UNIVERSIDADE DO VALE DO ITAJAÍ

CIÊNCIA DA COMPUTAÇÃO

PROGRAMAÇÃO ORIENTADA A OBJETOS

PROFESSOR: CARLOS HENRIQUE BUGHI

AUTORES:

GUSTAVO BARON LAURITZEN

MATHEUS BARON LAURITZEN

GABRIEL BOSIO

EDUARDO DA ROCHA WEBER

RELATÓRIO SOBRE PADRÕES DE PROJETO NA COMPUTAÇÃO

RELATÓRIO SOBRE OS PADRÕES DE PROJETOS: FACTORY METHOD E CHAIN OF RESPONSABILITY

ITAJAÍ

2023

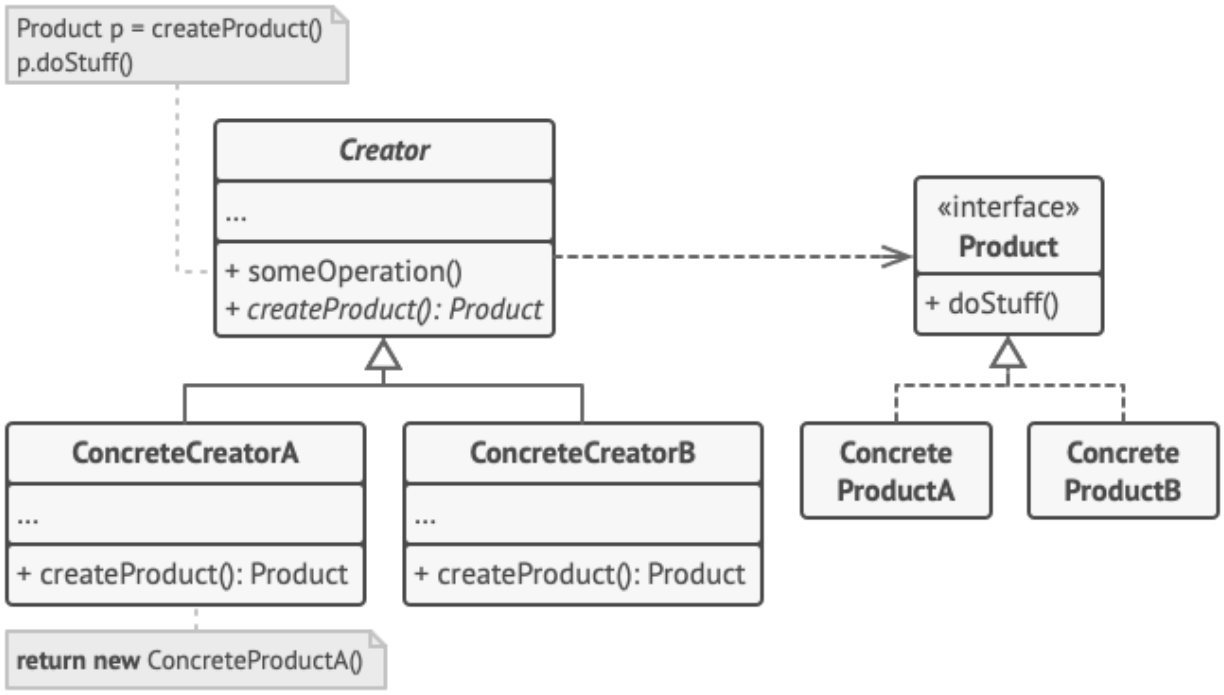
- Factory Method:

O padrão Factory Method é amplamente utilizado na programação orientada a objetos para resolver o problema de criação de objetos em um sistema. Ele é útil quando um sistema precisa criar objetos de diferentes tipos, mas o código do cliente não deve depender das classes concretas desses objetos.

