# Pilares para a criação de um código, uma visão sobre padrões de projeto e qualidade.

# Pillars for creating code, insight into design standards and quality.

aws certified Developer ASSOCIATE

aws Technical Accredited PARTNER

aws Cloud Economics Accredited PARTNER

aws Business Accredited PARTNER

CertiProf

# Weder Mariano de Sousa

Analist System Senior - GFT Brazil

+ 16 Years of Experience
+ 50 Projects Worked
Post Graduate in Media UFG
Post Graduate in Information Security
Computer scientist
IT Technician ITGO

GOJava

AWS User Group Goiânia

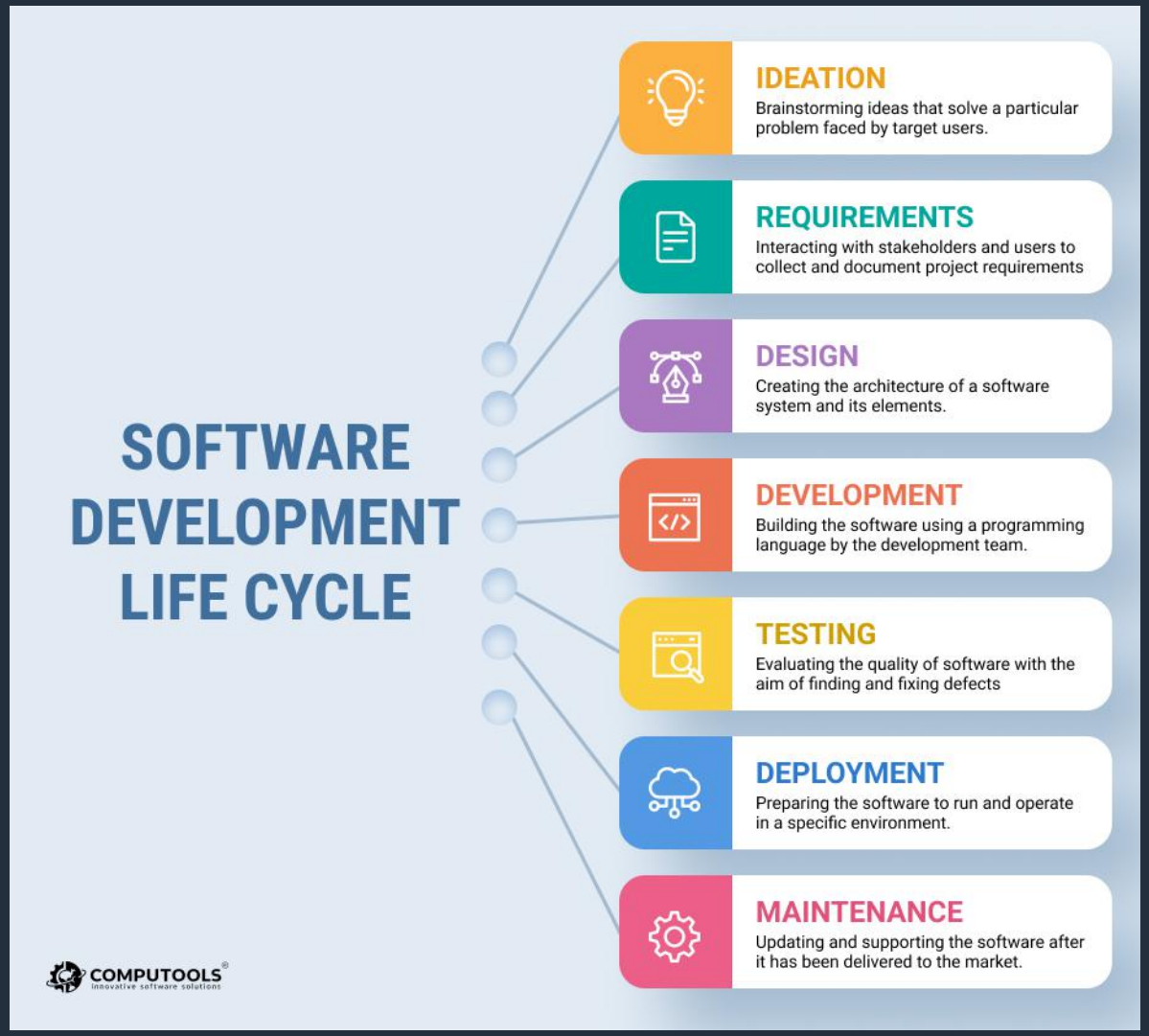
Grupy-GO

https://www.linkedin.com/in/wedermarianodesousa/
https://github.com/weder96

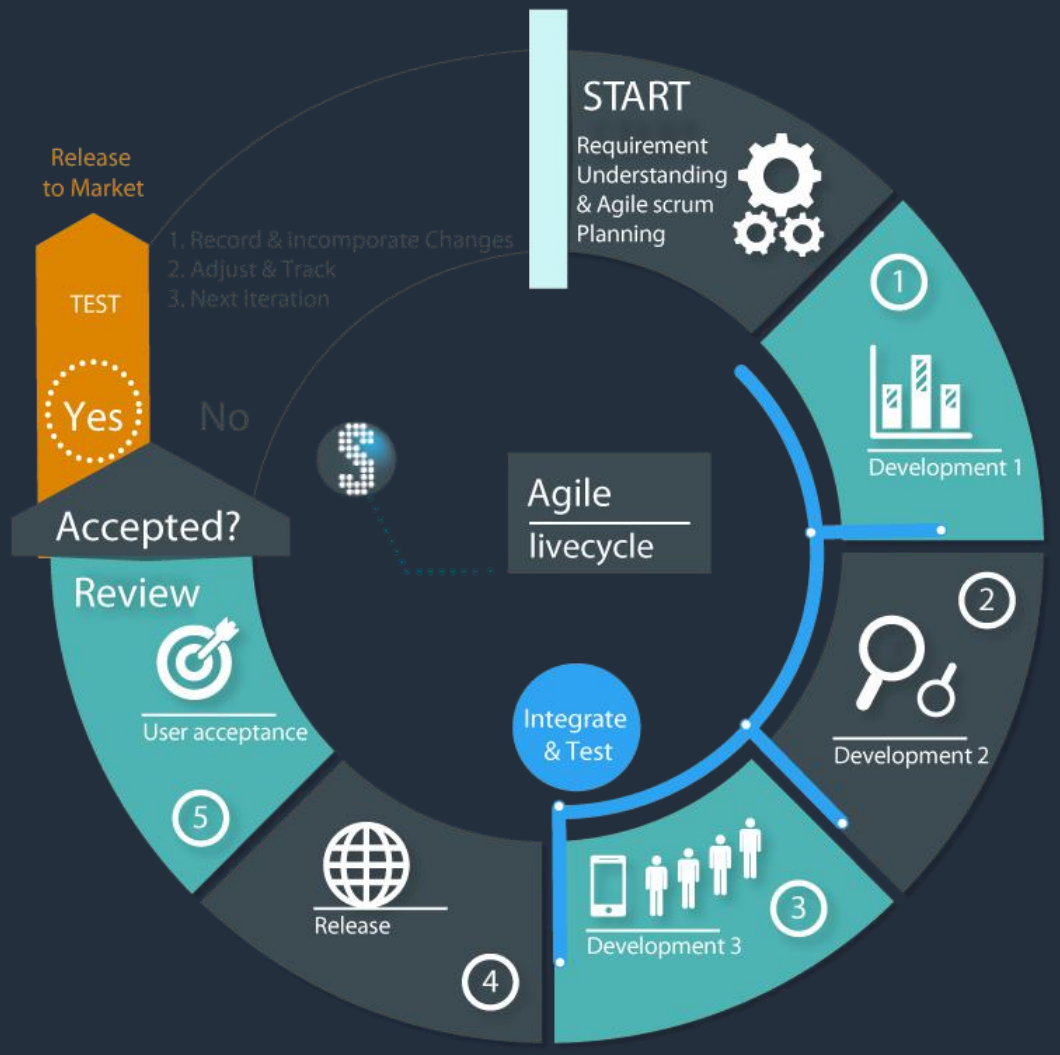
# Agenda

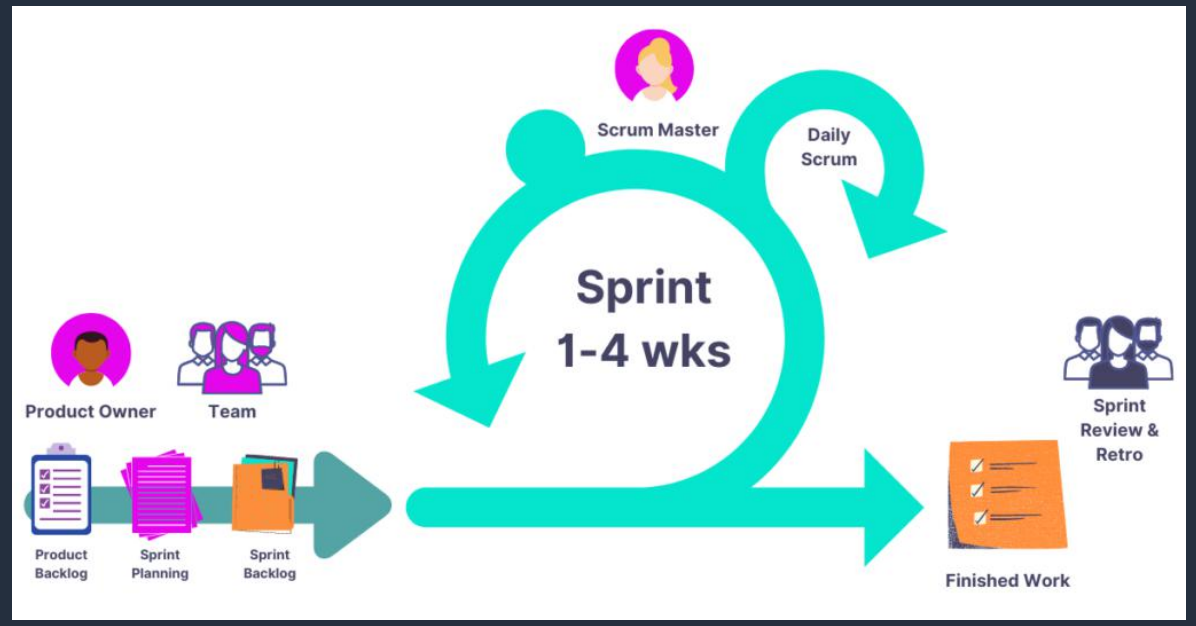
- Software Development Life Cycle
- Design Patterns
- Software Architecture
- Clean Architecture
- Monolithic
- Microservices
- Serverless
- Clean Code
- Pillars for Creating Code

# Software Development Life Cycle



# Agile Software Development Methodology

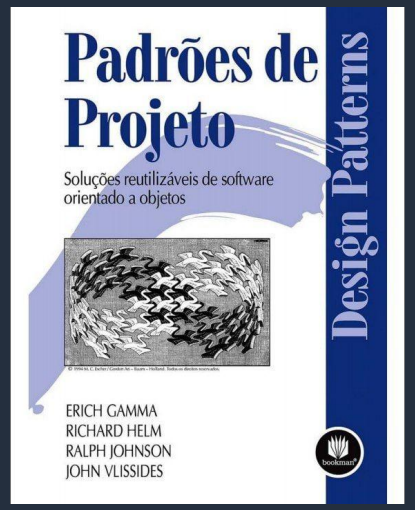



https://www.scrum.org/

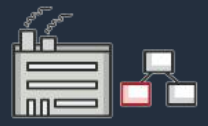

https://blog.geekhunter.com.br/manifesto-agil/

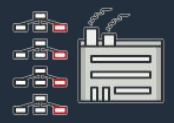
# DESIGN PATTERNS



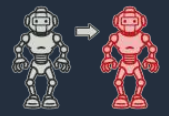
# Design Patterns Catalog

**Creational Patterns (5)**
These patterns provide various object creation mechanisms, which increase flexibility and reuse of existing code.


