DISCIPLINA DE
PADRÕES DE PROJETO (DESIGN PATTERNS)

Sumário

Apresentação

Everton Gomede

Doutorando em Engenharia Elétrica/Computação pelo Programa de Pós-graduação da Faculdade de Engenharia Elétrica e de Computação (FEEC) da Universidade Estadual de Campinas (UNICAMP). Mestre em Ciência da Computação pela Universidade Estadual de Londrina. MBA (Master of Business Administration) em Neurogestão pela Fundação Getúlio Vargas. MBA em Gerenciamento de Projetos pela FGV. Especialista em Engenharia de Software e Banco de Dados pela Universidade Estadual de Londrina. Graduado em Processamento de Dados. Sun Certified Programmer for Java 2 Platform 1.4 e ITIL® Foundation Certificate in IT Service Management. Experiência de mais de 10 anos na área de TI em arquitetura de software, engenharia de software, mapeamento de processos de negócio, gerenciamento de projetos, desenvolvimento de sistemas transacionais e de suporte à decisão. Experiência em projetos nacionais e internacionais em instituições financeiras e empresas de desenvolvimento de software. Professor de cursos de pós-graduação em universidades públicas e privadas. Membro do PMI 1580489. Publicações nacionais e internacionais. Proficiência em inglês atestada pelo TOIEC e TOEFL. Áreas de Interesse: Estrutura de Dados e Organização de Arquivos, Análise de Sistemas e Engenharia de Software, Lógica de Programação, Redes de Computadores, Linguagem de Programação, Administração de Banco de Dados, Governança de TIC (ITIL/CobIT), Data Warehouse, Data Mining, Big Data, Gerenciamento de Projetos, Gerenciamento de Processos de Negócio (BPM/BPMN™), Pesquisa Operacional, Algoritmos, Sistemas de Suporte à Decisão, Data Science e Machine Learning.

Ricardo Satin

Mestre em Engenharia de Software pela UTFPR; Especialista em Computação pela Unicesumar; Graduado em Computação pela Universidade Estadual de Maringá - UEM; MBA em Gerenciamento de Projetos pela FCV. Profissional certificado em: gerenciamento de projeto pelo P.M.I. (profissional PMP); Gerenciamento ágil de projetos (SCRUM); Gerenciamento de serviços de T.I. (ITIL). Professor de ensino superior para cursos da área de T.I. e pós-graduação para diversas áreas. Atua como consultor em gerenciamento de projetos; processos de negócio; melhoria de processos e machine learning.

UNIDADE 1 – INTRODUÇÃO

Objetivos de aprendizagem

- Conceitos fundamentais de engenharia de software
- Conceitos fundamentais de arquitetura de software
- Acoplamento e coesão
- Padrões de projetos

Conceitos fundamentais de engenharia de software

O objetivo principal da engenharia de software é produzir software de qualidade. Ela não é sobre processos, artefatos, documentos, técnicas, etc. Claro que esses assuntos permeiam a engenharia de software e são a matéria prima da mesma. Apesar disso, o objetivo mantem-se sempre o mesmo: a produção de software de qualidade.

Mas como nós produzimos software de qualidade? Essa pergunta acompanha sempre os profissionais responsáveis por esse tipo de trabalho. Alguns apostam em processos, outros em ferramentas, outros em tecnologias. Cada um desses aspectos possui sua relevância. Entretanto, os aspectos fundamentais geralmente são ignorados. Quando falamos desses aspectos estamos tratando e princípios que não dependem de tecnologia, linguagem, ferramenta e/ou processos. Estamos falando de leis que são quase tão imutáveis como as leis da física. Alguns desses princípios são conhecidos como padrões de projetos, ou o termo mais conhecido, em inglês, Design Patterns.

Existem vários padrões de projetos. Dentre os mais conhecidos estão os catalogados pelo *Gang of Four* (GoF). São padrões de criação, estruturais e comportamentais. Cada qual com o seu contexto especifico de aplicação. Dessa forma, a escolha de qual padrão utilizar depende muito do contexto. Mas antes de entrarmos nos detalhes de padrões, vamos aprender alguns conceitos arquiteturais de software.

Conceitos fundamentais de arquitetura de software

Enquanto a engenharia de software se preocupa com a produção de software de qualidade, a arquitetura se preocupa em produzir software que atendam a requisitos não funcionais. Esses requisitos geralmente são negligenciados por equipes iniciantes e fazem com que o custo e o tempo para a criação do software aumentem exponencialmente. Tais requisitos estão associados com escalabilidade, manutenabilidade, etc.

#ATENÇÃO#
Existe uma norma sobre requisitos não funcionais chamada NBR ISO/IEC 9126-1. Essa norma é importante pois categoriza os tipos de requisitos, padroniza a forma de avaliação e, por fim, facilita o entendimento e a comunicação.

#ATENÇÃO#

Um desses requisitos está relacionado com a manutenção do software. Um dos efeitos que sempre tentamos evitar ao realizar manutenções é a propagação de comportamentos indesejados em locais diferentes do que alteramos.

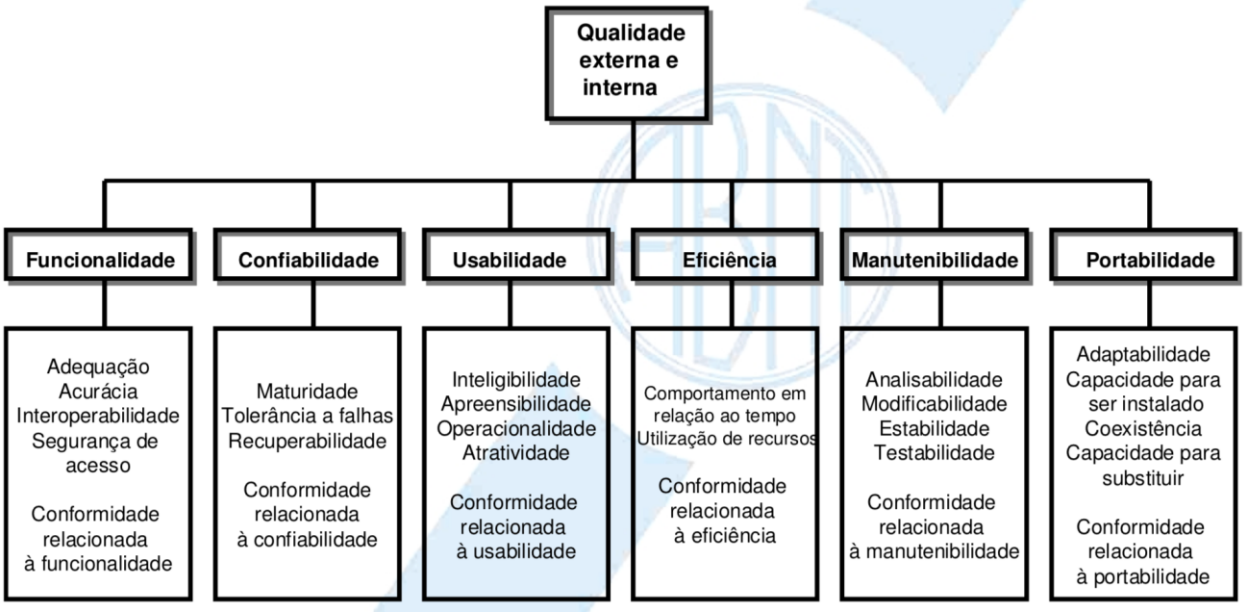

Figura 1 – Requisitos não Funcionais

Para alcançar esse beneficio precisamos estruturar nosso código de maneira a não permitir a propagação de efeitos colaterais. Com isso chegamos a dois princípios fundamentais de arquitetura que norteiam todos os padrões de projetos.

Acoplamento e coesão

Os dois conceitos fundamentais que todo software precisa incorporar são o acoplamento e a coesão (FREEMAN e FREEMAN, 2007). O primeiro conceito refere-se ao quanto os componentes do nosso software está ligado a outro. Por exemplo, imagine que temos uma classe conforme o exemplo abaixo:

```
public class BancoDados {
      public MySQLConnector conector = new MySQLConnector();
}
```

Agora imagine, que por qualquer motivo, mudamos a classe `MySQLConnector` para ter um outro comportamento. O que vai acontecer com o comportamento da classe `BancoDados`? Ele vai ser afetado e consequentemente todas as classes que fizerem referência a mesma. Esse fenômeno é conhecido como acoplamento. Quanto maior o nível de acoplamento, menor é a qualidade do software pois se torna quase impossível a manutenção se efeitos colaterais indesejáveis.

Outro principio fundamental é o de coesão. Mas o que isso significa? Coesão é manter cada elemento fazendo uma única tarefa, sendo o mais especifica possível. Vamos ver um exemplo, imagine uma classe conforme a abaixo:

```
public class ClasseQueFazMuitaCoisa {
      public void metodoQueFazIsso() {}
      public void metodoQueFazAquilo() {}
      public void metodoQueFazMaisIsso() {}
}
```

Essa classe possui muitas atribuições e, dessa forma, baixa coesão. Em outras palavras, como ela prove um conjunto enorme de tarefas (representadas por métodos) provavelmente ela será consumida por varias outras classes, aumentando dessa forma o acoplamento, fazendo com que sempre que seja necessária uma manutenção no software essa classe será impactada. Portanto, quanto menor a coesão dos elementos, menor a qualidade do software.

#RESUMO#
Devemos sempre procurar o menor acoplamento possível e a maior coesão possível. Com esses dois princípios em mente, pode ter certeza que aumentaremos muito a qualidade do software a ser produzido (LARMAN, 2007).
#RESUMO#

Padrões de projetos

Um padrão de projeto (ou em inglês, *Design Pattern*) é uma solução geral para um problema que ocorre com frequência dentro de contexto (FREEMAN e FREEMAN, 2007). Um padrão de projeto não é algo finalizado que pode ser diretamente transformado em código fonte, ele é um modelo de como resolver o problema e que pode ser usado em situações diferentes. Padrões são melhores práticas, formalizadas, que podem ser usadas para resolver problemas comuns ao projetar um software. Padrões de projeto orientados a objeto normalmente mostram relacionamentos e interações entre classes ou objetos, sem especificar as classes ou objetos da aplicação final que estão envolvidas (LARMAN, 2007).

Os padrões de projeto visam facilitar a reutilização de soluções de desenho, isto é, soluções na fase de projeto do software estabelecendo um vocabulário comum de desenho, facilitando comunicação, documentação e aprendizado dos sistemas de software. São classificados em 3 tipos (a) criacionais, (b) estruturais e (c) comportamentais (FREEMAN e FREEMAN, 2007). Cada um deles são aplicáveis a um contexto bem especifico e devem ser utilizados com poucas variações, pois isso pode fazer com que as consequências de implementação dos mesmos não sejam conhecidas. Com o padrão de projetos você terá vários benefícios dentre eles são: código mais enxuto, limpo, organizado, aumentar a qualidade e diminuir a complexidade do seu código. Os *Design Patterns* do GoF possuem o seguinte formato, sendo:

1. **Nome:** Uma identificação para o patterns
2. **Objetivo / Intenção:** Também conhecido como (Also Known As)
3. **Motivação:** Um cenário mostrando o problema e a necessidade
4. **Aplicabilidade:** Como identificar as situações nas quais os padrões são aplicáveis
5. **Estrutura:** Uma representação gráfica da estrutura das classes usando um diagrama de classes (UML)
6. **Consequências:** Vantagens e desvantagem

7. **Implementações:** Quais detalhes devemos nos preocupar quando implementamos o padrão. Detalhes de cada linguagem
8. **Usos conhecidos**
9. **Padrões Relacionados**

Vamos falar sobre os padrões GoF, dividindo-os pelas suas famílias e pelo seu escopo, vamos ver também uma breve descrição desses padrões para podermos ter uma pequena ideia do que cada um pode fazer para solucionar determinados problemas.