- Princípios Subjacentes:
    - O Factory Method baseia-se em princípios de abstração, encapsulamento e polimorfismo;
    - Ele promove a abstração ao permitir que o código do cliente trabalhe com a interface do objeto criado, em vez de lidar diretamente com sua implementação concreta;
    - Encapsula a lógica de criação em uma classe separada, proporcionando um local centralizado para a criação de objetos;
    - O polimorfismo é alcançado ao permitir que diferentes fábricas criem objetos diferentes que implementam a mesma interface e/ou sobrescrevam/sobrecarregam seus métodos/atributos;
- Características:
    - Definição de uma interface ou classe abstrata que declara o método de fábrica, responsável por criar os objetos;
    - As classes concretas que implementam essa interface ou herdam dessa classe abstrata são as fábricas reais que fornecem a implementação do método de fábrica.
    - Cada fábrica concreta pode criar um tipo específico de objeto, mas todos os objetos criados devem seguir a mesma interface comum;
    - O código cliente pode trabalhar com qualquer objeto criado pela fábrica, sem se preocupar com sua classe concreta;
- Quando aplicar adequadamente:
    - O código do cliente precisa trabalhar com múltiplas implementações de uma mesma interface, sem conhecer antecipadamente as classes concretas;
    - Não souber de antemão os tipos e dependências exatas dos objetos com os quais seu código deve funcionar. O Factory Method separa o código de construção do produto do código que realmente usa o produto. Portanto, é mais fácil estender o código de construção do produto independentemente do restante do código;
    - É necessário adicionar novos tipos de objetos no futuro, sem modificar o código existente;
- Alguns de seus benefícios:
    - Separação de responsabilidades:
        - O padrão Factory Method separa a lógica de criação de objetos do código do cliente, garantindo uma melhor organização e manutenibilidade do código;
    - Flexibilidade:
        - O código do cliente pode ser facilmente adaptado para trabalhar com novos tipos de objetos criados pelas fábricas, sem a necessidade de alterar seu código;
    - Reutilização de código:
        - O padrão permite reutilizar a lógica de criação de objetos em diferentes partes do sistema, evitando a duplicação de código;
    - Escalabilidade:

- À medida que novos tipos de objetos são adicionados, basta criar uma nova fábrica correspondente, mantendo o código do cliente inalterado. Isso facilita a expansão do sistema;

- Estrutura genérica em UML(Unified Modeling Language):



Em resumo, o padrão de projeto Factory Method oferece uma abordagem flexível e eficiente para a criação de objetos em um sistema. Ele promove a separação de responsabilidades, a reutilização de código e a escalabilidade. Ao usar esse padrão, o código do cliente se torna mais desacoplado das classes concretas, resultando em um design com melhor manutenção/adaptação e escalabilidade.

- Exemplo de Implementação “Gerenciamento de uma Garagem” em Java:

Neste exemplo, a Interface Veículo desempenha o papel do produto e a Classe Iate e Helicoptero desempenham uma função de produto concreto e a Classe VeiculoTerrestre atua como criador dos produtos Caminhao e Carro. Além disso, existe a Classe Galpao que gerencia a criação de qualquer veículo.

- Interface Veiculo:

```
package br.univali.cc.prog3.avaliacao.dominio;

/**

*

* @author 7853653

*/

public interface Veiculo {

   String obterDescricaoVeiculo();

}
```

- Produto concreto(Classe Iate, Classe Helicoptero):

```
//Classe Iate
package br.univali.cc.prog3.avaliacao.dominio;

/**
*
* @author 7853653
*/
public class Iate implements Veiculo {

    private String nome;
    private String capitao;

    public Iate(String nome) {
        this.nome = nome;
    }

    public void setCapitao(String capitao) {
        this.capitao = capitao;
    }

    public String getNome() {
        return nome;
    }

    @Override
    public String obterDescricaoVeiculo() {
        return "Iate: "+ "\nNome: " + getNome() + "\nCapitão: " + capitao;
    }
}
//Classe Helicoptero
package br.univali.cc.prog3.avaliacao.dominio;
```

```
/**
*
* @author 7853653
*/
public class Helicoptero implements Veiculo {

    private String modelo;
    private String sigla;
    private String piloto;

    public Helicoptero(String modelo, String sigla) {
        this.modelo = modelo;
        this.sigla = sigla;
    }

    public void setPiloto(String piloto) {
        this.piloto = piloto;
    }

    public String detalhar(){
        return "\nModelo: " + modelo + "\nSigla: " + sigla + "\nPiloto: " + piloto;
    }

    @Override
    public String obterDescricaoVeiculo() {
        return "Helicoptero: " + detalhar();
    }
}
```

