Alfonso De La Guardia

S.O.L.I.D Principles

Overview

When we talk about design and application development, SOLID is a word that you should know as one of the fundamentals of software architecture and development.

SOLID is the acronym coined by Michael Feathers programming on the principles of object-oriented programming that Robert C. Martin had compiled in 2000 in his document "Design Principles and Design Patterns".

Eight years later, Uncle Bob continued to compile advice and good development practices and got lost in the father of the clean code with his clean book Clean Code.

Table of Contents

Project Requirements

- You must use GIT
- Create a clear and orderly directory structure
- Both the code and the comments must be written in English
- Use the camelCase code style for defining variables and functions
- In the case of using HTML, never use online styles
- In the case of using different programming languages always define the implementation in separate terms
- Remember that it is important to divide the tasks into several sub-tasks so that in this way you can associate each particular step of the construction with a specific commitment
- You should try as much as possible that the commits and the planned tasks are the same
- Delete terms that are not used or are not necessary to evaluate the project

Risk Management Plan

Every project has risks. These risks must be taken into account to improve the workflow of the project. I've listed the risks for the project, along with the impact they might have, and the priority of them.

iD	**Risk**	**Consequence**	**Prob. (1-5)**	**imp. (1-5)**	**Pri. (1-25)**	Mitigation approach
1	Breaking my computer	Can't do anything	1	5	5	Keep my repo work to date, look for other computers
2	Getting sick	Wouldn't be as productive	2	3	6	Eat and sleep well
3	Not concentrating enough	Won't com	2	5	10	Focus on the Minimum Viable Product.
4	Unrealistic deadlines	Deadlines wouldn't be met + development shortcuts would have been	2	3	6	Be more organized with the tasks, set new deadlines.

		taken affecting the robustness of the code				
5	Being unfocused	Loss of control over the development flow of the project	4	5	20	Focus on finishing the most important tasks only.

Project Tasks

Defining this part is crucial to the development of the project. It is important to make a good analysis of the situation to organize the project in a good way:

#	Task	Priority (1-5)	Description	Difficulty (1-5)	Estimation
1	Reading the description	4	Reading the description of the project	1	30 min
2	Create Repo	3	Creating git repo for the project	1	2 min

1. General analysis

1.1. Research about the STUPID principles:

As a first step you should do a search about the term STUPID that contains

the initials that refer to the following terms:

- S - Singleton
- T - Tight coupling
- U - Instability
- P - Premature Optimization
- I - Indescriptive Appointment
- D - Duplication

At a minimum, you must resolve the following questions:

- What is STUPID?

STUPID is an acronym that describes bad practices in Object-Oriented Programming.

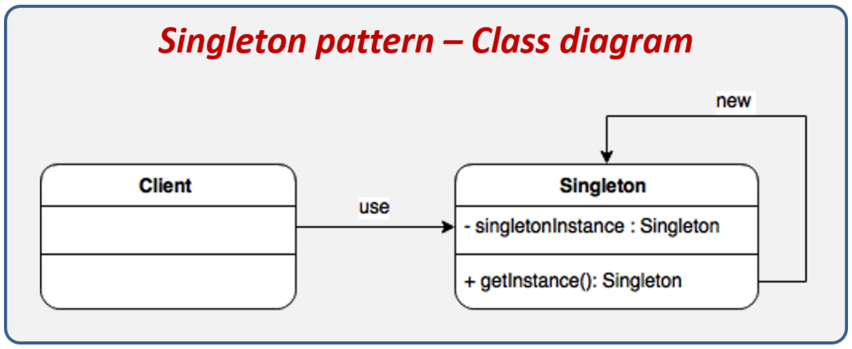
- Explain what each term that makes up "STUPID" refers to.

-S - Singleton:

The Singleton Pattern is probable the most well-known design pattern, but also the most misunderstood one. Singleton syndrome is when you think the Singleton Pattern is the most appropriate pattern for the current case you have. Singletons are controversial and often considered anti-patterns. Try to avoid them. The use of singleton is not the problem, but the symptom of a problem. Here are two reasons why:

❏ Programs using global state are very difficult to test;
❏ Programs that rely on global state hide their dependencies.

