Seven principles of software engineering

> [*Seven Samurai, 1954, Akira Kurosawa*]

Seven principles of software engineering

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

Anticipation of change

Change is unavoidable in software systems.

- user requirements may not be fully understood in the initial phase of the project
- users and environments change
- we want / we have to improve the software

We need to identify

- changes that will probably happen in the near future
- plan for change

Example and discussions

Request: write a calculator

Minimal version: implement addition and multiplication on integer numbers.

Anticipation of change:

design the software so that you can add new operations and new data types - if necessary - **without changing the old code (or changing it as little as possible)**

Possible changes:

- the user could ask to manipulate real numbers, complex numbers, time measurements, angles in degrees and in radians, matrices...
- the user could ask to implement subtractions, divisions, powers, linear algebra...

Your design must allow you to **add** these new features **without changing the old code (or changing it as little as possible)**

(we'll see later the **open-close principle**)

What you do not have to do

Do not implement what was not required!

The user asked you for a program to sum and multiply integer numbers! (you will be paid only for this!)

Do not fall into temptation to implement the sum of complex numbers, matrices, and so on!

You may find yourself facing unespected problems as you try to design software that calculates integrals of complex matrix functions, or software that implements the whole body of linear algebra!

How are 'good' systems made?

There is a limit to the complexity that a human being can handle. We can not have everything under control.

Small systems managed by one person are examples of *'heroic programming'*. We can not afford it.

I often see software made by one huge script. - A single script must be divided into **functions**. - The functions must be small.

How are 'good' systems made?

There is a limit to the complexity that a human being can handle. We can not have everything under control.

Small systems managed by one person are examples of *'heroic programming'*. We can not afford it.

I often see software made by one huge script. - A single script must be divided into **functions**. - The functions must be small. - **They must be about half of what you are thinking.** :wink:

[.build-lists: true]

Problems that arise from using a single huge script.

- We can not have everything under control.
- It is difficult to make changes. How many parts of the code do we need to master before making a change? (*'non-locality'* of the code).
- It is hard to reuse part of this huge script (*'non-locality'* of the code)
- If the code does not work, we have to check *everything* to find the error.
- It is not possible (or it's very hard) to share the work with other people.
- It is hard to understand the code written by others. (*Or by youself a month ago. Or maybe a week ago. Or maybe yesterday...*)
- ...

Modularity

Modularity

A complex system that can be divided into smaller parts (modules) is called **modular**.

Our system becomes a collection of modules. The module is a 'piece' of system that **can be considered separately**.

Two different approaches:

- Bottom-up: first we design the modules, then we create the composition
- Top-down: first we break down problem into small pieces (subproblems), then we design a module for each subproblem

---

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**
- **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, ...)
- **work in groups**

Advantages of modularity

- write **understandable** software: we can understand the software system as a function of its parts
- ability to **change a system** by modifying only a small set of its parts
- isolate errors. Check single modules
- reuse part (one or more modules) of the software
- ...

Interaction between modules

1. A module modifies the data - or even the instructions (e.g. using Assembly) - that are local to another module :scream: :scream:
2. A module can communicate with another module through a common data area, such as a global variable in C or in Python :weary:
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 :smile:

Relaction between modules

Let S be a software system composed of n modules.

$$S = \{M_1, M_2, \ldots, M_n\}$$

A binary relation r on S is a subset of the Cartesian product $S \times S$

$$< M_i, M_j > \in r \text{ , or } M_i \; r \; M_j$$

The r relationships taken into consideration are **not reflective** (a module can not be in relation with itself)

Relaction between modules

The relation r^+ is the **transitive closure**[1] of r on S. r^+ is also a relation on S.

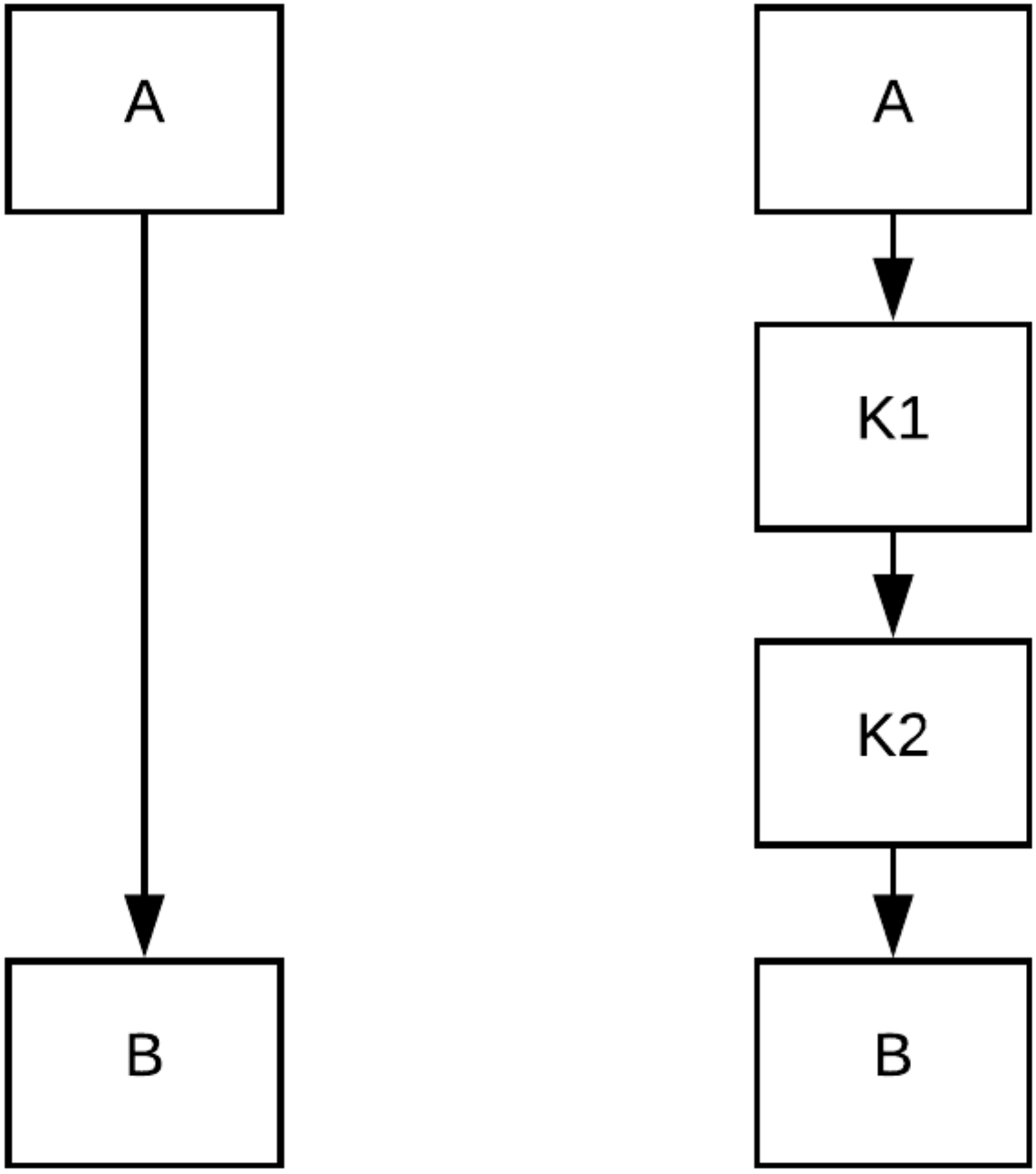
We can define r^+ recursively: considering two modules M_i and $M_j \in S$,

$$M_i \; r^+ \; M_j \iff M_i \; r \; M_j$$

OR there is a module $M_k \in S$ st.

