Relatório da Simulação de Sprint Scrum A2

Engenharia de Software

Gerardo Mikael do Carmo Pereira
Thiago Franke Melchiors
Yonathan Rabinovici Gherman

Professor: Rafael de Pinho

Monitor: Luís Henrique Domingues Bueno

RIO DE JANEIRO
2024

Conteúdo

1 Introdução

Este relatório tem como objetivo apresentar e justificar as escolhas de padrões de design adotadas pela nossa equipe na implementação da versão final do projeto para a tarefa A2 da disciplina de Engenharia de Software. Ao longo deste documento, detalharemos os padrões de design aplicados ao sistema, explicando como cada um foi integrado à arquitetura do projeto e os impactos que tiveram na implementação final. Além disso, discutiremos as alternativas que foram consideradas, como seria a solução sem a utilização desses padrões e as vantagens que eles trouxeram em termos de modularidade, manutenibilidade e escalabilidade.

2 Padrões de Projeto

2.1 Factory Method e Strategy na classe Report

A primeira modificação realizada foi na classe `ProblemReport` (no diretório `src/classes/problemreport.py`. A versão anterior do código possuía uma classe `ProblemReport` que poderia ser instanciada tanto para reportar erros de máquinas de venda quanto erros do sistema. A diferenciação entre esses tipos era feita de forma explícita nos inputs durante a interação com o usuário. Embora funcional, essa abordagem era pouco modular, difícil de expandir e de manter.

Para melhorar a flexibilidade e modularização, aplicamos o padrão de projeto `Factory Method`, que encapsula a lógica de criação dos objetos. Agora, a classe `ProblemReport` mantém apenas as definições básicas, enquanto a criação de diferentes tipos de relatórios é centralizada na fábrica `ProblemReportFactory`. A fábrica verifica se o tipo de relatório é "system"ou "machine"e cria o objeto correspondente. Isso facilita a escalabilidade, pois novos tipos de problemas podem ser adicionados à fábrica sem alterar a lógica da classe `ProblemReport`. A persistência de dados com `ProblemReportDAO` não precisou ser modificada, já que as alterações focaram na parte memória "efêmera"da aplicação.

Além disso, adotamos o padrão `Strategy` no diretório `src/customer_flow/report_service.py`, com a classe `ProblemReportStrategy`, que serve como uma interface para definir o método `collect_data`. As subclasses `MachineProblemReportStrategy` e `SystemProblemReportStrategy` implementam a lógica específica para coletar os dados de acordo com o tipo de problema. Esse padrão foi essencial porque, antes de criar um relatório, é necessário interagir com o usuário por meio de inputs, e a coleta dos dados varia dependendo do tipo de problema (relacionado a uma máquina ou ao sistema). O uso do `Strategy` desacoplou a lógica de interação com o usuário da criação do relatório, permitindo a definição de diferentes fluxos de entrada para cada tipo de problema.

A função `create_problem_report` escolhe a estratégia apropriada com base na entrada do usuário ('m' para máquina e 's' para sistema), utiliza a estratégia selecionada para coletar os dados do problema e, em seguida, utiliza o `Factory Method` para criar o `ProblemReport`, que é então salvo no banco de dados.

Essa refatoração proporciona uma série de benefícios. A lógica de coleta de dados do usuário é agora separada da lógica de criação do relatório, o que facilita a manu-

tenção e evolução do sistema. No futuro, se for necessário alterar a forma de coletar dados para um tipo específico de problema, isso pode ser feito facilmente ajustando a estratégia correspondente, sem afetar o restante do código. Da mesma forma, novas estratégias podem ser adicionadas sem necessidade de modificações profundas na função `create_problem_report`.

2.2 Factory Method e Strategy na classe Review

A segunda modificação foi na classe `Review` (no diretório `src/classes/review.py`). Análogo ao caso de `Report`, a versão anterior do código de Review possuía uma classe genérica, usada tanto para avaliações de produtos quanto de máquinas e a diferenciação era feita nos inputs durante a interação com o usuário. Era funcional, mas tinha os problemas já citados.

Da mesma forma, aplicamos o padrão de projeto `Factory Method`, que centraliza a criação de objetos `Review` através da classe `ReviewFactory`. A fábrica decide, com base no tipo de avaliação ('product' ou 'machine'), qual tipo específico de `Review` deve ser instanciado, seja `ProductReview` ou `MachineReview`. Isso facilita a adição de novos tipos de avaliações no futuro sem a necessidade de modificar a lógica principal da classe `Review`.

Juntamente com o `Factory Method`, adotamos o padrão `Strategy` para lidar com os diferentes fluxos de entrada para coletar os dados necessários para as avaliações. A interface `ReviewStrategy` define o método `collect_data`, que é implementado pelas classes filhas `ProductReviewStrategy` e `MachineReviewStrategy`, responsáveis por coletar os dados específicos dependendo do tipo de avaliação.

A função `create_review` escolhe a estratégia adequada com base no tipo de avaliação escolhido pelo usuário ('p' para produto e 'm' para máquina), coleta os dados usando a estratégia selecionada e, em seguida, cria a avaliação com a `ReviewFactory`. O objeto `Review` é então salvo no banco de dados.

