INSTITUTO FEDERAL DA PARAÍBA - IFPB

CURSO SUPERIOR TECNOLÓGICO EM ANÁLISE E DESENVOLVIMENTO DE SISTEMAS

JOZIMAR SOARES DA COSTA

LYNDEMBERG BATISTA NERY

RÔMULO SOARES BEZERRA

RELATÓRIO TÉCNICO DE PADRÕES DE PROJETO

SISTEMA DE GERENCIAMENTO DE ATENDIMENTOS PARA SALÕES DE BELEZA/BARBEARIA

Cajazeiras - PB

2018

JOZIMAR SOARES DA COSTA
LYNDEMBERG BATISTA NERY
RÔMULO SOARES BEZERRA

RELATÓRIO TÉCNICO DE PADRÕES DE PROJETO
SISTEMA DE GERENCIAMENTO DE ATENDIMENTOS PARA SALÕES DE BELEZA/BARBEARIA

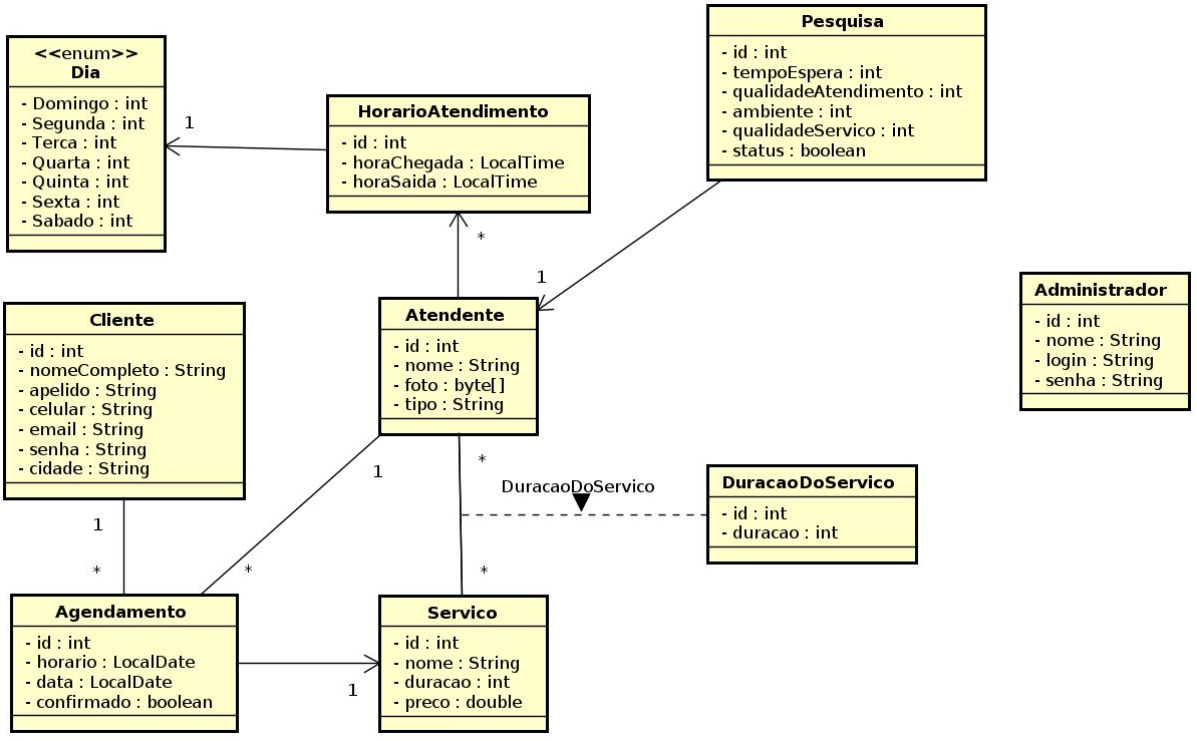
Relatório técnico apresentado como requisito parcial para obtenção de aprovação na disciplina de Padrões de Projeto, no Curso de Análise e Desenvolvimento de Sistemas, no Instituto Federal da Paraíba.

Prof. Diogo Dantas Moreira

Cajazeiras - PB
2018

VISÃO GERAL DO SISTEMA

O Wazbarber é um sistema direcionado tanto à salões de beleza como à barbearias, onde o principal problema a ser solucionado através do mesmo é a possibilidade do agendamento de horários, de modo que evite a utilização da famosa "ordem de chegada", e fazendo com que os clientes não percam muito tempo esperando para serem atendidos. Logo abaixo é possível visualizar a estrutura do sistema a nível de classes.



