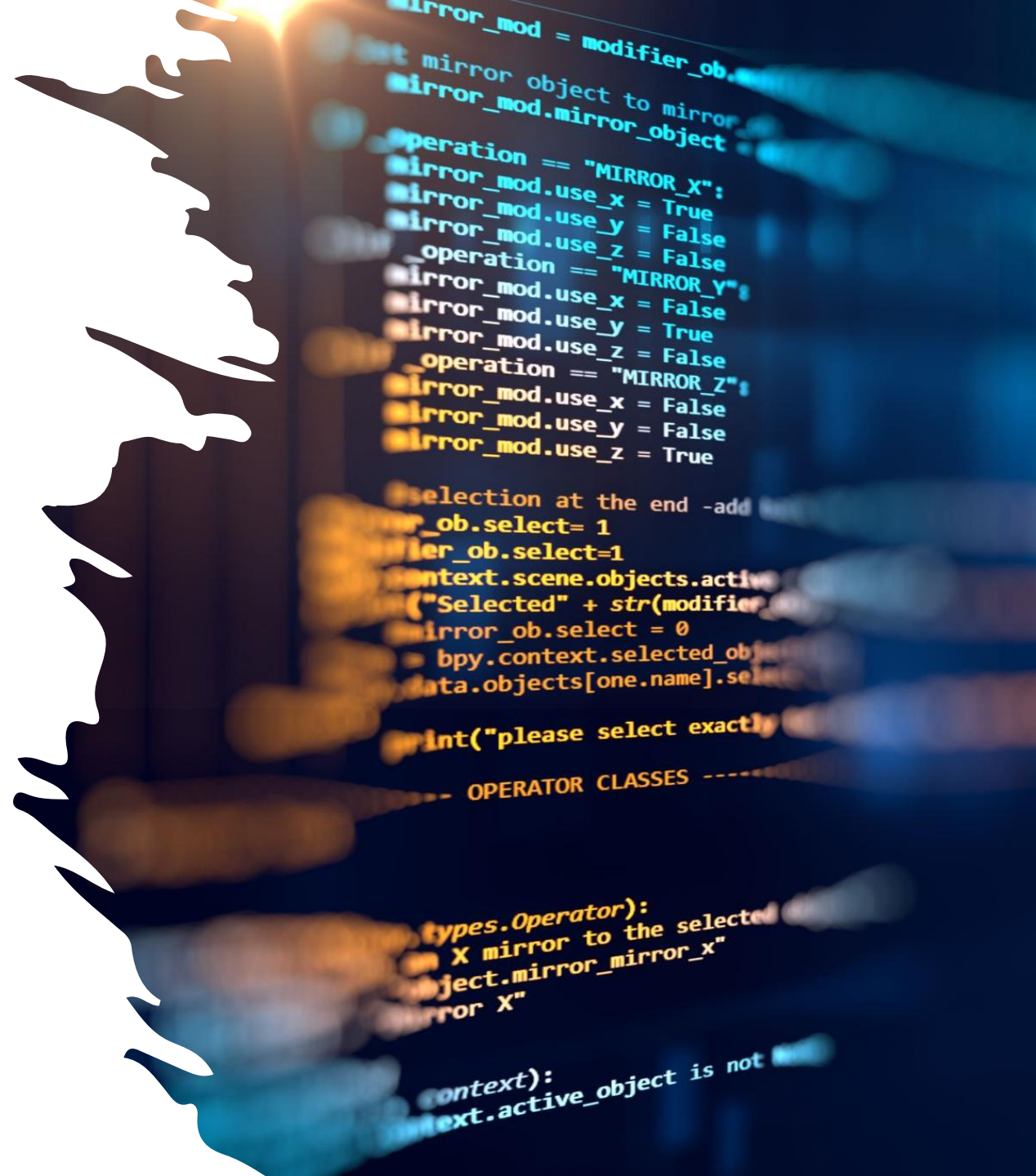


Boas práticas de programação

Especialização em desenvolvimento Web – Turma 6

PROFESSOR: DR. RODRIGO
DA CRUZ FUJIOKA

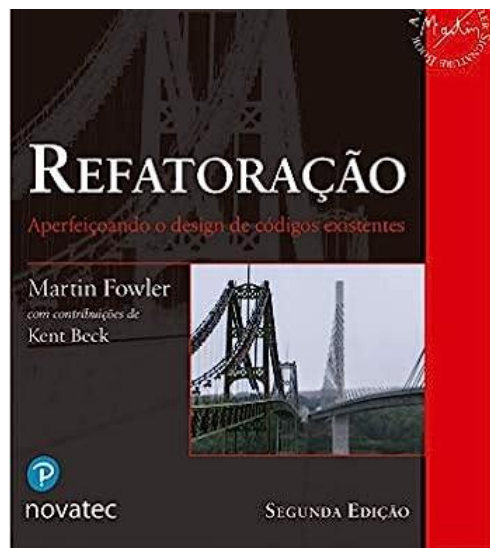


```
mirror_mod = modifier_ob.  
set mirror object to mirror.  
mirror_mod.mirror_object =  
operation == "MIRROR_X":  
mirror_mod.use_x = True  
mirror_mod.use_y = False  
mirror_mod.use_z = False  
operation == "MIRROR_Y":  
mirror_mod.use_x = False  
mirror_mod.use_y = True  
mirror_mod.use_z = False  
operation == "MIRROR_Z":  
mirror_mod.use_x = False  
mirror_mod.use_y = False  
mirror_mod.use_z = True
```

```
selection at the end -add  
_ob.select= 1  
_ob.select=1  
context.scene.objects.active  
("Selected" + str(modifier_ob.  
mirror_ob.select = 0  
= bpy.context.selected_object  
data.objects[one.name].select  
print("please select exactly
```

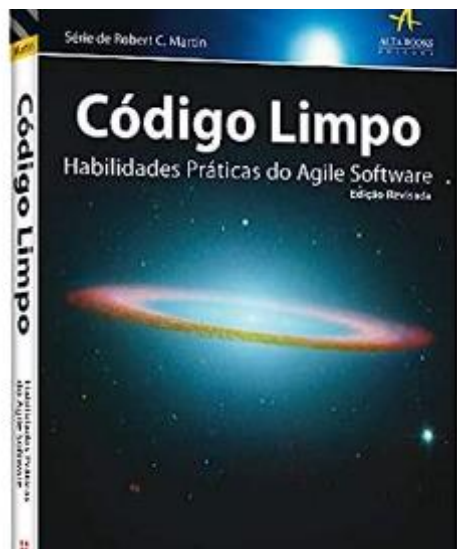
```
-- OPERATOR CLASSES --  
types.Operator):  
X mirror to the selected  
object.mirror_mirror_x"  
mirror X"
```

```
context):  
context.active_object is not
```



Recomendações de Leitura

(REFATORAÇÃO DE CÓDIGO E
SOLID)





Boas práticas de Programação

REFATORAÇÃO DE CÓDIGO.

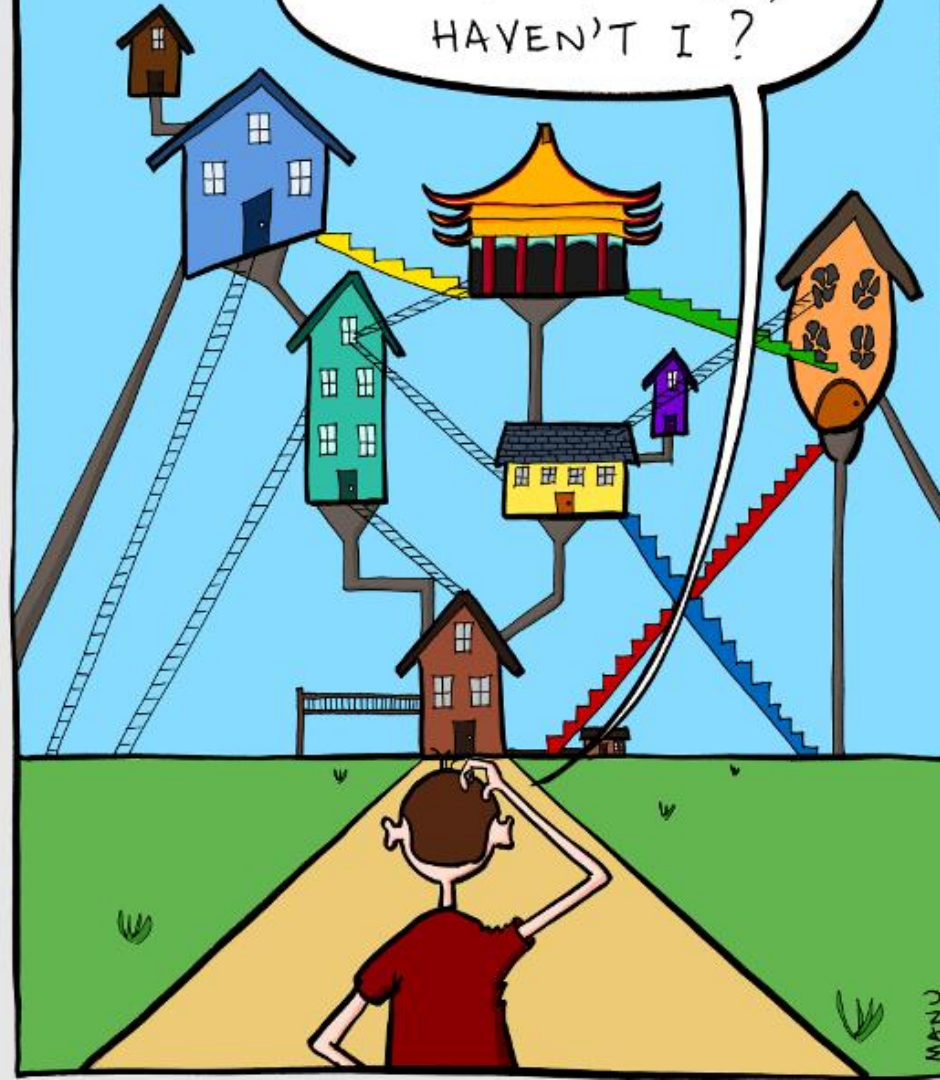
THE LIFE OF A SOFTWARE ENGINEER.