- Criador Concreto (Classe VeiculoTerrestre):

```
package br.univali.cc.prog3.avaliacao.dominio;
/**
```

```
*
* @author 7853653
*/
abstract public class VeiculoTerrestre {

    protected String modelo;
    protected String placa;
    protected String motorista;

    public VeiculoTerrestre(String modelo, String placa) {
        this.modelo = modelo;
        this.placa = placa;
    }
    public void setMotorista(String motorista) {
        this.motorista = motorista;
    }
    public String getPlaca() {
        return placa;
    }
    abstract public String detalhar();

}
```

- Criador concreto(Classe Caminhao, Classe Carro):

```
//Classe Carro
package br.univali.cc.prog3.avaliacao.dominio;

/**
*
* @author 7853653
*/
public class Carro extends VeiculoTerrestre implements Veiculo{
```

```
    private double potencia;

    public Carro(double potencia, String modelo, String placa) {
        super(modelo, placa);
        this.potencia = potencia;
    }

    public double getPotencia() {
        return potencia;
    }

    @Override
    public String detalhar() {
        return "\nModelo: " + modelo + "\nPlaca: " + placa + "\nMotorista: " + motorista + "\nPotência: " +
potencia + "cv";
    }

    @Override
    public String obterDescricaoVeiculo() {
        return "Carro: " + detalhar();
    }

}

//Classe Caminhao
package br.univali.cc.prog3.avaliacao.dominio;

/**
*
* @author 7853653
*/
public class Caminhao extends VeiculoTerrestre implements Veiculo {
```

```
    private double capacidadeCarga;

    public Caminhao(double capacidadeCarga, String modelo, String placa) {
        super(modelo, placa);
        this.capacidadeCarga = capacidadeCarga;
    }

    public double getCapacidadeCarga() {
        return capacidadeCarga;
    }

    @Override
    public String detalhar() {
        return "\nModelo: " + modelo + "\nPlaca: " + placa + "\nMotorista: " + motorista + "\nCapacidade de carga: " + capacidadeCarga + "kg";
    }

    @Override
    public String obterDescricaoVeiculo() {
        return "Caminhão: " + detalhar();
    }
}
```

- Classe Gerenciadora(Classe Galpao):

```
package br.univali.cc.prog3.avaliacao.dominio;
/**
*
* @author 7853653
*/
public class Galpao {

    private Veiculo veiculos[] = new Veiculo[999];
    private int contadorVeiculos = 0;
```

```
    public Galpao() {
    }

    public void adicionarVeiculo(Veiculo veiculo){
        this.veiculos[contadorVeiculos++] = veiculo;
    }

    public String imprimirRelacaoVeiculos(){
        String message = "";
        for (Veiculo veiculo : veiculos) {
            if (veiculo != null) {
                message = message + veiculo.obterDescricaoVeiculo() + "\n\n";
            }
        }
        if("".equals(message)){
            return "Galpao Vazio";
        }else{
            return message;
        }
    }
}
```

//Fim da Implementacao

- Chain of Responsability:

O padrão de projeto Chain of Responsibility, ou "Cadeia de Responsabilidade" em português, é um padrão comportamental que permite tratar solicitações de forma flexível, desacoplada e escalável. Ele é baseado em alguns princípios subjacentes e possui características específicas que o tornam útil em diversas situações.

Princípios subjacentes:

- Princípio da Responsabilidade Única: Cada objeto na cadeia de responsabilidade tem a responsabilidade de tratar ou encaminhar a solicitação de acordo com sua lógica específica. Isso promove a coesão e ajuda a manter cada objeto focado em uma única responsabilidade.
- Princípio do Encapsulamento: O padrão Chain of Responsibility encapsula a lógica de tratamento de solicitações em cada objeto da cadeia. Isso permite que cada objeto mantenha sua própria implementação privada e oculte detalhes internos, garantindo um baixo acoplamento entre os objetos.

