Modularity

The **IS_COMPONENT_OF** relation

The **IS_COMPONENT_OF** relation

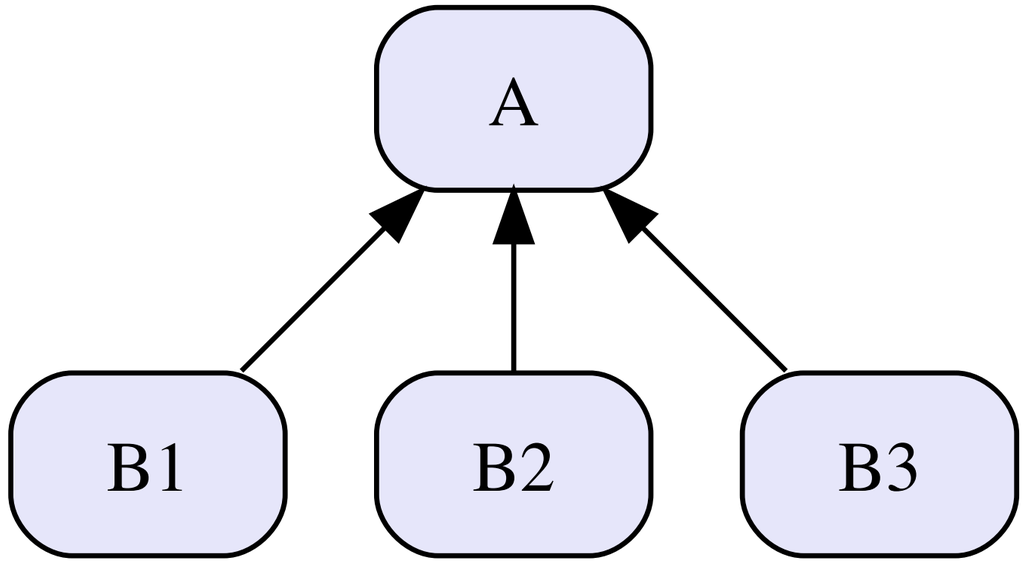
It describes an architecture in terms of a **module that is composed of other modules**

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



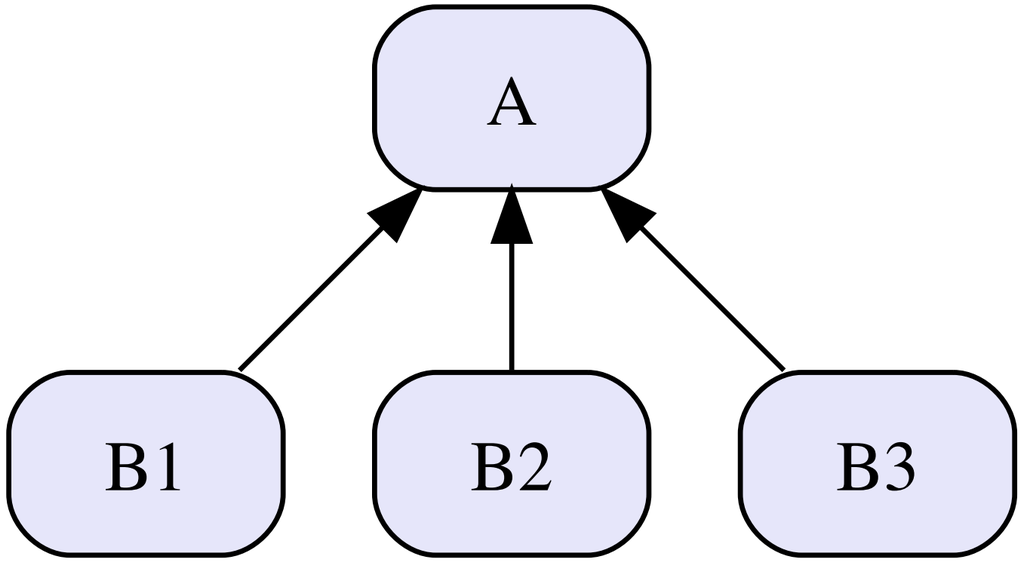
The **IS_COMPONENT_OF** relation

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.

