polymorphism or inclusion polymorphism): when a name denotes instances of many different classes related by some common superclass.
- **Ad hoc polymorphism**: defines a common interface for an arbitrary set of individually specified types.
- **Parametric polymorphism**: when one or more types are not specified by name but by abstract symbols that can represent any type.

Access modifiers

All types and type members have an **accessibility level**. The accessibility level controls whether they can be used from other code in your assembly or other assemblies. Use the following access modifiers to specify the accessibility of a type or member when you declare it:

- `public`: The type or member can be accessed by any other code in the same assembly or another assembly that references it.
- `private`: The type or member can be accessed only by code in the same class or struct.
- `protected`: The type or member can be accessed only by code in the same class, or in a class that is derived from that class.
- `internal`: The type or member can be accessed by any code in the same assembly, but not from another assembly.
- `protected internal`: The type or member can be accessed by any code in the assembly in which it's declared, or from within a derived class in another assembly.
- `private protected`: The type or member can be accessed only within its declaring assembly, by code in the same class or in a type that is derived from that class.

visibility / keyword	Containing Classes	Derived Classes	Containing Assembly	Anywhere outside the containing assembly
public	yes	yes	yes	yes
protected internal	yes	yes	yes	no
protected	yes	yes	no	no
private	yes	no	no	no
internal	yes	no	yes	no

Discrete-event simulation

A **discrete-event simulation** (DES) models the operation of a system as a **(discrete) sequence of events in time**. Each event occurs at a particular instant in time and marks a **change of state** in the system. **Between consecutive events, no change in the system is assumed to occur**; thus the simulation time can directly jump to the occurrence time of the next event, which is called next-event time progression.

In addition to next-event time progression, there is also an alternative approach, called fixed-increment time progression, where time is broken up into small time slices and the system state is updated according to the set of events/activities happening in the time slice. Because not every time slice has to be simulated, a next-event time simulation can typically run much faster than a corresponding fixed-increment time simulation.

Both forms of DES contrast with continuous simulation in which the system state is changed continuously over time on the basis of a set of differential equations defining the rates of change of state variables.

State diagram (stavový diagram, diagram stavů)

A state diagram is a type of diagram used in computer science and related fields to describe the behavior of systems. State diagrams require that the system described is composed of a finite number of states; sometimes, this is indeed the case, while at other times this is a reasonable abstraction. Many forms of state diagrams exist, which differ slightly and have different semantics.

SOLID

Single-responsibility principle

> A class should only have a single responsibility, that is, only changes to one part of the software's specification should be able to affect the specification of the class.

Classy ktoré vytvárame, by nemali byť *know-it-all* classy, vo význame, že riadia strašne velkú časť programu. Dobrý indikátor toho, že classaby mala byť rozdelená je, že máme nutkanie jej dať názov napr. ako `Processor`, `Manager` atd.

Ako výsledok dostávame veľa malých class s triviálnym chovaním, čo sa môže niekomu zdať ako zbytočnosť. Je ale lepšie mať triviálnu classu a hneď chápať čo robí, ako lúštiť, čo robí nejaká iná kolosálna classa. Naviac funkcionalita má tendenciu skorej pribúdať, takže sa z triviálnych class môžu stať po čase zložitejšie, ktoré budeme musieť znova rozdeliť do menších.

Open–closed principle

> "Software entities ... should be open for extension, but closed for modification."
>
> - A module will be said to be open if it is still available for extension. For example, it should be possible to add fields to the data structures it contains, or new elements to the set of functions it performs.
> - A module will be said to be closed if [it] is available for use by other modules. This assumes that the module has been given a well-defined, stable description (the interface in the sense of information hiding).

Liskov substitution principle

> "Objects in a program should be replaceable with instances of their subtypes without altering the correctness of that program." See also design by contract.

Interface segregation principle

> "Many client-specific interfaces are better than one general-purpose interface."

Dependency inversion principle

> One should "depend upon abstractions, [not] concretions."

Delegates

A *delegate* is an object that knows how to call a method.

A *delegate type* defines the kind of method that delegate instances can call. Specifically, it defines the method's return type and its parameter types.

All delegate instances have multicast capability. This means that a delegate instance can reference not just a single target method, but also a list of target methods. The + and += operators combine delegate instances.

Test-driven development

1. Add a test
2. Run all tests and see if the new test fails
3. Write the code
4. Run tests
5. Refactor code
6. Repeat