Trabalho Prático 3

Projeto de Software

Projeto de software é, muitas vezes, entendido como uma atividade dentro do processo de desenvolvimento em que uma equipe define as estruturas do software e as interações entre elas. O intuito com esse tipo de abordagem é facilitar o desenvolvimento à partir dos artefatos gerados durante a atividade de projeto. Os artefatos gerados servem como uma especificação do que o software deve fazer e quais as propriedades que devem ser garantidas quando ele estiver em operação.

Contudo, a realidade é diferente. O código é também uma atividade de projeto de software. Basta relembrar do conceito de Refatoração. Martin Fowler, em seu livro de refatoração, define como sendo "o aperfeiçoamento do projeto de código sem alterar o seu comportamento externamente observável." (Fowler, 1999). Por mais detalhado que sejam os artefatos de software criados durante a atividade de projeto de software, eles não conseguirão capturar todos os detalhes que são considerados durante a codificação. Os menores detalhes escapam dos modelos gerados e, geralmente, são descobertos durante a atividade de codificação sob a forma de inconsistências, falhas ou erros. Disso segue um conceito-chave muito importante:

Conceito chave:
Programar é uma atividade de projeto de software.

Portanto, são os programadores quem refinam e extendem o projeto de software na medida em que, constantemente, várias vezes ao dia, refinam e extendem o código da aplicação. Isso não significa que a ideia inicial de um projeto deva ser descartada. Pelo contrário, ter em mente o projeto que se deseja alcançar é importante, pois ele se torna uma meta a ser alcançada. Essa meta de projeto deve ser minimamente coerente para que o projeto resultante não se transforme em um projeto desestruturado e de difícil manutenção e evolução.

Projeto de Software

Desenvolvedores, em suas funções, realizam atividades de projeto de código, como classes, objetos e estruturas de dados. Essas pequenas contribuições vão sendo incorporadas ao projeto do software na medida em que são desenvolvidas. Em escala ascendente, os níveis de projeto de software são:

1. **Funções:** nível mais elementar das atividades de projeto, mas não menos importante. Se as rotinas são projetadas sem cuidado todo o sistema será impactado. Uma vez definido quais são as funções necessárias, trabalha-se na implementação interna, definindo algoritmos e o fluxo de execução.
2. **Classes e tipos de dados:** após a definição das funções, as atividades de codificação que resultam em alterações no projeto concentram-se nas definições de interfaces para utilização dessas funções. Nesse nível busca-se projetar interfaces que sejam representem o conjunto das funções encapsuladas. Isso se dá, no contexto de orientação a objetos, por meio da definição de classes e interfaces.

3. **Módulos e componentes:** o passo seguinte é organizar as classes desenvolvidas em conjunto de classes, compreensíveis, que guardem alguma relação de similaridade ou proximidade entre si. Costuma-se chamar esses agrupamentos de módulos ou componentes, contudo, é necessário atentar-se ao que se entende por módulos. Por exemplo, pode ser considerado como um conjunto de classes em linguagens orientadas por objetos ou como bibliotecas em linguagens procedurais. Nesse ponto do projeto o que se busca é a definição de interfaces publicadas, as quais deverão ser estáveis pois desempenham o papel de contratos entre os módulos e entre as equipes que trabalham nelas.
4. **Arquitetura do sistema:** nesse ponto observa-se o sistema e seus subsistemas como um todo. O projeto arquitetural desempenha forte influência no desempenho e nas características do sistema como um todo.

O projeto de software é a base sobre a qual todo o software será construído, uma vez que essa base é bem projetada e estruturada aumentam-se as chances do software construído ter qualidade. São características de um bom projeto de software:

- fácil de escrever;
- fácil de entender;
- fácil de manter;
- menos provável de ter bugs e, por fim,
- mais resiliente a mudanças.

Tudo isso exposto, conclui-se que ter um projeto de software correto, principalmente desde o início, é muito importante. É possível pensar em características de bom projeto em nível micro, durante a codificação, de modo que as boas práticas se acumularão e terão impactos no projeto do software como um todo. Vários desses princípios guardam estreita relação com as operações de refatoração de modo que é possível realizar melhorias em projetos de código existentes aplicando uma ou uma sequência de operações. Em linhas gerais, os princípios de um bom projeto de código são:

- simplicidade;
- elegância;
- modularidade;
- boas interfaces;
- extensibilidade;
- evitar duplicação;
- portabilidade;
- código deve ser idiomático e bem documentado.

Enunciado

Com base na descrição acima, cada grupo de trabalho deverá responder as seguintes perguntas:

1. Para cada um dos princípios de bom projeto de código mencionados acima, apresente sua definição e relacione-o com os maus-cheiros de código apresentados por Fowler em sua obra.