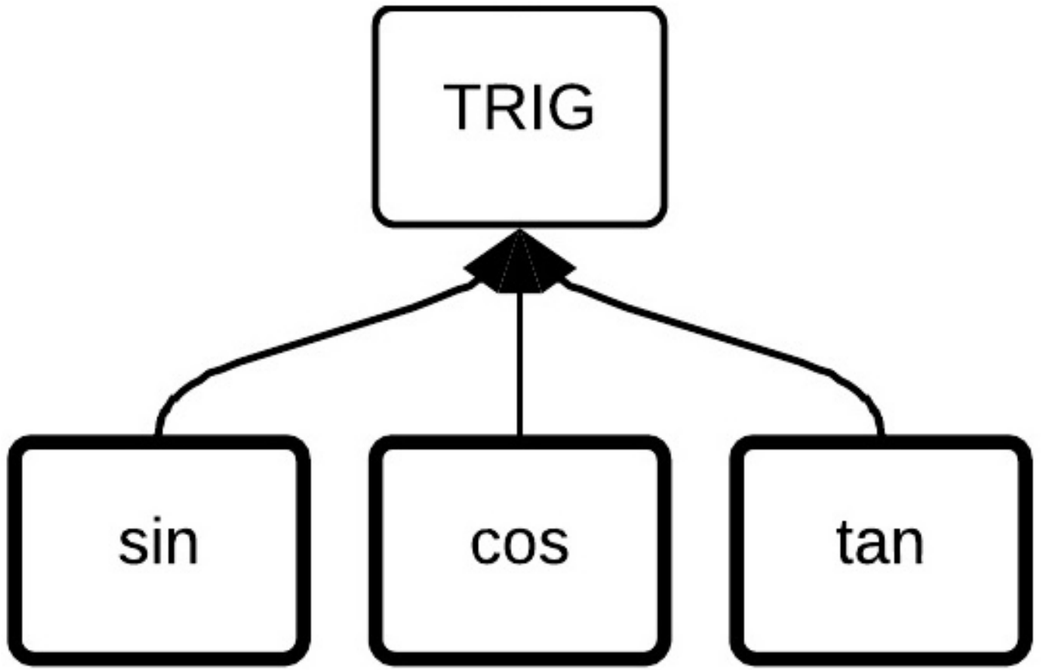
At the end of the decomposition process **only modules not made up of other modules are 'real components' of the system**. The others modules are kept only for descriptive reasons.



Example - IS_COMPONENT_OF

The entire software system is ultimately composed of modules

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

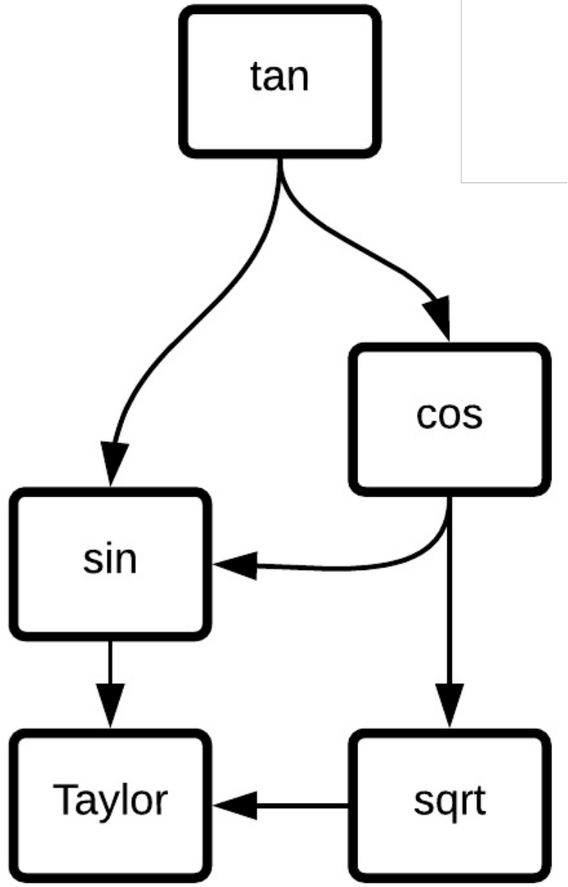


USE and IS_COMPONENT_OF

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.

