



FRUIZIONE E UTILIZZO DEI MATERIALI DIDATTICI

- ➡ **E' vietata** la **copia**, la **rielaborazione**, la **riproduzione** dei contenuti e immagini presenti nelle lezioni in qualsiasi forma
- ➡ **E' inoltre vietata** la **diffusione**, la **redistribuzione** e la **pubblicazione** dei contenuti e immagini, incluse le registrazioni delle videolezioni con qualsiasi modalità e mezzo non autorizzati espressamente dall'autore o da Unica

Modularity

Modularity

One of the so-called 'Seven principles of software engineering'

- Rigor and Formality
- Abstraction
- Generality
- Incrementality
- **Anticipation of change**
- **Separation of concerns**
- **Modularity**

Modularity

There is a limit to the complexity that a human being can handle. *We need to divide problems into more straightforward problems.*

A complex system that can be divided into smaller parts (modules) is called **modular**. The module is a 'piece' of system that **can be considered separately**.

Advantages of modularity

- **manage and control complexity**
 - ability to break down a complex system into simpler parts (top-down)
 - ability to compose a complex system starting from existing modules (bottom-up)
- face the **anticipation of change** - we can identify the modules that we will probably modify to meet future needs
- possibility to **change a system** by modifying only a small set of its parts. In a 'monolithic' software it is difficult to **make changes**. How many parts of the code do we need to master before making a change?
- **work in groups**. In a 'monolithic' software it is not possible (or it is hard) to **share** the work with other people.

- **separation of concerns** - modularity allows us to deal with different aspects of the problem, **focusing our attention on each of them separately** (i.e. one module deals with calculating areas, another with drawing geometric figures, ...)
- write **understandable** software: we can understand the software system as a function of its parts
- isolate errors. Check the single modules one at a time, instead of checking everything to find the error.
- **reuse** one or more modules of the software. It is hard to **reuse** part of an huge script (*'non-locality'* of the code)

Modularity: Interaction between modules

Interaction between modules

Interaction between modules

1. A module modifies the data - or even the instructions (e.g. using Assembly) - that are local to another module



Interaction between modules




1. A module modifies the data - or even the instructions (e.g. using Assembly) - that are local to another module



2. A module can communicate with another module through a common data area, such as a global variable in C or in Python



Interaction between modules

1. A module modifies the data - or even the instructions (e.g. using Assembly) - that are local to another module

2. A module can communicate with another module through a common data area, such as a global variable in C or in Python 
3. A module invokes another one and transfers information using a specific *interface*. This is a traditional and disciplined way of interaction between two modules 

The **USES** relationship

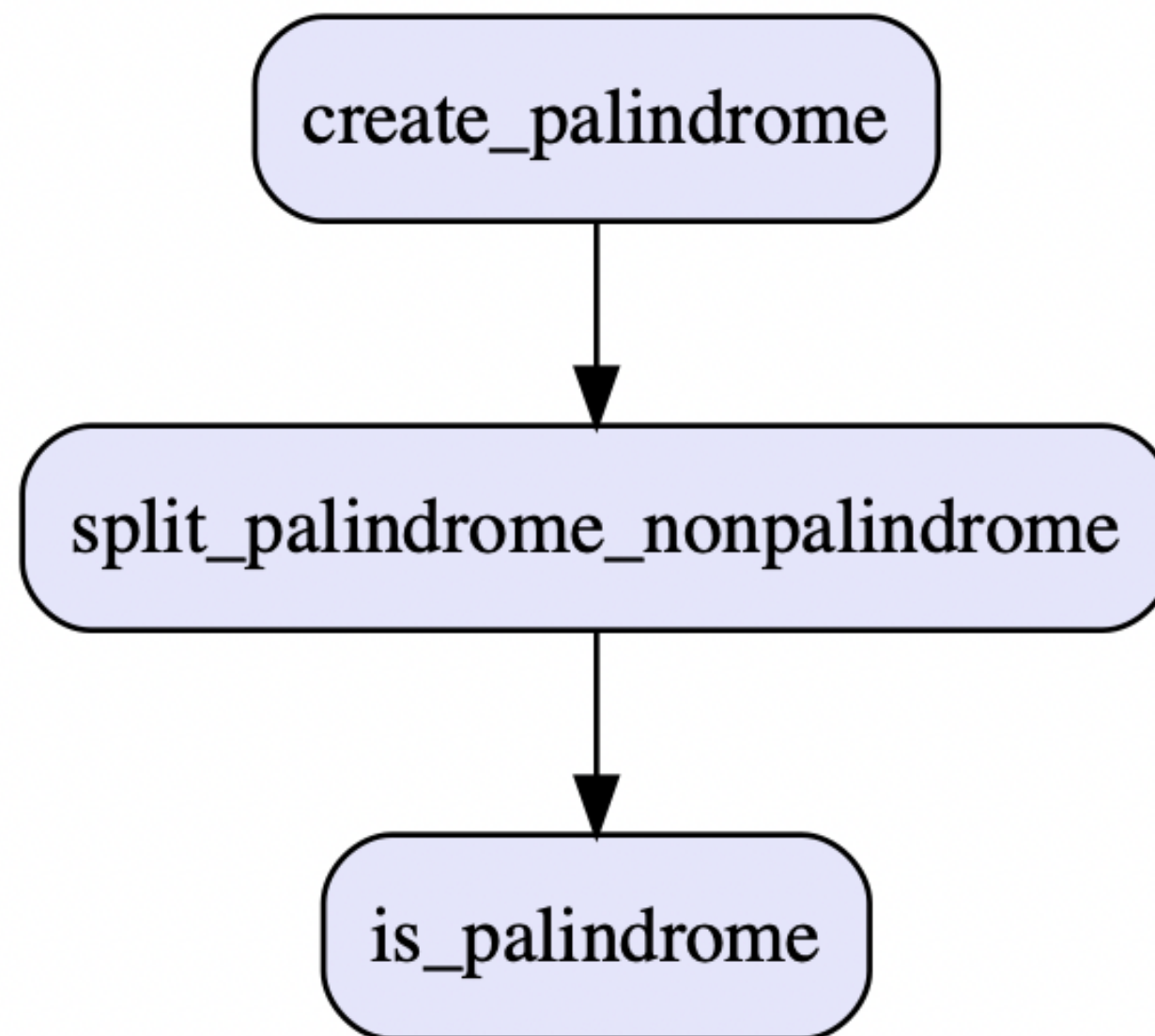
A useful relation for describing the modular structure of a software system is the so-called **USES** relation.