$$M_i \; r \; M_k \text{ and } M_k \; r^+ \; M_j$$

A **transitive closure** corresponds to the intuitive notion of **direct and indirect relation**. $A \; r^+ \; B$ implies that $A \; r \; B$ directly OR indirectly through a chain of relations.

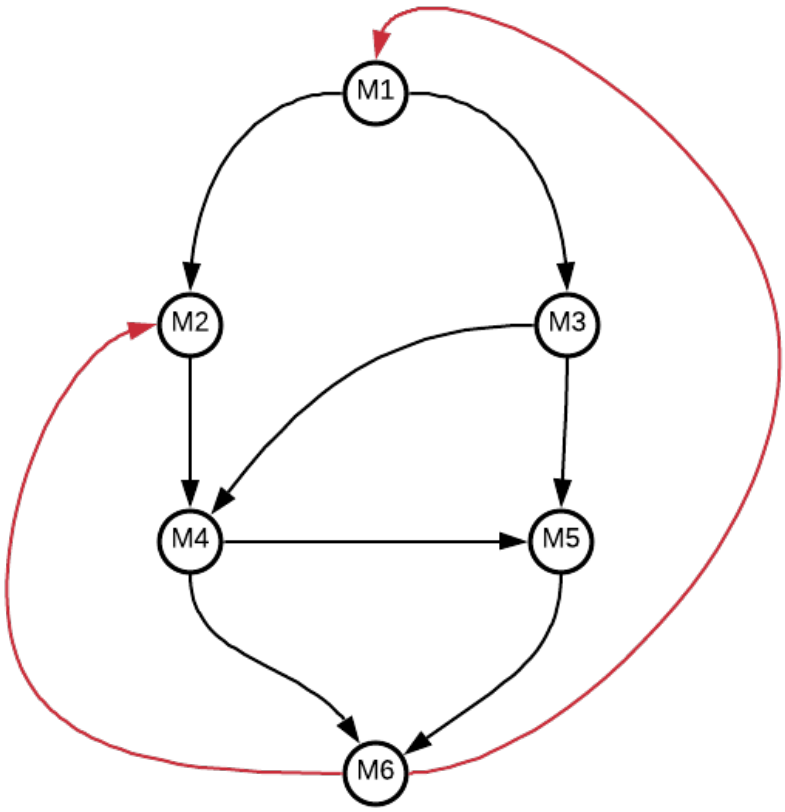


[fit]Representation by oriented graphs

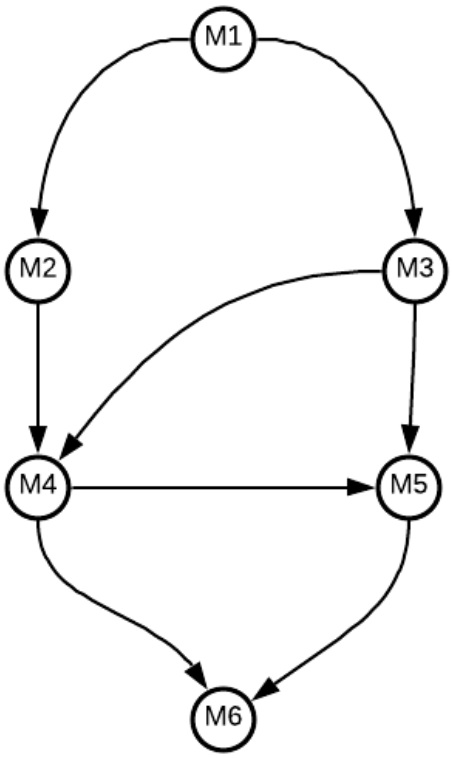
We can represent the modules as the nodes of a graph.

There is a **directed arc** (or directed edge) between M_i and M_j if and only if $M_i \; r \, M_j$

A relation is **hierarchical** if and only if **there are no** cycles in the graph $\Rightarrow$ **DAG, Directed Acyclic Graph**



CYCLES



DAG

The USES relation

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.

Module A is a **client** of B, because A requires a service provided by B. B is called **server**.

eg. A USES B if A calls a procedure/function contained in B, or if A makes use of a type defined in B.

The USES relation

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):

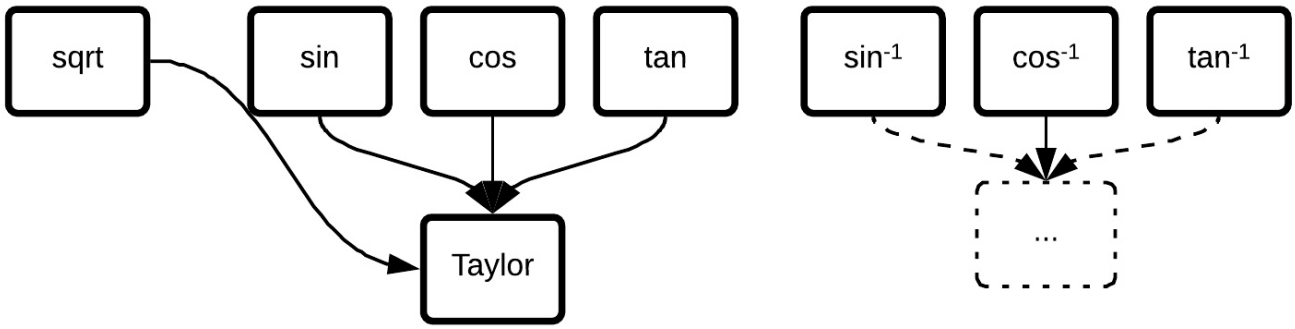
$$\sin(x) = \sum_{n=0}^{\infty} \frac{(-1)^n}{(2n+1)!} x^{2n+1}$$

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

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

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:

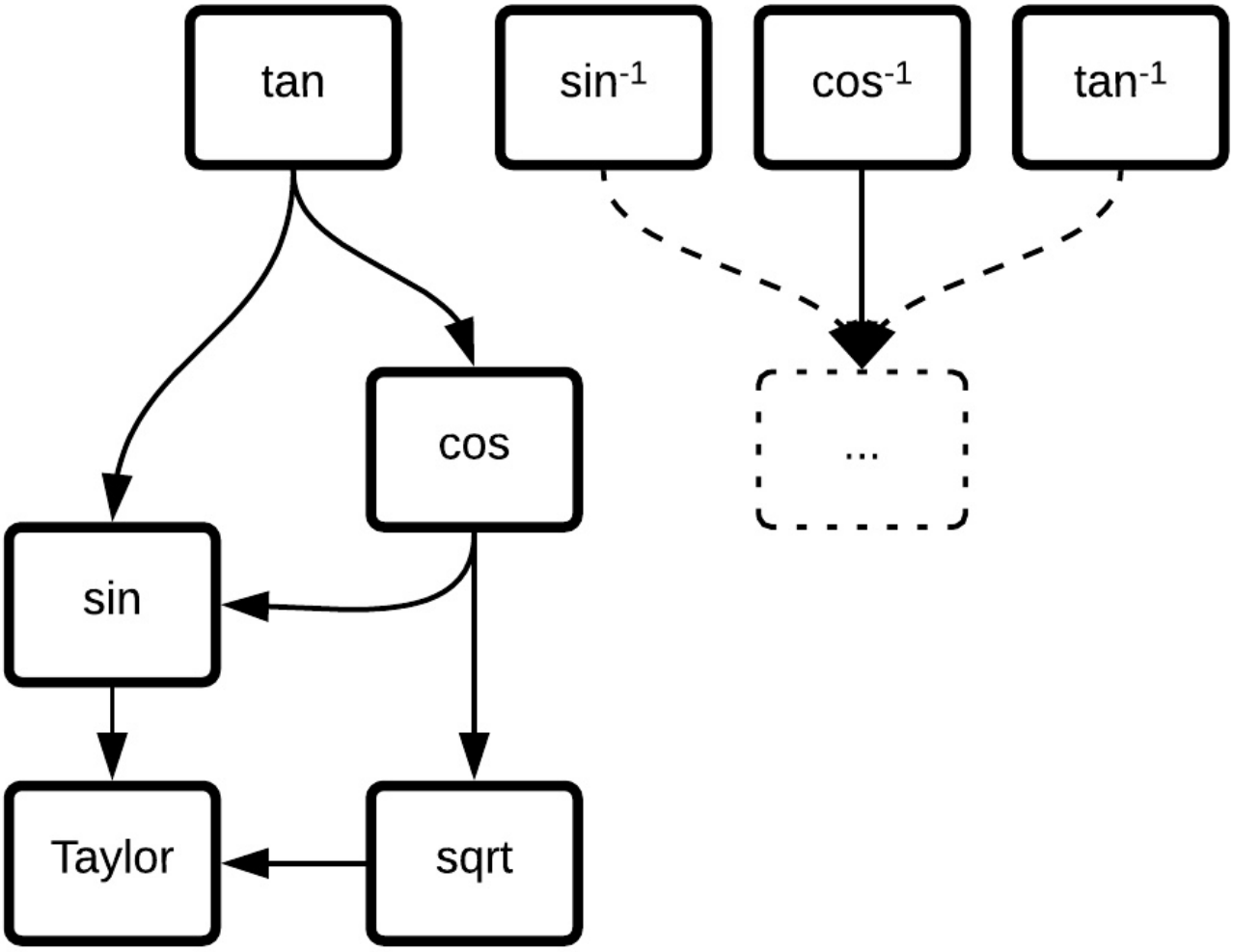
$$\sin(x) = \sum_{n=0}^{\infty} \frac{(-1)^n}{(2n+1)!} x^{2n+1}$$

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

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

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

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



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

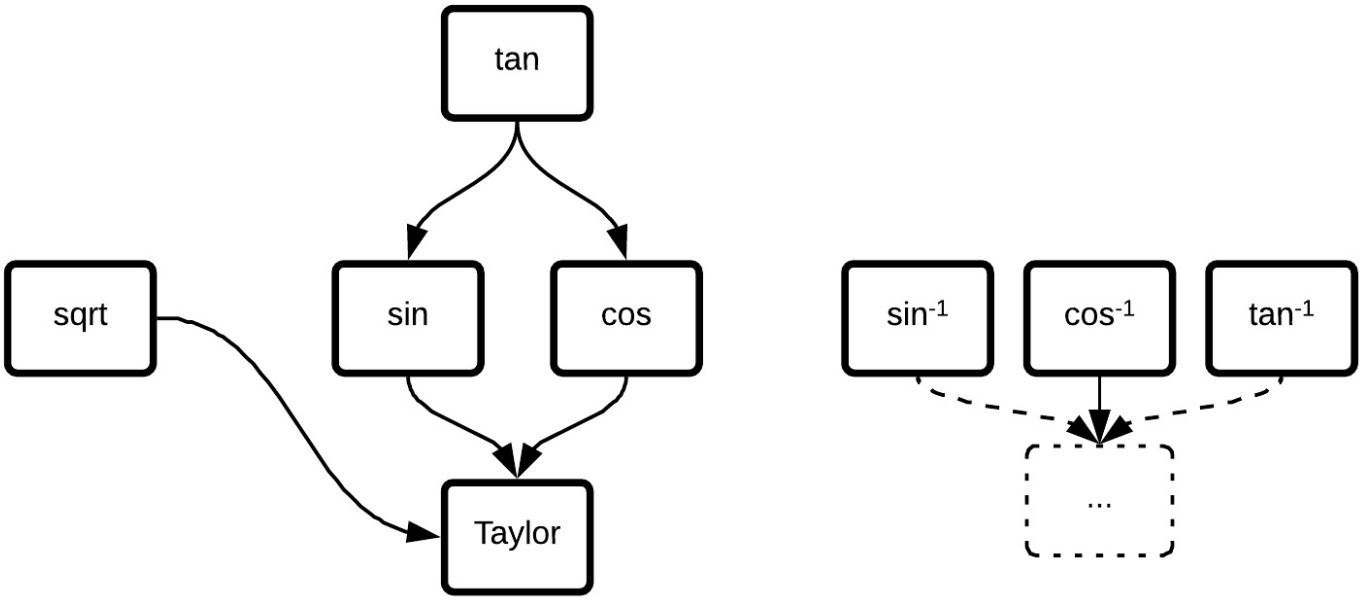
$$\sin(x) = \sum_{n=0}^{\infty} \frac{(-1)^n}{(2n+1)!} x^{2n+1}$$

