Object Oriented Design Principles

https://stackabuse.com/object-oriented-design-principles-in-java/

Prof. Dr. Sérgio Furgeri – sergio.furgeri@fatec.sp.gov.br

Some of the most important design principles in the object oriented paradigm are listed, but this is by no means an exhaustive list:

1. Don't Repeat Yourself (DRY)
2. Keep It Simple and Stupid (KISS)
3. The Single Responsibility Principle (SRP)
4. The Open/Closed Principle
5. Liskov Substitution Principle (LSP)
6. The Interface Segregation Principle (ISP)
7. The Dependency Inversion Principle (DIP)
8. The Composition Over Inheritance

https://stackabuse.com/object-oriented-design-principles-in-java/

Prof. Dr. Sérgio Furgeri – sergio.furgeri@fatec.sp.gov.br

1. Don't Repeat Yourself (DRY)

The Don't Repeat Yourself (DRY) is a common principle across programming paradigms, but it is especially important in OOP.

- Every piece of knowledge or logic must have a single, unambiguous representation within a system.
- This means utilizing abstract classes, interfaces, and public constants.
- Whenever there's a functionality common across classes, it either might make sense to abstract them away into a common parent class or use interfaces to couple their functionality.
- For an example, we'll be using these constants several times, and eventually we'll be changing their values manually to optimize some algorithm. It would be easy to make a mistake if we had to update each of these values at multiple places.

https://stackabuse.com/object-oriented-design-principles-in-java/

Prof. Dr. Sérgio Furgeri – sergio.furgeri@fatec.sp.gov.br

1. Don't Repeat Yourself (DRY)

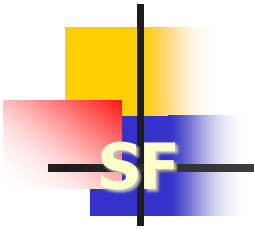
- Whenever there's a constant that's used multiple times, it's good practice to define it as a public constant:

```
public static final int VELOCIDADE_MAXIMA = 100;
public static final int MAXIMO_TENTATIVAS = 3;
```

- Also, we don't want to make a mistake and programmatically change these values during execution, so we're also introducing the final modifier.

- The purpose of this principle is to ensure easy **maintenance** of code, because when a functionality or a constant changes you have to edit the code only in one place.

- This not only makes the job easier, but ensures that mistakes won't happen in the future. You may forget to edit the code in multiple places, or somebody else who's not as familiar with your project may not know that you've repeated code and may end up editing it in just one place.



Prof. Dr. Sérgio Furgeri – sergio.furgeri@fatec.sp.gov.br

Violations of the DRY Principle

- Violations of the DRY Principle are often referred to as WET solutions. WET can be an abbreviation for multiple things:
  - We Enjoy Typing
  - Waste Everyone's Time
  - Write Every Time
  - Write Everything Twice
- WET solutions aren't always bad, as repetition is sometimes advisable in inherently dissimilar classes, or in order to make code more readable, less inter-dependent, etc.

https://stackabuse.com/object-oriented-design-principles-in-java/

Prof. Dr. Sérgio Furgeri – sergio.furgeri@fatec.sp.gov.br

2. Keep It Simple and Stupid (KISS)

- The principle is a reminder to keep your code simple and readable for humans. If your method handles multiple use-cases, split them into smaller functions. If it performs multiple functionalities, make multiple methods instead.

- The core of this principle is that for most cases, unless efficiency is extremely crucial, another stack call isn't going to severely affect the performance of your program.

- On the other hand, unreadable and long methods will be very hard to maintain for human programmers, bugs will be harder to find, and you might find yourself violating DRY.

https://stackabuse.com/object-oriented-design-principles-in-java/

Prof. Dr. Sérgio Furgeri – sergio.furgeri@fatec.sp.gov.br

2. Keep It Simple and Stupid (KISS)

- All in all, if you find yourself tangled up in your own code and unsure what each part does, it's time for reevaluation.

- if you are having trouble as the one who designed it while it's all still fresh in your mind, think about how somebody who sees it for the first time in the future will perform.

Prof. Dr. Sérgio Furgeri – sergio.furgeri@fatec.sp.gov.br

3. The Single Responsibility Principle (SRP)

- There should never be two functionalities in one class. Sometimes, it's paraphrased as: ”A class should only have one, and only one, reason to be changed”. Where a "reason to be changed" is the responsibility of the class. If there are more than one responsibilities, there are more reasons to change that class at some point. (Aluno → manter e matricula).