Factory Method

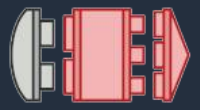

Abstract Factory


Bulder


Prototype

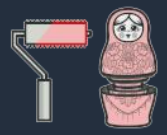

Singleton

**Structural Patterns (7)**
These patterns explain how to assemble objects and classes into larger structures while keeping these structures flexible and efficient.

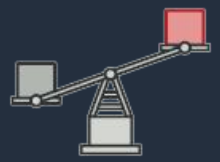

Adapter


Bridge


Composite


Decorator


Facade

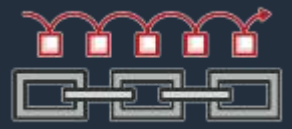

Flyweight


Proxy

https://refactoring.guru/design-patterns/catalog
https://github.com/weder96/javaaula21/tree/main/ProjectDesignPartners/src/main/java/Pack20DesignPatterns

# Design Patterns Catalog

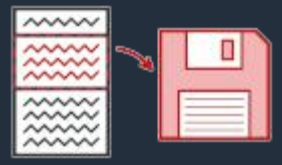
**Behavioral Patterns (10)**
These patterns are concerned with algorithms and the assignment of responsibilities between objects.


Chain of Responsability


Command


Iterator


Mediator


Memento


Observer


State


Strategy


Template Method


Visitor

https://refactoring.guru/design-patterns/catalog
https://github.com/weder96/javaaula21/tree/main/ProjectDesignPartners/src/main/java/Pack20DesignPatterns

# Design Patterns Catalog

**Architectural Standards**

- ☑ Data Access Object
- ☑ Flux
- ☑ Lazy Loading
- ☑ Model View Controller
- ☑ Model View Presenter
- ☑ Model View View Model

https://github.com/weder96/javaaula21/tree/main/ProjectDesignPartners/src/main/java/Pack20DesignPatterns

# Design Patterns Catalog

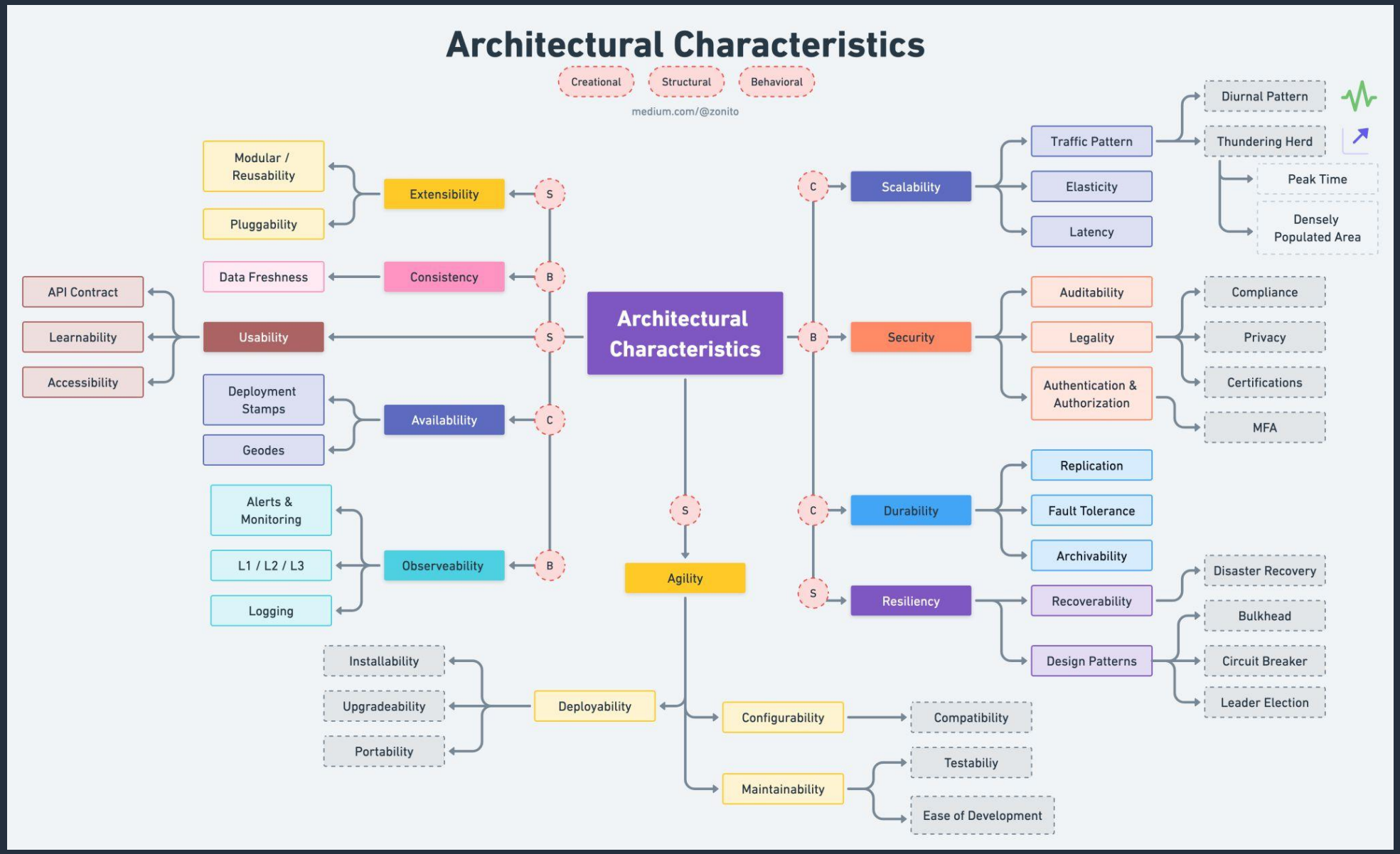
## Other Design Patterns

- Acyclicvisitor ☑
- Ambassador ☑
- Balking ☑
- Bytecode ☑
- Callback ☑
- Circuit Breaker ☑
- Collection Pipeline ☑
- Combinator ☑
- Data Mapper ☑
- Databus ☑
- Data Transfer ☑
- Delegate ☑
- Dependency Injection ☑
- Double Dispatch ☑
- Event Aggregator ☑
- EventQueue ☑
- Execute Around ☑
- Locality ☑
- Properrty ☑
- Private Class Data ☑
- monostate ☑
- multiton ☑
- mutex ☑
- nullobject ☑
- object Pool ☑
- pipeline ☑
- Resource Acquisition IsInitialization ☑
- Servant ☑
- Service Layer ☑
- Specification ☑
- Thread Pool ☑
- Tolerant Reader ☑
- Twin ☑

https://github.com/weder96/javaaula21/tree/main/ProjectDesignPartners/src/main/java/Pack20DesignPatterns

# Software Architecture

# Software Architecture




1. Scalability
2. Availability
3. Extensibility
4. Consistency
5. Resiliency
6. Usability
7. Observability
8. Security
9. Durability
10. Agility

https://blog.devgenius.io/top-10-architecture-characteristics-non-functional-requirements-with-cheatsheat-7ad14bbb0a9b

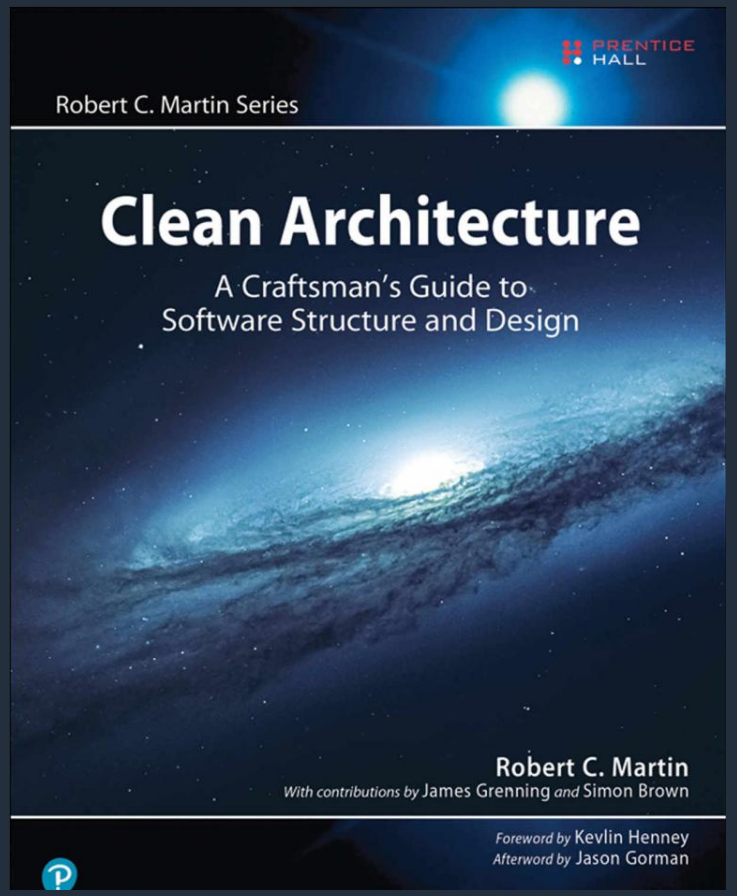
# Clean Architecture

# Clean Architecture

The goal of software architecture is to minimize the human resources required to build and maintain the required system.