A USES B if A requires the presence of B to work.

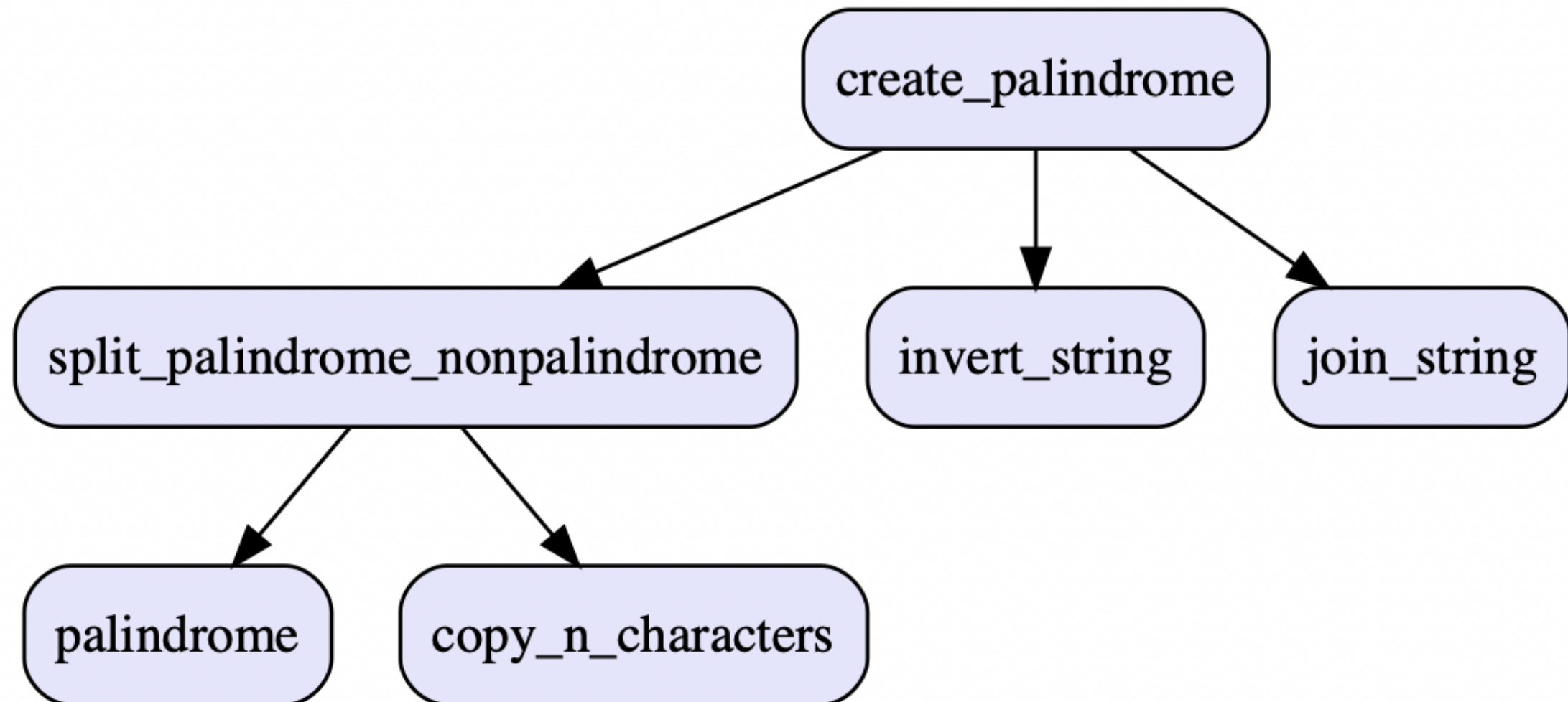
A is a **client** of *B*. *B* is a **server**.