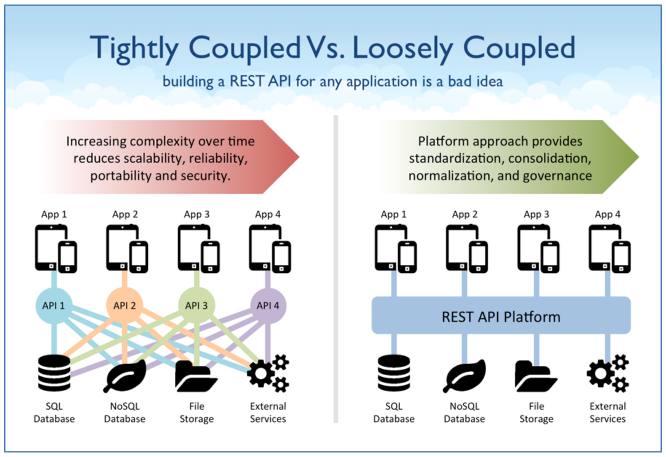
You should always avoid it. You can always replace singleton with something else. Avoiding static things is important in something called tight coupling.


Singleton pattern – Class diagram
new
Client
use
Singleton
- singletonInstance : Singleton
+ getInstance(): Singleton

-T - Tight coupling:

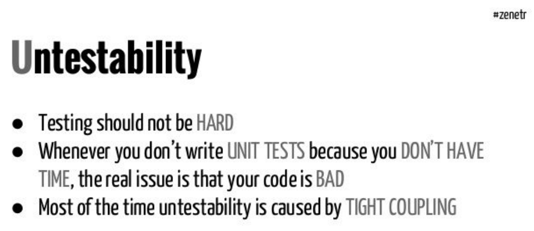
Tight coupling, also known as strong coupling, is a generalization of the Singleton issue. Basically you should reduce coupling between your modules. Coupling is the degree to which each program module relies on each one of the other modules.

If making a change in one module in your application requires you to change to another module, then coupling exists. For instance, you instantiate objects in your constructor's class instead of passing instances as arguments. That is because it doesn't allow further changes such as replacing the instance by an instance of a sub-class, a mock or whatever. Tightly coupled modules are difficult to use and hard to test.


Tightly Coupled Vs. Loosely Coupled
building a REST API for any application is a bad idea
Increasing complexity over time reduces scalability, reliability, portability and security.
Platform approach provides standardization, consolidation, normalization, and governance
App 1
App 2
App 3
App 4
API 1
API 2
API 3
API 4
REST API Platform
SQL Database
NoSQL Database
File Storage
External Services

-U - Untestability:

In my opinion, testing should not be hard! No, really. Whenever you don't write unit tests because you don't have time, the real issue is that your code is bad, but that is another story. Most of the time, untestability is caused by tight coupling.


#zenetr

Untestability

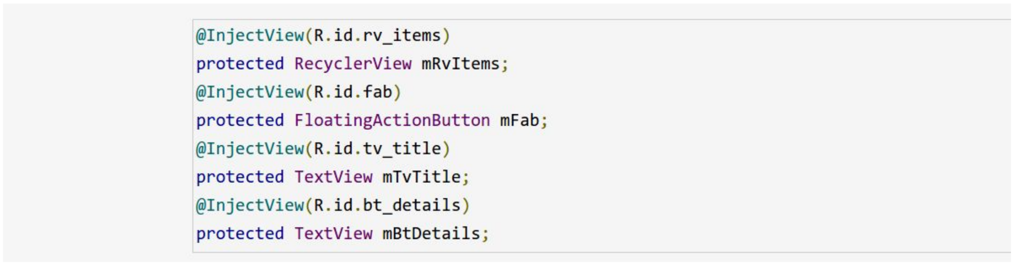
- Testing should not be HARD
- Whenever you don't write UNIT TESTS because you DON'T HAVE TIME, the real issue is that your code is BAD
- Most of the time untestability is caused by TIGHT COUPLING

-P - Premature Optimization:

Donald Knuth said: “Premature optimization is the root of all evil. There is only cost and no benefit”. Actually, optimized systems are much more complex than just rewriting a loop or using pre-increment instead of post-increment. You will just end up with unreadable code. That is why Premature Optimization is often considered an anti-pattern.