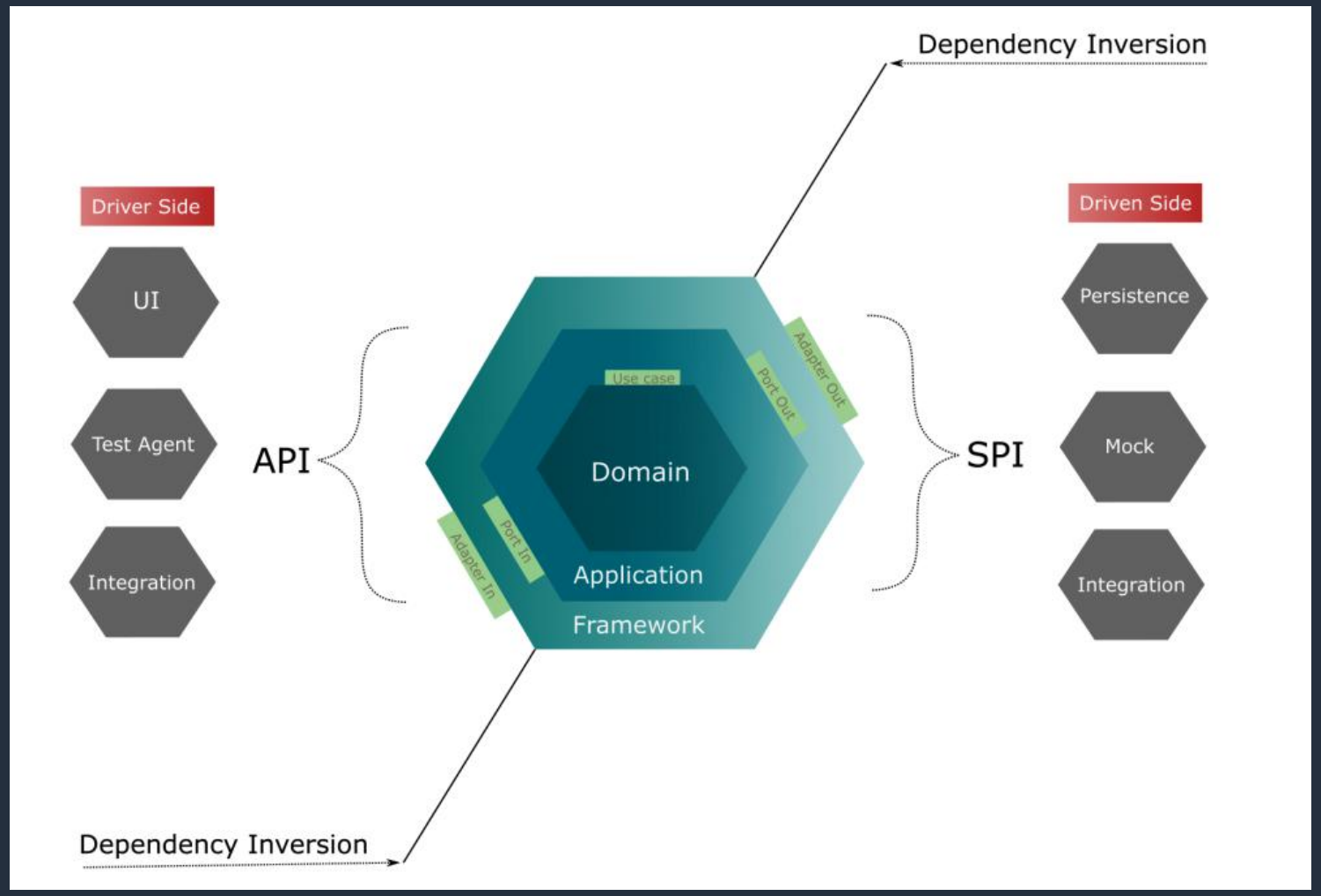
SOLID

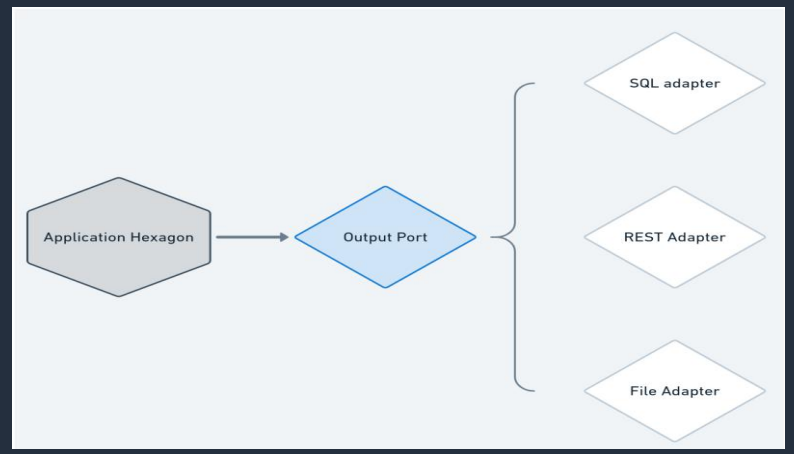
Hexagonal

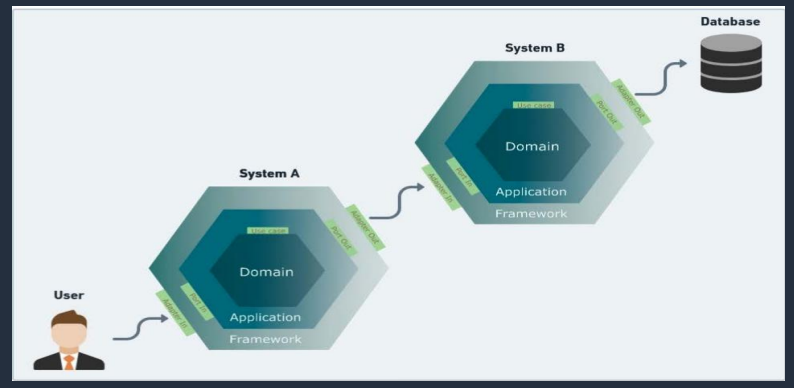



https://www.zup.com.br/blog/clean-architecture-arquitetura-limpa

# Understanding the Hexagonal Architecture



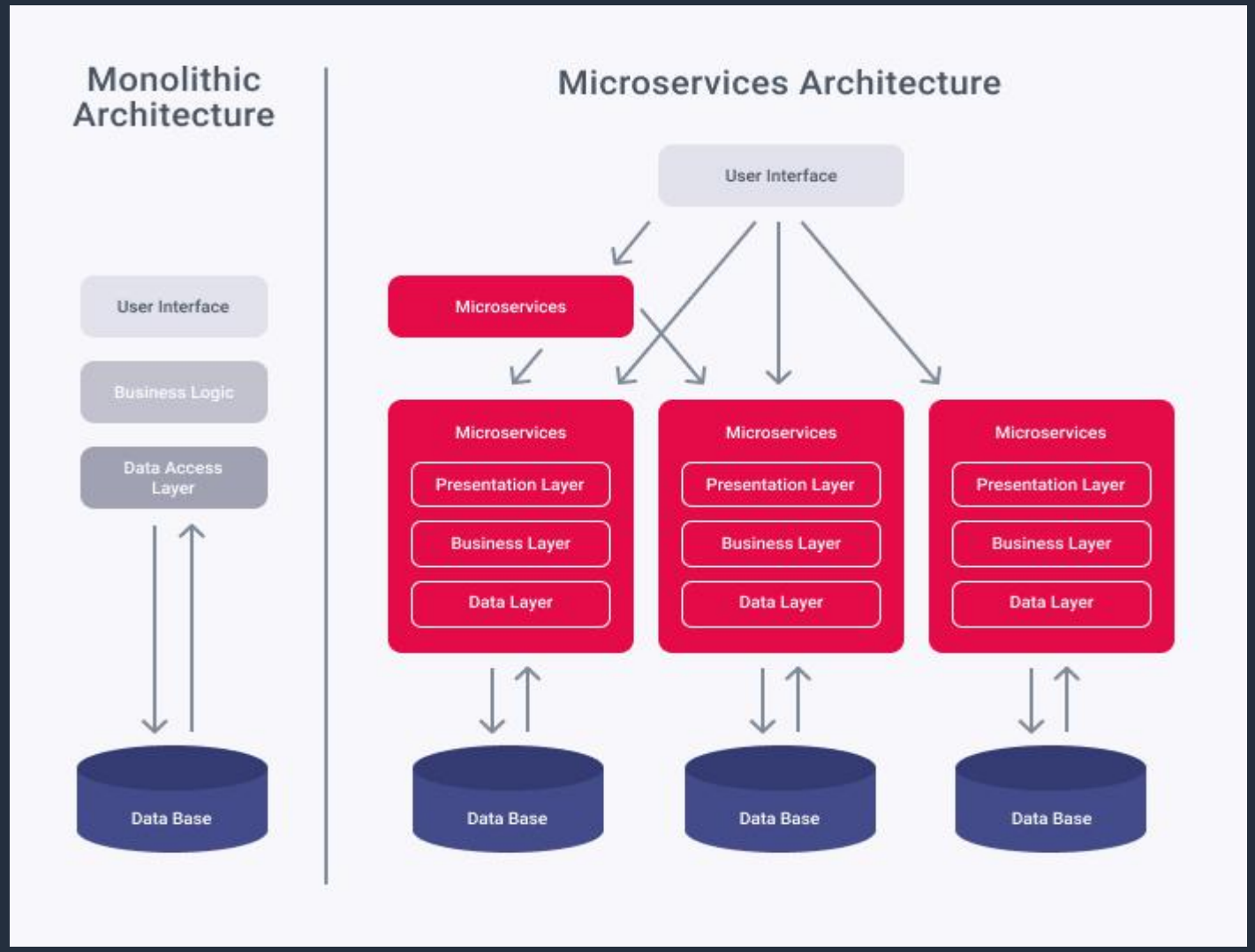

https://www.zup.com.br/blog/clean-architecture-arquitetura-limpa

Monolithic

Microservices

Serverless

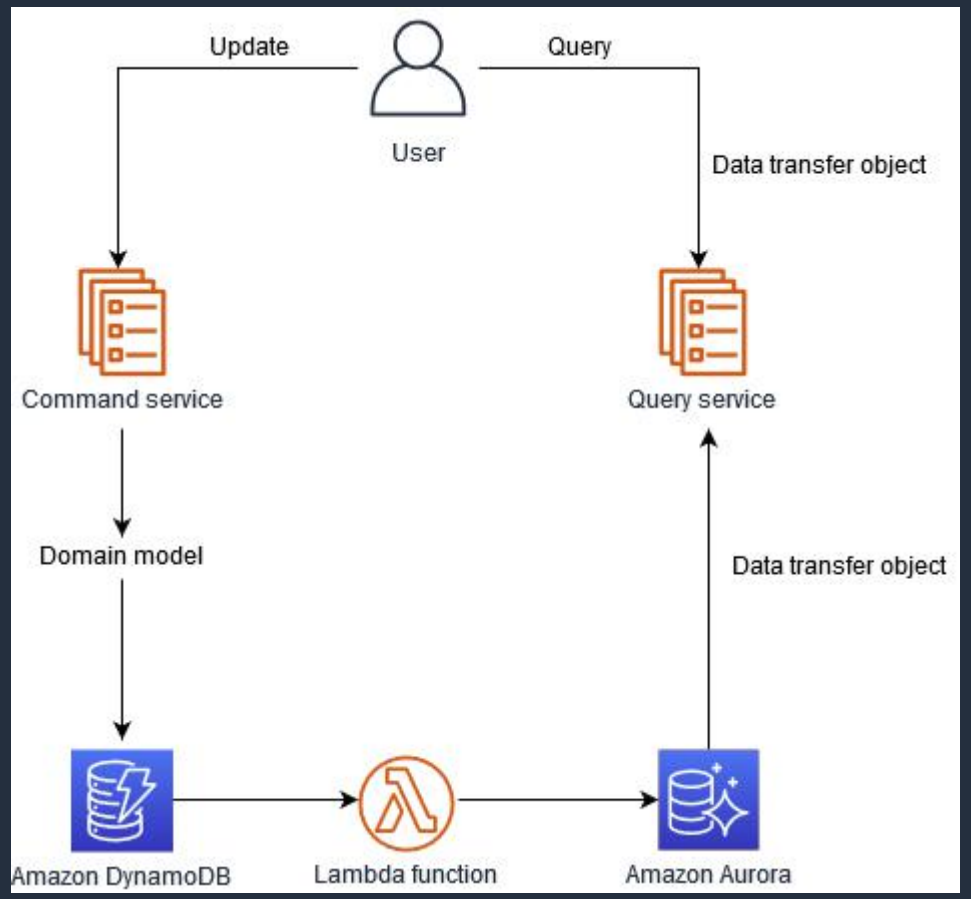
# Monolithic vs Microservices




https://xbsoftware.com/blog/microservices-vs-monolithic-architecture/
https://arquiteturadesoftware.online/integrando-sistemas-de-software/

# Microservices Management Database

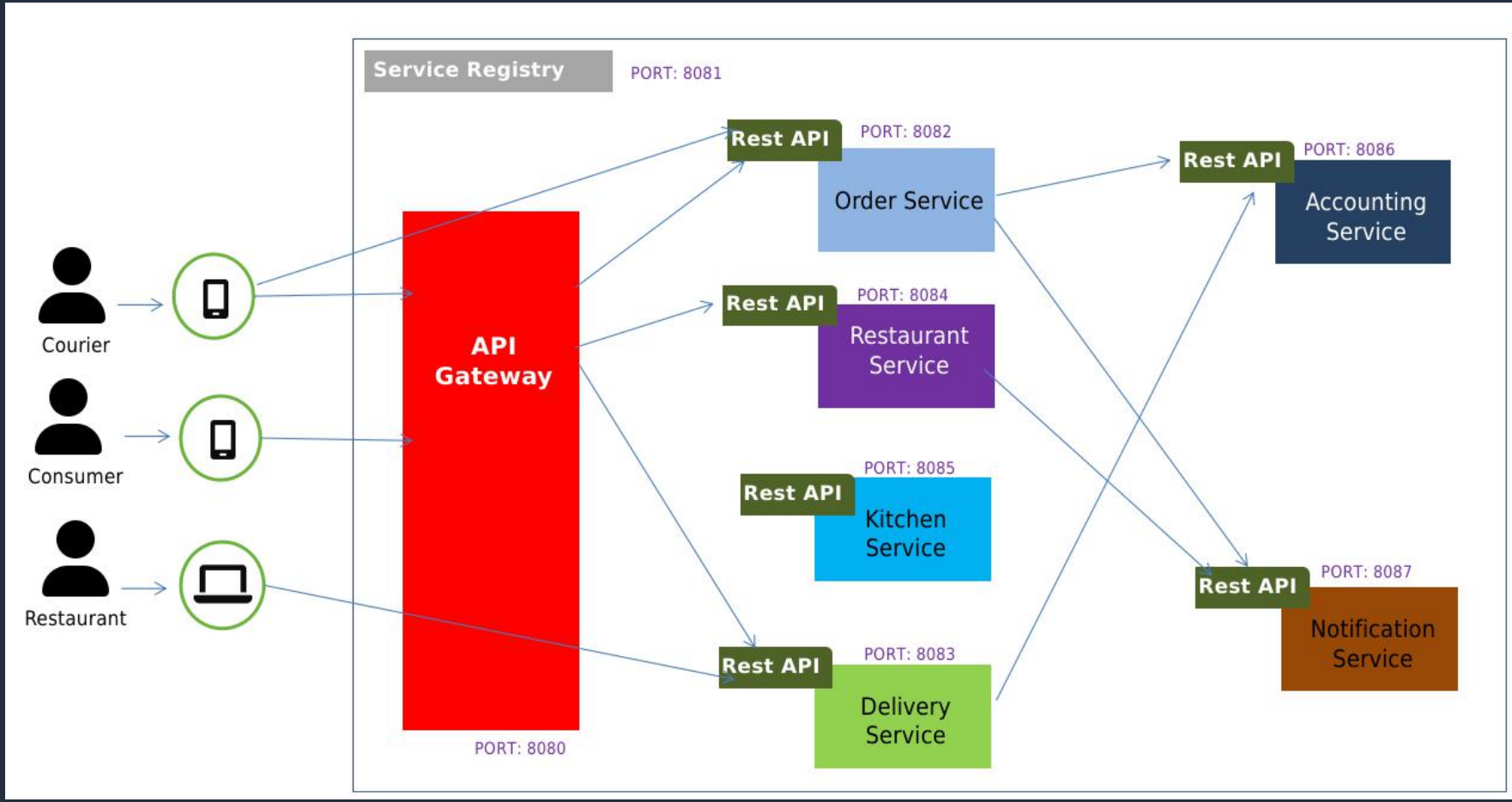
- ☑ Database per Service
