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):

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

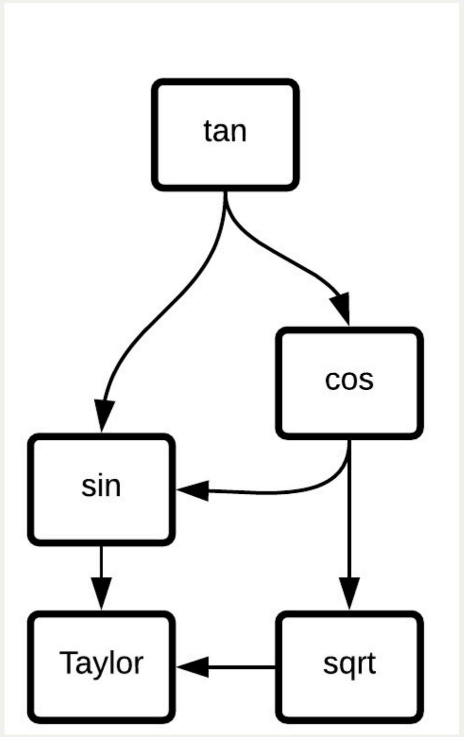
$$\cos(x) = \sum_{n=0}^{\infty} \frac{(-1)^n}{(2n)!} x^{2n}$$

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

Hierarchy

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

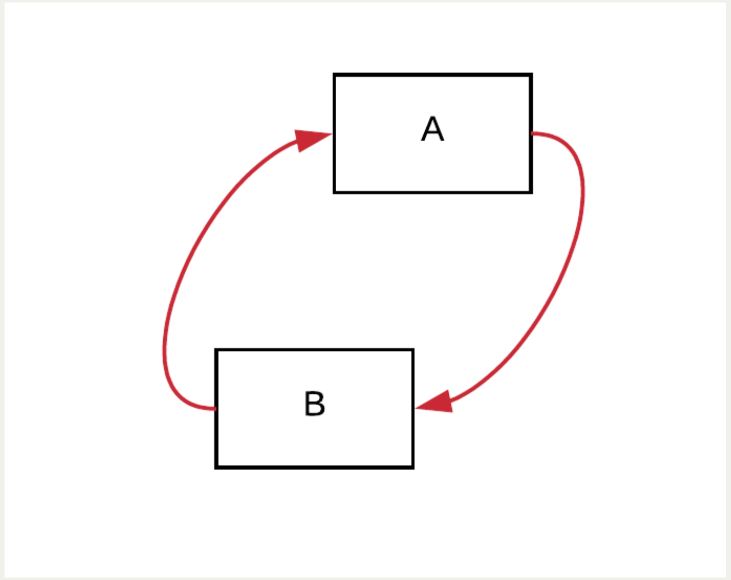
- we can **test at least one module independently of the others** (there is at least one module that is only SERVER and not CLIENT)
- we can test easily the **entire system**



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]

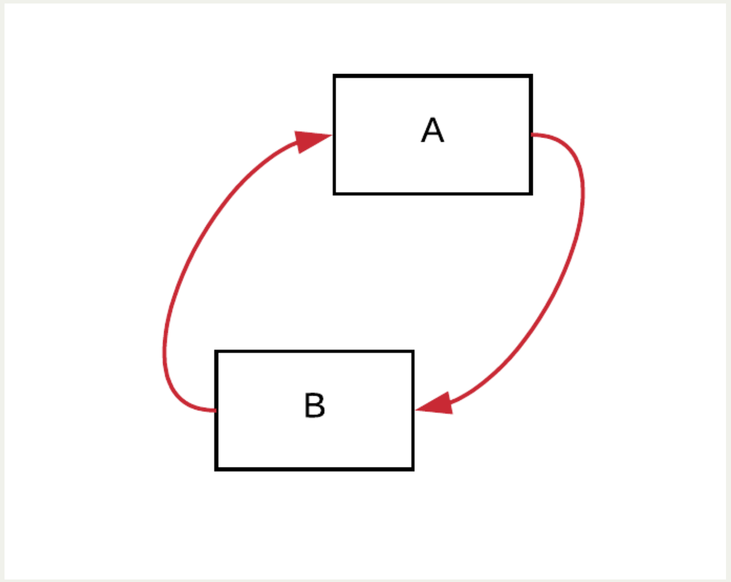


The **presence of a loop** in the **USES** relation means that **no module in the loop can be used or tested in isolation**.