Criacionais

- ***Singleton***: assegura que somente um objeto de uma determinada classe seja criado em todo o projeto
- ***Abstract Factory***: permite que um cliente crie famílias de objetos sem especificar suas classes concretas
- ***Builder***: encapsular a construção de um produto e permitir que ele seja construído em etapas
- ***Prototype***: permite você criar novas instâncias simplesmente copiando instâncias existentes
- ***Factory Method***: as subclasses decidem quais classes concretas serão criadas

Estruturais

- ***Decorator***: envelopa um objeto para fornecer novos comportamentos;
- **Proxy**: envelopa um objeto para controlar o acesso a ele;
- FlyWeigth: uma instância de uma classe pode ser usada para fornecer muitas instâncias virtuais
- ***Facade***: simplifica a interface de um conjunto de classes
- ***Composite***: Os clientes tratam as coleções de objetos e os objetos individuais de maneira uniforme
- ***Bridge***: permite criar uma ponte para variar não apenas a sua implementação, como também as suas abstrações
- ***Adapter***: envelopa um objeto e fornece a ele uma interface diferente

Comportamentais

- ***Template Method***: As subclasses decidem como implementar os passos de um algoritmo
- ***Visitor***: permite acrescentar novos recursos a um composto de objetos e o encapsulamento não é importante
- ***Command***: encapsula uma solicitação como um objeto
- ***Strategy***: encapsula comportamentos intercambiáveis e usa a delegação para decidir qual deles será usado
- ***Chain of Responsability***: permite dar a mais de um objeto a oportunidade de processar uma solicitação

- ***Iterator***: fornece uma maneira de acessar sequencialmente uma coleção de objetos sem expor a sua implementação
- ***Mediator***: centraliza operações complexas de comunicação e controle entre objetos relacionados
- ***Memento***: permite restaurar um objeto a um dos seus estados prévios, por exemplo, quando o usuário seleciona um desfazer
- ***Interpreter***: permite construir um intérprete para uma linguagem
- ***State***: encapsula comportamentos baseados em estados e usa a delegação para alternar comportamentos
- ***Observer***: permite notificar outros objetos quando ocorre uma mudança de estado

Esses padrões podem ser classificados, do ponto de vista de escopo, dentro de classe e objeto.

		Propósito		
		Criação	**Estrutura**	**Comportamento**
Escopo	**Classe**	Factory Method	Class Adapter	Interpreter Template Method
	Classe	Abstract Factory Builder Prototype Singleton	Object Adapter Bridge Composite Decorator Façade Flyweight Proxy	Chain of Responsability Command Iterator Mediator Memento Observer State Strategy Visitor

Figura 2 – Classificação dos padrões segundo o GoF

Considerações Finais

Projetar software de qualidade é uma tarefa difícil (LARMAN, 2007). Mesmo que tenhamos as melhores ferramentas, processos e técnicas sempre devemos nos atentar para os princípios que norteiam esse tipo de trabalho. Alguns desses princípios são os padrões de projetos. Todos os padrões então apoiados em dois conceitos conhecidos como acoplamento e coesão. Ambos visam reduzir os efeitos de propagação de efeitos colaterais indesejáveis e melhorando o requisito não funcional de manutenabilidade.

UNIDADE 2 – PADRÕES CRIACIONAIS

Objetivos de aprendizagem

- Os padrões de projetos criacionais
- Os padrões *Abstract Factory, Builder, Factory Method, Prototype* e *Singleton*
- Como identificar a necessidade desses padrões

Os padrões de projetos criacionais

Os padrões de criação têm como intenção principal abstrair o processo de criação de objetos, ou seja, a sua instanciação. Desta maneira, o sistema não precisa se preocupar com questões sobre, como o objeto é criado, como é composto, qual a sua representação real. Quando se diz que o sistema não precisa se preocupar com a instanciação do objeto quer dizer que, se ocorrer alguma mudança neste ponto, o sistema em geral não será afetado. Isso é a famosa flexibilidade que os padrões de projeto buscam. Padrões de criação com escopo de classe vão utilizar herança para garantir que a flexibilidade. Por exemplo, o padrão *Factory Method* pode criar várias subclasses para criar o produto. Já os padrões com escopo de Objeto, como o *Prototype*, delegam para um objeto (no caso o protótipo) a responsabilidade de instanciar novos objetos (LARMAN, 2007).

Os padrões Abstract Factory, Builder, Factory Method, Prototype e Singleton

A seguir vamos estudar os padrões criacionais e como utilizá-los em projetos de software. Para o primeiro padrão, vamos adotar uma técnica diferente, e ir derivando o conceito intuitivo ao longo de alguns padrões até chegar no conceito pronto.

Factory Method

Esse padrão tem como objetivo isolar a criação de objetos de forma a reduzir o acoplamento entre classes e aumentar a coesão da classe responsável pela criação (LARMAN, 2007). Quando temos um conjunto de classes concretas e precisamos instanciá-lá de acordo com uma condição, normalmente usamos um código semelhante ao abaixo:

```
public class Pizzaria {
      Pizza pizza;
      public void escolher(String tipo) {
            if("queijo".equalsIgnoreCase(tipo)) {
                  pizza = new PizzaQueijo();
            }
            if("presunto".equalsIgnoreCase(tipo)) {
                  pizza = new PizzaPresunto();
            }
```

```
            pizza.preparar();
            pizza.assar();
      }
}
```

Um problema dessa classe é que em caso de mudanças nos sabores das pizzas, teremos que alterar o código da classe `Pizzaria`. Ou seja, a pizzaria está acoplada com as pizzas que produz. Isso não é uma boa estratégia de implementação pois sempre que algum novo sabor for adicionado ou removido, a classe `Pizzaria` deverá ser aberta e o código examinado e modificado. Dessa forma, devemos cuidar da criação dos objetos para evitar o acoplamento. Uma maneira é utilizar uma interface para isso. Quando codificados para interfaces, o código fica protegido contra alterações. Ele sempre funcionará para classes que implementam o polimorfismo. Com isso chegamos a um dos princípios de orientação a objetos (OO).

#REFLEXAO#

Identificar as partes que variam e separar do que continua igual é um dos princípios fundamentais dos padrões de projetos.

#REFLEXAO#

Vamos analisar o código anterior para aplicar esse princípio.

```
public class Pizzaria {
      Pizza pizza;
      public void escolher(String tipo) {
            if("queijo".equalsIgnoreCase(tipo)) {
                  pizza = new PizzaQueijo();
            }
            if("presunto".equalsIgnoreCase(tipo)) {
                  pizza = new PizzaPresunto();
            }
            pizza.preparar();
            pizza.assar();
      }
}
```

A parte do código em amarelo é a parte que muda e a parte em cinza tende a ser estática. Dessa forma, precisamos isolar o código em amarelo. E como fazer isso? Vamos criar uma classe que fabrica pizzas chamada de `PizzaFactory`.

```
public class PizzaFactory {
      public Pizza criar(String tipo) {
            if("queijo".equalsIgnoreCase(tipo)) {
                  return new PizzaQueijo();
            }
            if("presunto".equalsIgnoreCase(tipo)) {
                  return new PizzaPresunto();
            }
      }
}
```

Vamos adaptar essa fábrica de pizzas em nossa pizzaria.

```
public class Pizzaria {
	PizzaFactory factory;
	public Pizzaria(PizzaFactory factory) {
		this.factory = factory;
	}
	Pizza pizza;
	public void escolher(String tipo) {
		pizza = factory.criar(tipo);
		pizza.preparar();
		pizza.assar();
	}
}
```

Mas o que foi ganho com essa reorganização de código? Não passamos o problema para outro objeto? A diferença agora é que vários clientes podem usar a fábrica e as alterações ficam localizadas no código encapsulado. A `PizzaFactory` tornou o código da pizzaria fechado a modificações mas quebrou o vínculo entre a `Pizza` e a `Pizzaria`. Vamos criar uma forma de estabelecer um vínculo entre a Pizzaria e a forma de instanciar as pizzas mas mantendo a flexibilidade.

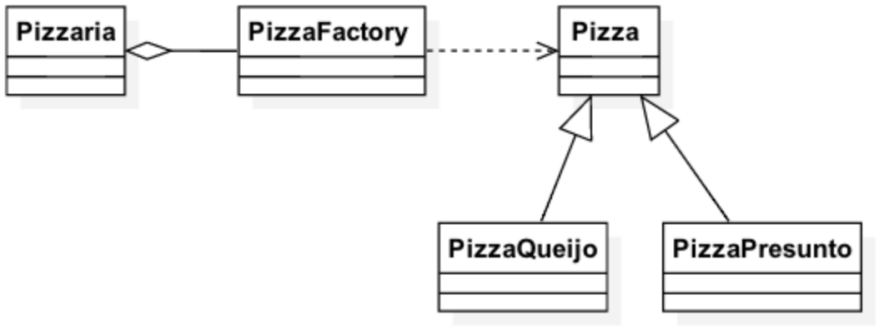

Figura 3 – O padrão de projetos Factory Method

Dessa forma, isolamos a pizzaria dos produtos que ela produz. Com isso podemos criar novas pizza sem a necessidade de alterar o código da pizzaria. Lembra do acoplamento e coesão? Reduzimos o acoplamento da pizzaria com as pizzas e aumentamos a coesão da pizzaria que ficou especializada em preparar e assar a pizza, independente de qual. A `PizzaFactory` não é exatamente um padrão de projetos A `PizzaFactory` é um programming idiom. Se quiséssemos abrir franquias de criação de pizza, como faríamos? Podemos criar diferentes classes para cada franquia e usar diferentes fábricas usando diferentes classes `PizzaFactory`.

```
PizzaFactorySP factory = new PizzaFactorySP();
Pizzaria pizzaria = new Pizzaria(factory);

PizzaFactoryRS factory = new PizzaFactoryRS();
Pizzaria pizzaria = new Pizzaria(factory);
```

Isso resolve o problema mas não temos garantias de que cada franquia segue o padrão (fugindo ao conceito de franquia).

```
public abstract class Pizzaria {
	Pizza pizza;
	public void escolher(String tipo) {
		pizza = criar(tipo);
		pizza.preparar();
		pizza.assar();
	}
	public abstract Pizza criar(String tipo);
}
```

Dessa forma, as pizzarias que estendem a classe `Pizzaria` que decidem qual o tipo de pizza criar.

```
public class PizzariaSP extends Pizzaria {
	public Pizza criar(String tipo) {
		return new PizzaQueijoSP();
	}
}
```

Vamos ver como fica nosso modelo?

