Universidade de Brasília - UnB

Faculdade UnB Gama - FGA

Técnicas de Programação para Plataformas Emergentes - FGA0242

TRABALHO PRÁTICO 03

Grupo 06:

Bruna Almeida Santos 17/0100626

Bruno Carmo Nunes 18/0117548

Damarcones Porto 15/0122187

Estevão de Jesus Reis 18/0052616

Marcos Vinícius Rodrigues da Conceição 17/0150747

Tomás Veloso Peixoto 18/0138596

Docente:

André Luiz Peron Martins Lanna

Tema:

Qualidade do projeto de código

SUMÁRIO

1. Introdução

A qualidade de código fonte é aspecto essencial para o sucesso de um projeto de software. A boa qualidade do código é definida como um código limpo e que faz o que foi projetado para fazer [1]. Ter um código limpo faz com que o desenvolvimento flua melhor e que necessite de menos manutenção. Em resumo, um bom código deve ser simples, bem testado, refatorado, bem documentado e , principalmente, de boa performance [2]

Segundo Goodliffe [3], “bons projetos têm uma série de características atraentes, cujos opostos são indicadores seguros de design ruim”. Em seu estudo de 2006, Goodliffe lista nove características que definem um bom projeto, sendo eles:

- Simplicidade;
- Elegância;
- Modularidade;
- Boas interfaces;
- Extensibilidade;
- Ausência de duplicidades;
- Portabilidade;
- Idiomático e;
- Boa documentação.

As características de um bom projeto levantadas por Goodliffe [3] são essenciais no combate ao surgimento dos ditos maus cheiros de código, definidos por Fowler [4]. Trata-se de pontos no código em que princípios de bom projeto não são considerados ou podem ser melhorados [6]. Duplicidade de código, métodos longos demais e classes com muitas variáveis são exemplos de maus cheiros, que podem diminuir a qualidade do código, torná-lo lento e de difícil entendimento.

A refatoração é a solução mais eficiente para solucionar esses maus cheiros de código [6]. são pontos em que há possibilidades para aplicação de refatoração. Refatoração é o processo de alterar um software de uma maneira que não mude o seu comportamento externo e ainda melhore a sua estrutura interna [8].

Diante desses fatos, o objetivo do presente trabalho é descrever algumas das mais importantes características que definem a qualidade de um projeto. A descrição conta com:

um detalhamento sobre seus efeitos no código do projeto; relação com os maus cheiros de código definidos por Fowler [4]; e, quando possível, uma operação de refatoração capaz de levar o projeto de código a ter a característica em análise. A equipe escolheu 5 entre 9 aspectos que qualificam um bom projeto levantados por Goodliffe [3].

2. Características de um bom projeto

As características de um bom projeto de software escolhidas pela equipe foram: simplicidade; modularidade; extensibilidade; portabilidade; e boa documentação. Todos esses aspectos serão detalhados a seguir.

Simplicidade

Um código não é simples por possuir poucas linhas, e sim por apresentar características concisas e coerentes com a necessidade em que será aplicado. Cada aplicação é desenvolvida com um objetivo e, quando o objetivo estiver bem definido, é a vez do desenvolvedor pôr em prática suas habilidades.

Refatorar qualquer código para torná-lo "simples" não é algo simples de se fazer, requer habilidades do desenvolvedor para aplicar as mudanças sem correr o risco de estragar todo o funcionamento do projeto. O desenvolvedor que consegue retirar partes desnecessárias de código terá uma vantagem sobre os demais, sempre que for necessário consertar erros eventuais ou prestar manutenção, pois a leitura do código será otimizada e as partes críticas estarão evidentes.

A simplicidade no código evita o desperdício de recursos computacionais. Em um código com a função de leitura e soma de dez números, por exemplo, é mais eficiente ler e somar todos na mesma interação do que criar dois loops separados, um para leitura e outro para soma dos números. Esse exemplo pode ser relacionado com o mal-cheiro intitulado de código duplicado, em que repetidos loops realizam a mesma tarefa que um único loop. Essa economia pode ser decisiva na escolha de melhores projetos em maratonas de programação, por exemplo.

E com os conceitos apresentados por Goodliffe [3], em que afirma que “menos é mais” e “Se uma estrutura parece óbvia, não pense que foi simples de chegar nela”, pode-se inferir que a simplicidade é necessária e de grande importância em projetos complexos, com o objetivo de torná-los eficientes.

Modularidade

Segundo Moura [5], modularidade é um conceito extremamente importante em projetos de software robustos. Através da modularização, o software pode ser dividido em diferentes partes. Isso resulta no aumento da produtividade, tanto no início do desenvolvimento como nos estágios de testes e de manutenção.

