SOLID Principles

CENG522

General Design Principles

- Encapsulation (Accessors & Mutators)
- Cohesion
- Decoupling
- Separation of concerns

General design principle: Encapsulation

- Aka Information Hiding
- While encapsulation is a fundamental attribute of the Object-Oriented paradigm, it also describes a fundamental principle of good class design; namely, hide all implementation details from the user of the class.
- The reason for doing this is so that you can change the underlying implementation without requiring user changes.
- A class that makes internal representations visible is usually poorly designed.

Accessors & Mutators (aka getters and setters)

- The usual way of accessing the properties (attributes) of a class.
  - Good encapsulation will hide the data representation.
  - The user of a class should be unaware of whether there is an actual field in the object for a property or if the property is calculated. The accessors and mutators are the interfaces to the properties.

Accessors

- Accessors retrieve the property.
  - An accessor should not have any side effects. This means that an accessor should not change the state of the property's object.
  - Further, it is not good practice to return a property as a value that, if you change it, will be reflected in the original object.
    - For example, assume object A has a Vector, v, that it uses to store some set of items and provides an accessor method, getV(). Now, if getV() returns the reference to the actual vector v, the caller to getV() can modify the contents of the vector.
  - Unless there is a critical need to allow such modifications, you should return a clone of the vector.

Mutators

- Mutators (or setters) are methods that allow (controlled) modification of properties. In effect, the mutators change the state of the object.
- Mutators should also be very specific in their effect. They should only modify the property specified and cause no other side effects to the state of the object.

Discussion

- Most editors let you automatically generate accessor and mutator methods (getters and setters) for the fields that exist in your classes.
- Should you provide accessors and mutators for every property?
- There are several disadvantages in doing so.
  - First of all, you may not need them. Whenever you provide an accessor or mutator to a property, you are telling other programmers that they are free to use them. You have to maintain these methods from that point on.
  - Second, you may not want a property to change. If so, don't provide a mutator.

Separation of concerns

- Term probably coined by Edsger W. Dijkstra in 1974.
- One of the most fundamental principles in software development.
- Quality of the process more than quality of the product.

General design principle: Cohesion

- Cohesion examines how the activities within a module are related to one another.
- Cohesion is the measure of similarity by the set of duties, level of details, and locality.
- The cohesion of a module may determine how tightly it will be coupled to other modules.
- The objective of designers is to create highly cohesive modules where all the elements of a module are closely related.

General design principle: Decoupling

- Aka uncoupling, aka coupling
- is the measure of dependence of the part on the rest of the system.
- The elements of one module should not be closely related to the elements of another module.
- Such a relationship leads to tight coupling between modules.
- Ensuring high cohesion within modules is one way of reducing tight coupling between modules.

Benefits of the Loose Coupling and High Cohesion

- Better code **clarity**. It is much easier to understand what is going on in the program when each module has a concise and clear API with a logically scoped set of methods.
- Better code **reusability** (DRY principle). The main benefit of reusing the code is reduced maintenance costs. Whenever you need to extend the functionality or fix a bug - it's much less painful to do when you're certain it appears in one place only.
- Better **testability**. Independent modules with properly scoped functionality and isolation from the rest of the app are a breeze to test.
- Faster project **evolution**. Whether it's a new feature or an update of the existing one, isolation of the modules helps with scoping out the areas of the program that may be affected by the change, thus speeding up the development.
- It is easier to organize **simultaneous development** by multiple engineers. They just need to agree on which module they are working on to make sure they don't interfere with each other

SOLID

- The **S**ingle Responsibility Principle - **SRP**
- The **O**pen-Closed Principle - **OCP**
- The **L**iskov Substitution Principle - **LSP**
- The **I**nterface Segregation Principle - **ISP**
- The **D**ependency Inversion Principle - **DIP**

What has changed?

- In the early 2000's, Java and C++ were king.
- Since then, the changes in the software industry have been profound;
  - Dynamically-typed languages
  - Non-object-oriented paradigms
  - Open-source software
  - Microservices and software as a service

Source: https://stackoverflow.blog/2021/11/01/why-solid-principles-are-still-the-foundation-for-modern-software-architecture/

What hasn’t changed?

- But there are some things that haven’t changed and likely won’t. These include:
  - Code is written and modified by people.
  - Code is organized into modules.
  - Code can be internal or external.

