CES-28 Fundamentos de Engenharia de

Subsistema Plugável e Padrões Façade e Singleton

Dylan Sugimoto

Gabriel Adriano

1. Dados o Subsistema Mercado Virtual e a classe cliente Aplicação, crie as classes do subsistema e os seus métodos necessários, bem como a classe BandoDeDados, que deve ser uma classe fake, de modo que a classe cliente Aplicação funcione! Use dois packages: um para o Subsistema Mercado Virtual e outro para a Aplicação.

Subsistema Mercado Virtual

public class Carrinho{

public static Carrinho create() {...}

public void adicionar (Produto p) {...}

public double getTotal() {...}

...

}

public class Produto{

public static Produto create(String nome, int id, double Preco) {...}

public getPreco() {...}

...

}

public class Cliente{

public static Cliente create(String nome, int id) {...}

public void adicionarCarrinho(Carrinho c) {...}

public Carrinho getCarrinho() {...}

...

}

public class BancoDeDados{

public cliente selectCliente(int id) {...}

public Produto selectProduto(int id) {...}

public void processarPagamento() {...}

...

}

public class Aplicacao {

…

BancoDeDados banco = new BancoDeDados();

// registrar comprador

Cliente cliente = Cliente.create(“Zé”, 123);

Carrinho car = Carrinho.create();

cliente.adicionarCarrinho(car);

// realizar uma compra

Produto produto = banco.selectProduto(223);

cliente.getCarrinho().adicionar(produto);

// realizar outra compra

Produto produto = banco.selectProduto(342);

cliente.getCarrinho().adicionar(produto);

// fechar compra

double valor = cliente.getCarrinho().getTotal();

banco.processarPagamento(cliente, valor);

…

}

1. Incorpore uma fachada de entrada (interface provida) ao Subsistema Mercado Virtual, que satisfaça as necessidades da classe Aplicação abaixo.

public class Aplicacao {

…

Facade f = new Facade();

f.registrar(“Zé”, 123);

f.comprar(223, 123);

f.comprar(342, 123);

f.fecharCompra(123);

…

}

public class Facade{

public void registrar(String nome, int clienteId) {

Cliente cliente = Cliente.create(nome, clienteId);

Carrinho car = Carrinho.create();

cliente.adicionarCarrinho(car);

}

public void comprar(int prodId, int clienteId) {

Cliente cliente = banco.selectCliente(Id);

Produto produto = banco.selectProduto(prodId);

cliente.getCarrinho().adicionar(produto);

}

public void fecharCompra(int clienteId) {

Cliente cliente = banco.selectCliente(clienteId);

double valor = cliente.getCarrinho().getTotal();

banco.processarPagamento(cliente, valor);

}

BancoDeDados banco = new BancoDeDados();

}

1. Delegar as tarefas da fachada de entrada às classes do Subsistema Mercado Virtual; crie classes novas apenas se e quando for estritamente necessário! Como tratar o acesso ao banco de dados encontrado no código anterior?

A implementação mais comum de Facade é ter um construtor privado – para impossibilitar a criação de uma instância da classe – e colocar os métodos da facade como métodos de fábrica estáticos!

public class Facade{

private Facade( ){

}

public static void registrar(String nome, int clienteId) {

<classe?>.registrar(String nome, int clienteId) ;

}

public static void comprar(int prodId, int clienteId) {

<classe?>. comprar(int prodId, int clienteId) ;

}

public static void fecharCompra(int clienteId) {

<classe?>. fecharCompra(int clienteId) ;

}

…

}

1. Outra implementação comum de Facade é usar o padrão Singleton preguiçoso, pois se deseja ter no máximo uma instância de uma dada Facade! Ou seja, faz-se uma implementação mais organizada e combinada com a versão do slide anterior!

Implementar a Façade de entrada por meio do padrão Singleton preguiçoso, mostrando também como ficará a classe Aplicação nesse caso. Faça o que achar necessário para tudo funcionar e exemplificar o uso do Singleton. Simule a lógica dos métodos registrar(), comprar() e fechar() escrevendo o valor dos parâmetros no console, onde cada um desses métodos deve pertencer a uma classe diferente do subsistema Mercado Virtual.

