SOLID principles

References

Clean Architecture: A Craftsman's Guide to Software Structure and Design (Robert C. Martin)

Synopsis

Why do we care about design?

Software architecture is akin to real architecture. You may very well obtain a mess with well-made base materials.

SOLID principles gives sensible advices on how to lay out these software bricks.

It gives us some simple rules as to how our classes (types), methods functions and data Structure should be composed and interconnected.

Objectives of SOLID

The principles were crafted with a few goals in mind:

- Be **easy to understand** and to apply
- Have **practical** applications
- Build software that is **robust to change**
- Can be applied to various types of systems*

*The SOLID principles are not restricted to OO.

What are they?

SRP: The Single Responsibility Principle

OCP: The Open-Closed Principle

LSP: The Liskov Substitution Principle

ISP: The Interface Segregation Principle

DIP: The Dependency Inversion Principle

Single Responsibility Principle

A software entity (class, function, object, method, module, component) should have one responsibility.

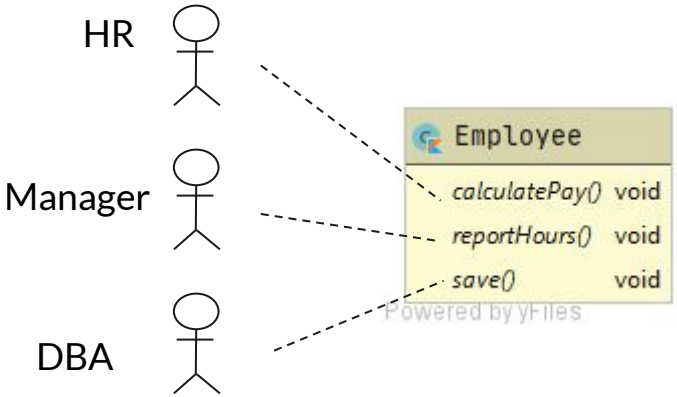
The word “responsibility” is voluntarily vague. This principle means that your entity should be responsible of one part of your Software specification.

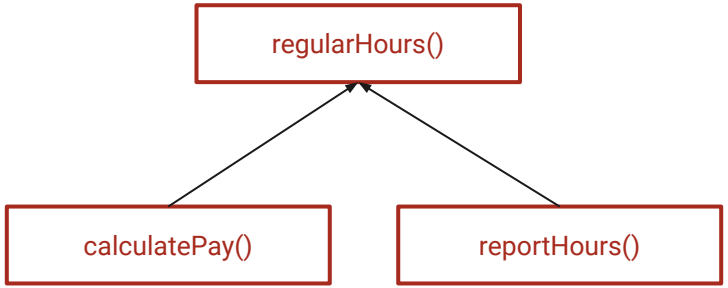
It DOESN’T mean that your entity should do “one thing”.

It however means that the entity should have “one, and only one, reason to change”.

For example, if a function both reads a file and parses XML. It has two responsibilities, i.e. one too many!

SRP


HR
Manager
DBA
Employee
calculatePay() void
reportHours() void
save() void
Powered by yFiles


regularHours()
calculatePay()
reportHours()

SRP: Symptoms

- **Growing code:** class, functions of methods with so much LOC that they become difficult to reason about.
- **God entities:** typically objects that are very pervasive to the rest of the code base. They control too much of it.
- **Methods or functions with lots of parameters**. Maybe some of these parameters need to be grouped in their own entities.
- **Names in -er:** entities coming with names like “SomethingManager” or “OtherThingController” *might* be a smell of too much responsibilities. “Managing” and “Controlling” are not sensible names for responsibilities.

