PRÁTICAS DE ENGENHARIA DE SOFTWARE

UNIRITTER LAUREATE INTERNATIONAL UNIVERSITIES

AIM1907 Prática de engenharia de software FAPA-N1

Profº: Jean Paul Lopes

Integrantes:
Eduardo Vaz(2020111574), Mauricio Henrique(2018111134), Pedro Goulart (202012393), Ricieri Pandolfo(201612919) e Roger Almeida Neto Teixeira(201718795)

Porto Alegre, 18 de Outubro de 2021

Template Method

Descrição do padrão:

É um padrão comportamental que possibilita que as subclasses sobrescrevam etapas específicas do algoritmo da superclasse sem alterar a estrutura do mesmo.

Uso da mesma:

Este padrão é utilizado quando queremos deixar com que apenas certas partes do algoritmo sejam alteradas, mas não sua estrutura como um todo. Permite transformar algoritmos monolíticos em uma série de etapas que podem ter trechos modificados sem alterar sua estrutura

Modelo de implementação:

As etapas do algoritmos deverão ser transformadas em métodos e nas etapas que é necessário uma implementação específica vamos definir o método como abstrato, assim obrigamos a subclasse, que irá estender nossa classe com os métodos padrão, a implementar esses métodos ou caso seja necessário a subclasse pode substituir a implementação de um método padrão.

Exemplos de uso:

Uma aplicação que extrai dados de pdf, csv ou doc. Nessa aplicação alguns métodos de processamento dos dados são parecidos, então vamos criar uma superclasse onde as etapas similares vão ser métodos padrão (podendo ser final para não ser substituída pelas subclasses). As etapas mais específicas ficam a cargo das subclasses implementarem, podendo também substituir alguns métodos padrão.

Correlação com outros padrões:

O padrão de factory method é uma especialização do template method podendo ser visto como uma etapa de um template method grande. Se assemelha também ao padrão Strategy, porém esse padrão trabalha com composição trabalhando a nível de objeto, permitindo que comportamentos sejam mudados durante a execução do código. Já o padrão template method é baseado em herança e funciona a nível de classe e por isso é estático

Observer

Descrição do Padrão:

O Observer é um padrão de projeto comportamental, ele permite a definição de mecanismos de assinatura que notificam um ou mais objetos sobre eventos que venham a acontecer com o objeto que eles estão observando.

Uso do mesmo:

O padrão de projeto Observer pode ser usado quando são necessárias mudanças em outros objetos, e o atual conjunto de objetos muda dinamicamente. Sendo assim, esse padrão permite que um objeto, observado, automaticamente notifique os objetos que estão vinculados a ele, nesse caso, os observadores.

Modelo de implementação:

Existirá um objeto que funcionará como "publicadora" e os demais serão um conjunto de classes "assinantes". Nesta classe "assinante" existirá um único método chamado Atualizar, no mínimo. A publicadora terá métodos para adicionar e remover objetos "assinantes" em sua lista. Cada vez que algo importante acontece dentro de uma "publicadora", os assinantes devem ser notificados. Implementar métodos de notificação de atualização nas classes "assinantes".

Exemplos de uso:

Mantendo o exemplo do modelo de implementação, podemos imaginar que temos Assinantes que assinam um serviço de uma Revista/Editora. Esta editora deverá notificar os assinantes caso haja novas edições. A edição nova é representada por um boolean "novaEdição" que será inicialmente False;
Existirá um método alterarEdicao() que promova a alteração do boolean para True caso haja nova edição.

Correlação com outros padrões:

Outros padrões de projeto abrangem várias maneiras de conectar remetentes e destinatários de pedidos, além do Observer, são eles o Command, Mediator e Chain of Responsability.
Os padrões Mediator e Observer possuem uma diferença obscura. É possível implementar qualquer um desses padrões, e também é possível implementar ambos simultaneamente.
Há uma implementação do padrão Mediador que depende do Observer, em que o objeto mediador faz o papel de um publicador, assim, os assinantes inscrevem-se ou removem sua inscrição pelos eventos do mediador. Dessa forma o mediador pode parecer muito similar ao Observer.

Adapter

Descrição do Padrão:

O Adapter é um padrão de projeto estrutural que permite objetos com interfaces incompatíveis colaborarem entre si.

Uso do mesmo:

O padrão de projeto Adapter deve ser utilizado quando quiser utilizar uma classe já existente, mas sua interface não for compatível com o restante do código. Esse padrão permite que crie uma classe de meio termo, servindo como um tradutor da nova classe e do código antigo. Também, o padrão pode ser utilizado quando quiser reutilizar diversas outras subclasses existentes mas que não possuam funcionalidade comum.

Modelo de implementação:

Deve-se verificar se possui pelo menos duas classes com interfaces compatíveis, uma classe serviço útil, que é modificável, e uma ou mais classes clientes que serão beneficiadas pela classe serviço. A interface cliente deve ser declarada e descrita como se comunicará com o serviço. Deve-se, então, criar a classe adaptadora e esta deve seguir a interface cliente. Adicione um campo para a classe do adaptador armazenar uma referência ao objeto do serviço, e implemente todos os métodos da interface cliente na classe adaptadora.

Exemplos de uso:

Um exemplo é uma aplicação que monitora o mercado de ações da bolsa. Caso queira melhorar a aplicação, integrando uma biblioteca de análise de terceiros, mas essa biblioteca funcione apenas com dados em formato diferente da sua aplicação, você pode criar um adaptador, que converte a interface do objeto para que o outro possa entendê-lo e se comunicar, mesmo tendo dados em formato diferente inicialmente.

