Universidade de Brasília - UnB
Faculdade UnB Gama - FGA
Engenharia Software

Técnicas de Programação para Plataformas Emergentes

Abner Filipe Cunha - 190041871
Antônio Ferreira De Castro - 190044799
Denys Rogeres Leles - 180041592
João Gabriel de Campos- 180042238
Sávio Cunha de Carvalho - 180130889
Yan Andrade de Sena - 180145363

Brasília, DF
2023

1. Simplicidade
1.1 Descrição

A simplicidade é uma característica fundamental na elaboração de código, pois ela permite que o código seja facilmente compreendido e modificado. Os efeitos de um design de código simples afetam diretamente um projeto de software possibilitando diversas vantagens, tais como:

- Facilidade de manutenção: um código simples é mais fácil de entender e modificar, o que diminui o tempo e o esforço necessário para realizar manutenções e atualizações.
- Colaboração mais fácil: um código simples é mais fácil de ser compreendido por outros desenvolvedores, o que facilita a colaboração em equipe e a transferência de conhecimento.
- Menor probabilidade de erros: um código simples é menos propenso a erros, pois é mais fácil de entender e testar.
- Maior escalabilidade: um código simples é mais fácil de ser escalado para atender a novos requisitos, sem a necessidade de mudanças significativas.
- Maior performance: pode aumentar significativamente devido a não haver artefatos desnecessários, como interfaces genéricas com acessos de múltiplas partes
- Melhor acoplamento: o acoplamento torna-se baixo porque não há usos desnecessários entre as classes/módulos, apenas o necessário.

- Maior flexibilidade: um código simples é mais fácil de ser adaptado para atender a diferentes necessidades, sem a necessidade de mudanças significativas.

Em resumo, a simplicidade no desenvolvimento de código é uma característica fundamental para garantir que o software seja fácil de manter, escalar e evoluir, além de ser fácil de ser compreendido e trabalhado por outros desenvolvedores.

1.2 Relação com maus-cheiros de código

- **Método longo:** Métodos longos dificultam a compreensão e a manutenção do código, além de tornar mais propenso a erros.
- **Comentários ruins/comentários mentirosos:** Os comentários são uma parte importante da documentação do código, mas seus abusos podem levar a maus cheiros, como comentários redundantes ou desatualizados.
- **Classe grande**: As Classes grandes são difíceis de entender, manter e testar, pois contêm uma quantidade excessiva de informações em um único lugar.

1.3 Operações de refatoração corretivas

- Comentários apenas quando necessários: Uma forma de tornar o código mais simples e mais agradável visualmente é retirando comentários que não são estritamente necessários. Dessa forma evita-se atualizações no código e não nos comentários, deixando-os obsoletos e confusos. Além disso, há uma outra operação de refatoração que diminui ou até anula a necessidade de comentários: a refatoração Renomear método.
- Extrair classe: Classes grandes refletem código complexos e nada simples, nesse caso, a operação de refatoração Extrair classe ajudará a deixar o código mais simples visualmente e logicamente sem alterar seu comportamento se feito da maneira correta.
- Extrair método: Outro fator que deixa o código complexo são os métodos longos e nesse cenário deve-se utilizar a refatoração Extrair método. Essa operação vai transformar um método grande e complexo em vários métodos menores e mais simples que facilitarão a compreensão do leitor.

2. Elegância

2.1 Descrição

A elegância é uma característica importante na elaboração de código, pois reflete a capacidade do desenvolvedor de escrever código de forma clara, concisa e eficiente. Isso tem vários benefícios, incluindo:

- Manutenção mais fácil: código elegante é geralmente mais fácil de entender e modificar, o que diminui o tempo e o esforço necessários para realizar manutenções e atualizações.
- Menor probabilidade de erros: código elegante é menos propenso a erros, pois é mais fácil de entender e testar.
- Melhor performance: O código elegante geralmente é mais eficiente em termos de tempo e espaço, o que pode melhorar a performance do software.
- Maior escalabilidade: código elegante é mais fácil de ser escalado para atender a novos requisitos, sem a necessidade de mudanças significativas.
- Melhorar qualidade do código: código elegante é mais fácil de seguir as boas práticas de programação, tornando-o mais limpo, legível e compreensível.
- Maior satisfação do desenvolvedor: escrever código elegante é uma forma de arte e pode ser gratificante para o desenvolvedor, ele se sente mais orgulhoso do seu trabalho e isso pode levar a um melhor desempenho e motivação no trabalho.

