Skip to navigation

Marcos Brizeno

Desenvolvimento de Software #showmethecode

Mediator

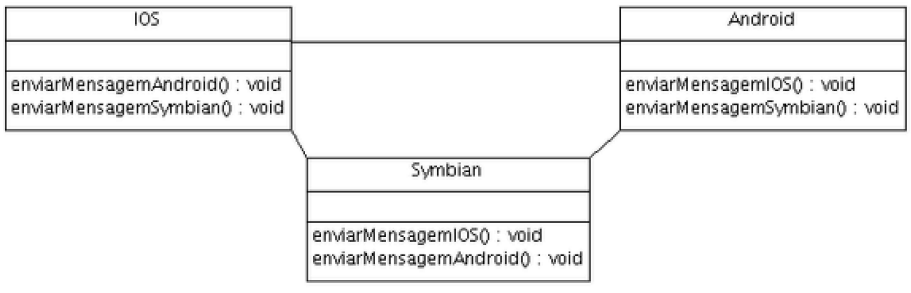
outubro 26, 2011

Mão na massa: Mediator

Problema

Pense na seguinte situação: seria legal ter um aplicativo que trocasse mensagem entre diversas plataformas móveis, um Android enviando mensagem para um iOS, um Symbian trocando mensagens com um Android… O problema é que cada uma destas plataforma implementa maneiras diferentes de receber mensagens.

Obviamente seria uma péssima solução criar vários métodos para cada plataforma. Analise o diagrama abaixo:



Imagine que agora o aplicativo vai incluir a plataforma BlackBerry OS, precisaríamos criar os métodos de comunicação com todas as outras plataforma existentes, além de adicionar métodos em todas as outras plataformas para que elas se comuniquem com o BlackBerry OS.

Esta ideia de relacionamento muitos para muitos pode deixar o design bem complexo, comprometendo a eficiência do sistema, bem como sua manutenibilidade.

Quando uma situação em que um relacionamento muitos para muitos é necessário em Banco de Dados, uma boa prática é criar uma tabela intermediária e deixar que ela relaciona uma entidade com outras várias e vice-e-versa. Esta é a ideia do padrão Mediator.

Mediator

Intenção:

“Definir um objeto que encapsula a forma como um conjunto de objetos interage. O Mediator promove o acoplamento fraco ao evitar que os objetos se refiram uns aos outros explicitamente e permitir variar suas interações independentemente.” [1]

Pela intenção podemos perceber que o Mediator atua como um mediador entre relacionamentos muitos para muitos, ao evitar uma referência explicita aos objetos. Outra vantagem que podemos notar é também que ele concentra a maneira como os objetos interagem.

O padrão Mediator consiste de duas figuras principais: o Mediator e o Colleague. O Mediator recebe mensagens de um Colleague, define qual protocolo utilizar e então envia a mensagem. O Colleague define como receberá uma mensagem e envia uma mensagem para um Mediator.

Vamos então implementar o Colleague que servirá como base para todos os outros:

```
1  public abstract class Colleague {
2      protected Mediator mediator;
3
4      public Colleague(Mediator m) {
5          mediator = m;
6      }
7
8      public void enviarMensagem(String mensagem) {
9          mediator.enviar(mensagem, this);
10     }
11
12     public abstract void receberMensagem(String mensagem);
13 }
```

Simples, define apenas a interface comum de qualquer Colleague. Todos possuem um Mediator, que deve ser compartilhado entre os objetos Colleague. Também define a maneira como todos os objetos Colleague enviam mensagens. O método “receberMensagem()” fica a cargo das subclasses.

Como exemplo de Colleague, vejamos as classes a seguir, que representam as plataformas Android e iOS:

```
public class IOSColleague extends Colleague {

    public IOSColleague(Mediator m) {
        super(m);
    }

    @Override
    public void receberMensagem(String mensagem) {
        System.out.println("iOs recebeu: " + mensagem);
    }
}
```

```
public class AndroidColleague extends Colleague {

    public AndroidColleague(Mediator m) {
        super(m);
    }

    @Override
    public void receberMensagem(String mensagem) {
        System.out.println("Android recebeu: " + mensagem);
    }
}
```

As classes Colleague concretas também são bem simples, apenas definem como a mensagem será recebida.

Vejamos então como funciona o Mediator. Vamos primeiro definir a interface comum de qualquer Mediator:

```
public interface Mediator {

    void enviar(String mensagem, Colleague colleague);

}
```

Ou seja, todo Mediator deverá definir uma maneira de enviar mensagens. Vejamos então como o Mediator concreto seria implementado:

```java
public class MensagemMediator implements Mediator {

    protected ArrayList<Colleague> contatos;

    public MensagemMediator() {
        contatos = new ArrayList<Colleague>();
    }

    public void adicionarColleague(Colleague colleague) {
        contatos.add(colleague);
    }

    @Override
    public void enviar(String mensagem, Colleague colleague) {
        for (Colleague contato : contatos) {
            if (contato != colleague) {
                definirProtocolo(contato);
                contato.receberMensagem(mensagem);
            }
        }
    }

    private void definirProtocolo(Colleague contato) {
        if (contato instanceof IOSColleague) {
            System.out.println("Protocolo iOS");
        } else if (contato instanceof AndroidColleague) {
            System.out.println("Protocolo Android");
        } else if (contato instanceof SymbianColleague) {
            System.out.println("Protocolo Symbian");
        }
    }

}
```

O Mediator possui uma lista de objetos Colleague que realizarão a comunicação e um método para adicionar um novo Colleague.