- ☑ Shared database
- ☑ Saga
- ☑ API Composition (API Gateway)
- ☑ CQRS (Command Query Responsibility Segregation)
- ☑ Domain event
- ☑ Event sourcing




https://microservices.io/
https://docs.aws.amazon.com/prescriptive-guidance/latest/modernization-data-persistence/cqrs-pattern.html
https://github.com/weder96/microServiceDemo

# Microservices - Service Discovery

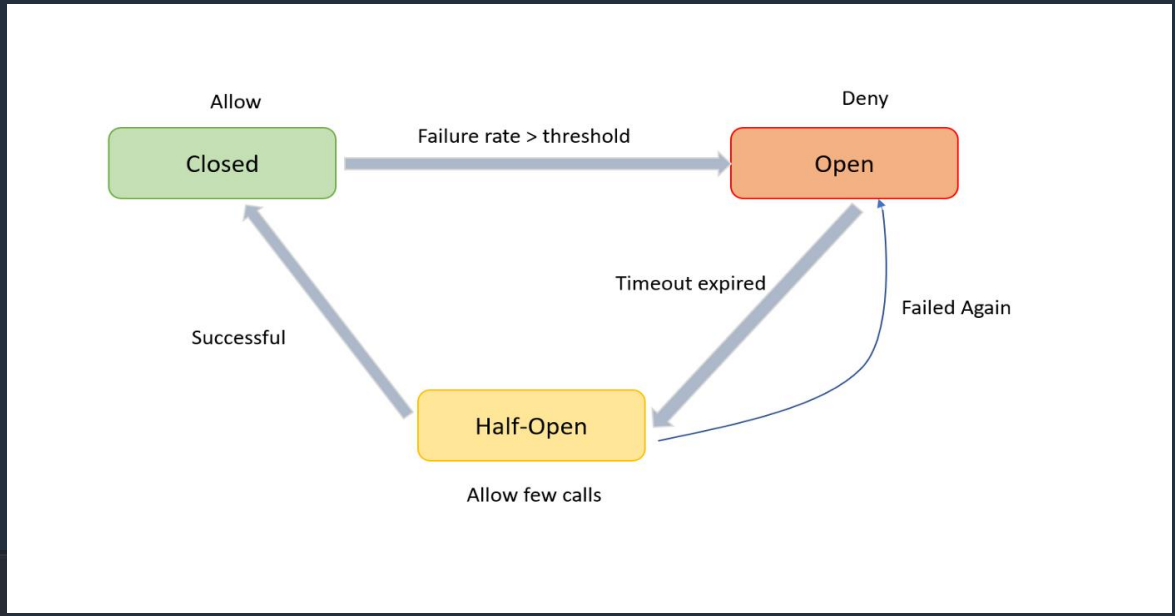
- ☑ Client-side discovery
- ☑ Server-side discovery
- ☑ Service registry
- ☑ Self registration
- ☑ 3rd party registration




https://microservices.io/
https://github.com/weder96/microServiceDemo
https://github.com/weder96/microServiceDemo/blob/main/documents/microservices-patterns-spring-projects-eureka-server-class-test.pdf