SOLID

- Why?
  - "To create understandable, readable, and testable code that many developers can collaboratively work on.“

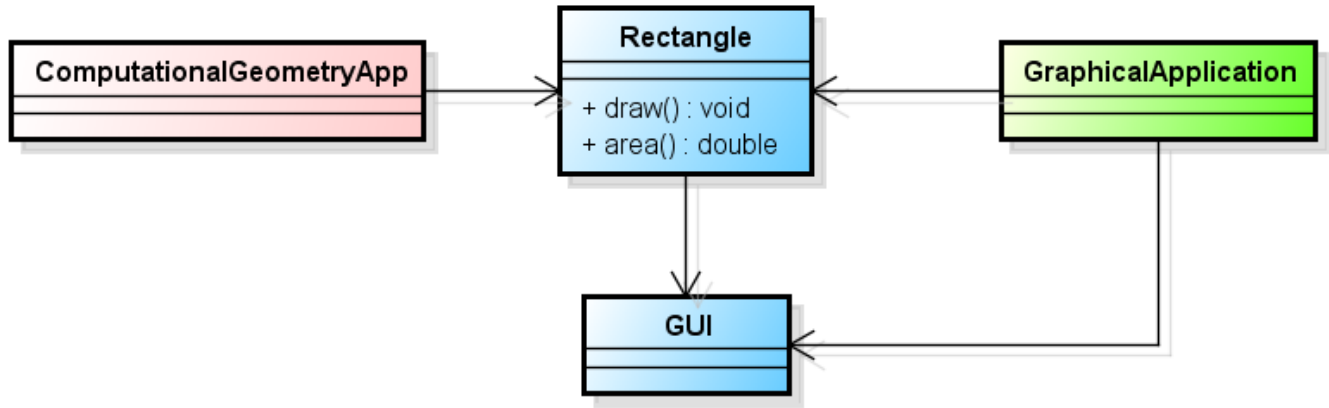
S — Single Responsibility

- *There is one and only one reason to change a class*.
- This principle aims to separate behaviours so that if bugs arise as a result of your change, it won't affect other unrelated behaviours.
- Then the code will be:
  - More readable
  - More robust
  - Better testable
  - Better maintainable and extendable

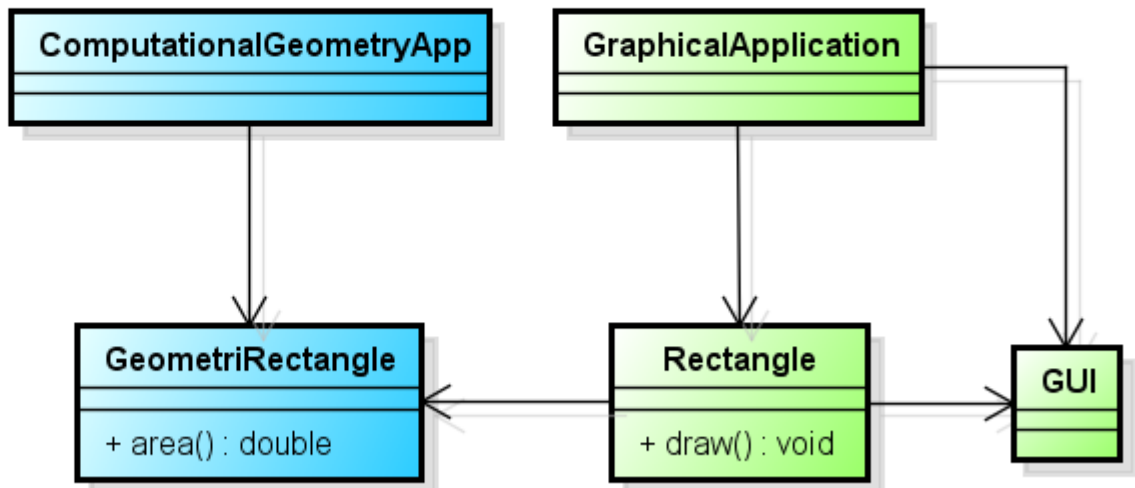
The principle of SRP can be used when:

- too much is allowed to the class object
- any change in the logic of the object's behavior leads to changes in other places of the application
- you have to test, fix errors, even if a third party is responsible for their performance
- not so simple to separate and apply a class in another area of the application, since it will pull unnecessary dependencies