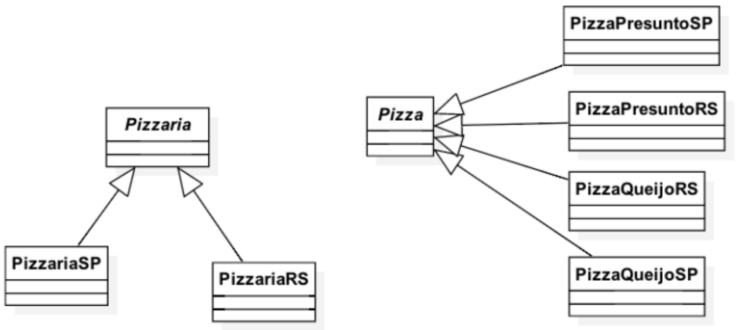

Figura 4 – O padrão de projetos Factory Method

Pronto! De maneira intuitiva chegamos ao nosso primeiro padrão de projetos: o *Factory Method*. Vamos ver a formalização do mesmo:

#DEFINIÇÃO#

Define uma interface para criar um objeto, mas permite as classes decidir qual classe instanciar. O *Factory Method* permite uma classe deferir a instanciação para subclasses (GAMMA et al., 2000).

DEFINIÇÃO

Com isso, podemos passar as implementações para cada uma das classes

concretas ao mesmo tempo que mantemos a padronização entre elas. Além disso, isso reduz o acoplamento pois referenciamos a classe abstrata. Dessa maneira, fechamos a dedução, de forma intuitiva, para o nosso primeiro padrão de projetos. O mesmo mecanismo pode ser utilizado para os demais padrões. Daqui pra frente, vamos omitir esse mecanismo e apresentar os padrões de maneira direta e como o mesmo pode ser aplicado.

Abstract Factory

Vamos estudar agora um outro padrão de criação, o *Abstract Factory*. Para isso, vamos analisar um outro principio de orientação a objetos.

#REFLEXÃO#

Dependa de abstrações e não de classes concretas (LARMAN, 2007).

#REFLEXÃO#

Agora, como podemos aplicar esse principio ao problema das pizzas? Precisamos desvincular a iniciação de classes concretas da classe `Pizza` da classe `Pizzaria`. O objetivo é que a classe `Pizzaria` dependa de uma abstração. Dessa forma, vamos criar uma fábrica abstrata, conforme o código abaixo.

```
public interface IngredientePizzaFactory {
	public Massa selecionarMassa();
	public Molho selecionarMolho();
	public Queijo selecionarQueijo();
}
```

Agora vamos ver como ficaria a implementação de uma `Pizza` com as preferencias de São Paulo por exemplo.

```
public class IngredientePizzaSP implements IngredientePizzaFactory {
	public Massa selecionarMassa(){
		return new MassaPan();
	}
	public Molho selecionarMolho(){
		return new MolhoVermelho();
	}
	public Queijo selecionarQueijo(){
		return new QueijoProvolone();
	}
}
```

As pizzas agora podem ser uma classe abstrata. Um produto abstrato formado por componentes abstratos, no caso, interfaces. Vamos ver o código.

```
public abstract class Pizza {
	String nome;
	Massa massa;
	Molho molho;
	Queijo queijo;
	void preparar() {
		System.out.println("Selecione os ingredientes");
	}
	void assar() {
		System.out.println("Assar por 25 minutos");
	}
}
```

Agora podemos ter as pizzas que estendem da pizza abstrata. Além disso, as preferencias das pizzas podem ser regionalizadas via a especificação de uma fábrica de ingredientes especifica.

```
public class PizzaQueijoSP extends Pizza {
	IngrediantePizzaFactory factory;
	public PizzaQueijoSP(IngrediantePizzaFactory factory) {
		this.factory = factory;
	}
	void preparar() {
		massa = factoy.selecionarMassa();
		molho = factory.selecionarMolho();
	}
}
```

Vamos ver como ficaria nosso diagrama de classes?

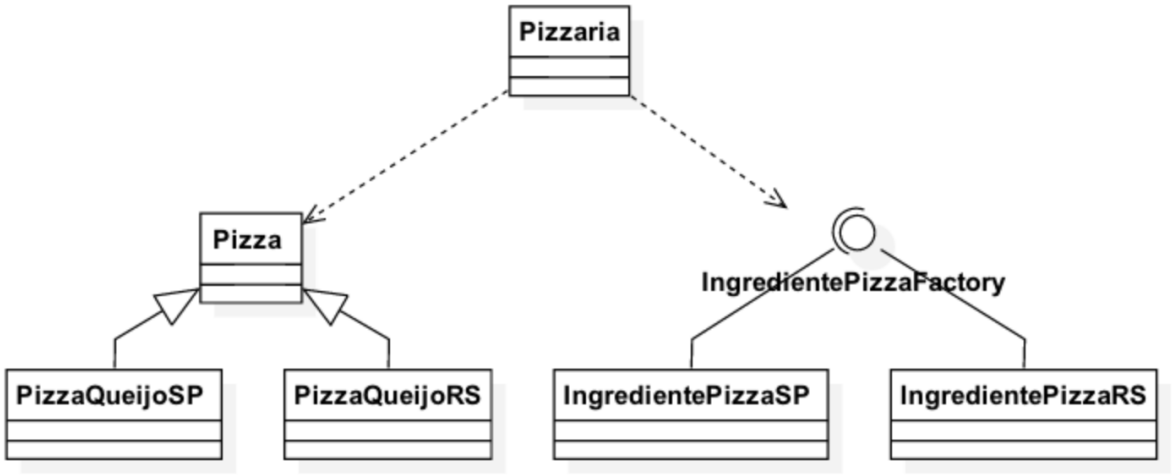

Figura 5 – O padrão de projetos Abstract Factory

Pronto! Temos o nosso padrão *Abstract Factory*. Vamos ver o mesmo padrão em uma forma mais geral?

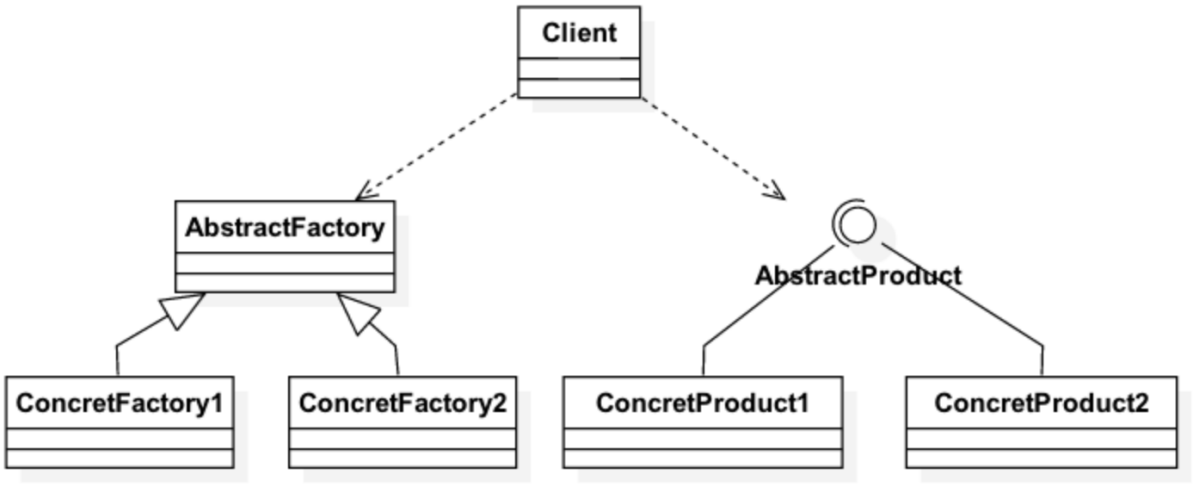

Figura 6 – O padrão de projetos Abstract Factory

Com isso temos um padrão de projetos que pode ser aplicado ao contexto de criação baseados em fábricas abstratas.

Builder

O *Builder* é usado para encapsular a lógica de construção de objetos. Este padrão é utilizado quando o processo de construção é complexo ou se trata da construção múltipla de uma mesma classe. Objetos complexos passam a ser construídos a partir de partes geradas de outros objetos, o que demanda uma necessidade maior de esforço em relação a sua construção. Desta forma, podemos

dizer que a aplicação precisará de um mecanismo de construção independente das classes que o compõem (LARMAN, 2007).

O *Builder* é composto por quatro componentes sendo a Interface (ou classe abstrata) `Builder`, o `Concrete Builder` (construtor concreto), o `Director` (Diretor) e o `Product` (produto). Vejamos a responsabilidade de cada um:

- **Classe Builder** – especifica uma interface ou uma classe abstrata para a criação das partes de um objeto a fim de criar corretamente o produto (Product)
- **Concrete Builder** – responsável pela construção e pela montagem das partes do produto por meio da implementação da classe builder
- **Director** – controla o algoritmo responsável por gerar o objeto do produto final. Um objeto Director é instanciado e seus métodos construtores são chamados. O método inclui um parâmetro para capturar objetos específicos do tipo Concrete Builder que serão então utilizados para gerar o produto (product). Dessa forma, o director, chama os métodos do concrete builder na ordem correta para gerar o objeto produto
- **Product** – o product representa o objeto complexo que está sendo construído. O concrete builder então constrói a representação interna do produto e define o processo pelo qual essa classe será montada. Na classe product são incluídas outras classes que definem as partes que a constituem, dentre elas, as interfaces para a montagem das partes no resultado final.

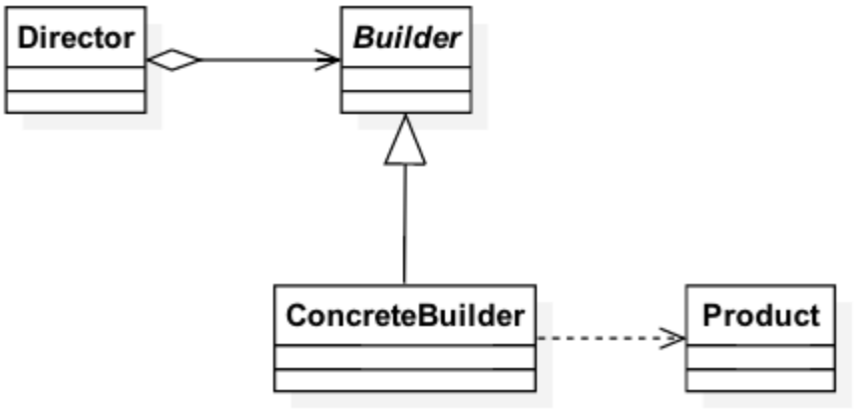

Figura 7 – O padrão de projetos Builder

O `Client` chamará o método `main()` da aplicação, iniciando assim as classes Builder e Director. A classe `Builder` representa o nosso objeto complexo que precisa ser construído em termos de objetos e tipos mais simples. O construtor da classe `Director` recebe um objeto `Builder` como sendo um parâmetro através do cliente sendo responsável por chamar os métodos da classe `Builder`. Para fornecer a classe cliente uma interface para os construtores concretos, a classe Builder deve ser uma classe abstrata. Desta forma, podemos adicionar novos tipos apenas definindo a estrutura e reutilizando a lógica para o processo real da construção. O cliente é o único que precisa saber sobre os novos tipos, já a classe

`Director`, precisa saber apenas quais os métodos que precisará chamar.

Prototype

O *Prototype* é outro padrão de criação. Assim seu intuito principal é criar objetos. Este intuito é muito parecido com todos os outros padrões criacionais, tornando todos eles bem semelhantes. Vamos mostrar como utilizá-lo com um problema de criação de carros. O problema consiste em uma lista de carros que o cliente precisa utilizar, mas que só serão conhecidos em tempo de execução. Vamos analisar então como o problema pode ser selecionado utilizando o padrão *Prototype*.