Sistemas bem modulares são mais fáceis de manter e evoluir pois podem ser entendidos e modificados de forma independente, sem que um módulo interfira um no outro. Além disso, reduz o tempo gasto para identificar a origem dos erros, e ameniza o risco de modificações criarem novos problemas em outras partes do software, diminuindo o esforço e tempo que os desenvolvedores gastam para fazer a manutenção do software. Consequentemente isso também reduz o dinheiro gasto pelas empresas com a manutenção de seus sistemas ou software.

A característica de modularidade em *software* possui uma forte relação com os maus cheiros de código, no sentido de evitar e até mesmo removê-los do sistema. Segundo Lanna [6], alguns desses mal cheiros são:

- **Código duplicado:** Esse tipo de mau cheiro remete à situação de um programa possuir em diversos lugares em seu código, blocos de códigos que realizam exatamente a mesma tarefa. Essa prática torna o *software* bem mais difícil de manter, visto que caso precise modificar o comportamento dessa tarefa, será preciso replicar a modificação em cada um dos lugares onde ela é utilizada. Modularizando esse bloco de código com um método ou função, pode ser utilizado apenas a chamada dele, onde é necessário ser utilizado e caso precise modificar seu comportamento ou corrigir um erro encontrado nele, será feito apenas uma vez e essa modificação será propagada em todos os lugares onde é chamado.
- **Método longo:** Esse mau cheiro afeta o sistema no sentido de que, quanto maior o

método é em termos de linhas de códigos e tarefas, mais difícil é de entendê-lo, o que torna mais difícil de debugar o código, fazer alterações quando necessário, evoluir o *software* e mantê-lo. Isso aumenta bastante os esforços e tempo dos desenvolvedores em qualquer tarefa que deve ser realizada neste método. Ao modularizar um método como esse, ele se torna mais enxuto, dividido em métodos menores e apenas utilizando suas chamadas para dentro do método original que será responsável principalmente por chamar cada um dos métodos, e devolver o retorno final caso seja necessário.

Consequentemente, para que a modularidade seja vista em sistemas de *software*, corrigindo e prevenindo maus cheiros no código, podem ser utilizados diversos métodos de refatoração para que o software esteja adequadamente modularizado. Exemplos disso são os seguintes métodos:

- **Extração de método:** É utilizado quando se tem um fragmento de código que pode ser agrupado em um método. Ele é aplicado transformando o fragmento em um método cujo nome explica o propósito do método. Esse método é capaz de eliminar maus cheiros como "código duplicado" ou "método longo".

- **Substituição de método por objeto-método:** É utilizado quando existe um método longo que usa variáveis locais que não te permitem aplicar a extração de Método. Ele é aplicado transformando o método em seu próprio objeto, em que todas as variáveis locais se tornam campos naquele objeto. Esse método é capaz de eliminar maus cheiros como "método longo".

Dessa forma é possível desenvolver adequadamente um sistema com as características de modularidade, elevando bastante a qualidade do software em relação à sua manutenibilidade.

Extensibilidade

Um código bem projetado permite que, quando necessário, que novas funcionalidades sejam encaixadas em lugares apropriados, afirma Goodliffe [3]. A periculosidade disso é se deixar levar por códigos sobre-engenhados, onde os mesmos tentam se relacionar com qualquer modificação futura.

A extensibilidade pode ser acomodada por *software scaffolding*, ou “andaime de *software*”, onde tem plugins dinamicamente carregados, que escolhem cuidadosamente a hierarquia de classes com interfaces abstratas no topo, provendo funções de *callback* úteis, e estruturas de código lógicas e maleáveis.

Um bom designer pensa cuidadosamente como o software vai ser estendido. Códigos aleatórios com *hooks* para extensibilidade podem na verdade degradar a qualidade. O certo seria balancear a funcionalidade que é necessária no código naquele momento, e o que pode ser necessário determinar o quão extensível o *design* deveria ser.

Em relação à evitar duplicação, Goodliffe [3] explica que um código com bom *design*, não contém duplicações, o código nunca deve se repetir. Duplicação é a inimiga de um *design* elegante e simples. Códigos redundantes levam a um programa frágil. Um exemplo disso são dois pedaços de código similar, que diferem somente em poucos detalhes, acarreta para a procura e reparo de *bugs* em um deles, e o outro acaba não tendo essa manutenção. Isso claramente compromete a segurança do código.