Bad:


ComputationalGeometryApp
Rectangle
+ draw() : void
+ area() : double
GraphicalApplication
GUI

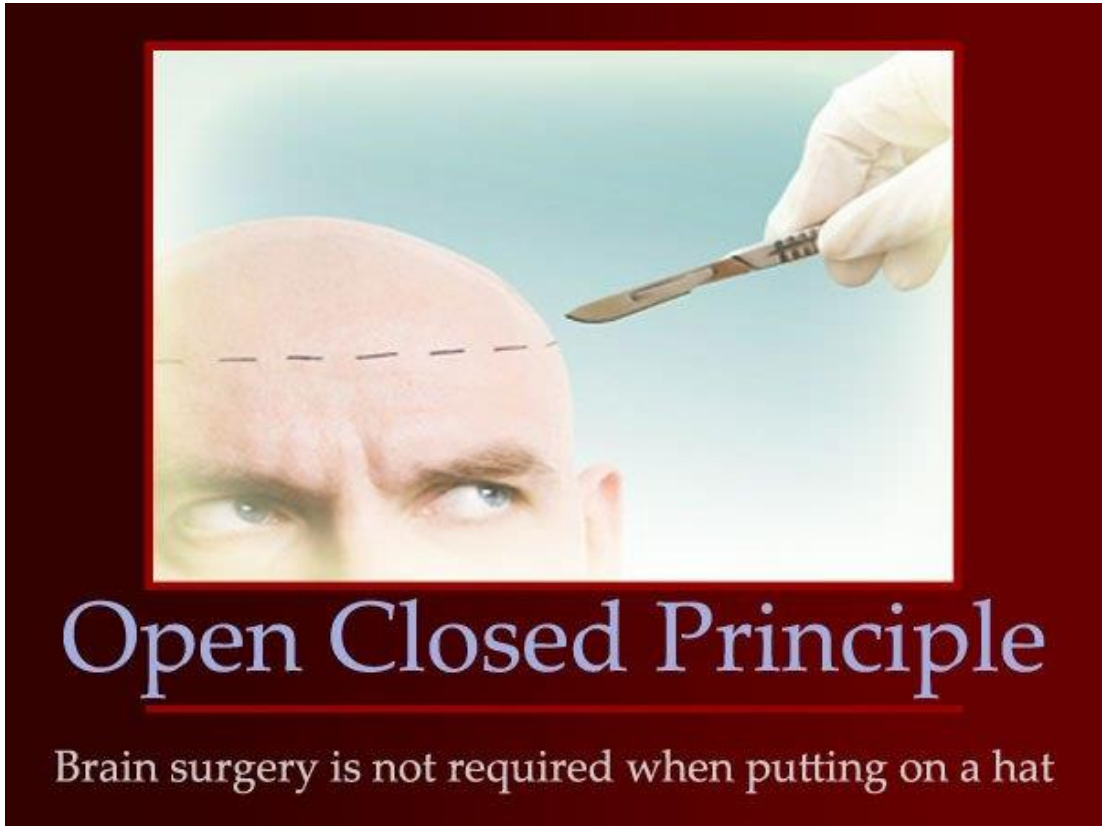
Good:


ComputationalGeometryApp
GraphicalApplication
GeometriRectangle
+ area() : double
Rectangle
+ draw() : void
GUI

SRP - DEMO

O — Open-Closed Principle

- Classes should be open for extension, but closed for modification
- This principle aims to extend a Class's behaviour without changing the existing behaviour of that Class. This is to avoid causing bugs wherever the Class is being used.



Open Closed Principle

- *Software entities should be open for extension but closed for modification*.
- There are several ways to extend a class/entity:
  - Inheritance
  - Composition
  - Proxy implementation

OCP

- The principle of OCP suggests that software entities must be:
  - should be open for extension: this means, that module can be extended. When applications requirements change, we are able to expand the module.
  - In other words, we have the ability to extend classes, making them more functional. At the same time, the behavior of the old methods does not change, and class itself is not changing to.
  - closed for modification: after the expansion of the entity behavior, no changes should be made to the code that uses these entities.
- This is especially important for code in a production environment:
  - any changes in the source code requires a revision of the entire code, where this entity / class is used.
  - revision of unit testing and other similar procedures.
- If code follows this principle, therefore does not require such effort.