Em resumo, a elegância no desenvolvimento de código é uma característica importante para garantir que o software seja fácil de manter, escalar e ser eficiente,

além de ser fácil de ser compreendido e trabalhado por outros desenvolvedores, e melhora a satisfação do desenvolvedor.

2.2 Relação com maus-cheiros de código

Pode relacionar todo característica que não se adequa ao *Clean Code,* podendo destacar:

- Método longo
- Aglomerado de dados
- Comentários ruins/ comentários mentirosos
- Classe inchada
- Instruções Switch
- Intimidade inapropriada
- Código duplicado
- Cadeias de mensagens

2.3 Operações de refatoração corretivas

- Renomear método: Um método pode ser renomeado de forma que venha a possuir um nome auto-explicativo, e consequentemente evitando a necessidade de comentários no código. Isso tornará o código mais limpo e coeso, pois muitas vezes o método é atualizado, mas os comentários não, o que pode gerar confusões para refatorações ou manutenções futuras.
- Extrair método/Dividir métodos longos em métodos menores: Ao se dividir um grande método em vários métodos pequenos, deixaremos o projeto de código mais elegante e agradável, além de aumentarmos a coesão e diminuirmos o acoplamento de código. O número de linhas de um bom método depende da linguagem que estamos utilizando no projeto, que podem ser mais ou menos verbosas, no entanto, nunca deve-se extrapolar e deixar o método crescer demais.

3. Modularidade

3.1 Descrição

A modularidade é uma característica importante na elaboração de código, pois permite que o código seja dividido em módulos ou componentes independentes e reutilizáveis. Em geral, a modularidade oferece maior capacidade de gerenciamento de desenvolvimento de software.

A modularidade de baixo acoplamento se refere à prática de dividir o código em módulos ou componentes independentes, de modo que cada módulo dependa o mínimo possível de outros módulos. Isso significa que os módulos devem ser

projetados de forma a serem independentes e não depender fortemente uns dos outros.

Por outro lado, a modularidade de alto acoplamento ocorre quando as diferentes partes ou módulos do código estão fortemente interligadas e dependentes uns dos outros. Um código de alto acoplamento é menos escalável, menos testável e mais difícil de manter do que um código de baixo acoplamento.

No contexto da engenharia de software, a modularidade permite que aplicativos típicos sejam divididos em módulos, bem como integração com módulos semelhantes, o que ajuda os desenvolvedores a usar código pré-escrito. Os módulos são divididos com base na funcionalidade, e os programadores não estão envolvidos com as funcionalidades de outros módulos. Assim, novas funcionalidades podem ser facilmente programadas em módulos separados. Isso tem vários benefícios, incluindo:

- Facilidade de manutenção: com o código dividido em módulos, é mais fácil identificar e corrigir problemas específicos sem afetar outras partes do código.
- Reutilização de código: com módulos independentes, é mais fácil reutilizar código em outras partes do projeto ou em projetos futuros.
- Escalabilidade: com módulos independentes, é mais fácil adicionar novas funcionalidades ou aumentar a capacidade do sistema sem afetar o código existente.
- Testabilidade: com módulos independentes, é mais fácil testar cada módulo separadamente, o que aumenta a confiabilidade do sistema.
- Melhora na colaboração: a modularidade permite a divisão do trabalho e a colaboração de vários desenvolvedores em um projeto, sem causar conflitos ou problemas.
- Melhoria na performance: com a modularidade, é possível otimizar cada módulo individualmente, o que pode melhorar significativamente a performance do sistema como um todo.

3.2 Relação com maus-cheiros de código

- **Classes inchadas:** As classes inchadas são meus-cheiros de código pois possuem uma quantidade excessiva de responsabilidades e que estão fazendo muito mais do que deveriam. Classes inchadas tornam o código difícil de entender, manter e testar, pois contêm muitas informações e lógicas diferentes.
- **Método longo:** Métodos longos tornam o código mais suscetível a erros e complicam sua compreensão e manutenção.
- **Código Duplicado:** Código duplicado pode aumentar a complexidade do código e tornar mais difícil a manutenção e correção de bugs. Além disso, o código duplicado também pode resultar em uma duplicação desnecessária de esforço e tempo de desenvolvimento, já que uma modificação pode ser transmitida por toda parte duplicada.