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

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

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

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



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

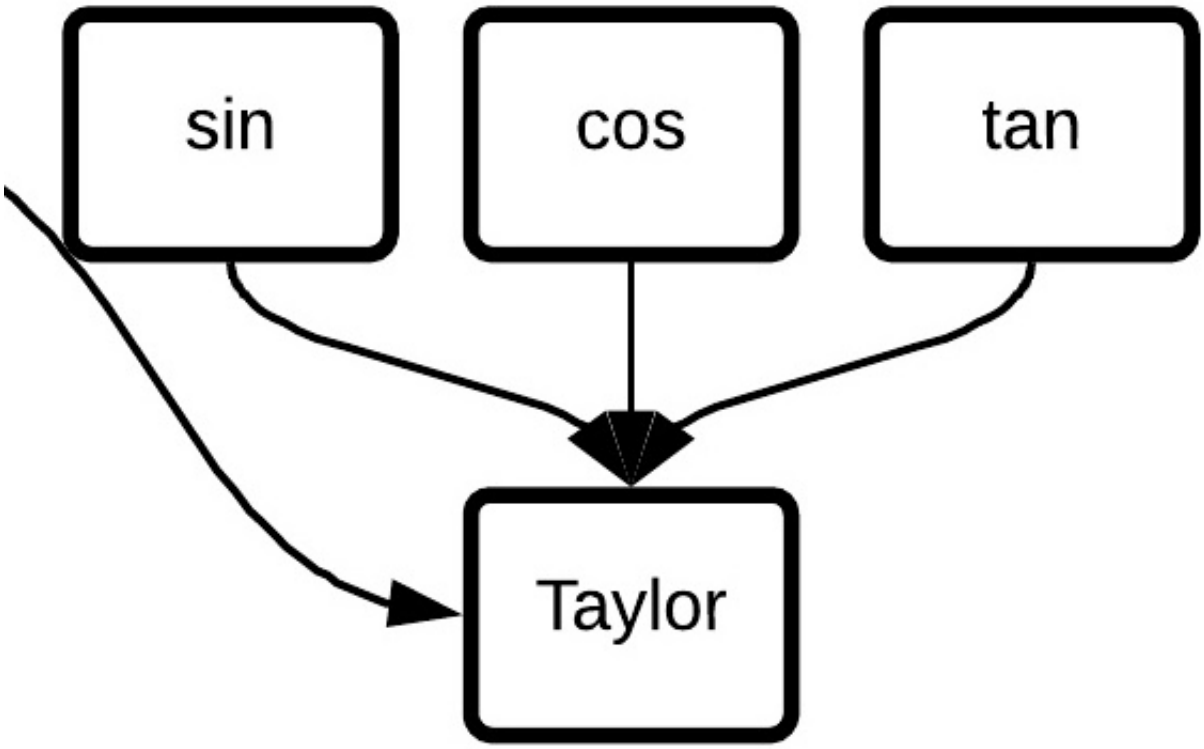
The three solutions outlined above involve three different implementations.

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)

- 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.

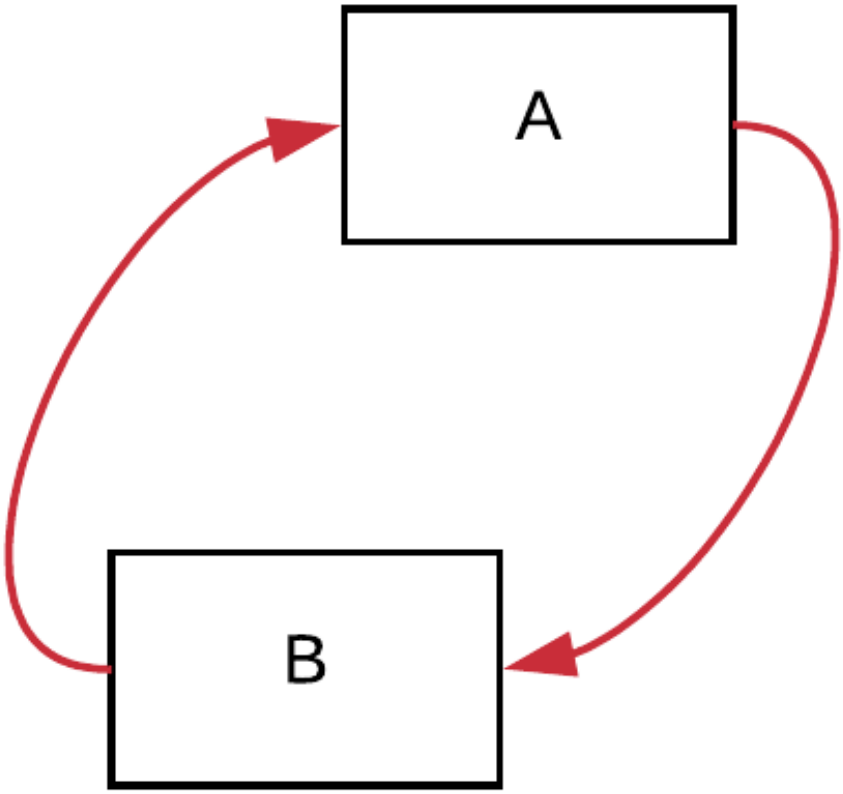


Hierarchy

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

(a generic USE relationship is not 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** (or subset of modules) **in the loop can be used or tested in isolation**.

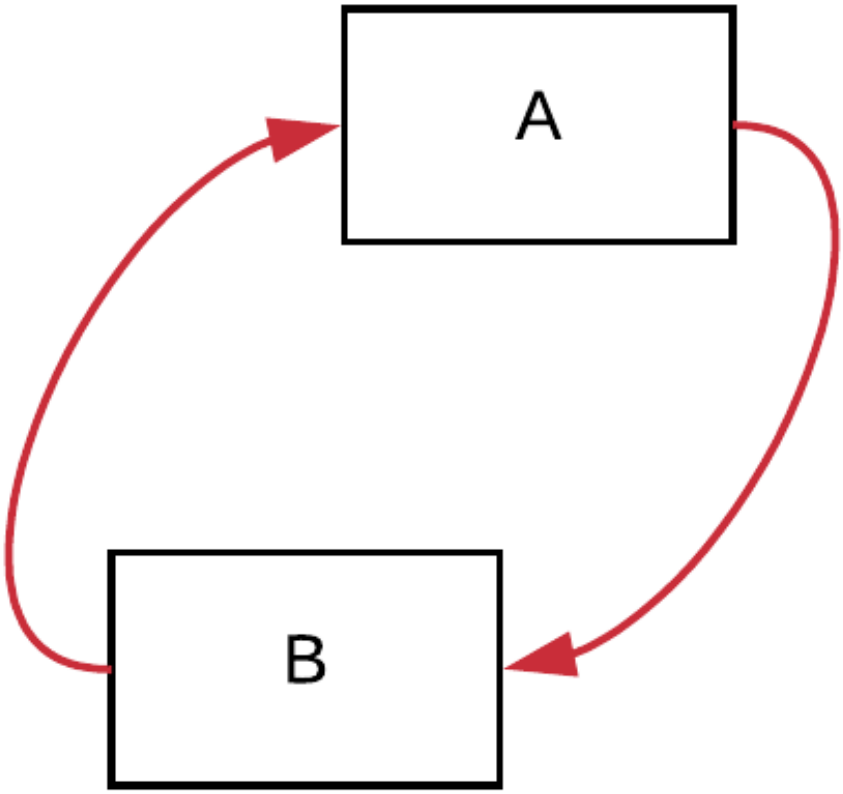
For example, if

```
`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. **Try to sketch an example of this configuration**.




