

Boas práticas e clean code

Bruno Yokio Tatsumi
Coffee & Code
Agroindústria
Assis, SP

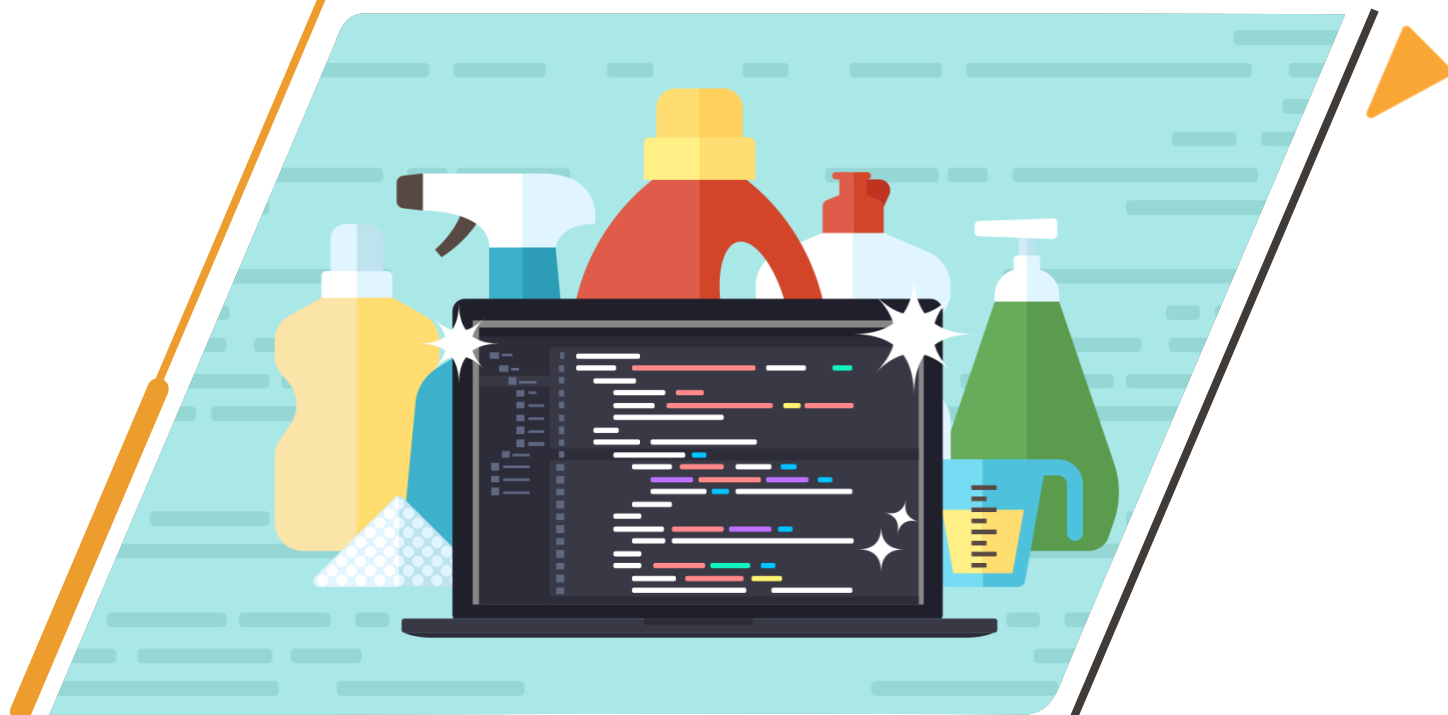


TODOS OS DIREITOS RESERVADOS

2019



Objetivos



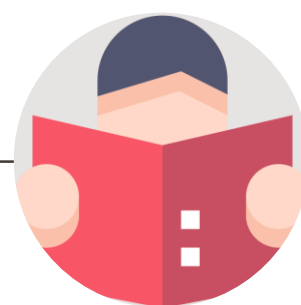
1. **Por quê boas práticas?**
2. **Regras gerais**
3. **SOLID, DRY, KISS, etc**
4. **Testes**
5. **Por onde começar?**



Por quê?