OCP - DEMO

L — Liskov Substitution

- If S is a subtype of T, then objects of type T in a program may be replaced with objects of type S without altering any of the desirable properties of that program.
- This principle aims to enforce consistency so that the parent Class or its child Class can be used in the same way without any errors.

LSP - Liskov Substitution Principle

- The LSP principles are followed in the following cases:
  - No new exceptions should be thrown by methods of the subtype, except where those exceptions are themselves subtypes of exceptions thrown by the methods of the supertype.
  - does not violate the functionality
  - returns the same type
  - subclass object has a contract with a superclass
- The LSP principle is violated in the following cases:
  - the parent class in the method calls some kind of external service, and the child completely rewrites the method.

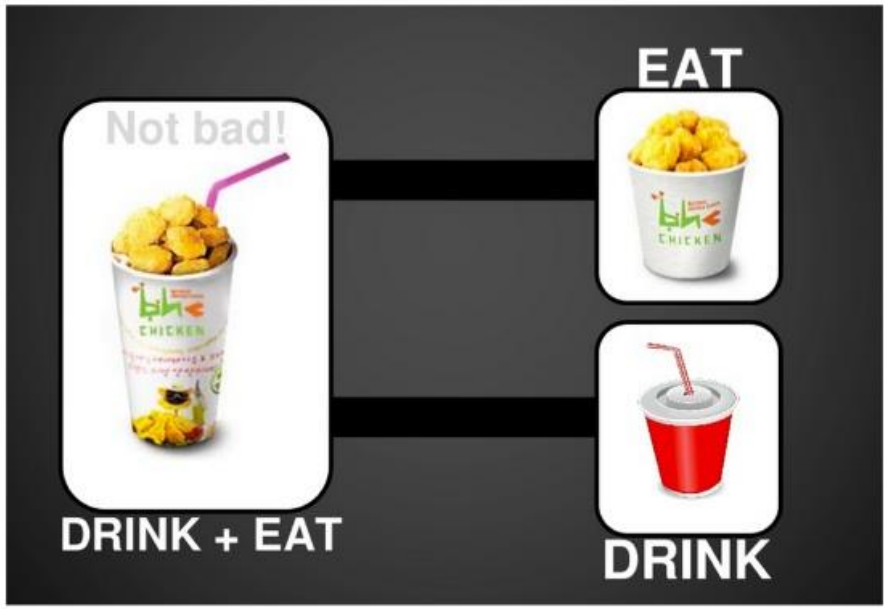
LSP - Liskov Substitution Principle

- If for each object o1 of type S there is an object o2 of type T such tha t for all programs P defined in terms of T, the behavior of P is unchan ged when o1 is substituted for o2, then S is a subtype of T.
  - ????
- Function that use pointers or references to base classes must be abl e to use objects of derived classes without knowing it.
- Given an entity with a behavior and some possible sub-entities that could implement the original behavior, the caller should not be surprised by anything if one of the sub entities are substitute d to the original entity.