```
// IT DOESN'T WORK!

typedef struct {
   int numberOfSeats;
   TypeEngine *engine;
} TypeCar;

typedef struct {
   int engineDisplacement;
   TypeCar *car;
} TypeEngine;
```

(not-so-good) SOLUTION:

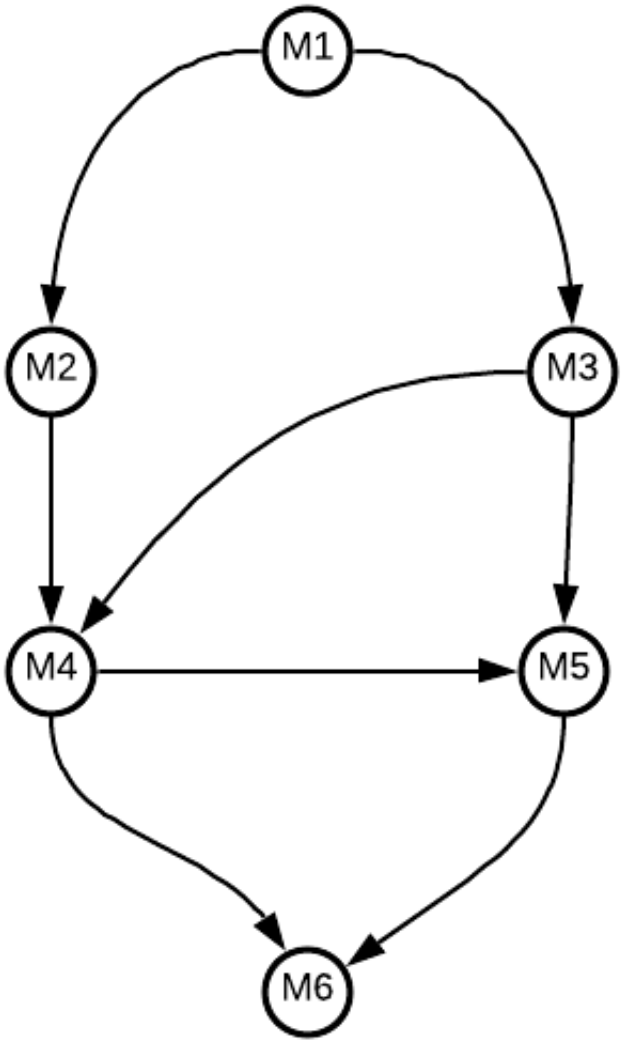
```
typedef struct structCar TypeCar;
typedef struct structEng TypeEngine;

struct structCar {
   int numberOfSeats;
   TypeEngine *engine;
};
struct structEng {
   int engineDisplacement;
   TypeCar *car;
};
```

(not-so-good) SOLUTION:

```
int main(void) {
   TypeEngine eng;
   TypeCar car;
   eng.engineDisplacement=100;
   eng.car=&car;
}
```

Another methodological advantage in the use of a hierarchy is that **the model defines the system through abstraction levels**



The system can be seen as a series of services defined and provided by the module M_1.

To implement such services, the module M_1 **USES** M_2 and M_3 (and so on).

The abstract services provided by M_1 are implemented using the lower level modules M_2 and M_3.

The **USES** relation is defined in a '**static**' way.

Consider a M module that can call the procedures $proc1 \in M_1$ and $proc2 \in M_2$

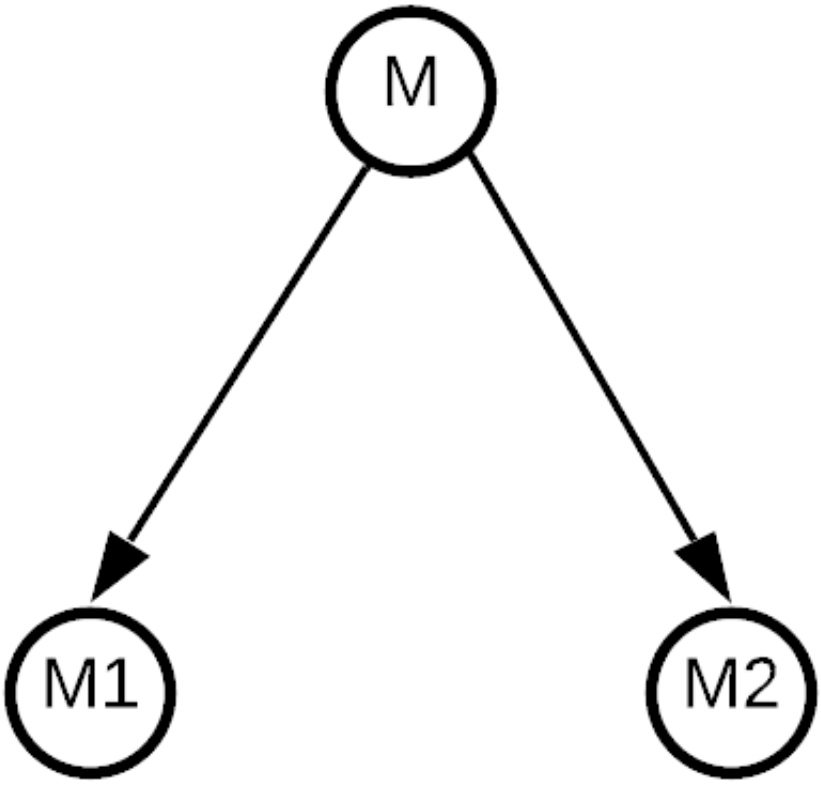
```
M:
...
if cond0 then
  if cond1 proc1
    else proc2
```

Relation: M USES M_1 and M USES M_2, even if in a specific run only M_1, or only M_2, or neither is

invoked.

The **USES** relation is defined in a '**static**' way.

