awari

NPM, Frameworks e MVC

‘’Adicionar um subtítulo’’

Gerenciando projetos com npm 

Comandos e dicas com npm 💻

Arquitetura MVC 💻

awari

Node.js com Express

Transformando nosso backend em API

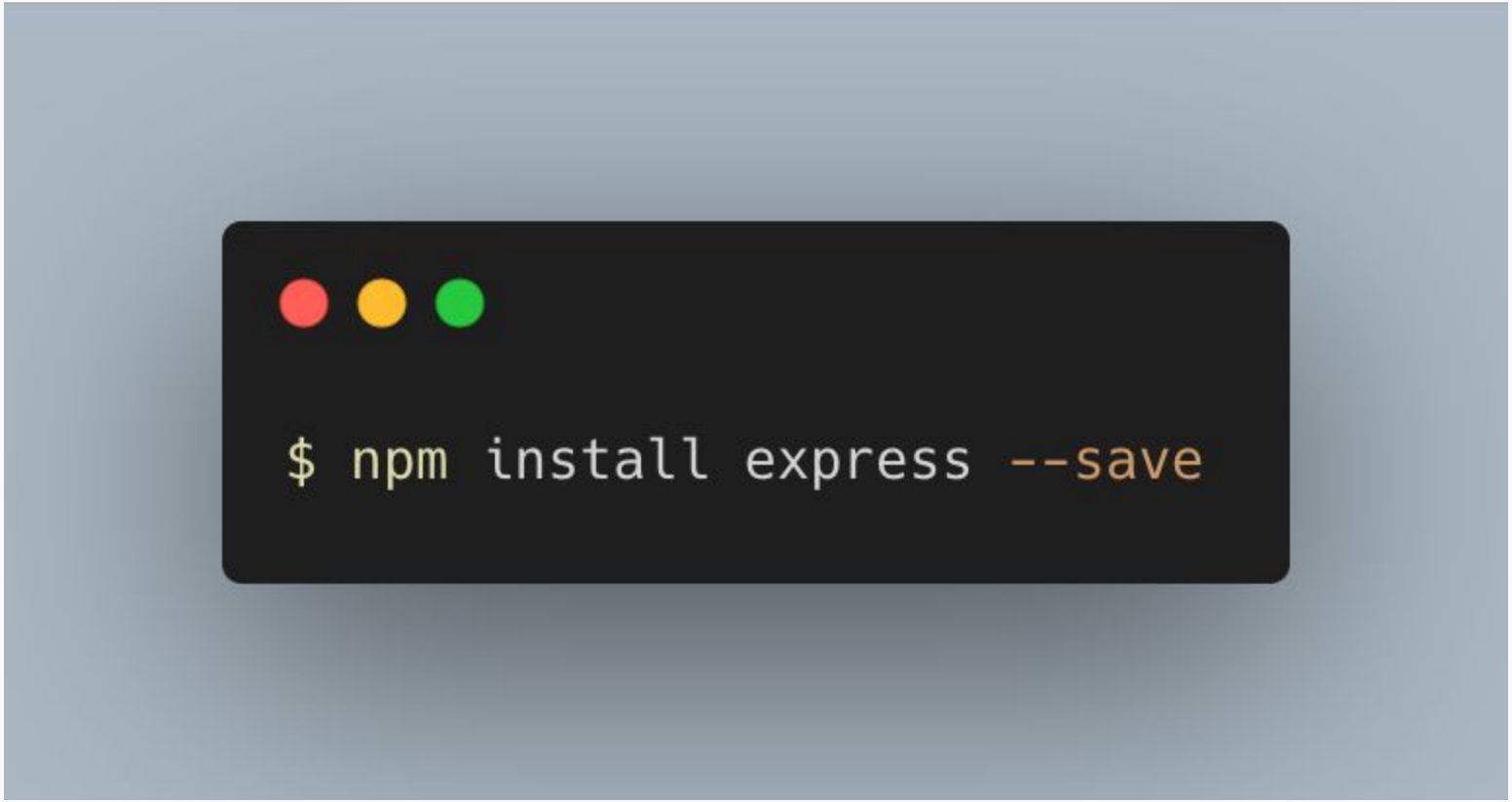
O que é Express? 

O **Express** é um framework para o **Node.js** desenvolvido para facilitar a criação de **API**'s. Ele traz uma série de recursos e abstrações que facilitam a vida na hora de construir seus códigos, como tratar as requisições, definição de regras de negócio, etc.

O **framework** foi construído pensado para o padrão de construção de **API**'s chamado **REST**, foi visto no capítulo anterior. Existem outros frameworks semelhantes no mercado, mas o Express é o mais utilizado pela ampla maioria, e os principais motivos são:

- Ele foi lançado no final de 2010. Ou seja, é um framework maduro e com grande adoção da comunidade
- Ele é um *"Un-Opinionated framework"* (framework não opinativo). Ou seja, ele não impõe um padrão de desenvolvimento na hora de escrever sua aplicação

Como começar com o Express? 💻



Como começar com o Express? 💻

Como começar com o Express? 💻

index.js

```
const express = require('express')
const app = express()
const port = 3000

app.get('/', (req, res) => {
  res.send('Hello World!')
})

app.listen(port, () => {
  console.log(`Aplicação ouvindo na porta ${port}`)
})
```

Como começar com o Express? 💻

```
const express = require('express')
const app = express()

app.get('/', (req, res) => {
  res.send('Hello World!')
})

app.post('/', (req, res) => {
  res.send('Got a POST request')
})

app.put('/user', (req, res) => {
  res.send('Got a PUT request at /user')
})

app.delete('/user', (req, res) => {
  res.send('Got a DELETE request at /user')
})
```

Documentação 💻

http://expressjs.com/pt-br/

awari

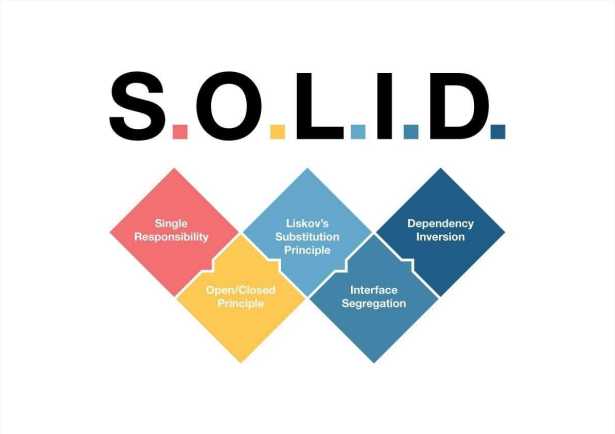
Princípios SOLID, DRY e KISS

Adicionando qualidade no seu código

O que é SOLID? 💻

SOLID é um princípio de design de software orientado a objeto (OOD).
Basicamente, **SOLID** é um acrônimo para:

S - Single Responsibility Principle (Princípio da Responsabilidade Única)
O - Open-closed Principle (Princípio Aberto-fechado)
L - Liskov Substitution Principle (Princípio da Substituição de Liskov)
I - Interface Segregation Principle (Princípio da Segregação de Interface)
D - Dependency Inversion Principle (Princípio da Inversão de Dependência)



Single Responsibility Principle

“Uma classe deve ter uma e apenas uma razão para mudança, significando que uma classe deve ter apenas uma responsabilidade.”

```
class User {
  private db: Database;

  constructor(private name: string, private email: string) {
    this.db = Database.connect("adm:pass@dbname", ["users"]);
  }

  getArrayUserData() {
    return [this.name, this.email];
  }

  save() {
    this.db.users.save({ title: this.name, email: this.email });
  }
}
```

Violando o princípio

Single Responsibility Principle

“Uma classe deve ter uma e apenas uma razão para mudança, significando que uma classe deve ter apenas uma responsabilidade.”

```
class User {

  constructor(private name: string, private email: string) {}

  getArrayUserData() {
    return [this.name, this.email];
  }
}

class UserRepository {
  private db: Database;

  constructor() {
    this.db = Database.connect("adm:pass@dbname", ["users"]);
  }

  save() {
    this.db.users.save({ title: this.name, email: this.email });
  }
}
```

Aplicando o princípio

awari

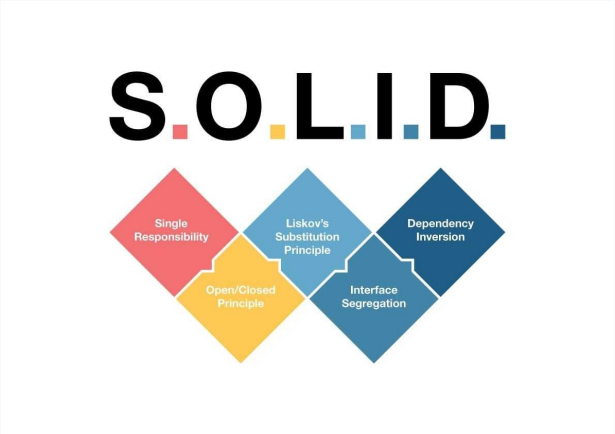
Princípios SOLID, DRY e KISS

Adicionando qualidade no seu código

O que é SOLID? 💻

SOLID é um princípio de design de software orientado a objeto (OOD).
Basicamente, **SOLID** é um acrônimo para:

S - Single Responsibility Principle (Princípio da Responsabilidade Única)
O - Open-closed Principle (Princípio Aberto-fechado)
L - Liskov Substitution Principle (Princípio da Substituição de Liskov)
I - Interface Segregation Principle (Princípio da Segregação de Interface)
D - Dependency Inversion Principle (Princípio da Inversão de Dependência)



Single Responsibility Principle

“Uma classe deve ter uma e apenas uma razão para mudança, significando que uma classe deve ter apenas uma responsabilidade.”

```
class User {
  private db: Database;

  constructor(private name: string, private email: string) {
    this.db = Database.connect("adm:pass@dbname", ["users"]);
  }

  getArrayUserData() {
    return [this.name, this.email];
  }

  save() {
    this.db.users.save({ title: this.name, email: this.email });
  }
}
```

Violando o princípio

Single Responsibility Principle

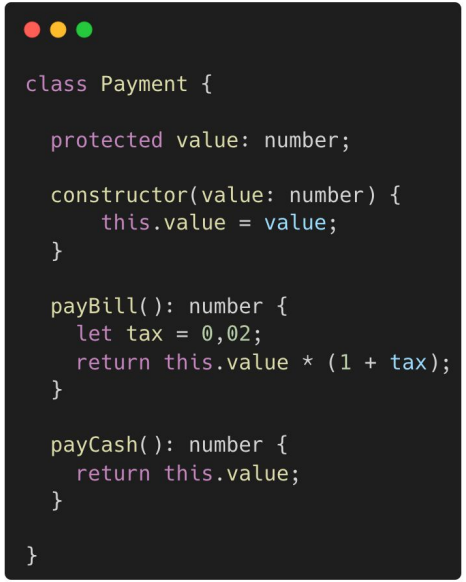
“Uma classe deve ter uma e apenas uma razão para mudança, significando que uma classe deve ter apenas uma responsabilidade.”

```
class User {

  constructor(private name: string, private email: string) {}

  getArrayUserData() {
    return [this.name, this.email];
  }
}

class UserRepository {
  private db: Database;

  constructor() {
    this.db = Database.connect("adm:pass@dbname", ["users"]);
  }

  save() {
    this.db.users.save({ title: this.name, email: this.email });
  }
}
```