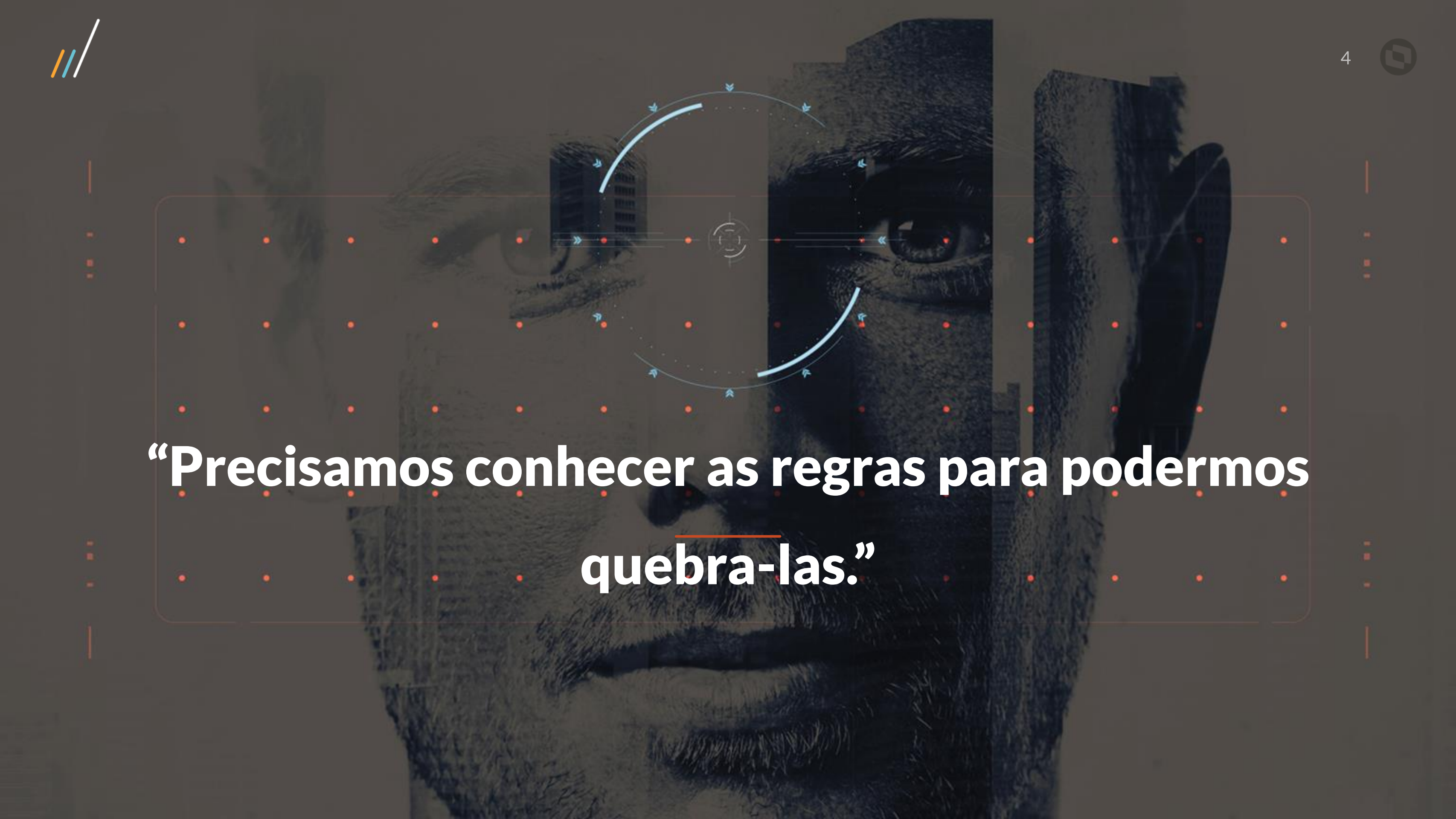
**Padronização de acordo com
mercado (facilidade de
adaptação de novos integrantes)**



**Facilidade na leitura =
(menos manutenções +
produtividade)**



**Por serem práticas já consolidadas
no mercado, não precisamos
reinventar a roda para saber o que
funciona (e o que não funciona)**



**“Precisamos conhecer as regras para podermos
quebra-las.”**



Regras gerais





Nomenclatura (de variáveis, métodos, classes, etc.)



No geral, devemos dar nomes descritivos (que identifiquem o que faz esse código) e se esse nome fica muito grande, podemos estar ferindo o princípio da responsabilidade única e talvez seja possível quebrar em mais partes.

Classes: Não deve ser uma frase, e deve conter o domínio ao qual essa classe é responsável. Opcionalmente podemos utilizar algum sufixo que identifique a especialidade em relação à este domínio. Ex: Produto, Cliente, ProdutoService, ClienteDAO.

Métodos: Normalmente é um verbo ou uma frase, que descreve exatamente o que ele faz.

Variáveis: O ideal é utilizarmos nomes significativos em relação à informação que estamos armazenando. São preferíveis nomes pequenos, porém, é melhor um nome grande do que um que ninguém vai entender. Ex: Melhor usar `codigoFuncionarioDemitido` do que `cdFuncDem`.



Métodos (ou funções)



Os métodos devem ter uma responsabilidade única (somente um motivo para existir) e com um nome que sirva como documentação.

- Dentro de um for/while/if/else/try, ter somente uma linha (com a chamada de um método) é uma boa prática;
- No geral, é indicado que tenhamos no máximo 3 argumentos (se necessário, criar um wrapper);
- Evitar argumentos booleanos;
- Quebrar um método grande em outros menores (melhor pra testar e pra ler).



```
public void excluirProduto(Produto produto) {  
    try {  
        if (!validacoesProduto.isPermitidoExcluir(produto)) {  
            throw new Exception("Não é permitido excluir o Produto.");  
        }  
        fotoProdutoService.excluirFotos(produto);  
        produtoService.excluir(produto);  
    } catch (Exception e) {  
        // tratamento de erro  
    }  
}
```




```
public void excluirProduto(Produto produto) {  
    try {  
        excluirProdutoComDependencias(produto);  
    } catch (Exception e) {  
        // tratamento de erro  
    }  
}
```



Métodos (ou funções)



```
public boolean validar(int quantidadeProduto, String codigoProduto, BigDecimal valorUnitario, String codigoPedido, Date dataPedido) {  
    //validações  
    return true;  
}
```



```
public boolean validar(Pedido pedido, Produto produto) {  
    //validações  
    return true;  
}
```




```
public boolean validar(Pedido pedido, Produto produto) {  
    if (produto.getCodigo() == null) {  
        throw new IllegalArgumentException("Produto com código inválido!");  
    }  
    if (produto.getDescricao() == null) {  
        throw new IllegalArgumentException("Produto com descrição inválida!");  
    }  
    if (produto.getValorUnitario() == null) {  
        throw new IllegalArgumentException("Produto com valor unitário inválido!");  
    }  
    if (pedido.getCodigo() == null) {  
        throw new IllegalArgumentException("Pedido com código inválido!");  
    }  
    PedidoService service = new PedidoService();  
    if (service.isFaturado(pedido)) {  
        throw new IllegalArgumentException("O pedido já foi faturado!");  
    }  
    return true;  
}
```



```
public boolean validar(Pedido pedido, Produto produto) {  
    validarInformacoesProduto(produto);  
    validarInformacoesPedido(pedido);  
    return true;  
}
```



Composição de objetos e estrutura de dados



- Pare de criar getter/setter pra tudo;
- Centralize regras para manipular o objeto (adicionar ou remover um item de uma lista);
- Expor só o necessário;
- Separar responsabilidades (negócio, infra, código específico de framework).