Características:

- Desacoplamento: O padrão promove um baixo acoplamento entre o remetente da solicitação e os objetos que a tratam. Isso permite que o remetente não precise conhecer explicitamente quais objetos irão tratar a solicitação, aumentando a flexibilidade e a extensibilidade do sistema.
- Encaminhamento sequencial: A solicitação percorre a cadeia de objetos de forma sequencial, passando de um objeto para outro até que seja tratada ou a cadeia seja percorrida completamente. Cada objeto decide se irá tratar a solicitação ou passá-la adiante para o próximo objeto na cadeia.
- Flexibilidade: O padrão Chain of Responsibility permite adicionar, remover ou reconfigurar objetos na cadeia de forma flexível, sem afetar o remetente ou outros objetos na cadeia. Isso facilita a adaptação e extensão do sistema, permitindo diferentes combinações e ordens de tratamento.
- Tratamento condicional: Cada objeto na cadeia pode decidir se trata a solicitação com base em alguma lógica específica. Isso permite a personalização do comportamento de tratamento de acordo com as necessidades de cada objeto.
- Tratamento padrão: O padrão Chain of Responsibility permite definir um objeto final na cadeia que trata as solicitações que não são tratadas por nenhum objeto anterior. Isso garante um tratamento padrão quando nenhum objeto específico na cadeia é capaz de tratar a solicitação.

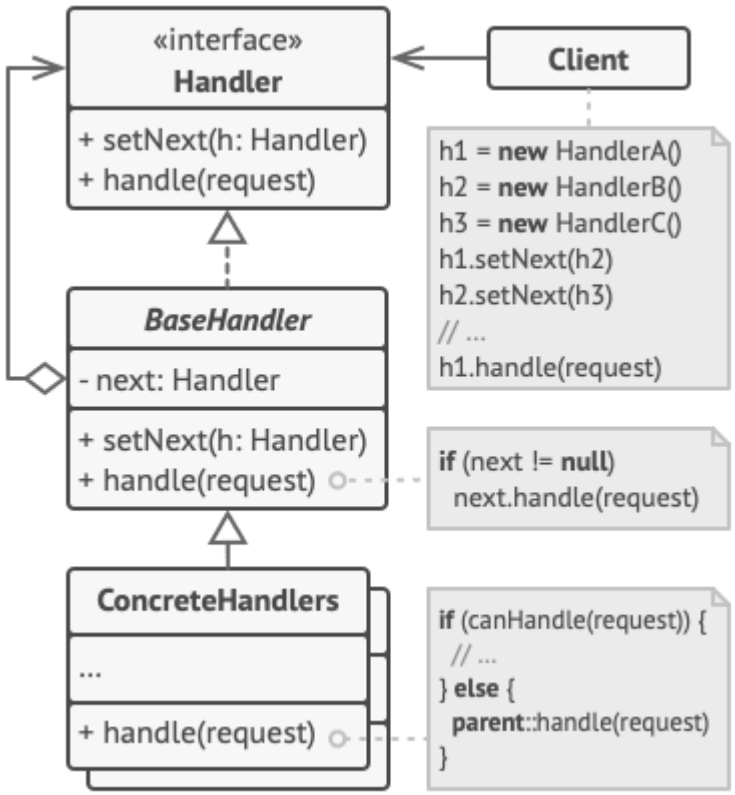
Quando se é adequado aplicar o padrão:

- O remetente de uma solicitação não precisa saber quais objetos irão tratá-la.
- Diferentes objetos podem tratar a solicitação, mas o objeto correto não é conhecido antecipadamente.
- É necessário adicionar ou alterar dinamicamente a lógica de tratamento de solicitações.
- A ordem de tratamento ou o conjunto de objetos que tratam a solicitação podem variar.

Benefícios:

- Flexibilidade: Permite adicionar, remover ou reconfigurar objetos na cadeia sem afetar o remetente ou outros objetos.
- Escalabilidade: A cadeia pode ser facilmente expandida para lidar com novos tipos de solicitações ou requisitos.
- Desacoplamento: Reduz o acoplamento entre remetente e destinatários, aumentando a modularidade do código.
- Manutenibilidade: Facilita a manutenção, pois as mudanças em um objeto não afetam os outros objetos na cadeia.
- Reutilização: Os objetos na cadeia podem ser reutilizados em diferentes contextos, resultando em um código mais modular e eficiente.

Estrutura genérica em UML(Unified Modeling Language):



Em resumo, o padrão Chain of Responsibility é útil quando se deseja desacoplar remetente e destinatários, permitindo tratamento flexível e escalável de solicitações. Ele promove a modularidade, reutilização e manutenibilidade do código, trazendo benefícios em termos de eficiência e flexibilidade para sistemas que lidam com lógica de tratamento de solicitações complexas.