The **USES** relationship

We can describe the solution of the *palindrome* exercise using the relation **USES**.



The same solution in c programming language.



The **USES** relationship

We can describe (and design) a library of mathematical functions using the relation **USES**.

Let's focus on the trigonometric functions.

We can implement trigonometric functions using Taylor series (only):

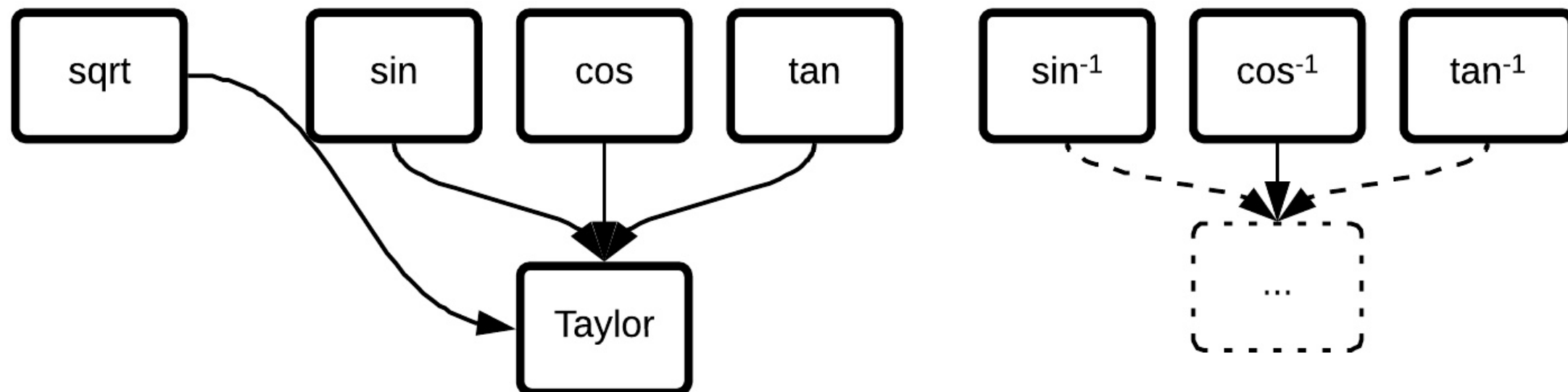
$$\sin(x) = \sum_{n=0}^{\infty} \frac{(-1)^n}{(2n+1)!} x^{2n+1}$$

$$\cos(x) = \sum_{n=0}^{\infty} \frac{(-1)^n}{(2n)!} x^{2n}$$

$$\tan(x) = x + \frac{x^3}{3} + \frac{2}{15}x^5 + o(x^6)$$

We can implement trigonometric functions using Taylor series (only):

$$\sin(x) = \sum_{n=0}^{\infty} \frac{(-1)^n}{(2n+1)!} x^{2n+1}; \quad \cos(x) = \sum_{n=0}^{\infty} \frac{(-1)^n}{(2n)!} x^{2n}; \quad \tan(x) = x + \frac{x^3}{3} + \frac{2}{15}x^5 + o(x^6)$$