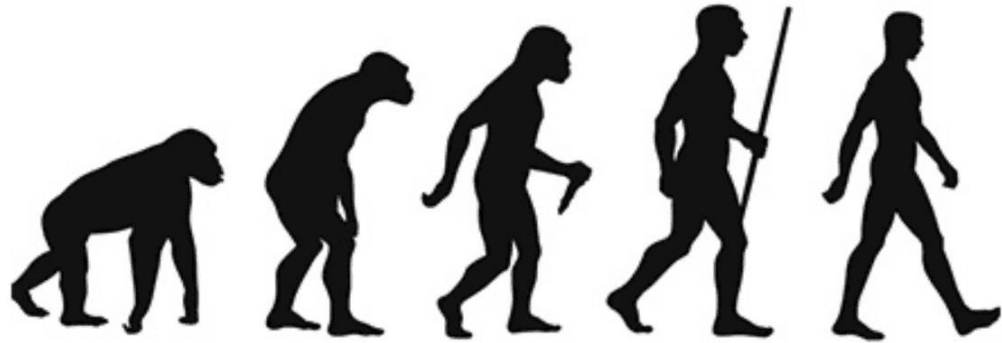
CLEAN SLATE. SOLID FOUNDATIONS. THIS TIME I WILL BUILD THINGS THE RIGHT WAY.



MUCH LATER...

OH MY. I'VE DONE IT AGAIN, HAVEN'T I ?





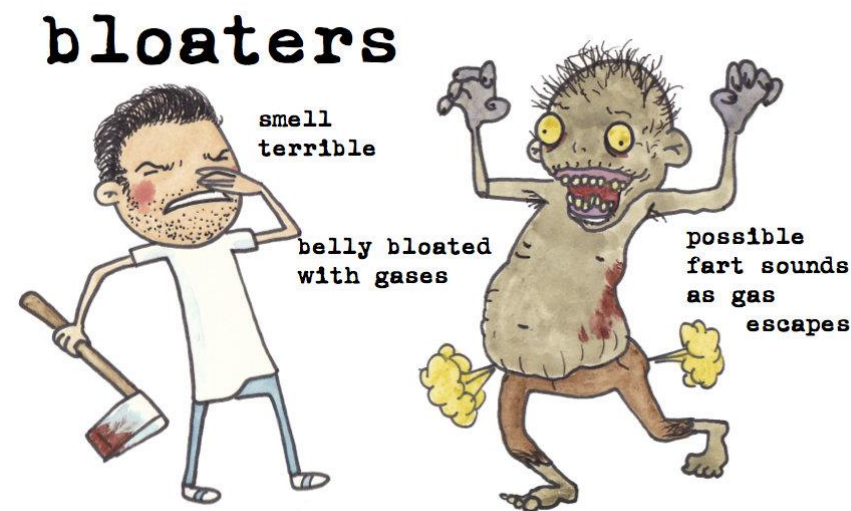
Refactoring

Improving the Design of Existing Code



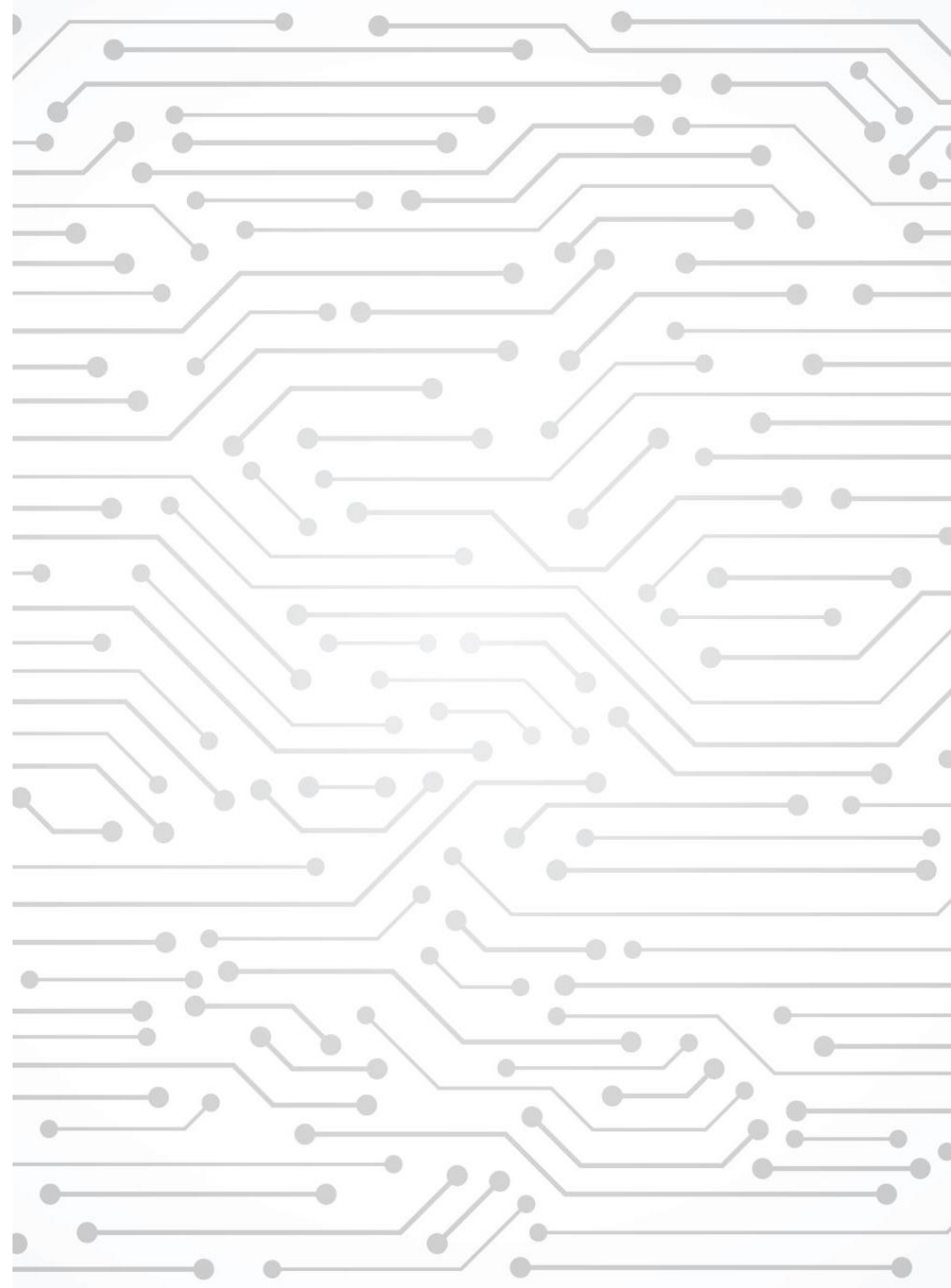
Bloters

- São códigos, métodos e classes que aumentaram ao longo do tempo. Esse é o tipo de coisa que vai aparecendo ao longo do tempo, que é mais comum em projetos legados. Pois é algo que vai sendo acumulado ao longo do tempo.



Long Method

- São métodos que contém muitas linhas de código, quando você observa métodos com mais de 10 linhas é um sinal para reflexão ou atenção.



Quando devo refatorar um código?

Sempre que possível e fizer sentido.

Lembre-se que refatorar código que já está em produção requer um cuidado especial.

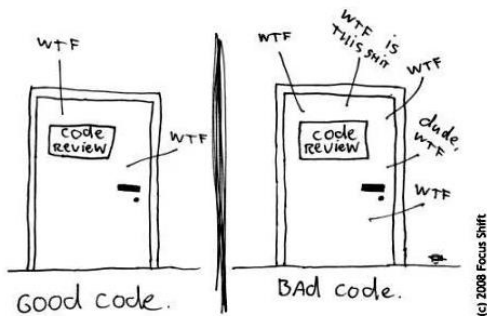
O Code review e análises e melhorias constantes ajudam a evitar a necessidade de refatorações complexas.



Exemplo


- Um método que é bastante reutilizado em um contexto.
- Então do desenvolvedor pensa, nossa são apenas mais 3 linhas adicionadas ao que já existe, para que criar um método novo.

The only valid measurement
of code quality: WTFs/minute



"Mas são apenas três ou quatro linhas, não adianta criar um método inteiro só para isso..."

Se você não entende algo dentro do código ou existe a necessidade de comentar para entender você deve colocar ele em um novo método.
(Lembre-se do Clean Code)



```
public Conta atualizar(Conta conta){
    if(conta.getId()==null){
        throw new RuntimeException("id da conta nulo");
    }
    conta = repository.save(conta);
    return conta;
}

public void excluir(Conta conta){
    if(conta.getId()==null){
        throw new RuntimeException("id da conta nulo");
    }
    repository.delete(conta);
}
```

```
public Conta atualizar(Conta conta){  
    validateSaveRemoveEntity(conta);  
    conta = repository.save(conta);  
    return conta;  
}
```

```
public void excluir(Conta conta){  
    validateSaveRemoveEntity(conta);  
    repository.delete(conta);  
}
```

```
public void validateSaveRemoveEntity(Conta conta){  
    if(conta.getId()==null){  
        throw new RuntimeException("id da conta nulo");  
    }  
}
```

```
public Conta atualizar(Conta conta){
    if(conta.getId()==null){
        throw new RuntimeException("id da conta nulo");
    }
    conta = repository.save(conta);
    return conta;
}

public void excluir(Conta conta){
    if(conta.getId()==null){
        throw new RuntimeException("id da conta nulo");
    }
    repository.delete(conta);
}
```

```
public Conta atualizar(Conta conta){
    validateSaveRemoveEntity(conta);
    conta = repository.save(conta);
    return conta;
}
```

```
public void excluir(Conta conta){
    validateSaveRemoveEntity(conta);
    repository.delete(conta);
}
```

```
public void validateSaveRemoveEntity(Conta conta){
    if(conta.getId()==null){
        throw new RuntimeException("id da conta nulo");
    }
}
```


"extract method". Quando podemos aplicar essa técnica de refatoração?

Quando encontrarmos métodos com muitas linhas de código.

Quando encontrarmos código repetido numa mesma classe.


Atenção:

Entre todos os tipos de código orientado a objetos, as classes com métodos curtos vivem mais. Quanto mais longo for um método ou função, mais difícil se torna entendê-lo e mantê-lo.

Além disso, os métodos longos oferecem o esconderijo perfeito para códigos duplicados indesejados.

Um aumento no número de métodos prejudica o desempenho, como muitas pessoas afirmam? Em quase todos os casos, o impacto é tão insignificante que nem vale a pena se preocupar.

Além disso, agora que você tem um código claro e compreensível, é mais provável que encontre métodos realmente eficazes para reestruturar o código e obter ganhos reais de desempenho, se necessário.