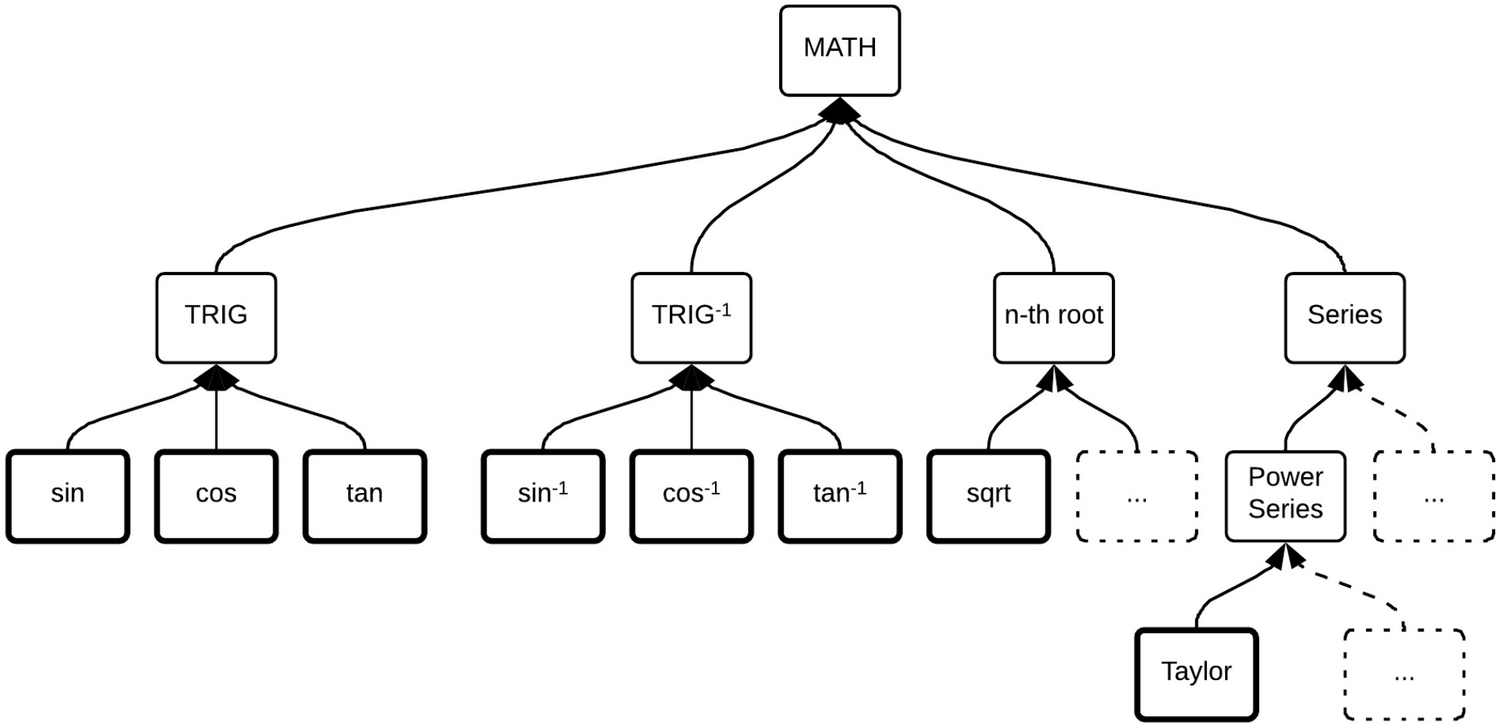
We can describe our math library using both the relations.

USE



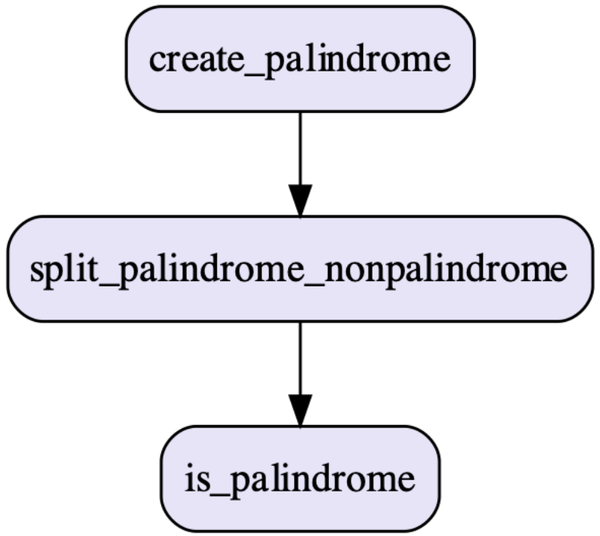
IS_COMPONENT_OF

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.



We can describe our **String** library using both the relations **USES** and **IS_COMPONENT_OF**.
In the **IS_COMPONENT_OF** relation, the modules with **bold border** are the only ones to be really implemented. The other modules represent an abstraction.

USE



IS_COMPONENT_OF