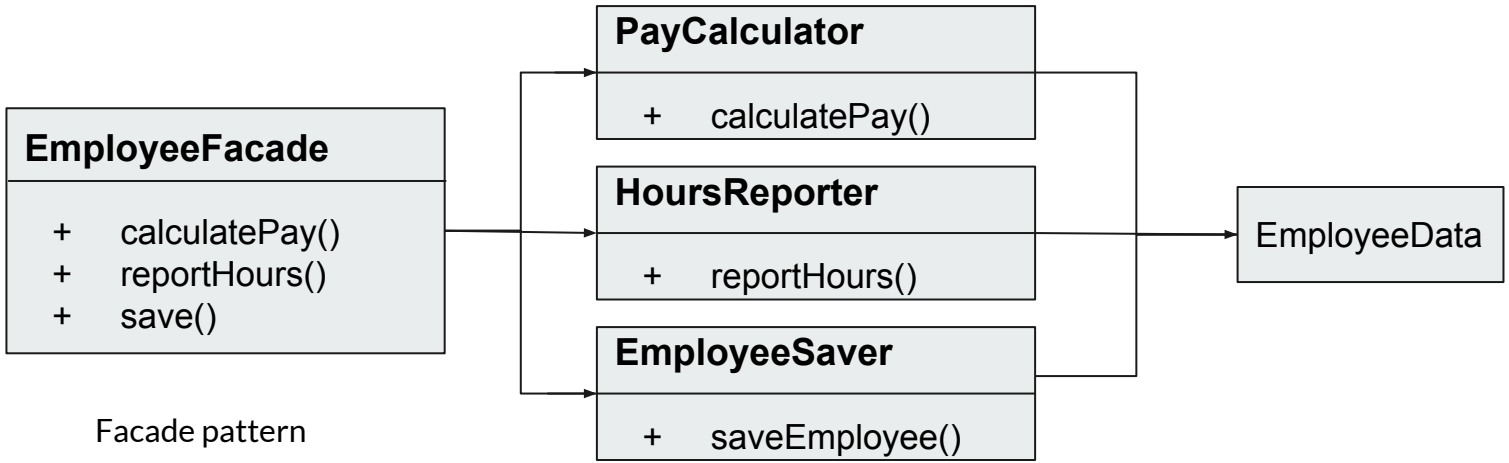
These symptoms are easy to spot: just be careful of something that is “too big” or something that is “everywhere”. Also Singletons have this tendency to contradict SRP as code grows.

SRP: Symptoms

- Accidental duplication
- Conflictual merges

SRP: Solutions

Split the entities. Divide responsibilities. Use composition.



Facade pattern

Open-Close Principle

“Software entities should be closed for modification and open for extension”.

Which means: if a specification change occurs, your system should be changed by adding new code, not changing old one.

The reason to that is simple to understand: modifying existing code introduces risks of regressions.

Adding code, on the other hand, will naturally isolate related bugs. (providing you have no side effect)

OCP: Symptoms

- **High rate local changes:** Some portion of code is changed very often and/or is very brittle. You need to rewrite whole portions of it.
- **High cyclomatic complexity**: the code has big if-else / switch portions
- **Sclerotic interface:** your interfaces (ex: Restful API) are difficult to evolve. Maybe you have expose things that should have been left as implementations details

OCP: Solutions

Use Polymorphism to reduce cyclomatic complexities.

Favor interfaces as parameters for functions calls. “Expect the least” from others.