-I - Indescriptive Appointment:

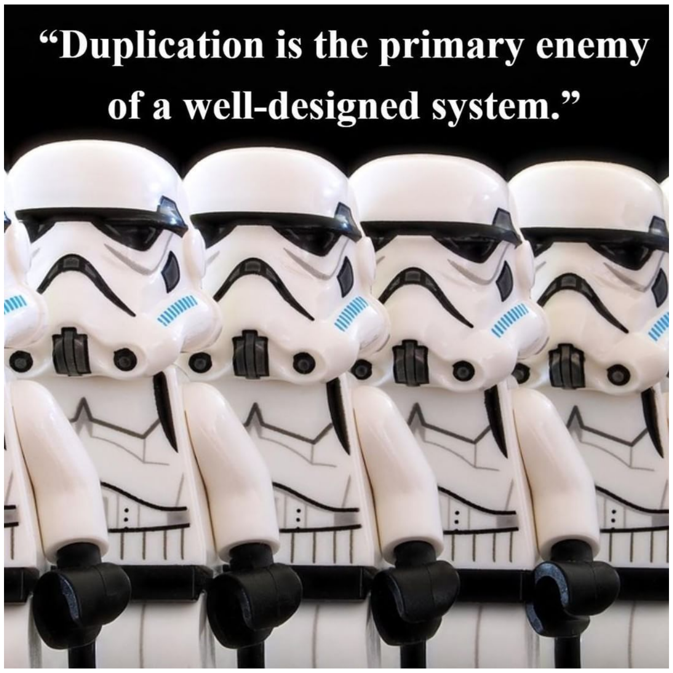
This should be obvious, but still needs to be said: name your classes, methods, attributes and variables properly, and don’t abbreviate. You write code for people not for computers. They don’t understand what you write anyway. Computers just understand 0 and 1. Programming languages are for humans.



```
@InjectView(R.id.rv_items)
protected RecyclerView mRvItems;
@InjectView(R.id.fab)
protected FloatingActionButton mFab;
@InjectView(R.id.tv_title)
protected TextView mTvTitle;
@InjectView(R.id.bt_details)
protected TextView mBtDetails;
```

-D - Duplication:

Duplicated code is bad, so remember to **Don’t Repeat Yourself (DRY) and also, Keep It Simple, Stupid (KISS).** Be lazy the right way, write code only once.


“Duplication is the primary enemy of a well-designed system.”

- Why is it advisable not to use the STUPID principle?

1.2. Research the SOLID principles:

As a second step you should do a search about the term SOLID that contains the initials that refer to the following terms:

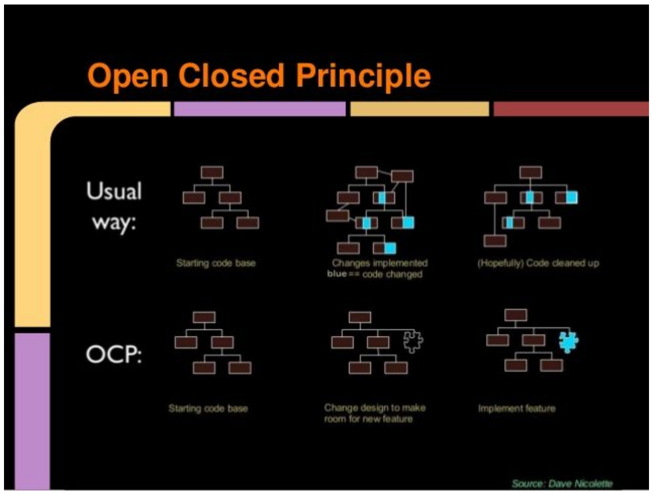
- S - Principle of Single Responsibility (SRP):

The Principle of Single Responsibility is a computer-programming principle that states that every module or class.should have responsibility over a single part of the functionality provided by the software, and that responsibility should be entirely encapsulated by a class, module or function. Like Uncle Bob says, “a class should have only one reason to change”. An example of this is to refactor a class with two methods into two classes with one method each.