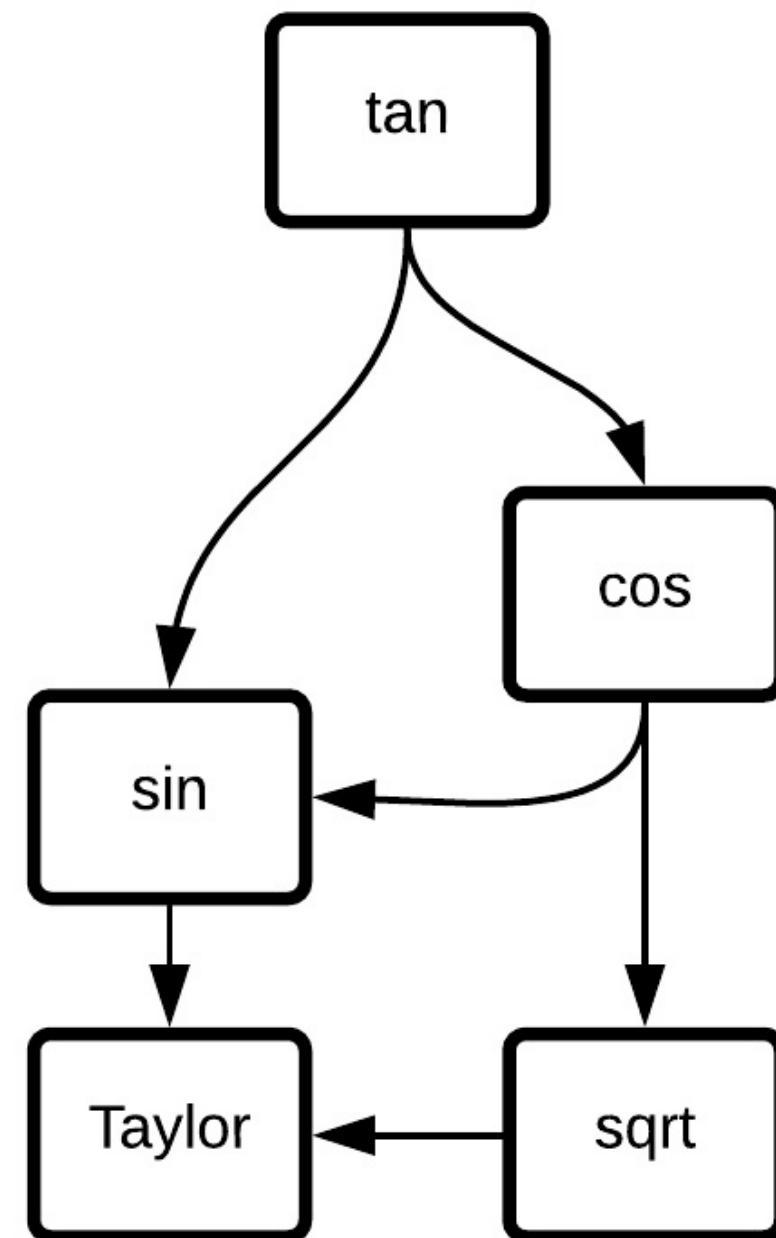


Or we can use (even) other relations between trigonometric functions:

$$\sin(x) = \sum_{n=0}^{\infty} \frac{(-1)^n}{(2n+1)!} x^{2n+1};$$

$$\cos(x) = \sqrt{1 - \sin^2(x)};$$

$$\tan(x) = \frac{\sin(x)}{\cos(x)};$$

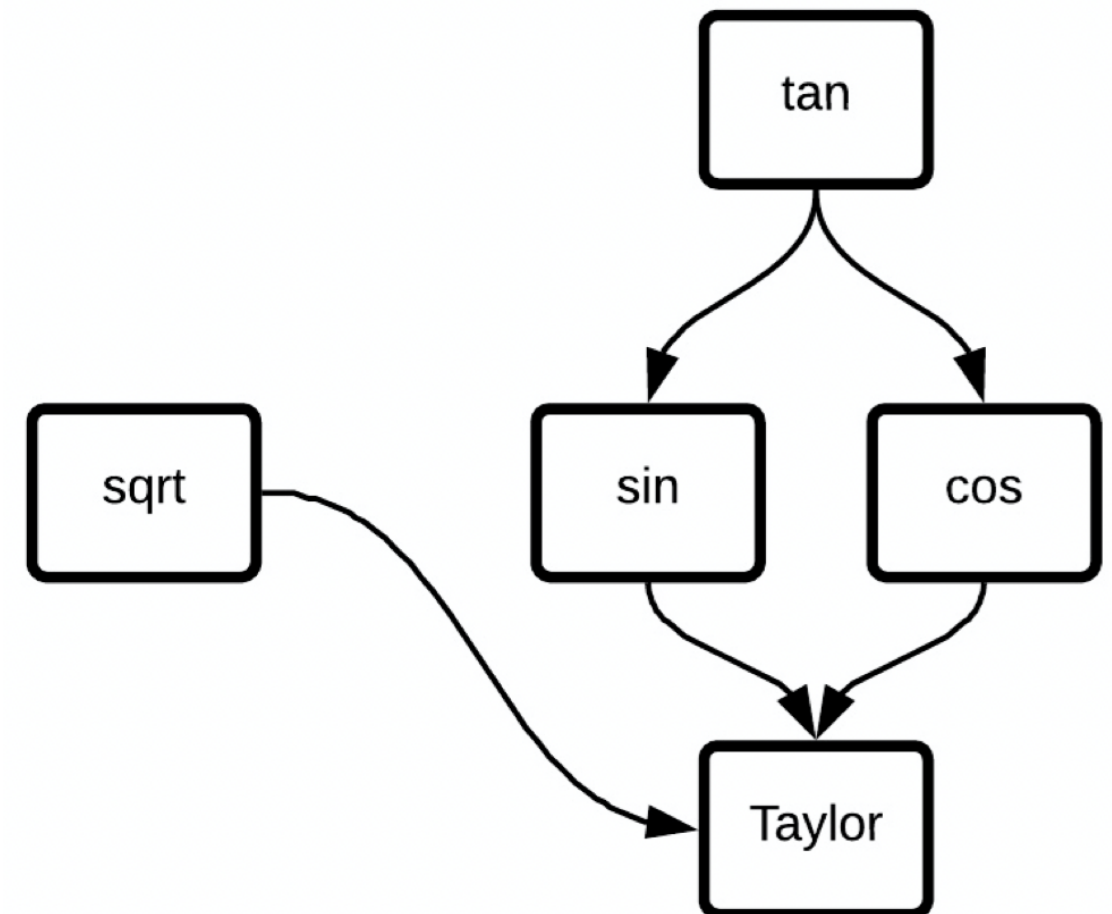


Or we can use (even) other relations between trigonometric functions:

$$\sin(x) = \sum_{n=0}^{\infty} \frac{(-1)^n}{(2n+1)!} x^{2n+1};$$

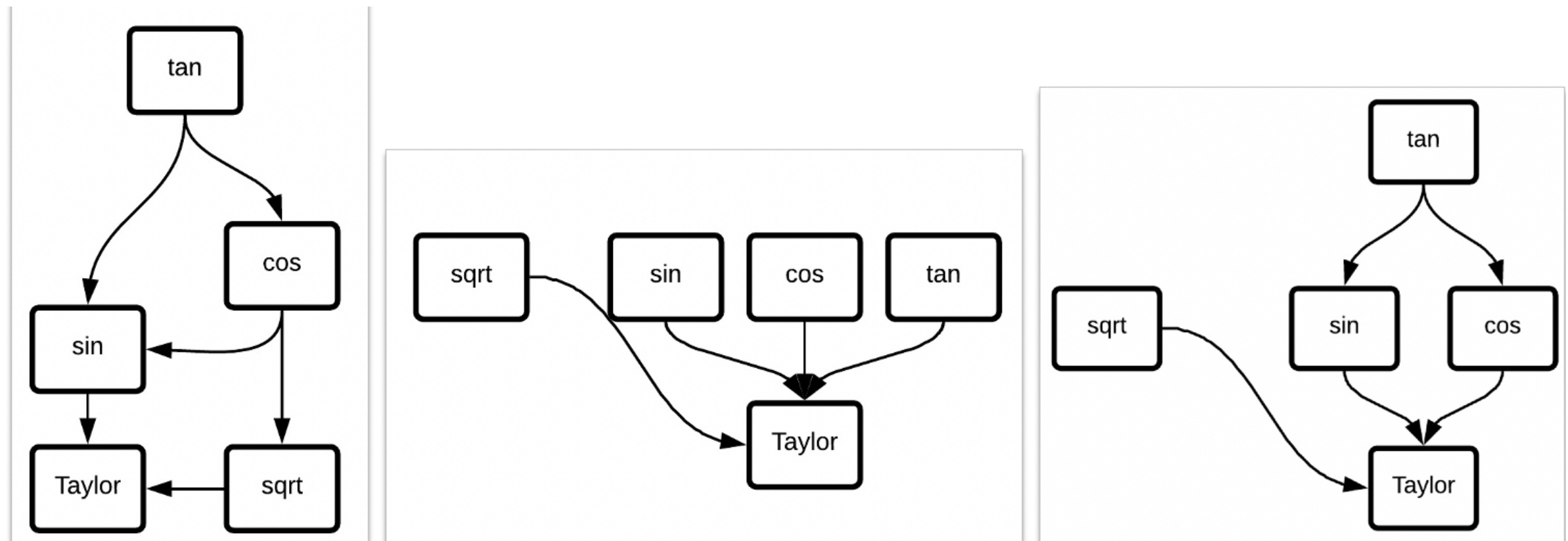
$$\cos(x) = \sum_{n=0}^{\infty} \frac{(-1)^n}{(2n)!} x^{2n};$$

$$\tan(x) = \frac{\sin(x)}{\cos(x)};$$



The **USE** relation helps us to design and to describe the software system.

The three solutions outlined below involve three different implementations.