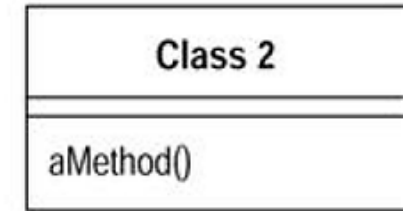
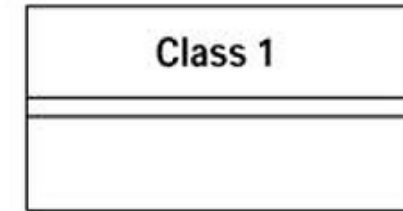
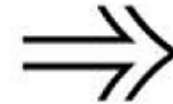
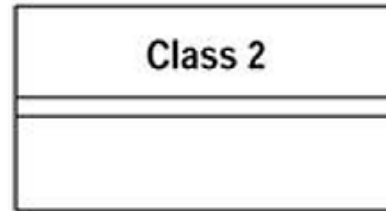
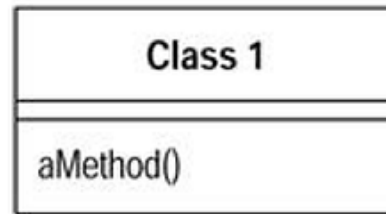


Mover Método (Move Method)

- Quando você possui um método que será utilizado muitas vezes por outra classe diferente da classe na qual ele foi definido, crie um método similar nesta classe que o está utilizando, depois copie o corpo do método copiado para o novo método, então substitua as chamadas para utilizarem o método local. Pode-se também avaliar se o método original deve ser preservado ou se pode ser eliminado.
- Na imagem abaixo vemos o método “***aMethod***” foi retirado da classe de origem “***Class1***” e movido para a classe “***Class2***”, onde ele faz mais sentido existir.



Mover
Método
(Move
Method)



Mover método



Problema

Um método é dividido em partes, cada uma das quais é executada dependendo do valor de um parâmetro.

Solução

Extraia as partes individuais do método em seus próprios métodos e chame-os em vez do método original.

```
void setValue(String name, int value) {  
    if (name.equals("height")) {  
        height = value;  
        return;  
    }  
    if (name.equals("width")) {  
        width = value;  
        return;  
    }  
    Assert.shouldNeverReachHere();  
}
```

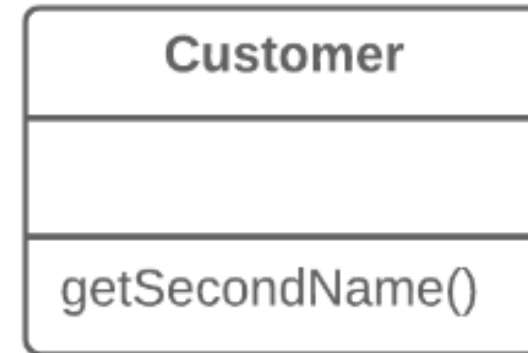
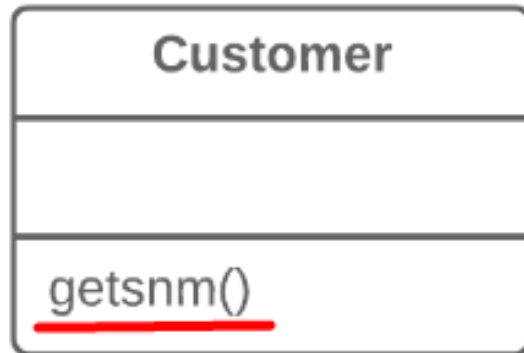
```
void setHeight(int arg) {  
    height = arg;  
}  
void setWidth(int arg) {  
    width = arg;  
}
```

Replace Parameter with Explicit Methods

- Um método contendo variantes dependentes de parâmetros cresceu muito. O código não trivial é executado em cada ramificação e novas variantes são adicionadas muito raramente.
- **Benefícios**
 - Melhora a legibilidade do código. É muito mais fácil entender o propósito de **startEngine()** do que **setValue("engineEnabled", true)**.
- **Quando não usar**
 - Não substitua um parâmetro por métodos explícitos se um método raramente for alterado e novas variantes não forem adicionadas a ele.

Por que refatorar

Refatoração : Renomear Método (Rename Method)

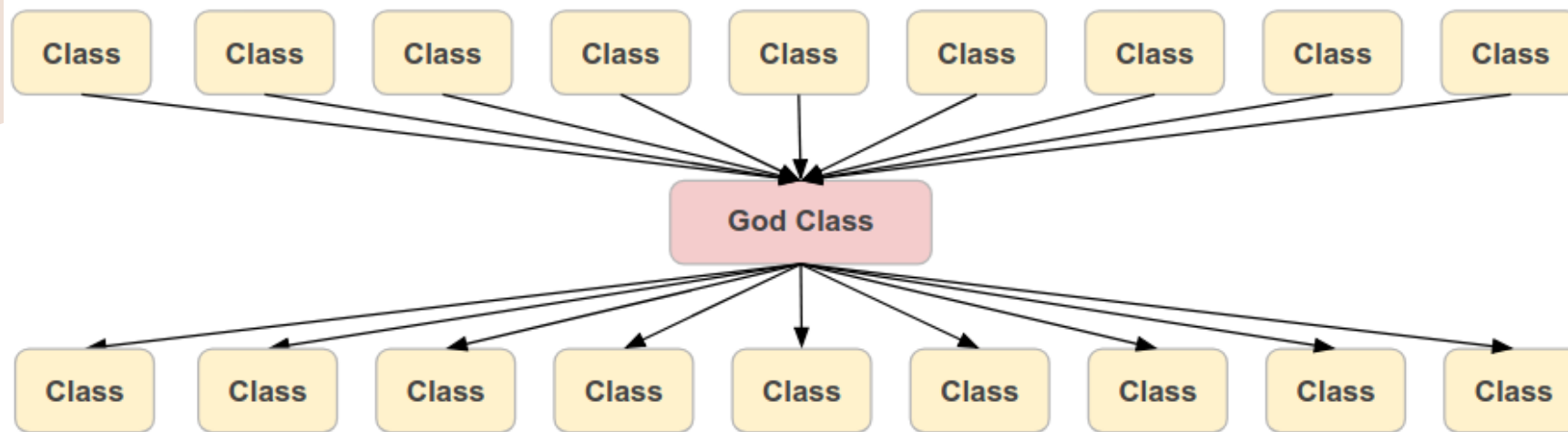


Por que refatorar

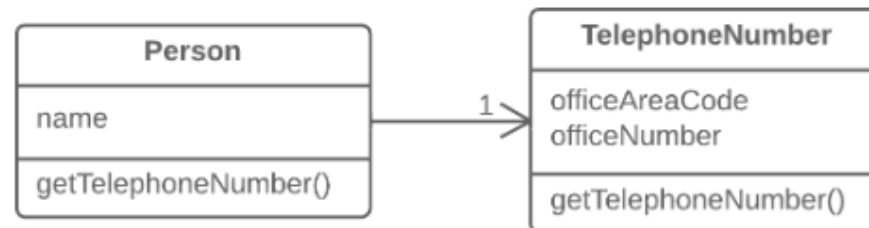
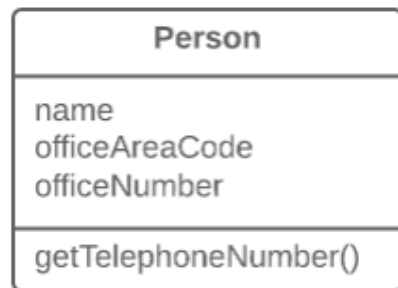
- Talvez um método tenha sido mal nomeado desde o início - por exemplo, alguém criou o método às pressas e não deu o devido cuidado para nomeá-lo bem.
- Ou talvez o método tenha sido bem nomeado no início, mas à medida que sua funcionalidade cresceu, o nome do método deixou de fazer sentido ou ficou confuso.
- Tente dar ao novo método um nome que reflita o que ele faz. Algo como `createOrder()`, `renderCustomerInfo()`, etc.

Como implementar

- Veja se o método está definido em uma superclasse ou subclasse. Se sim, você deve repetir todos os passos nestas classes também.
- O próximo método é importante para manter a funcionalidade do programa durante o processo de refatoração. Crie um novo método com um novo nome. Copie o código do método antigo para ele. Exclua todo o código do método antigo e, em vez dele, insira uma chamada para o novo método.
- Encontre todas as referências ao método antigo e substitua-as por referências ao novo.
- Exclua o método antigo. Se o método antigo fizer parte de uma interface pública, não execute esta etapa. Em vez disso, marque o método antigo como obsoleto.



Large class – God Class



- Quando você possui uma classe que está executando a tarefa de duas, então crie uma nova classe, depois mova todos os atributos relevantes para esta nova classe.

Extract Class

Benefícios

- Esse método de refatoração ajudará a manter a adesão ao Princípio da Responsabilidade Única . O código de suas classes será mais óbvio e compreensível.
- As classes de responsabilidade única são mais confiáveis e tolerantes a mudanças. Por exemplo, digamos que você tenha uma classe responsável por dez coisas diferentes. Quando você altera essa classe para torná-la melhor para uma coisa, corre o risco de quebrá-la para as outras nove.

Como implementar

Antes de começar, decida como exatamente você deseja dividir as responsabilidades da classe.

1 - Crie uma nova classe para conter a funcionalidade relevante.

2 - Crie um relacionamento entre a classe antiga e a nova. Idealmente, essa relação é unidirecional; isso permite reutilizar a segunda classe sem problemas. No entanto, se você acha que uma relação de mão dupla é necessária, isso sempre pode ser configurado.

3 - Use **Mover Campo e Mover Método** para cada campo e método que você decidiu mover para a nova classe. Para métodos, comece com os privados para reduzir o risco de cometer um grande número de erros. Tente mudar um pouco de cada vez e teste os resultados após cada movimento, a fim de evitar um acúmulo de correção de erros no final.

Depois que você terminar de se mover, dê mais uma olhada nas classes resultantes. Uma classe antiga com responsabilidades alteradas pode ser renomeada para maior clareza. Verifique novamente para ver se você pode se livrar das relações de classe de duas vias, se houver alguma.

4 - Pense também na acessibilidade externa à nova classe. Você pode ocultar a classe do cliente inteiramente tornando-a privada, gerenciando-a através dos campos da classe antiga. Alternativamente, você pode torná-lo público, permitindo que o cliente altere os valores diretamente. Sua decisão aqui depende de quão seguro é para o comportamento da classe antiga quando mudanças diretas inesperadas são feitas nos valores da nova classe.