#DEFINIÇÃO#

Especificar tipos de objetos a serem criados usando uma instância protótipo e criar novos objetos pela cópia desse protótipo (GAMMA et al., 2000).

DEFINIÇÃO#

Podemos perceber como o padrão vai resolver o problema. Precisamos criar novos objetos a partir de uma instância protótipo, que vai realizar uma cópia de si mesmo e retornar para o novo objeto. A estrutura do padrão inicia então com definição dos objetos protótipos. Para garantir a flexibilidade do sistema, vamos criar a classe base de todos os protótipos:

```
public abstract class CarroPrototype {
	protected double valorCompra;
	public abstract String info();
	public abstract CarroPrototype clonar();
	public double getValorCompra() {
		return valorCompra;
	}
	public void setValorCompra(double valorCompra) {
		this.valorCompra = valorCompra;
	}
}
```

Definimos a partir dela que todos os carros terão um valor de compra, que será manipulado por um conjunto de *getters* e *setters*. Também garantimos que todos eles possuem os métodos para exibir informações e para realizar a cópia do objeto. Para exemplificar uma classe protótipo concreta, vejamos a seguinte classe:

```
public class FiestaPrototype extends CarroPrototype {
```

```
    protected FiestaPrototype(FiestaPrototype fiestaPrototype) {
        this.valorCompra = fiestaPrototype.getValorCompra();
    }
    public FiestaPrototype() {
        valorCompra = 0.0;
    }
    @Override
    public String exibirInfo() {
        return "Fiesta: Ford" + "Valor: " + getValorCompra();
    }
    @Override
    public CarroPrototype clonar() {
        return new FiestaPrototype(this);
    }
}
```

Note que no início são definidos dois construtores, um protegido e outro público. O construtor protegido recebe como parâmetro um objeto da própria classe protótipo. Este é o chamado construtor por cópia, que recebe um outro objeto da mesma classe e cria um novo objeto com os mesmos valores nos atributos. A necessidade deste construtor será vista no método de cópia. O método `info()` exibe as informações referentes ao carro, retornando uma string com as informações. Ao final o método de clonagem retorna um novo objeto da classe protótipo concreta. Para garantir que será retornado um novo objeto, vamos utilizar o construtor por cópia definido anteriormente. Agora, sempre que for preciso criar um novo objeto `FiestaPrototype` vamos utilizar um único protótipo. Pense nesta operação como sendo um método fábrica, para evitar que o cliente fique responsável pela criação dos objetos e apenas utilize os objetos. Para verificar esta propriedade, vamos analisar o código cliente a seguir, que faz uso do padrão prototype.

```
public static void main(String[] args) {
    PalioPrototype prototipoPalio = new PalioPrototype();
    CarroPrototype palioNovo = prototipoPalio.clonar();
    palioNovo.setValorCompra(37900.0);
    CarroPrototype palioUsado = prototipoPalio.clonar();
    palioUsado.setValorCompra(31000.0);
    System.out.println(palioNovo.exibirInfo());
    System.out.println(palioUsado.exibirInfo());
}
```

Observe com atenção o cliente. Criamos dois protótipos de carros, cada um deles é criado utilizando o prótipo `PalioPrototype` instanciado anteriormente. Ou seja, a partir de uma instância de um protótipo é possível criar vários objetos a partir

da cópia deste protótipo. Outro detalhe é que, se a operação de clonagem não fosse feita utilizando o construtor de cópia, quando a chamada ao `setValorCompra` fosse feita, ela mudaria as duas instâncias, pois elas referenciariam ao mesmo objeto. O diagrama UML que representa a estrutura do padrão Prototype utilizada nesta solução é o seguinte:

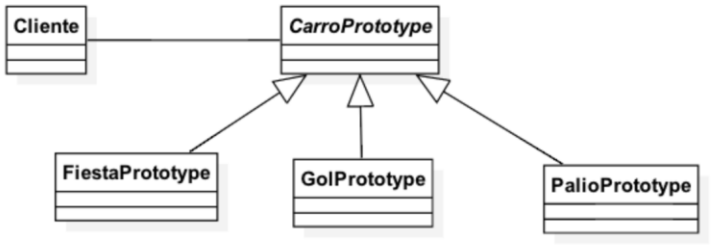

Figura 8 – O padrão de projetos Prototype

Singleton

Como já vimos antes nos padrões *Abstract Factory* e *Factory Method* é possível criar um objeto que fique responsável por criar outros objetos. Desta maneira, nós centralizamos a criação destes objetos e podemos ter mais controle sobre eles. Imagine o exemplo da fábrica de carros do padrão *Factory Method*. A classe fábrica centraliza a criação de objetos carro. Por exemplo, se fosse necessário armazenar quantos carros foram criados, para elaborar um relatório de quais foram os carros mais vendidos, seria bem simples não? Bastaria adicionar um contador para cada tipo de carro e, ao executar o método que cria um carro, incrementar o contador referente a ele (LARMAN, 2007). Vamos então ver como ficaria a nossa classe fábrica, para simplificar, apenas retornamos uma String para dizer que o carro foi criado:

```
public class FabricaDeCarro {
    protected int totalCarrosFiat;
    protected int totalCarrosFord;
    protected int totalCarrosVolks;
    public String criarCarroVolks() {
        return new String("Carro Volks #" + totalCarrosVolks++ + "
criado.");
    }
    public String criarCarroFord() {
        return new String("Carro Ford #" + totalCarrosFord++ + " criado.");
    }
    public String criarCarroFiat() {
        return new String("Carro Fiat #" + totalCarrosFiat++ + " criado.");
    }
```

```
    public String gerarRelatorio() {
        return new String("Total de carros Fiat vendidos: " +
totalCarrosFiat
                + "\nTotal de carros Ford vendidos: " + totalCarrosFord
                + "\nTotal de carros Volks vendidos: " + totalCarrosVolks);
    }
}
```

A cada carro criado nós incrementamos o contador e exibimos a informação que o carro foi criado. No final adicionamos um método que gera um relatório e mostra todas as vendas. O código cliente seria algo do tipo então:

```
public static void main(String[] args) {
    FabricaDeCarro fabrica = new FabricaDeCarro();
    System.out.println(fabrica.criarCarroFiat());
    System.out.println(fabrica.criarCarroFord());
    System.out.println(fabrica.criarCarroVolks());
    System.out.println(fabrica.gerarRelatorio());
}
```

Uma excelente solução não? Agora imagine que, em algum outro lugar do código acontece isso:

```
fabrica = new FabricaDeCarro();
System.out.println(fabrica.gerarRelatorio());
```

Todos os dados até o momento foram APAGADOS! Todas as informações estão ligadas a uma instância, quando alteramos a instância, perdemos todas as informações. Qual o centro do problema então? Temos que proteger que o objeto seja instanciado. Como fazer isso?

#DEFINIÇÃO#

Garantir que uma classe tenha somente uma instância e fornece um ponto global de acesso para a mesma (GAMMA et al., 2000).

#DEFINIÇÃO#

Pronto, encontramos a solução! Com o uso do padrão garantimos que só teremos uma instância da classe fábrica. O padrão é extremamente simples. Para utilizá-lo precisamos apenas de uma referência para um objeto fábrica, dentro da própria fábrica:

```
public class FabricaDeCarro{
```

```
    public static FabricaDeCarro instancia;
}
```

Não deixar o construtor com acesso público:

```
public class FabricaDeCarro {
    public static FabricaDeCarro instancia;
    protected FabricaDeCarro() {
    }
}
```

E por fim um método que retorna à referência para a fábrica:

```
public class FabricaDeCarro {
    public static FabricaDeCarro instancia;
    protected FabricaDeCarro() {
    }
    public static FabricaDeCarro getInstancia() {
        if (instancia == null)
            instancia = new FabricaDeCarro();
        return instancia;
    }
}
```

Pronto, esta agora é uma classe *Singleton*. Ao proteger o construtor nós evitamos que esta classe possa ser instanciada em qualquer outro lugar do código que não seja a própria classe. O código cliente ficaria da seguinte forma:

```
public static void main(String[] args) {
    FabricaDeCarro fabrica = FabricaDeCarro.getInstancia();
    System.out.println(fabrica.criarCarroFiat());
    System.out.println(fabrica.criarCarroFord());
    System.out.println(fabrica.criarCarroVolks());
    System.out.println(fabrica.gerarRelatorio());
}
```

Perceba que, mesmo que em outra parte do código apareça algo do tipo:

```
fabrica = FabricaDeCarro.getInstancia();
System.out.println(fabrica.gerarRelatorio());
```

Não será perdido nenhuma informação, pois não houve uma nova

instanciação. O método `getInstancia()` verifica se a referência é válida e, se preciso, instância ela e depois retorna para o código cliente. Seria possível, no código cliente, nem mesmo utilizar uma referência para uma fábrica, veja o código a seguir:

```
System.out.println(FabricaDeCarro.getInstancia().gerarRelatorio());
```

Como identificar a necessidade desses padrões

A criação de objetos tem uma grande importância para o bom desempenho e estabilidade dos softwares orientados a objetos. Caso os objetos não sejam criados da maneira adequada, podemos ter atributos vazios que levam aos famigerados erros de `NullPointerExcepetion`. Além disso, a criação de objetos em duplicidade pode levar a leituras incorretas, consumo excessivo de memória e instabilidades no geral, de difícil identificação das causas e, consequentemente, manutenção. Dessa forma, ao projetar a criação de objetos sempre que possível lance mão de projetos criacionais. Eles vão facilitar a manutenção e a estabilidade do seu software.

Considerações Finais

Neste estudo entendemos os padrões criacionais e como os mesmos podem ser aplicados aos projetos de software orientados a objetos. Esses padrões são fundamentais para a estabilidade do software e facilitam o entendimento e comunicação ao longo do ciclo de vida do produto.

UNIDADE 3 – PADRÕES ESTRUTURAIS

Objetivos de aprendizagem

- Os padrões de projetos estruturais
- Os padrões *Adapter, Bridge, Composite, Decorator, Façade (*ou *Facade), Business Delegate, Flyweight* e *Proxy*
- Como identificar a necessidade desses padrões

Os padrões de projetos estruturais

Os padrões estruturais vão se preocupar em como as classes e objetos são compostos, ou seja, como é a sua estrutura. O objetivo destes padrões e facilitar o design do sistema identificando maneiras de realizar o relacionamento entre as entidades, deixando o desenvolvedor livre desta preocupação. Os padrões com escopo de classe utilizam a herança para compor implementações ou interfaces. O padrão Adapter, por exemplo, pode definir uma nova interface para adaptar duas outras já existentes, assim uma nova classe é criada para adaptar uma interface a outra. Os padrões com escopo de objeto utilizam a composição de objetos para definir uma estrutura. Por exemplo, o padrão *Composite* define (explicitamente) uma estrutura de hierárquica para classes primitivas e compostas em um objeto (LARMAN, 2007).

