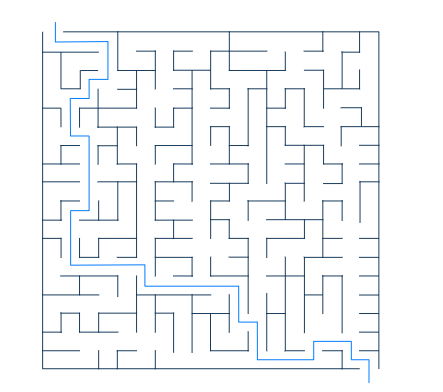
Creating software is like engineering: if a crucial part breaks, everything fails. How can we prevent such a disaster? Of course, we can test our programs, but is it sufficient? A program that passes all the tests might still have other problems:

* It can have a bad design, so no one uses it
* We cannot extend any part of it and add new features, because it's hard to understand how it works

In this topic, we will talk only about the inner design of programs and what helps us to improve and evolve it over time.

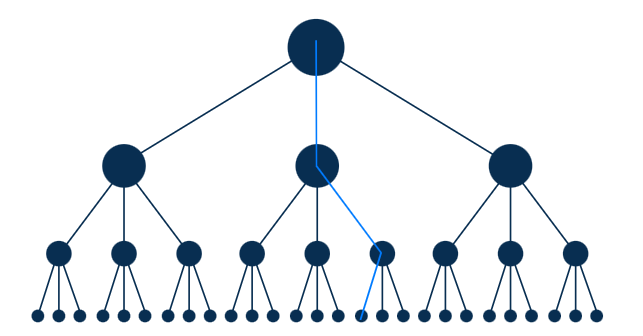
**Program design**

All programs manipulate data, and although all programs are different, let's think of them as pipelines with data: a program receives some input, processes it, and produces some output. A program can be like a maze:



The data goes from one function to another, and so on, and in the end, we get the result. It seems like we can control the data, but if there is an obstacle in our way, we are facing a complex problem. Moving only by straight paths doesn't save us from the growing complexity.

However, a program can have a different structure:



We keep the diversity of data paths but organize the code differently. We can say that we have another **design** for the program. The design of a program is the way to organize the code structure to achieve its primary goals.

Making a good design from scratch is not that simple, and we often don't know all its requirements at the very beginning. What we surely can do is follow guidelines or principles to make the design more effective and clear.

If you're working with a code base of several hundred lines of code, you can start with any design you like. Poor design decisions would hardly be such a huge problem for you.

**Design principles**

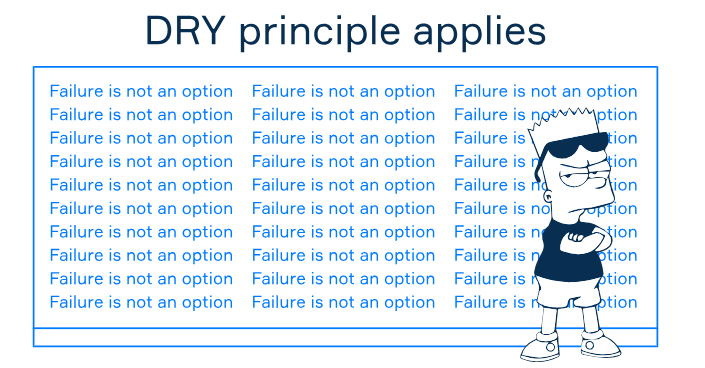
As we already know, applications have a design, but what can help us to make it better?

**Design principles**are rules that guide you to better decisions for the design of your program. You can refer to them when you want to add or update any part of the code.

Some of them come almost from common sense:

* Don't Repeat Yourself (**DRY**)
* You Ain't Gonna Need It (**YAGNI**)
* Keep It Simple, Stupid (**KISS**)

As you can guess, the *DRY* principle means, it's better to reuse code instead of copying it from one place to another. *YAGNI* is the principle of doing only the work that you need and not doing anything else. *KISS* stands for making the code simpler for better understanding.



Not all principles are easy to learn. Some of them, like **GRASP** (General Responsibility Assignment Software Patterns), are a whole ecosystem with many design patterns included and mostly adhere to object-oriented programming.

The other principles do not involve any additional knowledge, but it takes time to understand their meaning and significance. Let's look at the five software design principles hiding in the acronym — **SOLID.**

**SOLID**

There's no unanimous design principle to follow when designing your program. Why? Because as your program grows, it becomes more complex. You will need to familiarize yourself with different approaches to help manage your program through different kinds of complexities. To make the path to our goals smooth, we can rely on the *SOLID*design principles.

Each letter in SOLID refers to a distinct design principle:

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

That's the whole world behind this term! Each principle is independent of others, but applying them together works as a synergy for your design.

*SOLID* principles help you to organize your code in a way that any other developer familiar with these principles can use and extend it. Applying them, you make your code more structured and clean, so it doesn't look like a maze anymore.

Discussing each principle in detail is out of the scope of this topic, all of them deserve a whole independent topic for themselves.

**Conclusion**

To make a sustainable program that we can easily maintain for months or even years, we should think about its design first. Getting to work on it, we should apply our knowledge of design principles.

We can use our common sense with *DRY*, *YAGNI*, or *KISS*. However, when working on a huge project, it's better to use a more complex approach based on the set of *SOLID* principles.

## Handling Image Formats: Which Principle?

 Report a typo

Consider the following scenario: You are developing a software component that handles a variety of image files (JPEG, PNG, GIF, etc.). As per the design principles you've just learned, you intend to implement the software in such a way that adding support for new image formats in the future won't require you to revise the entire component. Which of the following software construction principles are you most directly applying in this case?

Select one option from the list

The Open-Closed Principle - Software entities should be open for extension, but closed for modification

Name the acronym for the principle that prevents you from writing unnecessary code.

YAGNI You Aren’t Gonna Need It (YAGNI)