Fonte: pessoal do autor

PADRÕES UTILIZADOS

O principal objetivo que nos foi imposto para realizar o desenvolvimento deste projeto é a utilização de padrões, de modo que os mesmos proporcionem uma melhor flexibilidade de código. Sendo assim utilizamos os seguintes padrões para alcançar este objetivo da melhor forma:

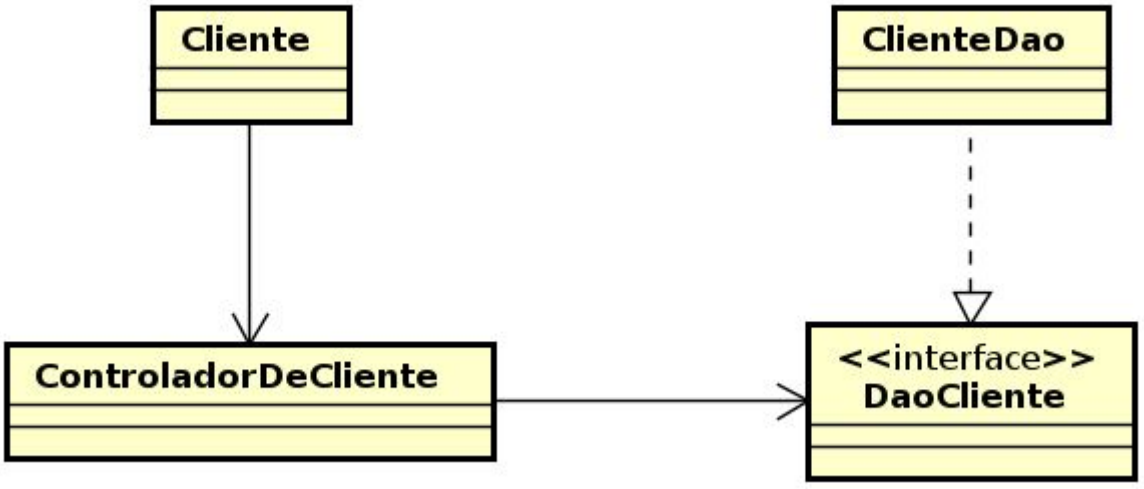
- Injeção de Dependência
- Builder
- Observer
- Singleton

Nas próximas seções destacamos de forma mais detalhada os padrões utilizados em nosso projeto, bem como os benefícios que os mesmos proporcionaram.

PADRÃO INJEÇÃO DE DEPENDÊNCIA

Neste projeto utilizamos a injeção de dependência através da anotação @Inject para instanciar componentes ejbs que foram usados. Para que a injeção funcione, devemos usá-la nos pontos que queremos uma injeção automática, sejam construtores, setters ou atributos privados. Essa anotação é usada em conjunto com o CDI (Context Dependency Injection) que é a especificação do Java EE responsável por cuidar da parte de injeção de dependências. Ela está incluída também na especificação JSR346, onde esta por sua vez possui anotações relacionadas à injeção de dependências no pacote `javax.inject`. Neste caso utilizamos a anotação para injetar objetos concretos em abstrações (interfaces) que representam as implementações dos DAOs (Data Access Object) nas classes controladoras, de modo que, estes controladores, por sua vez, sejam responsáveis pelo gerenciamento da interface com o usuário.

Como exemplo temos que `ControladorDeCliente.java` que será de escopo de requisição, por estar anotado com `@RequestScoped`, onde necessariamente deve ser do pacote `javax.enterprise.context` para pertencer ao contexto do CDI sendo usado em conjunto com a anotação `@Named` que faz com que a classe controladora seja um bean gerenciado pelo contexto do CDI, tornando a injeção de dependência - DI - possível. Logo mais abaixo está o diagrama representando o exemplo citado acima.