Os padrões Adapter, Bridge, Composite, Decorator, Façade (ou Facade), Business Delegate, Flyweight e Proxy

Adapter

A utilização de *frameworks* é muito comum atualmente, devido a comodidade e facilidade de utilização. No entanto, discute-se muito sobre a postura dos programadores frente aos *frameworks*. O ponto de discussão é que a utilização de *frameworks* deve ser considerada algo fora do projeto original, e não um componente interno que torna todo o sistema dependente de um *framework* específico. Vamos considerar o seguinte exemplo: é preciso fazer um sistema que manipule imagens, para isto será utilizado uma API que oferece essas funcionalidades. Suponhamos que será necessário ter um método para carregar a imagem de um arquivo e outro para exibir a imagem na tela. Como podemos então construir o sistema de maneira que ele fique independente de qual API será utilizada? Para exemplificar considere as duas classes a seguir que representam as APIs utilizadas. Como estas classes fazem parte da API não temos condições de alterá-las.

```
public class OpenGLImagem {
    public void glCarregarImagem(String arquivo) {
        System.out.println("Imagem " + arquivo + " carregada.");
    }
```

```
    public void glDesenharImagem(int X, int Y) {
        System.out.println("OpenGL Image desenhada");
    }
}

public class SDLSurface {
    public void SDLCarregarSurface(String arquivo) {
        System.out.println("Imagem " + arquivo + " carregada.");
    }
    public void SDLDesenharSurface(int largura, int altura, int X, int Y) {
        System.out.println("SDLSurface desenhada");
    }
}
```

Como podemos então, além de deixar o sistema independente, unificar uma interface de acesso para qualquer API? Veja que a assinatura dos métodos é bem diferente, desde o nome até quantidade de parâmetros. Poderíamos utilizar o *Template Method* para definir uma interface e deixar cada subclasse redefinir algumas operações, no entanto não existe aqui a ideia de um algoritmo comum. Vejamos agora o que é o padrão *Adapter*.

#DEFINIÇÃO#

Converter a interface de uma classe em outra interface, esperada pelo cliente. O *Adapter* permite que interfaces incompatíveis trabalhem em conjunto – o que, de outra forma, seria impossível (GAMMA et al., 2000).
#DEFINIÇÃO#

Ou seja, dado um conjunto de classes com mesma responsabilidade, mas interfaces diferentes, utilizamos o Adapter para unificar o acesso a qualquer uma delas. Precisamos então, inicialmente, fornecer uma interface comum para o cliente, oferecendo o comportamento que ele necessita:

```
public interface ImagemTarget {
    void carregarImagem(String nomeDoArquivo);
    void desenharImagem(int posX, int posY, int largura, int altura);
}
```

Agora vamos definir os adaptadores:

```
public class OpenGLImageAdapter extends OpenGLImage implements ImagemTarget
{
    @Override
    public void carregarImagem(String nomeDoArquivo) {
        glCarregarImagem(nomeDoArquivo);
    }
    @Override
    public void desenharImagem(int posX, int posY, int largura, int altura)
{
        glDesenharImagem(posX, posY);
    }

}
public class SDLImagemAdapter extends SDL_Surface implements ImagemTarget {
    @Override
    public void carregarImagem(String nomeDoArquivo) {
```

```
        SDL_CarregarSurface(nomeDoArquivo);
    }
    @Override
    public void desenharImagem(int posX, int posY, int largura, int altura)
{
        SDL_DesenharSurface(largura, altura, posX, posY);
    }
}
public static void main(String[] args) {
    ImagemTarget imagem = new SDLImagemAdapter();
    imagem.carregarImagem("imagem.png");
    imagem.desenharImagem(0, 0, 10, 10);
    imagem = new OpenGLImageAdapter();
    imagem.carregarImagem("teste.png");
    imagem.desenharImagem(0, 0, 10, 10);
}
```

Qualquer mudança em qual API será utilizada é facilmente feita, sem a necessidade de alterar o software inteiro. Caso fosse necessário utilizar uma nova API também seria simples, bastaria criar o adaptador para a API e utilizá-lo quando fosse necessário. Vejamos o diagrama UML abaixo:

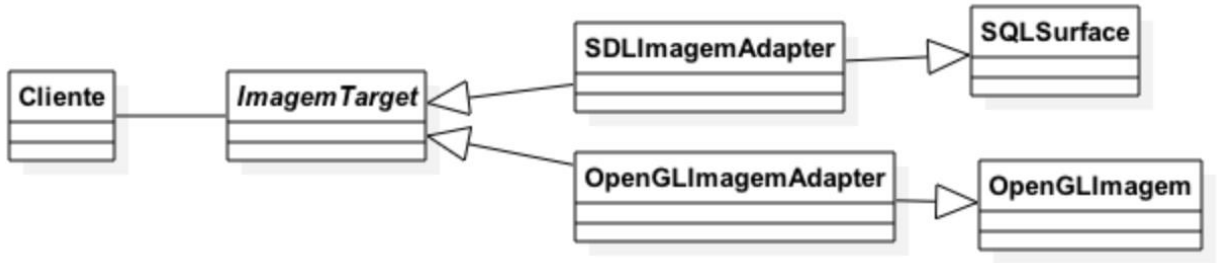

Figura 9 – O padrão de projetos Adapter

Bridge

Como vimos na seção anterior sobre o *Adapter*, utilizamos o padrão para liberar o nosso código cliente de detalhes de implementação de Frameworks/API/Biblioteca específicas. Suponha agora que é necessário fazer um programa que vá funcionar em várias plataformas, por exemplo, *Windows*, *Linux*, *Mac*, etc. O programa fará uso de diversas abstrações de janelas gráficas, por exemplo, janela de diálogo, janela de aviso, janela de erro, etc. Como podemos representar esta situação? A utilização de um *Adapter* para adaptar as janelas para as diversas plataformas parece ser boa, criaríamos um *Adapter* para *Windows*, *Linux* e *Mac* e então utilizaríamos de acordo com a necessidade (LARMAN, 2007). No entanto, teríamos que utilizar o adaptador de cada uma das plataformas para cada um dos tipos de abstrações de janelas. Por exemplo, para uma janela de diálogo, teríamos um *Adapter* para *Windows*, *Linux* e *Mac*, da mesma forma para as outras janelas. Vamos então partir para uma solução bem legal, o padrão *Bridge*!

#DEFINIÇÃO#

Desacoplar uma abstração da sua implementação, de modo que as duas possam variar independentemente (GAMMA et al., 2000).

#DEFINIÇÃO#

Ou seja, o *Bridge* fornece um nível de abstração maior que o *Adapter*, pois são separadas as implementações e as abstrações, permitindo que cada uma varie independentemente. Para o exemplo as implementações seriam as classes de `Janela` das plataformas. Vamos iniciar construindo-as, de maneira bem simples. A primeira classe será a interface comum a todas as implementações, chamadas de `JanelaImplementada`:

```
public interface JanelaImplementada {
    void desenharJanela(String titulo);
    void desenharBotao(String titulo);
}
```

Ou seja, nessa classe definimos que todas as janelas desenham uma janela e um botão. Vamos ver agora a classe concreta que desenha a janela na plataforma *Windows*:

```
public class JanelaWindows implements JanelaImplementada {
    @Override
    public void desenharJanela(String titulo) {
        System.out.println(titulo + " - Janela Windows");
    }
    @Override
    public void desenharBotao(String titulo) {
        System.out.println(titulo + " - Botão Windows");
    }
}
```

Também uma classe bem simples, apenas vamos exibir uma mensagem no terminal para saber que tudo correu bem. Agora vamos então para as abstrações. Elas são abstrações pois não definem uma janela específica, como a `JanelaWindows`, no entanto utilizarão os métodos destas janelas concretas para construir suas janelas. Vamos então iniciar a construção da classe abstrata que vai fornecer uma interface de acesso comum para as abstrações de janelas:

```
public abstract class JanelaAbstrata {
    protected JanelaImplementada janela;
    public JanelaAbstrata(JanelaImplementada j) {
        janela = j;
    }
    public void desenharJanela(String titulo) {
        janela.desenharJanela(titulo);
    }
    public void desenharBotao(String titulo) {
        janela.desenharBotao(titulo);
    }
    public abstract void desenhar();
}
```

Essa classe possui uma referência para a interface das janelas implementadas, com isso conseguimos variar a implementação de maneira bem simples. Agora veja o exemplo da classe de `JanelaDialogo`, que abstrai uma janela de diálogo para todas as plataformas:

```
public class JanelaDialogo extends JanelaAbstrata {
    public JanelaDialogo(JanelaImplementada j) {
        super(j);
    }
    @Override
    public void desenhar() {
        desenharJanela("Janela de Diálogo");
        desenharBotao("Botão Sim");
        desenharBotao("Botão Não");
        desenharBotao("Botão Cancelar");
    }
}
```

Uma janela de diálogo exibe sempre três botões: Sim, Não e Cancelar. Ou seja, independente de qual plataforma se está utilizando, a abstração é sempre a mesma. Para uma janela de aviso por exemplo, bastaria um botão Ok, então sua implementação seria algo do tipo:

```
public class JanelaAviso extends JanelaAbstrata {
    public JanelaAviso(JanelaImplementada j) {
        super(j);
    }
    @Override
    public void desenhar() {
        desenharJanela("Janela de Aviso");
        desenharBotao("Ok");
    }
}
```

Vamos ver então como seria o cliente da nossa aplicação. Ele fará uso apenas da classe que define uma `Janela`, assim poderá ficar livre de quaisquer detalhes de abstrações ou de implementações:

```
public static void main(String[] args) {
    JanelaAbstrata janela = new JanelaDialogo(new JanelaLinux());
    janela.desenhar();
    janela = new JanelaAviso(new JanelaLinux());
    janela.desenhar();
    janela = new JanelaDialogo(new JanelaWindows());
    janela.desenhar();
}
```

Note que é bem simples realizar trocas entre as abstrações de janelas e de plataformas. O diagrama UML para este exemplo seria algo do tipo:

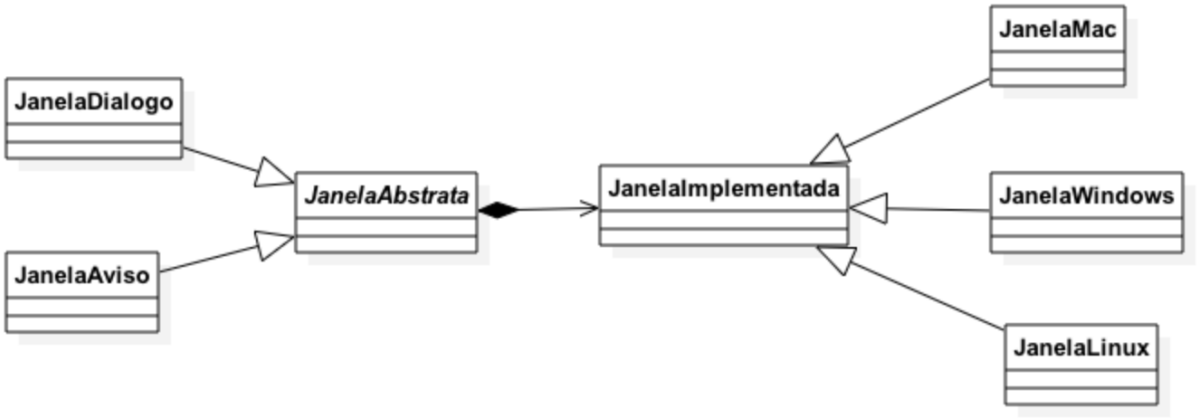

Figura 10 – O padrão de projetos Bridge

Composite

Imagine que você está fazendo um sistema de gerenciamento de arquivos. Como você já sabe é possível criar arquivos concretos (vídeos, textos, imagens, etc.) e arquivos pastas, que armazenam outros arquivos. O problema é o mesmo, como fazer um design que atenda estes requerimentos? Uma solução? Podemos criar uma classe que representa arquivos que são `Pastas`, estas pastas teriam uma lista de arquivos concretos e uma lista de arquivos de pastas. Então poderíamos adicionar pastas e arquivos em uma pasta e, a partir dela navegar pelas suas pastas e seus arquivos.