- **Simplicidade**: A simplicidade significa criar um código sem complexidade desnecessária, que faz apenas o que precisa. Um código simples é mais fácil de entender, testar e manter.
    - **Maus-cheiros**:
        - **Complexidade Acidental:** O código émais complicado do que deveria ser: implementação complicada e/ou funcionalidades desnecessárias;
        - **Códigos longos:** Métodos ou classes que fazem muito, podem ser divididos em unidades menores.
- **Elegância:** Um código elegante é aquele que resolve problemas de forma clara e direta, com soluções que parecem naturais para o problema em questão.
    - **Maus-cheiros:**
        - **Código Duplicado:** Repetir trechos de código;
        - **Código Desnecessário:** Presença de código que não contribui diretamente para a funcionalidade.
- **Modularidade:** Divisão do sistema em partes menores, módulos, que podem ser desenvolvidos, testados, e mantidos de forma independente.
    - **Maus-cheiros:**
        - **Classes Grandes:** Classes que fazem muito podem ser divididas em módulos menores;
        - **Métodos Longos:** Métodos que fazem muitas coisas diferentes podem ser quebrados em métodos menores e mais coesos.
- **Boas Interfaces:** Uma boa interface é intuitiva e fácil de usar, fornecendo exatamente o que o cliente precisa sem expor detalhes internos desnecessários.
    - **Maus-cheiros:**
        - **Código Ousado:** Quando uma classe usa mais do que deveria de outra classe, isso indica que a interface não está bem definida ou encapsulada;
        - **Invasão de Intimidade:** Classes que conhecem muitos detalhes internos de outras classes.
- **Extensibilidade:** Capacidade do sistema de crescer e acomodar novas funcionalidades sem precisar de mudanças significativas no código existente.
    - **Maus-cheiros:**
        - **Mudança Divergente:** Quando uma alteração em um aspecto do sistema requer modificações em várias classes;
        - **Rigidez:** Dificuldade para mudar o sistema é um sinal de que ele não foi projetado para ser extensível.
- **Evitar Duplicação:** Evitar duplicação significa não repetir o mesmo código em vários lugares.
    - **Maus-cheiros:**
        - **Código Duplicado:** O nome é autoexplicativo;
        - **Shotgun Surgery:** Quando uma pequena mudança requer várias modificações em locais diferentes, isso pode indicar que o código foi duplicado em muitos lugares.
- **Portabilidade:** Portabilidade é a capacidade de um sistema de ser executado em diferentes ambientes (por exemplo, sistemas operacionais ou arquiteturas de hardware) sem necessidade de grandes modificações.
    - **Maus-cheiros:**

        - **Dependências Externas Excessivas:** Código que depende fortemente de características específicas de um ambiente;
        - **Código Acoplado:** Alto acoplamento entre módulos, pois alterações em um módulo podem exigir mudanças em outros.
- **Código Idiomático e Bem Documentado:** Código idiomático segue as convenções e boas práticas da linguagem de programação utilizada. Um código bem documentado é claro e explica suas intenções, facilitando a compreensão.
    - **Maus-cheiros:**
        - **Nomes Sem Sentido:** Uso de nomes de variáveis, métodos ou classes que não refletem seu propósito dificulta a compreensão;
        - **Comentários Excessivos ou Faltantes:** Comentários mal utilizados podem indicar que o código não está claro ou que a intenção do programador não é clara.

2. Identifique quais são os maus-cheiros que persistem no trabalho prático 2 do grupo, indicando quais os princípios de bom projeto ainda estão sendo violados e indique quais as operações de refatoração são aplicáveis. **Atenção:** não é necessário aplicar as operações de refatoração, apenas indicar os princípios violados e operações possíveis de serem aplicadas.

Analisando o projeto ainda é possível encontrar maus-cheiros, por exemplo:

- Na classe SaleProcessor o método processSale:

```
public float processSale() {
    float amount = 0;

    for(Product p : this.products){
        amount+= p.getPrice();
    }

    float discount =
saleService.calculateDiscount(this.customer.getType(), amount,
this.paymentMethod);
    amount -= discount;
    float tax  = saleService.calculateTax(this.customer.getState(),
amount);
    float shipping =
saleService.calculateShipping(this.customer.getState(),
this.customer.isCapital(), this.customer.getType());
    amount += tax + shipping;

    float cashback = saleService.calculateCashback(this.customer, amount,
this.paymentMethod);

    Sale sale = new Sale(Date.from(Instant.now()), this.customer,
this.products, this.paymentMethod, amount);
    sale.setDiscount(discount);
    sale.setTax(tax);
    sale.setShipping(shipping);
```

```
        sale.setCashback(cashback);

        saleService.addSale(sale);

        System.out.println(sale);
        if (cashback > 0) {
            System.out.printf("Cashback recebido: R$ %.2f%n", cashback);
        }

        return amount;
    }
```

Há um mau-cheiro por ser um Método Longo que fere o príncipio da Modularidade. Este método faz várias coisas: calcula o total, aplica descontos, calcula impostos, calcula frete, etc. Seria ideal dividi-lo em métodos menores e mais coesos.

- Na classe SaleService método calculateShipping

```
    public float calculateShipping(RegionType region, boolean isCapital,
CustomerType customerType) {
        if(customerType.name().equals(CustomerType.PRIME.name())) return
0;

        float shippingPrice = switch (region) {
            case CENTRO_OESTE, SUL -> isCapital ? 10 : 13;
            case NORTE -> isCapital ? 20 : 25;
            case NORDESTE -> isCapital ? 15 : 18;
            case DISTRITO_FEDERAL -> 5;
            case SUDESTE -> isCapital ? 7 : 10;
            default -> 0;
        };

        return isSpecial(customerType) ? shippingPrice * 0.7f :
shippingPrice;
    }
```

O mesmo também fere o principio de modularidade, acumulando responsabilidade que poderia estar em outro método. O trecho de código relacionado ao switch poderia estar encapsulado em uma novo método, simplificando e modularizando esta funcionalidade que retorna o preço do frete.