# *C++ Software Engineering*

***for engineers of other disciplines***

Module 8

*"Software Engineering"*

*4th Lecture: Pattern*

ALTEN

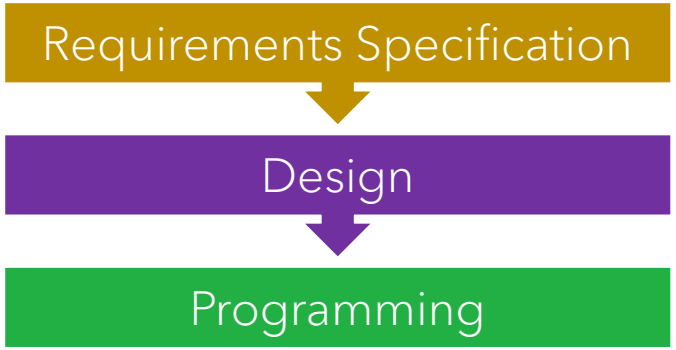
***Spring 2022***

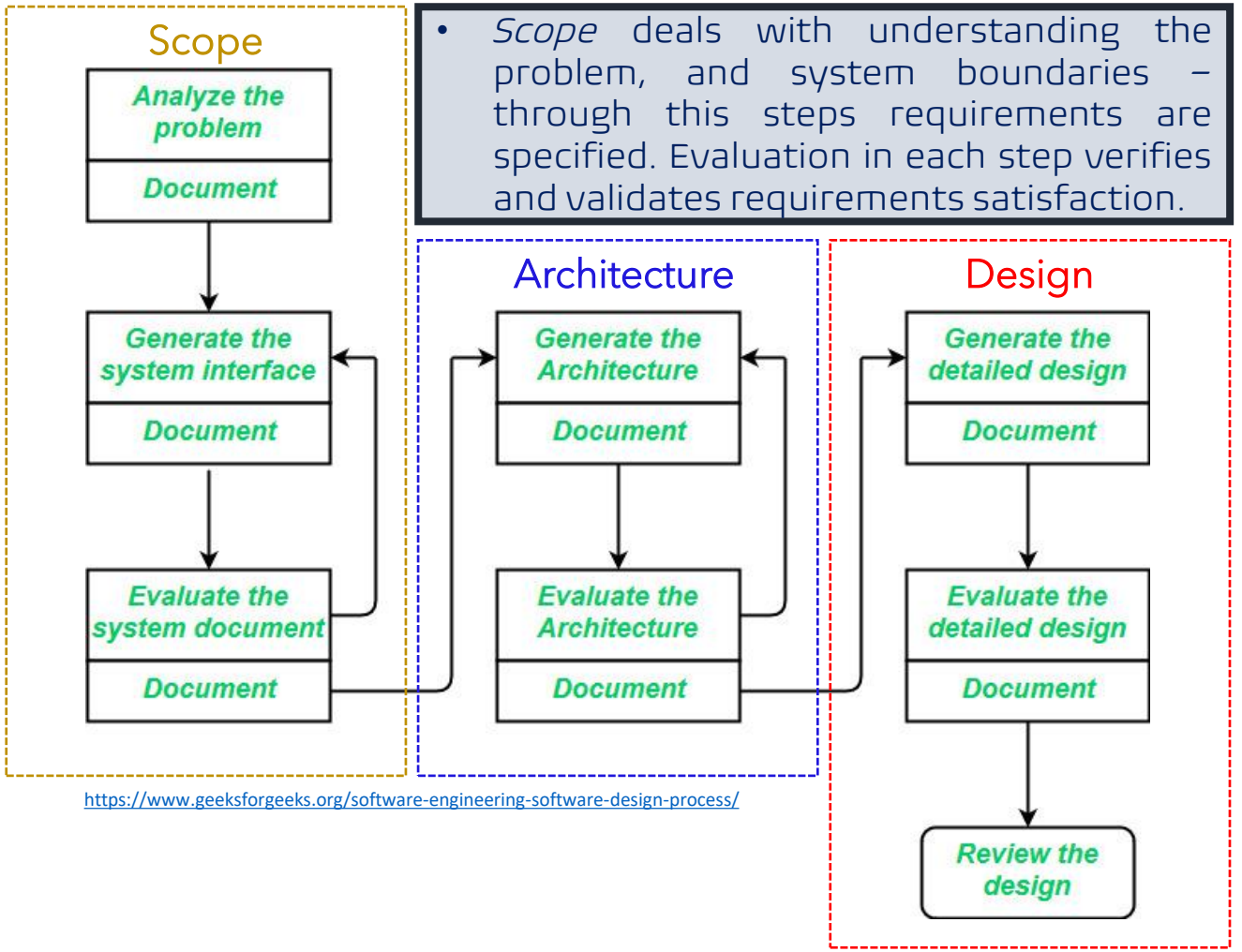
***Gothenburg, Sweden***

# Software Design



- Software design is the process of *creating software specification* which satisfies certain set of requirements.



