**Roteiro 03**

Os roteiros a serem desenvolvidos a seguir visam trazer a percepção da evolução no processo de desenvolvimento de software. Por isso, iremos criar um projeto com vários pacotes, onde cada pacote representa a evolução da implementação deste projeto.

Neste caso iremos explorar os Padrões de Projeto do GoF (Gang of Four), e neste roteiro iremos trabalhar com o **Observer**.

O **Padrão Observer** define uma dependência de um para muitos entre objetos, de modo que quando um objeto muda seu estado, todos seus dependentes são notificados e atualizados automaticamente.

As imagens abaixo tentam ilustrar de forma lúdica esta definição do padrão Observer. Temos a relação de dependência entre um objeto e vários objetos “observadores”.

Do lado esquerdo da imagem temos essa relação em forma de requisições realizadas em um certo intervalo de tempo. Não se sabe ao certo o intervalo de tempo adequado para que essas requisições sejam atendidas e que todos os “observadores” sejam atualizados. Não se sabe nem da necessidade de serem notificados nem quando serão notificados.

Já do lado direito da imagem todos os “observadores” serão atualizados apenas quando algo acontecer e quando realmente for necessário que todos sejam notificados.

Diagrama

O conteúdo gerado por IA pode estar incorreto.

Uma imagem contendo Texto

O conteúdo gerado por IA pode estar incorreto.

Abaixo temos o modelo formal do padrão Observer. Em um modelo mais genérico a classe Subject pode ser uma classe Abstrata ou uma Interface.

Diagrama

O conteúdo gerado por IA pode estar incorreto.

No cenário a seguir vamos detalhar melhor o desenvolvimento deste padrão de projeto.

**Cenário:**

Neste cenário precisamos resolver um problema comum na maioria das empresas que é a falha na comunicação entre funcionários, clientes, fornecedores e etc. Precisamos desenvolver um sistema de notificações para que a empresa comunique seus eventos, promoções, transmita avisos aos seus funcionários e etc.

Já temos parte do sistema desenvolvido, mas trata apenas da comunicação com seus clientes.

**Pacote : roteiro3.parte1**

1 – Dê sequência ao mesmo projeto no NetBeans chamado **PADROESroteiros**

2 – Como foi explicado no cenário inicial, temos uma modelagem inicial que trata da comunicação com os clientes da empresa. Por isso, crie as classes abaixo no pacote **roteiro3.parte1**.

Temos um modelo bem simples de “Newsletter” para mandar avisos com mensagens de ofertas, promoções, e coisas deste tipo para os clientes via email.

Diagrama

O conteúdo gerado por IA pode estar incorreto.

package roteiro3.parte1;

public class Cliente {

private String nome;

private String email;

public Cliente(String nome, String email) {

this.nome = nome;

this.email = email;

}

public void receberNotificacao(String mensagem) {

System.out.println("----------- NOTIFICACAO CLIENTE --------------------------------");

System.out.println("Email enviado para " + nome + " (" + email + ")");

System.out.println("Mensagem: " + mensagem);

System.out.println("-------------------------------------------");

}

}

package roteiro3.parte1;

import java.util.ArrayList;

public class Newsletter {

private ArrayList<Cliente> clientes;

public Newsletter() {

this.clientes = new ArrayList<>();

}

public void adicionarCliente(Cliente cliente) {

clientes.add(cliente);

}

public void removerCliente(Cliente cliente) {

clientes.remove(cliente);

}

public void enviarMensagem(String mensagem) {

for (Cliente cliente : clientes) {

cliente.receberNotificacao(mensagem);

}

}

}

package roteiro3.parte1;

public class TesteNotificacao {

public static void main(String[] args) {

Newsletter newsletter = new Newsletter();

Cliente cli01 = new Cliente("Cliente Jose", "jose@empresa.com");

Cliente cli02 = new Cliente("Cliente Maria", "maria@empresa.com");

newsletter.adicionarCliente(cli01);

newsletter.adicionarCliente(cli02);

newsletter.enviarMensagem("Oferta Especial !");

}

}

3 – Observe que este início de sistema ainda não atende os requisitos do cenário apresentado. Ainda não existe nada relacionado a funcionários e fornecedores. Uma solução que parece ser simples para evoluir este sistema é “replicar” a classe Cliente, criando as classes Funcionário e Fornecedor. Afinal, elas também vão receber as notificações. Essa solução parece ser interessante ? Consegue ver problemas nesta modelagem ?

**Pacote : roteiro3.parte2**

Tentaremos agora fazer a refatoração do código de uma forma bem flexível e que permita atender aos requisitos, incluindo outros “observadores”. Neste caso, seriam os funcionários e fornecedores.

1 – Dê sequência ao mesmo projeto no NetBeans chamado **PADROESroteiros**

2 – Crie o pacote **roteiro3.parte2** e copie as classes do pacote anterior.

3 – Para tornar a modelagem o mais flexível possível e adotar os princípios do SOLID, crie a interface **Observer** conforme segue abaixo

package roteiro3.parte2;

public interface Observer {

void update(String mensagem);

}

4 – Precisamos agora refatorar as classes existentes neste pacote para que agora trabalhem com a interface Observer. Comece pela classe Cliente.