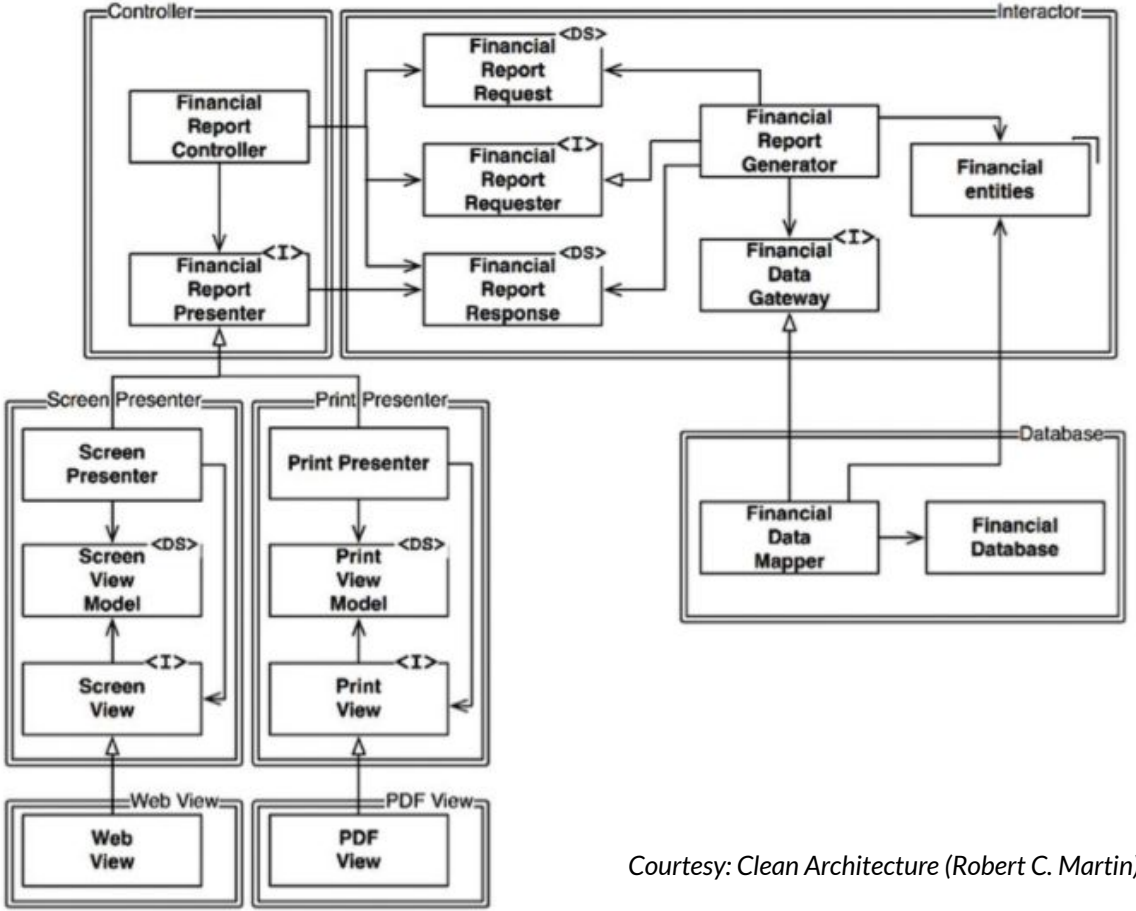
Reduce the number of arguments in your methods. Decompose them if needed.

Favor smaller entities.

OCP

Dependency flows in one direction from one component to another.

Your dependencies should point to the part of your code you want to protect from change.



Courtesy: Clean Architecture (Robert C. Martin)

Liskov Substitution Principle

"Substitutability"

In a program, instances of a type should be replaceable with instance of any of its subtype without compromising the software integrities.

Note that this is a principle that is more applicable to inheritance.

LSP: Symptoms

- **Heavy usage of reflection** (or type inspection or whatever corresponding features) in places where polymorphism was more appropriate.
- **Type switches:** a particular case of reflection.
- **Side effects** in your methods that are transient to parent. The child implementation breaks implementation of its parent and modify the behaviour of the object.

LSP: Solutions

Favor polymorphism.

Be reluctant to use reflection/type switching, unless needed.

Favor interface of inheritance.

Favor composition over inheritance.

ISP: Interface Segregation Principle

"Many client-specific interfaces are better than one general-purpose interface."

ISP: Symptoms

- **Empty implementations:** especially in inheritance based scenarios. Some child don't care or can't implement all of the methods. They would typically then throw exceptions or errors that the consumer will have to check.
- Lots of similar and **duplicated code**

ISP: Solutions

Design smaller interfaces. But more of them.

Split big interfaces by theme.

“Expect the least” from others. Use interfaces in your function calls instead of concrete implementation.

Dependency Inversion Principle

Depend on abstraction, not concrete implementation.

High level modules should not depend on low level modules. Both should depend on a common Abstraction.

Abstraction should not depend on details. Details should depends on abstraction. Hence the “Inversion”.