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.



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 the 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.

Cohesion and coupling

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**.

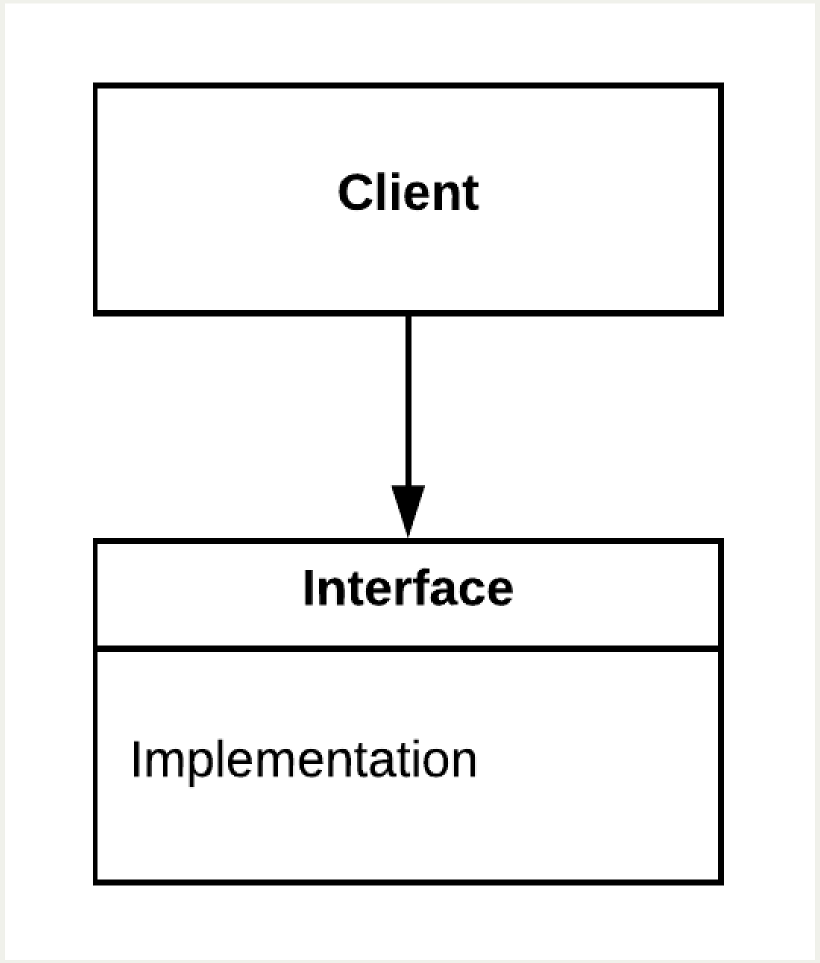
Interface

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 of good design.



Interface

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

- It describes the offered services, and what clients must understand to utilize them. Clients gain knowledge of a module's services solely through its interface; **the implementation remains hidden**.
- If the interface remains unchanged, the module can change without affecting its clients.
- Those developing clients need only know the server interface and can (*and should*) ignore the implementation.
- It is possible to employ and test the module as a black box.
- The code becomes more easily reusable.

Interface

What should be **exposed** by the interface and what should be **concealed** within the implementation?

The interface should disclose the minimum necessary information, providing enough for other modules to utilize the services offered.

- Revealing unnecessary details:
  - Adds unnecessary complexity to the interface
  - Reduces the comprehensibility of the system
  - Increases the likelihood that changes in implementation affect the interface and its clients

Encapsulation, Abstraction, Information Hiding

Abstraction - The interface is an **abstraction of the module**. The knowledge of the interface is sufficient to use the module.

- It is not *necessary* to know anything else.