```
M:
...
if cond0 then
  if cond1 proc1
    else proc2
```



...even if in a specific run only M_1, or only M_2, or neither is invoked.

(why?)

The IS_COMPONENT_OF relation

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

B **IS_COMPONENT_OF**

$$A$$

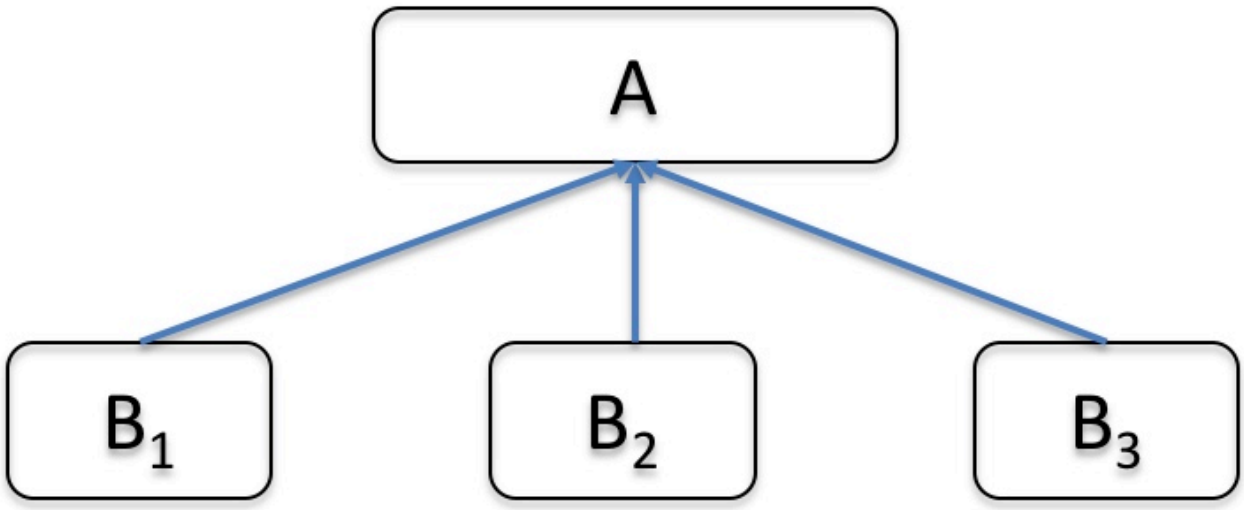
A is formed by aggregating several modules, one of which is B

B_1, B_2, B_3 modules **implement**

$$A$$

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

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

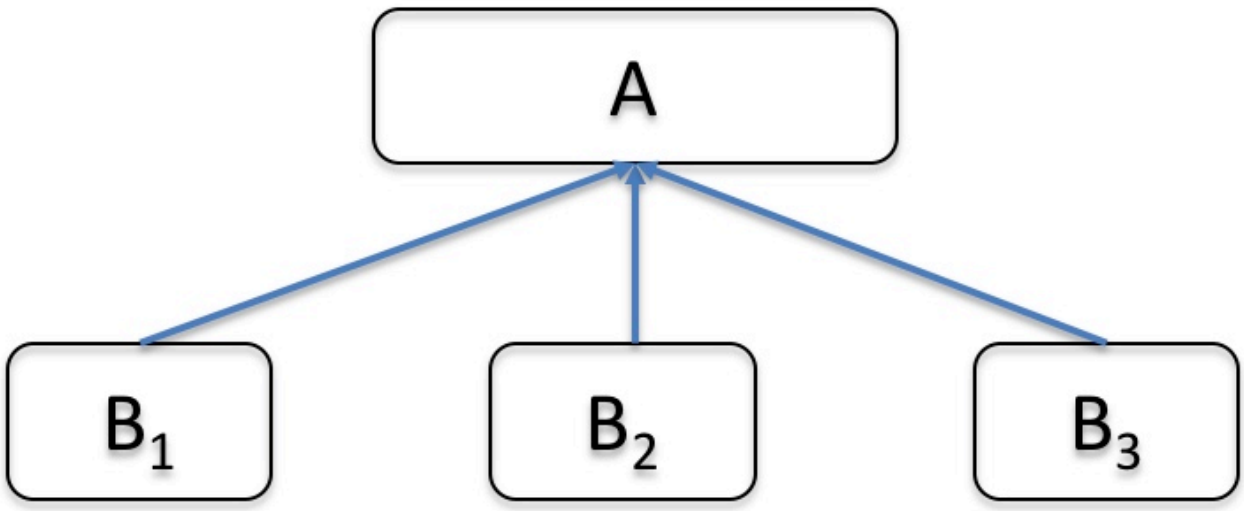


Once that A is decomposed into the set of B_1, B_2, B_3, we can replace A.

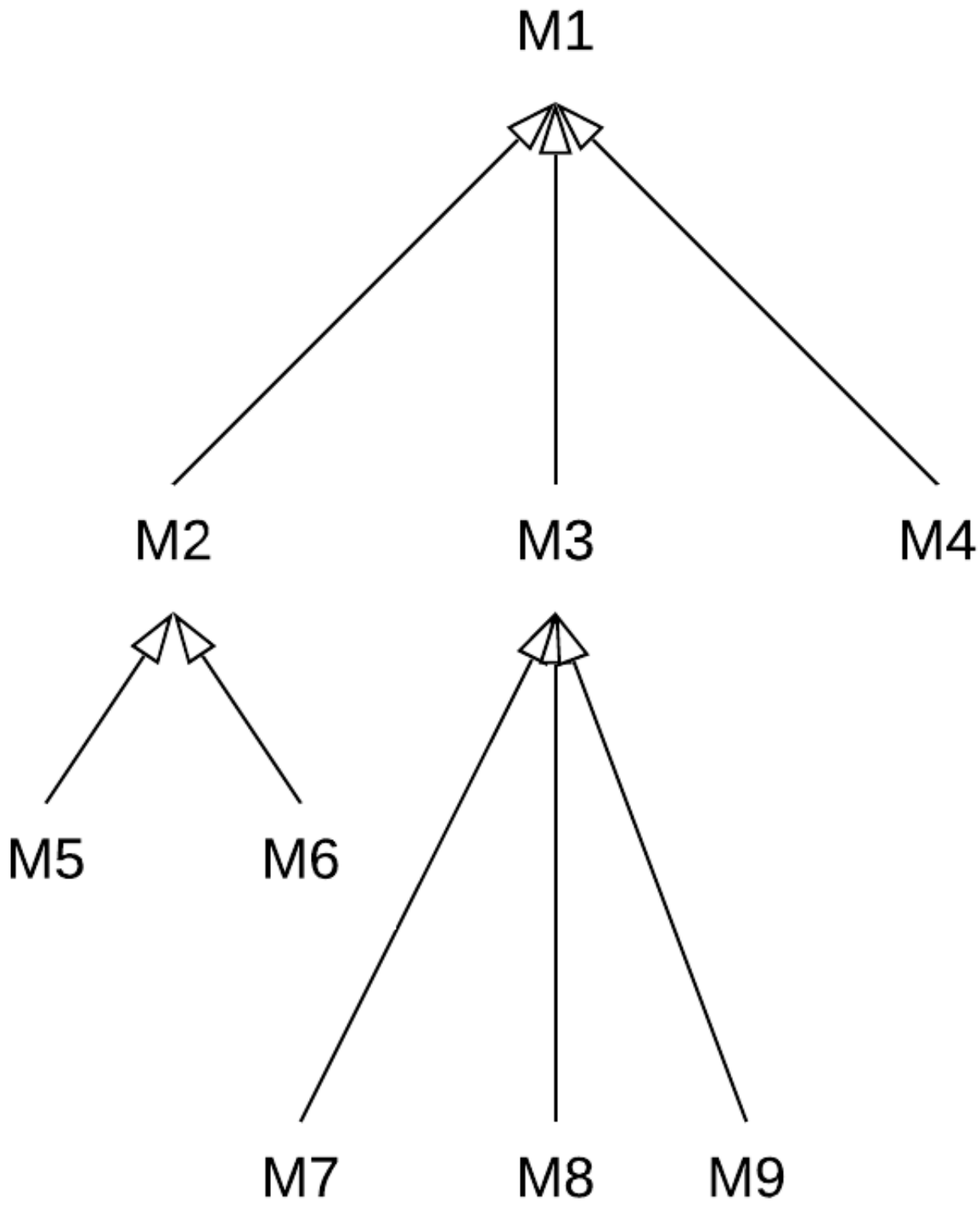
The module A is an *abstraction* implemented in terms of simpler abstractions.

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

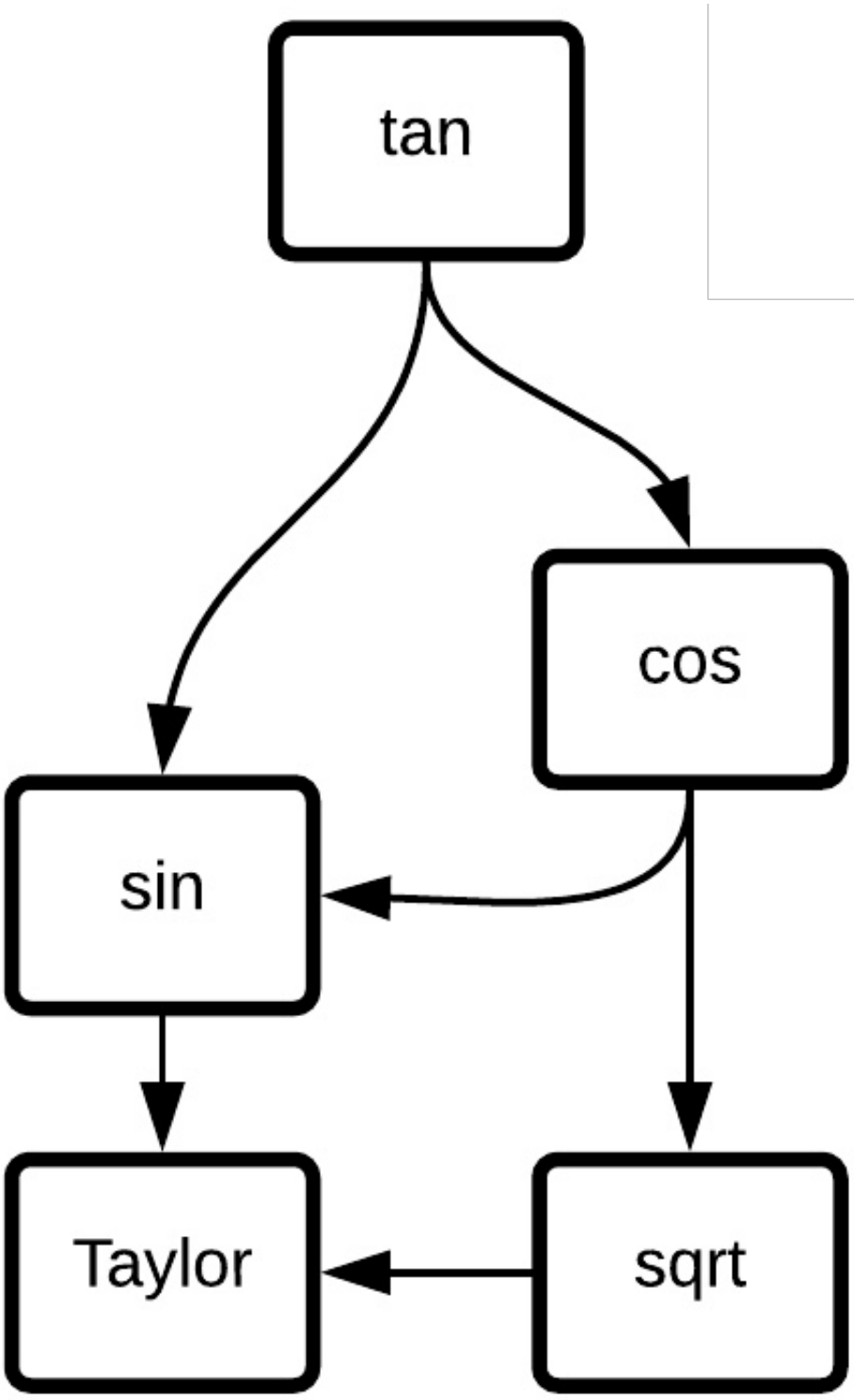