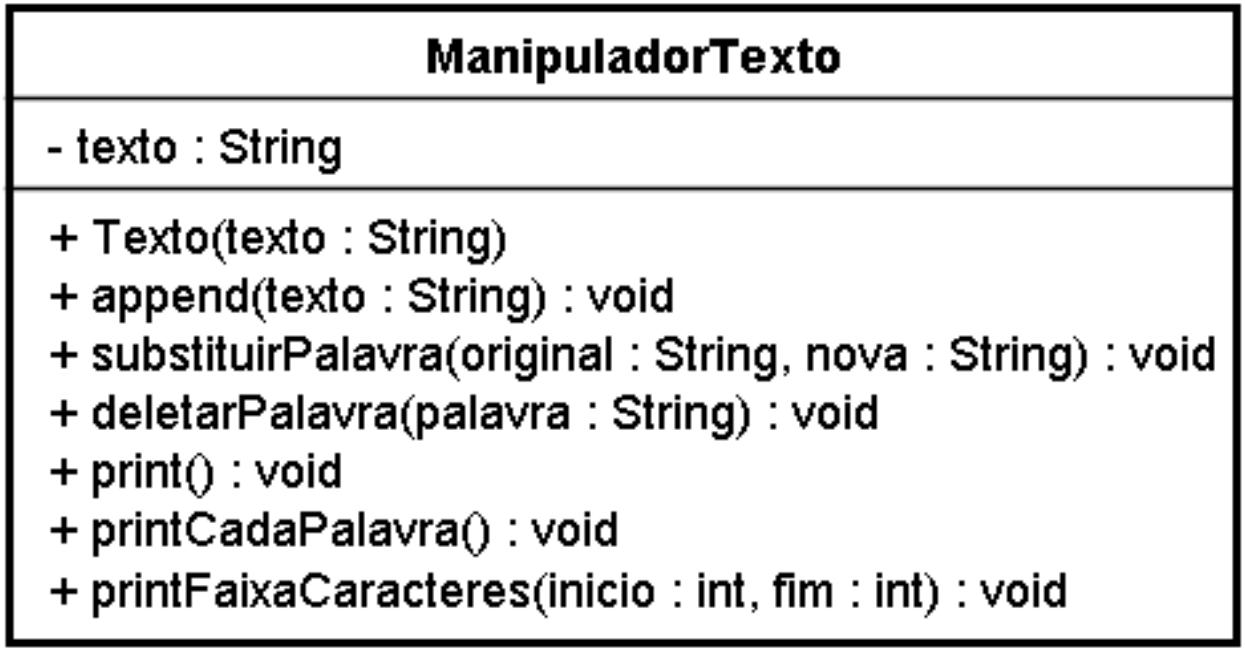
- This means that in the event of a functionality needing an update, there shouldn't be multiple separate functionalities in that same class that may be affected.

- This principle makes it easier to deal with bugs, to implement changes without confusing co-dependencies, and to inherit from a class without having to implement or inherit methods your class doesn't need.

https://stackabuse.com/object-oriented-design-principles-in-java/

Prof. Dr. Sérgio Furgeri – sergio.furgeri@fatec.sp.gov.br

3. The Single Responsibility Principle (SRP)



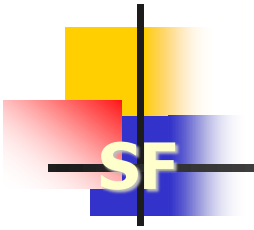
Prof. Dr. Sérgio Furgeri – sergio.furgeri@fatec.sp.gov.br

3. The Single Responsibility Principle (SRP)

- While it may seem that this encourages you to rely on dependencies a lot, this sort of modularity is much more important. Some level of dependency between classes is inevitable, which is why we also have principles and patterns to deal with that.

- For an example, say our application should retrieve some product information from the database, then process it and finally display it to the end-user. **We could use a single class to handle the database call, process the information and push the information to the presentation layer. Though, bundling these functionalities makes our code unreadable and illogical.**

- What we'd do instead is defining a class, such as ProductDAO that would fetch the product from the database, a ProductController to process the info and then we'd display it in a presentation layer - either an HTML page or another class/GUI.



Prof. Dr. Sérgio Furgeri – sergio.furgeri@fatec.sp.gov.br

4. The Open/Closed Principle

- The Open/Closed principle states that classes or objects and methods should be open for extension, but closed for modifications.

- What this means in essence is that you should design your classes and modules with possible future updates in mind, so they should have a generic design that you won't need to change the class itself in order to extend their behavior.

- You can add more fields or methods, but in such a way that you don't need to rewrite old methods, delete old fields and modify the old code in order to make it work again. Thinking ahead will help you write stable code, before and after an update of requirements.

- This principle is important in order to ensure backwards compatibility and prevent regressions - a bug which happens when your programs features or efficiency breaks after an update.

https://stackabuse.com/object-oriented-design-principles-in-java/

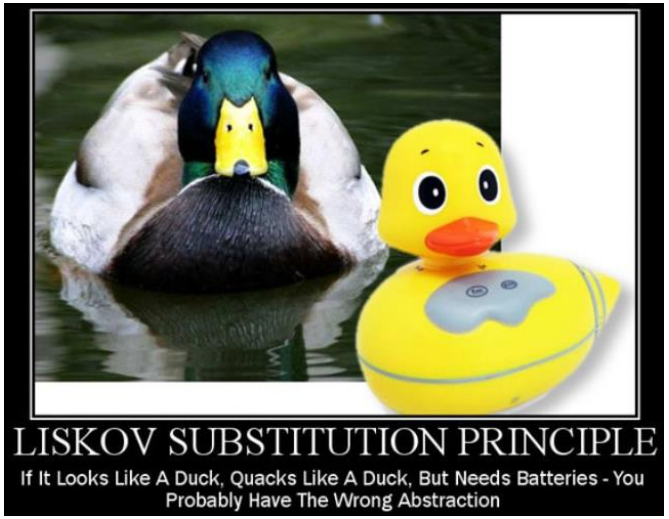
Prof. Dr. Sérgio Furgeri – sergio.furgeri@fatec.sp.gov.br

5. Liskov Substitution Principle (LSP)

- The Liskov Substitution principle was introduced by Barbara Liskov in her conference keynote "Data abstraction" in 1987.

- 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.



https://stackabuse.com/object-oriented-design-principles-in-java/

Prof. Dr. Sérgio Furgeri – sergio.furgeri@fatec.sp.gov.br

5. Liskov Substitution Principle (LSP)

- An overridden method of a subclass needs to accept the same input parameter values as the method of the superclass.
- That means you can implement less restrictive validation rules, but you are not allowed to enforce stricter ones in your 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 value of the method of the superclass.