Encapsulation, Abstraction, Information Hiding

Encapsulation - Encapsulation is used to **hide** the details, implementation, and state of a module (**the secret of a module**). The client cannot access anything beyond the interface. Encapsulation **prevents access** to details that are not necessary.

- It is not *possible* to know anything else.

Encapsulation, Abstraction, Information Hiding

Information Hiding $\Longrightarrow$ Abstraction + Encapsulation

NB: these definitions are not univocally accepted by all authors - See[1]

---

[1] Stevens, Perdita, and Rob J. Pooley. Using UML: software engineering with objects and components. Pearson Education, 2006.

Example of **high coupling**

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

Generalize the FIZZ BUZZ exercise.

The client can choose to use the `is_multiple_of()` or `is_greater_than()` function

```python
def is_multiple_of(n, d):
  return n % d == 0

def is_greater_than(a, b):
  return a > b

...

# CLIENT

# use is_multiple_of
fb(...)

# use is_greater_than
fb(...)
```

```python
def is_multiple_of(n, d):
  return n % d == 0

def is_greater_than(a, b):
  return a > b

def fb(i, div1, div2, selector):
  if selector == 0:
    cond1 = is_multiple_of(i, div1)
    cond2 = is_multiple_of(i, div2)
  elif selector == 1:
    cond1 = is_greater_than(i, div1)
    cond2 = is_greater_than(i, div2)
  if cond1 and cond2:
    print("fizzbuzz")
    elif cond1:
        print("fizz")
    elif cond2:
        print("buzz")
    else:
        print(i)
```

```python
# client
div1 = 3
div2 = 5
selector = 1
for i in range(10):
  fb(i, div1, div2, selector)
```

```python
def is_multiple_of(n, d):
  return n % d == 0

def is_greater_than(a, b):
  return a > b

def fb(i, div1, div2, selector):
  if selector == 0:
    cond1 = is_multiple_of(i, div1)
    cond2 = is_multiple_of(i, div2)
  elif selector == 1:
    cond1 = is_greater_than(i, div1)
    cond2 = is_greater_than(i, div2)
  if cond1 and cond2:
    print("fizzbuzz")
    elif cond1:
        print("fizz")
    elif cond2:
        print("buzz")
    else:
        print(i)
```

```python
# client
div1 = 3
div2 = 5
selector = 1
for i in range(10):
  fb(i, div1, div2, selector)
```

The modules exhibit tight coupling (**control coupling**).

Clients utilizing the `fb()` function must understand its implementation details for proper usage, including the need to **set the selector to the correct value**: 0 for the first condition and 1 for the second.
Ideally, these details should remain internal to the module (a secret of the module).

```python
def is_multiple_of(n, d):
  return n % d == 0

def is_greater_than(a, b):
  return a > b

def fb(i, div1, div2, selector):
  if selector == 0:
    cond1 = is_multiple_of(i, div1)
    cond2 = is_multiple_of(i, div2)
  elif selector == 1:
    cond1 = is_greater_than(i, div1)
    cond2 = is_greater_than(i, div2)
  if cond1 and cond2:
    print("fizzbuzz")
    elif cond1:
        print("fizz")
    elif cond2:
        print("buzz")
    else:
        print(i)
```

```python
# client
div1 = 3
div2 = 5
selector = 1
for i in range(10):
  fb(i, div1, div2, selector)
```

The designer is responsible for notifying all clients using the `fb()` function about any changes made to the implementation.

The introduction of a new condition requires not only creating a new function but also modifying the `fb()` function by inserting a new branch in the conditional structure.

Exercise

Write a software that manages a time interval, providing a data structure and four functions:

`set_h_min()`, `set_min()`, `get_h_min()`, `get_min()`

Choose one of the following implementations:

1. store hours and minutes separately (*e.g.* 1:20)
2. store the total amount of minutes (*e.g.*, 80 minutes)

Implement the software using only data and functions, avoiding the use of OOP for now.

Key point: Ensure that the interface remains the same regardless of the chosen implementation. Any changes to the implementation should not impact the interface.