LSP - DEMO

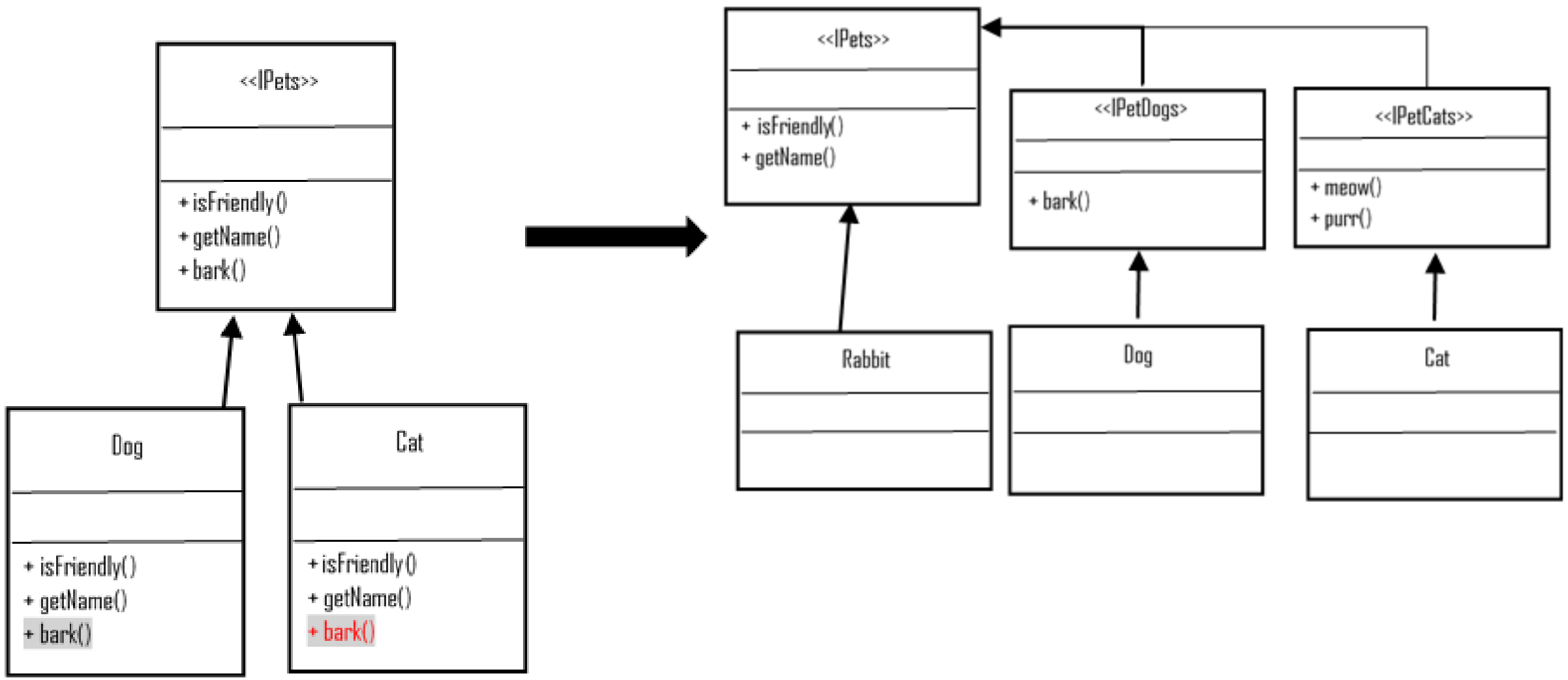
I — Interface Segregation

- Clients should not be forced to depend on methods that they do not use.
- This principle aims at splitting a set of actions into smaller sets so that a Class executes ONLY the set of actions it requires.