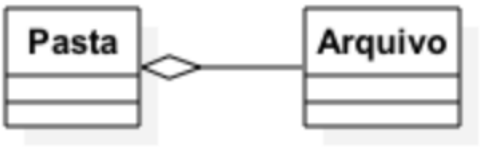

Figura 11 – O padrão de projetos Composite

O problema com este *design* é a interface que esta classe deverá ter. Como são duas listas diferentes precisaríamos de métodos específicos para tratar cada uma delas, ou seja, um método para inserir arquivos e outro para inserir pastas, um método para excluir pastas e outro para excluir arquivos, e por aí vai. Sempre que quisermos inserir uma nova funcionalidade no gerenciador, precisaremos criar a mesma funcionalidade para arquivos e pastas. Além disso, sempre que quisermos percorrer uma pasta será necessário percorrer as duas listas, mesmo que vazias. Bom, vamos pensar em outra solução. E se utilizássemos uma classe base `Arquivo` para todos os arquivos, assim precisaríamos apenas de uma lista e de um conjunto de funções. Vejamos abaixo:

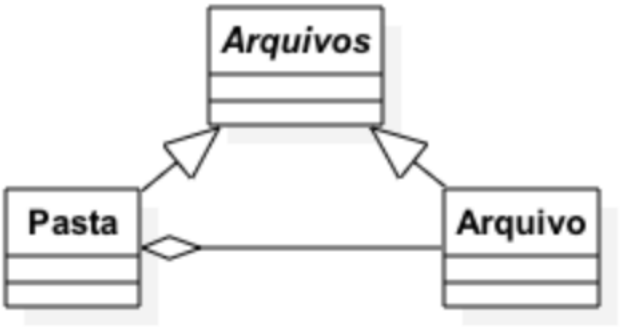

Figura 12 – O padrão de projetos Composite

Pronto, resolvido o problema dos métodos duplicados. E agora, será que está tudo bem? Como faríamos a diferenciação entre um `Arquivo` e uma `Pasta`? Poderíamos utilizar o `instaceof` e verificar qual o tipo do objeto, o problema é que seria necessário fazer isso SEMPRE, pois não poderíamos confiar que, dado um objeto qualquer, ele é um arquivo ou uma pasta! Sempre teríamos que fazer isso:

```
if(arquivo instanceof Arquivo){
    // Código para tratar arquivos de video
} else if(arquivo instanceof Pasta){
    // Código para tratar arquivos de audio
}
```

Ok, então vamos ver uma boa solução para o problema: o padrão *Composite*!

#DEFINIÇÃO#

Compor objetos em estruturas de árvore para representar hierarquia partes-todo. *Composite* permite aos clientes tratarem de maneira uniforme objetos individuais e composições de objetos (GAMMA et al., 2000).
#DEFINIÇÃO#

Bom, desta vez a intenção não está tão bem clara. A estrutura de árvore será explicada mais adiante, no momento o que interessa é a segunda parte da intenção: tratar de maneira uniforme objetos individuais. Como o nosso problema era uniformizar o acesso aos arquivos e pastas, provavelmente o *Composite* seja uma boa solução. A ideia do *Composite* é criar uma classe base que contém toda a interface necessária para todos os elementos e criar um elemento especial que agrega outros elementos. Vamos trazer para o nosso exemplo inicial para tentar esclarecer:

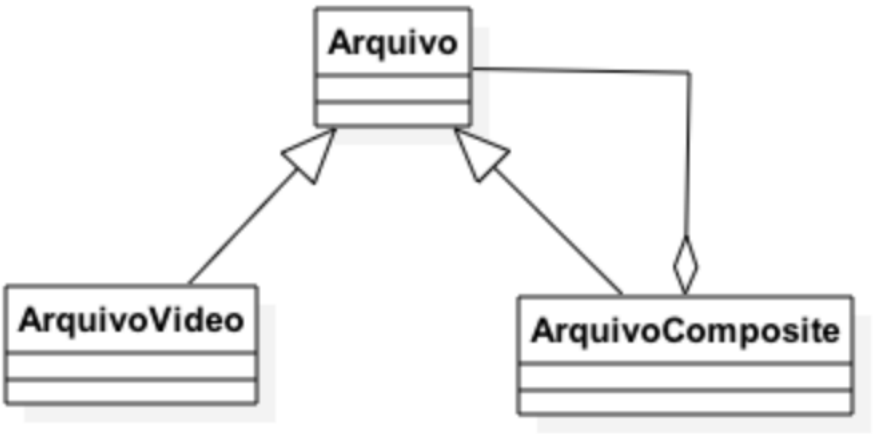

Figura 13 – O padrão de projetos Composite

A classe base `Arquivo` implementa todos os métodos necessários para arquivos e pastas, no entanto considera como implementação padrão a do arquivo, ou seja, caso o usuário tente inserir um arquivo em outro arquivo uma exceção será disparada. Veja o código da classe abaixo:

```
public abstract class ArquivoComponent {
    String nomeDoArquivo;
    public void printNomeDoArquivo() {
        System.out.println(this.nomeDoArquivo);
    }
    public String getNomeDoArquivo() {
        return this.nomeDoArquivo;
    }
    public void adicionar(ArquivoComponent novoArquivo) throws Exception {
        throw new Exception("Não pode inserir em: " + this.nomeDoArquivo)
    }
    public void remover(String nomeDoArquivo) throws Exception {
        throw new Exception("Não pode remover em: " + this.nomeDoArquivo);
    }
    public ArquivoComponent getArquivo(String nomeArquivo) throws Exception
{
        throw new Exception("Não pode pesquisar em: " + this.nomeArquivo)
    }
}
```

Uma vez que tudo foi definido nesta classe, para criar um arquivo de vídeo por exemplo, basta implementar o construtor:

```
public class ArquivoVideo extends ArquivoComponent {
    public ArquivoVideo(String nomeDoArquivo) {
        this.nomeDoArquivo = nomeDoArquivo;
    }
}
```

Já na classe que representa a `Pasta` nós sobrescrevemos o comportamento padrão e repassamos a chamada para todos os arquivos, sejam arquivos ou pastas, como podemos ver a seguir:

```
public class ArquivoComposite extends ArquivoComponent {
```

```
    ArrayList<ArquivoComponent>          arquivos          =          new
ArrayList<ArquivoComponent>();
    public ArquivoComposite(String nomeDoArquivo) {
        this.nomeDoArquivo = nomeDoArquivo;
    }
    @Override
    public void printNomeDoArquivo() {
        System.out.println(this.nomeDoArquivo);
        for (ArquivoComponent arquivoTmp : arquivos) {
            arquivoTmp.printNomeDoArquivo();
        }
    }
    @Override
    public void adicionar(ArquivoComponent novoArquivo) {
        this.arquivos.add(novoArquivo);
    }
    @Override
    public void remover(String nomeDoArquivo) throws Exception {
        for (ArquivoComponent arquivoTmp : arquivos) {
            if (arquivoTmp.getNomeDoArquivo() == nomeDoArquivo) {
                this.arquivos.remove(arquivoTmp);
                return;
            }
        }
        throw new Exception("Não existe este arquivo");
    }
    @Override
    public ArquivoComponent getArquivo(String nomeArquivo) throws Exception
{
        for (ArquivoComponent arquivoTmp : arquivos) {
            if (arquivoTmp.getNomeDoArquivo() == nomeArquivo) {
                return arquivoTmp;
            }
        }
        throw new Exception("Não existe este arquivo");
    }
}
```

Com isto não é necessário conhecer a implementação dos objetos concretos, muito menos fazer `cast`. Veja como poderíamos utilizar o código do `Composite`:

```
public static void main(String[] args) {
    ArquivoComponent minhaPasta = new ArquivoComposite("Minha Pasta/");
    ArquivoComponent meuVideo = new ArquivoVideo("video.avi");
    ArquivoComponent meuOutroVideo = new ArquivoVideo("serieS01E01.mkv");
    try {
        meuVideo.adicionar(meuOutroVideo);
    } catch (Exception e) {
        System.out.println(e.getMessage());
    }
    try {
        minhaPasta.adicionar(meuVideo);
        minhaPasta.adicionar(meuOutroVideo);
        minhaPasta.printNomeDoArquivo();
    } catch (Exception e) {
        System.out.println(e.getMessage());
    }
    try {
        System.out.println("Pesquisando arquivos:");
        minhaPasta.getArquivo(meuVideo.getNomeDoArquivo())
```

```
                .printNomeDoArquivo();
        System.out.println("Remover arquivos");
        minhaPasta.remover(meuVideo.getNomeDoArquivo());
        minhaPasta.printNomeDoArquivo();
    } catch (Exception e) {
        e.printStackTrace();
    }
}
```

Agora podemos visualizar a tal estrutura de árvore, suponha que temos pastas dentro de pastas com arquivos, a estrutura seria parecida com a de uma árvore, veja a seguir:

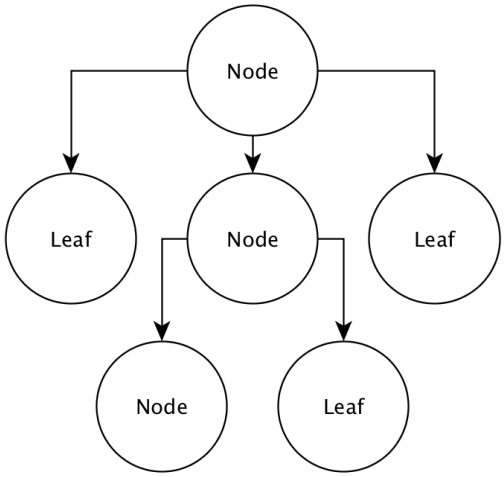

Figura 14 – O padrão de projetos Composite

Como uma estrutura de árvore temos Nós (*Node*) e Folhas (*Leaf*). No padrão *Composite* os arquivos concretos do nosso exemplo são chamados de Folhas, pois não possuem filhos e os arquivos pasta são chamados de Nós, pois possuem filhos e fornecem operações sobre esses filhos.