Introduce Parameter Object

Problema

Seus métodos contêm um grupo repetido de parâmetros.

Solução

Substitua esses parâmetros por um objeto.

Customer
amountInvoicedIn (start : Date, end : Date) amountReceivedIn (start : Date, end : Date) amountOverdueIn (start : Date, end : Date)

Customer
amountInvoicedIn (date : DateRange) amountReceivedIn (date : DateRange) amountOverdueIn (date : DateRange)

- Grupos idênticos de parâmetros são frequentemente encontrados em vários métodos. Isso causa a duplicação de código dos próprios parâmetros e das operações relacionadas. Ao consolidar parâmetros em uma única classe, você também pode mover os métodos de manipulação desses dados para lá, liberando os outros métodos desse código.

Introduce Parameter Object

- **Benefícios**

- Código mais legível. Em vez de uma miscelânea de parâmetros, você vê um único objeto com um nome compreensível.
- Grupos idênticos de parâmetros espalhados aqui e ali criam seu próprio tipo de duplicação de código: enquanto um código idêntico não está sendo chamado, grupos idênticos de parâmetros e argumentos são constantemente encontrados.

- **Desvantagens**

- Se você mover apenas dados para uma nova classe e não planeja mover nenhum comportamento ou operação relacionada para lá, isso começa a cheirar a uma [Data Class](#)

Como implementar - Introduce Parameter Object

- Crie uma nova classe que representará seu grupo de parâmetros. Torne a classe imutável.
- No método que você deseja refatorar, use [Add Parameter](#) , que é para onde seu objeto de parâmetro será passado. Em todas as chamadas de métodos, passe o objeto criado a partir de parâmetros de métodos antigos para este parâmetro.
- Agora comece a deletar os parâmetros antigos do método um por um, substituindo-os no código por campos do objeto de parâmetro. Teste o programa após cada substituição de parâmetro.
- Quando terminar, veja se há algum ponto em mover uma parte do método (ou às vezes até mesmo o método inteiro) para uma classe de objeto de parâmetro. Em caso afirmativo, use [Move Method](#) ou [Extract Method](#) .

Quando devo refatorar um código?

Sempre que possível e fizer sentido.

Lembre-se que refatorar código que já está em produção requer um cuidado especial.

O Code review e análises e melhorias constantes ajudam a evitar a necessidade de refatorações complexas.



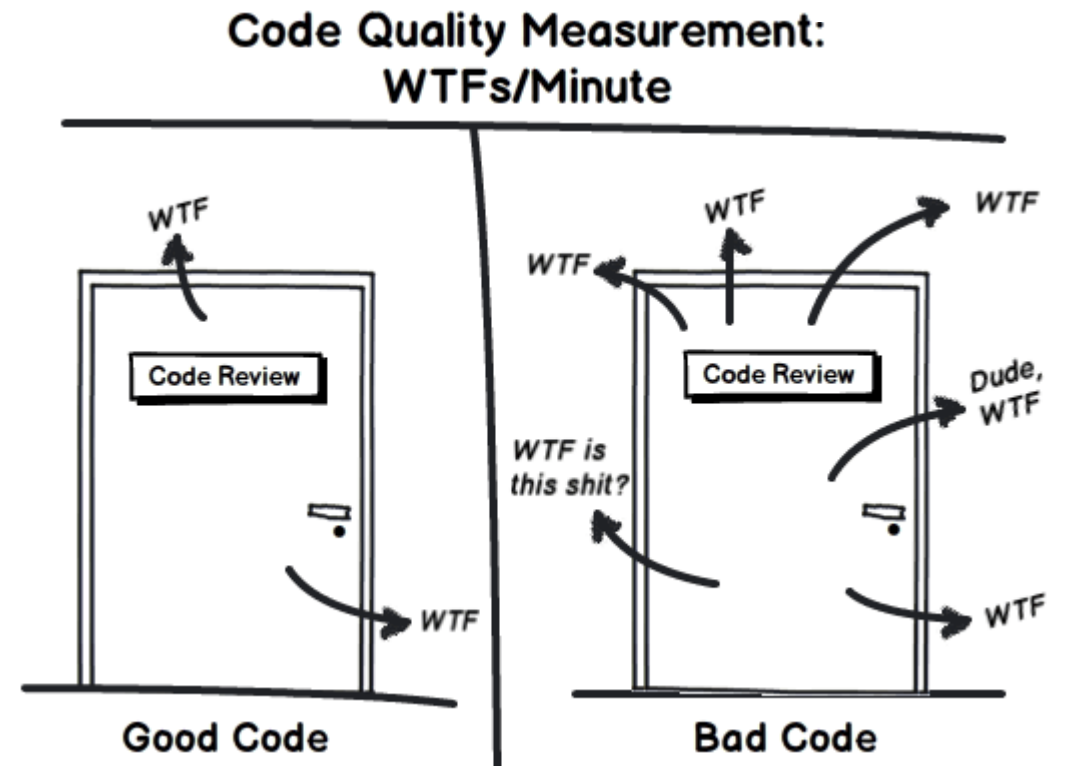


Pontos importantes

Code Review

- Revisão de código realizado por outro desenvolvedor que é realizado a cada PR ou commit.

Momento onde se analisa a utilização de boas práticas de desenvolvimento e orientações de melhoria no código ou outras formas de solução para trechos de código.



Importância do code review

- Distribuição do conhecimento
 - Outros desenvolvedores vão ter conhecimento sobre uma feature ou correção, permitindo que outros programadores além do criador possam trabalhar naquele projeto ou entender melhor a aplicação.
- **Ganho de produtividade**
- Ambiente de desenvolvimento mais **coletivo**.
- **Aprendizado** - desenvolvedores conseguem aprender a

```
public void validateSaveRemoveEntity(Conta conta){  
    if(conta.getId()==null){  
        throw new RuntimeException("id da conta nulo");  
    }  
}
```

```
public void validateSaveRemoveEntity(Conta conta){  
    if(Objects.isNull(conta.getId())){  
        throw new RuntimeException("id da conta nulo");  
    }  
}
```

Code Review Dicas de análise

- Sempre fazer sugestões sobre o código e não sobre a pessoa.
 - Humildade para receber as críticas e melhorar.
 - Eu sugiro que...", "Eu acho...", "Para mim, esse ponto.
 - Evite "Você está fazendo a implementação errada"
 - Tente:

Observação "Essa implementação é repetida em outro contexto, poderia ser reutilizada"

Impacto "Essa implementação torna a compreensão do real objetivo do método não tão claro para mim"

Request " Para esse cenário eu sugiro usar X padrão de projeto, por N motivos"Revisar o código utilizando um Checklist

- syntactic correctness
 - feature completeness
 - meaningful use of git
- Com o tempo isso termina sendo natural, para devs menos experientes, ter um checklist ajuda no que ele pode sugerir ou analisar a cada PR.

<https://nebulab.com/blog/a-guide-to-effective-pull-request-reviews/>

<https://smartbear.com/learn/code-review/best-practices-for-peer-code-review/>

Syntactic correctness

Essa é a primeira checagem: é fácil e rápido (dependendo da quantidade de código) e não requer um cérebro novo e, normalmente, pode até acontecer tarde do dia porque é só uma questão de checar:

- Os testes passam? A build esta verde?
- A análise de código estático relata problemas (ou seja, **Sonar** e amigos)?
- Aparência geral do código, há algum Code Smells?
- A descrição e as mensagens do commit estão escritas corretamente?
- O PR respeita todas as convenções do projeto (ou seja, nomeação de branches, ...)?

Geralmente, se algo está claramente errado, eu assumo que o PR não está pronto para ser revisado ou imediatamente peço uma correção nessas coisas simples antes de investigar o PR.

<https://docs.github.com/en/communities/using-templates-to-encourage-useful-issues-and-pull-requests/creating-a-pull-request-template-for-your-repository>

<https://community.atlassian.com/t5/Bitbucket-questions/Creating-a-template-for-pull-request/qaq->

feature completeness

- Em primeiro lugar, tente ler uma ou duas vezes o problema que o PR está tentando resolver (deveria estar vinculado na descrição do PR), depois revise as alterações de código.
- A ideia aqui é tentar entender se a pessoa que escreveu o código:
 - tinha uma boa compreensão do assunto (fez as perguntas certas, tinha as habilidades necessárias, ...).
 - escreveu o código que realmente resolveu o problema.
 - não perdeu nenhuma regra ou alterou algo que não deveria.
 - usou os padrões de design corretos (**e não exagerou na engenharia do código**)
- Todos os itens acima podem ser complicados porque geralmente você está vendo o resultado de várias horas de trabalho em um curto período de tempo (**às vezes você precisa desse código em produção e isso não ajudará**).

meaningful use of git

- Então, o código parece sólido, agora nos commits. Por que os commits são tão importantes?

Pense na história do git como um novelo de lã que precisa ser desembaraçado:

- você prefere que o fio seja enrolado ordenadamente...
- ou você prefere que ele seja montado aleatoriamente apenas para construir uma forma de bola?