\*\*\*\*\*\*\*\*\* ATÉ AQUI FOI FEITO NA AULA PASSADA \*\*\*\*\*\*\*\*\*\*\*\*

1. Considere que a classe BancoDeDados agora pertence a outro subsistema, Repositorio. Estruture o subsistema Repositorio em outro package. Teste tudo com a Aplicação do item anterior.

\*\*\*\* FEITO na versão chamada Singleton, que possui 3 packages separados e uma fachada singleton para o package Mercado. \*\*\*\*

1. Serviço de nomes: considere e entenda o seguinte codigo (em um package chamado FakeNameService):

package FakeNameService;

import java.util.HashMap;

import java.lang.String;

public enum NameSrv

{

INSTANCE;

// instance vars, constructor

private final static HashMap<String, Object> names = new HashMap<String, Object>();

NameSrv(){ }

// Static getter

public static NameSrv getInstance()

{ return INSTANCE; }

public Object get(String name)

{ return names.get(name); }

public Object set(String name, Object o)

{ return names.put(name,o); }

public void printNames() {

System.out.println("Stored names are: ");

for ( String key : names.keySet() ) {

System.out.println( key );

}

}

}//class enum NameSrv

NameSrv() é um singleton que implementa um dicionario de objetos, onde objetos podem ser armazenados e obtidos atraves de chaves String. Isto representa um serviço de nomes em um framework de componentes, onde, ao invés de simples ponteiros para objetos, seriam armazenadas informações sobre onde o objeto estaria disponível na rede, e quais as suas interfaces.

O propósito é permitir que componentes encontrem os seus componentes requeridos através de nomes.

package FakeNameService;

import mercadoVirtual.Cliente;

import mercadoVirtual.Produto;

public interface IBancoDados {

public abstract Cliente selectCliente(int id);

public abstract Produto selectProduto(int id);

public abstract void processarPagamento(Cliente cliente, double valor);

public abstract void registrarCliente(Cliente cliente);

}

package FakeNameService;

public interface IMercado {

public int registraNovoCliente(String nome, int id) ;

public void adicionaProduto(int idCliente, int idProduto);

public void processarPagamento(int idCliente) ;

public IBancoDados setIBanco(String nomebanco) ;

}

Estas duas interfaces representam as interfaces dos componentes. Um componente implementa a interface e a PROVÊ, outro usa esta interface, ou seja, REQUER um outro componente com esta interface. Ambos os lados de uma comunicação devem conhecer a interface, então colocamos as interfaces em um package comum. Na realidade haveria algum tipo de linguagem para descrever a interface (muito parecido com o conceito de interfaces java), e as descrições estariam acessíveis para todos os componentes.

\*\* esta fornecido o código do serviço de nomes \*\*\*\*

1. Modifique os packages MercadoVirtual e Repositorio de forma que implementem as respectivas interfaces, e NÃO IMPORTEM DIRETAMENTE CLASSES DE OUTROS MÓDULOS (\*). MercadoVirtual importa apenas a interface IBancoDados. A aplicação importa apenas a interface IMercado (\*\*). Repositorio, por sua vez, não tem interface requerida, e não precisa importar nada, apenas os tipos que devem ser armazenados (Cliente e Produto) (\*\*\*). O Objetivo é fazer o seguinte código da aplicação funcionar:

package Application;

import mercadoVirtual.Facade; // (\*) nao temos ativador externo

import BancoDados.BancoDeDados;// ativamos comps aqui mesmo

import FakeNameService.IMercado;

import FakeNameService.NameSrv;

public class Aplicacao {

public static void main(String[] args) {

initSubsystems();

NameSrv NS = NameSrv.getInstance();

NS.printNames();

IMercado mercado = (IMercado)NS.get("Mercado");

int idCliente = mercado.registraNovoCliente("Ze", 123);

mercado.adicionaProduto(idCliente, 223);

mercado.adicionaProduto(idCliente, 342);

mercado.processarPagamento(idCliente);

createOtherBanco();

mercado.setIBanco("BancoNovo");

System.out.println("try old client:" );

int idCliente2 = mercado.registraNovoCliente("Ze", 100);

mercado.adicionaProduto(idCliente2, 100);

mercado.adicionaProduto(idCliente2, 150);

mercado.processarPagamento(idCliente2);

// uncomente: o 1o cliente nao existe, e da erro.