Interface (an attempt):

```
def create_time_slot( ):
  """generates the data structure"""
  ...

def get_m( ):
  """returns a string representing the total amount of minutes"""
  ...

def get_h_m( ):
  """returns a tuple representing  (hours, minutes)"""
  ...

def set_m( ):
  """initializes the data structure with the total amount of minutes"""
  ...

def set_h_m( ):
  """initialize the data structure with (hours, minutes)"""
  ...
```

Other possibilities:

- `set` can initialize the data structure provided as input
  Example: `set_m(t, 100)`
- or can return the initialized structure
  Example: `t = set_m(100)` -> in this case, `create_time_slot` is not necessary.

```python
# CLIENT - an example

t1 = create_time_slot()
# You must create a time slot "object" before using set and get

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

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

Implementation 1: store hours and minutes separately

```python
minutes_in_hour = 60

def create_time_slot(h=0, m=0):
  time_slot = {"h": h, "m": m}
  return time_slot

def set_h_m(time_slot, h, m):
  time_slot["h"] = h
  time_slot["m"] = m

def set_m(time_slot, m):
  time_slot["h"] = int(m / minutes_in_hour)
  time_slot["m"] = m % minutes_in_hour

def get_h_m(time_slot):
  return time_slot["h"], time_slot["m"]

def get_m(time_slot):
  return time_slot["h"] * minutes_in_hour + time_slot["m"]

# CLIENT
t1 = create_time_slot()
```

The client has the flexibility to define either the pair (hours, minutes) or the total minutes.
In both scenarios, hours and minutes are stored independently.

The manner in which the data is stored within the structure is an **implementation detail**, including the names of dictionary keys or the decision to use a dictionary. Alternative structures such as lists could be employed.

These implementation details can be modified without altering the interface, and, therefore, without disrupting the client's functionality.

We can adopt an alternative implementation, with the important note that the interface (and the client) remain unaltered.

Implementation 2 - store the total amount of minutes

```python
minutes_in_hour = 60

def create_time_slot(h=0, m=0):
  tot_min = h * minutes_in_hour + m
  time_slot = {"minutes": tot_min}
  return time_slot

def set_time_slot_h_m(time_slot, h, m):
  tot_min = h * minutes_in_hour + m
  time_slot["minutes"] = tot_min

def set_m(time_slot, m):
  time_slot["minutes"] = m

def get_time_slot_h_m(time_slot):
  tot_min = time_slot["minutes"]
  hours = int(tot_min / minutes_in_hour)
  minutes = tot_min - hours * minutes_in_hour
  return hours, minutes