Princípios





Termo cunhado por Robert C. Martin (uncle Bob) – autor de livros como: Clean code, Agile principles patterns, Clean architecture, Clean coder, entre outros. O objetivo destes princípios é definir alguns padrões que deixam o código mais limpo, e, conseqüentemente mais fácil de se entender, testar e manter.

- S = Single responsibility principle
- O = Open/close principle
- L = Liskov substitution
- I = Interface segregation
- D = Dependence inversion

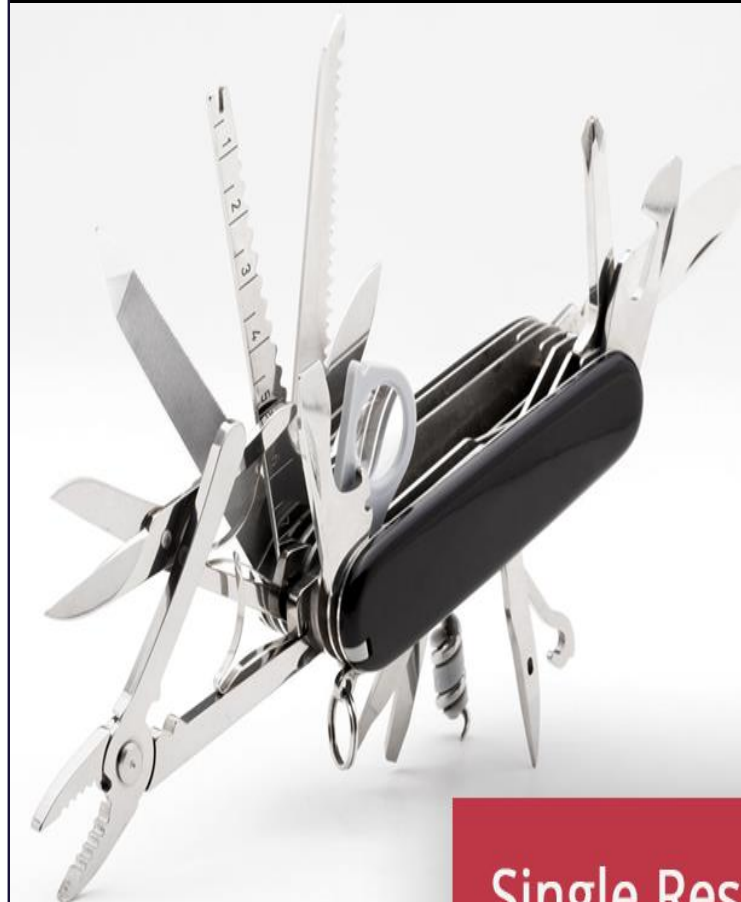


S.O.L.I.D – Single responsibility

17



Toda unidade de código (muitos interpretam que essa unidade de código seria uma classe, mas atualmente a maioria considera essa unidade como uma função ou método) deve ter somente um motivo para existir, ou seja, se um método está realizando mais de uma tarefa (por exemplo buscando informações no banco de dados e realizando alguma validação), significa que podemos refatorar, quebrando-o em mais métodos.



"Because your class is not a multitool"

Single Responsibility Principle



S.O.L.I.D – Open/close

18



Este princípio prega que deixemos as nossas classes fechadas para modificações e, ao mesmo tempo, abertas à evolução. A ideia é que nossas APIs sejam fáceis de se evoluir, e ao mesmo tempo difíceis de serem quebradas por alterações erradas. Se seguirmos este princípio, quando for necessário evoluir alguma funcionalidade, vai ser necessário criar um novo código, e não alterar um existente. Exemplo: fábrica de validações.



OPEN CLOSED PRINCIPLE

Brain surgery is not necessary when putting on a hat.