Aplicando o princípio

Open-closed Principle 💻

“Objetos ou entidades devem ser abertas para extensão, mas fechadas para modificação.”



```
class Payment {

  protected value: number;

  constructor(value: number) {
      this.value = value;
  }

  payBill(): number {
    let tax = 0,02;
    return this.value * (1 + tax);
  }

  payCash(): number {
    return this.value;
  }

}
```

Violando o princípio

Open-closed Principle 💻

“Objetos ou entidades devem ser abertas para extensão, mas fechadas para modificação.”

```
class Payment {

  protected value: number;

  constructor(value: number) {
      this.value = value;
  }

  pay(): number {
    return this.value;
  }
}

class Bill extends Payment {

  pay(): number {
    let tax = 0,02;
    return this.value * (1 + tax);
  }
}

class Cash extends Payment {

}
```

Aplicando o princípio

Liskov Substitution Principle 💻

“Uma classe derivada deve ser substituída por sua classe base.”

```
class UserRepository {
    // seu código
}

class EmployeeRepository extends UserRepository {
    // seu código
}

class ContractorRepository extends UserRepository {
    // seu código
}
```

Aplicando o princípio

Liskov Substitution Principle 💻

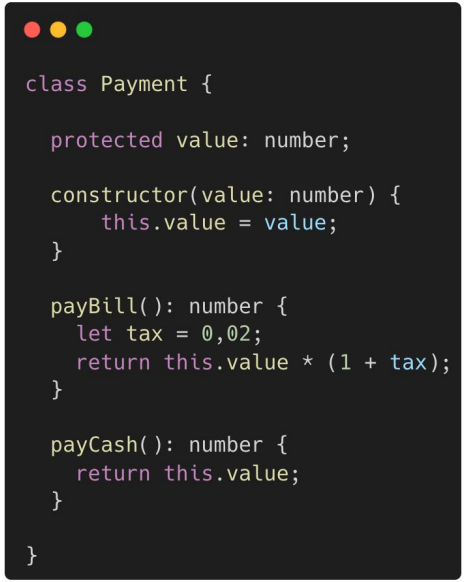
“Uma classe derivada deve ser substituída por sua classe base.”

```
class UserService {
    private employee: employeeRepository;
    private contractor: contractorRepository;

    checkUserType(type: String, data: Array) {
        if(type === 'employee') {
            this.register(data, this.employee);
        }

        if(type === 'contractor') {
            this.register(data, this.contractor);
        }
    }

    register(data: Array, user: userRepository)
    {
        user.save(data);
    }
}
```

Aplicando o princípio

Open-closed Principle 💻

“Objetos ou entidades devem ser abertas para extensão, mas fechadas para modificação.”



```
class Payment {

  protected value: number;

  constructor(value: number) {
      this.value = value;
  }

  payBill(): number {
    let tax = 0,02;
    return this.value * (1 + tax);
  }

  payCash(): number {
    return this.value;
  }

}
```

Violando o princípio

Open-closed Principle 💻

“Objetos ou entidades devem ser abertas para extensão, mas fechadas para modificação.”

```
class Payment {

  protected value: number;

  constructor(value: number) {
      this.value = value;
  }

  pay(): number {
    return this.value;
  }
}

class Bill extends Payment {

  pay(): number {
    let tax = 0,02;
    return this.value * (1 + tax);
  }
}

class Cash extends Payment {

}
```

Aplicando o princípio

Liskov Substitution Principle 💻

“Uma classe derivada deve ser substituída por sua classe base.”

```
class UserRepository {
    // seu código
}

class EmployeeRepository extends UserRepository {
    // seu código
}

class ContractorRepository extends UserRepository {
    // seu código
}
```

Aplicando o princípio

Liskov Substitution Principle 💻

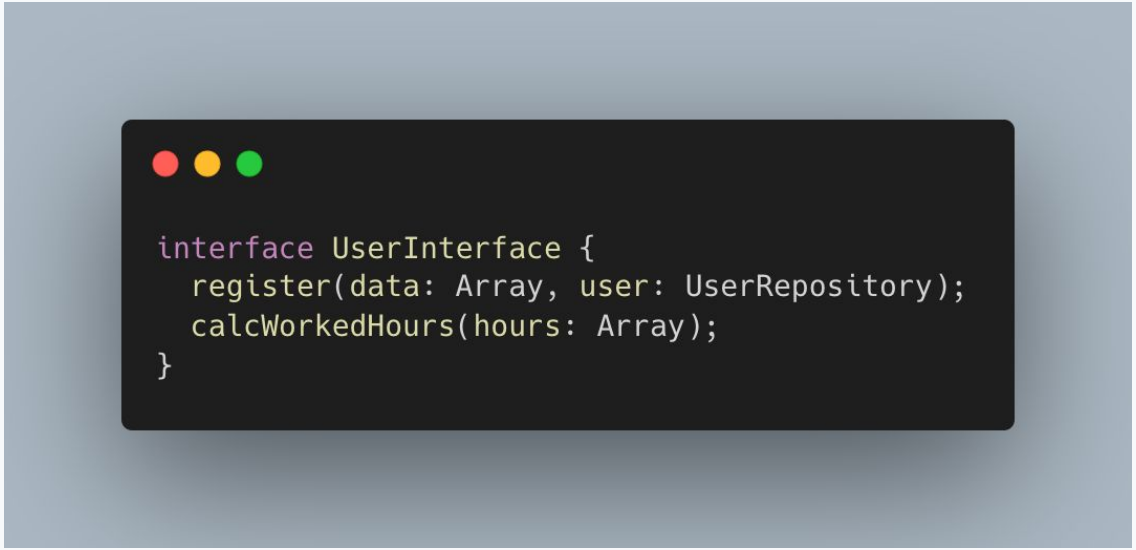
“Uma classe derivada deve ser substituída por sua classe base.”

```
class UserService {
    private employee: employeeRepository;
    private contractor: contractorRepository;

    checkUserType(type: String, data: Array) {
        if(type === 'employee') {
            this.register(data, this.employee);
        }

        if(type === 'contractor') {
            this.register(data, this.contractor);
        }
    }

    register(data: Array, user: userRepository)
    {
        user.save(data);
    }
}
```

Aplicando o princípio

Interface Segregation Principle 💻

“Uma classe não deve ser forçada a implementar interfaces que não irá utilizar.”

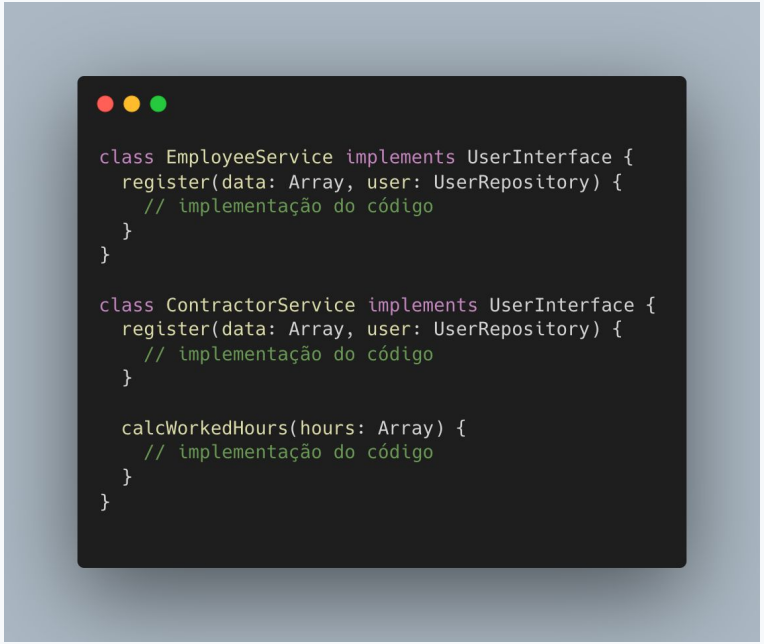


```
interface UserInterface {
  register(data: Array, user: UserRepository);
  calcWorkedHours(hours: Array);
}
```

Violando o princípio

Interface Segregation Principle 💻

“Uma classe não deve ser forçada a implementar interfaces que não irá utilizar.”

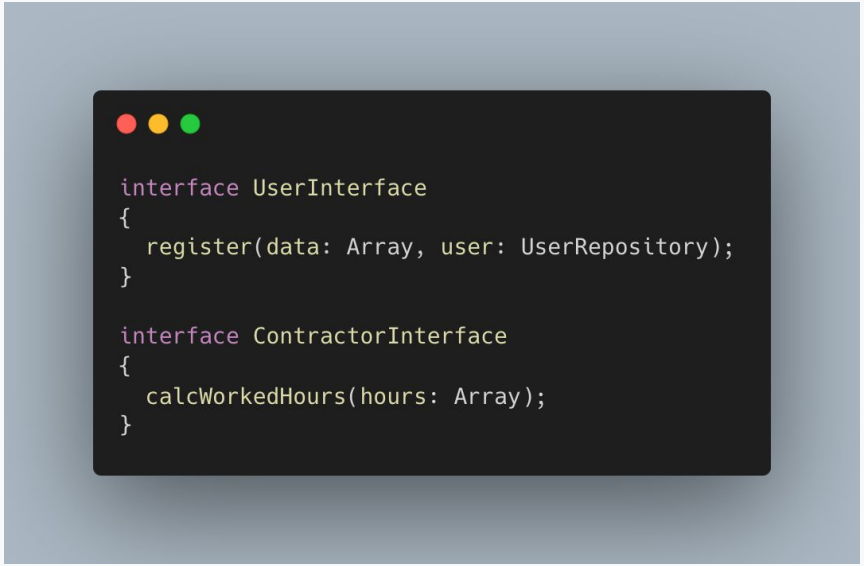


```
class EmployeeService implements UserInterface {
  register(data: Array, user: UserRepository) {
    // implementação do código
  }
}

class ContractorService implements UserInterface {
  register(data: Array, user: UserRepository) {
    // implementação do código
  }

  calcWorkedHours(hours: Array) {
    // implementação do código
  }
}
```

Violando o princípio

Interface Segregation Principle 💻

“Uma classe não deve ser forçada a implementar interfaces que não irá utilizar.”



```
interface UserInterface
{
  register(data: Array, user: UserRepository);
}

interface ContractorInterface
{
  calcWorkedHours(hours: Array);
}
```