Fonte: pessoal do autor

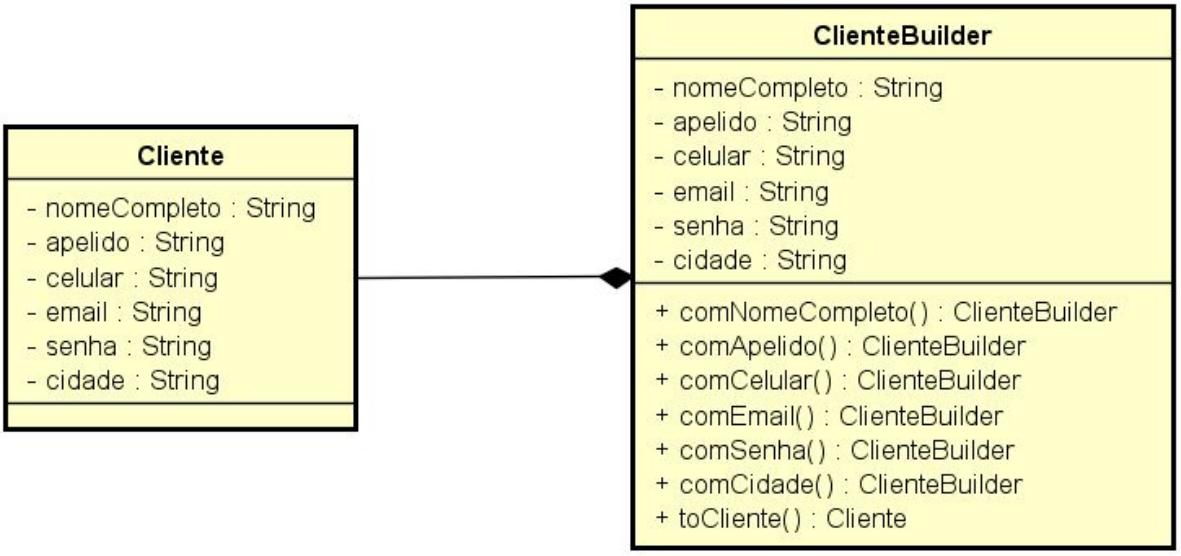
BENEFÍCIOS ENCONTRADOS

- Mantivemos baixo acoplamento entre diferentes camadas do sistema: o consumidor do serviço não depende diretamente da classe concreta, mas sim está dependendo da interface que é injetada, uma vez que utilizamos abstrações para prover os serviços.
- Amenizou a responsabilidade do programador: nesse modo o responsável por injetar esse recurso será o container de injeção, eliminando a carga do programador.
- Eliminou a necessidade do desenvolvedor criar uma instância programaticamente (`new Instance()`).
- Facilitou a manutenção do sistema, fazendo com que as manutenções em módulos, classes de implementação, não afetassem o restante do sistema.
- Códigos mais legíveis e limpos, o que tornou mais fácil a compreensão do sistema como um todo.

PADRÃO BUILDER

Em nosso projeto há uma classe chamada `Cliente` que possui uma série de atributos, dentre os quais a maioria deles são obrigatórios, no entanto nesta classe podemos adicionar também atributos opcionais. Decidimos então utilizar o padrão

Builder que proporcionou um melhor controle sobre a construção do objeto cliente. Um exemplo deste controle proporcionado está no cadastro de cliente, onde foram utilizados todos os seus atributos, podendo escolher criar o objeto com ou sem atributos opcionais. Abaixo demonstramos um diagrama de classes representando o uso do padrão no nosso sistema.



Fonte: pessoal do autor

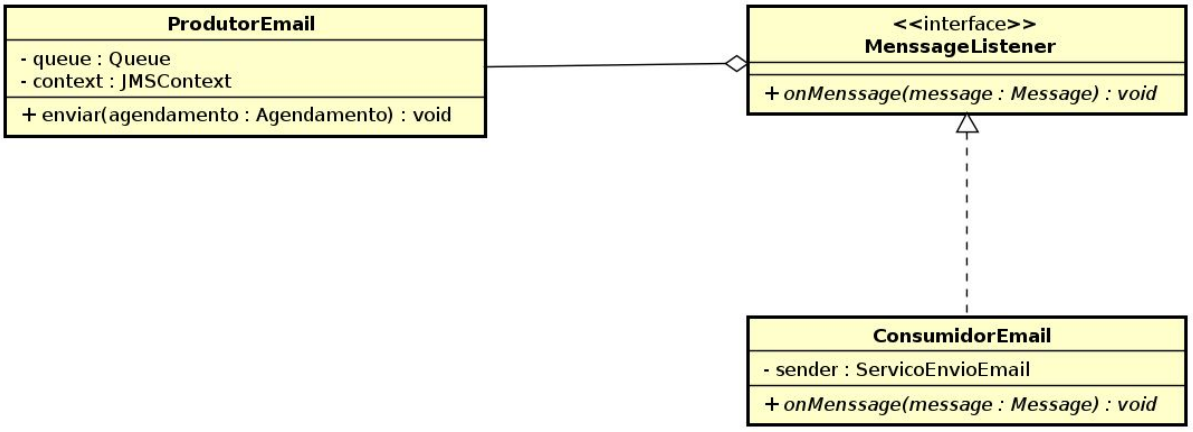
BENEFÍCIOS ENCONTRADOS

- Possibilitou uma maior flexibilidade com relação ao manuseio dos atributos da classe, já que a construção deste objeto foi separado da sua representação e para cada atributo da classe foi criado um método construtor.

