Instituto Politécnico de Viseu

Escola Superior de Tecnologia e Gestão de Viseu

Departamento de Informática

Relatório de Engenharia de Software II

Trabalho Padrões de Desenho

Realizado por:

Breno Salles

Docente:

Carlos Cunha

Viseu, 2020

Instituto Politécnico de Viseu

Escola Superior de Tecnologia e Gestão de Viseu

Departamento de Informática

Relatório de Engenharia de Software II

Licenciatura em Engenharia Informática

Trabalho Padrões de Desenho

Ano Letivo 2019/2020

Realizado por:

Breno Salles, 17035

Docente:

Carlos Cunha

Viseu, 2020

Índice

Introdução 1

Desenvolvimento 2

Resolução 1 2

Para que serve o *Singleton Pattern*? 2

Vantagens do Singleton Pattern? 2

Desvantagens do *Singleton Pattern*? 2

Resolução 2 3

Para que serve o *Composite Pattern*? 3

Vantagens do Composite Pattern? 3

Resolução 3 4

Para que serve o *Bridge Pattern*? 4

Vantagens do *Bridge Pattern*? 4

Desvantagens do *Bridge Pattern*? 5

Resolução 4 6

Para que serve o *Factory Method*? 6

Para que serve o *Object Pool*? 6

Vantagens do *Factory Method*? 7

Desvantagens do *Factory Method*? 7

Vantagens do *Object Pool*? 7

Desvantagens do *Object Pool*? 7

Conclusão 8

Índice de fíguras

Figura 1 - Singleton Pattern aplicado à autenticação 2

Figura 2 - Composite Pattern aplicado aos diferentes níveis de medicação 3

Figura 3 - Bridge Pattern aplicado às diferentes possibilidades de transporte 4

Figura 4 - Factory Method e Object Pool para criação e reutilização de objetos 6

Introdução

Este trabalho tem como objetivo desenhar e desenvolver uma aplicação em UML, com o propósito de sedimentar os conhecimentos sobre *design patterns* que foram dados na primeira fase (de duas) do semestre.

Posteriormente, e para acompanhar o desenho em UML, uma implementação, de todos os padrões desenhados, é necessária. *JAVA* é a linguagem escolhida para esta implementação.

O propósito deste trabalho é o seguinte:

Um aluno empreendedor finalista do curso de Engenharia Informática decidiu criar uma empresa de transporte de medicamentos. Uma das necessidades mais imediatas é criar uma aplicação informática para suportar o seu negócio.

Os requisitos da aplicação são:

1. A **autenticação** dos motoristas deverá ser implementada num objeto global com o *username* e *password* dos condutores.
2. Os medicamentos a transportar poderão estar agrupados por níveis. O motorista pode transportar contentores ou caixas contendo embalagens de medicamentos.
3. O **custo de transporte deve ser calculado** como uma **percentagem do somatório do preço das embalagens** de medicamentos transportados, de duas formas distintas:
    a. 5% do valor dos medicamentos durante períodos normais.
    b. 10% do valor dos medicamentos durante períodos especiais.

    A aplicação deverá estar preparada para aceitar novos esquemas de cobrança.

4. As caixas e os contentores poderão ser reutilizados, pelo que, durante o **processo de criação dos objetos** pertencentes a um destes tipos, deverá ser dada a possibilidade de **reutilizar** um objeto já existente.

Desenvolvimento

Resolução 1

Para que serve o *Singleton Pattern*?

- Deve-se usar *Singleton Pattern* quando uma classe no programa deve apenas conter uma, e uma só, instanciação disponivel para todos os clientes. Por exemplo, um objeto de base de dados disponivel em diversas partes do programa.
- Deve-se usar *Singleton Pattern* quando necessitamos de um controlo mais restrito sobre variáveis globais.

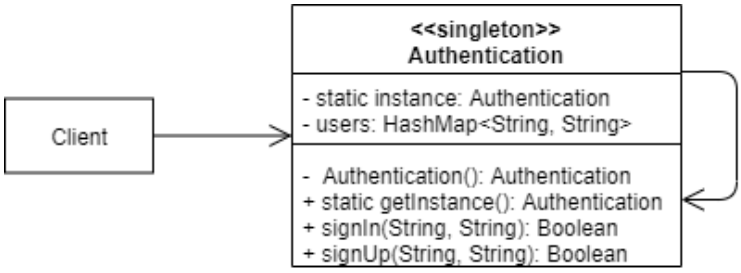

Figura 1 - Singleton Pattern aplicado à autenticação

Vantagens do Singleton Pattern?