Aplicando o princípio

Interface Segregation Principle 💻

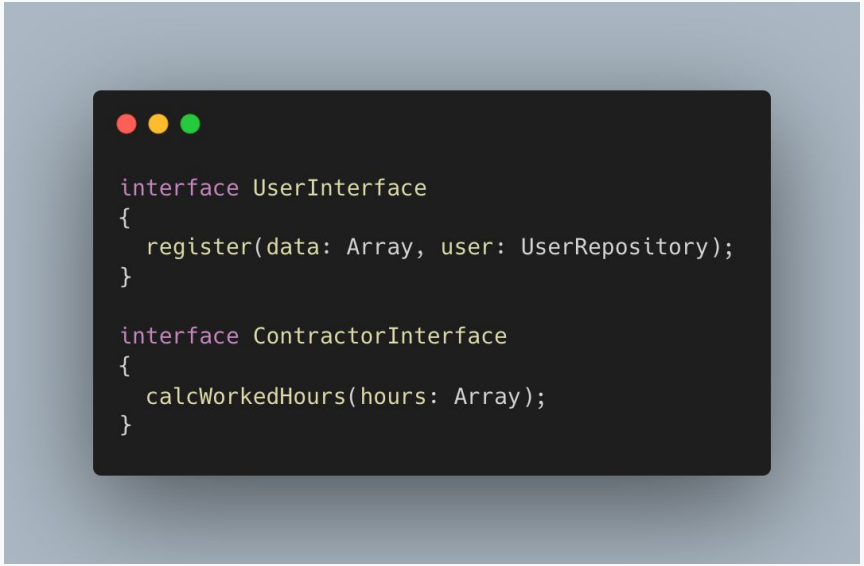
“Uma classe não deve ser forçada a implementar interfaces que não irá utilizar.”

```
class EmployeeService implements UserInterface {
  register(data: Array, user: UserRepository) {
    // implementação do código
  }
}

class ContractorService implements UserInterface, ContractorInterface
{
  register(data: Array, user: UserRepository) {
    // implementação do código
  }

  calcWorkedHours(hours: Array) {
    // implementação do código
  }
}
```

Aplicando o princípio

Dependency Inversion Principle 💻

“Entidades devem depender de abstrações e não de algo concreto. Isso significa que módulos de alto nível não podem depender de módulos de baixo nível, e sim de abstrações.”



```
interface UserInterface
{
  register(data: Array, user: UserRepository);
}

interface ContractorInterface
{
  calcWorkedHours(hours: Array);
}
```

Violando o princípio

Dependency Inversion Principle 💻

“Entidades devem depender de abstrações e não de algo concreto. Isso significa que módulos de alto nível não podem depender de módulos de baixo nível, e sim de abstrações.”

```
class UserRegistrationController {
    private userService;
    private contractorService;

    constructor(userService: UserInterface, contractorService: ContractorInterface) {
        this.userService = userService;
        this.contractorService = contractorService;
    }

    // implementação do código
}
```

Violando o princípio

Dependency Inversion Principle 💻

“Entidades devem depender de abstrações e não de algo concreto. Isso significa que módulos de alto nível não podem depender de módulos de baixo nível, e sim de abstrações.”

```
interface ServiceInterface
{
}

interface UserInterface extends ServiceInterface {
  register(data: Array, user: UserRepository);
}

interface ContractorInterface extends ServiceInterface {
  calcWorkedHours(hours: Array);
}
```

Aplicando o princípio

Dependency Inversion Principle 💻

“Entidades devem depender de abstrações e não de algo concreto. Isso significa que módulos de alto nível não podem depender de módulos de baixo nível, e sim de abstrações.”

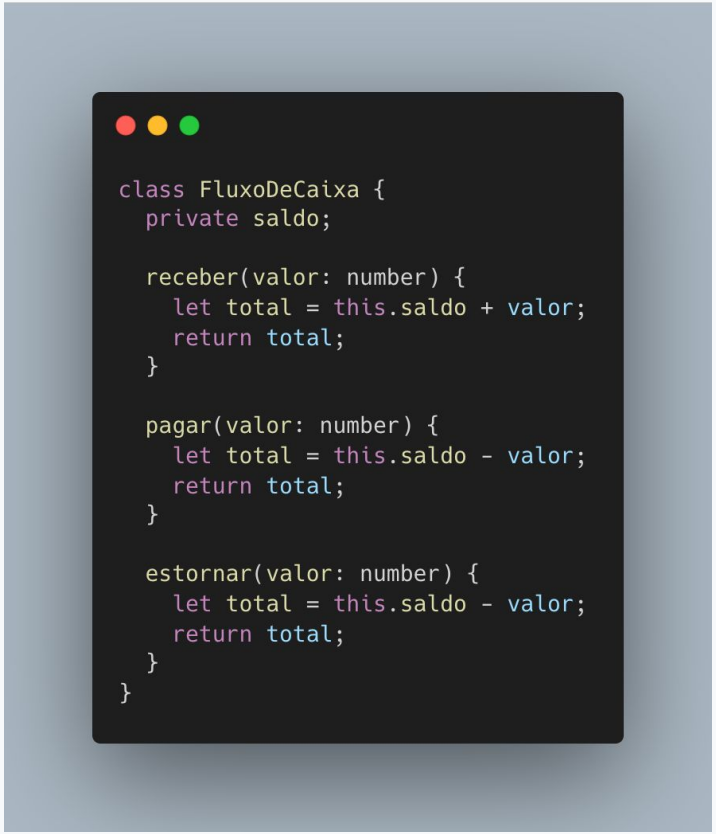
Aplicando o princípio

O que é DRY? 💻

DRY é o acrônimo para *Don't Repeat Yourself* (Não se repita). Princípio criado por Andy Hunt, que propõe que cada funcionalidade em um projeto deve ser representada apenas 1 vez.

Pode ser aplicado no desenvolvimento de aplicação, modelo de dados, rotina de testes, etc.

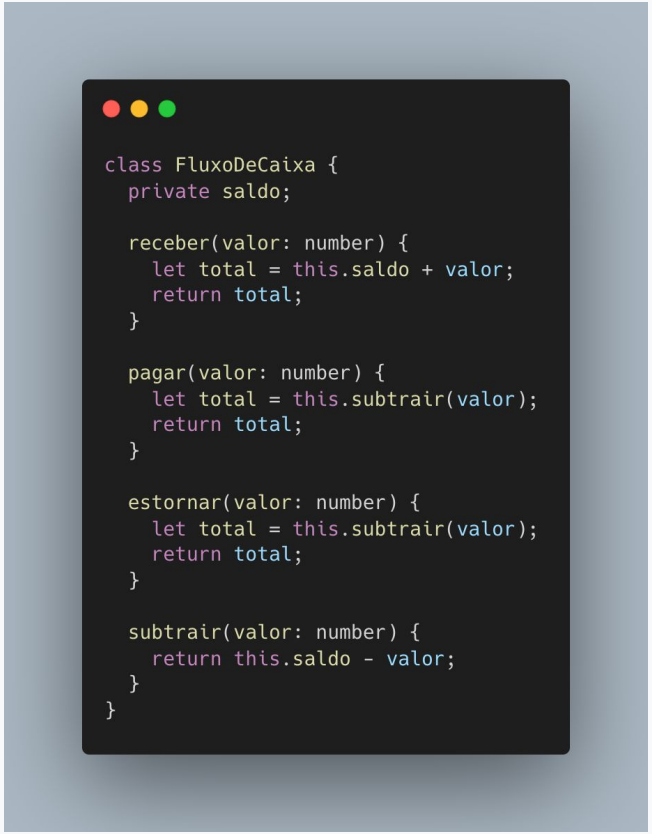
Don't Repeat Yourself



```
class FluxoDeCaixa {
  private saldo;

  receber(valor: number) {
    let total = this.saldo + valor;
    return total;
  }

  pagar(valor: number) {
    let total = this.saldo - valor;
    return total;
  }

  estornar(valor: number) {
    let total = this.saldo - valor;
    return total;
  }
}
```

Violando o princípio

Don't Repeat Yourself



```
class FluxoDeCaixa {
  private saldo;

  receber(valor: number) {
    let total = this.saldo + valor;
    return total;
  }

  pagar(valor: number) {
    let total = this.subtrair(valor);
    return total;
  }

  estornar(valor: number) {
    let total = this.subtrair(valor);
    return total;
  }

  subtrair(valor: number) {
    return this.saldo - valor;
  }
}
```

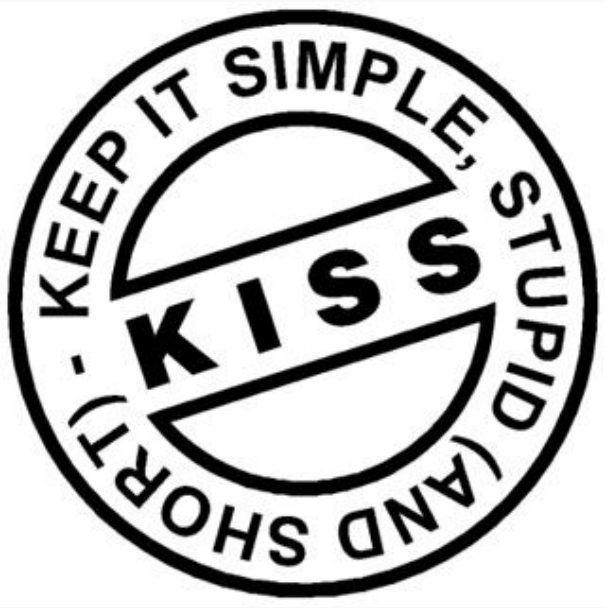
Aplicando o princípio

O que é KISS? 💻

KISS é o acrônimo para *Keep It Simple, Stupid* (Mantenha Isso Simples, Estúpido). **KISS** é um princípio que afirma que projetos e/ou sistemas devem ser tão simples o quanto conseguir. Sempre que possível, a complexidade deve ser evitada, pois a simplicidade garante maiores níveis de aceitação. O **KISS** é usado em uma variedade de disciplinas, como design de interface, design de produto e desenvolvimento de software.

O princípio **KISS** também existe em outras variações com o mesmo significado, como por exemplo: *Keep it short and simple* e *Keep it simple and straightforward*.