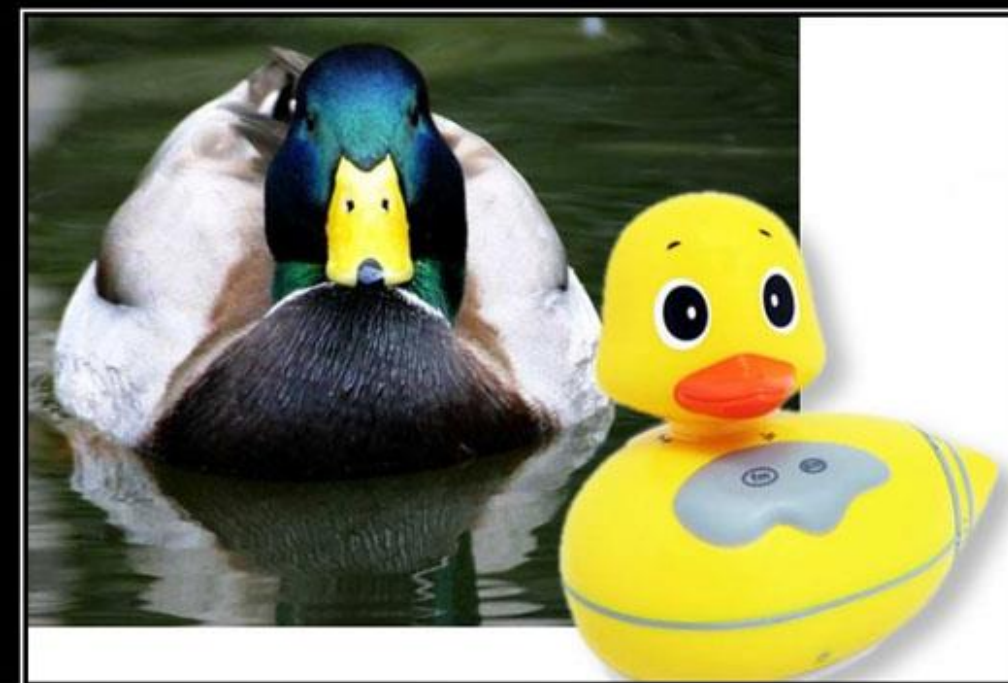


S.O.L.I.D – Liskov substitution

19



O nome deste princípio tem como origem um artigo da Barbara Liskov que trata sobre os relacionamentos de objetos em OOP. Basicamente, a indicação é que os subtipos possam ser substituídos pelos seus tipos bases. Se vamos utilizar a herança, devemos definir exatamente qual deverá ser o comportamento nas nossas classes bases (para termos o máximo de aproveitamento do recurso de herança), e uma forma de medir, é exatamente o que é sugerido: “Subtypes must be substitutable for their base types.”.



LISKOV SUBSTITUTION PRINCIPLE

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



S.O.L.I.D – Interface segregation

Este princípio se refere à quanto das implementações estarão expostas ao cliente (cliente aqui não é um browser ou aplicativo, mas sim o código que irá usar nossa biblioteca/API). O indicado aqui é que tenhamos o mínimo possível de métodos expostos (públicos). Um exemplo comum, é o caso dos getters e setters, muitas vezes acabamos criando, por padrão, esses métodos para todos os atributos (mesmo sem necessidade, e sem pensar nas consequências disso).



INTERFACE SEGREGATION

Tailor interfaces to individual clients' needs.



S.O.L.I.D – Dependency inversion

Este é um princípio que busca separar as implementações, evitando que elas “vazem” de contextos. A ideia é que as nossas classes não dependam diretamente de uma implementação, mas de uma interface (contrato), com isso, podemos ter múltiplas implementações e também conseguimos mantê-las facilmente, evitando que o módulo que depende delas fique com erros. Exemplo: Classes com o sufixo Repository (muito utilizadas no Spring).

Obs: Tomar cuidado para não cometer o erro de ficar criando interfaces que tenham somente uma implementação.



Dependency Inversion Principle

Would you solder a lamp directly to the electrical wiring in a wall?

KISS

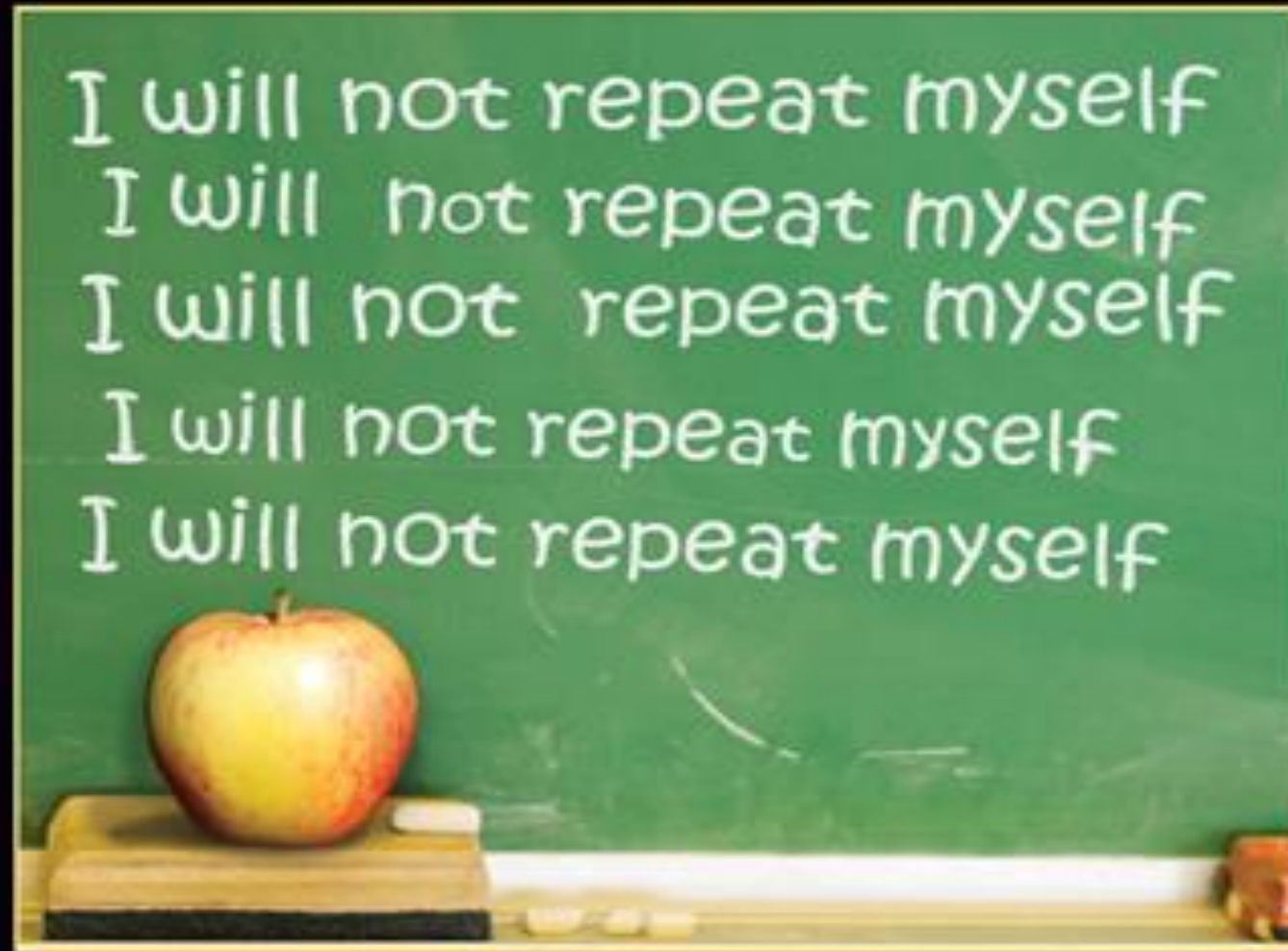
keep.it.simple.stupid.