SINGLE RESPONSIBILITY PRINCIPLE
Every object should have a single responsibility, and all its services should be narrowly aligned with that responsibility.

- O - Open/Closed Principle (OCP)

The Open/Closed Principle states that “software entities (classes, modules, functions, etc) should be open for extension, but closed for modification”; that is, such an entity can allow its behaviour, to be extended without modifying its source code. The name open/closed principle has been used in two ways. Both ways use generalizations (for instance, inheritance or delegate functions) to resolve the dilemma, but the goals, results and techniques are different.



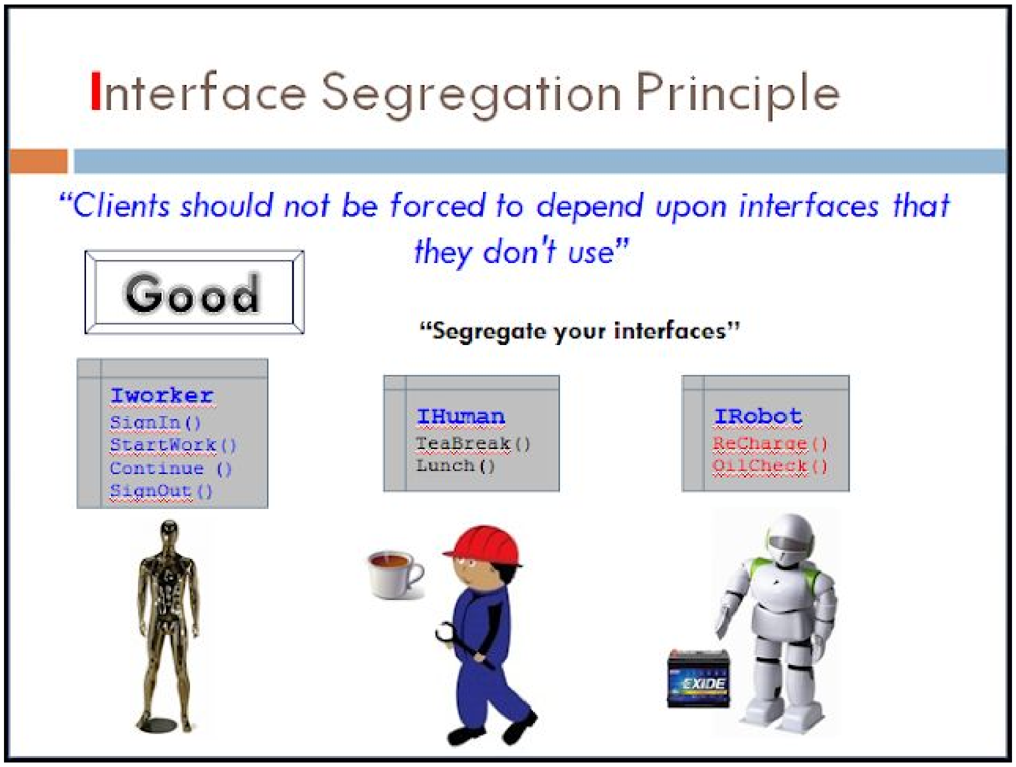
- L - Liskov Substitution Principle (LSP):

LSP was introduced by computer scientist Barbara Liskov in 1987 in her conference “Data Abstraction”. The LSP extends the Open/Closed Principle by focusing on the behavior of a superclass and its subtypes. The principle defines that objects of a superclass shall be replaceable with objects of its subclasses without breaking the application. That requires the objects of your subclasses to behave in the same way as the objects of your superclass. An overriden method of a subclass needs to accept the same input parameter values as the method of the superclass. That means that you can implement less restrictive validation rules, but you are not allowed to enforce stricter ones in your subclass. Otherwise, any code that calls this method on an object of the superclass might cause an exception, if it gets called with an object of the subclass. Similar rules apply to the return value of the method. The return value of a method of the subclass needs to comply with the same rules as the return values of the methods of the superclass.