def get_time_slot_m(time_slot):
  return time_slot["minutes"]
```

The **IS_COMPONENT_OF** relationship

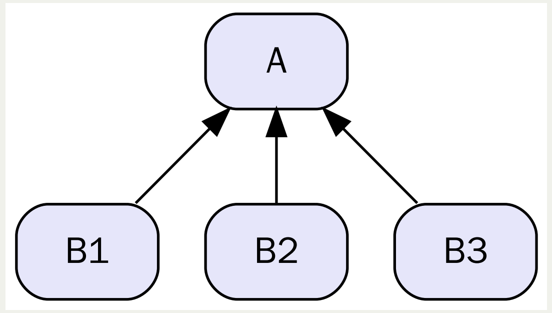
It describes an architecture in terms of a **module that is composed of other modules**

B **IS_COMPONENT_OF** A

A is formed by aggregating several modules, one of which is B

B_1, B_2, B_3 modules **implement** A

The relationship is not reflective and constitutes (**ALWAYS**) a hierarchy.



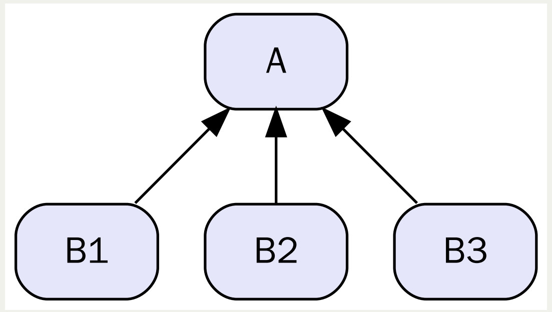
The **IS_COMPONENT_OF** relationship

The B_i modules provide all the services that should be provided by A

Once that A is decomposed into the set of B_1, B_2, B_3, we can replace A. **The module A is an *abstraction* implemented in terms of simpler abstractions.**

The only reason to keep A in the modular description of a system is that it makes the project clearer and more understandable.

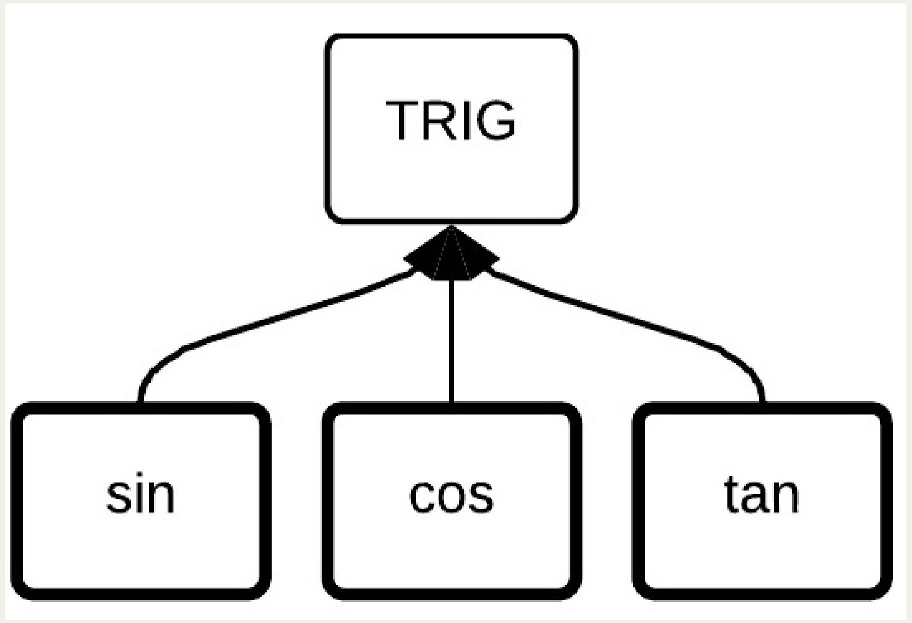