# Microservices - Reliability

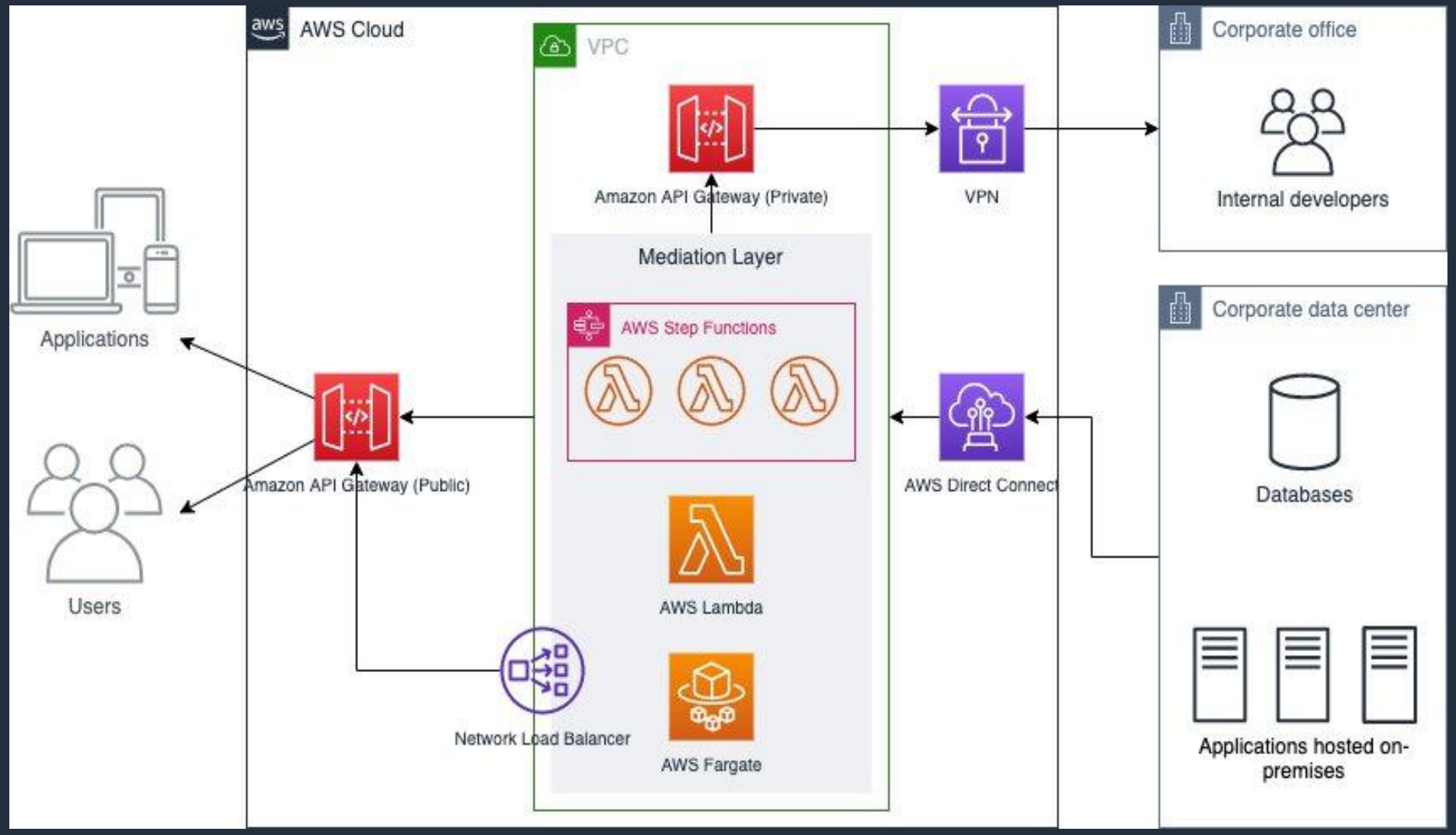
- ☑ Circuit Breaker




```
spring.application.name=order
server.port=8092

eureka.instance.prefer-ip-address=true
eureka.client.service-url.defaultZone=http://localhost:8091/eureka/
eureka.client.register-with-eureka=true

resilience4j.circuitbreaker.instances.orderCB.registerHealthIndicator = true
resilience4j.circuitbreaker.instances.orderCB.minimumNumberOfCalls = 4
resilience4j.circuitbreaker.instances.orderCB.slidingWindowSize= 100

api.url.accounting = http://localhost:8096/v1/accounting
```

https://microservices.io/
https://github.com/weder96/microServiceDemo/blob/main/order/src/main/java/com/wsousa/order/controller/OrderController.java

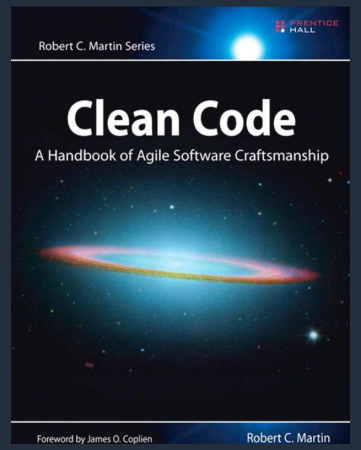
# Serverless Architecture




https://aws.amazon.com/pt/blogs/architecture/unlocking-data-from-existing-systems-with-serverless-api-facade/

# Clean Code

# Clean Code




Why are we talking so much about clean code (Clean Code) and why is this so important to us? In fact the maintenance of a software is as important as its construction.

As reported by **Robert C. Martin** in his classic book, **Clean Code**, a Best Seller in our area, some practices and visions are very important to maintain the life of our software.

Companies invest millions in software every year, but with so many changes in the team and technologies, how to make this investment last? How to guarantee a good maintenance, durability, life to the software? .

https://balta.io/artigos/clean-code
https://www.sydle.com/br/blog/clean-code-602bef23da4d09680935509b/

# S.O.L.I.D - The 5 Principles of OOP

SRP — Single Responsiblity Principle (Princípio da responsabilidade única)

OCP — Open-Closed Principle (Princípio Aberto-Fechado)

LSP — Liskov Substitution Principle (Princípio da substituição de Liskov)

ISP — Interface Segregation Principle (Princípio da Segregação da Interface)

DIP — Dependency Inversion Principle (Princípio da inversão da dependência)

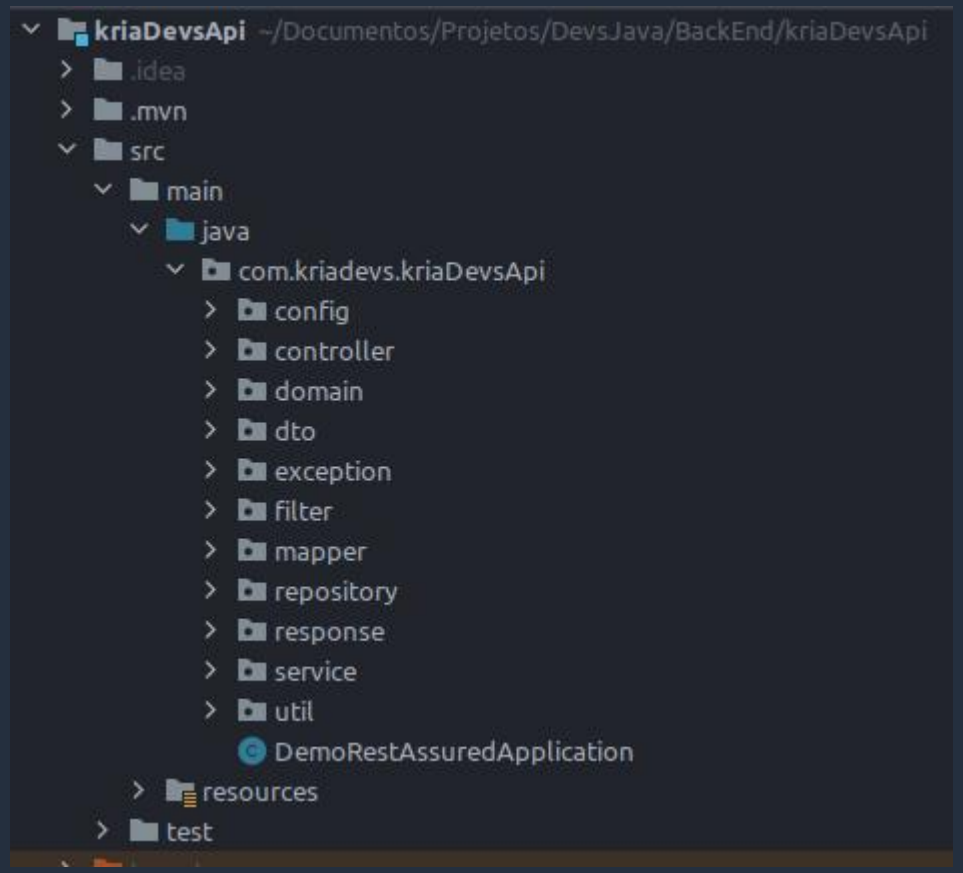
# SRP — Single Responsiblity Principle

**SRP: The Single Responsibility Principle**

A module should have one, and only one, reason to change.

A module should be responsible to one, and only one, user or stakeholder.

A module should be responsible to one, and only one, actor.