A maioria da duplicação vem da programação de copiar e colar - copiando o código no editor de texto. Podendo então levantar subitamente a “reinvenção da roda” por programadores que não entendem o sistema.

Goodliffe [3] sugere que:

- Ao deparar-se com códigos similares em seções separadas do mesmo projeto, generalize para uma função com parâmetros apropriados. Agora o programador terá um único local para reparar qualquer falha. Onde o próprio benefício é ter a funcionalidade mais clara que é descrita pelo nome da função.

- Classes que parecem similares em algumas funcionalidades podem ser colocadas em uma superclasse ou o desaparecimento de uma interface que descreve um comportamento comum.

Acima, vemos as descrições de mau cheiro de Fowler [4], onde o Código duplicado pode ser solucionado ao extrair o método. E o posterior, um classe inchada com código duplicado, que também tem como solução a extração de classe, ou a outra proposta de extrair uma interface.

Para uma operação de refatoração, temos um dos exemplos do catálogo de refatoração

de Fowler [4], demonstrado na Figura 1:

```
void printOwing() {
                printBanner();

                //print details
                System.out.println ("name:     " + _name);
                System.out.println ("amount    " + getOutstanding());
        }
```

```
void printOwing() {
                printBanner();
                printDetails(getOutstanding());
        }

        void printDetails (double outstanding) {
                System.out.println ("name:     " + _name);
                System.out.println ("amount    " + outstanding);
        }
```

Figura 1 - Exemplo sobre extrair métodos.

Portabilidade

A ênfase nas melhores práticas e em padrões de software são importantes para garantir a qualidade de software. Uma boa maneira de gerenciar um projeto de software é a preocupação com a portabilidade do sistema.

A obtenção de portabilidade para diferentes plataformas de hardware e software é um dos maiores problemas no projeto de sistemas. Redirecionar o Lexi para uma nova plataforma não deveria exigir uma revisão geral, pois, então, não valeria a pena o redirecionamento. Deveríamos tornar a portabilidade tão fácil quanto possível (GAMMA, 2000) .

A ISO/IEC 9126 [10] define a portabilidade como “a capacidade do sistema ser transferido de um ambiente para outro”. Como "ambiente", devemos considerar todos os fatores de adaptação, tais como diferentes condições de infraestrutura (sistemas operacionais, versões de bancos de dados, etc.), diferentes tipos e recursos de *hardware* (tal como aproveitar um número maior de processadores ou memória). Além destes, fatores como idioma ou a

facilidade para se criar ambientes de testes devem ser considerados como características de portabilidade. Suas subcaracterísticas são:

- Adaptabilidade, representando a capacidade do *software* se adaptar a diferentes ambientes sem a necessidade de ações adicionais (configurações);
- Capacidade para ser instalado identifica a facilidade com que pode se instalar o sistema em um novo ambiente;
- Coexistência mede o quão facilmente um *software* convive com outros instalados no mesmo ambiente;
- Capacidade para substituir representa a capacidade que o sistema tem de substituir outro sistema especificado, em um contexto de uso e ambiente específicos. Este atributo interage tanto com adaptabilidade quanto com a capacidade para ser instalado;
- Conformidade: Capacidade do produto de software de estar de acordo com normas, convenções, guias de estilo ou regulamentações relacionadas à portabilidade.

A portabilidade deve-se levar em consideração a não dependência da plataforma, e a forma de garantir a qualidade do projeto ao utilizar da portabilidade, é utilizar apenas quando for necessário, sendo uma má prática tentar aplicar a portabilidade em todos os momentos. Revisões no desenvolvimento de *software* ocorrem em todo o ciclo de vida do desenvolvimento do sistema, para a avaliação da qualidade do projeto. Um empecilho à prática da portabilidade é a diversidade de padrões de interação, que são as formas de como o *software* apresenta-se ao usuário. Dessa maneira, um sistema que possui portabilidade entre sistemas, e é utilizado em várias plataformas, deve seguir o padrão de interação e apresentação para o usuário de acordo com cada plataforma.

Desenvolver um código que seja executado em diversas plataformas é uma atividade difícil, e eliminar também a dependência das classes em relação às outras também não é uma tarefa fácil, as boas práticas são as técnicas de refatoração e redesenho do sistema. A refatoração é uma atividade que melhora o código, eliminando e diminuindo as dependências de classes, métodos e heranças, tornando assim, o código mais independente e aumentando a qualidade do *software* no quesito da portabilidade. O redesenho procura melhores estratégias na escolha dos padrões de projeto para a aplicação.