Correlação com outros padrões:

Esse padrão pode ser relacionado com vários outros métodos. Enquanto o Adapter é usado em aplicações existentes para que classes não compatíveis consigam trabalhar juntas, o Bridge é usado com antecedência, o que permite que partes de uma aplicação possam ser desenvolvidas independentemente de outras. O Adaptar muda uma interface já existente de um objeto, já o Decorator não muda a interface de um objeto, apenas melhora. Enquanto o Adapter fornece para objetos existentes uma nova interface, o Proxy fornece a mesma interface para o objeto, e o Decorator uma melhor interface. Enquanto o Adapter faz uma interface já existente ser utilizável, o Facade define para objetos já existentes uma nova interface.

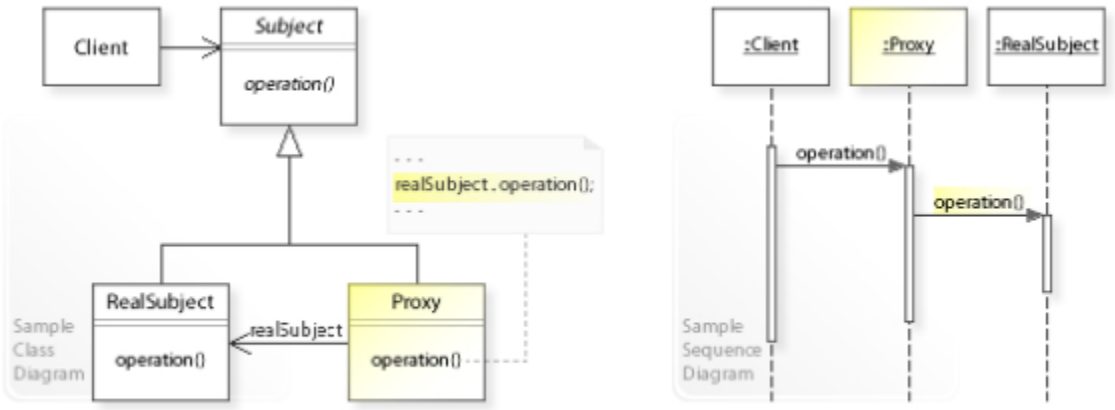
Proxy

Descrição do Padrão:

De acordo com a definição GoF de padrão de design de proxy , um objeto proxy fornece um substituto ou espaço reservado para outro objeto para controlar o acesso a ele. Um proxy é basicamente um substituto para um objeto pretendido que criamos devido a muitas razões, por exemplo, razões de segurança ou custo associado à criação de um objeto original totalmente inicializado.

Um objeto proxy oculta o objeto original e controla o acesso a ele. Podemos usar proxy quando quisermos usar uma classe que pode funcionar como uma interface para outra coisa. O proxy é muito usado para implementar casos de uso relacionados ao carregamento lento em que não queremos criar o objeto completo até que seja realmente necessário. Um proxy pode ser usado para adicionar uma camada de segurança adicional ao redor do objeto original também.

Arquitetura



Padrão de design proxy

- Subject - é uma interface que expõe as funcionalidades disponíveis para serem utilizadas pelos clientes.
- Real Subject - é uma implementação de classe Subjecte é uma implementação concreta que precisa ser escondida atrás de um proxy.
- Proxy - oculta o objeto real estendendo-o e os clientes se comunicam com o objeto real por meio deste objeto proxy. Normalmente, os frameworks criam este objeto proxy quando o cliente solicita um objeto real.

Exemplo de padrão de design de proxy

No exemplo dado, temos a RealObjectqual cliente precisa acessar para fazer algo. Ele solicitará que a estrutura forneça uma instância de RealObject. Mas como o acesso a esse objeto precisa ser protegido, o framework retorna a referência para RealObjectProxy.

Qualquer chamada para o objeto proxy é usada para requisitos adicionais e a chamada é passada para o objeto real.

RealObject.java

```java
public interface RealObject
{
    public void doSomething();
}
```

RealObjectImpl.java

```java
public class RealObjectImpl implements RealObject {

    @Override
    public void doSomething() {
        System.out.println("Performing work in real object");
    }

}
```

RealObjectProxy.java

```java
public class RealObjectProxy extends RealObjectImpl
{
    @Override
    public void doSomething()
    {
        //Perform additional logic and security
        //Even we can block the operation execution
        System.out.println("Delegating work on real object");
        super.doSomething();
    }
}
```

Client.java

```java
public class Client
{
    public static void main(String[] args)
    {
        RealObject proxy = new RealObjectProxy();
        proxy.doSomething();
    }
}
```

Console

```
Delegating work on real object
Performing work in real object
```

Os proxies são geralmente divididos em quatro tipos -

1. Proxy remoto - representa um objeto remotamente enxuto. Para falar com objetos remotos, o cliente precisa fazer um trabalho adicional na comunicação pela rede. Um objeto proxy faz essa comunicação em nome do objeto original e o cliente concentra-se em coisas reais para fazer.
2. Proxy virtual - atrasa a criação e inicialização de objetos caros até que sejam necessários, onde os objetos são criados sob demanda. As entidades de proxy criadas pelo Hibernate são exemplos de proxies virtuais.
3. Proxy de proteção - ajuda a implementar segurança sobre o objeto original. Eles podem verificar os direitos de acesso antes das invocações de método e permitir ou negar o acesso com base na conclusão.
4. Proxy inteligente - executa trabalho de manutenção adicional quando um objeto é acessado por um cliente. Um exemplo pode ser verificar se o objeto real está bloqueado antes de ser acessado para garantir que nenhum outro objeto possa alterá-lo.