Skip to navigation

Marcos Brizeno

Desenvolvimento de Software #showmethecode

Observer

outubro 17, 2011

Mão na massa: Observer

Problema

Suponha que em um programa é necessário fazer várias representações de um mesmo conjunto de dados. Este conjunto de dados consiste de uma estrutura que contém 3 atributos: valorA, valorB e valorC, como mostra o código a seguir:

```
1  public class Dados {
2      int valorA, valorB, valorC;
3
4      public Dados(int a, int b, int c) {
5          valorA = a;
6          valorB = b;
7          valorC = c;
8      }
9  }
```

Como exemplo vamos considerar que é necessário representar dados em uma tabela, que simplesmente exibe os número, uma representação em gráficos de barras, onde os valores são exibidos em barras e outra representação em porcentagem, relativo a soma total dos valores.

A representação deve ser feita de modo que qualquer alteração no conjunto de dados compartilhados provoque alterações em todas as formas de representação, garantindo assim que uma visão nunca tenha dados invalidados.

Também queremos que as representações só sejam redesenhadas somente quando necessário. Ou seja, sempre que um valor for alterado.

Uma primeira solução poderia ser manter uma lista com as possíveis representações e ficar verificando por mudanças no conjunto de dados, assim que fosse feita uma mudança, as visualizações seriam avisadas.

O problema é que precisamos sempre verificar se houve ou não mudança no conjunto de dados, dessa forma o processamento seria muito caro, ou então a atualização seria demorada. Vamos ver então como o padrão Observer pode ajudar.

Observer

Intenção:

"Definir uma dependência um para muitos entre objetos, de maneira que quando um objeto muda de estado todos os seus dependentes são notificados e atualizados automaticamente." [1]

O padrão Observer parece ser uma boa solução para o problema, pois ele define uma dependência um para muitos, que será necessária para fazer a relação entre um conjunto de dados e várias representações, além de permitir que, quando um objeto mude de estado, todos os dependentes sejam notificados.

Para garantir isto o padrão faz o seguinte: cria uma classe que mantém o conjunto de dados e uma lista de dependentes deste conjunto de dados, assim a cada mudança no conjunto de dados todos os dependentes são notificados. Vejamos então o código desta classe por partes:

```
1  public class DadosSubject {
2
3      protected ArrayList&lt;DadosObserver&gt; observers;
4      protected Dados dados;
5
6      public DadosSubject() {
7          observers = new ArrayList&lt;DadosObserver&gt;();
8      }
9
10     public void attach(DadosObserver observer) {
11         observers.add(observer);
12     }
13
14     public void detach(int indice) {
15         observers.remove(indice);
16     }
17 }
```

Inicialmente definimos a lista de observadores (DadosObserver é uma interface comum aos observadores e será definida a seguir) e o conjunto de dados a ser compartilhado. Também definimos os métodos para adicionar e remover observadores, assim cada novo observador poderá facilmente acompanhar as mudanças.

Dentro da mesma classe, vamos definir as mudanças no estado, ou seja o conjunto de dados:

```
public void setState(Dados dados) {
    this.dados = dados;
    notifyObservers();
}

private void notifyObservers() {
    for (DadosObserver observer : observers) {
        observer.update();
    }
}

public Dados getState() {
    return dados;
}
```

Sempre que for feita uma mudança no conjunto de dados, utilizando o método "setState()" é chamado o método que vai notificar todos os observadores, executando um update para informar que o conjunto de dados mudou.

Vamos ver então como seria um observador. Vamos definir então a interface comum a todos os observadores, que é utilizada para manter a lista de observadores na classe que controla o conjunto de dados:

```
public abstract class DadosObserver {

    protected DadosSubject dados;

    public DadosObserver(DadosSubject dados) {
        this.dados = dados;
    }

    public abstract void update();
}
```

Definida a interface vamos então construir o observador que mostra os dados em uma tabela:

```
public class TabelaObserver extends DadosObserver {

    public TabelaObserver(DadosSubject dados) {
        super(dados);
    }

    @Override
    public void update() {
        System.out.println("Tabela:\nValor A: " + dados.getState().valorA
                + "\nValor B: " + dados.getState().valorB + "\nValor C: "
                + dados.getState().valorC);
    }
}
```