Essa refatoração trouxe os mesmos benefícios mencionados no caso dos relatórios: a separação das lógicas de coleta de dados e criação das avaliações, facilitando futuras manutenções e expansões. Novos tipos de avaliações ou estratégias de coleta de dados podem ser facilmente adicionados sem a necessidade de alterar a função `create_review`, mantendo o código modular e de fácil manutenção.

2.3 Factory Method na classe User

A aplicação do padrão `Factory Method` também foi estendida para a criação de usuários no sistema (no diretório `src/classes/user.py`). Na versão anterior, as classes `Administrator`, `Customer` e `Seller` já herdavam de `User`, mas a criação das instâncias era feita de forma explícita nas funções `create_customer_account` e `create_seller_account`. A criação de um `Administrator` não está explicitamente implementada, pois optamos por não criar contas de administrador diretamente.

Embora a estrutura de herança já estivesse presente, a abordagem anterior não seguia a implementação clássica do padrão `Factory Method`, pois a criação das instâncias de usuário não era centralizada e estava espalhada por várias funções.

Para reparar isso e melhorar a modularização, centralizamos a lógica de criação dos diferentes tipos de usuários em uma classe `UserFactory`. A `UserFactory` agora é responsável por criar o tipo de usuário adequado (`Customer` ou `Seller`) com suas características e permissões específicas. Através dessa abordagem, a criação de usuários se torna mais flexível, modular e facilmente extensível. Essa refatoração melhora a manutenção do código, pois, a responsabilidade de criar um usuário foi isolada em um único ponto, facilitando futuras modificações ou adições de novos tipos de usuários diretamente em `UserFactory`, sem precisar alterar outras partes do sistema.

2.4 Object Pool Pattern nas conexões com o banco de dados

Inicialmente, cogitamos o uso do padrão `Singleton` para gerenciar as conexões com o banco de dados, considerando a simplicidade e a eficiência para um sistema com poucos usuários simultâneos. No entanto, ao analisar a escalabilidade e o impacto do `Singleton` em sistemas com múltiplos usuários, percebemos que ele poderia gerar problemas de concorrência, como gargalos e bloqueios, especialmente se a aplicação fosse expandir no futuro.

Após uma pesquisa por outras opções, decidimos então adotar o `Object Pool`, que permite gerenciar um número fixo de conexões abertas e reutilizáveis. Este padrão oferece maior flexibilidade, pois possibilita que várias conexões sejam usadas simultaneamente, sem problemas de concorrência, e melhora a performance ao evitar a criação e destruição constantes de conexões.

Criamos uma classe `DBConnectionPool` (em `src/db_initializer.py`) para gerenciar as conexões com o banco de dados. O pool mantém um número máximo de conexões simultâneas, que são distribuídas conforme necessário. Cada vez que uma função precisa acessar o banco de dados, ela obtém uma conexão do pool e a devolve após o uso, garantindo que as conexões sejam reutilizadas de forma eficiente.

Embora a aplicação ainda esteja em um ambiente local e sem problemas de concorrência, o uso do `Object Pool` prepara o sistema para um possível crescimento futuro. Ele garante que o sistema seja escalável e capaz de lidar com múltiplos acessos simultâneos, evitando problemas de desempenho e permitindo que o sistema seja facilmente adaptado para um ambiente de produção com maior carga de usuários.

Essa abordagem aumenta a flexibilidade, pois no futuro, se necessário, o número de conexões pode ser ajustado alterando-se o parâmetro `max_connections` para um valor maior para atender a uma demanda simultânea maior sem impactar a performance da aplicação.

2.5 Observer em Favorite Products e Machines

O padrão Observer (encontrado em `src/database/favorite_product_machine_dao.py`) foi utilizado para gerenciar a relação de muitos para muitos entre usuários e seus produtos e máquinas de venda favoritas. O emprego desse padrão foi necessário para que os usuários fossem automaticamente notificados sobre alterações nos itens que favoritaram, como quando um produto sai de estoque.

A escolha do padrão Observer se deu pela necessidade de notificar automaticamente

os usuários (observadores) quando houvesse mudanças nos produtos ou máquinas de venda (sujeitos) que eles haviam favoritado. O padrão possibilita o desacoplamento entre as entidades envolvidas, já que o sujeito (produto ou máquina) não precisa saber quem são os observadores, apenas notifica aqueles interessados quando há uma mudança de estado.

Além disso, o padrão facilita a escalabilidade do sistema, permitindo que novos tipos de notificações (como e-mails ou SMS) sejam facilmente implementados no futuro, sem modificar o código das classes de favoritos.