* A classe Cliente agora deve implementar a interface Observer
* Ao implementar a classe Observer deveremos ter o Override do método update na classe Cliente
  + A implementação deste método deve basicamente substituir o método receberNotificacao já existente na classe Cliente

package roteiro3.parte2;

public class Cliente implements Observer {

\*\*\*\*

\*\*\*\*

\*\*\*\*

~~public void receberNotificacao(String mensagem) {~~

~~System.out.println("----------- NOTIFICACAO CLIENTE --------------------------------");~~

~~System.out.println("Email enviado para " + nome + " (" + email + ")");~~

~~System.out.println("Mensagem: " + mensagem);~~

~~System.out.println("-------------------------------------------");~~

~~}~~

@Override

public void update(String mensagem) {

System.out.println("----------- NOTIFICACAO CLIENTE --------------------------------");

System.out.println("Email enviado para " + nome + " (" + email + ")");

System.out.println("Mensagem: " + mensagem);

System.out.println("-------------------------------------------");

}

}

5 – Vamos agora refatorar a classe Newsletter. Veja que existe um forte acoplamento entre as classes Newsletters e Cliente. A interface Observer foi criada para reduzir este acoplamento e tornar a modelagem mais flexível. Neste sentido, Newsletter deve possuir uma lista de observers e não de clientes. Faça os ajustes indicados abaixo

package roteiro3.parte2;

import java.util.ArrayList;

public class Newsletter {

private ArrayList<Observer> observers;

public Newsletter() {

REFATORAR

}

public void adicionarObserver(Observer observers) {

REFATORAR

}

public void removerObserver(Observer observers) {

REFATORAR

}

public void enviarMensagem(String mensagem) {

REFATORAR

}

}

}

6 – Faça os ajustes e testes necessários na classe TesteNotificacao

7 – Qual princípio do SOLID estamos aplicando nesta refatoração ?

**Pacote : roteiro3.parte3**

1 – No mesmo projeto crie o pacote roteiro3.parte3

2 – Copie as classes do pacote anterior

3 – Vamos agora evoluir o projeto criando as classes **Funcionario** e **Fornecedor.**

Estas classes são muito parecidas com a classe Cliente, mas possuem outros atributos.

* Funcionario e Fornecedor também devem implementar a interface Observer
* Atributos de Funcionario:
  + nome, email, cargo
* Atributos de Fornecedor:
  + nome, email, cnpj

Abaixo temos os respectivos construtores

public Funcionario(String nome, String email, String cargo) {

this.nome = nome;

this.email = email;

this.cargo = cargo;

}

public Fornecedor(String nome, String email, String cnpj) {

this.nome = nome;

this.email = email;

this.cnpj = cnpj;

}

Além destas diferenças, cada uma das classes deve implementar o método update, pois tem origem na interface Observer. A implementação deste método é idêntica entre as classes. Altere apenas a primeira linha do método de cada classe apenas para diferenciamos durante a execução.

System.out.println("----------- NOTIFICACAO CLIENTE --------------------------------");

System.out.println("----------- NOTIFICACAO FUNCIONARIO --------------------------------");

System.out.println("----------- NOTIFICACAO FORNECEDOR --------------------------------");

4 – Faça os devidos teste utilizando a classe TesteNotificacao abaixo.

Observe que agora temos vários “observers”, utilizando uma modelagem flexível independente das entidades Cliente, Funcionario e Fornecedor.

package roteiro3.parte3;

public class TesteNotificacao {

public static void main(String[] args) {

Newsletter newsletter = new Newsletter();

Observer cli01 = new Cliente("Cliente Jose", "jose@empresa.com");

Observer cli02 = new Cliente("Cliente Maria", "maria@empresa.com");

Observer func01 = new Funcionario("Funcionario Pedro", "pedro@empresa.com","Diretor");

Observer fornecedor01 = new Fornecedor("Funcionario Pedro", "pedro@empresa.com","111222333666777");

newsletter.adicionarObserver(cli01);

newsletter.adicionarObserver(cli02);

newsletter.adicionarObserver(func01);

newsletter.adicionarObserver(fornecedor01);

newsletter.enviarMensagem("Oferta Especial !");

}

}

5 – Os ajustes indicados no item 4, buscam estar coerente a que princípio do SOLID ?

**Pacote : roteiro3.parte4**

**Evolução do Cenário :**

Precisamos flexibilizar agora forma de notificação, onde cada observador pode indicar a forma como será notificado. São elas :

* Email
* SMS
* WhatsApp

Neste cenário iremos adorar o padrão **Strategy** associado ao padrão **Observer**

1 – No mesmo projeto crie o pacote roteiro3.parte4

2 – Copie as classes do pacote anterior

3 – Crie a interface NotificacaoStrategy

package roteiro3.parte4;

public interface NotificacaoStrategy {

void enviarMensagem(String destinatario, String mensagem);