//mercado.processarPagamento(idCliente);

}

// with a real component system, this would not be needed

// an external activator would load and run the components

public static void initSubsystems() {

NameSrv NS = NameSrv.getInstance();

// BancoDeDados is not a singleton. It also works.

NS.set("Banco",new BancoDeDados());

// Mercado is singleton

NS.set("Mercado",MercadoFacade.getInstance());

// Mercado can find Banco itself via the NS. using the default name

NS.printNames();// just to check

}

public static void createOtherBanco() {

NameSrv NS = NameSrv.getInstance();

NS.set("BancoNovo",new BancoDeDados());

}

}

(\*) com as exceções indicadas pelos itens (\*\*) e (\*\*\*)

(\*\*) A aplicação, a princípio, precisaria apenas conhecer as interfaces dos componentes requeridas diretamente por ela, ou seja, só IMercado, e não Mercado (implementação) e muito menos IBancoDados ou BancoDeDados. E em um framework de componentes distribuídos, as classes IMercado que a aplicação usaria seriam Proxies - *procure pelo padrão de projeto* ***Proxy*** *[GoF94] para entender* - enquanto os componentes reais estariam em outras threads, processos, ou máquinas na rede. Mas aqui é fake, e precisamos importar também as fachadas de cada modulo para poder inicializa-las, por isso, aplicação importa BancoDeDados, o que cria dependências indesejadas.

(\*\*\*) Se queremos transmitir dados entre componentes, ambos devem conhecer o formato dos dados. Tambem haveria uma API e ferramentas para isso, mas aqui vamos importar diretamente as classes para simplificar.

Mas explicadas as exceções, o código desta aplicação usa o servico de nomes para encontrar alguém que implementa IMercado, usando o nome padronizado. Mercado, internamente, procura no Name Service quem implementa IBancoDados, usando o nome default (“Banco”).

A função initSubsystems é um "cheat". Em um framework real, uma ferramenta de ativação leria um arquivo de configuração que especificaria os componentes a serem carregados com parâmetros de configuração, os registraria no serviço de nomes, e faria os acertos de parâmetros e nomes necessários para comunicar um lado com o outro.

Então, a aplicação pode usar o Mercado, e nem precisa saber onde esta o banco de dados, ou mesmo o que é Mercado, a única coisa que ela sabe é que encontrou alguém que implementa IMercado.

Mais um cheat: createOtherBanco cria um segundo componente Banco, e o registra com outro nome. E o nosso componente Mercado tem uma função que permite mudar o Banco utilizado, passando apenas o nome (Dependency Injection).

Dado o código fornecido para o package Infraestrutura (Versão chamada NameService), modifique o resto do código para funcionar e responda:

1. como mudou o acoplamento entre BancoDeDados e Mercado? (isto é o mais importante, desacoplá-los!)

Mercado ficou desacoplado com o BancoDeDados, apenas com a sua abstração IBancoDados, que poderia ser remoto.

1. Precisamos mesmo usar Dependency Injection em Mercado? Porque? Isso é bom, necessário, ou ruim??

Mercado precisa utilizar a Injeção de Dependência, pois ele precisa receber de alguém (NameService) qual banco de dados se deve utilizar. Isso é necessário na arquitetura de componentes, pois se deve falar para um componente que outro componente este deve utilizar, já que eles são criados separadamente.

1. Depois da inicialização, a aplicação pode usar o Mercado da mesma forma que antes? A aplicação anterior que usava Mercado ainda funcionaria apenas mudando a inicialização?

Sim. A diferença está apenas na inicialização, a aplicação funcionaria da mesma forma que antes.

1. Usamos um objeto fachada e uma interface para os módulos. Mas também poderíamos usar troca de mensagens. Utilize o objeto que simula um serviço de eventos (Publish/Subscriber) para trocar mensagens entre os módulos, para pelo menos uma das operações. Ou seja, elimine uma operação da interface e implemente-a atraves de uma mensagem enviada via Pub/Sub. Ao invés de chamar um método de uma fachada, um módulo, que atua como publisher, publica uma mensagem. Outro módulo, um subscriber, recebe a mensagem, decodifica seus campos (pode ser um objeto) e realiza a operação comandada. CADA MODULO DEVE ENCONTRAR O OBJETO PUB/SUB VIA SERVICO DE NOMES.