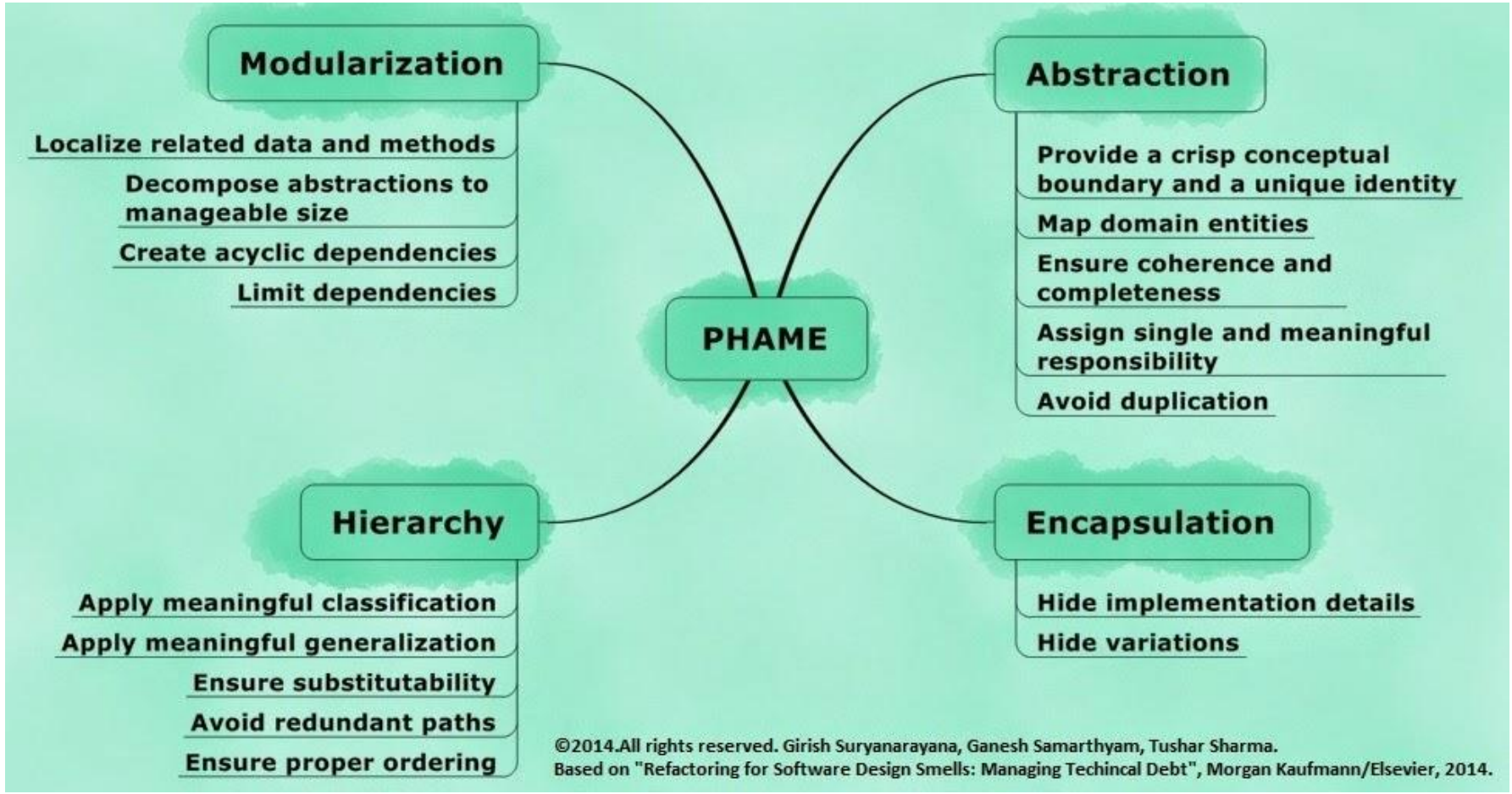




https://www.geeksforgeeks.org/software-engineering-software-design-process/

# PHAME Design Concepts



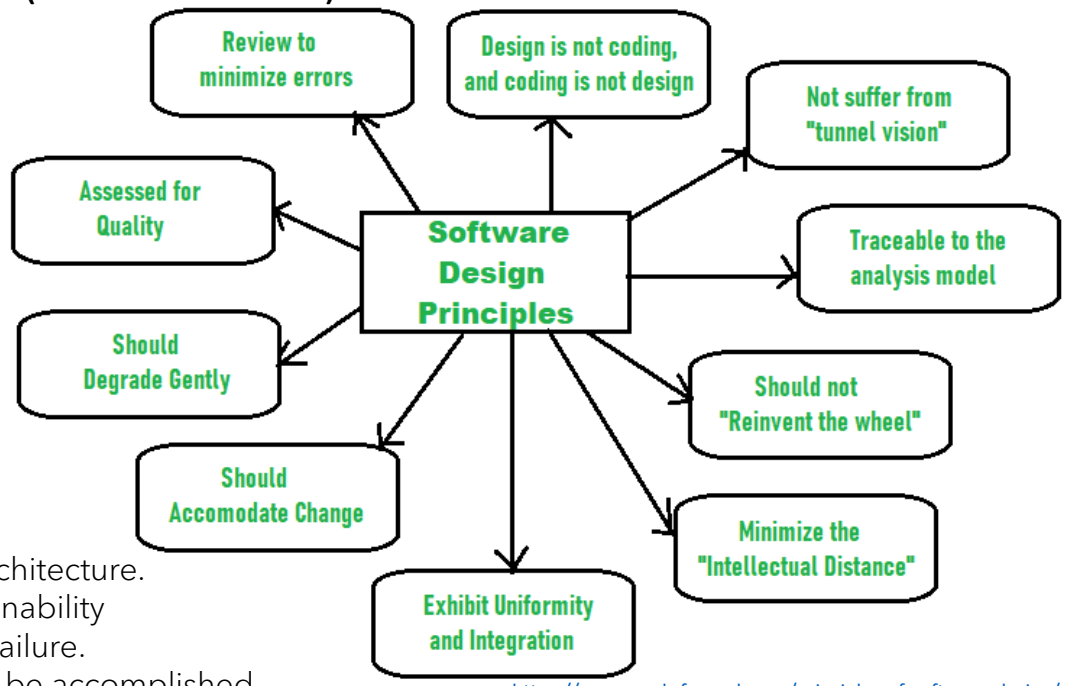
- Providing a *foundation* from which more sophisticated methods could be applied.



# Software Design General Principles



- A good design has considered alternatives and is not biased ("*tunnel vision*").
- *Early debugging is the cheapest!*
- *Assessment* shall happen at creation and development.




https://www.geeksforgeeks.org/principles-of-software-design/

Design considerations:

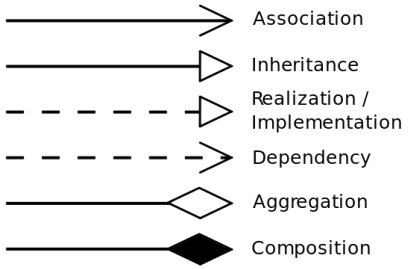
Compatibility – ability to operate with other products or an older version of itself.
Extensibility – adding new capabilities without major changes to the underlying architecture.
Modularity - well defined, independent components which leads to better maintainability
Fault-tolerance - The software is resistant to and able to recover from component failure.
Maintainability - A measure of how easily bug fixes or functional modifications can be accomplished.
Reusability - ability to use some or all the aspects of the preexisting software in other projects with little to no modification.
Reliability (Software durability) – ability to perform a required function under stated conditions for a specified period.
Robustness - ability to operate under stress or tolerate unpredictable or invalid input.
Security – ability to withstand and resist hostile acts and influences.
Usability - must be usable for its target user/audience.
Performance – ability to perform its tasks within a time-frame and memory usage that is *reasonable*.
Portability - being usable across several different conditions and environments.
Scalability – ability to adapt to increasing data or added features or number of users.

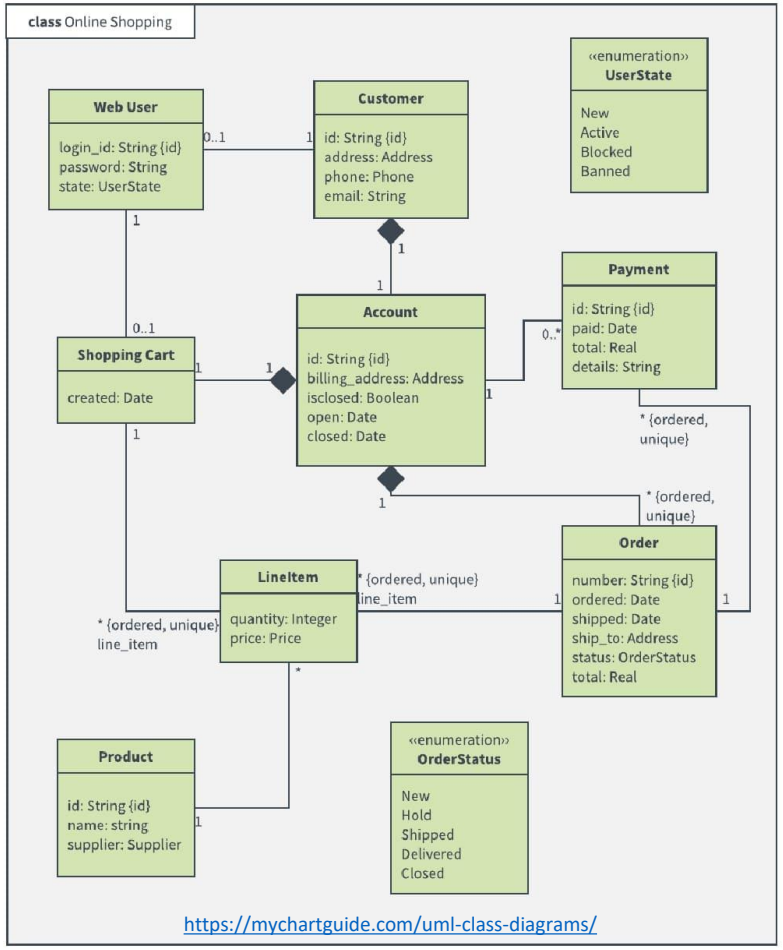
https://en.wikipedia.org/wiki/Software_design#Design_Concepts

# OO Design Inputs



- ***Conceptual Model***: Results of *object-oriented* analysis.
- ***Use Case***: *Scenarios* for system interactions.
- ***System Sequence Diagram(SSD)***: Illustrates external actors, internal order, and messages passed for each *use case.*
- ***User Interface***
- ***Relational Data Model***




- *Conceptual Model* is often represented by *Domain Model*, that incorporates both data and behavior. UML's class diagram is used to draw *Domain Models.*
- Both SSD & Domain Model are abstract and would be later detailed for *needed* use cases or classes.

| | |
|---|---|
| 0 | No instances (rare) |
| 0..1 | No instances, or one instance |
| 1 | Exactly one instance |
| 1..1 | Exactly one instance |
| 0..* | Zero or more instances |
| * | Zero or more instances |
| 1..* | One or more instances |