- Don't Repeat Your
- MVP: Minimum Viable Product
- Minimalismo
- Menos é mais
- Pragmatismo



awari

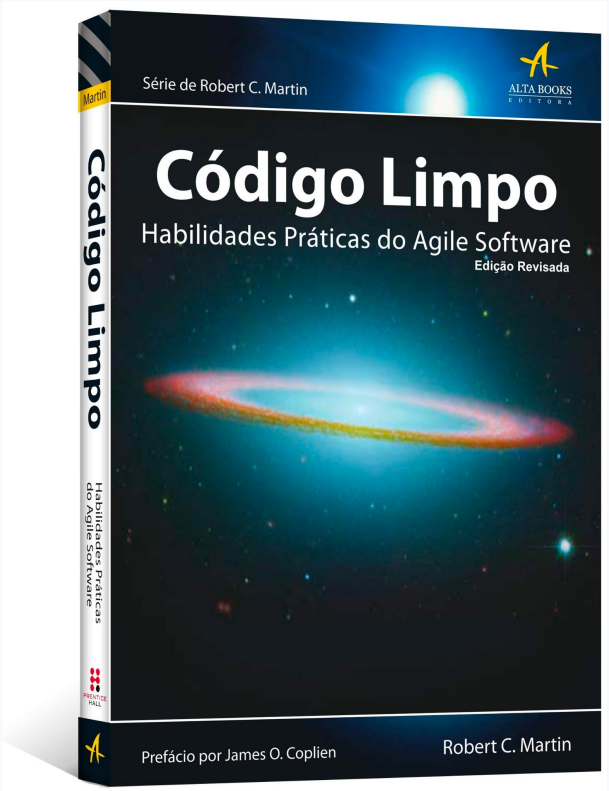
Clean Code

Deixando seu código limpo

O que é Clean Code? 💻

Clean Code é um termo criado por Robert C. Martin (conhecido como Uncle Bob), que dá nome a um livro escrito por ele, que é considerado um best-seller na área de computação.

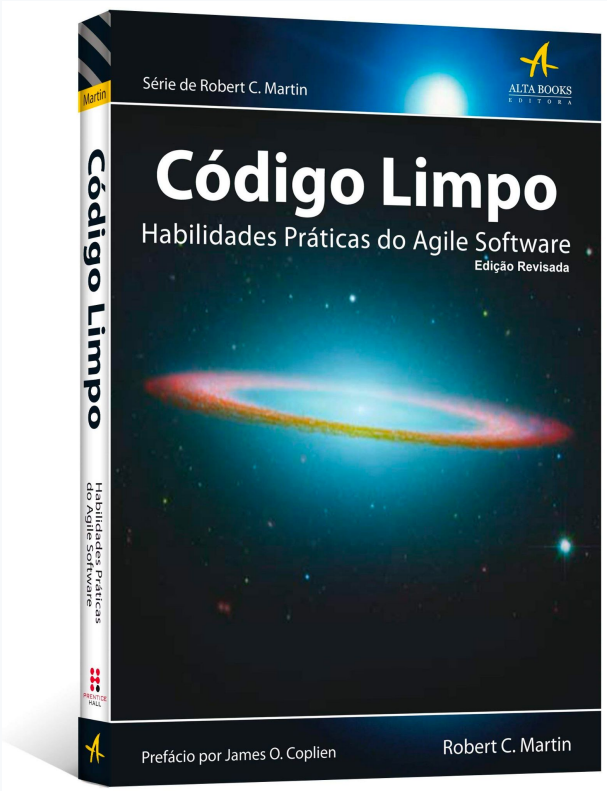
Clean Code é uma série de práticas e princípios que nos ajudam a deixar nosso código mais legível e de fácil entendimento. Ou seja, deixa o nosso código limpo.



Clean Code

Regras Gerais:

- Siga as convenções
- KISS
- Regra do Escoteiro (Deixe sempre o acampamento mais limpo do que quando você o encontrou)
- Causa Raíz



Dependency Inversion Principle 💻

“Entidades devem depender de abstrações e não de algo concreto. Isso significa que módulos de alto nível não podem depender de módulos de baixo nível, e sim de abstrações.”

```
interface ServiceInterface
{
}

interface UserInterface extends ServiceInterface {
  register(data: Array, user: UserRepository);
}

interface ContractorInterface extends ServiceInterface {
  calcWorkedHours(hours: Array);
}
```

Aplicando o princípio

awari

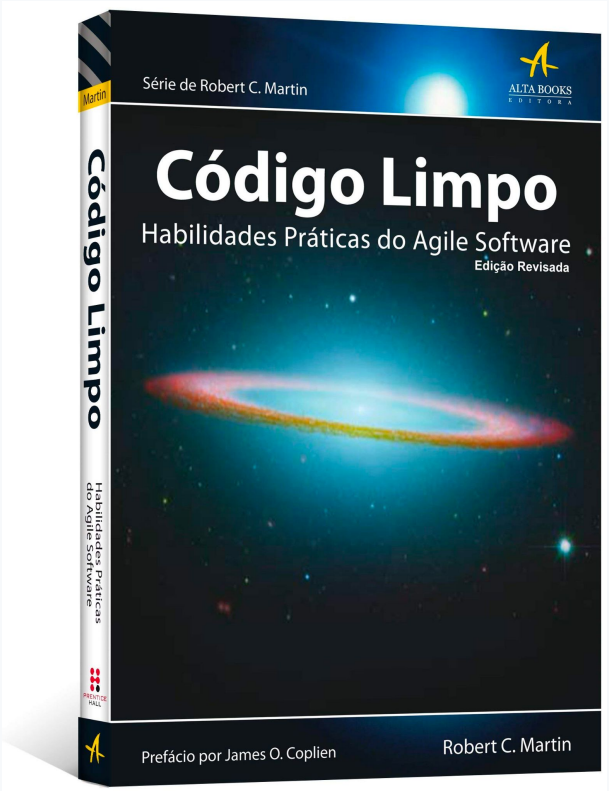
Clean Code

Deixando seu código limpo

O que é Clean Code? 💻

Clean Code é um termo criado por Robert C. Martin (conhecido como Uncle Bob), que dá nome a um livro escrito por ele, que é considerado um best-seller na área de computação.

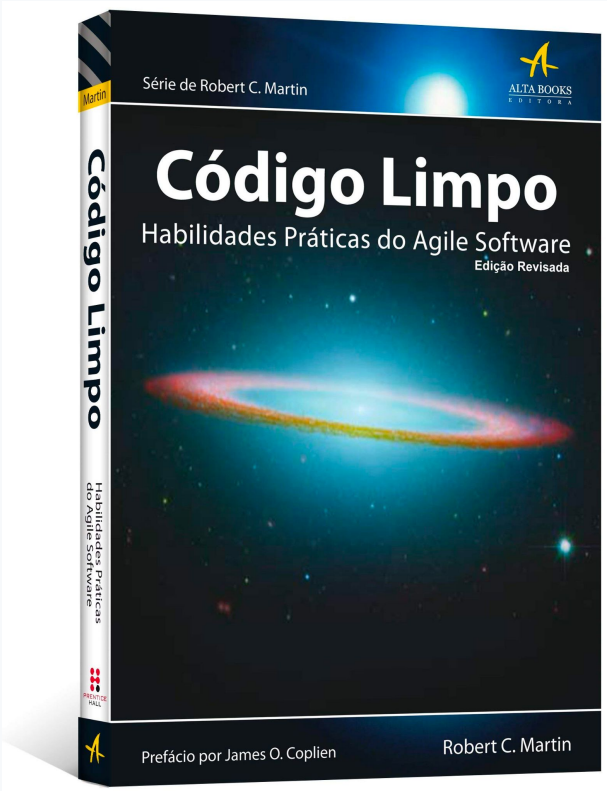
Clean Code é uma série de práticas e princípios que nos ajudam a deixar nosso código mais legível e de fácil entendimento. Ou seja, deixa o nosso código limpo.



Clean Code

Regras Gerais:

- Siga as convenções
- KISS
- Regra do Escoteiro (Deixe sempre o acampamento mais limpo do que quando você o encontrou)
- Causa Raíz



Clean Code

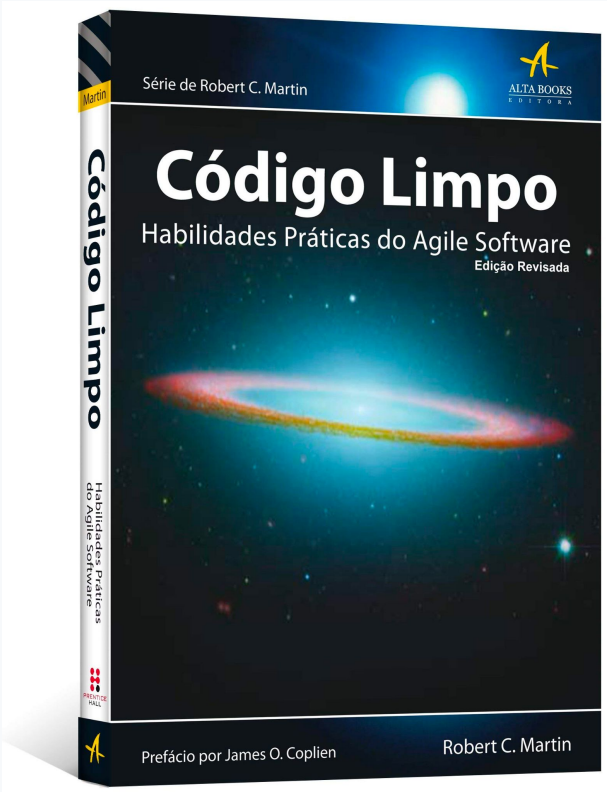
Regras sobre o entendimento do código:

- Seja consistente

```
// Codificando em inglês
class UserRepository { ... }

// Agora mudou para "portuglês"
class UsuarioRepository { ... }

// Agora é português
class RepositorioUsuario { ... }
```

- Utilize variáveis auto-explicativas

```
// Total do que?
var total = 0;

// Total do pedido
var orderTotal = 0;
```



Clean Code

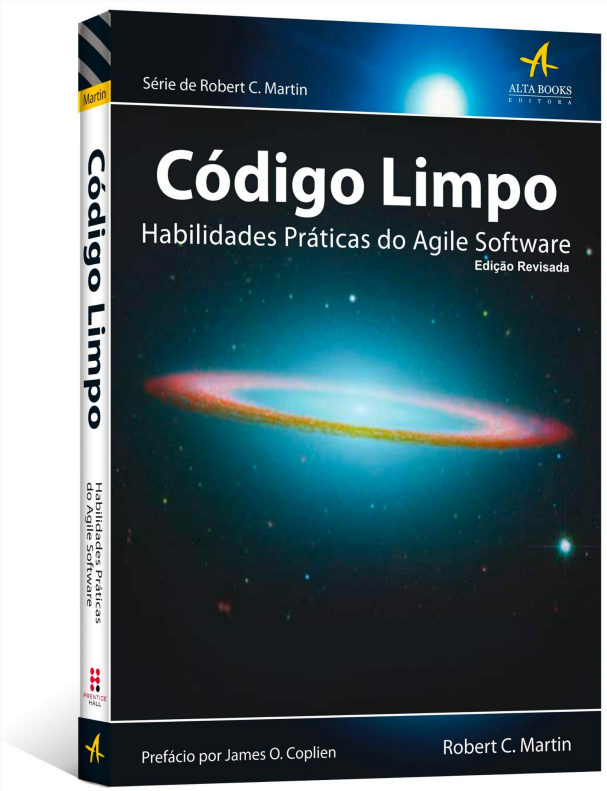
Regras de nomes:

- Evite o uso excessivo de strings

```
// Evite
if(environment == "PROD") {
  ...
}

// Utilize
const string ENV = "PROD";

if(environment == ENV) {
  ...
}
```

- Não use prefixo ou caracteres especiais

```
// Evite
var strNome = "João";

// Evite
var ação = "Enviar";
```



Clean Code

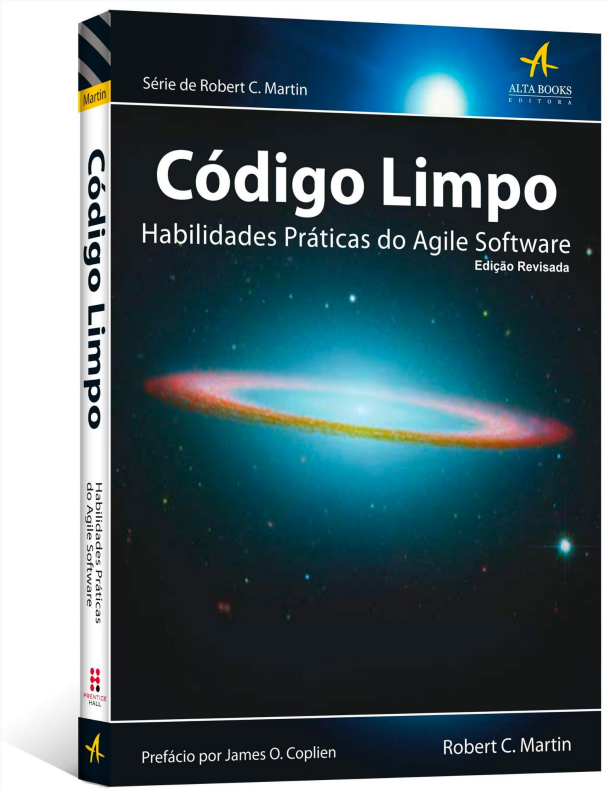
Regras para funções ou métodos:

- Pequenas e com apenas um objetivo

```
// Evite
makeOrder() {
    // Cadastra o cliente
    // Aplica o desconto
    // Atualiza o estoque
    // Salva o pedido
}

// Utilize
saveCustomer() { ... }
applyDiscount() { ... }
updateInventory() { ... }
placeOrder() { ... }
```

- Opte por poucos parâmetros

```
// Evite
saveCustomer(street: String, numberStreet: String, neighborhood: String, zipCode: String) { ... }

// Utilize
saveCustomer(address: Address) { ... }
```



Clean Code

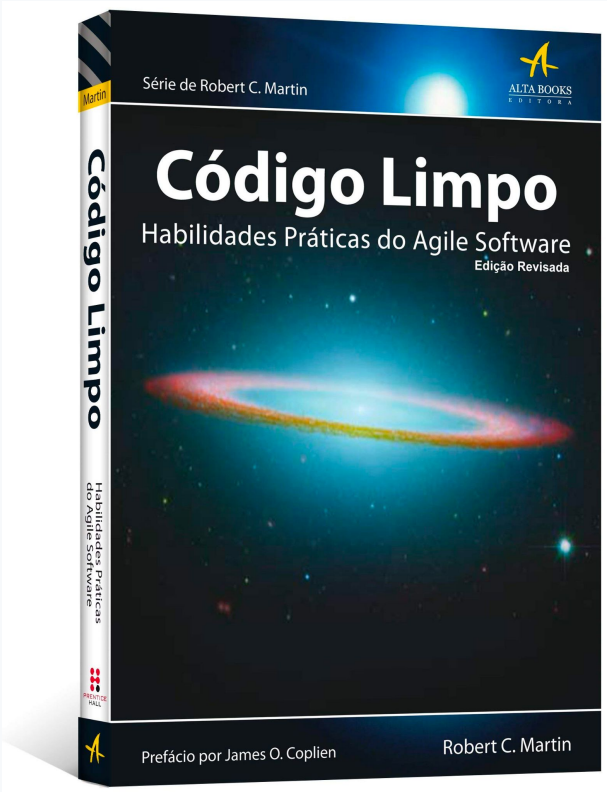
Regras para comentários:

- Não seja redundante

```
// Evite

// Função para adicionar usuário
addUser() { ... }
```

- Evite código comentado

```
// Evite
updateHours()
{
    // var hour = [];
    // calcHours() {... }
}
```



awari

Clean Architecture

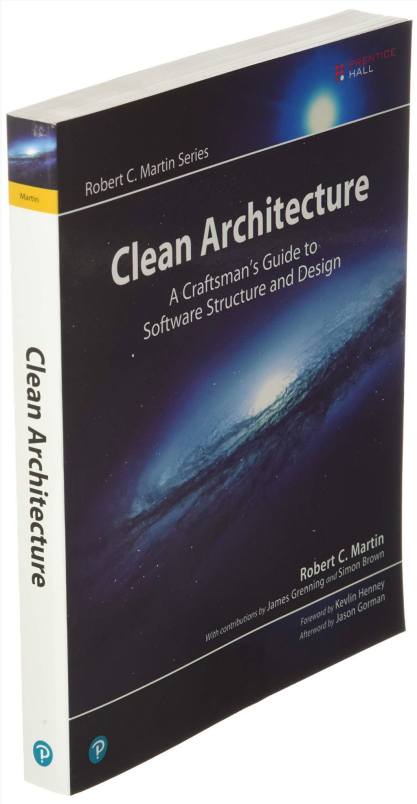
Estruturas reutilizáveis

O que é Clean Architecture? 💻

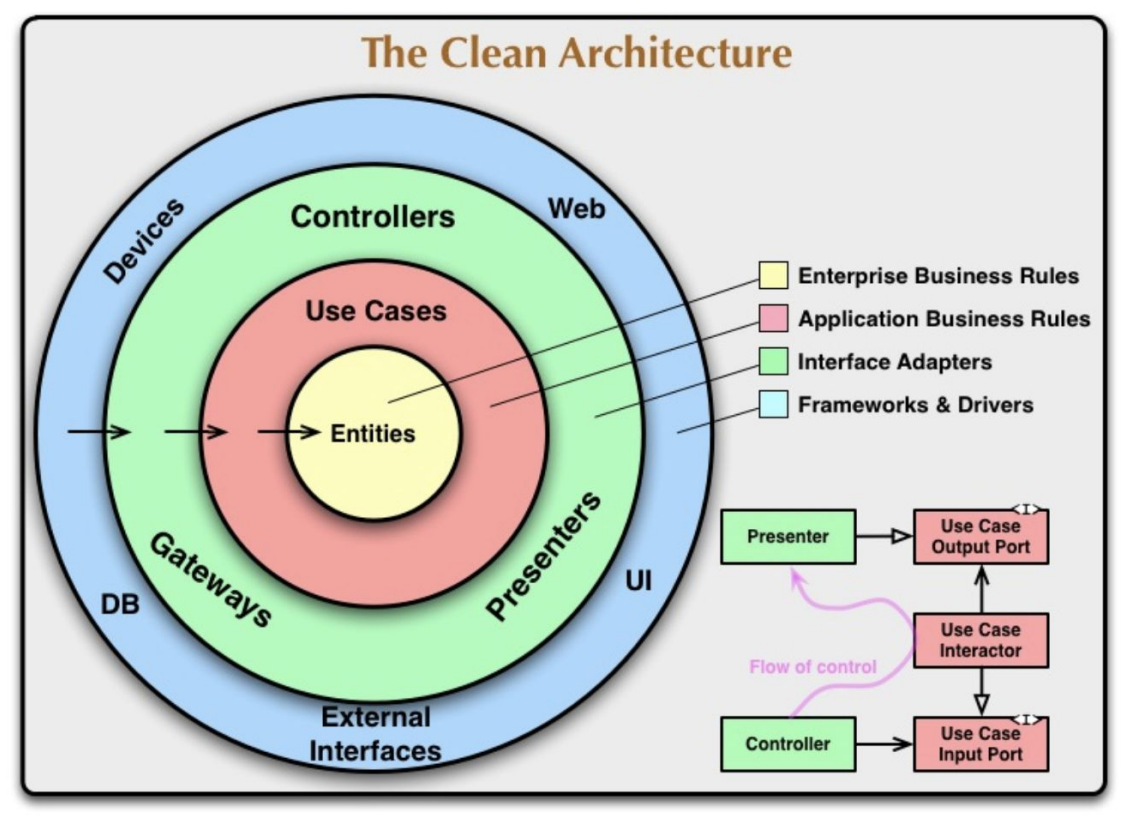
Clean Architecture também é um termo criado por Robert C. Martin (conhecido como Uncle Bob), que dá nome a um livro escrito por ele, que também é considerado um best-seller na área de computação.

Clean Architecture fornece uma metodologia a ser usada na arquitetura da aplicação, com o intuito de facilitar o desenvolvimento de códigos, permitindo uma melhor manutenção, e com menos dependências.

Um objetivo importante da **Clean Architecture** é separar o código em camadas, de uma forma que encapsule a regra de negócios, mas ao mesmo tempo, mantenha uma comunicação com as regras da empresa e as camadas de interface.