\*\*\* código fornecido para o package infraestrutura, versão chamada NameEvtSrvs \*\*\*\*\*\*\*\*\*

* 1. Qual a operação mais fácil de eliminar? Dica: pub sub não possui mecanismo para facilitar a implementação de respostas. O mais fácil é apenas enviar uma mensagem. Para operações que exigem respostas há conceitos mais elaborados: services e actions.
  2. Pode usar o seu próprio Serviço de Eventos da aula anterior, ao invés de usar este disponibilizado. Mas responda para este modelo, e para o seu modelo: é modelo push ou pull?
  3. Para este e para o seu modelo de Serviço de Eventos responda: Há duas opções, qual é a utilizada: I) O serviço de eventos apenas conecta pubs com subs, e cada mensagem posteriormente é enviada diretamente de publishers a subscribers sem passar pelo Serviço. II) O serviço de eventos recebe todas as mensagens de todos os seus publishers, e re-envia cada mensagem para os subscribers apropriados.
  4. Quais as vantagens e desvantagens dos dois modelos descritos em (c)?

1. Na camada inferior, implemente um gerador de eventos periódicos. Poderia servir, por exemplo, para checks de consistência no BD. Ou para gerar promoções periódicas. Contrua uma thread que gere um evento a cada 5 segundos (procure um exemplo de gerar evento usando o relógio na net). A camada inferior deve chamar a camada superior não diretamente, mas por meio de um callback. Este callback pode ser implementado a) por meio do mesmo objecto publish/subscribe, ou b) pode haver uma nova função na fachada para que a camada superior forneça um ponteiro para um objeto de callback. Implemente uma destas opções.
2. Responda, a) qual a diferença entre 9a) e 9b)??? b) E se publisher, subscriber estivessem em processos ou threads diferentes??? E se o serviço de eventos fosse um terceiro processo/thread??? Dica: pense no problema de escala (muitos callbacks, com muitos subscribers para cada um).
3. Dada uma nova versão do package Infraesturura, onde o Servico de nomes não fornece um BancoDeDados mas um Proxy correspondente. Estude o código, e modifique o resto da aplicação para funcionar.
   1. Se o usuário definir uma interface como IBancoDeDados, seria possível que um framework gerasse automaticamente o código de ProxyIBancoDeDados, incluindo no Proxy o código necessário para encontrar na rede a instancia correta de BancoDeDados registrada no serviço de nomes?
4. Detalhes que não tratamos: Quem chama um serviço remoto, normalmente possui um Proxy, e o Proxy se encarrega de esconder os detalhes de rede. Analogamente, o framework deveria gerar automaticamente uma classe StubIBancoDados, que se encarrega de receber e processar a mensagem recebida remotamente. Normalmente, o banco de dados real, do lado do servidor, deveria herdar ou implementar o Stub, que chamaria os métodos da classe real quando uma mensagem remota eh recebida, e enviaria de volta o valor de retorno.
5. Outro detalhe: Produto, Cliente, Carrinho, ainda estão visíveis em ambos os lados, acoplando os packages. Que alternativas teríamos para diminuir ainda mais o acoplamento, e qual a dificuldade nesse caso específico? Porque estas classes dificultam o desacoplamento? Quais as vantagens e desvantagens?
   1. Uma alternativa seria nunca enviar classes completas, mas apenas dados. Cada lado usaria os dados para construir classes, que sequer precisam corresponder.
   2. Outra alternativa, principalmente quando o sistema não é distribuído, mas se trata do mesmo processo, é aceitar a dependência de classes que representam dados comuns a vários packages. Se importam essas classes quando necessário.

Apresentar o código das classes dos subsistemas e da aplicação em texto logo apos cada item acima, bem como na forma de fonte Java zipado à parte, identificados pelo número do item, de modo que eu possa testá-lo no Eclipse.