No livro do autor Goodliffe [3], é mencionado como conceito chave, a seguinte definição: Gerencie a portabilidade do seu código em seu design, em vez de invadi-lo como uma reflexão tardia. É uma definição importante, pensar na portabilidade no momento inicial do projeto, em cada revisão realizada, e nos momentos de refatoração, pois dessa maneira diminui o tempo gasto para a conclusão do projeto, deixa o projeto com uma flexibilidade maior para novas alterações, o tempo que seria necessário para adaptar o projeto para mudanças que ocorrem ao longo do desenvolvimento, como os requisitos, também é menor. Dessa forma a portabilidade é uma característica importante para a qualidade do projeto.

Boa Documentação

A principal razão que os desenvolvedores dão para não documentar seu código é a falta de tempo. Ao desenvolver uma nova funcionalidade que precisa ser entregue dentro de um determinado período de tempo, raramente temos um momento para parar tudo e concentrarmos em documentar nosso código. Na equação de projetar, escrever o código, passar por revisão, testes (automação, unitários, integração e etc..) a documentação quase sempre fica de fora. Esse detalhe deixado de lado pode fazer uma grande diferença no futuro.

Uma coisa que precisa ser muito clara na mente de quem é desenvolvedor é que aquele código que você está trabalhando hoje, muito provavelmente vai passar por muitas outras mãos. Quando esse dia chegar você não vai se lembrar bem o que escreveu e o porquê. A solução para esse problema é a documentação. Levar um tempo extra para escrever uma descrição adequada sobre o que você trabalhou vai salvar um tempo muito maior no futuro. Quando alguém ou até mesmo você precisa entender o que acontece e como seu código trabalha, basta ir buscar na documentação, você como indivíduo deixa de ser parte extremamente necessária para o fluxo de entendimento do código. E isso salva o tempo de todo mundo, quem tem dúvida não precisa entrar em contato e você mesmo não tem que quebrar a cabeça para entender o que “aquela linha de código” está fazendo.

É espalhado por muitos cantos que um bom código não precisa ser documentado. Um código bem escrito, conciso, bem pensado é explícito por si só. Tal princípio segundo Fowler [7] não está dizendo que o código é a única documentação. Embora muitas vezes ele tenha ouvido isso sobre *Extreme Programming* - ele nunca ouviu os líderes do movimento *Extreme*

Programming dizerem isso. Normalmente, há necessidade de documentação adicional para atuar como um complemento ao código.

A justificativa para o código ser a principal fonte de documentação é que ele é o único suficientemente detalhado e preciso para desempenhar esse papel - um ponto feito de forma tão eloquente pelo famoso ensaio de Jack Reeves "O que é design de software?"

A consequência importante desse princípio é que se torna necessário que os programadores sejam capazes em garantir que o código fique da maneira mais clara e legível possível. Para que isso seja alcançado, é necessário que se utilize sempre que possível das várias técnicas de refatoração para a retirada de maus cheiros do código, o tornando mais claro e simples possível.

Na construção de uma classe, por exemplo, com o passar do tempo ela pode ir crescendo e ficando cada vez mais difícil trabalhar de forma eficaz e com clareza, como mostrado na Figura 2.

```
class DateUtil {
    boolean isAfter(int year1, int month1, int day1, int year2, int month2,
int day2) {
        // implementation
    }

    int differenceInDays(int year1, int month1, int day1, int year2, int
month2, int day2) {
        // implementation
    }

    // other date methods
}
```

Figura 2 - Exemplo de mau cheiro: *Bloater.*

Todos os métodos acima funcionam com datas. Portanto, todos eles recebem três argumentos inteiros: ano, mês e dia. A simples ação de agrupá-los em uma classe Date torna o código mais legível, exemplificado na Figura 3.

```
class Date {
    int year;
    int month;
    int day;
}

class DateUtil {
    boolean isAfter(Date date1, Date date2) {
        // implementation
    }

    int differenceInDays(Date date1, Date date2) {
        // implementation
    }

    // other date methods
}
```

Figura 3 - Exemplo de mau cheiro: *No Bloater.*

O simples fato de diminuir a quantidade de maus cheiros no seu código o torna mais limpo, conciso e a sua documentação ainda mais simples de entender. Apesar de trabalhoso, esse processo é necessário para garantir a melhor manutenibilidade do sistema. Só isso não dispensa a necessidade de documentação, e aqui está o porquê:

- Estamos todos muito familiarizados com a robustez do código que compreende um recurso. Olhando para uma seção de código, pode não deixar claro que existem outras seções que estão profundamente vinculadas a ela;
- Cada serviço que você cria tem uma Interface de Programação de Aplicação (do inglês *Application Programming Interface* - API) exclusiva para ele. Escrever como usar essa API requer documentação que pode ser lida fora do código. Você não quer inflar a própria classe com detalhes sobre como usar a API;
- Colegas de trabalho que trabalham em departamentos diferentes (que podem não ser desenvolvedores) podem querer entender o que você fez e como funciona;
- Apenas o ato em si pode fazer com que você olhe de forma diferente para o código que escreveu. Explicar o que seu código faz fará com que você avalie a validade dele e talvez considere mudar as coisas se elas não atenderem às suas expectativas;
- Pelo motivo de facilitar a vida na posteridade.