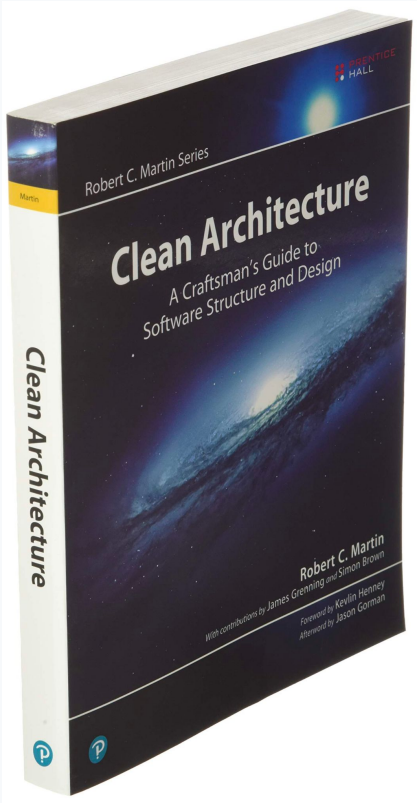


Entendendo a Clean Architecture 💻



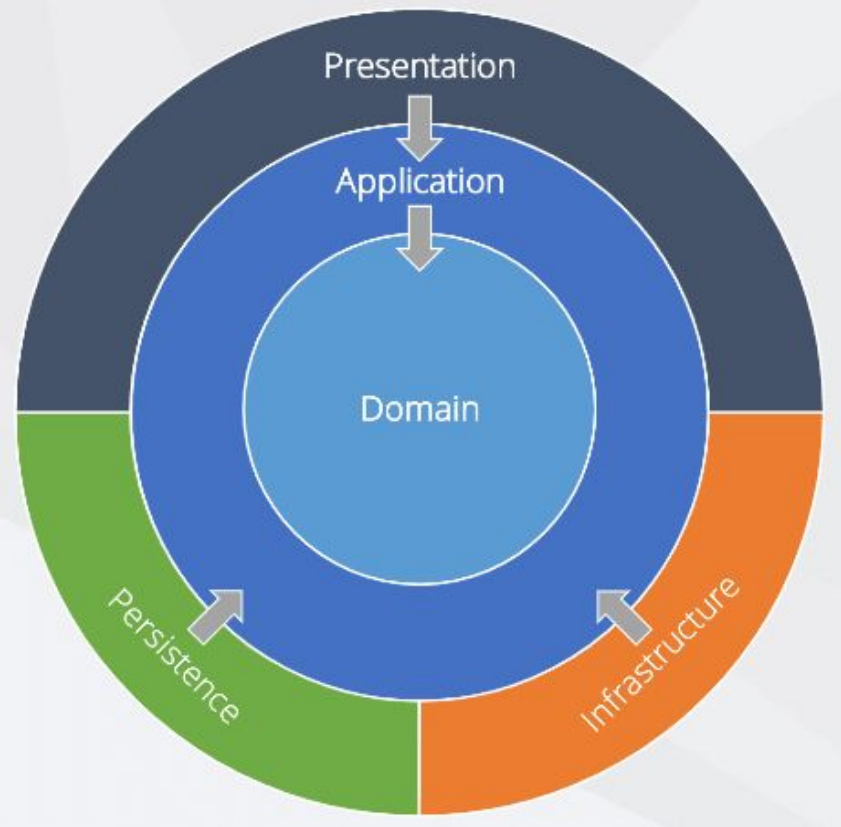
Vantagens da Arquitetura em Camadas

- **Testável:** cada camada pode ser testada separadamente.
- **Independente de interface do usuário:** a interface do usuário pode mudar facilmente, sem alterar o restante do sistema. Uma API Web pode ser substituída por um console, por exemplo, sem alterar as regras de negócios.
- **Independente de banco de dados:** você pode trocar o MySQL por Mongo ou qualquer outro banco. Suas regras de negócios não estão vinculadas ao banco de dados.
- **Independente de qualquer agente externo:** suas regras de negócios simplesmente não sabem nada sobre o mundo exterior, não estão ligadas a nenhum framework.



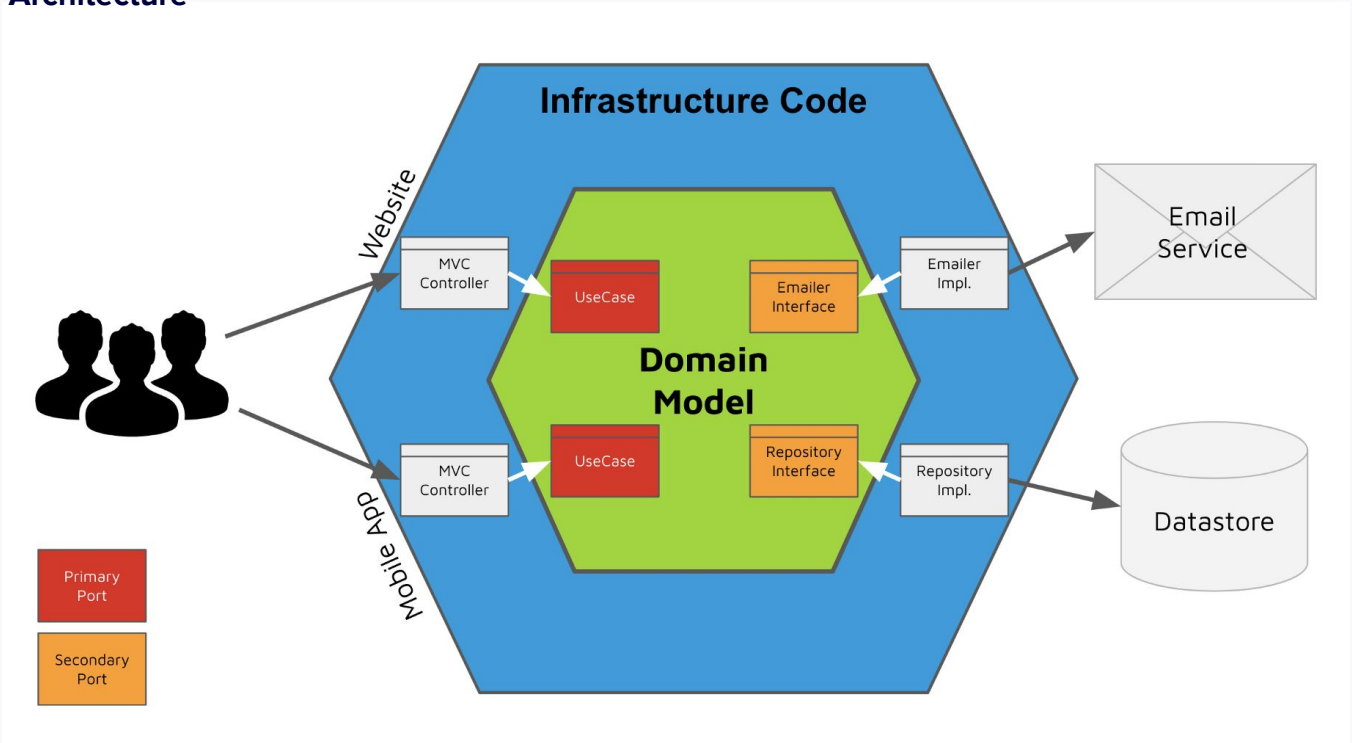
Modelos de SoC (Separation of Concerns) 💻