$$M_4, M_5, M_6, M_7, M_8, M_9$$

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.

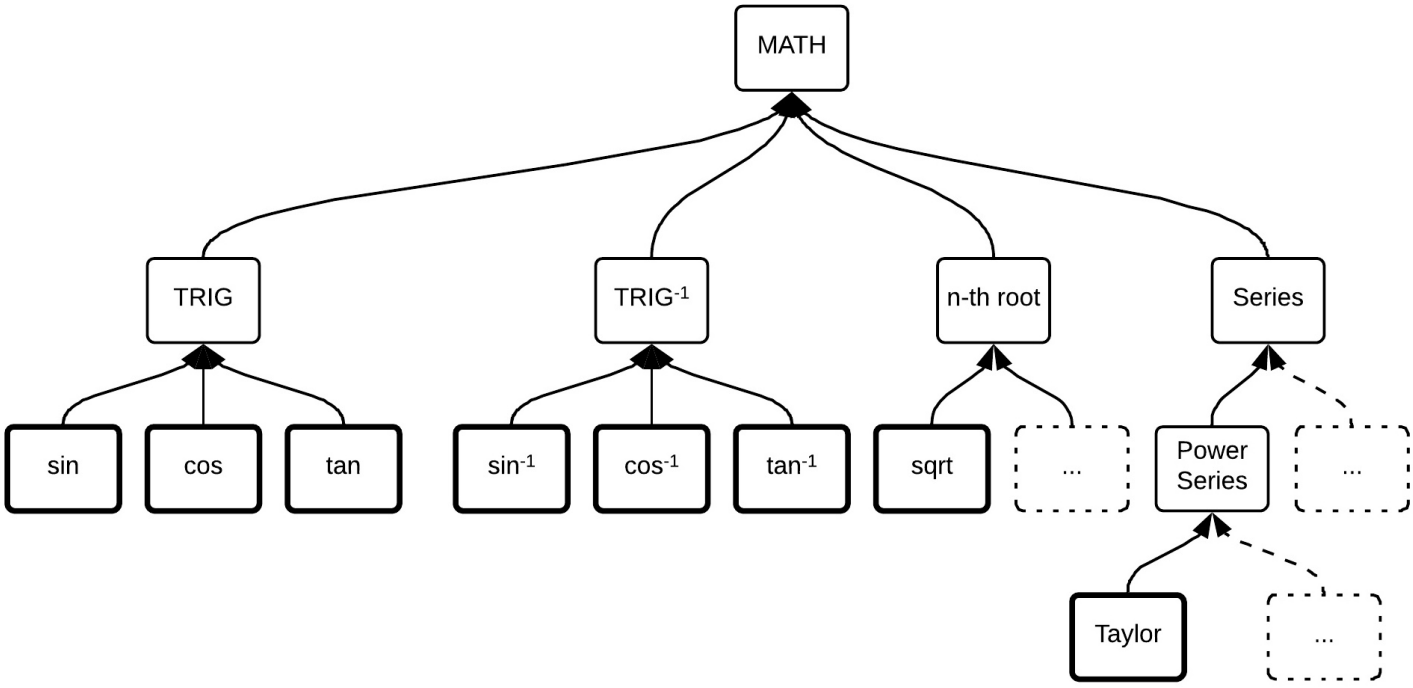
We can describe our math library using both the relations.

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.



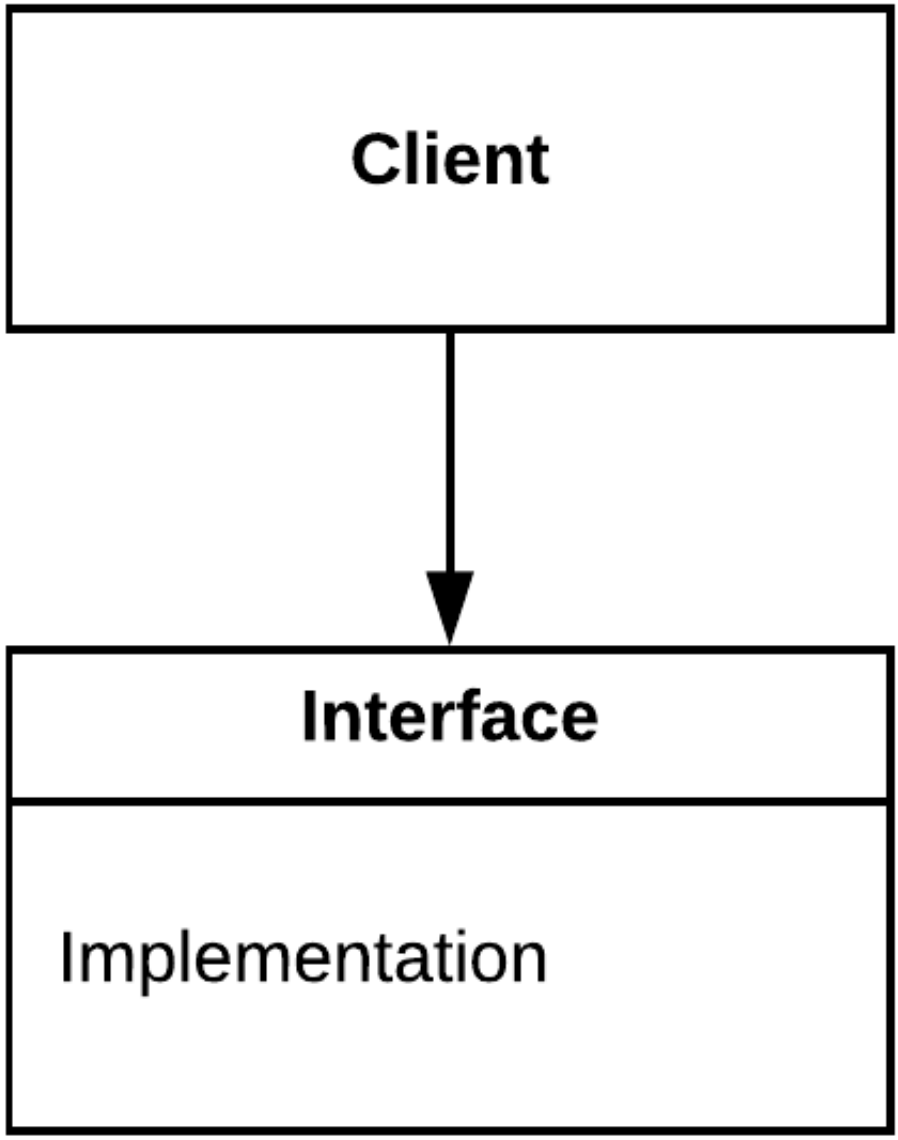
INTERFACE

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.

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.

INTERFACE

What should be **shown** by the interface and what should remain **hidden** within the implementation?

The interface should reveal as *little information as possible*, but sufficient for the other modules to use the services provided.

Revealing unnecessary information: - makes the interface unnecessarily complex - reduces the comprehensibility of the system - a change in implementation is more likely to be reflected on its interface (and in the 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 - Encapsulation is used to **hide** the details, implementation, state of a module (**the secret of a module**). The client can not know anything besides the interface. Encapsulation **prevents access** to details that are not useful.

- 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 [2]

Support to the modularity

A programming language supports *modularity* and *encapsulation* to the extent that - it automatically checks the consistency between the module and its interface - client modules call server services based only on what is specified by the interface.

Not all programming languages fully supports modularity.

1. https://en.wikipedia.org/wiki/Transitive_closure ↩

2. Stevens, Perdita, and Rob J. Pooley. Using UML: software engineering with objects and components. Pearson Education, 2006. ↩