- Capacidade de resolver mais problemas (e mais rapidamente);
- Resolver problemas complexos, com menos linhas de código;
- Maior qualidade de código;
- Melhorar a manutenibilidade dos nossos sistemas;
- Base de código mais flexível, fácil de se evoluir e/ou refatorar;
- Facilidade em se trabalhar com equipes maiores, uma vez que o código passa a ser escrito de maneira simples.



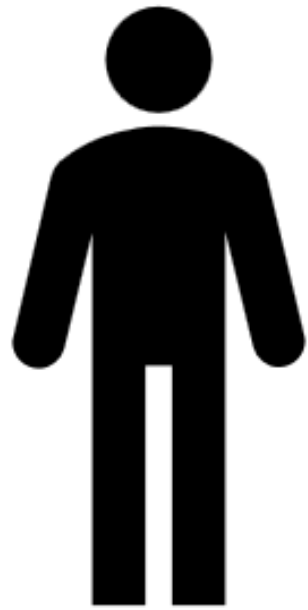
- Seja humilde (o primeiro erro, é achar que não está errado);
- Quebre suas tarefas em pequenas sub-tarefas que não durem entre 4 e 12 horas;
- Quebre os problemas a serem resolvidos em problemas menores. Cada problema deveria ser resolvido com uma (ou poucas classes);
- Métodos com no máximo 40 linhas. Se tiver uma condição dentro de um método, ela deve ser outro método (Use o poder de refatoração da IDE);
- **Primeiro resolva o problema real, depois programe;**
- Vá programando e refatorando (e programando novamente, e depois refatorando...);
- Não tente reinventar a roda. Em todos os cenários, tente fazer da maneira mais simples possível.



DON'T REPEAT YOURSELF

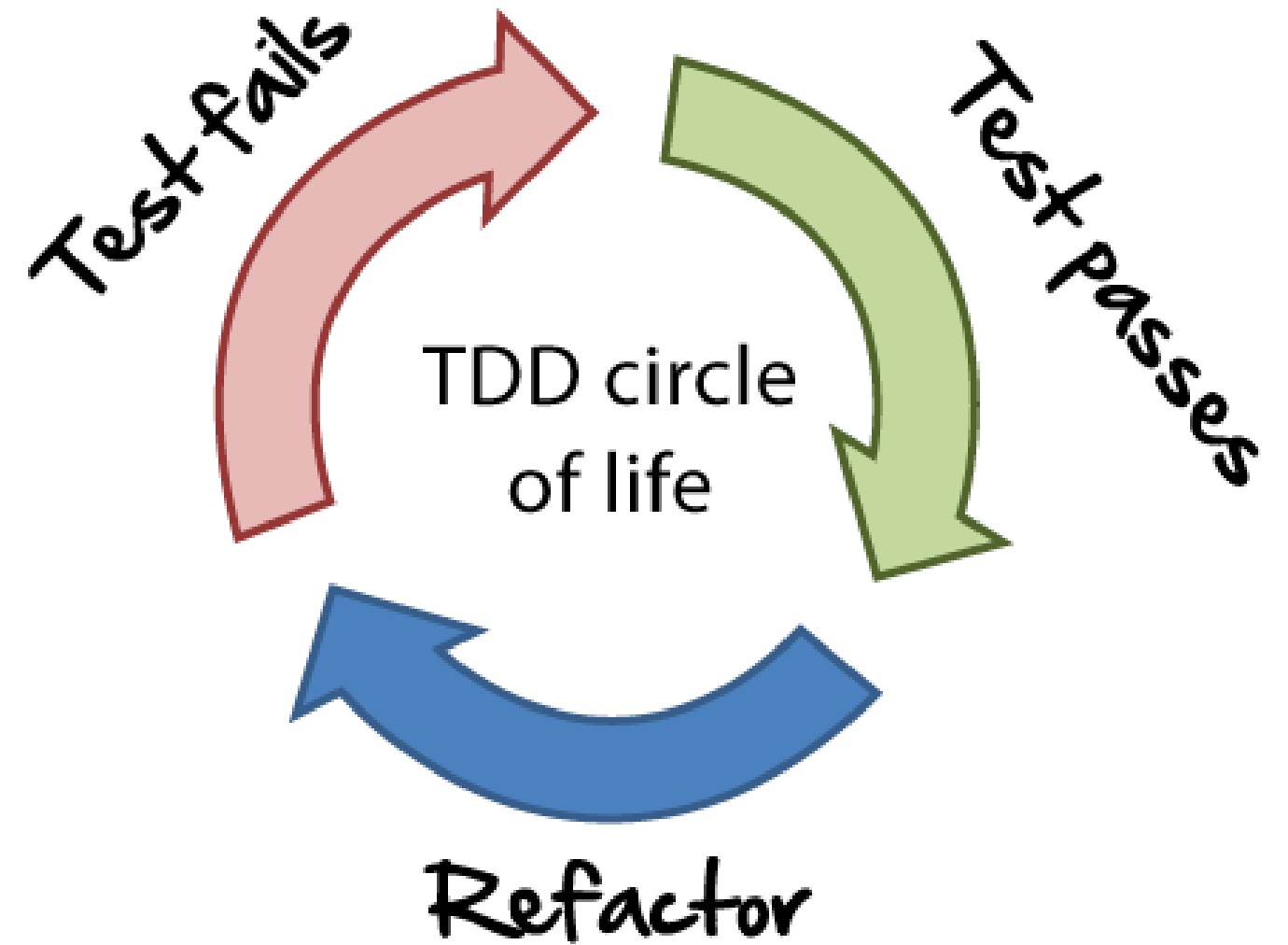
Repetition is the root of all software evil.