LISKOV SUBSTITUTION PRINCIPLE
If It Looks Like A Duck, Quacks Like A Duck, But Needs Batteries - You Probably Have The Wrong Abstraction

- Interface Segregation Principle (ISP):

This principle states that no 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 so that clients will only have to know about the methods it does not use. ISP splits interfaces that are very large into smaller and more specific ones so that clients will only have to know about the methods that are of interest to them. Such shrunken interfaces are also called role interfaces. ISP is intended to keep a system decoupled and thus easier to refactor, change and redeploy.



Interface Segregation Principle

"Clients should not be forced to depend upon interfaces that they don't use"

Good

"Segregate your interfaces"

```
Iworker
SignIn()
StartWork()
Continue ()
SignOut()
```

```
IHuman
TeaBreak()
Lunch()
```

```
IRobot
ReCharge()
OilCheck()
```

- Dependency Investment Principle (DIP):

The goal of the Interface Segregation Principle is to reduce the side effects and frequency of required changes by splitting the software into multiple, independent parts.

Similar to the Single Responsibility Principle, the goal of the Interface Segregation Principle is to reduce the side effects and frequency of required changes by splitting the software into multiple, independent parts.

At a minimum, you must resolve the following questions:

- What is SOLID?

In Object-Oriented Computer Programming, SOLID is a mnemonic acronym for five design principles intended to make software design more understandable, flexible and maintainable. It is not related to the GRASP software design principles. The principles are a subset of many principles promoted by American software engineer and instructor Robert C. Martin, also known as Uncle Bob. SOLID principles can also form a core philosophy for methodologies such as agile development or adaptive software development.

- Explain what each term that makes up "SOLID" refers to.

-Single responsibility principle

A class should only have a single responsibility, that is, only changes to one part of the software's specification should be able to affect the specification of the class.

-Open–closed principle

"Software entities ... should be open for extension, but closed for modification."

-Liskov substitution principle

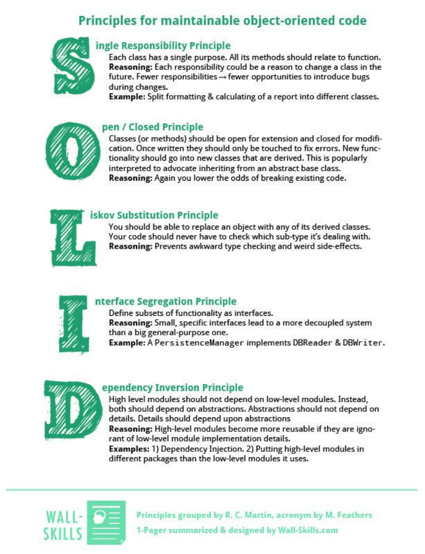
"Objects in a program should be replaceable with instances of their subtypes without altering the correctness of that program." See also design by contract.

-Interface segregation principle

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

-Dependency inversion principle

One should "depend upon abstractions, [not] concretions."



Principles for maintainable object-oriented code

Single Responsibility Principle
Each class has a single purpose. All its methods should relate to function.
Reasoning: Each responsibility could be a reason to change a class in the future. Fewer responsibilities → fewer opportunities to introduce bugs during changes.
Example: Split formatting & calculating of a report into different classes.

Open / Closed Principle
Classes (or methods) should be open for extension and closed for modification. Once written they should only be touched to fix errors. New functionality should go into new classes that are derived. This is popularly interpreted to advocate inheriting from an abstract base class.
Reasoning: Again you lower the odds of breaking existing code.

Liskov Substitution Principle
You should be able to replace an object with any of its derived classes. Your code should never have to check which sub-type it's dealing with.
Reasoning: Prevents awkward type checking and weird side-effects.

Interface Segregation Principle
Define subsets of functionality as interfaces.
Reasoning: Small, specific interfaces lead to a more decoupled system than a big general-purpose one.
Example: A `PersistenceManager` implements `DBReader` & `DBWriter`.

Dependency Inversion Principle
High level modules should not depend on low-level modules. Instead, both should depend on abstractions. Abstractions should not depend on details. Details should depend upon abstractions
Reasoning: High-level modules become more reusable if they are ignorant of low-level module implementation details.
Examples: 1) Dependency Injection. 2) Putting high-level modules in different packages than the low-level modules it uses.