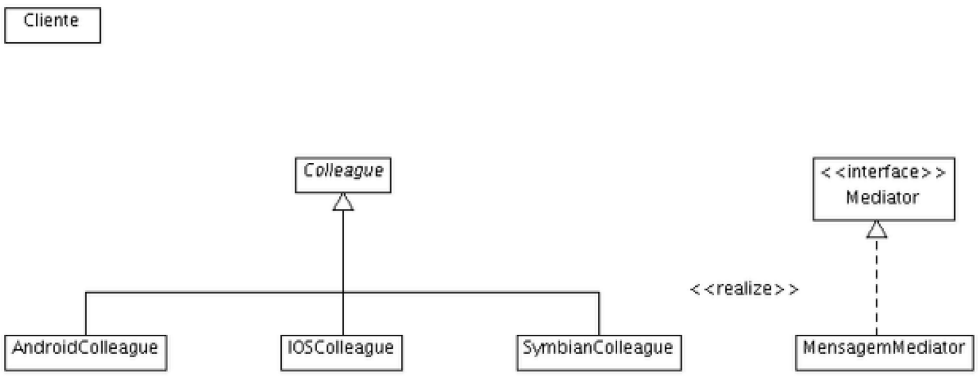
O método “enviar()” percorre toda a lista de contatos e envia mensagens. Note que dentro deste métodos foi feita uma comparação para evitar a mensagem seja enviada para a pessoa que enviou. Para enviar a mensagem primeiro deve ser definido qual protocolo utilizar e em seguida enviar a mensagem.

No nosso exemplo, o método “definirProtocolo()” apenas imprime na tela o tipo do Colleague que enviou a mensagem, utilizar para isso a verificação instanceof.

Desta maneira, o cliente poderia ser algo do tipo:

```
1  public static void main(String[] args) {
2      MensagemMediator mediador = new MensagemMediator();
3
4      AndroidColleague android = new AndroidColleague(mediador);
5      IOSColleague ios = new IOSColleague(mediador);
6      SymbianColleague symbian = new SymbianColleague(mediador);
7
8      mediador.adicionarColleague(android);
9      mediador.adicionarColleague(ios);
10     mediador.adicionarColleague(symbian);
11
12     symbian.enviarMensagem("Oi, eu sou um Symbian!");
13     System.out.println("=========");
14     android.enviarMensagem("Oi Symbian! Eu sou um Android!");
15     System.out.println("=========");
16     ios.enviarMensagem("Olá todos, sou um iOs!");
17 }
```

O diagrama UML para este exemplo seria o seguinte:



Um pouco de teoria

O padrão Mediator tem como principal objetivo diminuir a complexidade de relacionamentos entre objetos, garantindo assim que todos fiquem mais livres para sofrer mudanças, bem como facilitando a introdução de novos tipos de objetos ao relacionamento.

Outro ganho é a centralização da lógica de controle de comunicação entre os objetos, imagine que o protocolo de comunicação com o Android precisasse ser alterado, a mudança seria em um local bem específico da classe Mediator.

Uma vantagem não muito explorada nesse exemplo é que o Mediator centraliza também o controle dos objetos Colleague. Como citamos no post anterior sobre o padrão Observer, quando o relacionamento entre objetos Observer e Subject fica muito complexo, pode ser necessário utilizar uma classe intermediária que mapeie o relacionamento, facilitando o envio de mensagens aos objetos Observer.

Ao introduzir o Mediator vimos que a complexidade das classes Colleague foi transferida para o Mediator, o que tornou as classes Colleague bem mais simples e fáceis de manter. No entanto isto também pode ser um problema, pois a classe Mediator pode crescer em complexidade e se tornar

difícil de manter.

Código fonte completo

O código completo pode ser baixado no seguinte repositório Git: https://github.com/MarcosX/Padr-es-de-Projeto.

Os arquivos estão como um projeto do eclipse, então basta colocar no seu Workspace e fazer o import.

Se gostou do post compartilhe com seus amigos e colegas, senão, comente o que pode ser melhorado. Encontrou algum erro no código? Comente também. Possui alguma outra opinião ou alguma informação adicional? Comenta ai! 😀

Referências:

[1] GAMMA, Erich et al. Padrões de Projeto: Soluções reutilizáveis de software orientado a objetos.

 Mediator, Padrões de Projeto Java, Mediator, Padrões, Projeto 2 Comentários

__PRESENT