Manual Testing



VS

Automated Testing





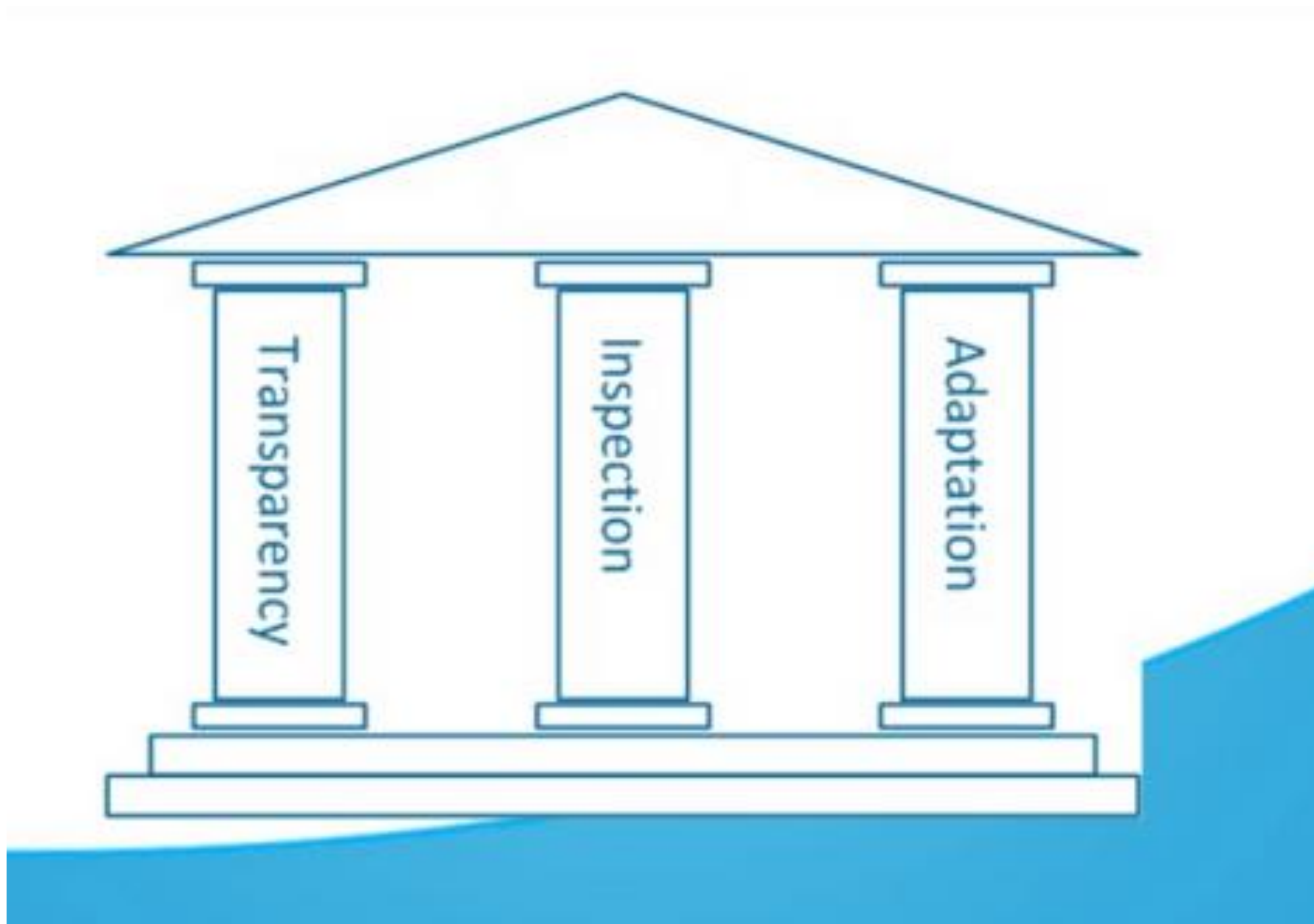
Por onde começar?