Decorator

Imagine que você está desenvolvendo um sistema para um bar especializado em coquetéis, onde existem vários tipos que devem ser cadastrados para controlar a venda. Os coquetéis são feitos pela combinação de uma bebida base e vários outros adicionais que compõe a bebida. Por exemplo:

- Conjunto de bebidas
    - Cachaça
    - Rum
    - Vodka
    - Tequila

- Conjunto de adicionais
    - Limão
    - Refrigerante
    - Suco
    - Leite condensado

- Gelo
- Açúcar

Então, como possíveis coquetéis temos, por exemplo:

- Vodka + Suco + Gelo + Açúcar
- Tequila + Limão + Sal
- Cachaça + Leite Condensado + Açúcar + Gelo

E então, como representar isto em um sistema computacional? Bom, poderíamos utilizar como uma solução simples uma classe abstrata `Coquetel` genérica e, para cada tipo de coquetel construir uma classe concreta. Então teríamos a classe base `Coquetel`:

```
public abstract class Coquetel {
    String nome;
    double preco;
    public String getNome() {
        return nome;
    }
    public double getPreco() {
        return preco;
    }
}
```

A nossa classe define apenas o nome e o preço da bebida para facilitar a exemplificação. Uma classe `Coquetel` concreta seria, por exemplo, a `Caipirinha`:

```
public class Caipirinha extends Coquetel {
    public Caipirinha() {
        nome = "Caipirinha";
        preco = 3.5;
    }
}
```

No entanto, como a especialidade do bar são coquetéis, o cliente pode escolher montar seu próprio coquetel com os adicionais que ele quiser. De acordo com nosso modelo teríamos então que criar várias classes para prever o que um possível cliente solicitaria! Imagine agora a quantidade de combinações possíveis? Veja o diagrama UML abaixo para visualizar o tamanho do problema:

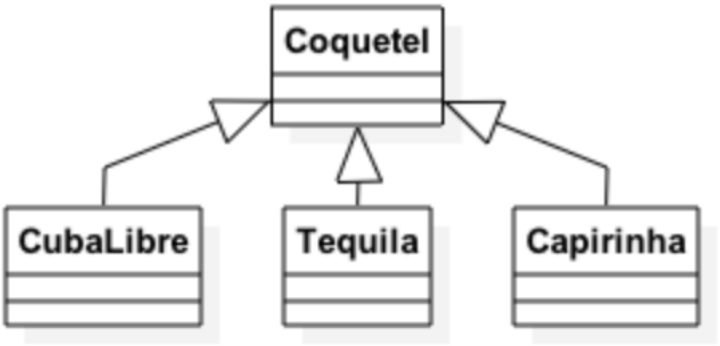

Figura 15 – O padrão de projetos Decorator

Além disso, pode ser que o cliente deseje adicionar doses extras de determinados adicionais, desse modo não seria possível modelar o sistema para prever todas as possibilidades! Então, como resolver o problema?

#DEFINIÇÃO#

Dinamicamente, agregar responsabilidades adicionais a objetos. Os Decorators fornecem uma alternativa flexível ao uso de subclasses para extensão de funcionalidades (GAMMA et al., 2000).
#DEFINIÇÃO#

Perfeito, exatamente a solução do nosso problema! Queremos que, dado um objeto `Coquetel`, seja possível adicionar funcionalidades a ele, e somente a ele, em tempo de execução. Vamos ver a arquitetura sugerida pelo padrão:

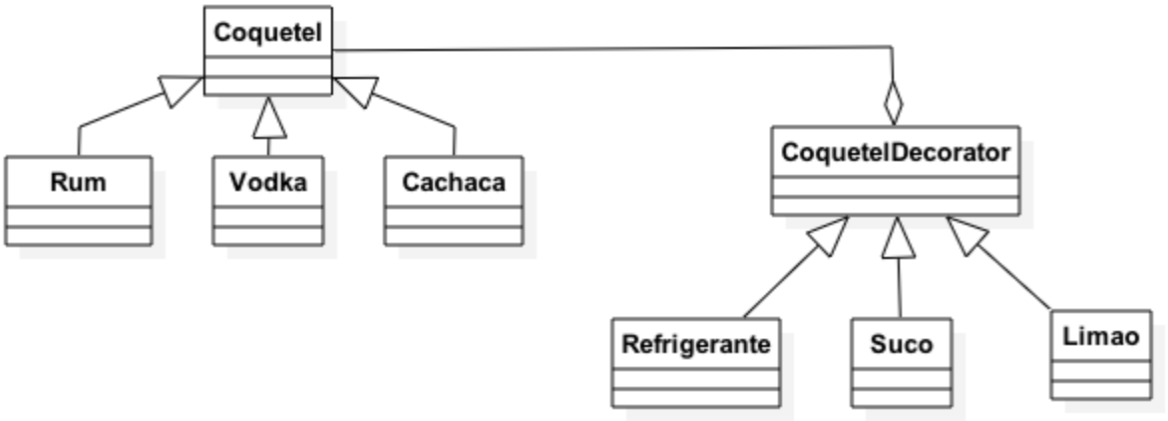

Figura 16 – O padrão de projetos Decorator

Certo, então todos os objetos possuem o mesmo tipo `Coquetel`, esta classe define o que todos os objetos possuem e é igual a classe já feita antes. As classes de bebidas concretas definem apenas os dados relativos a ela. Como exemplo vejamos o código da bebida `Cachaça`:

```
public class Cachaca extends Coquetel {
    public Cachaca() {
```

```
        nome = "Cachaça";
        preco = 1.5;
    }
}
```

Todas as classes de bebidas possuirão a mesma estrutura, apenas definem os seus atributos. A classe `Decorator` abstrata define que todos os decoradores possuem um objeto `Coquetel`, ao qual decoram, e um método que é aplicado a este objeto. Vejamos o código para exemplificar:

```
public abstract class CoquetelDecorator extends Coquetel {
    Coquetel coquetel;
    public CoquetelDecorator(Coquetel umCoquetel) {
        coquetel = umCoquetel;
    }
    @Override
    public String getNome() {
        return coquetel.getNome()  + " + " + nome;
    }
    public double getPreco() {
        return coquetel.getPreco() + preco;
    }
}
```

Lembre-se de que como o decorador também é um `Coquetel` ele herda os atributos nome e preço. Nas classes concretas apenas definimos os modificadores que serão aplicados, de maneira semelhante as classes de bebidas concretas, vejamos o exemplo do adicional `Refrigerante`:

```
public class Refrigerante extends CoquetelDecorator {
    public Refrigerante(Coquetel umCoquetel) {
        super(umCoquetel);
        nome = "Refrigerante";
        preco = 1.0;
    }
}
```

Perceba que no construtor do decorador é necessário passar um objeto `Coquetel` qualquer, este objeto pode ser tanto uma bebida quanto outro decorador. Aí está o conceito chave para o padrão *Decorator*. Vamos acrescentando vários decoradores em qualquer ordem em uma bebida. Vamos ver agora como o padrão seria utilizado, veja o seguinte código do método main:

```
public static void main(String[] args) {
        Coquetel meuCoquetel = new Cachaca();
        System.out.println(meuCoquetel.getNome()+meuCoquetel.getPreco());
        meuCoquetel = new Refrigerante(meuCoquetel);
        System.out.println(meuCoquetel.getNome()+meuCoquetel.getPreco());
}
```

Perceba que o tipo do coquetel varia de acordo com o decorador aplicado. Então, quando chamamos o método `getNome` ou `getPreco` o primeiro método chamado é o método do último decorador aplicado. O método do decorador por sua vez chama o método da classe mãe, este método então chama o método do `Coquetel` ao qual ele decora. Se esse coquetel for outro decorador o pedido é

repassado até chegar a um coquetel que é uma bebida de fato e finalmente responde a requisição sem repassar a nenhum outro objeto. De maneira semelhante a recursão, os valores calculados vão sendo retornados até chegar no último decorador aplicado e então são repassados ao objeto. É como se os decoradores englobassem tanto outros decoradores quanto o componente em si.

Façade (ou Facade)

No desenvolvimento de jogos é comum a utilização de subsistemas de um *Framework*/API. Por exemplo, para reproduzir um determinado som é utilizado o subsistema de Audio, que normalmente provê funcionalidades desde a configuração da reprodução de audio, até a reprodução de um determinado arquivo. Antes de iniciar o jogo de fato é necessário realizar ajustes em todos os subsistemas que serão utilizados, por exemplo, é necessário configurar a resolução do subsistema de Video para que este possa renderizar imagens corretamente. Para exemplificar veja as interfaces das seguintes classes, que representa o subsistema de Audio:

```
public class SistemaDeAudio {
    public void configurarFrequencia() {
        System.out.println("Frequência configurada");
    }
    public void configurarVolume() {
        System.out.println("Volume configurado");
    }
    public void configurarCanais() {
        System.out.println("Canais configurados");
    }
    public void reproduzirAudio(String arquivo) {
        System.out.println("Reproduzindo: " + arquivo);
    }
}
```

Como falado ela fornece os métodos para configuração e reprodução de arquivos de audio. Para reproduzir um arquivo de audio, por exemplo, seria necessário realizar as seguintes operações:

```
public static void main(String[] args) {
    System.out.println("Configurando subsistemas");
    SistemaDeAudio audio = new SistemaDeAudio();
    audio.configurarCanais();
    audio.configurarFrequencia();
    audio.configurarVolume();
    System.out.println("Utilizando subsistemas");
    audio.reproduzirAudio("teste.mp3");
}
```

Neste exemplo de código cliente, o próprio cliente deve instanciar e configurar o subsistema para que só depois seja possível a utilização dos mesmos. Além disso, existe um comportamento padrão que é executado antes de reproduzir o som: sempre deve ser configurado o canal, a frequência e o volume. Agora pense como seria caso fosse necessário utilizar vários subsistemas? O código cliente ficaria muito sobrecarregado com responsabilidades que não são dele:

```
public static void main(String[] args) {
    System.out.println("Configurando subsistemas");
    SistemaDeAudio audio = new SistemaDeAudio();
    audio.configurarCanais();
    audio.configurarFrequencia();
    audio.configurarVolume();
    SistemaDeInput input = new SistemaDeInput();
    input.configurarTeclado();
    input.configurarJoystick();
    SistemaDeVideo video = new SistemaDeVideo();
    video.configurarCores();
    video.configurarResolucao();
    System.out.println("Utilizando subsistemas");
    audio.reproduzirAudio("teste.mp3");
    input.lerInput();
    video.renderizarImagem("imagem.png");
}
```

Vamos ver então como o padrão Facade pode resolver este pequeno problema.

#DEFINIÇÃO#

Fornecer uma interface unificada para um conjunto de interfaces em um subsistema. *Facade* define uma interface de nível mais alto que torna o subsistema mais fácil de ser usado (GAMMA et al., 2000).
#DEFINIÇÃO#

Pela intenção é possível notar que o padrão pode ajudar bastante na resolução do nosso problema. O conjunto de interfaces seria exatamente o conjunto de subsistemas. Um subsistema é análogo a uma classe, uma classe encapsula estados e operações, enquanto um subsistema encapsula classes (GAMMA et al., 2000). Nesse sentido o *Facade* vai definir operações a serem realizadas com estes subsistemas. Assim, é possível definir uma operação padrão para configurar o subsistema de audio, evitando a necessidade de chamar os métodos de configuração de audio a cada novo arquivo de audio que precise ser reproduzido. A utilização do padrão *Facade* é bem simples. apenas é necessário criar a classe fachada que irá se comunicar com os subsistemas no lugar no cliente:

```
public class SistemasFacade {
    protected SistemaDeAudio audio;
    protected SistemaDeInput input;
    protected SistemaDeVideo video;
    public void inicializarSubsistemas() {
        video = new SistemaDeVideo();
        video.configurarCores();
        video.configurarResolucao();

        input = new SistemaDeInput();
        input.configurarJoystick();
        input.configurarTeclado();

        audio = new SistemaDeAudio();
        audio.configurarCanais();
        audio.configurarFrequencia();
        audio.configurarVolume();
    }
```

```
    public void reproduzirAudio(String arquivo) {
        audio.reproduzirAudio(arquivo);
    }
    public void renderizarImagem(String imagem) {
        video.renderizarImagem(imagem);
    }
    public void lerInput() {
        input.lerInput();
    }
}
```

A classe fachada realiza a inicialização de todos os subsistemas e oferece acesso aos métodos necessários, por exemplo o método de renderização de uma imagem, a reprodução de um audio. Com esta mudança, tiramos toda a responsabilidade do cliente, que agora precisa se preocupar apenas em utilizar os subsistemas que desejar.

```
public static void main(String[] args) {
    System.out.println("Configurando subsistemas");
    SistemasFacade fachada = new SistemasFacade();
    fachada.inicializarSubsistemas();

    System.out.println("Utilizando subsistemas");
    fachada.renderizarImagem("imagem.png");
    fachada.reproduzirAudio("teste.mp3");
    fachada.lerInput();
}
```

A utilização do padrão é bem simples e poder ser aplicado em várias situações.