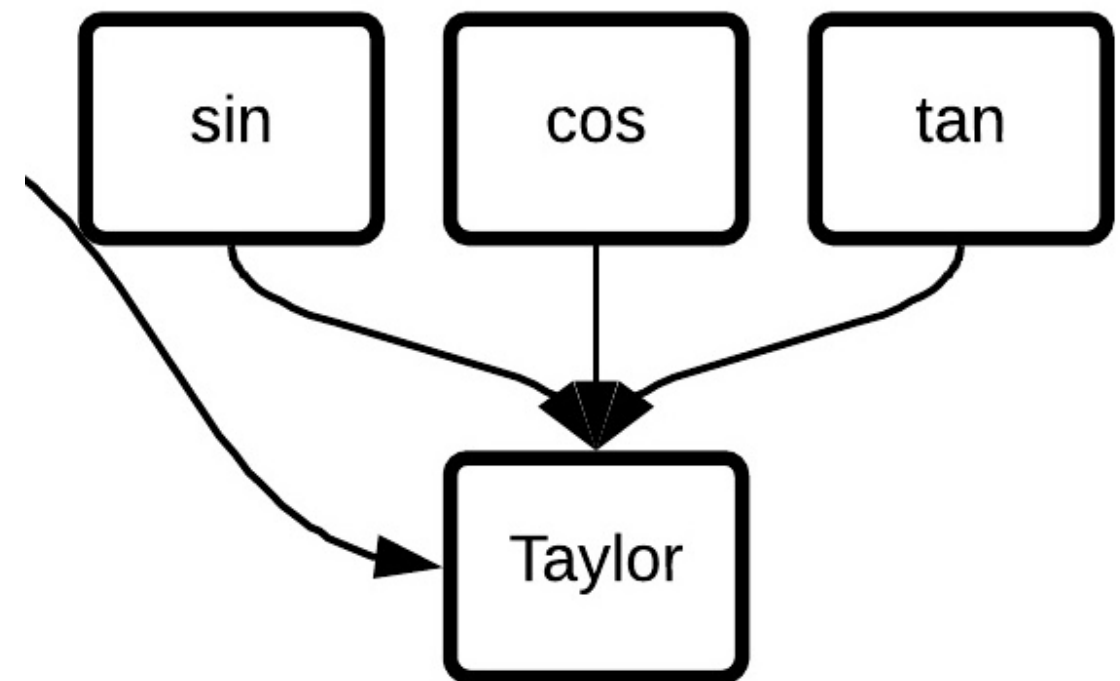


Hierarchy

We can **impose** that the **USE** relationship is **hierarchical**.

(a generic USE relationship is not necessarily hierarchical, but it is useful to add this constraint)

- Hierarchical systems are **easier to understand** than non-hierarchical ones: once the abstractions provided by the server modules are clear, clients can be understood without having to look at server implementation.

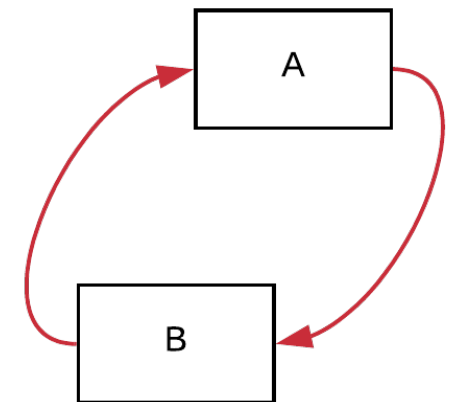
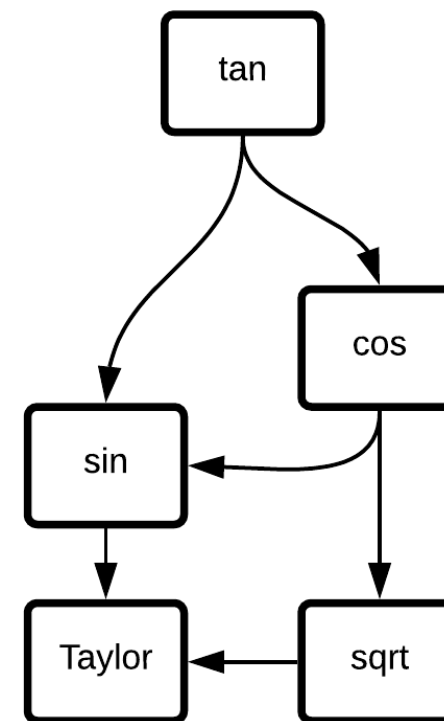


Hierarchy

We can **impose** that the **USE** relationship is **hierarchical**.

(a generic USE relationship is not necessarily hierarchical, but it is useful to add this constraint)

-
- If the structure is hierarchical,
 - we can **test at least one module independently of the others** (there is at least one module that is only SERVER and not CLIENT)
 - we can test easily the **entire system**

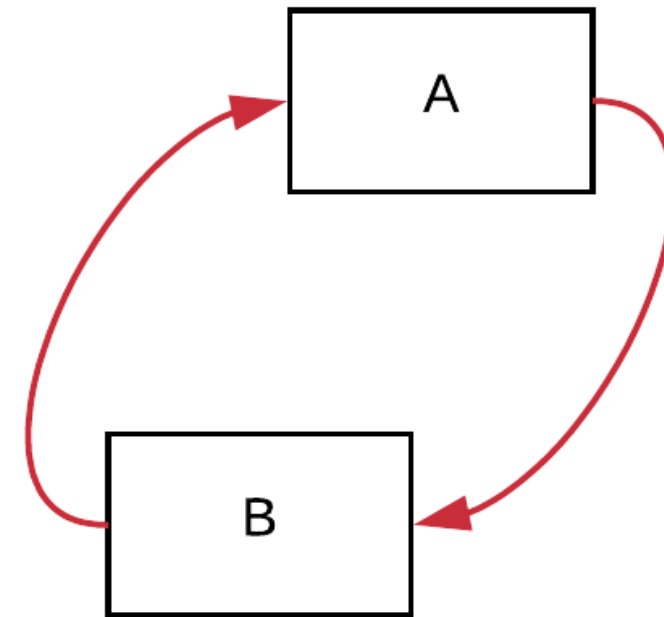


Hierarchy

We can **impose** that the **USE** relationship is **hierarchical**.

