# Modularity

Instructors **Battista Biggio**, **Angelo Sotgiu** and **Leonardo Regano**

M.Sc. in Computer Engineering, Cybersecurity and Artificial Intelligence
University of Cagliari, Italy

# Modularity

## Lesson outline

- Definition and advantages of modularity
- "Uses" relationship
- Cohesion and coupling
- Interfaces
- Encapsulation, Abstraction, Information Hiding
- "Is component of" relationship

# Modularity

Modularity is a fundamental building block in the development of non-trivial software, a process driven not by a single developer but by the collaborative effort of a team

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.

# Advantages of modularity

- **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

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**.

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

- 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.

# Example

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

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

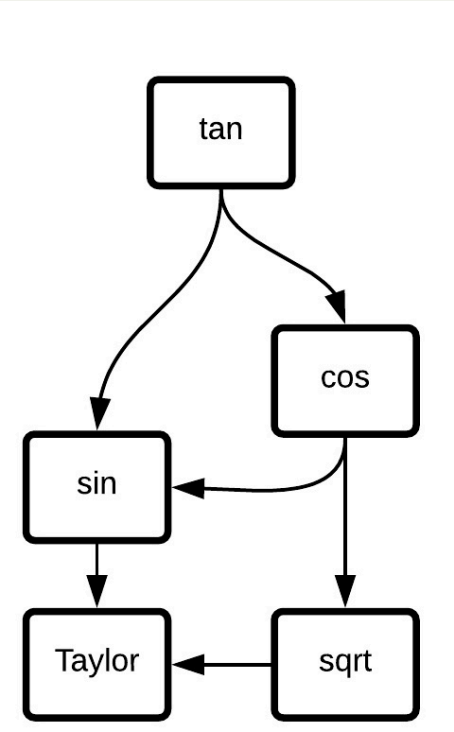
$$\cos(x) = \sum_{n=0}^{\infty} \frac{(-1)^n}{(2n)!} x^{2n}$$

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

# Hierarchy

If the structure is hierarchical (*i.e.*, no cycles):

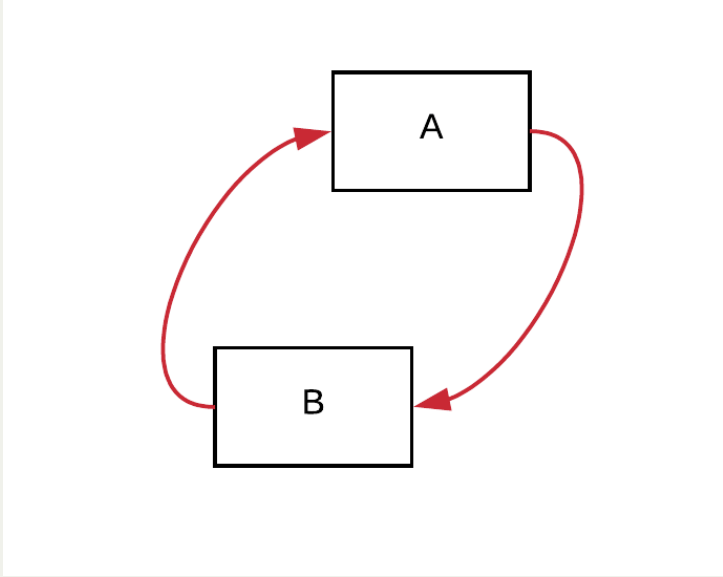
- we can **test at least one module independently of the others** (there is at least one module that is only SERVER and not CLIENT)
- we can test easily the **entire system**



# Hierarchy

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

- a generic USE relationship is not necessarely 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.