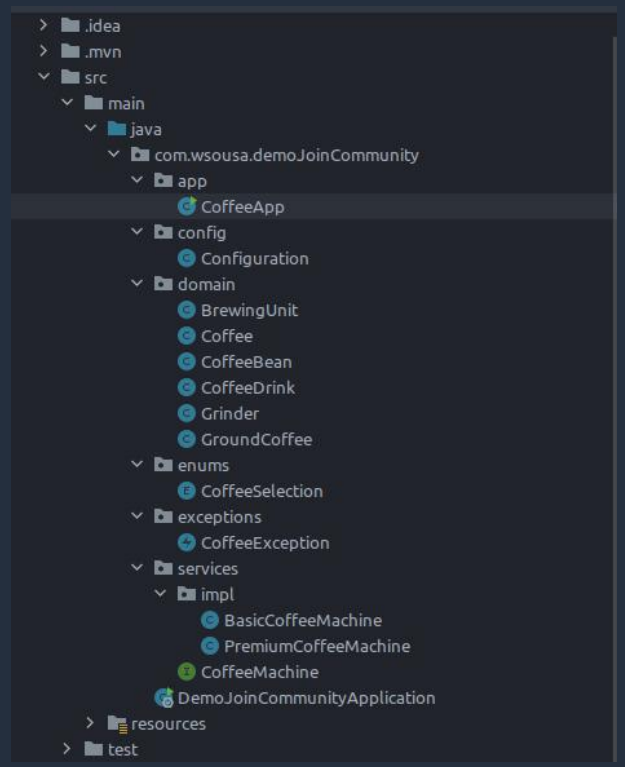


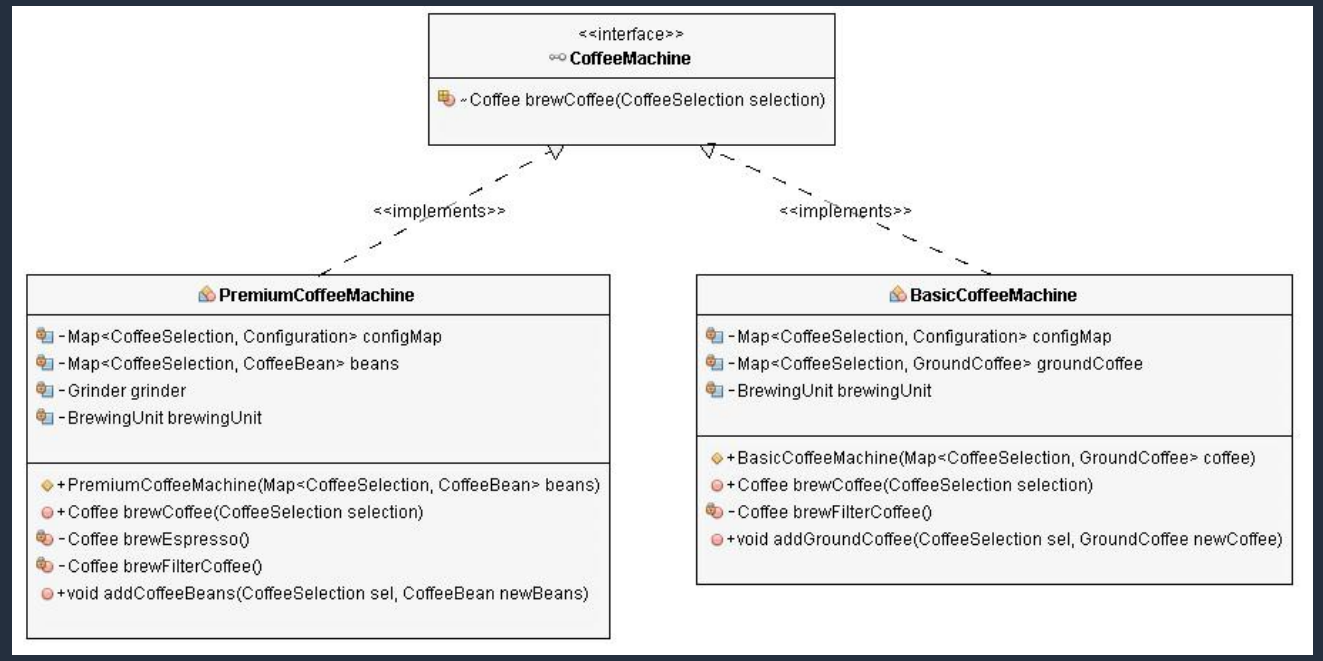

The Single Responsibility Principle (part of SOLID, by Robert C. Martin 2000)

# OCP — Open-Closed Principle

*“Software entities (classes, modules, functions, etc.) should be open for extension, but closed for modification.”*

*“A class is closed, since it may be compiled, stored in a library, baselined, and used by client classes. But it is also open, since any new class may use it as parent, adding new features. When a descendant class is defined, there is no need to change the original or to disturb its clients.”*




https://stackify.com/solid-design-open-closed-principle/

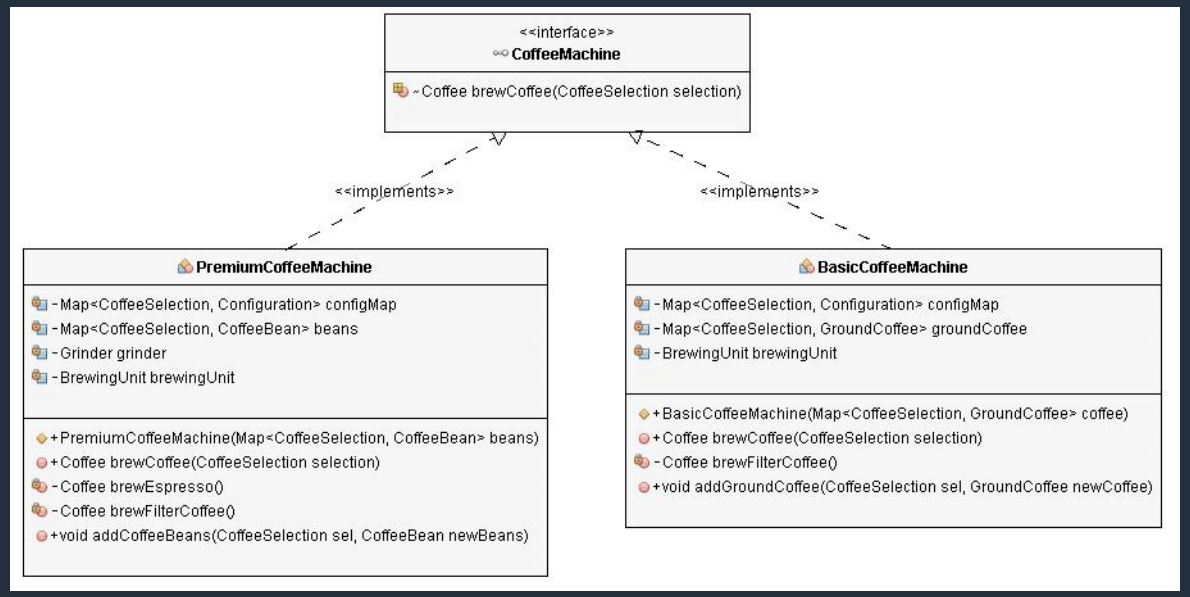
# LSP — Liskov Substitution Principle

*“Let Φ(x) be a property provable about objects x of type T. Then Φ(y) should be true for objects y of type S where S is a subtype of T.”*

The Liskov Substitution Principle is the third of Robert C. Martin’s SOLID design principles. It **extends** the **Open/Closed principle** and enables you to replace objects of a parent class with objects of a subclass without breaking the application. This requires all subclasses to behave in the same way as the parent class. To achieve that, your subclasses need to follow these rules:

Don’t implement any stricter validation rules on input parameters than implemented by the parent class.

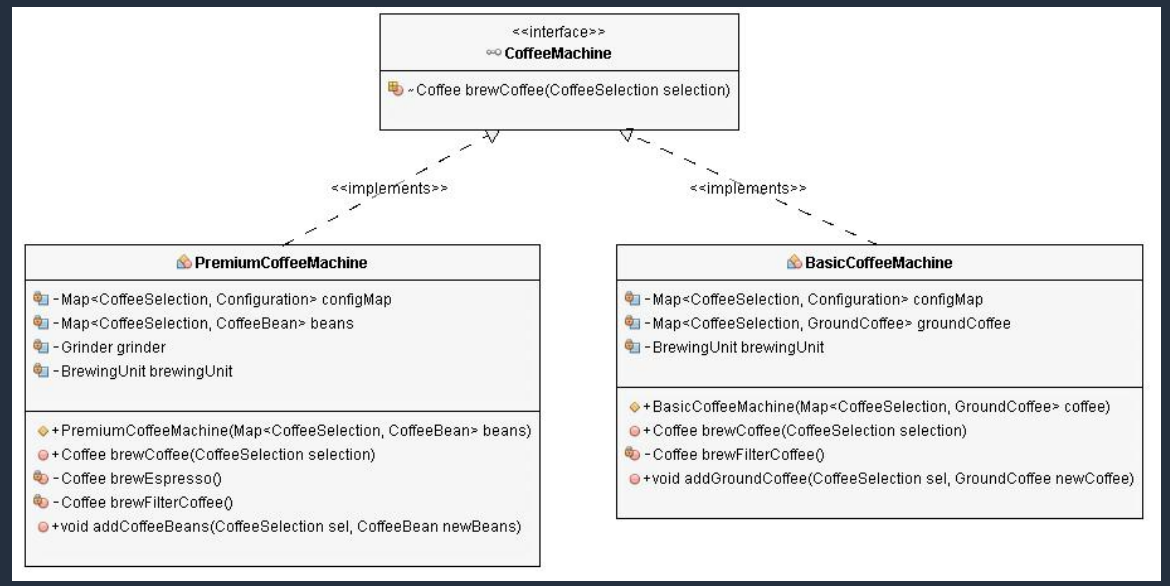
Apply at the least the same rules to all output parameters as applied by the parent class.

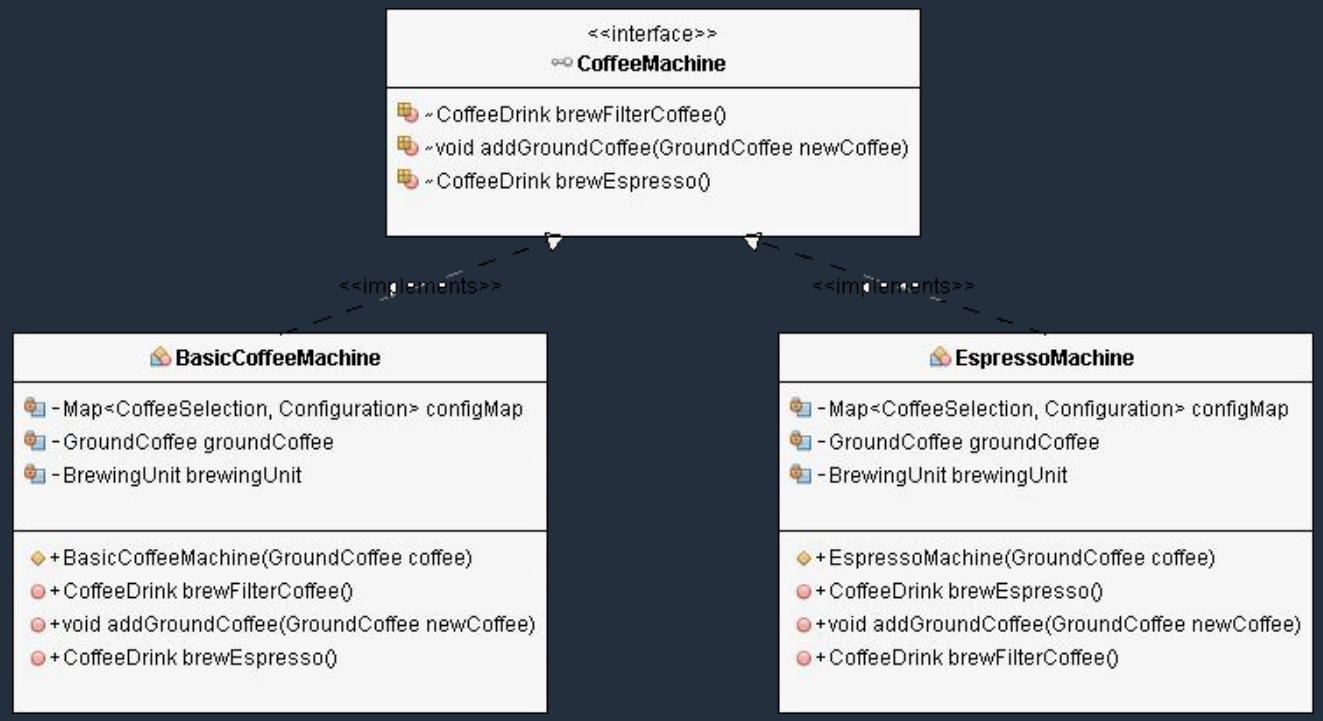



https://stackify.com/solid-design-liskov-substitution-principle/

# ISP — Interface Segregation Principle

"Clients should not be forced to depend upon interfaces that they do not use."