- Garantimos apenas uma instância da classe.
- Ganhamos o acesso global a essa instância.
- O objeto *Singleton* apenas é inicializado quando é chamado pela primeira vez (*lazy initialization*).

Desvantagens do *Singleton Pattern*?

- Viola um dos princípios **SOLID** (*Single Responsibility Principle*) uma vez que resolve dois problemas ao mesmo tempo.
- O *Singleton Pattern* pode "mascarar" mau desenho de software.
- É difícil fazer *unit tests* para código que implementa *Singleton Pattern*.

Resolução 2

Para que serve o *Composite Pattern*?

- Devemos usar *Composite Pattern* quando temos de implementar uma estrutura similar à de uma árvore, isto é, ramos que se dividem em outros ramos ou em folhas.
- Devemos usar *Composite Pattern* quando queremos que a aplicação trate quer as folhas, quer os ramos, da mesma maneira, através de uma *interface* comum.

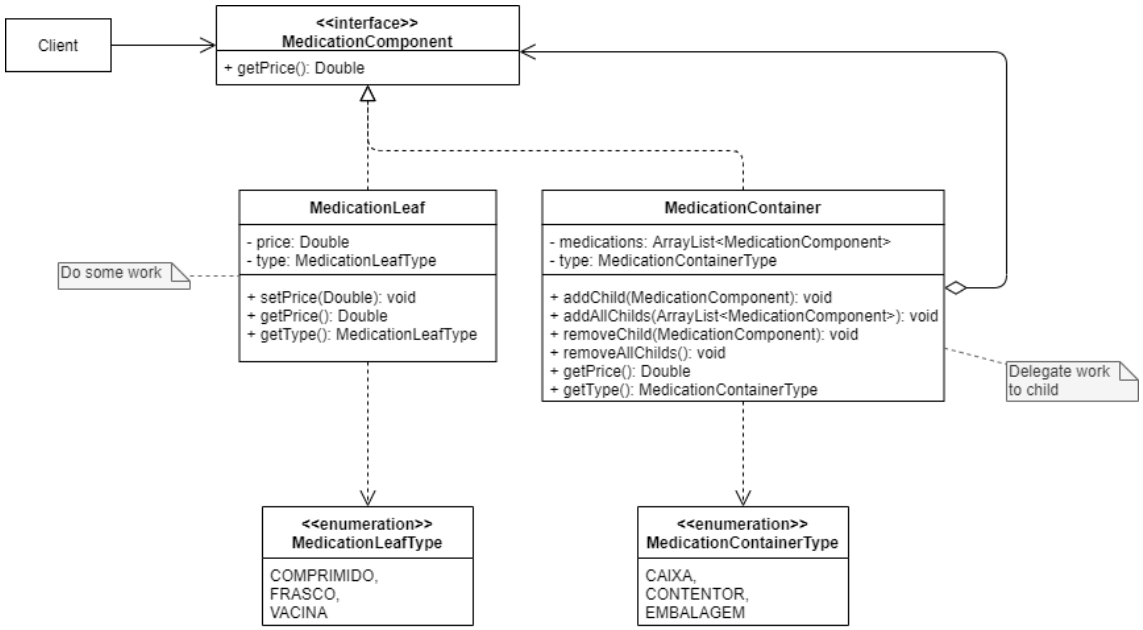

Figura 2 - Composite Pattern aplicado aos diferentes níveis de medicação

Vantagens do Composite Pattern?

- Conseguimos trabalhar com estruturas mais complexas de maneira mais fácil, ao utilizarmos polimorfismo e recursividade.
- Podemos introduzir novos tipos de elementos à aplicação sem "quebrar" código já existente, garantindo assim o *Open/Closed Principle*.

Resolução 3

Para que serve o *Bridge Pattern*?

- Deve-se usar *Bridge Pattern* quando queremos dividir e organizar uma classe monolítica, que possuí várias variações de funcionalidades.
- Deve-se usar *Bridge Pattern* quando queremos garantir que a nossa classe possa expandir ortogonalmente e independentemente em todas as dimensões.
- Deve-se usar *Bridge Pattern* se quisermos mudar de implementações em *runtime*.