meaningful use of git

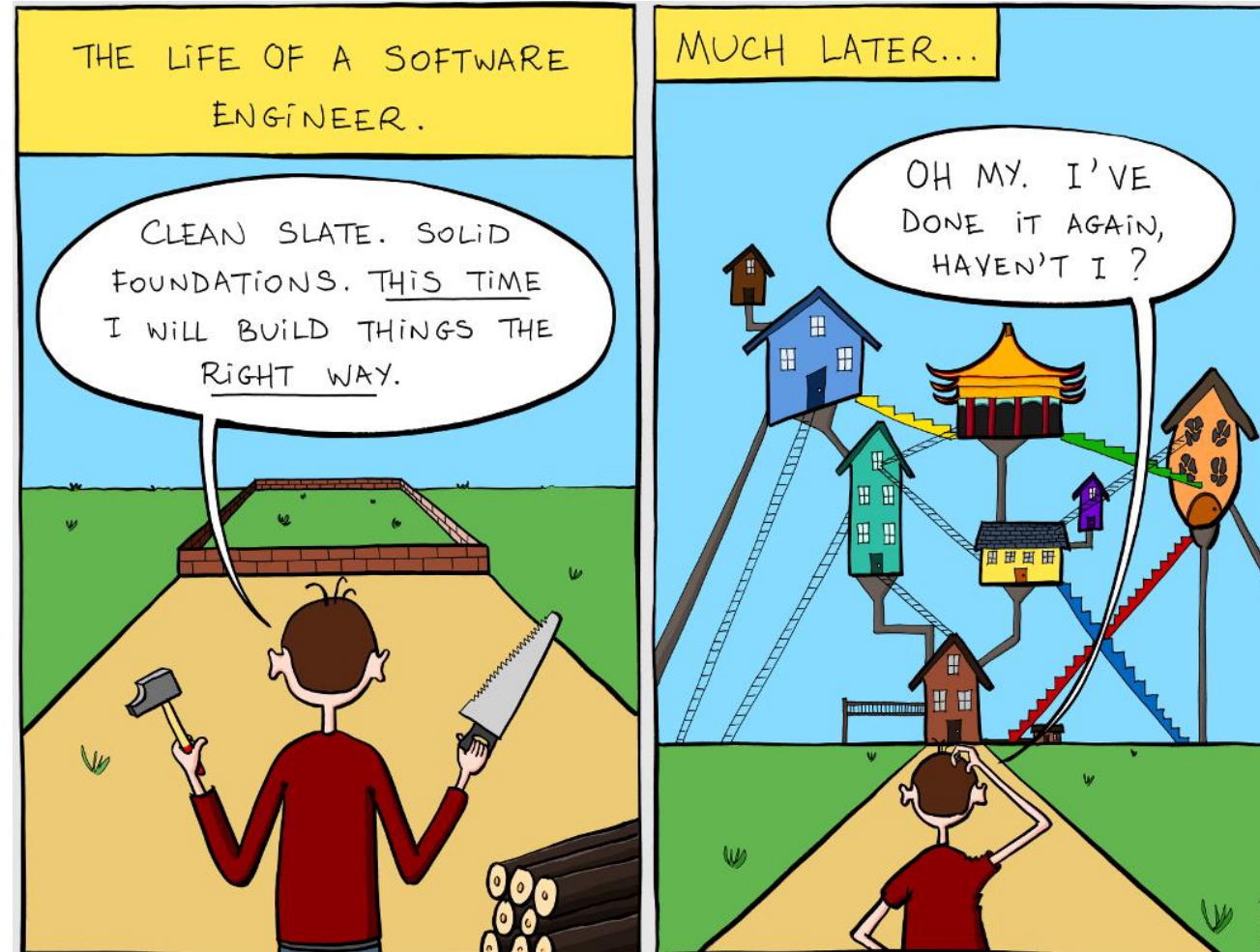
- Isso faz sentido para mim porque sempre usamos o desenvolvimento baseado em Branchs, então
 - O histórico de nossos projetos master é uma única linha de commits com cada commit se conectando (espero) ao seguinte, em um único fluxo de alterações.
- À medida que os recursos crescem e evoluem, o histórico do git representa linearmente o que aconteceu e por quê.

<https://cbea.ms/git-commit/>

	COMMENT	DATE
○	CREATED MAIN LOOP & TIMING CONTROL	14 HOURS AGO
○	ENABLED CONFIG FILE PARSING	9 HOURS AGO
○	MISC BUGFIXES	5 HOURS AGO
○	CODE ADDITIONS/EDITS	4 HOURS AGO
○	MORE CODE	4 HOURS AGO
○	HERE HAVE CODE	4 HOURS AGO
○	AAAAAAAAA	3 HOURS AGO
○	ADKFJSLKDFJSDKLFJ	3 HOURS AGO
○	MY HANDS ARE TYPING WORDS	2 HOURS AGO
○	HAAAAAAAAAANDS	2 HOURS AGO

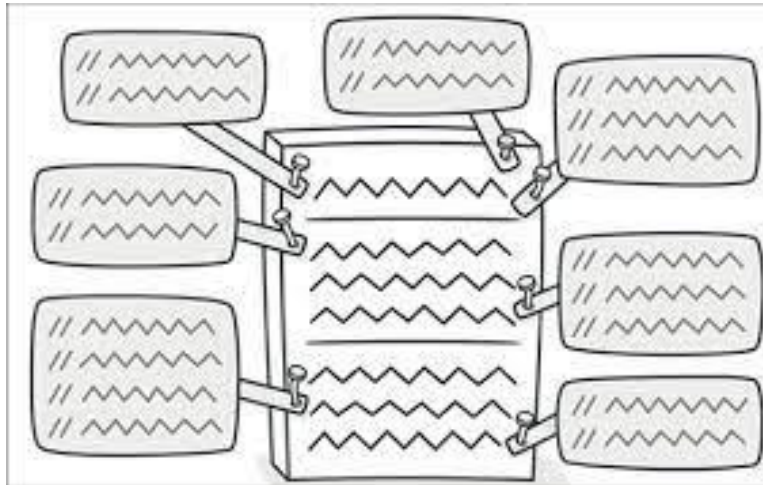
AS A PROJECT DRAGS ON, MY GIT COMMIT
MESSAGES GET LESS AND LESS INFORMATIVE.

Code review reduz a probabilidade



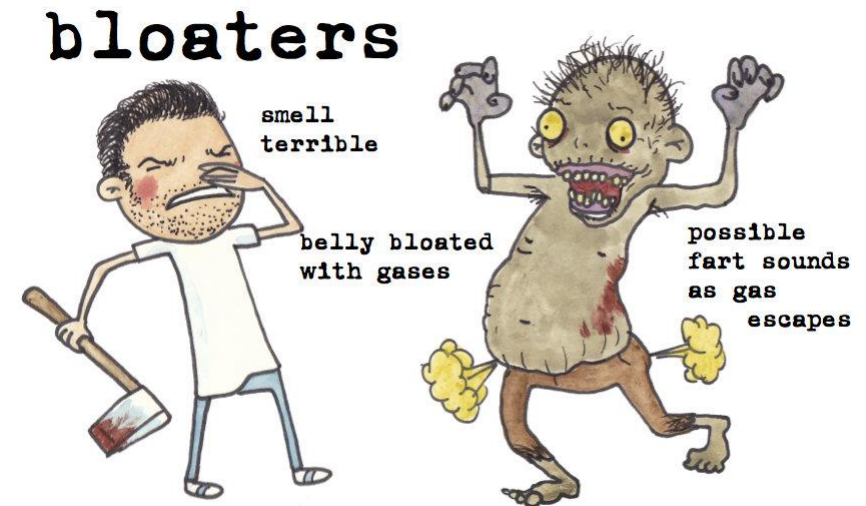
Code Smells

- Quando um projeto esta com muito code smells, é um sinal de que a refatoração é necessária.



Bloters

- São códigos, métodos e classes que aumentaram ao longo do tempo. Esse é o tipo de coisa que vai aparecendo ao longo do tempo, que é mais comum em projetos legados. Pois é algo que vai sendo acumulado ao longo do tempo.



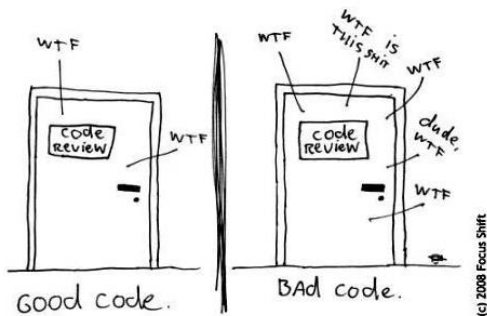
Long Method

- São métodos que contêm muitas linhas de código, quando você observa métodos com mais de 10 linhas é um sinal para reflexão ou atenção.

Exemplo


- Um método que é bastante reutilizado em um contexto.
- Então do desenvolvedor pensa, nossa são apenas mais 3 linhas adicionadas ao que já existe, para que criar um método novo.

The only valid measurement
of code quality: WTFs/minute



"Mas são apenas três ou quatro linhas, não adianta criar um método inteiro só para isso..."

Se você não entende algo dentro do código ou existe a necessidade de comentar para entender você deve colocar ele em um novo método.
(Lembre-se do Clean Code)



```
public Conta atualizar(Conta conta){
    if(conta.getId()==null){
        throw new RuntimeException("id da conta nulo");
    }
    conta = repository.save(conta);
    return conta;
}

public void excluir(Conta conta){
    if(conta.getId()==null){
        throw new RuntimeException("id da conta nulo");
    }
    repository.delete(conta);
}
```

```

29 +   }
30 +
22 31     public Filme atualizar(Filme filme){
23 32         if(filme.getId()==null){

```

Write

Preview



Colocar o teste que verifica se o ID é nulo em um método separado.
 Vai deixar o código mais legível e vai reaproveitar código
 Você tem um fragmento de código que pode ser agrupado e reaproveitado

Attach files by dragging & dropping, selecting or pasting them.



Cancel

Add single comment

Start a review

```

24 -         throw new RuntimeException("ID Nulo");
33 +         throw new RuntimeException("id da conta nulo");
25 34     }
26 35     filme = repository.save(filme);
27 36     return filme;
28 37 }
29 38
30 39     public void excluir(Filme filme){
31 40         if(filme.getId()==null){
32 -         throw new RuntimeException("ID Nulo");
41 +         throw new RuntimeException("id da conta nulo");
33 42     }
34 43     repository.delete(filme);
35 44 }
36 45

```

```
public Conta atualizar(Conta conta){
    validateSaveRemoveEntity(conta);
    conta = repository.save(conta);
    return conta;
}

public void excluir(Conta conta){
    validateSaveRemoveEntity(conta);
    repository.delete(conta);
}

public void validateSaveRemoveEntity(Conta conta){
    if(conta.getId()==null){
        throw new RuntimeException("id da conta nulo");
    }
}
```



```
public Conta atualizar(Conta conta){
    if(conta.getId()==null){
        throw new RuntimeException("id da conta nulo");
    }
    conta = repository.save(conta);
    return conta;
}

public void excluir(Conta conta){
    if(conta.getId()==null){
        throw new RuntimeException("id da conta nulo");
    }
    repository.delete(conta);
}
```