(a generic USE relationship is not necessarily hierarchical, but it is useful to add this constraint)

- If the structure is *not* hierarchical, we can have a system "where **nothing works until everything works**" [Parnas, 1979]

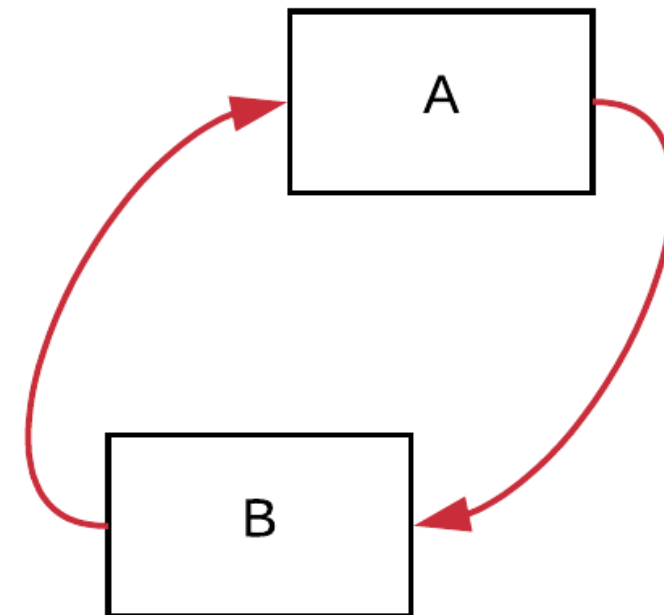


The **presence of a loop** in the **USES** relation means that **no module in the loop can be used or tested in isolation.**

For example, if

A USES B *and* B USES A

I need both A and B to run A or B. This configuration can also cause garbage collection problems.



Cohesion and coupling

Cohesion refers to the **degree to which the elements inside a module belong together**.

It is a measure of the strength of relationship between different parts of the same module.

We must collect in the same module instructions and data logically linked. These instructions and data will cooperate to achieve the module's goal.

Coupling is the degree of interdependence between software modules. It measures how closely connected two modules are. Low coupling is often a sign of a good design. Two modules have a *high coupling* if they are strictly dependent on each other.

The modules must be characterized by **high cohesion and low coupling**.

Example of **high coupling**

(the client must know something about the secret of the module)

```
def fb(i, v1, v2, flag_cond):  
    if flag_cond == 1:  
        cond_1 = is_multiple_of(i, v1)  
        cond_2 = is_multiple_of(i, v2)  
    elif flag_cond == 2:  
        cond_1 = is_greater_than(i, v1)  
        cond_2 = is_greater_than(i, v2)
```

Example of high coupling
(the client must know something about the secret of the module)

```
def fb(i, v1, v2, flag_cond):  
    if flag_cond == 1:  
        cond_1 = is_multiple_of(i, v1)  
        cond_2 = is_multiple_of(i, v2)  
    elif flag_cond == 2:  
        cond_1 = is_greater_than(i, v1)  
        cond_2 = is_greater_than(i, v2)
```

Example of **low coupling**

```
def fb(i, v1, v2, f_eval_cond):  
    cond_1 = f_eval_cond(i, v1)  
    cond_2 = f_eval_cond(i, v2)
```