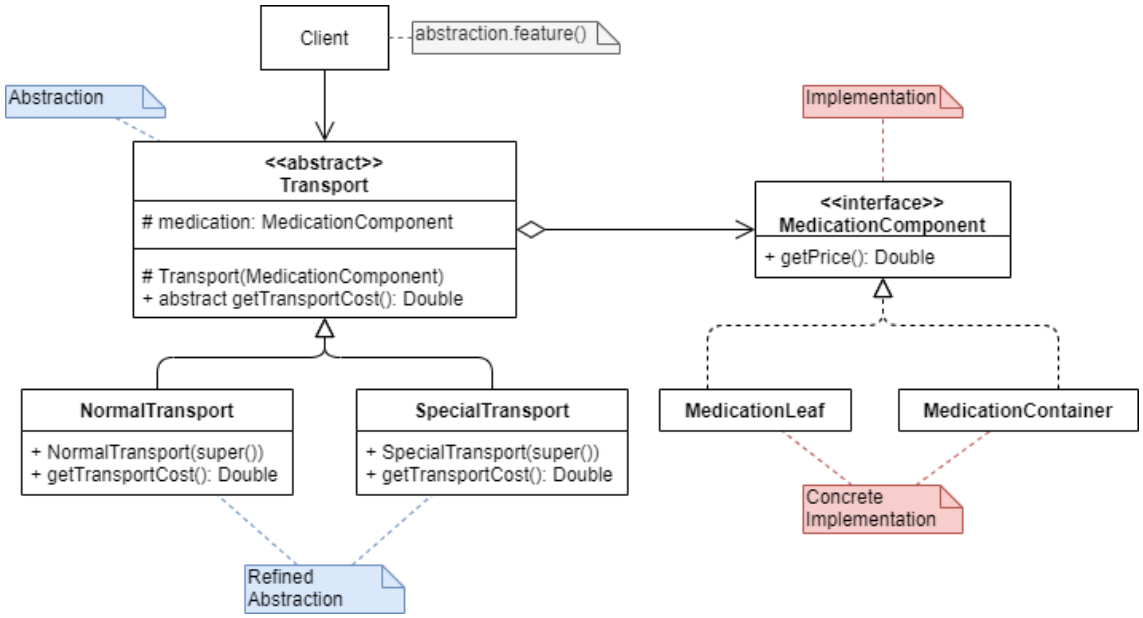

Figura 3 - Bridge Pattern aplicado às diferentes possibilidades de transporte

Vantagens do *Bridge Pattern*?

- Conseguimos criar classes e aplicações independentes da plataforma.
- O cliente trabalha com abstrações de alto nível, não expondo os detalhes da plataforma.
- Podemos introduzir novas abstrações independentes das outras garantindo assim o *Open/Closed Principle*.
- Podemos focar-nos na lógica de alto nível na abstração e nos detalhes da plataforma na implementação, garantindo o *Single Responsibility Principle*.

Desvantagens do *Bridge Pattern*?

- Podemos tornar o código mais complicado ao aplicarmos o *pattern* a uma classe com elevada coesão.

Resolução 4

Para que serve o *Factory Method*?

- Deve-se usar *Factory Method* quando não sabemos de antemão os tipos exatos e dependências dos objetos com que o código deve trabalhar.
- Deve-se usar *Factory Method* quando queremos dar aos utilizadores do nosso código uma maneira de incrementar os seus objetos internos.

Para que serve o *Object Pool*?

- Deve-se usar *Object Pool* quando queremos poupar em recursos do sistema com a reutilização de objetos existentes em vez de estarmos sempre a criá-los.