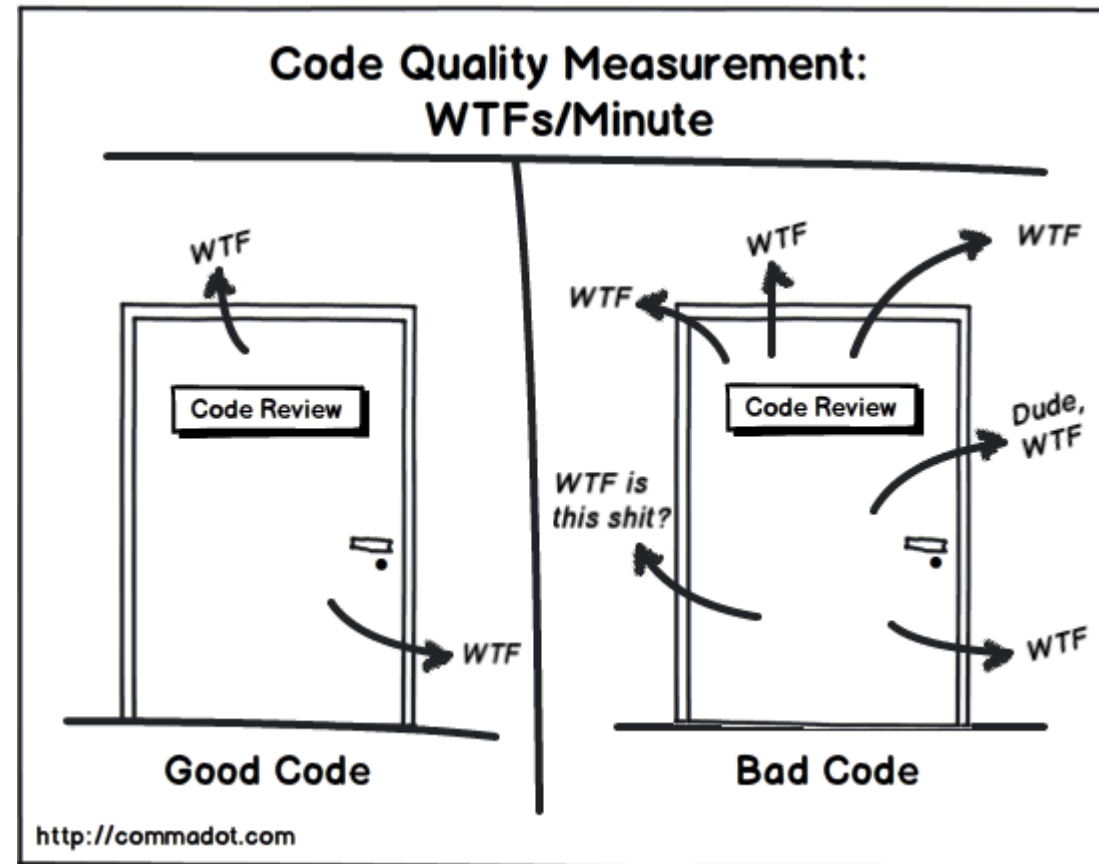
```
public Conta atualizar(Conta conta){
    validateSaveRemoveEntity(conta);
    conta = repository.save(conta);
    return conta;
}
```

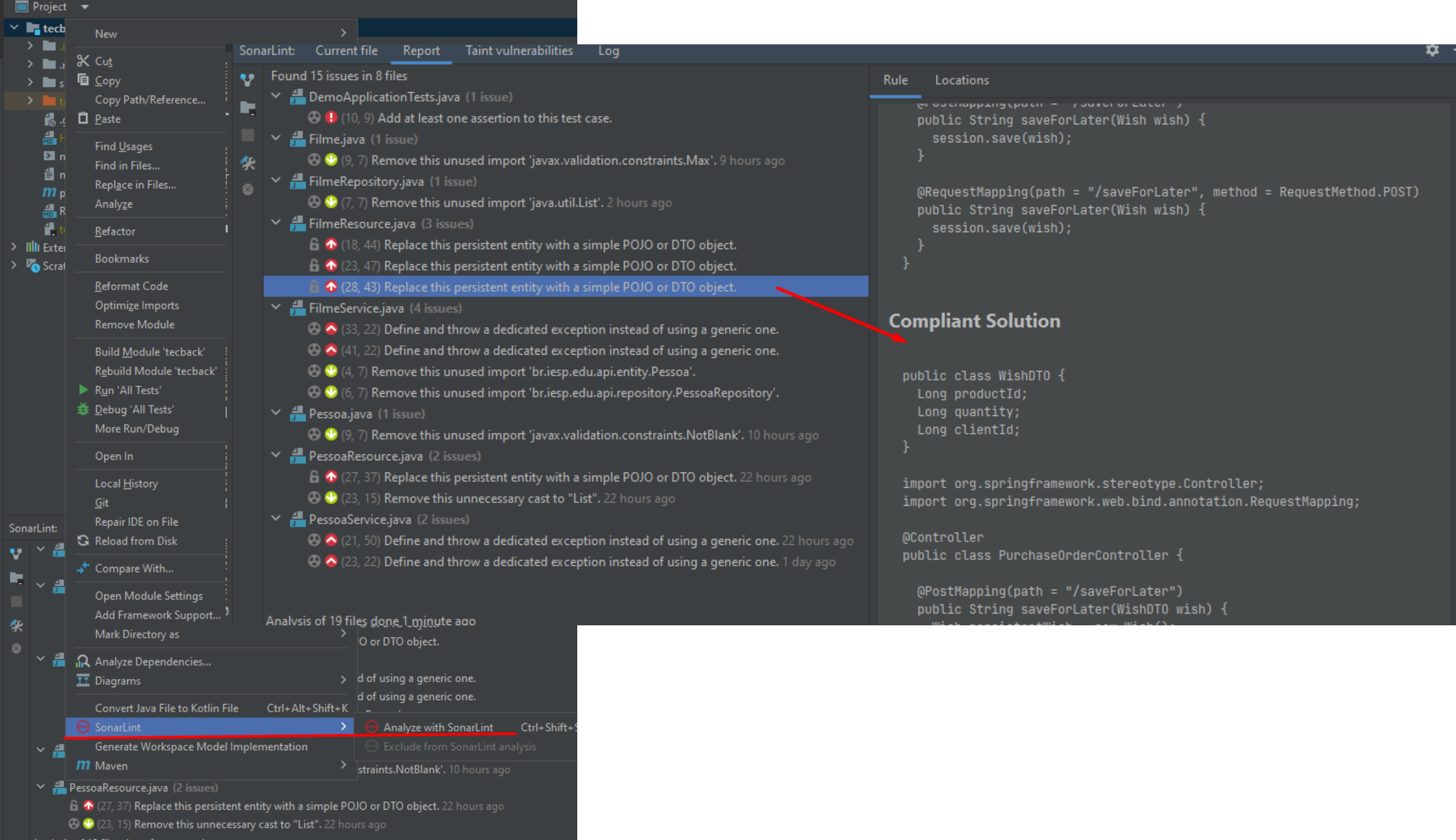
```
public void excluir(Conta conta){
    validateSaveRemoveEntity(conta);
    repository.delete(conta);
}
```

```
public void validateSaveRemoveEntity(Conta conta){
    if(conta.getId()==null){
        throw new RuntimeException("id da conta nulo");
    }
}
```

Code review é algo cultural

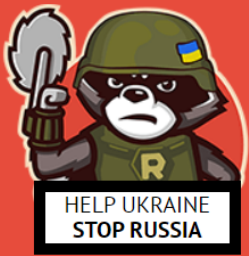
- P.Os
 - Scrum Masters
 - Tech leaders
 - Developers
- Engajem seus times com essa cultura, vai ajudar muito o time em termos de produtividade. O tempo que se ~~perde~~ investe em code review se ganha no dia a dia.





Perguntas

English Español Français 日本語
한국어 Polski Русский Українська
中文 Português-Br



REFACTORING
• GURU •

★ Premium Content

⌘ Refactoring

⌘ Design Patterns

Facebook Twitter


Hello, world!

Refactoring.Guru makes it easy for you to discover everything you need to know about refactoring, design patterns, SOLID principles, and other smart programming topics.

This site shows you the big picture, how all these subjects intersect, work together, and are still relevant. I don't pretend to be the inventor of these concepts—most of them were invented by others during the past 20 years. But I think that the connection between refactoring, patterns and general programming principles still remains a mystery for the majority of programmers. This is the problem I would like to solve here.

P.S. While I update the project constantly, you can already find tons of info on refactoring and design patterns right here on the website. Track the project progress via [email](#) or on [Facebook](#).

✉ Subscribe for updates

 Facebook page

— Alexander Shvets
The one-man band behind Refactoring.Guru

SOLID

- Compreender os princípios do SOLID e sua importância no desenvolvimento de software.
- Explorar exemplos de código em Java e Spring Boot para aplicar os princípios do SOLID.
- Capacitar os alunos a escreverem código mais robusto, extensível e de fácil manutenção.

Solid



O SOLID é um acrônimo que representa cinco princípios fundamentais da programação orientada a objetos e design de software.



Foi introduzido por Robert C. Martin, também conhecido como Uncle Bob.



Esses princípios são guias que ajudam os desenvolvedores a criar sistemas mais sólidos e resilientes, facilitando a manutenção e a escalabilidade do código.

2. Os cinco princípios SOLID:

- Vamos agora explicar cada um dos cinco princípios do SOLID:

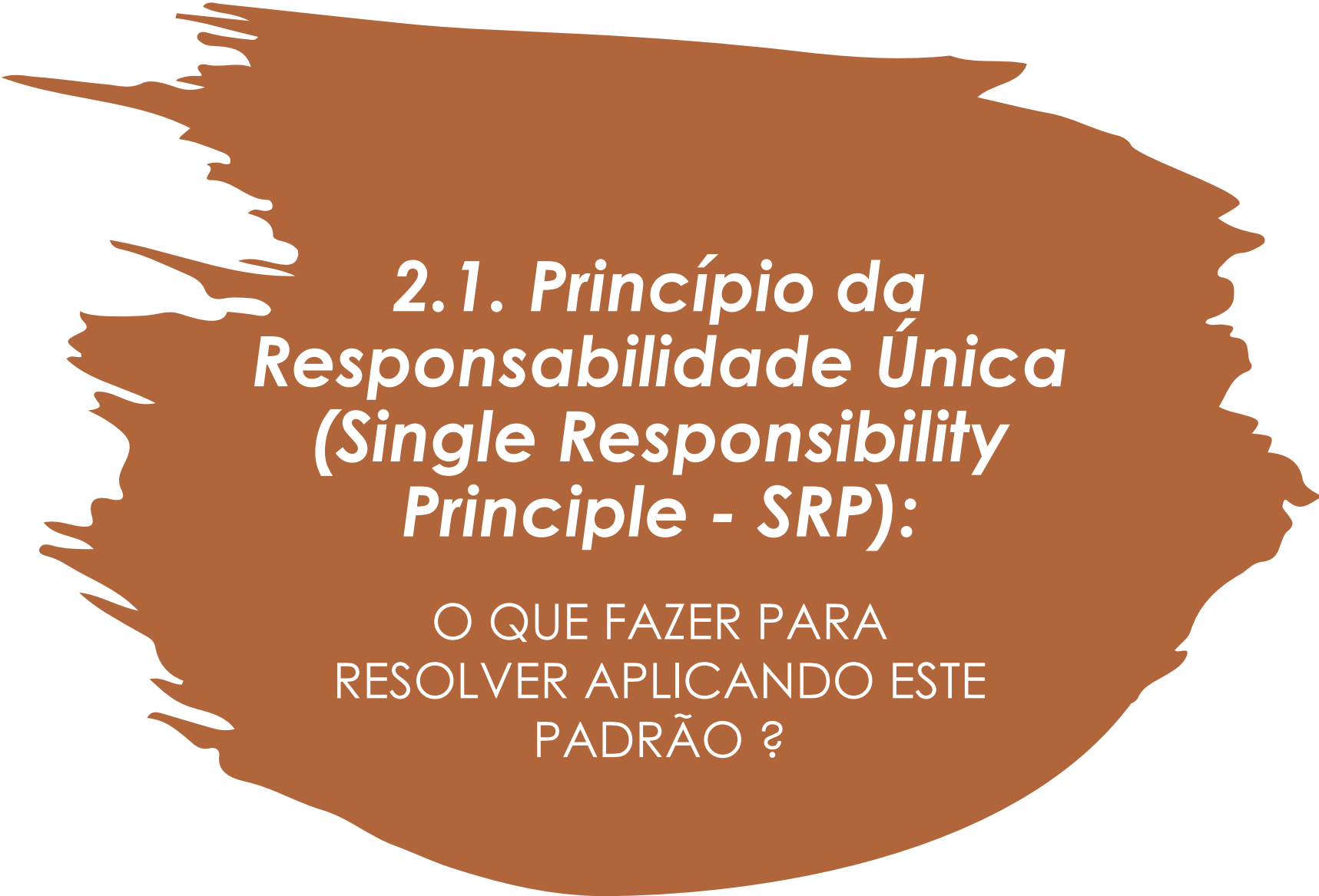
2.1. Princípio da Responsabilidade Única (Single Responsibility Principle - SRP):

- Esse princípio afirma que uma classe deve ter uma única responsabilidade, ou seja, ela deve ter apenas um motivo para mudar. Evitar classes que fazem muitas coisas diferentes é fundamental para facilitar a manutenção do código e tornar o sistema mais compreensível.

2.1. Princípio da Responsabilidade Única (Single Responsibility Principle - SRP):

- Neste exemplo, temos uma classe chamada Usuario que representa um usuário em um sistema. Porém, essa classe está lidando com várias responsabilidades ao mesmo tempo, o que viola o SRP.

```
public class Usuario {  
    private String nome;  
    private String email;  
    private String senha;  
  
    // Métodos de acesso e modificação para os atributos  
  
    public void salvarUsuario() {  
        // Lógica para salvar o usuário no banco de dados  
    }  
  
    public void enviarEmailDeBoasVindas() {  
        // Lógica para enviar o e-mail de boas-vindas ao usuário  
    }  
  
    public void gerarRelatorioDeAtividades() {  
        // Lógica para gerar um relatório de atividades do usuário  
    }  
}
```

A large, irregular brown brushstroke shape serves as a background for the text.

2.1. Princípio da Responsabilidade Única (Single Responsibility Principle - SRP):

O QUE FAZER PARA
RESOLVER APLICANDO ESTE
PADRÃO ?

Resolução

Agora, temos uma classe `Usuario` que representa apenas um usuário, e as responsabilidades de salvar no banco de dados, enviar e-mails e gerar relatórios foram separadas em classes distintas, cada uma com sua função específica. Isso torna o código mais modular, facilitando a manutenção e o entendimento das funcionalidades.

Além disso, caso uma dessas responsabilidades precise mudar, isso não afetará as outras partes do código, tornando-o mais resiliente a mudanças.

```
public class Usuario {  
    private String nome;  
    private String email;  
    private String senha;  
  
    // Métodos de acesso e modificação para os atributos  
}  
  
public class UsuarioDAO {  
    public void salvarUsuario(Usuario usuario) {  
        // Lógica para salvar o usuário no banco de dados  
    }  
}  
  
public class EmailService {  
    public void enviarEmailDeBoasVindas(Usuario usuario) {  
        // Lógica para enviar o e-mail de boas-vindas ao usuário  
    }  
}  
  
public class RelatorioService {  
    public void gerarRelatorioDeAtividades(Usuario usuario) {  
        // Lógica para gerar um relatório de atividades do usuário  
    }  
}
```

2.1. Princípio da Responsabilidade Única (Single Responsibility Principle - SRP):

Neste exemplo, a classe **Usuario** possui atributos e métodos relacionados ao próprio usuário, mas também lida com a persistência no banco de dados e envio de e-mails.

Isso torna a classe menos coesa e mais difícil de manter, especialmente quando cada responsabilidade precisa ser modificada ou evoluir independentemente.

2.2. Princípio do Aberto/Fechado (Open/Closed Principle - OCP):

- O Princípio do Aberto/Fechado preconiza que uma classe deve ser aberta para extensão, mas fechada para modificação. Isso significa que você pode adicionar novos recursos ou funcionalidades através de herança, composição ou interfaces, sem precisar modificar o código-fonte da classe existente.
- Dessa forma, você evita introduzir bugs em código funcional e mantém a estabilidade do sistema.

2.2. Princípio do Aberto/Fechado (Open/Closed Principle - OCP):

- Suponha que temos uma classe Forma que representa uma forma geométrica.
- Inicialmente, temos apenas duas subclasses Quadrado e Circulo que representam um quadrado e um círculo, respectivamente

```
public abstract class Forma {  
    public abstract void desenhar();  
}  
  
public class Quadrado extends Forma {  
    @Override  
    public void desenhar() {  
        System.out.println("Desenhando um quadrado");  
    }  
}  
  
public class Circulo extends Forma {  
    @Override  
    public void desenhar() {  
        System.out.println("Desenhando um círculo");  
    }  
}
```


- Agora, suponha que desejamos adicionar novas funcionalidades, como calcular a área das formas. A forma ideal de fazer isso é através da extensão, sem modificar o código existente.
- Porém, se adicionarmos o método **calcularArea()** diretamente na classe Forma, estaríamos violando o OCP, pois estaríamos modificando a classe existente para adicionar funcionalidades.

Exemplo - Aplicando o OCP:

- Para obedecer ao Princípio do Aberto/Fechado, criaremos uma interface chamada **AreaCalculavel** que será implementada pelas classes que precisam calcular a área.
- Em seguida, criaremos novas classes que estendem Forma e também implementam **AreaCalculavel**, adicionando assim a nova funcionalidade sem modificar o código existente.

```
public interface AreaCalculavel {  
    double calcularArea();  
}  
  
public abstract class Forma {  
    public abstract void desenhar();  
}  
  
public class Quadrado extends Forma {  
    @Override  
    public void desenhar() {  
        System.out.println("Desenhando um quadrado");  
    }  
}  
  
public class Circulo extends Forma {  
    @Override  
    public void desenhar() {  
        System.out.println("Desenhando um círculo");  
    }  
}
```

```
public class Retangulo extends Forma implements AreaCalculavel {  
    private double largura;  
    private double altura;  
  
    public Retangulo(double largura, double altura) {  
        this.largura = largura;  
        this.altura = altura;  
    }  
  
    @Override  
    public void desenhar() {  
        System.out.println("Desenhando um retângulo");  
    }  
  
    @Override  
    public double calcularArea() {  
        return largura * altura;  
    }  
}
```

- 
- Agora, a classe **Forma** e suas subclasses estão fechadas para modificação, pois não precisamos alterá-las para adicionar a funcionalidade de calcular a área. Em vez disso, criamos uma nova classe **Retangulo** que estende **Forma** e implementa **AreaCalculavel**, adicionando a funcionalidade de cálculo de área sem afetar o código existente.

2.3. Princípio da Substituição de Liskov (Liskov Substitution Principle - LSP):

- Estabelece que objetos de uma superclasse devem ser substituíveis por objetos de suas subclasses sem alterar a corretude do programa. Em outras palavras, uma subclasse deve ser capaz de substituir a classe base sem afetar o comportamento esperado do programa.

Vamos ver um exemplo de código em Java para ilustrar o LSP:

Exemplo - Violando o Liskov Substitution Principle:

- Neste exemplo, temos uma classe **Retangulo** que representa um retângulo com atributos **largura** e **altura**. Em seguida, temos uma classe **Quadrado** que herda de **Retangulo**. A classe **Quadrado** possui um construtor que recebe apenas um lado, e internamente, tanto a largura quanto a altura são definidas com o mesmo valor (já que o quadrado tem lados iguais).

O problema nesse código ocorre quando tentamos tratar um objeto Quadrado como um objeto **Retangulo**. Se tentarmos modificar a **largura** ou a **altura** de um **quadrado** após sua criação, a inconsistência do LSP se torna evidente:

```
1  public class Retangulo {
2      protected double largura;
3      protected double altura;
4
5      public Retangulo(double largura, double altura) {
6          this.largura = largura;
7          this.altura = altura;
8      }
9
10     public double getArea() {
11         return largura * altura;
12     }
13 }
14
15 public class Quadrado extends Retangulo {
16     public Quadrado(double lado) {
17         super(lado, lado); // O quadrado possui lados iguais,
18                             //entao largura = altura = lado
19     }
20 }
```

```
1 public static void main(String[] args) {  
2     Retangulo retangulo = new Quadrado(5);  
3     System.out.println("Area: " + retangulo.getArea());  
4     // Area: 25  
5  
6     // Tentando modificar a largura do quadrado  
7     retangulo.setLargura(10);  
8     System.out.println("Area: " + retangulo.getArea());  
9     // Area: 50 (esperado 100)  
10 }
```

Neste exemplo, ao tentarmos modificar a largura de um objeto Quadrado, também estamos alterando a altura, o que não deveria acontecer, pois o quadrado possui lados iguais. Portanto, essa **violação do LSP** indica que a herança de Quadrado de Retangulo não foi bem modelada, pois não há uma substituição adequada entre os objetos.

Exemplo - Aplicando o Liskov Substitution Principle:

Para aplicar corretamente o LSP, a relação entre **Quadrado** e **Retangulo** não deve ser de herança, pois a relação "é um" entre eles não é apropriada. Uma solução mais adequada é usar composição ou outra abordagem para tratar a relação entre as classes.