ISP - Interface segregation principle

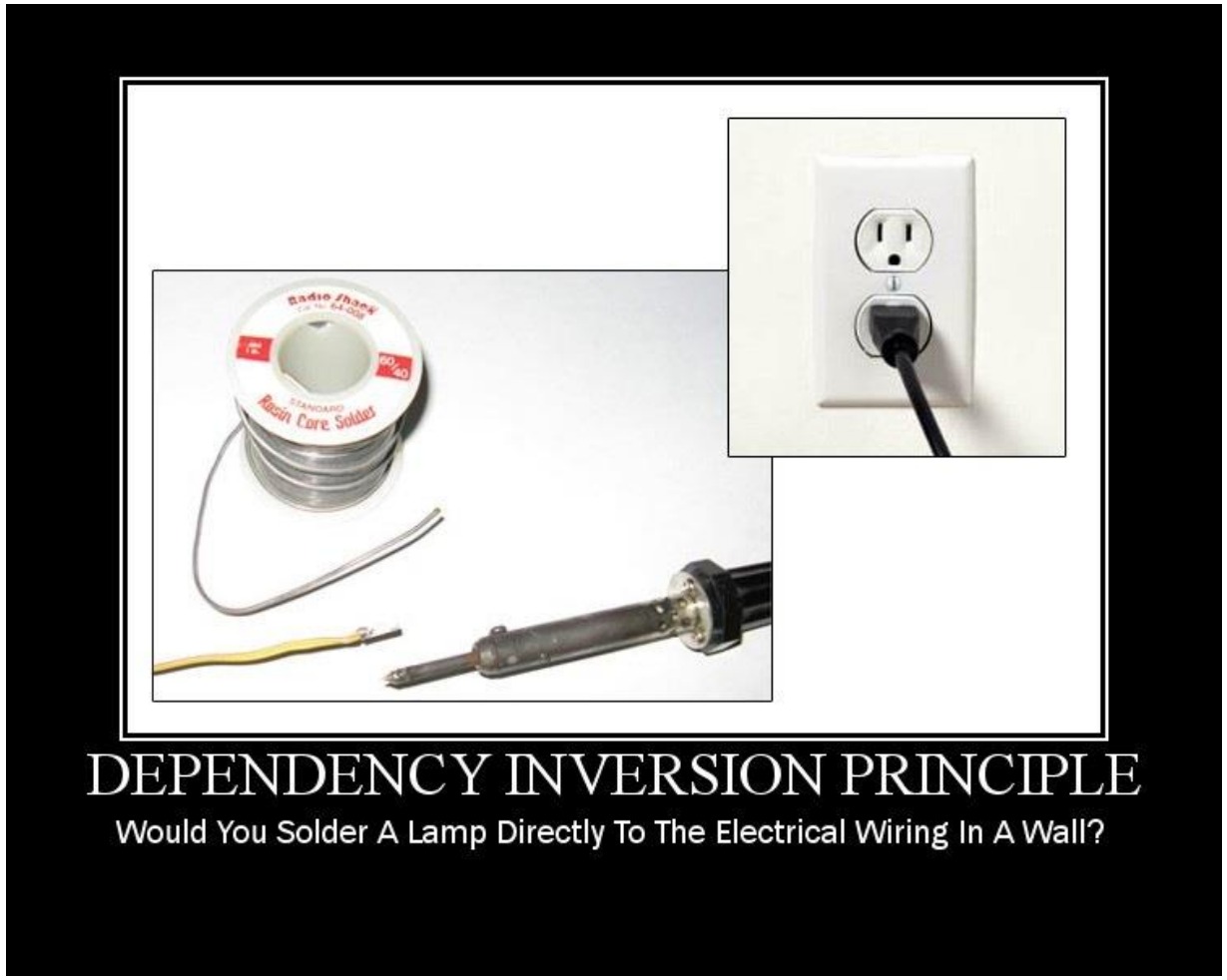
- following this principle helps the system stay flexible when making changes to the logic of work and suitable for refactoring.
- No any client should be forced to depend on methods it does not use.
- ISP splits interfaces that are very large into smaller and more specific ones
- clients will only have to know about the methods that are of interest to them
- ISP is intended to keep a system decoupled and thus easier to refactor, change, and redeploy


<<IPets>>
+ isFriendly()
+ getName()
+ bark()
Dog
+ isFriendly()
+ getName()
+ bark()
Cat
+ isFriendly()
+ getName()
+ bark()
<<IPets>>
+ isFriendly()
+ getName()
<<IPetDogs>>
+ bark()
<<IPetCats>>
+ meow()
+ purr()
Rabbit
Dog
Cat

ISP DEMO

D — Dependency Inversion

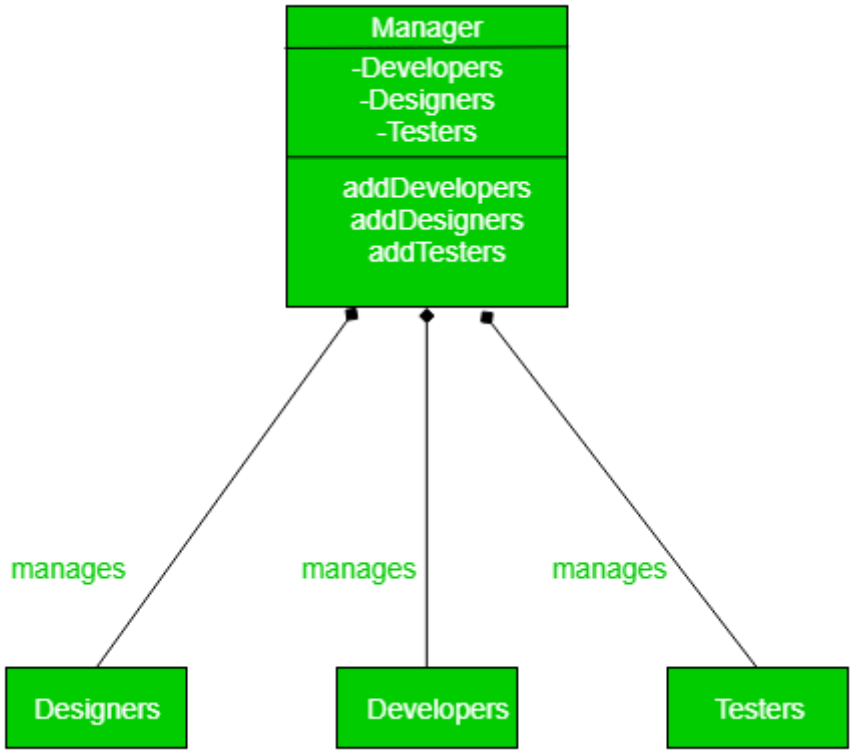
- High-level modules should not depend on low-level modules. Both should depend on the abstraction.
- Abstractions should not depend on details. Details should depend on abstractions.
- This principle aims at reducing the dependency of a high-level Class on the low-level Class by introducing an interface.