WALL-SKILLS

Principles grouped by R. C. Martin, acronym by M. Feathers
1-Pager summarized & designed by Wall-Skills.com

- Why is it advisable to use the SOLID principle?

SOLID is an acronym that represents five principles very important when we develop with the OOP paradigm, in addition it is an essential knowledge that every developer must know.

Understanding and applying these principles will allow you to write better quality code and therefore be a better developer.

Here are the benefits of applying each of the concepts of SOLID:

- **Single Responsibility**

If your code adheres to this one, it’s easier to be followed, understood, debugged, removed, and refactored. You can make more courageous changes. After some years you might want to (you’ll need to, you’ll be forced to) change things, and respecting this principle is going to be a vital key for easier changes. And if you break something, you break one thing (or fewer), not an entire system.

- **Open/Closed**

By following it, changes and usage of objects become cheap in matter of time and risks. Objects can evolve with the possibility of working on them step by step, when needed, and not always feeling like you have to rewrite everything. When you can’t rewrite, just leave the past behind to be rewritten in small steps.

- **Liskov Substitution**

It’s about many objects which can be easily replaced by objects of the same nature. New features around an existing one and changes are subjects of this principles. Integrate new objects fast.

- **Interface Segregation**

Somehow goes towards the single responsibility. Your system’s user (the developer) will have a clear way to use exactly what they need, instead of being forced to interact with functionalities they don’t need. By not forcing them to handle what they don’t need, they need less code. Less code, fewer problems, closer deadlines.

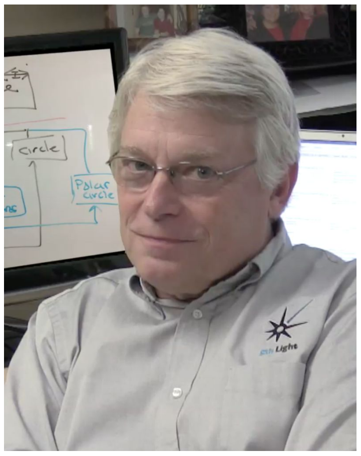
- **Dependency Inversion**

By sending dependencies from the outside world, you can change them more easily. And you know reasons for change: Libraries get really outdated or discontinued, business people change their mind, technical requirements change. If your code uses a dependency from outside (not only sent from outside, but also abstracted based on a contract), changing it will be cheaper. Also, you can test your units of code.

- Who introduced the concept of SOLID?

SOLID was introduced by Robert Cecil Martin, colloquially known as "Uncle Bob", an American software engineer and instructor. He is best known for being one of the authors of the Agile Manifesto and for developing several software design principles. He was also the editor-in-chief of C++ Report magazine and served as the first chairman of the Agile Alliance.

Martin operated the now-defunct company, Object Mentor, which provided instructor-led training courses about extreme programming methodology. He now operates two companies: Uncle Bob Consulting, which provides consulting and training services, and Clean Coders, which provides training videos.