*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.*

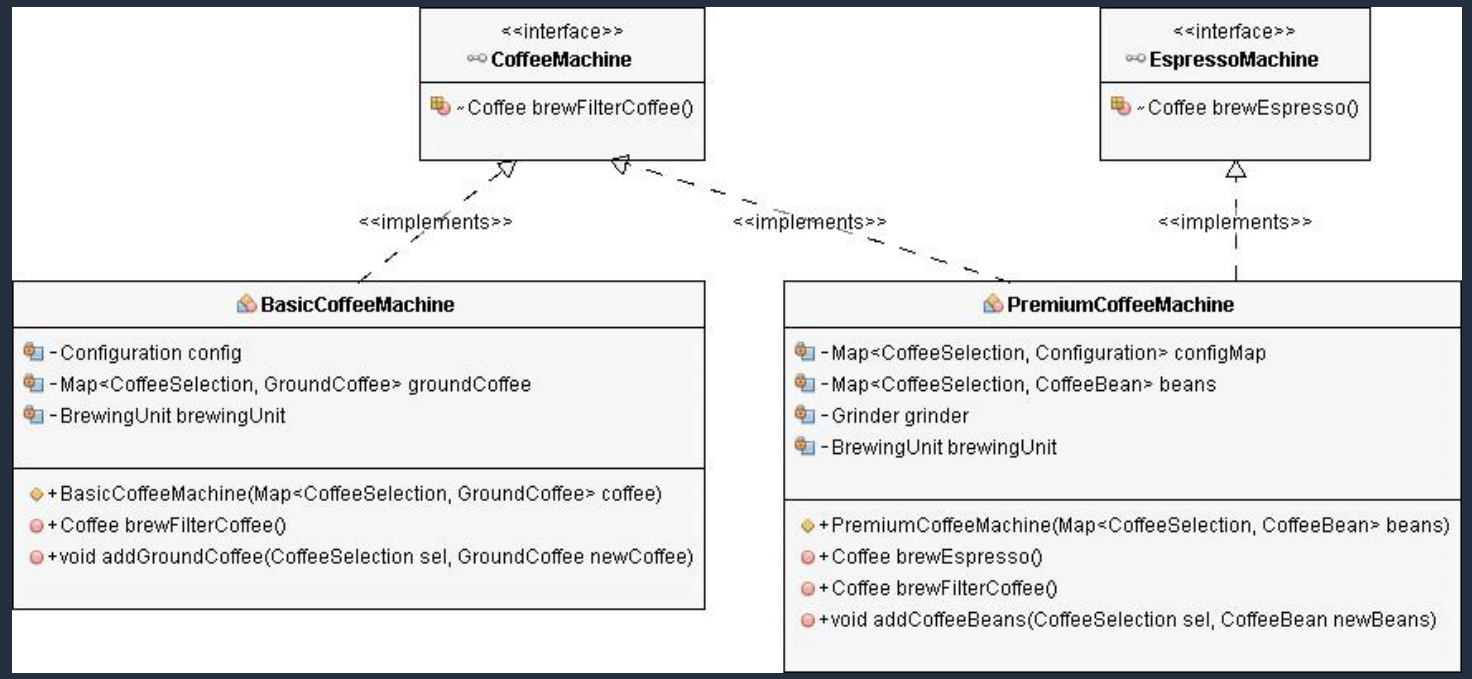
https://stackify.com/interface-segregation-principle/

# DIP — Dependency Inversion Principle

*Dependency Inversion Principle consists of two parts:*

*1. **High-level modules** should not depend on **low-level modules**. Both should depend on abstractions.*

*2. Abstractions should not depend on details. Details should depend on abstractions.*




This might sound more complex than it often is. If you consequently apply the ***Open/Closed Principle*** and the **Liskov Substitution Principle** to your code, it will also follow the Dependency Inversion Principle.

https://stackify.com/dependency-inversion-principle/

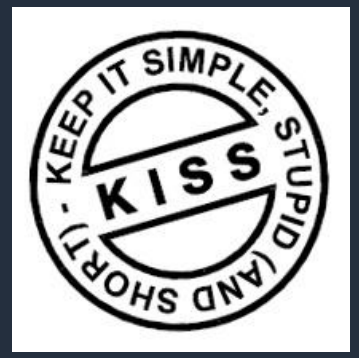
# ***KISS*** - Keep It Simple, but Sensational

The KISS principle states that most systems work best if they are kept simple rather than complicated. Therefore, simplicity must be a fundamental objective in the design and unnecessary complexity must be avoided.

The KISS Principle (Keep It Simple) is self-descriptive and recognizes two things:

1. People – the users of products and services – often want simple things, which are easy to learn and use;

2. The company that manufactures products or provides services also has a lot to bring to the concept of “keep it simple”, as it tends to reduce time and reduce internal cost.




https://harve.com.br/blog/marketing-digital-blog/keep-it-simple-but-sensational-o-principio-kiss-para-impactar/

# The "Don't Repeat Yourself" Principle (***DRY***)

The DRY principle, an acronym for Don’t Repeat Yourself,
created by Andy Hunt, proposes that every feature in a project should be represented
only once.

Who, at some point in their life, has never witnessed others or themselves changing various classes,
pages, or files, while maintaining some system, just to fix a single detail?

This is the chasm that repetition, without proper refactoring or planning, gives us.
Without attention and care for the code, entropy kicks in and chaos sets in easily and quickly,
often silently, until it trumpets at a later time during a bug, rollout of new features, or any other maintenance.

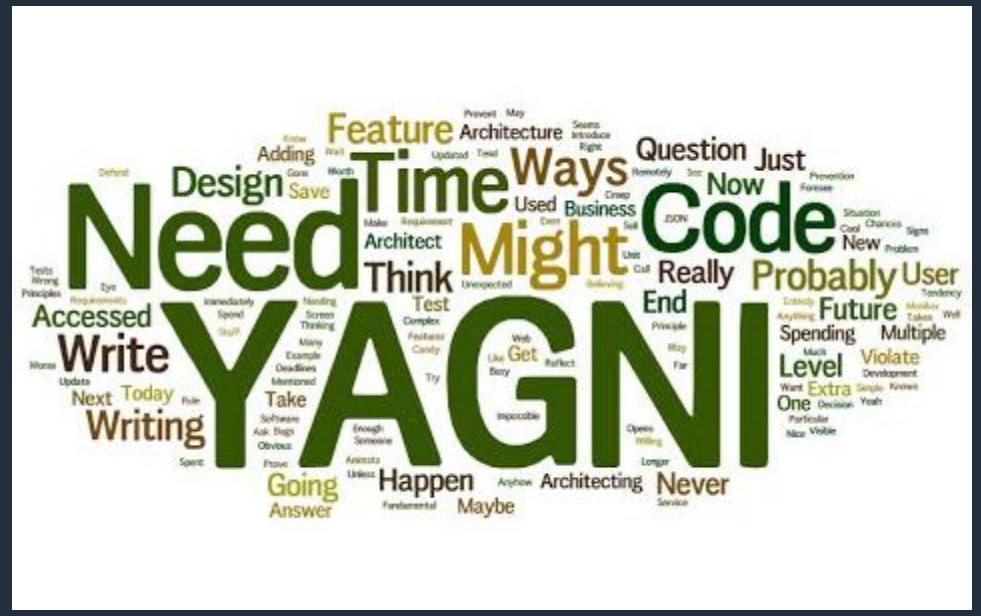

https://www.profissionaisti.com.br/o-principio-do-nao-se-repita-dry/

# ***YAGNI*** - You Aren't Gonna Need It

This is a principle of the agile methodology ***XP (Extreme Programming)*** that suggests that you should not add things that you will not use in your code.

On a day-to-day basis, I see this very related to Over Engineering, that is, creating a much more complex solution than necessary.

Thinking about a future situation that may never exist.




https://dev.to/thiagomr/desvendando-o-mundo-magico-dos-acronimos-solid-kiss-yagni-dry-ddd-tdd-2onp

# Why use Code Conventions

***80% of the time programmers*** should be focused on business rules, understanding, studies and another ***20% programming***, and there would still be time to improve the code;

Hardly a code will be kept "***forever***" by its original creator;

Well-written, well-descriptive, well-annotated code increases productivity, decreases the amount of training, facilitates reading and is visually pleasing;

If you sell your code as a product, you must make sure it is a well-tested product, packaged to be delivered and that does what it promises, delivering value to its customer;

Conventions standardize usability methods with good practices based on knowledge and experience of a body specialized in the area;

https://www.sonarlint.org/

# Pillars for creating code, insight into design standards and quality

# Pilares para Codificação

1. A prioridade máxima é funcionar de acordo com o caso de uso. Beleza e formosura são somente detalhes.

2. Execute o seu código o mais rápido possível. Estou falando de execução real, não de testes automatizados.

3. Protegemos as bordas do sistema como se não houvesse amanhã.

4. Quanto mais externa a borda, mais proteção temos.

5. Não retornamos nulo dentro das regras da aplicação. Pense que seu computador vai explodir.

6. Separamos as bordas externas do sistema do seu núcleo.  Não ligamos parâmetros de requisição externa com objetos de domínio diretamente, assim como não serializamos objetos de domínio para respostas de API.

https://github.com/asouza/pilares-design-codigo

# Pillars for Coding

1. Top priority is to work according to the use case. Beauty and beauty are just details.

2. Run your code as quickly as possible. I'm talking about actual execution, not automated tests.

3. We protect the edges of the system like there's no tomorrow.

4. The further the edge, the more protection we have.

5. We do not return null within the application rules. Think your computer is going to explode.

6. We separate the outer edges of the system from its core. We don't bind external request parameters with domain objects directly, just as we don't serialize domain objects to API responses.

https://github.com/asouza/pilares-design-codigo

# Pilares para Codificação

7. Avaliamos de forma lógica o nível de complicação de cada trecho de código.
Aqui não tem espaço para feeling.
Usamos a teoria da carga cognitiva e medimos o nível de pontos de entendimentos necessário por trecho.