Onion Architecture



Modelos de SoC (Separation of Concerns) 💻

Hexagonal Architecture

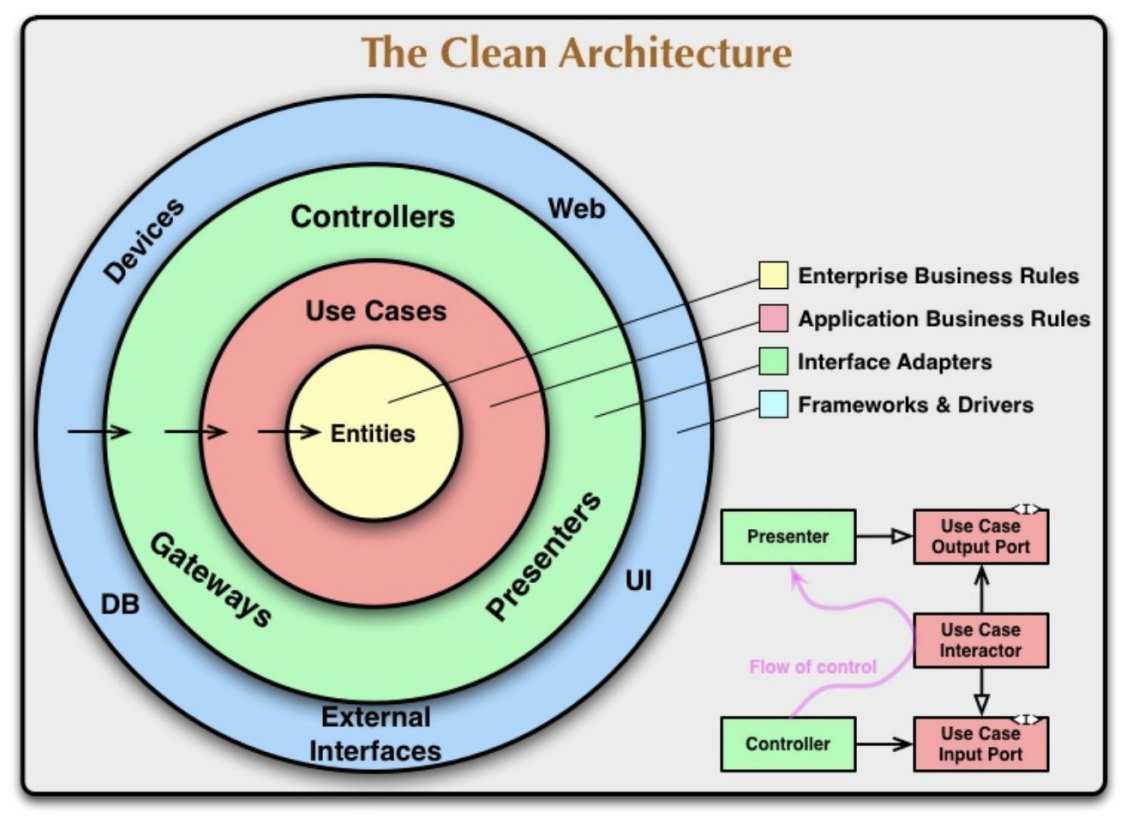


awari

ESLint e Sonar

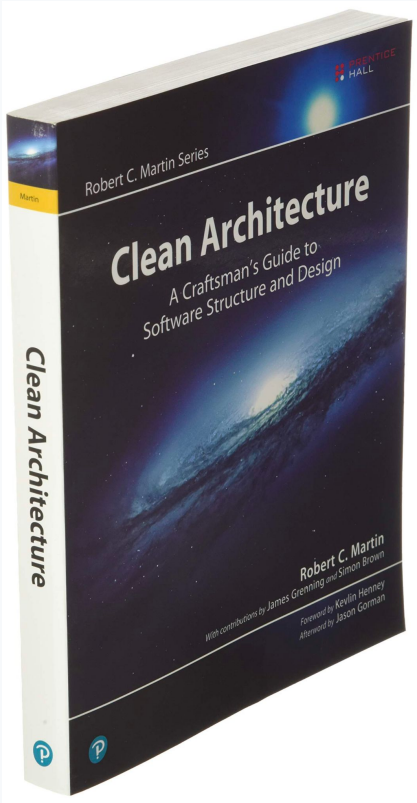
Automatizando a correção de código

Entendendo a Clean Architecture 💻



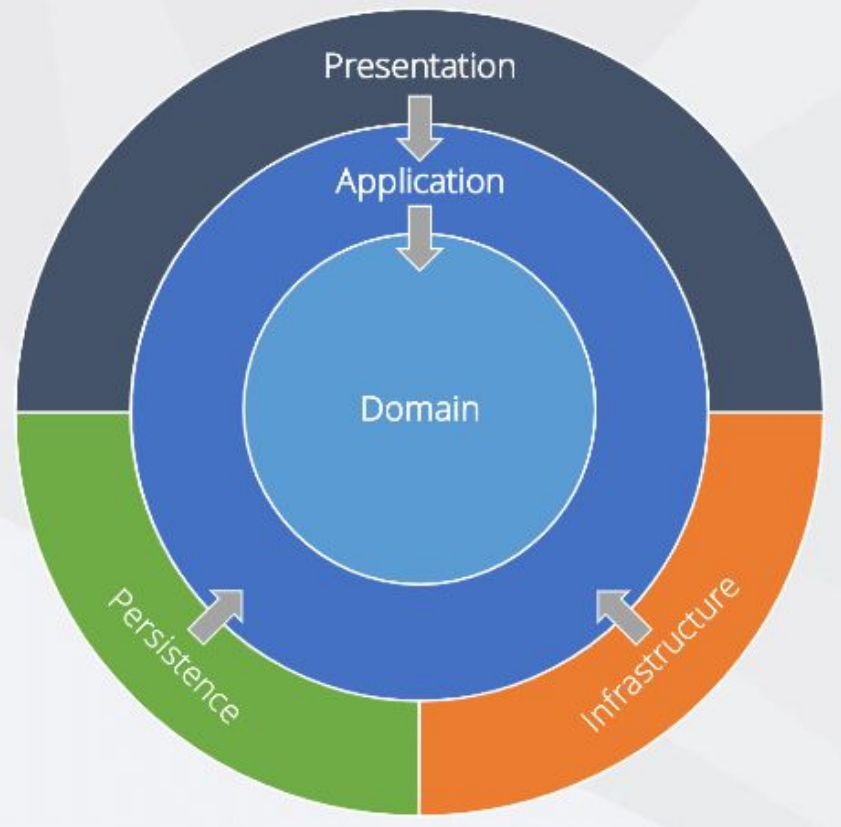
Vantagens da Arquitetura em Camadas

- **Testável:** cada camada pode ser testada separadamente.
- **Independente de interface do usuário:** a interface do usuário pode mudar facilmente, sem alterar o restante do sistema. Uma API Web pode ser substituída por um console, por exemplo, sem alterar as regras de negócios.
- **Independente de banco de dados:** você pode trocar o MySQL por Mongo ou qualquer outro banco. Suas regras de negócios não estão vinculadas ao banco de dados.
- **Independente de qualquer agente externo:** suas regras de negócios simplesmente não sabem nada sobre o mundo exterior, não estão ligadas a nenhum framework.



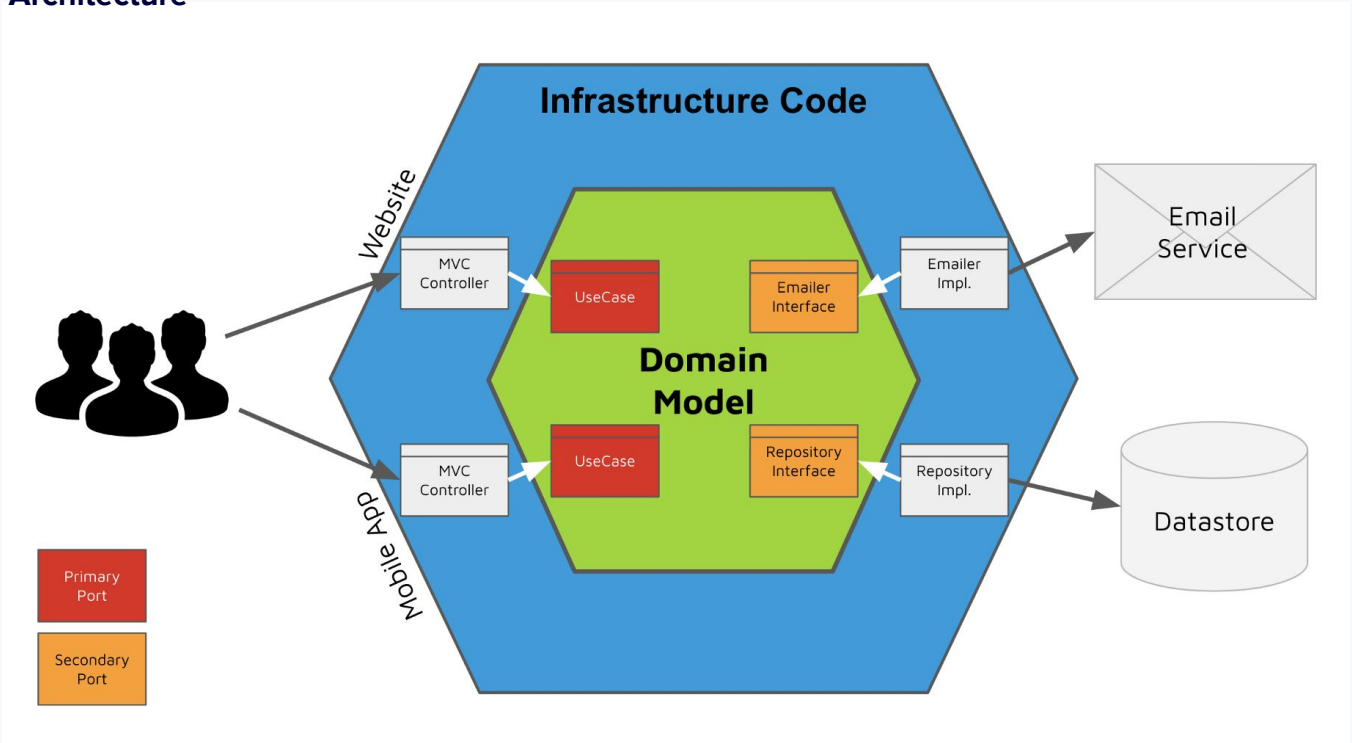
Modelos de SoC (Separation of Concerns) 💻

Onion Architecture



Modelos de SoC (Separation of Concerns) 💻

Hexagonal Architecture



awari

ESLint e Sonar

Automatizando a correção de código

O que é ESLint?

O **ESLint** é uma ferramenta de análise de código estático que, juntamente com a extensão de mesmo nome disponível no **VSCode**, permite identificar erros quanto ao padrão de escrita que definimos.

Resumindo, o **ESLint** vai analisar seu código, com base nos padrões que você definiu, e vai indicar o que está errado.

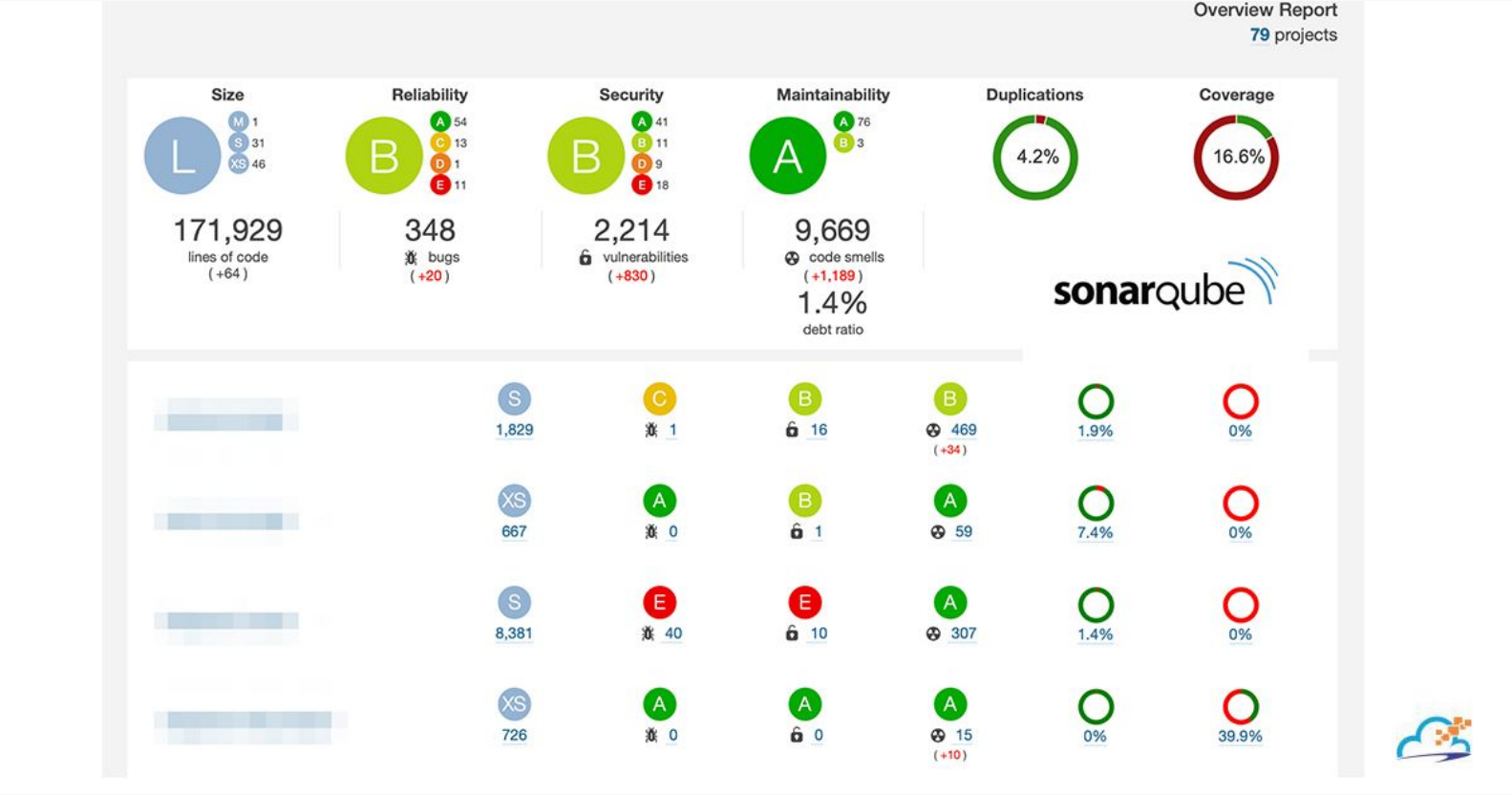
O que é Sonar? 

O **Sonar** é uma ferramenta *open-source* construída para avaliar a qualidade do código que está sendo desenvolvido.

Questões de arquitetura, código duplicado, pontos potenciais de bugs, nível de complexidade e cobertura de testes, são alguns dos itens avaliados pelo **Sonar**, de modo a dar para a equipe um feedback sobre a qualidade do seu código.

O **Sonar** pode ser configurado para armazenar todas as informações do seu código em um banco de dados para que se possa fazer também um acompanhamento da evolução da qualidade do código.

O que é Sonar?