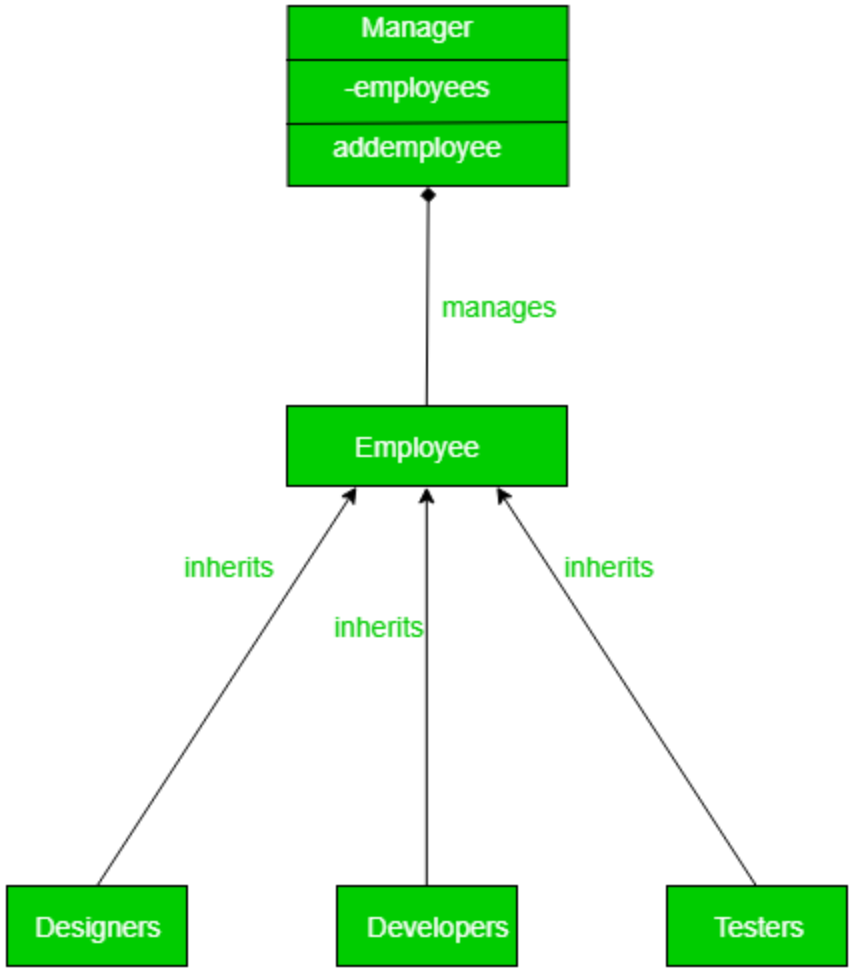


DIP - Dependency Inversion Principle

- Dependency inversion talks about the coupling between the different classes or modules.
- Any higher classes should always depend upon the abstraction of the class rather than the detail.
- This aims to reduce the coupling between the classes is achieved by introducing abstraction between the layer, thus doesn’t care about the real implementation.

DIP - Dependency Inversion Principle





DIP DEMO

Conclusion

- Good design is needed to successfully deal with change
- The main forces driving your design should be high cohesion and low coupling
- SOLID principles put you on the right path
- Warning: these principles cannot be applied blindly :)

Questions?

Thank You!