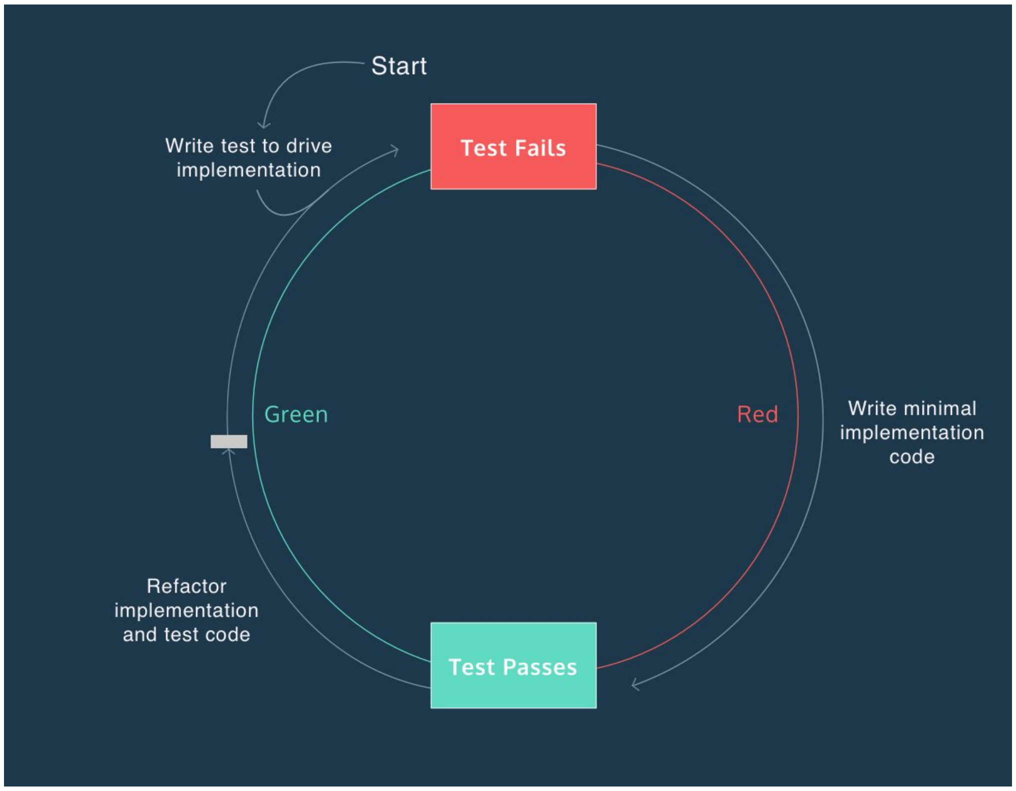
1.3. Research about "Think - Red - Green - Refactor":

Finally you should investigate about the TDD approach called red, green, refactor.

At a minimum, you must resolve the following questions:

- What is Red - Green - Refactor?

Test-driven development (TDD) is an approach to software development where you write tests first, then use those tests to drive the design and development of your software application.


Start
Write test to drive implementation
Test Fails
Green
Red
Write minimal implementation code
Refactor implementation and test code
Test Passes

- Explain what each term that makes up “Red - Green - Refactor” refers to.

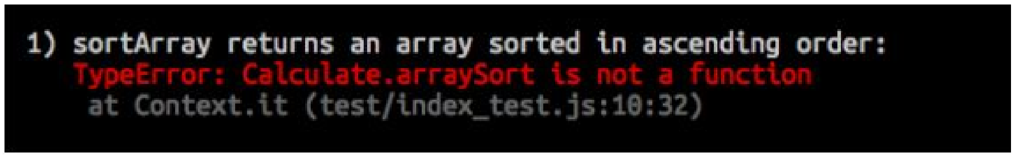
The red, green, refactor approach helps developers compartmentalize their focus into three phases:

- Red — think about what you want to develop.

For example, imagine you want to create a function called `sortArray` that sorts the numerical values of an array in ascending order. You may start by writing a test that checks the following input and output:

- Input: `[2, 4, 1]`
- Output: `[1, 2, 4]`

When you run this test, you may see an error message like:



```
1) sortArray returns an array sorted in ascending order:
   TypeError: Calculate.arraySort is not a function
    at Context.it (test/index_test.js:10:32)
```

As you can see, the purpose of this phase was to define what you want to implement. The resulting error message from this test informs your approach to implementation. At this point, you are considered “in the red.”

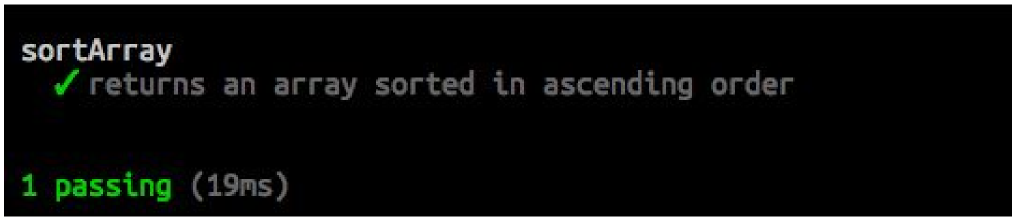
- Green — think about how to make your tests pass.

The green phase is where you implement code to make your test pass. The goal is to find a solution, without worrying about optimizing your implementation.