DIP: Symptoms

- **Code is untestable**: You cannot design unittest because of the code's dependencies.
- **Nested dependencies** everywhere
- Code is **difficult to refactor** due to strong coupling

Corollary to DIP : IOC and DI

IOC: Inversion of Control

DI: Dependency injection

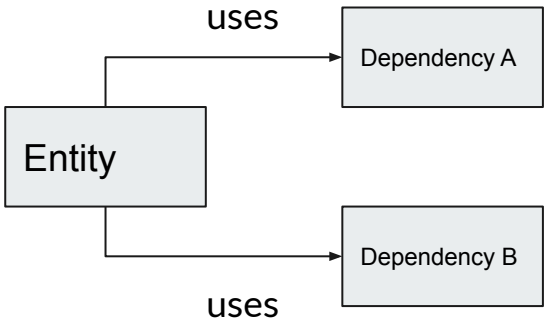
Inversion of control

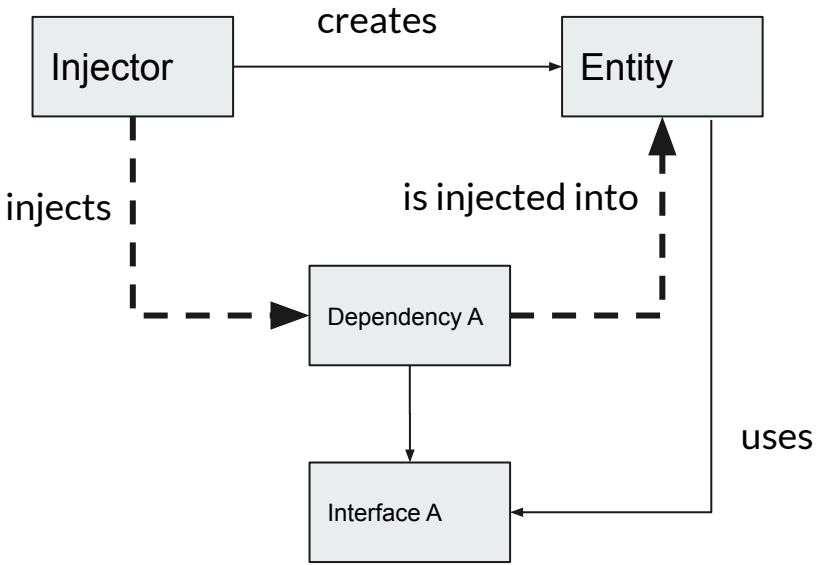
Control (usage) relationships can also be reversed.

Ex: Be wary of the “Controller” and “Manager” that controls the lifecycle of other objects.

Note: it’s not always desirable to reverse control. Sometimes straightforward master-slave relationships are just more efficient and easier to understand.

Dependency injection





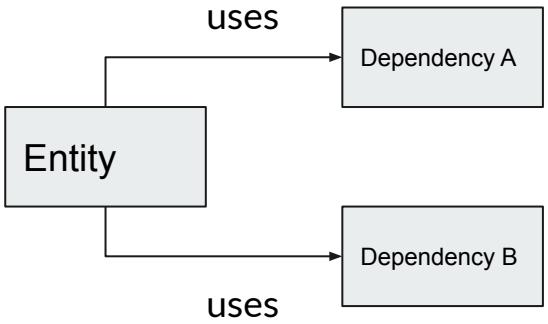
Note: in this figure, the interface is not strictly necessary

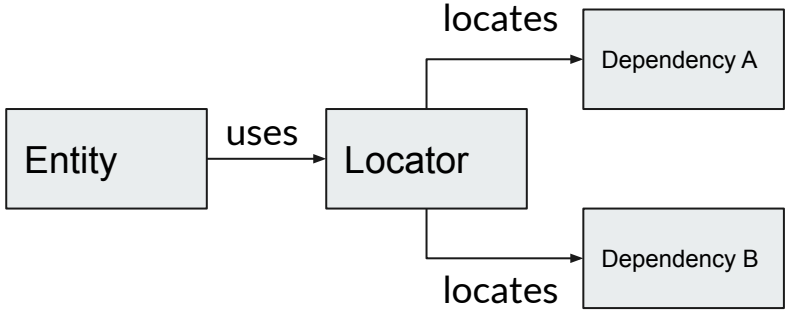
Dependency injection

Typically we distinguish three types of injection (but there are variants):

- Constructor injection
- Setter injection
- Interface injection

Service Locator





Exercise

Implement a game of chess