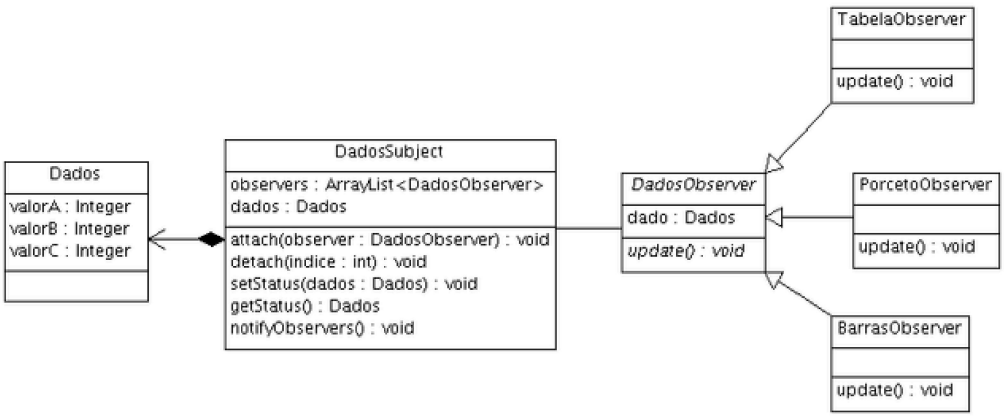
Este observador simplesmente exibe o valor dos dados. Assim, quando o método update for chamado ele irá redesenhar a tabela de dados. Para o exemplo apenas vamos exibir algumas informações no terminal.

Outros observers podem definir outras maneiras de mostrar o conjunto de dados, por exemplo o observer que exibe os valores em porcentagem:

```
1  public class PorcentoObserver extends DadosObserver {
2
3      public PorcentoObserver(DadosSubject dados) {
4          super(dados);
5      }
6
7      @Override
8      public void update() {
9          int somaDosValores = dados.getState().valorA + dados.getState().val
10                 + dados.getState().valorC;
11         DecimalFormat formatador = new DecimalFormat("#.##");
12         String porcentagemA = formatador.format((double) dados.getState().v
13                 / somaDosValores);
14         String porcentagemB = formatador.format((double) dados.getState().v
15                 / somaDosValores);
16         String porcentagemC = formatador.format((double) dados.getState().v
17                 / somaDosValores);
18         System.out.println("Porcentagem:\nValor A: " + porcentagemA
19                 + "%\nValor B: " + porcentagemB + "%\nValor C: " + porcenta
20                 + "%");
21     }
22
23 }
```

Para este observer é feito inicialmente o cálculo da soma dos valores e depois é calculado cada valor em relação a este total, exibindo o resultado com duas casas decimais.

A representação UML desta solução é a seguinte:



Um pouco de teoria

Na nomenclatura do padrão Observer temos os duas classes principais: Subject e Observer. O Subject é o que mantém os dados compartilhados e a lista de observadores que compartilham o dado. O Observer é o que faz utilização dos dados compartilhados e deve ser atualizado a cada modificação.

Como vimos no nosso exemplo o padrão Observer oferece uma excelente maneira de compartilhar um recurso, utilizando uma técnica parecida com o *broadcast*, onde todos os observres cadastrados em um subject são notificados sobre mudanças.

Ao realizar uma mudança é necessário ter cuidado, pois não se sabe exatamente os efeitos desta mudança nos seus observers ou o custo das atualizações nos observers. Caso as mudanças sejam muito complexas ou muito custosas pode ser interessante implementar uma estrutura intermediária para gerenciar os subjects e observers e suas mudanças.

Outro motivo para se utilizar uma estrutura intermediária entre subject e observer é quando existem muitos subjects e muitos observers interligados. Uma estrutura para mapear subjects e observers pode ser mais eficiente que uma lista de observers em cada subject.

Essa estrutura intermediária muitas vezes pode ser uma instância do padrão Mediator, que vamos abordar no próximo post da série. Também é interessante que esta classe intermediária seja uma instância do padrão Singleton, pois é interessante que apenas um objeto centralize o controle de subjects e observers.

Note também que no exemplo acima cometemos um pequeno "erro", pois não definimos a classe Subject como uma interface, isso dificultaria bastante alterações nesta classe. Geralmente a interface de Subject define apenas os métodos de adição e remoção de Observers e o método de notificação.

As subclasses de Subject vão definir como será a inserção e remoção de observers, e como estes observers serão notificados. Além disso ela deve prover uma maneira dos observers acessarem os dados compartilhados, definindo assim os métodos de "getState()" e "setState()"

Código fonte completo

O código completo pode ser baixado no seguinte repositório Git: https://github.com/MarcosX/Padroes-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.

 Observer, Padrões de Projeto Java, Observer, Padrões, Projeto 4 Comentários

__PRESENT