8. Toda indireção aumenta a dificuldade de entendimento da aplicação como um todo,
ela precisa merecer existir. ou seja, precisa ajudar a distribuir a carga intrínseca pelo sistema.

9. Usamos o construtor para criar o objeto no estado válido.

https://github.com/asouza/pilares-design-codigo

# Pillars for Coding

7. We logically assess the level of complication of each piece of code.
There's no space for feeling.
We use cognitive load theory and measure the level of understanding points needed per stretch.

8. Any indirection increases the difficulty of understanding the application as a whole, it must deserve to exist.
that is, it needs to help distribute the intrinsic load across the system.

9. We use the constructor to create the object in the valid state.

https://github.com/asouza/pilares-design-codigo

# Pilares para Codificação

10. A complicação do nosso é código é proporcional a complicação da nossa feature.
    Quanto mais simples, melhor.

11. Usamos tudo que conhecemos que está pronto.
    Só fazemos código do zero se for estritamente necessário.

12. Idealmente, todo código escrito deveria ser chamado por alguém.
    Se não tem ninguém chamando, ele não deveria existir.

13. Só alteramos estado de referências que criamos. Não mexemos nos objetos alheios.
A não ser que esse objeto seja criado para isso, como é o caso de argumentos de métodos de borda mais externa.
Estes são, geralmente, associados a frameworks.

https://github.com/asouza/pilares-design-codigo

# Pillars for Coding

10. The complication of our code is proportional to the complication of our feature.
    The simpler the better.

11. We use everything we know that is ready.
    We only code from scratch if strictly necessary.

12. Ideally, all code written should be called by someone.
    If there's no one calling, he won't should exist.

13. We only change the status of references we create. We don't touch other people's objects.
    Unless this object be created for this, as is the case for outermost edge method arguments.
    These are usually associated with frameworks.

https://github.com/asouza/pilares-design-codigo

# Pilares para Codificação

14. A versão mais eficiente de uma pessoa programando é aquela que entende, questiona e implementa estritamente o que foi combinado.
    Não inventamos coisas que não foram pedidas,
    Não fazemos suposição de funcionalidade e nem caímos na armadilha de achar que entendemos mais do que a pessoa que solicitou a funcionalidade.

https://github.com/asouza/pilares-design-codigo

# Pillars for Coding

14. The most efficient version of a programming person is one that understands, questions, and implements strictly what was agreed.
    We don't invent things that weren't asked for,
    We don't make assumptions about functionality and we don't fall for the trap of thinking we understand more than the person who requested the functionality.

https://github.com/asouza/pilares-design-codigo

# Pilares para Codificação

15. Você precisa entender o que está usando e olhar sempre o lado negativo de cada decisão.

16. Deixamos pistas que facilitem o uso do código onde não conseguimos resolver com compilação.

17. A sua api deve deixar claro o caminho que deve ser seguido pelo ponto do código que decide usá-la. Não espere que ninguém lembre de invocar nada. Faça de tudo para gerar obrigações. Quanto mais específico é seu código, menos democrático ele é.

https://github.com/asouza/pilares-design-codigo

# Pillars for Coding

15. You need to understand what you are using and always look at the negative side of every decision.

16. We leave clues that facilitate the use of the code where we can't solve with compilation.

17. Your api should make clear the path that should be followed by the code point that decides to use it. Don't expect anyone to remember to summon anything. Do everything to generate obligations. the more specific is its code, the less democratic it is.

https://github.com/asouza/pilares-design-codigo

# Pilares para Codificação

18. Não usamos exception para controle de fluxo.

19. Regras de negócio devem ser declaradas de maneira explícita na nossa aplicação.

20. Favorecemos a coesão através do encapsulamento.

21. Criamos testes automatizados para que ele nos ajude a revelar e consertar bugs na aplicação.

https://github.com/asouza/pilares-design-codigo

# Pillars for Coding

18. We don't use exception for flow control.

19. Business rules must be stated explicitly in our application.

20. We favor cohesion through encapsulation.

21. We create automated tests so that it helps us to reveal and fix bugs in the application.

https://github.com/asouza/pilares-design-codigo

# Cyclomatic Complexity

```java
private BigDecimal priceCoffee(boolean premium, boolean express, int special){
    BigDecimal preco = BigDecimal.ZERO;
    if(premium){
        preco = BigDecimal.valueOf(20);
        if(express){
            preco = preco.add(BigDecimal.valueOf(2));
            if(special > 1){
                preco = preco.add(BigDecimal.valueOf(2));
            } else {
                preco = preco.add(BigDecimal.valueOf(1));
            }
        } else {
            preco = preco.add(BigDecimal.valueOf(1));
        }
    } else {
        preco = preco.add(BigDecimal.valueOf(15))
                     .add(BigDecimal.valueOf(1))
                     .add(BigDecimal.valueOf(1));
    }
    return preco;
}
```

```
Run: CoffeeCyclomaticComplexity
/opt/jdk-19/bin/java ...
Price Coffee : 24.0

Process finished with exit code 0
```

https://artesoftware.com.br/2019/02/09/complexidade-ciclomatica/

# Cyclomatic Complexity - Refactory

```java
private BigDecimal priceCoffeeRefactory(boolean premium, boolean express, int special){
    BigDecimal preco = BigDecimal.ZERO;
    if(premium){
        preco = BigDecimal.valueOf(20);
        preco = computePriceExpressCoffee(express, special, preco);
    } else {
        preco = computePriceCoffeeComplex(preco);
    }
    return preco;
}
```

```java
private BigDecimal computePriceExpressCoffee(boolean express, int special, BigDecimal preco) {
    if(express){
        preco = computePriceCoffeeBasic(preco, valueAdd: 2L);
        preco = computePriceSpecialCoffee(special, preco);
    } else {
        preco = computePriceCoffeeBasic(preco, valueAdd: 1L);
    }
    return preco;
}
```

```java
private BigDecimal computePriceSpecialCoffee(int special, BigDecimal preco) {
    if(special > 1){
        preco =  computePriceCoffeeBasic(preco, valueAdd: 2);
    } else {
        preco = computePriceCoffeeBasic(preco, valueAdd: 1L);
    }
    return preco;
}
```

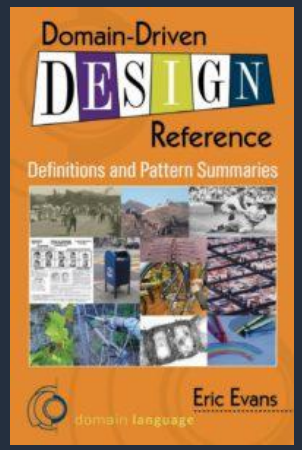
```java
private BigDecimal computePriceCoffeeComplex(BigDecimal preco) {
    return preco.add(BigDecimal.valueOf(15))
            .add(BigDecimal.valueOf(1))
            .add(BigDecimal.valueOf(1));
}
```

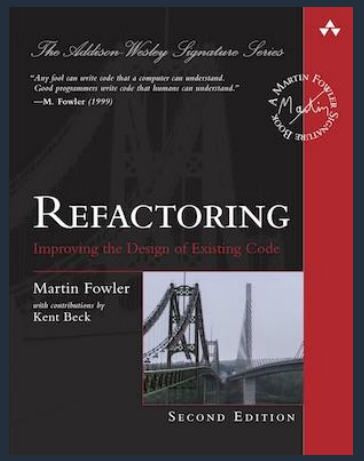
```java
private BigDecimal computePriceCoffeeBasic(BigDecimal preco, double valueAdd){
    return preco.add(BigDecimal.valueOf(valueAdd));
}
```

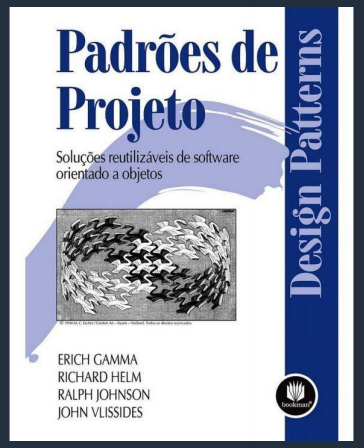
```
Run: CoffeeCyclomaticComplexity
/opt/jdk-19/bin/java ...
Price Coffee : 24.0

Process finished with exit code 0
```

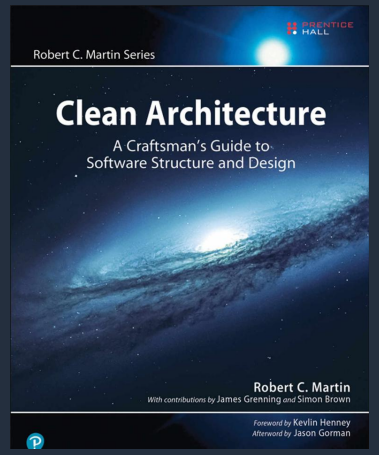
https://artesoftware.com.br/2019/02/09/complexidade-ciclomatica/

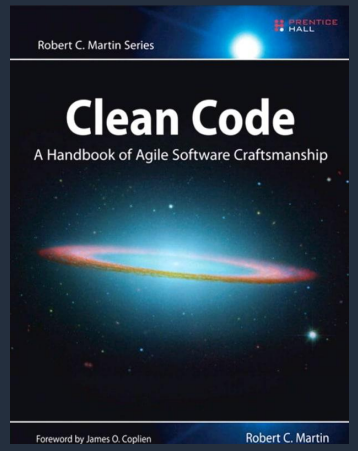
# Books that changed my mind




https://github.com/asouza/pilares-design-codigo

aws certified Developer ASSOCIATE

aws Technical Accredited PARTNER

aws Cloud Economics Accredited PARTNER

aws Business Accredited PARTNER

CertiProf

# Weder Mariano de Sousa

Analist System Senior - GFT Brazil

+ 16 Years of Experience
+ 50 Projects Worked
Post Graduate in Media UFG
Post Graduate in Information Security
Computer scientist
IT Technician ITGO

GOJava

AWS User Group Goiânia

Grupy-GO

# Q & A

https://www.linkedin.com/in/wedermarianodesousa/
https://github.com/weder96

# THANK YOU