# Design Patterns



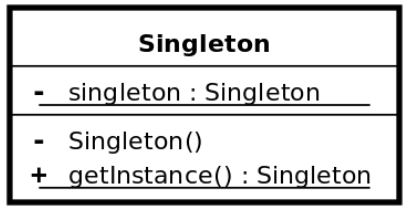
- Software Design Patterns are *general*, *reusable* solutions to a commonly occurring problem, and can speed up software development procedure.
- It is not a refined finished design ready to be translated to code, rather is a template for how to tackle the same issue which happens often.
- Design Pattern includes a ***Design Motif***; the *micro-architecture* that should be *adapted* to solve the problem, and are classified in:
  - *Creational patterns*
  - *Structural patterns*
  - *Behavioral Patterns*
  - *Concurrency patterns*

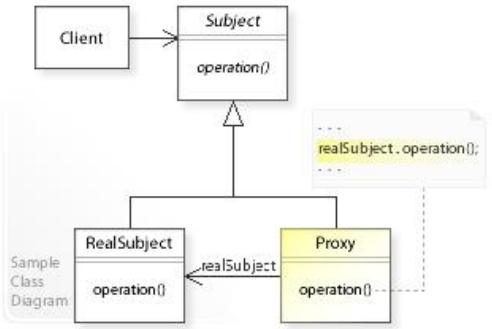
- There has been some criticisms on the design patterns elaborating on the fact that many of the patterns are *signs of missing features* in programming languages like C++ and Java – many of the patterns could be simplified or eliminated should another programming language be used.
- Inappropriate use of patterns would unnecessarily increases the complexity of the project and is a bad sign.

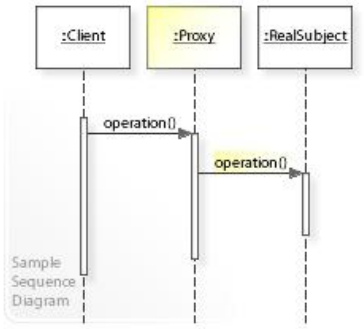
# Design Patterns



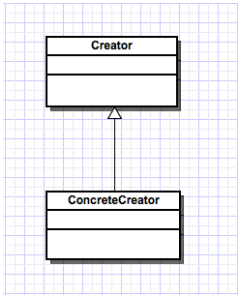
- *Behavioral patterns* deal with the communication among objects.
- *Structural patterns* deal with the relation between entities.
- *Creational patterns* deal with how the objects are created.




Pattern's *Micro-Architecture Examples*




| Creational |
|---|
| Abstract factory |
| Builder |
| Dependency Injection |
| Factory method |
| Lazy initialization |
| Multiton |
| Object pool |
| Prototype |
| Resource acquisition is initialization (RAII) |
| Singleton |

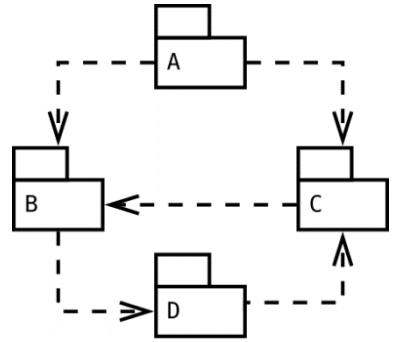
| Structural |
|---|
| Adapter, Wrapper, or Translator |
| Bridge |
| Composite |
| Decorator |
| Extension object |
| Facade |
| Flyweight |
| Front controller |
| Marker |
| Module |
| Proxy |
| Twin [20] |

| Behavioral |
|---|
| Blackboard |
| Chain of responsibility |
| Command |
| Interpreter |
| Iterator |
| Mediator |
| Memento |
| Null object |
| Observer or Publish/subscribe |
| Servant |
| Specification |
| State |
| Strategy |
| Template method |
| Visitor |