}

4 – Vamos agora criar as classes concretas para cada estratégia de notificação (**NotificacaoEmail**, **NotificacaoSMS**, **NotificacaoWhatsApp**). Cada estratégia deve implementar a interface NotificacaoStrategy

package roteiro3.parte4;

public class NotificacaoEmail implements NotificacaoStrategy{

@Override

public void enviarMensagem(String destinatario, String mensagem) {

System.out.println("------------------------------------------------------ ");

System.out.println("Email enviado para " + destinatario + ": \n" + mensagem);

}

}

Faça o mesmo para as classes NotificacaoSMS, NotificacaoWhatsApp. Mude apenas a mensagem abaixo para facilitar a identificação :

System.out.println("SMS enviado para " + destinatario + ": \n" + mensagem);

System.out.println("WhatsApp enviado para " + destinatario + ": \n" + mensagem);

5 – Devemos agora refatorar cada observer (Cliente, Fornecedor e Funcionario). Cada um deles deve possuir o atributo :

private NotificacaoStrategy estrategiaNotificacao;

E no construtor de cada observer deve ser passado como parâmetro a estratégia de notificação do observer.

6 – Para facilitar a identificação de cada tipo de observador vamos criar o método abaixo na interface Observer :

package roteiro3.parte4;

public interface Observer {

void update(String mensagem);

String getTipoObserver();

}

Assim cada observer deve implementar este método com o tipo correspondente. Abaixo temos o exemplo do observer Cliente:

@Override

public String getTipoObserver() {

return "CLIENTE";

}

O mesmo deve ser feito para os outros observers

return "FUNCIONARIO";

return "FORNECEDOR";

Em tese este método seria desnecessário, mas foi utilizado aqui apenas para facilitar a visualização nos testes.

7 – Precisamos agora refatorar o método **update** de **cada observer**, contemplando todas as refatorações e o uso do padrão Strategy , conforme o código abaixo

public void update(String mensagem) {

String destinatario = (estrategiaNotificacao instanceof NotificacaoEmail) ? email : telefone;

StringBuilder mensagemFormatada = new StringBuilder();

mensagemFormatada.append("---------------NOTIFICACAO " + getTipoObserver() + "------------------\n");

mensagemFormatada.append("Notificação enviada para " + nome + " (" + destinatario + ")\n");

mensagemFormatada.append(mensagem+"\n");

mensagemFormatada.append("-----------------------------------------------------------------------\n");

this.estrategiaNotificacao.enviarMensagem(destinatario, mensagemFormatada.toString());

}

8 – Faça os devidos testes usando a classe TesteNotificacao conforme o código abaixo

package roteiro3.parte4;

public class TesteNotificacao {

public static void main(String[] args) {

Newsletter newsletter = new Newsletter();

NotificacaoStrategy email = new NotificacaoEmail();

NotificacaoStrategy sms = new NotificacaoSMS();

NotificacaoStrategy whatsapp = new NotificacaoWhatsApp();

Observer cli01 = new Cliente("Cliente Jose", "jose@empresa.com", "71991347220", email);

Observer cli02 = new Cliente("Cliente Maria", "maria@empresa.com", "71991347220", sms);

Observer func01 = new Funcionario("Funcionario Pedro", "pedro@empresa.com","Diretor", "71991347220", whatsapp);

Observer fornecedor01 = new Fornecedor("Funcionario Pedro", "pedro@empresa.com","111222333666777", "71991347220", email);

newsletter.adicionarObserver(cli01);

newsletter.adicionarObserver(cli02);

newsletter.adicionarObserver(func01);

newsletter.adicionarObserver(fornecedor01);

newsletter.enviarMensagem("Oferta Especial !");

}

}

9 – Depois de criar todos os observers (cli01, cli02, func01, fornecedor01), adicioná-los na lista do newsletter, enviar a mensagem. Seria possível alterar o meio de notificação de algum observer, e fazer o envio de uma nova mensagem ? Como você faria ? implemente a solução e faça novos testes .

10 – Utilize uma ferramenta de software qualquer para geração do diagrama de classes para esta etapa do projeto (Sugestão : Astah Community). Obs.: Adicione aqui o diagrama para que seja disponibilizado no teams

Obs: Outra possibilidade para geração dos diagramas é o StarUML : <https://staruml.io/> . Para fazer a engenharia reversa do código siga as orientações conforme o vídeo a seguir : <https://www.youtube.com/watch?v=z6DDuZQArro>

**Pacote : roteiro3.parte5**

Vamos exercitar um pouco a aplicação do padrão Observer.

Imagine um novo cenário onde temos que implementar um painel de controle para monitoramento de sensores para um sistema de IoT. Temos que monitorar atualmente os sensores de temperatura, pressão e umidade, mas precisamos ter a flexibilidade de adicionar ou remover novos sensores. O painel deve apresentar avisos de alerta caso algumas leituras indiquem valores fora da especificação.

* Temperatura dispara alerta acima de 40°C.
* Pressão dispara alerta acima de 100 (Pa).
* Umidade dispara alerta abaixo de 30% da umidade relativa do ar.

O sistema deve emitir alertas automáticos sempre setarmos algum valor de leitura para um dos sensores, de acordo com as especificações acima.

Importante implementar a solução e montar o diagrama de classes para validar a modelagem.