3. Considerações Finais

Este trabalho teve como objetivo descrever características que definem a qualidade de um projeto. A descrição desses aspectos incluíram um detalhamento sobre seus efeitos no código do projeto, sua relação com alguns dos maus cheiros de código definidos por Fowler [4] e, por fim, uma operação de refatoração capaz de levar o projeto de código a ter a característica em análise.

A presença das cinco características descritas neste trabalho garantem uma boa qualidade de projetos de software. Enquanto a simplicidade garante fácil entendimento e implementação do código, a modularidade assegura que o código tenha coesão e acoplamento. Por sua vez, a extensibilidade permite que novas funcionalidades sejam adicionadas ao código sem grandes dificuldades, e a portabilidade garante que, quando necessário, o código seja apropriadamente portátil. Por fim, a documentação de um bom projeto deve ser clara, sucinta e coerente.

Além das características descritas no trabalho, outros aspectos são igualmente importantes para o êxito de um projeto em desenvolvimento. A elegância anda de mãos dadas com a simplicidade, e boas interfaces garantem a satisfação do usuário. A ausência de duplicidade garante um código mais limpo e claro, diminuindo a ocorrência de *bugs*. E finalmente, um bom projeto emprega naturalmente as melhores práticas, adequando-se tanto à metodologia de projeto quanto às expressões idiomáticas da linguagem de implementação.

É crucial que equipes de desenvolvimento levem em consideração essas nove características para garantir o sucesso de seu projeto de *software*.

REFERÊNCIAS

[1] BRING. **Qualidade do código e sua importância para um desenvolvimento bem sucedido.** Disponível em: *https://bring.com.br/blog/2019/09/10/qualidade-do-codigo-e-sua-importancia-para-um-desenvolvimento-bem-sucedido/*. Acesso em 02 de maio de 2022.

[2] PANTALIÃO, André. **6 questões para a qualidade do código**. Disponível em: *https://vizir.com.br/2016/09/6-questoes-para-a-qualidade-do-codigo-ruby-conf-br-4/*. Acesso em 02 de maio de 2022.

[3] GOODLIFFE, Pete. **Code craft: the practice of writing excellent code**. 1ª ed. San Francisco: No Starch Press, Inc. 2006.

[4] FOWLER, M. , **Refactoring: Improving the Design of Existing Code**, 2ª ed. Addison-Wesley , Boston, MA, USA, 2019

[5] MOURA, Lucas Ferreira. **MODULARIDADE DE SISTEMAS DE SOFTWARE**. 2017. Disponível em: *https://iftm.edu.br/ERP/MPES/EVENTOS/arquivos/030517150836_resumo_sin_lucas_moura.pdf*. Acesso em 30 de Abril de 2022.

[6] LANNA, André Martins. **Oportunidades de Refatoração**. Disponível em: *https://docs.google.com/presentation/d/1BG1DVjtOZeG-j3Fmj1cY1gz-4AW9FphX/edit#slide=id.p1*. Acesso em 30 de Abril de 2022.

[7] FOWLER, M. , **Code As Documentation.** 2005. Disponível em: *https://martinfowler.com/bliki/CodeAsDocumentation.html*. Acesso em 30 de Abril de 2020.

[8] DEVMEDIA. **Introdução à Refatoração**. Disponível em: *https://www.devmedia.com.br/introducao-a-refatoracao/21377*. Acesso em 02 de maio de 2022.

[9] FREECODECAMP. **Why documentation matters, and why you should include it in your code.** Disponível em: *https://www.freecodecamp.org/news/why-documentation-matters-and-why-you-should-include-it-in-your-code-41ef62dd5c2f/*. Acesso em 02 de maio de 2022.

[10] WIKIPÉDIA, a enciclopédia livre. **ISO/IEC 9126.** Disponível em: *https://pt.wikipedia.org/wiki/ISO/IEC_9126*. Acesso em 02 de maio de 2022.