3.3 Operações de refatoração corretivas

- Mover método: Essa operação de refatoração consiste em mover um método de um classe para outra classe que faça mais sentido e que torne o projeto mais modularizado. Ao analisar uma classe, pode ser que seja encontrado um método que nem sequer utiliza dados da mesma, nesse cenário, é interessante utilizar a operação de refatoração Mover método que irá tornar o código mais coeso e dividido corretamente em módulos.
- Dividir classes grandes em classes atômicas: Em projetos pouco modularizados, uma das principais características é a presença de classes grandes, com várias linhas de código e que realizam diversas funções. Uma operação de refatoração válida nesse sentido é dividir tais classes grandes em classes menores e independentes, com funções melhor definidas. Essa refatoração irá deixar o código com melhor modularidade sem afetar o comportamento do projeto de código como um todo.
- Extrair classe: Com a operação de refatoração Extrair classe, é possível extrair uma classe de dentro de outra, em outras palavras, geramos uma nova classe que terá uma função melhor definida, tornando o código mais modularizado.

4. Portabilidade

4.1 Descrição

A portabilidade do código se refere à capacidade de um software ser executado em diferentes sistemas operacionais ou plataformas sem necessidade de modificações significativas. Um código portátil é mais flexível e pode ser facilmente adaptado para diferentes ambientes, o que é importante em projetos de software que precisam ser executados em diferentes dispositivos ou sistemas.

Outro viés da portabilidade é que um componente ou sistema de software é proporcional à quantidade de esforço necessário para que funcione num novo ambiente. A portabilidade também pode ser importante para garantir a escalabilidade do projeto, pois é possível mover o software para plataformas maiores ou mais poderosas sem precisar refazer completamente o código. Além disso, a portabilidade também pode tornar o software mais atraente para potenciais clientes ou usuários, pois pode ser executado em uma variedade de dispositivos ou sistemas.

A portabilidade é importante para o desenvolvimento de software, pois pode facilitar a manutenção e o teste do código, já que não é necessário se preocupar com as diferenças entre diferentes plataformas.

Em resumo, a portabilidade do código é importante para garantir a flexibilidade, escalabilidade e atratividade do projeto, além de facilitar a manutenção e teste do código.

4.2 Relação com maus-cheiros de código

- **Dependência de plataforma:** Isso se refere ao fato de que o código depende de recursos específicos de uma plataforma, tornando-o menos portável.
- **Hardcoded values:** Valores codificados no código que são específicos de uma determinada plataforma ou configuração, o que os torna menos portáveis.
- **Uso inadequado de bibliotecas ou recursos:** O uso inadequado de bibliotecas ou recursos que são específicos de uma plataforma pode afetar a portabilidade do código.

4.3 Operações de refatoração corretivas

- Extração de interface - Essa técnica consiste em criar uma interface que descreva os comportamentos necessários para a execução de determinada funcionalidade, independentemente da plataforma. Em seguida, você implementa a interface para a plataforma específica e usa a interface para garantir a independência da plataforma no código principal. Dessa forma, ao precisar mudar a plataforma, você pode fazer isso sem afetar o código principal, pois ele já está desacoplado da plataforma específica.
- Extração de constantes - Essa técnica consiste em extrair valores fixos presentes no código e colocá-los em constantes ou variáveis de configuração, de forma que sejam fáceis de alterar e não precisam ser procurados e modificados em vários pontos do código. Dessa forma, você torna o código mais flexível e menos propenso a erros, já que você só precisa alterar o valor em um único lugar, sem afetar o resto do código. Além disso, ao usar constantes, você aumenta a legibilidade do código, já que ele se torna mais claro e conciso.
- Substituição de biblioteca - Essa técnica consiste em avaliar se a biblioteca ou recurso está sendo utilizado corretamente e se existe uma alternativa mais adequada. Se houver, você pode substituir a biblioteca ou recurso atual pela alternativa mais adequada, o que pode melhorar a performance, a escalabilidade e a manutenibilidade do código. Além disso, ao substituir uma biblioteca ou recurso, você pode corrigir erros ou problemas de segurança presentes no uso inadequado da biblioteca ou recurso original.