At the end of the decomposition process **only modules not made up of other modules are 'real components' of the system**. The others modules are kept only for descriptive reasons.



Example - IS_COMPONENT_OF

The entire software system is ultimately composed of modules

`sin( )`, `cos( )`, `tan( )`

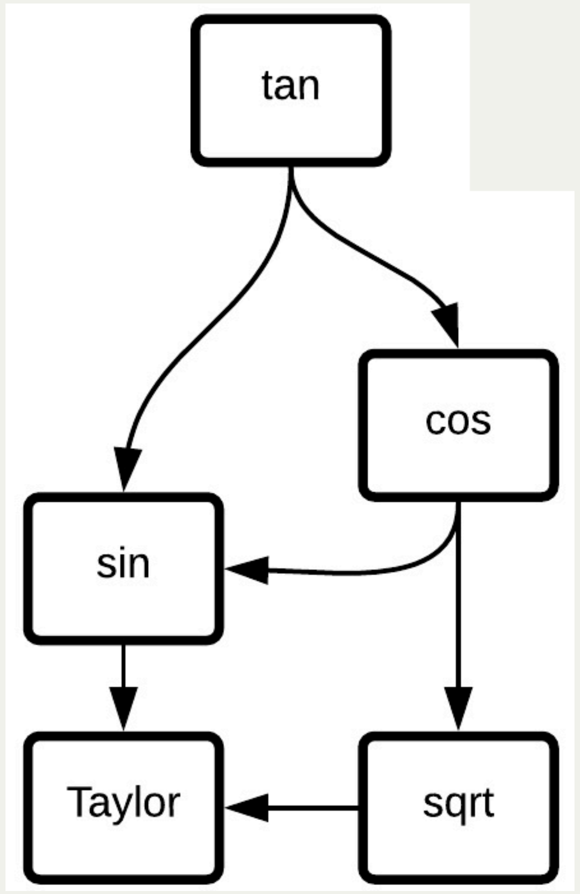


USE and IS_COMPONENT_OF

The two relations **USES** and **IS_COMPONENT_OF** can be used together (*on different graphs*) to provide alternative and complementary views of the same design.

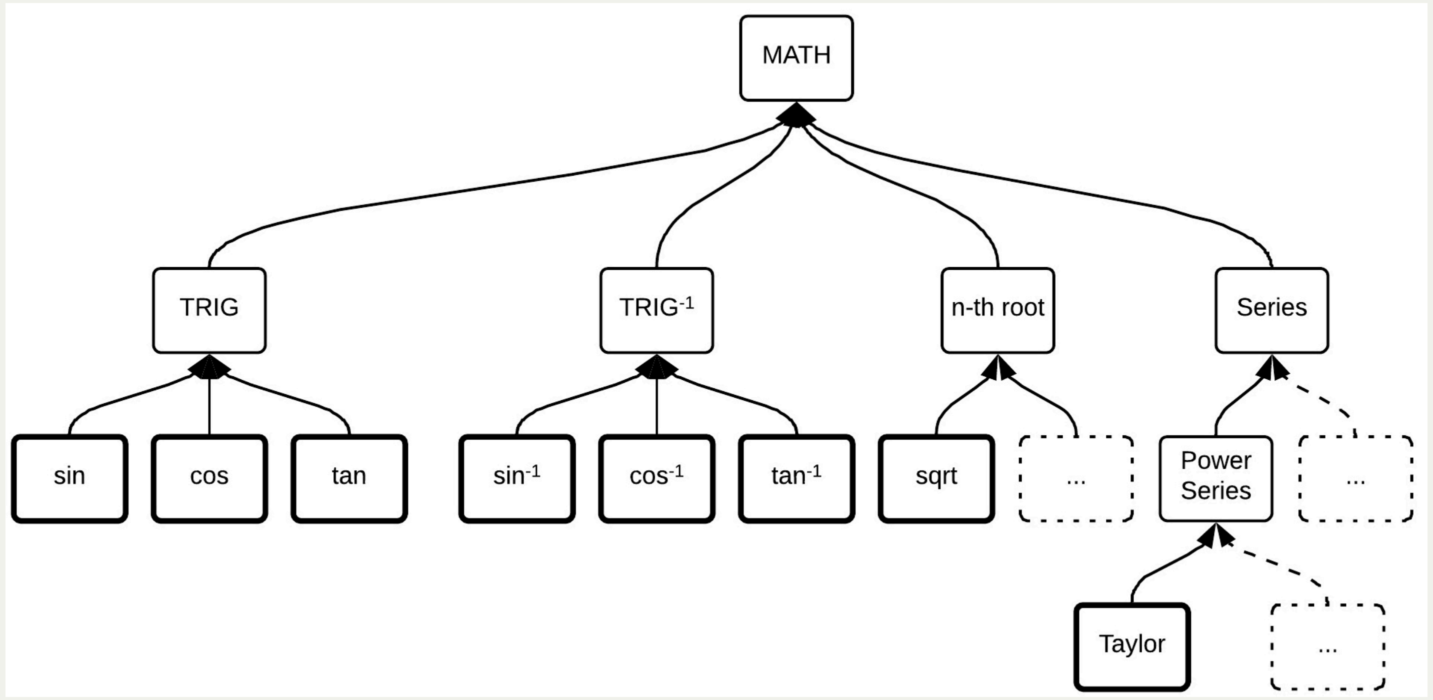
We can describe our math library using both the relations.

USE



IS_COMPONENT_OF

We can describe our math library using the relation **IS_COMPONENT_OF**. The modules with **bold border** are the only ones to be really implemented. The other modules represent an abstraction.


MATH
TRIG
TRIG^{-1}
n-th root
Series
sin
cos
tan
sin^{-1}
cos^{-1}
tan^{-1}
sqrt
...
Power Series
...
Taylor
...