Por exemplo, podemos ter uma interface **Forma** que define um método para calcular a área e, em seguida, implementar **Retangulo** e **Quadrado** de maneira independente:

```
1  interface Forma {  
2      |   double getArea();  
3  }  
4
```

```

5  class Retangulo implements Forma {
6      protected double largura;
7      protected double altura;
8
9      public Retangulo(double largura, double altura) {
10         this.largura = largura;
11         this.altura = altura;
12     }
13
14     @Override
15     public double getArea() {
16         return largura * altura;
17     }
18
19     // Getters e setters para largura e altura
20     // (nao fazem parte do LSP, mas apenas para acesso)
21     public double getLargura() {
22         return largura;
23     }
24
25     public void setLargura(double largura) {
26         this.largura = largura;
27     }
28
29     public double getAltura() {
30         return altura;
31     }
32
33     public void setAltura(double altura) {
34         this.altura = altura;
35     }
36 }

```

Princípio da Substituição de Liskov

- Agora, não temos mais a relação de herança entre Quadrado e Retangulo, mas ambas implementam a interface Forma, garantindo que sejam substituíveis em qualquer contexto que espere uma forma geométrica. Dessa forma, respeitamos o Liskov Substitution Principle

```

38 class Quadrado implements Forma {
39     private double lado;
40
41     public Quadrado(double lado) {
42         this.lado = lado;
43     }
44
45     @Override
46     public double getArea() {
47         return lado * lado;
48     }
49
50     // Getter e setter para o lado
51     // (nao fazem parte do LSP, mas apenas para acesso)
52     public double getLado() {
53         return lado;
54     }
55
56     public void setLado(double lado) {
57         this.lado = lado;
58     }
59 }

```

2.4. Princípio da Segregação de Interfaces (Interface Segregation Principle - ISP):

- O Princípio da Segregação de Interfaces declara que uma classe não deve ser forçada a depender de interfaces que ela não utiliza.
- Em vez de criar interfaces monolíticas, é preferível ter interfaces mais específicas para cada contexto de uso, evitando assim que as classes implementem métodos desnecessários.

Exemplo não aplicando o ISP:

- Suponha que temos uma interface chamada `Trabalhador` em um aplicativo Spring Boot para gerenciar funcionários de uma empresa

```
public interface Trabalhador {  
    void trabalhar();  
    void comer();  
    void dormir();  
}
```

Em seguida, temos uma classe chamada `Engenheiro` que implementa a interface `Trabalhador`. Porém, um engenheiro não precisa dos métodos `comer()` e `dormir()`, pois essas ações não estão relacionadas à sua função:

Nesse caso, a classe `Engenheiro` está sendo forçada a implementar métodos que não são relevantes para sua função, o que viola o ISP. Isso pode levar a um código inchado e confuso, onde as classes precisam implementar funcionalidades que não são pertinentes

```
public class Engenheiro implements Trabalhador {  
    @Override  
    public void trabalhar() {  
        System.out.println("Engenheiro trabalhando...");  
        // Lógica específica do trabalho de um engenheiro  
    }  
  
    @Override  
    public void comer() {  
        System.out.println("Engenheiro comendo...");  
        // Lógica de alimentação do engenheiro (desnecessária)  
    }  
  
    @Override  
    public void dormir() {  
        System.out.println("Engenheiro dormindo...");  
        // Lógica de sono do engenheiro (desnecessária)  
    }  
}
```


Exemplo aplicando o ISP:

- Para aplicar o ISP corretamente, devemos dividir a interface Trabalhador em interfaces menores e mais específicas, de modo que as classes só implementem as interfaces relevantes para suas funcionalidades.

```
public interface Trabalhador {  
    void trabalhar();  
}  
  
public interface Comedor {  
    void comer();  
}  
  
public interface Dorminhoco {  
    void dormir();  
}
```

Agora, podemos ajustar a classe **Engenheiro** para implementar apenas a interface **Trabalhador**, sem a necessidade de implementar os métodos **comer()** e **dormir()**.

```
public class Engenheiro implements Trabalhador {  
    @Override  
    public void trabalhar() {  
        System.out.println("Engenheiro trabalhando...");  
        // Lógica específica do trabalho de um engenheiro  
    }  
}
```

Com a segregação das interfaces, evitamos a dependência desnecessária da classe **Engenheiro** em métodos que não são relevantes para ela, deixando o código mais limpo e coeso. Em um contexto do Spring Boot, a aplicação do ISP pode ser feita através da injeção de dependências mais específicas para cada componente ou serviço.

Isso permitirá que cada classe dependa apenas das interfaces que realmente precisa, promovendo a separação de responsabilidades e tornando o código mais modular e fácil de entender.

2.5. Princípio da Inversão de Dependência (Dependency Inversion Principle - DIP):

- **O Princípio da Inversão de Dependência sugere que as classes de alto nível não devem depender das classes de baixo nível diretamente, mas sim de abstrações.**
- **Além disso, as abstrações não devem depender dos detalhes, mas os detalhes devem depender das abstrações. Isso permite uma maior flexibilidade, facilitando a troca de implementações e a realização de testes unitários.**

Exemplo aplicando o DIP:

- Vamos criar um exemplo de um serviço que envia mensagens por meio de diferentes canais de comunicação: e-mail e SMS. O DIP será aplicado usando interfaces para abstrair os detalhes de implementação dos canais de comunicação.

```
// Interface abstrata para o serviço de mensagem
public interface MensagemService {
    void enviarMensagem(String mensagem);
}


// Implementação do serviço de envio de e-mails
public class EmailService implements MensagemService {
    @Override
    public void enviarMensagem(String mensagem) {
        // Lógica para enviar e-mail
        System.out.println("Enviando e-mail: " + mensagem);
    }
}

// Implementação do serviço de envio de SMS
public class SMSService implements MensagemService {
    @Override
    public void enviarMensagem(String mensagem) {
        // Lógica para enviar SMS
        System.out.println("Enviando SMS: " + mensagem);
    }
}
```

```
// Classe de alto nível que utiliza o serviço de mensagem
public class Notificador {
    private MensagemService mensagemService;

    public Notificador(MensagemService mensagemService) {
        this.mensagemService = mensagemService;
    }

    public void notificarCliente(String mensagem) {
        mensagemService.enviarMensagem(mensagem);
    }
}
```

- 
- Neste exemplo, a classe Notificador é uma classe de alto nível que depende de uma abstração MensagemService. As implementações concretas, como EmailService e SMSService, são injetadas na classe Notificador através do construtor.
 - Isso permite que a classe Notificador seja independente das implementações concretas dos serviços de mensagem, seguindo assim o Princípio da Inversão de Dependência.

Exemplo violando o DIP:

- Neste exemplo, a classe Notificador depende diretamente das classes concretas EmailService e SMSService.
- Isso viola o DIP, pois as classes de alto nível não deveriam depender diretamente das classes de baixo nível. Isso torna a classe Notificador mais acoplada e menos flexível para mudanças, além de dificultar a realização de testes unitários isolados.

```
// Classe de alto nível que depende de classes de baixo nível
public class Notificador {
    private EmailService emailService;
    private SMSService smsService;

    public Notificador() {
        this.emailService = new EmailService();
        this.smsService = new SMSService();
    }

    public void notificarCliente(String mensagem) {
        emailService.enviarEmail(mensagem);
        smsService.enviarSMS(mensagem);
    }
}
```

```
// Implementação concreta do serviço de e-mails
public class EmailService {
    public void enviarEmail(String mensagem) {
        // Lógica para enviar e-mail
        System.out.println("Enviando e-mail: " + mensagem);
    }
}

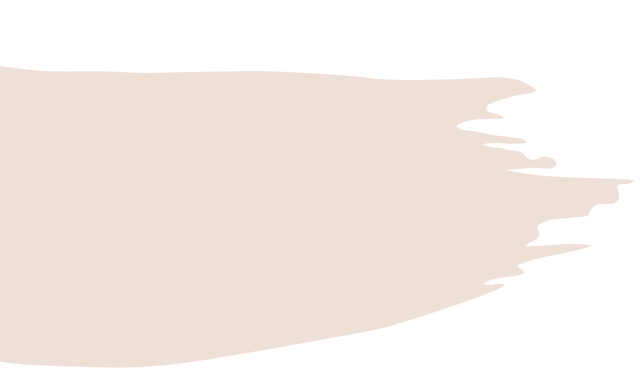
// Implementação concreta do serviço de SMS
public class SMSService {
    public void enviarSMS(String mensagem) {
        // Lógica para enviar SMS
        System.out.println("Enviando SMS: " + mensagem);
    }
}
```

3. Benefícios do SOLID no desenvolvimento de software:

- A aplicação dos princípios SOLID traz diversos benefícios ao desenvolvimento de software:
- **Manutenção facilitada:** Classes bem projetadas com responsabilidades únicas são mais fáceis de manter, pois as alterações têm menos chance de afetar outras partes do sistema.
- **Flexibilidade e extensibilidade:** O Princípio do Aberto/Fechado e o Princípio da Inversão de Dependência possibilitam adicionar novos recursos sem modificar o código existente.

3. Benefícios do SOLID no desenvolvimento de software:

- **Baixo acoplamento:** Ao seguir os princípios LSP e ISP, evita-se o acoplamento excessivo entre as classes, tornando o código mais modular e isolado.
- **Facilita a escrita de testes unitários:** Classes bem divididas e de baixo acoplamento são mais fáceis de testar, permitindo uma cobertura de testes mais completa.
- **Maior compreensão do código:** Ao seguir o SRP, o código se torna mais legível e compreensível, facilitando o trabalho colaborativo entre os desenvolvedores.
- **Redução de erros e bugs:** Com um código mais organizado e estruturado, a probabilidade de erros e bugs é reduzida.

- 
- Em resumo, aplicar os princípios SOLID não apenas melhora a qualidade do código, mas também ajuda a criar sistemas mais robustos, flexíveis e fáceis de manter ao longo do tempo.

Conclusão:

- Nesta aula, aprendemos os cinco princípios do SOLID (SRP, OCP, LSP, ISP e DIP) e compreendemos a importância de aplicá-los no desenvolvimento de software.
- Esses princípios fornecem orientações valiosas para projetar sistemas mais sólidos, flexíveis e de fácil manutenção. Ao aplicar o SOLID em nossos projetos, estamos investindo em um código de melhor qualidade e maior durabilidade, o que é essencial para o sucesso a longo prazo de qualquer projeto de desenvolvimento de software.