A seguir, descreveremos com mais detalhes como isso funciona:

- Classe `Subject`: É uma classe base que gerencia os métodos de adicionar (`attach`), remover (`detach`) e notificar (`notify_observers`) os observadores. No entanto, devido à persistência dos dados, os métodos não manipulam uma lista de observadores em memória (como é a versão clássica), mas sim no banco de dados.
- `FavoriteProductDAO` e `FavoriteMachineDAO`: Estas classes são responsáveis pela persistência da relação entre os usuários e os itens favoritados. O método `attach` registra a relação no banco de dados, enquanto o `detach` a remove.
- Métodos `is_favorite_product` e `is_favorite_machine`: Verifica se um produto ou máquina de venda estão nas listas de favoritos de um usuário. Essa funcionalidade é ativada quando o usuário solicita favoritar ou 'desfavoritar' um item.
- Método `get_observers`: Ao invés de retornar uma lista de observadores em memória, o método `get_observers` consulta o banco de dados para retornar todos os usuários que favoritaram um produto ou uma máquina. item Método `notify_observers`: Quando um item tem seu status alterado, o método `notify_observers` envia notificações para os usuários que o favoritaram. Essas notificações personalizadas são registradas no banco de dados e podem ser visualizadas na tela de notificações do usuário.

Apesar de a estrutura básica do Observer ter sido mantida, houve modificações importantes para adaptar o padrão à necessidade de persistência em banco de dados. Em vez de usar listas em memória para gerenciar os observadores, foi necessário realizar operações de banco de dados para garantir que as relações de favoritos fossem corretamente registradas e as notificações fossem enviadas aos usuários de forma escalável.

Com a aplicação do padrão Observer, conseguimos criar um sistema eficiente e desacoplado para gerenciar os favoritos dos usuários, além de permitir que as notificações fossem enviadas automaticamente a cada alteração relevante. Isso garante a flexibilidade do sistema, pois as notificações podem ser facilmente expandidas para novos canais, como e-mail ou SMS, e a escalabilidade, já que a persistência no banco de dados permite que o sistema lide com grandes volumes de dados sem sobrecarregar a memória.

2.6 Command na gestão do carrinho de compras

O padrão de projeto Command (em `src/classes/command.py` e `src/customer_flow/transaction_service.py`) foi aplicado ao sistema de carrinho de compras para desacoplar a lógica de execução das ações do carrinho da interface do usuário. Antes da implementação do padrão Command, a manipulação de ações no carrinho de compras estava diretamente atrelada aos métodos responsáveis pela lógica de negócio, como a adição de produtos no carrinho, além do processo de finalização da compra. Essa implementação inicial apresentava alguns problemas relacionados ao controle de estado e à gestão de múltiplas ações no carrinho, o que gerava dificuldade no rastreamento e controle das operações realizadas.

A introdução do padrão Command visa melhorar o controle sobre as ações executadas, proporcionando um maior desacoplamento entre os objetos responsáveis pela execução das ações e a interface que interage com o usuário. Na aplicação do padrão Command, o papel de cada comando foi bem definido: a `AddToCartCommand`, `RemoveFromCartCommand` e `CheckoutCommand` encapsulam, respectivamente, a ação de adicionar um produto ao carrinho, remover um produto do carrinho e finalizar a compra. O `ShoppingCartInvoker` é o objeto que armazena e executa esses comandos.

Com a utilização do padrão Command, cada operação realizada no carrinho de compras passou a ser tratada como um comando autônomo. No caso da adição de produtos ao carrinho, por exemplo, antes da aplicação do padrão, a verificação do estoque e a atualização do carrinho eram feitas diretamente dentro da lógica de controle do fluxo de compras. Isso gerava um alto acoplamento entre a lógica de interação com o usuário e as operações de negócio. A introdução da classe `AddToCartCommand` desacoplou essa lógica, tornando a adição de produtos ao carrinho mais modular. Agora, o comando responsável pela adição de produtos à lista de compras recebe o produto, a quantidade e o carrinho como parâmetros, realizando a verificação da disponibilidade de estoque e atualizando o carrinho de forma autônoma.

Além disso, o padrão Command oferece uma flexibilidade adicional, pois permite a adição de novos comportamentos de forma simples e sem a necessidade de alterar o código existente. A implementação da classe `RemoveFromCartCommand`, por exemplo, foi facilitada pela estrutura do padrão (observe, no entanto, que a classe existe, mas ainda não foi conectada ao sistema na presente versão do aplicativo). A operação de finalização de compra foi igualmente desacoplada por meio do `CheckoutCommand`, que encapsula a lógica de pagamento, verificação do saldo do cliente, dedução do valor da compra e atualização de estoque.

Em relação à implementação anterior, a principal mudança é o controle das ações por meio de comandos, sem a necessidade de interagir diretamente com a lógica de negócio. Antes, a lógica de atualização do carrinho e de execução do pagamento estava misturada com a interface de usuário e o controle de fluxo. Com o uso do padrão Command, a lógica de manipulação do carrinho de compras foi encapsulada dentro das classes de comando, como `AddToCartCommand` e `CheckoutCommand`, e a interface do usuário passou a interagir com o `ShoppingCartInvoker`, que gerencia a execução dos comandos. Nesse sentido, os beneficios da inclusão desse padrão incluem a modularização do sistema e o desacoplamento entre a interface do usuário e a lógica de negócios; possibilidade de adicionar múltiplos comandos ao invocador

`ShoppingCartInvoker` sem causar duplicação de lógica ou violação da ordem de execução ou invocar apenas o comando atual proporcionou uma experiência de uso mais fluida e sem erros