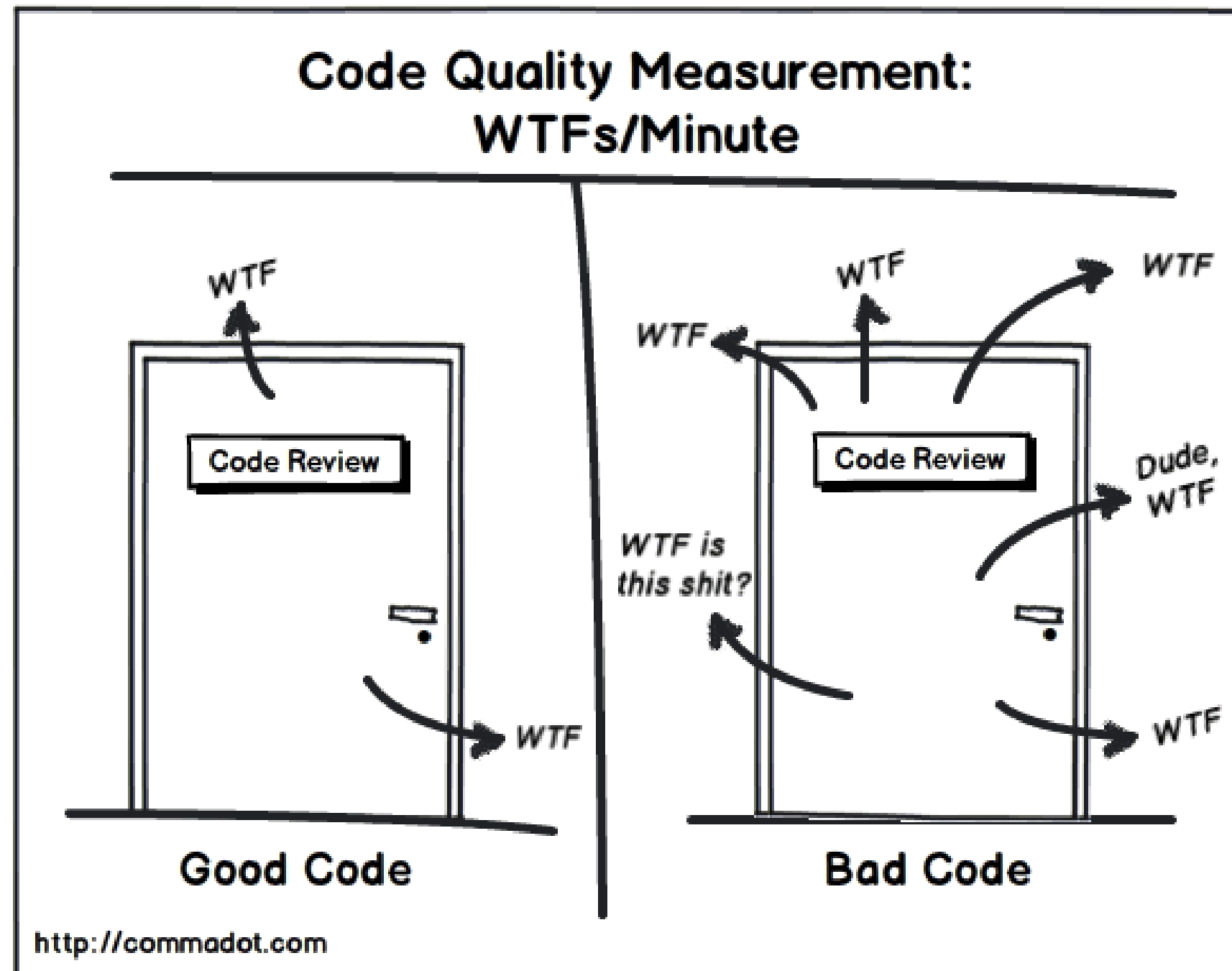




Por onde começar?

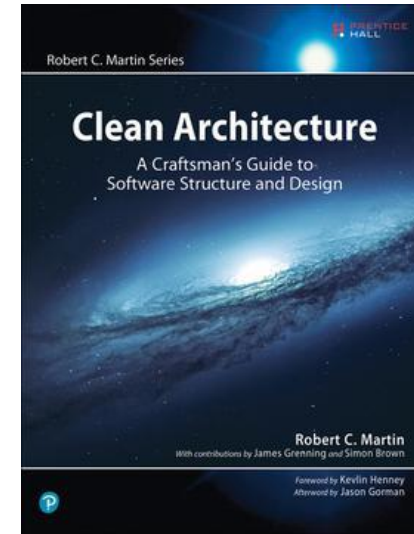
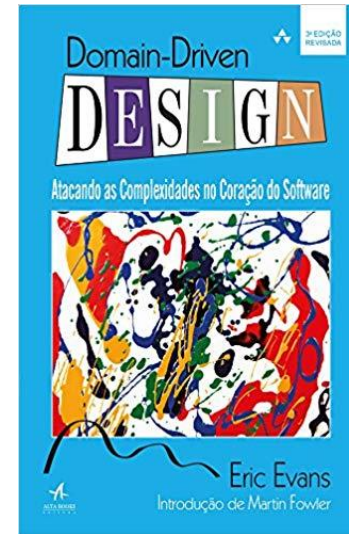
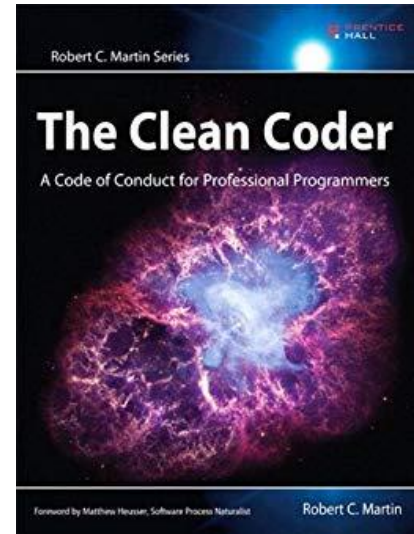
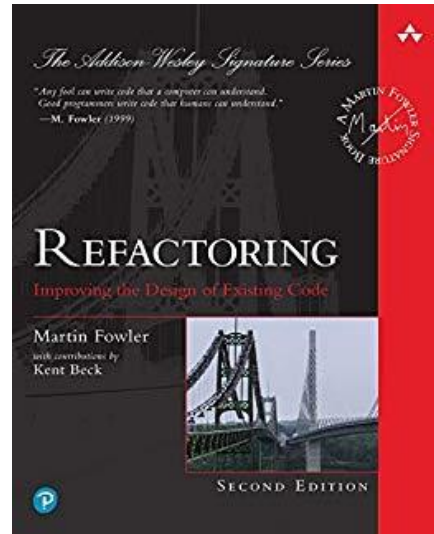
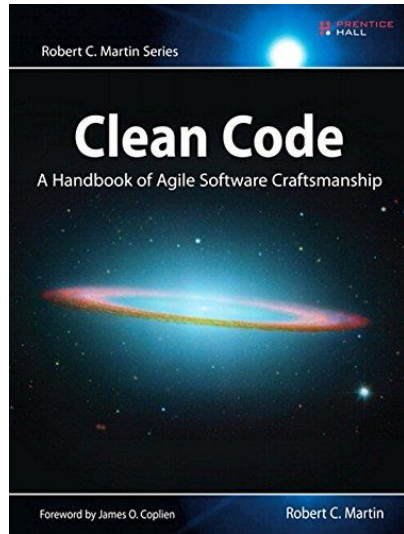
28