Um exemplo prático do padrão Chain of Responsibility é em sistemas de tratamento de requisições HTTP, onde cada objeto na cadeia pode verificar se pode lidar com uma determinada requisição (por exemplo, com base no tipo de requisição, rota ou cabeçalhos) e, em seguida, tratar a requisição ou passá-la adiante para o próximo objeto na cadeia.

```
import java.util.HashMap;
import java.util.Map;
```

- Classe que representa uma requisição HTTP

```
class HttpRequest {
	private String method;
	private String route;
	private Map<String, String> headers;

	public HttpRequest(String method, String route, Map<String, String> headers) {
		this.method = method;
		this.route = route;
		this.headers = headers;
	}
```

```
	public String getMethod() {
		return method;
	}

	public String getRoute() {
		return route;
	}

	public Map<String, String> getHeaders() {
		return headers;
	}
}
```

- Interface para os objetos da cadeia de responsabilidade

```
interface RequestHandler {
	void handleRequest(HttpRequest request);
}
```

- Implementação base da interface RequestHandler

```
abstract class BaseRequestHandler implements RequestHandler {
	private RequestHandler nextHandler;

	public void setNextHandler(RequestHandler nextHandler) {
		this.nextHandler = nextHandler;
	}

	public void handleRequest(HttpRequest request) {
		if (canHandle(request)) {
			processRequest(request);
		} else if (nextHandler != null) {
			nextHandler.handleRequest(request);
		} else {
			System.out.println("Nenhum manipulador encontrado para a requisição.");
		}
	}

	protected abstract boolean canHandle(HttpRequest request);

	protected abstract void processRequest(HttpRequest request);
}
```

- Exemplo de manipulador específico para tratamento de requisições GET

```
class GetRequestHandler extends BaseRequestHandler {
	protected boolean canHandle(HttpRequest request) {
		return request.getMethod().equalsIgnoreCase("GET");
```

```
    }

    protected void processRequest(HttpRequest request) {
        System.out.println("Manipulando a requisição GET para a rota: " + request.getRoute());
    }
}
```

- Exemplo de manipulador específico para tratamento de requisições POST

```
class PostRequestHandler extends BaseRequestHandler {
    protected boolean canHandle(HttpRequest request) {
        return request.getMethod().equalsIgnoreCase("POST");
    }

    protected void processRequest(HttpRequest request) {
        System.out.println("Manipulando a requisição POST para a rota: " + request.getRoute());
    }
}
```

- Exemplo de manipulador específico para tratamento de requisições com um determinado cabeçalho

```
class HeaderRequestHandler extends BaseRequestHandler {
    private String headerName;

    public HeaderRequestHandler(String headerName) {
        this.headerName = headerName;
    }

    protected boolean canHandle(HttpRequest request) {
        return request.getHeaders().containsKey(headerName);
    }

    protected void processRequest(HttpRequest request) {
        System.out.println("Manipulando a requisição com o cabeçalho " + headerName);
    }
}
```

- Exemplo de uso

```
public class Main {
    public static void main(String[] args) {
        // Construindo a cadeia de responsabilidade
        RequestHandler getHandler = new GetRequestHandler();
        RequestHandler postHandler = new PostRequestHandler();
        RequestHandler headerHandler = new HeaderRequestHandler("Authorization");
```

```
        getHandler.setNextHandler(postHandler);
        postHandler.setNextHandler(headerHandler);

        // Criando uma requisição HTTP
        Map<String, String> headers = new HashMap<>();
        headers.put("Authorization", "Bearer token123");
        HttpRequest request = new HttpRequest("POST", "/api/users", headers);

        // Executando o tratamento da requisição
        getHandler.handleRequest(request);
    }
}
```

- Bibliografia:

- Refactoring Guru. Padrões de Projeto: Factory Method. Disponível em: https://refactoring.guru/pt-br/design-patterns/factory-method. Acesso em: [16/06/2023].
- GAMMA, Erich et al. Design Patterns: Elements of Reusable Object-Oriented Software. [S.l.]: Addison-Wesley Professional, 1994.
- Tutorialspoint. Factory Pattern. Disponível em: https://www.tutorialspoint.com/design_pattern/factory_pattern.htm. Acesso em: [18/06/2023].
- Refactoring Guru. Padrões de Projeto: Chain of Responsability. Disponível em: https://refactoring.guru/pt-br/design-patterns/chain-of-responsibility. Acesso em: [16/06/2023].