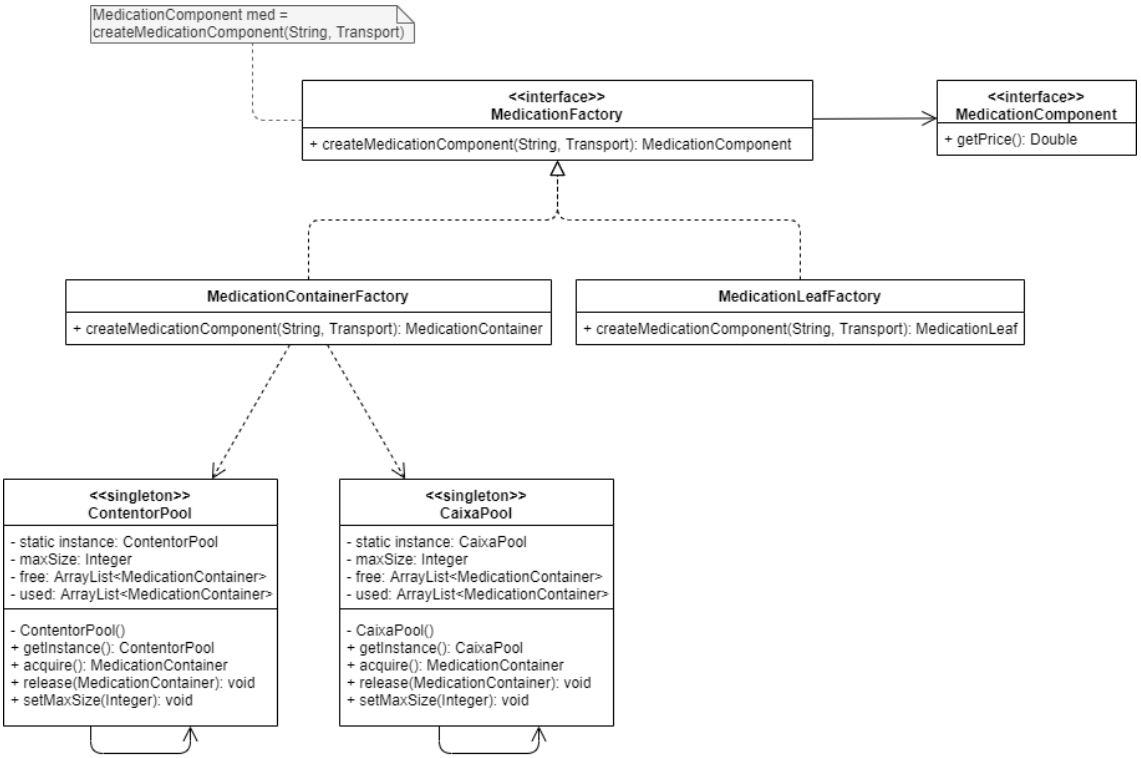

Figura 4 - Factory Method e Object Pool para criação e reutilização de objetos

Vantagens do *Factory Method*?

- Evitamos acoplamento entre o criador e os produtos concretos.
- Preserva o *Single Responsibility Principle* uma vez que podemos mudar a *factory* para uma outra parte do programa.
- Preserva o *Open/Closed Principle* pois podemos introduzir novos objetos sem "quebrar" código existente.

Desvantagens do *Factory Method*?

- O código pode tornar-se mais complexo uma vez que é necessário introduzir novas subclasses que implementam o *pattern*.

Vantagens do *Object Pool*?

- Pode oferecer uma melhoria de performance em situações cujos custo de criação de objetos é elevado e a quantidade de vezes que eles são criados e destruídos é elevada.

Desvantagens do *Object Pool*?

- Existem algumas linguagens em que, o custo de manutenção de uma *Object Pool* é superior ao custo de criação e destruição de objetos.
- Em linguagens que implementam um *Garbage Collector*, pode ser uma desvantagem uma vez que este não consegue eliminar objetos presentes na *pool*, mesmo que estes nunca voltem a ser usados.

Conclusão

Design Patterns ou Padrões de Desenho são soluções comuns para problemas em engenharia de software. Um padrão difere de um algoritmo na medida em que, enquanto um algoritmo é uma equivalente a uma receita de culinária em que seguimos todos os passos à risca, um padrão é uma *blueprint* que podemos personalizar para resolver determinados problemas de *design* na nossa aplicação.

Também são vantajosos porque garantem que não cometemos erros que outros programadores no passado já cometeram, uma vez que, estes padrões são soluções testadas e aprovadas para a resolução de problemas comuns de software.

Com este trabalho foi possível perceber as vantagens de utilizar *Design Patterns* uma vez que, cada implementação de um dos padrões facilmente interagiu com as anteriores, sem recorrer a um *"hack"* para funcionar. Também, e com um estudo associado, percebeu-se que *design patterns* não são a solução milagrosa para todos os problemas em engenharia de software.

Também foram adicionados testes simples para cada um dos *patterns* implementados, sendo que, no futuro, estes deveriam aumentar de modo a ter uma maior *coverage*.