sonarlint

- Identifique situações que façam sentido utilizar TDD;
- Cada bug, um teste;
- Cada regra, mais um teste.

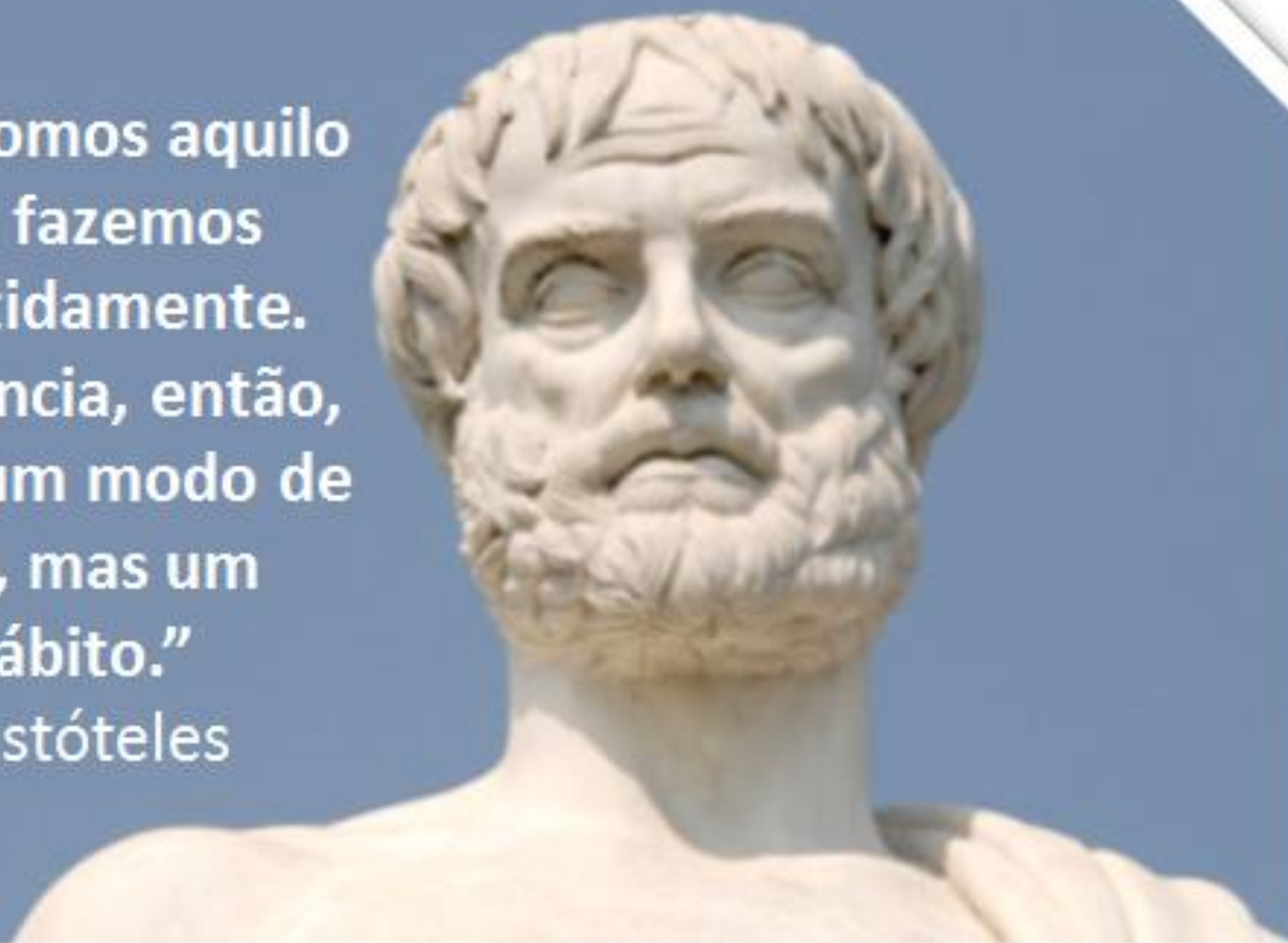




Praticar, praticar e praticar



“Nós somos aquilo
que fazemos
repetidamente.
Excelência, então,
não é um modo de
agir, mas um
hábito.”
Aristóteles





Bruno Tatsumi

Desenvolvimento

bruno.tatsumi@totvs.com.br

**Tecnologia + Conhecimento são nosso DNA.
O sucesso do cliente é o nosso sucesso.
Valorizamos gente boa que é boa gente.**

 totvs.com

 company/totvs

 @totvs

 fluig.com

#SOMOSTOTVERS