In our sortArray example, the goal is to accept an array like [2, 4, 1] and return [1, 2, 4].

I decide to approach the problem by writing a loop that iterates over the array, and moves the current number over if it is larger than the number to the right. I nest this loop inside of a loop that repeats until all of the numbers are sorted (the length of the array). Sorting an array with [3, 4, 5, 6, 2, 1] would look like:

After I implement this, I should see a passing message that looks like:



```
sortArray
  ✓ returns an array sorted in ascending order

1 passing (19ms)
```

At the end of this phase, you are consider “in the green.” You can begin thinking about optimizing your codebase, while having a descriptive test if you do something wrong.

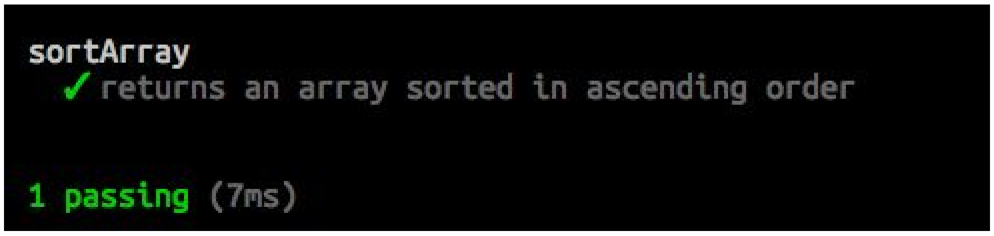
- Refactor — think about how to improve your existing implementation:

In the refactor phase, you are still “in the green.” You can begin thinking about how to implement your code better or more efficiently. If you are thinking about refactoring your test suite, you should always consider the characteristics of a good test, MC-FIRE. If you want to think about refactoring your implementation code, you can think about how to accomplish the same output with more descriptive or faster code.

Let’s take a moment to think about how I would approach refactoring my sortArray function. After doing some research of sorting methods, I find that I implemented the bubble sort method, which, it turns out, is not a particularly fast sorting method.

I choose to change the method I use in sortArray() to the merge sort algorithm, because it has a faster average sorting speed than bubble sort.

As I refactor sortArray(), I have the safety net of my test to notify me if my implementation goes off the tracks, and returns the wrong answer. When I complete my refactor and run my suite again, I receive the following output:



```
sortArray
  ✓ returns an array sorted in ascending order


1 passing (7ms)
```

Do you notice anything different about this output versus the output we received at the end of the green phase?

The new test code ran faster than before. The (7ms) next to 1 passing is shorthand for seven milliseconds, the amount of time it takes to run the test. At the end of the green phase, our test suite took nineteen milliseconds. Although a twelve millisecond difference is trivial here, the amount of time it takes to run a test suite increases considerably as your codebase grows.

- Why is this pattern recommended?

As your product grows and evolves the software powering it will need to change. The easier software is to change the better your job will be. Dancing the "red, green, refactor" dance is a way to keep software easy to change.

By taking the time to look at the code you just wrote and ask the question "how can this be improved" you'll end up with a better implementation than if you'd tried to get it right from the start. "Red, green, refactor" produces a development pace to helps keeping focus and momentum. By cleaning up as you go you'll set yourself up for success and reduce the need to spend time on large and complex refactors.

Technologies used

- Google
- Google Docs

Lessons learned

- SOLID is a mnemonic acronym for five design principles intended to make software designs more understandable, flexible and maintainable.
- Single **responsibility** principle - Class has one job to do. Each change in requirements can be done by changing just one class.
- **Open/closed** principle - Class is happy (open) to be used by others. Class is not happy (closed) to be changed by others.
- Liskov **substitution** principle - Class can be replaced by any of its children. Children classes inherit parent's behaviours.
- **Interface segregation** principle - When classes promise each other something, they should separate these promises (interfaces) into many small promises, so it's easier to understand.
- **Dependency** inversion principle - When classes talk to each other in a very specific way, they both depend on each other to never change. Instead classes should use promises (interfaces, parents), so classes can change as long as they keep the promise.

Incidents

- In the beginning, some difficulties understanding the basic concepts about SOLID and STUPID.