5. Ausência de Duplicidade

5.1 Descrição

As duplicidades ocorrem quando existem trechos de código repetidos no mesmo projeto. Essa duplicidade pode vir a causar diversos empecilhos para a realização da manutenção de softwares, pois causa a necessidade de alterar diversas linhas de código repetidas, além disso, torna mais complexo o entendimento do mesmo possibilitando o surgimento de diversos bugs.

A ausência de duplicidade no código é importante para garantir que o projeto seja fácil de manter, fácil de ler e entender, escalável e fácil de testar. Vantagens da ausência de duplicidade no código:

- Manutenção: Duplicar código significa que há múltiplas cópias do mesmo trecho de código que precisam ser mantidas e atualizadas. Isso pode ser desafiador, especialmente em projetos de software de grande escala, e pode levar a erros e inconsistências. Ao remover a duplicidade, o código se torna mais fácil de manter e corrigir.
- Leitura e compreensão: Código duplicado pode ser confuso e difícil de ler, especialmente se as diferentes cópias estão espalhadas por diferentes partes do projeto. Isso pode dificultar a compreensão do código por parte dos desenvolvedores e tornar mais difícil encontrar e corrigir erros.
- Escalabilidade: Código duplicado pode ser ineficiente e difícil de escalar. Se há múltiplas cópias de um trecho de código, é mais provável que essas cópias sejam executadas múltiplas vezes, o que pode levar a problemas de desempenho. Além disso, a remoção de duplicidade pode permitir a criação de funções ou módulos que podem ser reutilizados em diferentes partes do projeto, o que pode aumentar a escalabilidade do código.
- Teste: Código duplicado pode ser mais difícil de testar, pois é necessário testar cada cópia separadamente. Além disso, se houver erros em uma cópia do código, é mais provável que esses erros sejam replicados em outras cópias. Ao remover a duplicidade, o código é mais fácil de testar e menos propenso a erros.

5.2 Relação com maus-cheiros de código

A duplicação de código é um mau-cheiro de código que ocorre quando o mesmo código ou conjunto de instruções é repetido em dois ou mais locais do software. Segundo o livro *“The Pragmatic Programmer”* o qual teve forte influência de Martin Fowler, a duplicação de código ocorre pelos seguintes motivos:

- Duplicação imposta: Quando a duplicação de código é introduzida por restrições externas, como sistemas operacionais diferentes, bibliotecas não disponíveis em todas as plataformas ou limitações de hardware. Nestes casos, a duplicação não pode ser evitada, mas pode ser minimizada ou isolada, para tornar sua manutenção mais fácil.

- Duplicação inadvertida: Neste caso a duplicação ocorre quando o código é duplicado sem que o programador tenha consciência disso. O principal motivo para isso ocorrer é a falta de conhecimento sobre o código existente.
- Duplicação impaciente: Ocorre quando o código é duplicado por pressa ou falta de planejamento. Isso pode ocorrer quando o programador não tem tempo ou recurso suficiente para encontrar uma solução mais adequada ou quando a pressão para entregar o software é muito alta.
- Duplicação entre desenvolvedores: A duplicação ocorre quando vários programadores estão trabalhando em diferentes partes do mesmo software ao mesmo tempo. Como estão trabalhando em tarefas diferentes, podem não saber que outro desenvolvedor já tenha escrito um código semelhante que pode ser reaproveitado.

5.3 Operações de refatoração corretivas

- Extração de método: Quando houver vários trechos de código repetidos e realizando a mesma função, é interessante realizar a extração de método, ou seja, colocar esse trecho de código que se repete dentro de um método que será chamado várias vezes dentro do código. Essa simples operação de refatoração é essencial para evitar a duplicação de código e garantir a ausência de duplicidade/repetição.
- Extração de método - Puxar para cima: A extração de método - Puxar para cima é utilizada quando há vários trechos de código repetido nas classes filhas de um projeto. Nesse caso, essa operação de refatoração irá retirar os trechos repetidos das classes filhas e irá “puxar para cima”, gerando um método na classe mãe, que poderá ser herdado para as classes filhas e chamados quantas vezes precisar. Essa operação é importante para evitar a duplicação de código e também para não haver métodos iguais nas classes filhas.