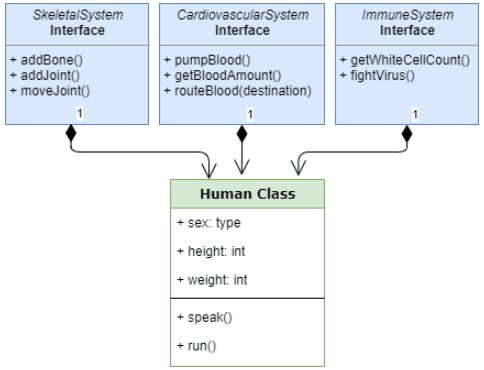
# Design principles -- OO



- *Common* object-oriented design strategies and principles are:
  - ***Composition over inheritance***
  - ***Dependency injection***
  - ***Acyclic dependencies***




https://en.wikipedia.org/wiki/Acyclic_dependencies_principle



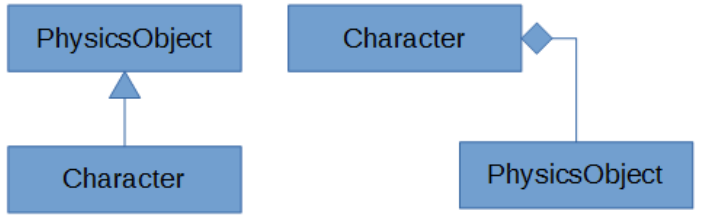

https://dev.to/romansery/why-you-should-favor-composition-over-inheritance-57m6

```
// Constructor
Client(Service service) {
    // Save the reference to the passed-in service inside this client
    this.service = service;
}
// Service setter interface.
public interface ServiceSetter {
    public void setService(Service service);
}

// Client class
public class Client implements ServiceSetter {
    // Internal reference to the service used by this client.
    private Service service;

    // Set the service that this client is to use.
    @Override
    public void setService(Service service) {
        this.service = service;
    }
}
```

4.2 Types of dependency injection

- 4.2.1 Other types
- 4.2.2 Constructor injection
- 4.2.3 Setter injection
- 4.2.4 Interface injection
- 4.2.5 Constructor injection comparison
- 4.2.6 Setter injection comparison
- 4.2.7 Interface injection comparison

https://en.wikipedia.org/wiki/Dependency_injection




**To the left an inheritance relationship, to the right composition.**

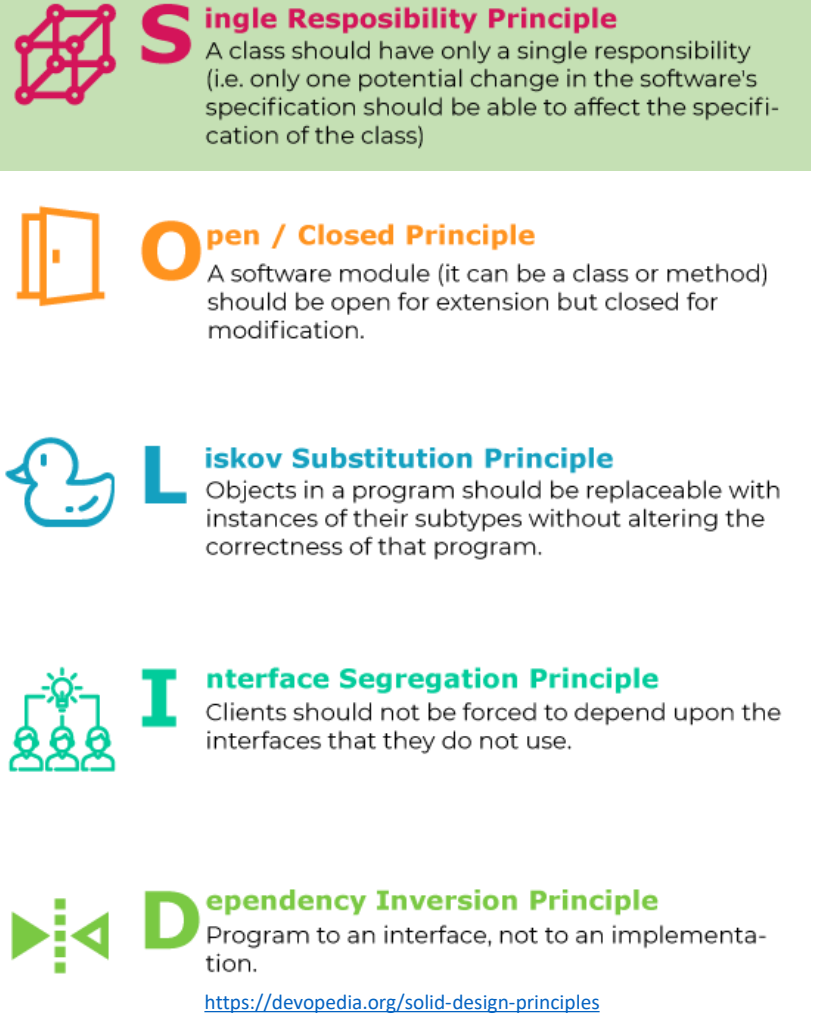
http://joostdevblog.blogspot.com/2014/07/why-composition-is-often-better-than.html