INTERFACE

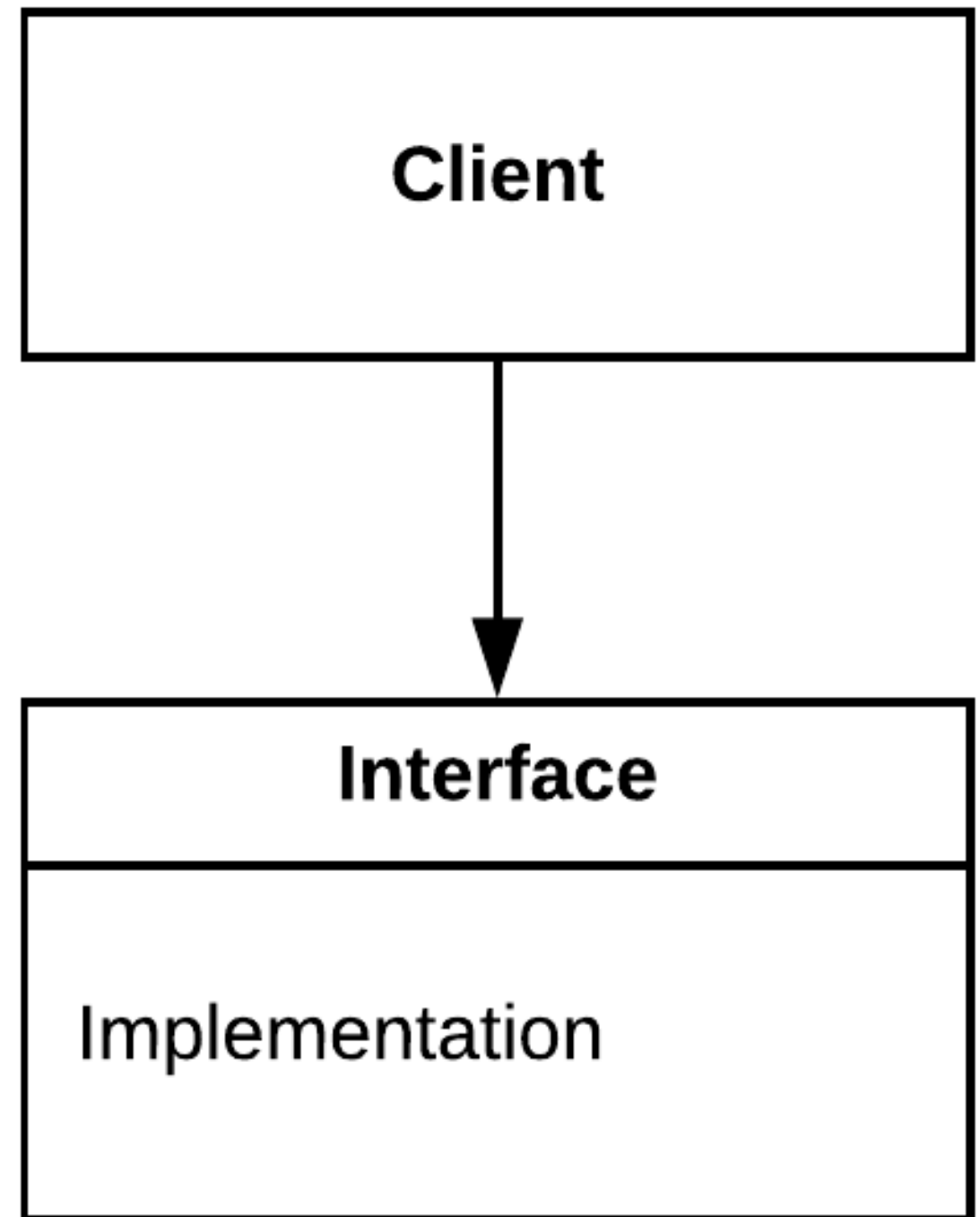
INTERFACE

Interface: set of services offered by the module.

The services are made available (exported) by the server module and imported by the clients.

Their implementation is a **secret of the module**.

The distinction between interface and implementation is a key aspect in good design.



INTERFACE

The interface is a **abstraction of the module**, hiding the details that the programmer of the **clients** must not know.

INTERFACE

The interface is a **abstraction of the module**, hiding the details that the programmer of the **clients** must not know.

- It describes the offered services, and what clients need to know to use them. The clients know the services of a module M only through its interface; **the implementation remains hidden**.

INTERFACE

The interface is a **abstraction of the module**, hiding the details that the programmer of the **clients** must not know.

- It describes the offered services, and what clients need to know to use them. The clients know the services of a module M only through its interface; **the implementation remains hidden**.
- If the interface remains the same, the module can change without causing repercussions on its clients.

INTERFACE

The interface is a **abstraction of the module**, hiding the details that the programmer of the **clients** must not know.

- It describes the offered services, and what clients need to know to use them. The clients know the services of a module M only through its interface; **the implementation remains hidden**.
- If the interface remains the same, the module can change without causing repercussions on its clients.
- Who writes clients must know only the server interface, and can (*should*) ignore the implementation.

INTERFACE

The interface is a **abstraction of the module**, hiding the details that the programmer of the **clients** must not know.

- It describes the offered services, and what clients need to know to use them. The clients know the services of a module M only through its interface; **the implementation remains hidden**.
- If the interface remains the same, the module can change without causing repercussions on its clients.
- Who writes clients must know only the server interface, and can (*should*) ignore the implementation.
- It is possible to use and test the module as a black box.

INTERFACE

The interface is a **abstraction of the module**, hiding the details that the programmer of the **clients** must not know.

- It describes the offered services, and what clients need to know to use them. The clients know the services of a module M only through its interface; **the implementation remains hidden**.
- If the interface remains the same, the module can change without causing repercussions on its clients.
- Who writes clients must know only the server interface, and can (*should*) ignore the implementation.
- It is possible to use and test the module as a black box.
- The code can be reused more easily.

EXERCISE

Write a software that can **store a time interval**. The software must provide the data structure to store the time interval, and the 4 functions `set_h_min()`, `set_min()`, `get_h_min()`, `get_min()`

The data structure can be designed in two different ways (choose only one of them)

- it can store hours and minutes
- it can store minutes

Implement the software using only functions and dictionaries, for now. No OOP is required. Key point: the interface must not change if you change the implementation. *If the implementation changes, the main continues to work without changes.*

The functions and data structures you write will work in this way:

```
t1 = create_time_slot() # t1 is a dictionary
```

```
set_h_m(t1, 2, 20)
print(get_m(t1))      # Expected value: 140
print(get_h_m(t1))    # Expected value: 2, 120
```

```
set_m(t1, 140)
print(get_m(t1))      # Expected value: 140
print(get_h_m(t1))    # Expected value: 2, 120
```

(NO OOP solution: timeslot_1.py)

The functions and data structures you write will work in this way (OOP):

```
t1 = TimeSlot() # t1 is an object
```

```
t1.set_h_m(2, 20)
print(t1.get_m()) # Expected value: 140
print(t1.get_h_m()) # Expected value: 2, 120
```

```
t1.set_m(140)
print(t1.get_m()) # Expected value: 140
print(t1.get_h_m()) # Expected value: 2, 120
```

(OOP solution: timeslot_2.py)