PADRÃO OBSERVER

O padrão Observer é bem representado no nosso projeto, como mostra o diagrama abaixo. Seguindo a mesma estrutura convencional do Observer, o nosso projeto se utiliza de uma Classe `ProdutorEmail`, de uma `ConsumidorEmail` e da interface `MessageListener`. Usamos o JMS, que por sua vez acaba

implementando a ideia de Observer, porém acrescido de um outro padrão chamado Mediator. O `ProdutorEmail` acaba fazendo o papel de subject ou objeto interessante: é o objeto que será observado - ao notificar um agendamento para a queue; o MessageListener é a classe abstrata possuidora do método que será usado pela classe concreta para recuperar a mensagem enviada (o agendamento). O `ConsumidorEmail` tem o papel de observador ou ouvinte, permanece à espera de uma notificação para a realização de uma tarefa, que é enviar email para o cliente do agendamento.



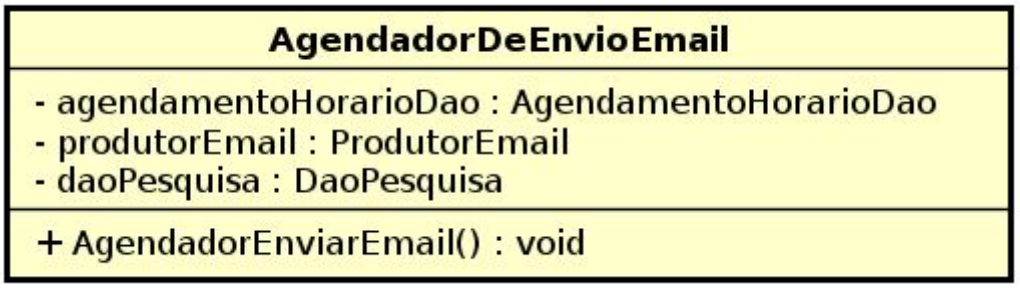
Fonte: pessoal do autor

BENEFÍCIOS ENCONTRADOS

- Reduziu o acoplamento entre as classes relacionadas: o acoplamento entre `ProdutorEmail` e `ConsumidorEmail` é mínimo, já que o observer implementa uma abstração simples. O subject não conhece nenhuma classe concreta de observer.
- O `ConsumidorEmail` é avisado de qualquer mudança de estado do objeto de mensagem - agendamento (objeto interessante) - para execução de seu serviço.

PADRÃO SINGLETON

Explanando o padrão: este padrão garante a existência de apenas uma instância de uma classe, durante o ciclo de vida da aplicação, mantendo um ponto global de acesso ao seu objeto. Nosso projeto faz uso de dois singletons, não programaticamente, mas por uso de anotação de framework. Para exemplificar o uso desse padrão vamos usar o singleton do diagrama abaixo. O singleton abaixo foi usado apenas com o objetivo de enviar emails programados. Esse singleton representa a classe `AgendadorDeEnvioEmail` anotada com `@Singleton`, possuindo o método de envio de email (`agendadorEnviarEmail()`). Pode-se perceber que não se há o método estático `getInstance()` ou Construtor ou atributo do mesmo tipo da classe, como no padrão Singleton normalmente, pois, o próprio contêiner se responsabiliza de fazer essa tarefa de instanciação e manter um único objeto por aplicação quando a mesma é "*startada"*.



Fonte: pessoal do autor

BENEFÍCIOS ENCONTRADOS

- Permitiu iniciar o serviço de envio de e-mails a partir do momento em que a aplicação foi executada.
- Não necessitamos de uma nova instância de objeto a cada momento que quiséssemos realizar o serviço programado.
- Poupou recurso: Uma única instância na aplicação poupa recurso e simplifica o trabalho já que tal instância pode servir inúmeras vezes, como se é o desejado.