# Design principles – SOLID



- *SOLID* is the acronym for five design principles trying to make software more *understandable*, *flexible*, and *maintainable.*

- Open-Close principle suggests an entity should be able to extend its functionality without any modification on the existing code.
- LSP is also called *strong behavioral subtyping.*
- ISP is about splitting interfaces with smaller ones which packs related functionalities to together.
- Dependency Inversion Principles recommends that no *high-level* module (which might have virtual function or is expected to be used as base class) should depend on *low-level* ( i.e. concrete objects) modules. Dependencies should be toward interfaces. And abstraction should not depend on detail, it should be the other way!



# Design principles -- Abstraction



ALTEN

- *Principle of Abstraction* recommends reduction of duplication wherever practical.
- Too much abstraction could increase complexity: ***KISS!***
- Abstraction which is only used once is not good: ***YAGNI!***
- ***Rule of 3*** is abstraction principle in code.
- ***DRY*** is a generalization of abstraction principle.

*" The origins of the (Abstraction) principle are uncertain; it has been reinvented a number of times, sometimes under a different name, with slight variations[…]*

- *(1983) […] "Avoid requiring something to be stated more than once; factor out the recurring pattern".*

*[…] The definition of this principle was rather succinct in its first appearance: "no duplicate code". It has later been elaborated as applicable to other issues in software development: "Automate every process that's worth automating. If you find yourself performing a task many times, script it.""* https://en.wikipedia.org/wiki/Abstraction_principle_(computer_programming)

- You Aint Gonna Need It (YAGNI) is an XP principle advocating implementation of only necessary.
- Rule of 3 suggest to refactor a code which is used (*copied)* more than twice into a function.
- Don't Repeat Yourself (DRY) states "Every piece of knowledge must have a single, unambiguous, authoritative representation within a system".

# Design principles -- more



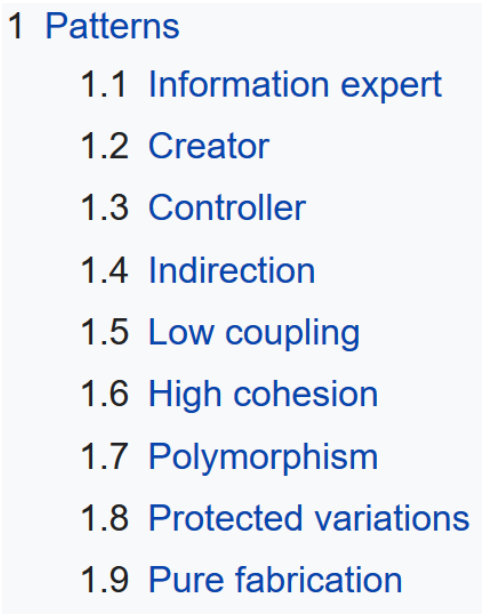
ALTEN

- ***Robustness principle****: Be conservative in what you do, be liberal in what you accept from others.*
- ***Release early, release often***: development philosophy emphasizing on frequent releases.
- ***Zero One Infinity:*** argues arbitrary restriction on number of instances of an object shall not be allowed from within the software.

*"Allow none of foo, one of foo, or any number of foo.*
*The only reasonable numbers are zero, one and infinity."*

– Bruce J. MacLennan

- ***G****eneral* ***R****esponsibility* ***A****ssignment* ***S****oftware* ***P****atterns* ***(GRASP):***
  - Guidelines for assigning responsibilities to classes and objects
  - Mental toolset, a learning aid for OO design




https://en.wikipedia.org/wiki/GRASP_(object-oriented_design)