Overview Report
79 projects
Size
L
M 1
S 31
XS 46
171,929
lines of code
(+64)
Reliability
B
A 54
C 13
D 1
E 11
348
bugs
(+20)
Security
B
A 41
B 11
D 9
E 18
2,214
vulnerabilities
(+830)
Maintainability
A
A 76
B 3
9,669
code smells
(+1,189)
1.4%
debt ratio
Duplications
4.2%
Coverage
16.6%
sonarqube
S 1,829
C 1
B 16
B 469
(+34)
1.9%
0%
XS 667
A 0
B 1
A 59
7.4%
0%
S 8,381
E 40
E 10
A 307
1.4%
0%
XS 726
A 0
A 0
A 15
(+10)
0%
39.9%

O que é ESLint?

O **ESLint** é uma ferramenta de análise de código estático que, juntamente com a extensão de mesmo nome disponível no **VSCode**, permite identificar erros quanto ao padrão de escrita que definimos.

Resumindo, o **ESLint** vai analisar seu código, com base nos padrões que você definiu, e vai indicar o que está errado.

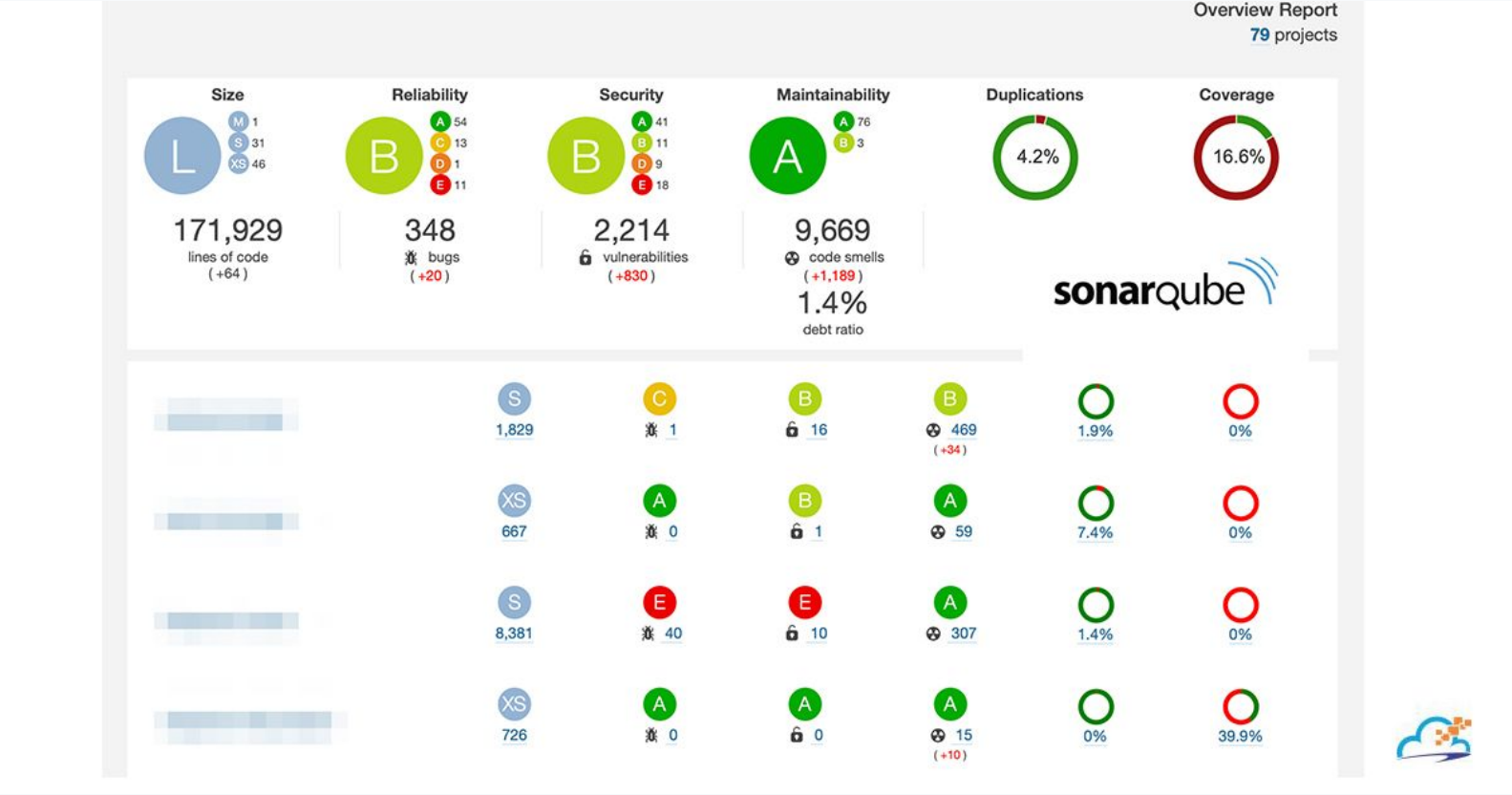
O que é Sonar? 

O **Sonar** é uma ferramenta *open-source* construída para avaliar a qualidade do código que está sendo desenvolvido.

Questões de arquitetura, código duplicado, pontos potenciais de bugs, nível de complexidade e cobertura de testes, são alguns dos itens avaliados pelo **Sonar**, de modo a dar para a equipe um feedback sobre a qualidade do seu código.

O **Sonar** pode ser configurado para armazenar todas as informações do seu código em um banco de dados para que se possa fazer também um acompanhamento da evolução da qualidade do código.

O que é Sonar?


Overview Report
79 projects
Size
L
M 1
S 31
XS 46
171,929
lines of code
(+64)
Reliability
B
A 54
C 13
D 1
E 11
348
bugs
(+20)
Security
B
A 41
B 11
D 9
E 18
2,214
vulnerabilities
(+830)
Maintainability
A
A 76
B 3
9,669
code smells
(+1,189)
1.4%
debt ratio
Duplications
4.2%
Coverage
16.6%
sonarqube
S 1,829
C 1
B 16
B 469
(+34)
1.9%
0%
XS 667
A 0
B 1
A 59
7.4%
0%
S 8,381
E 40
E 10
A 307
1.4%
0%
XS 726
A 0
A 0
A 15
(+10)
0%
39.9%

awari

Padrões de Projeto

Soluções para problemas recorrentes

O que é um Padrão de Projeto? 

Definir o que é um **padrão de projeto** de maneira clara e objetiva, tem sido o desafio da comunidade de software, desde a década de 80.

O primeiro a apresentar uma definição do que seria um padrão, foi o professor de arquitetura suíço Christopher Alexander, no seu livro *"The Timeless Way of Building"* (Oxford University Press, 1979), que é: "Cada padrão é uma **regra de três partes**, que expressa uma relação entre um certo **contexto**, um **problema** e uma **solução**".

Então, podemos definir o **padrão de projeto** como uma solução recorrente para um problema em um contexto específico, mesmo que em áreas distintas.

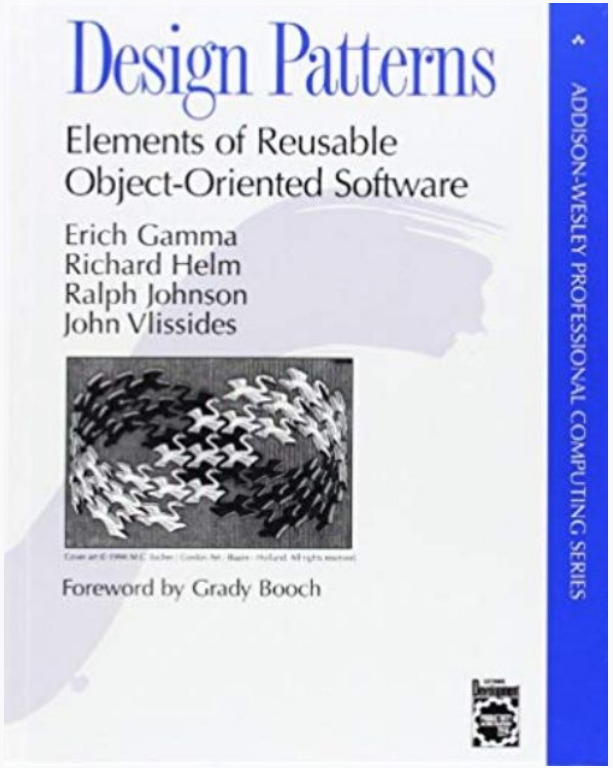
GoF - Gang of Four

Alguns programadores mais experientes começaram a perceber que os mesmos problemas começaram a aparecer várias e várias vezes, e a solução para aqueles problemas eram sempre as mesmas. Assim, eles começaram a catalogar esses padrões.

Em 1995, um grupo de quatro pessoas escreveu um livro instituindo os *Design Patterns* (Padrões de Projeto) mais conhecidos de mercado, são eles:

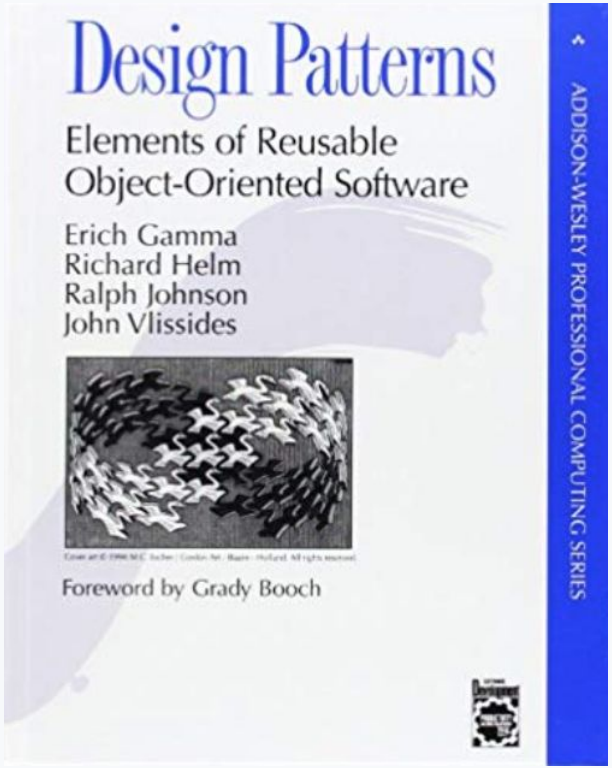
- Erich Gamma
- Richard Helm
- Ralph Johnson
- John Vlissides

Eles ficaram conhecidos como Gang of Four ou, simplesmente, GoF.



GoF - Gang of Four

Criação	Estrutural	Comportamental	
Abstract Factory	Adapter	Chain of Responsibility	Observer
Builder	Facade	Command	State
Factory Method	Bridge	Interpreter	Strategy
Prototype	Decorator	Iterator	Template Method
Singleton	Flyweight	Mediator	Visitor
	Composite	Memento	
	Proxy		



Galeria de ícones

Cores

#1673FA

#F7F9FB

#FFFFFF

#202124

#595959

Fontes

Título

Highlight

Body - Title

Body - Regular

Sombra

Logo

awari awari

awari