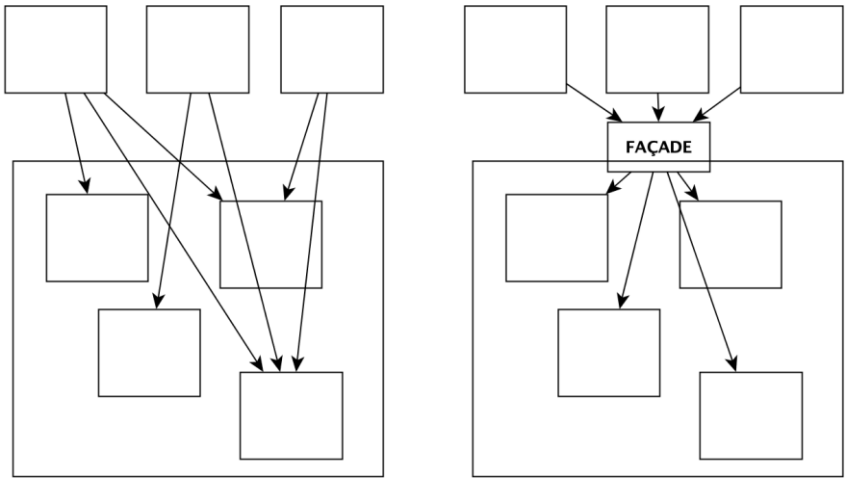

Figura 17 – O padrão de projetos Façade

Flyweight

No desenvolvimento de jogos são utilizadas várias imagens. Elas representam as entidades que compõe o jogo, por exemplo, cenários, jogadores, inimigos, entre outros. Ao criar classes que representam estas entidades, é

necessário vincular a elas um conjunto de imagens, que representam as animações. Quem desenvolve jogos pode ter pensado na duplicação de informação quando as imagens são criadas pelos objetos que representam estas entidades, por exemplo, a classe que representa um inimigo carrega suas imagens. Quando são exibidos vários inimigos do mesmo tipo na tela, o mesmo conjunto de imagens é criado repetidamente. A solução para esta situação de duplicação de informações pelos objetos é a utilização do padrão *Flyweight* (LARMAN, 2007).

#DEFINIÇÃO#

Usar compartilhamento para suportar eficientemente grandes quantidades de objetos de granularidade fina (GAMMA et al., 2000).

#DEFINIÇÃO#

Pela intenção percebemos que o padrão *Flyweight* cria uma estrutura de compartilhamento de objetos pequenos. Para o exemplo citado, o padrão será utilizado no compartilhamento de imagens entre as entidades. Antes de exemplificar vamos entende um pouco sobre a estrutura do padrão. A classe *Flyweight* fornece uma interface com uma operação que deve ser realizado sobre um estado interno. No exemplo esta classe irá fornecer uma operação para desenhar a imagem em um determinado ponto. Desta forma a imagem é o estado intrínseco, que consiste de uma informação que não depende de um contexto externo. O ponto passado como parâmetro é o estado extrínseco, que varia de acordo com o contexto. Vamos então ao código da imagem e do ponto:

```
public class Imagem {
    protected String nomeDaImagem;
    public Imagem(String imagem) {
        nomeDaImagem = imagem;
    }
    public void desenharImagem() {
        System.out.println(nomeDaImagem + " desenhada!");
    }
}
```

Para simplificar o exemplo, será apenas exibida uma mensagem no terminal, indicando que a imagem foi desenhada.

```
public class Ponto {
    public int x, y;
    public Ponto(int x, int y) {
        this.x = x;
        this.y = y;
    }
}
```

A classe *Flyweight* vai apenas fornecer a interface para desenho da imagem em um ponto.

```
public abstract class SpriteFlyweight {
    public abstract void desenharImagem(Ponto ponto);
}
```

Outro componente da estrutura do *Flyweight* é a classe *Flyweight* concreta, que implementa a operação de fato:

```
public class Sprite extends SpriteFlyweight {
    protected Imagem imagem;
    public Sprite(String nomeDaImagem) {
        imagem = new Imagem(nomeDaImagem);
    }
    @Override
    public void desenharImagem(Ponto ponto) {
        imagem.desenharImagem();
        System.out.println("No ponto (" + ponto.x + "," + ponto.y + ")!");
    }
}
```

Nesta classe também será apenas exibida uma mensagem no terminal para dizer que a imagem foi desenhada no ponto dado. O próximo componente da estrutura do *Flyweight* consiste em uma classe fábrica, que vai criar os vários objetos `flyweight` que serão compartilhados.

```
public class FlyweightFactory {
    protected ArrayList<SpriteFlyweight> flyweights;
    public enum Sprites {
        JOGADOR, INIMIGO_1, INIMIGO_2, INIMIGO_3, CENARIO_1, CENARIO_2
    }
    public FlyweightFactory() {
        flyweights = new ArrayList<SpriteFlyweight>();
        flyweights.add(new Sprite("jogador.png"));
        flyweights.add(new Sprite("inimigo1.png"));
        flyweights.add(new Sprite("inimigo2.png"));
        flyweights.add(new Sprite("inimigo3.png"));
        flyweights.add(new Sprite("cenario1.png"));
        flyweights.add(new Sprite("cenario2.png"));
    }
```

```
    public SpriteFlyweight getFlyweight(Sprites jogador) {
        switch (jogador) {
        case JOGADOR:
            return flyweights.get(0);
        case INIMIGO_1:
            return flyweights.get(1);
        case INIMIGO_2:
            return flyweights.get(2);
        case INIMIGO_3:
            return flyweights.get(3);
        case CENARIO_1:
            return flyweights.get(4);
        default:
            return flyweights.get(5);
        }
    }
}
```

Além de criar os vários objetos a serem compartilhados, a classe fábrica oferece um método para obter o objeto, assim, o acesso a estes objetos fica centralizado e unificado a partir desta classe. Para exemplificar a utilização do padrão, vejamos o seguinte código cliente:

```
public static void main(String[] args) {
    FlyweightFactory factory = new FlyweightFactory();
    factory.getFlyweight(Sprites.CENARIO_1).desenharImagem(new Ponto(0, 0));
    factory.getFlyweight(Sprites.JOGADOR).desenharImagem(new Ponto(10, 10));
    factory.getFlyweight(Sprites.INIMIGO_1).desenharImagem(new Ponto(100, 10));
    factory.getFlyweight(Sprites.INIMIGO_1).desenharImagem(new Ponto(120, 10));
    factory.getFlyweight(Sprites.INIMIGO_1).desenharImagem(new Ponto(140, 10));
    factory.getFlyweight(Sprites.INIMIGO_2).desenharImagem(new Ponto(60, 10));
    factory.getFlyweight(Sprites.INIMIGO_2).desenharImagem(new Ponto(50, 10));
    factory.getFlyweight(Sprites.INIMIGO_3).desenharImagem(new Ponto(170, 10));
}
```

É exibido um conjunto de imagens para exemplificar o uso em um jogo. São desenhados inimigos de vários tipos, o cenário do jogo e o jogador. Note que o acesso aos objetos fica centralizado apenas na classe fábrica. No desenvolvimento de jogos real, as referências dos objetos seriam espalhadas pelas entidades, garantindo a não duplicação de conteúdo. O diagrama UML que representa esta implementação é o seguinte:

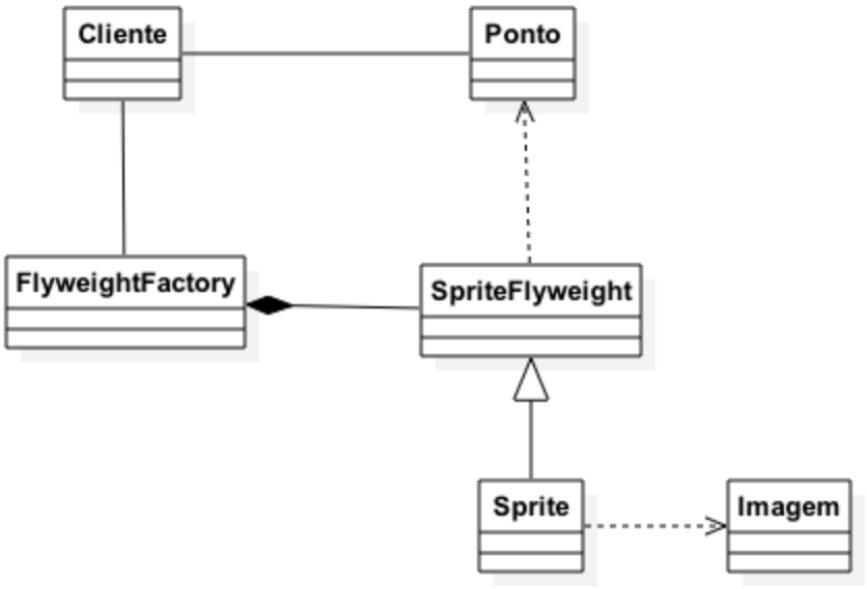

Figura 18 – O padrão de projetos Flyweight

Proxy

Suponha que um determinado programa faz uma conexão com o banco de dados para pegar algumas informações relativas aos usuários do sistema. Para simplificar o exemplo, considere a seguinte classe:

```
public class BancoUsuarios{
    private int quantidadeDeUsuarios;
    private int usuariosConectados;
    public BancoUsuarios() {
        quantidadeDeUsuarios = (int) (Math.random() * 100);
        usuariosConectados = (int) (Math.random() * 10);
    }
    public String getNumeroDeUsuarios() {
        return new String("Total de usuários: " + quantidadeDeUsuarios);
    }
    public String getUsuariosConectados() {
        return new String("Usuários conectados: " + usuariosConectados);
    }
}
```

Nesta classe, nós consultamos o número de usuários que estão conectados e o total de usuários do sistema. Poderiam existir diversas outras operações de consulta ao banco de dados, apenas simplificamos o exemplo. O que queremos é implementar uma nova funcionalidade que permite verificar se um usuário possui ou não permissão para visualizar as informações do banco. A classe nos fornece uma

maneira de acessar o banco de dados do sistema, no entanto ela não possui nenhuma proteção sobre quem está tentando acessá-la. Como podemos implementar um mecanismo de proteção?

Uma primeira solução seria adicionar os campos de usuário e senha e verificar se quem está tentando acessá-la possui as devidas permissões. O problema com essa abordagem é que, como precisaríamos alterar a própria classe, todo o resto do programa que usa essa classe teria que ser alterado também. Uma solução melhor seria utilizar outra classe para verificar se o usuário possui permissão de acesso e só então exibir as informações do banco. Vamos mostrar esta solução utilizando o padrão *Proxy*.

#DEFINIÇÃO#

Fornecer um substituto ou marcador da localização de outro objeto para controlar o acesso a esse objeto (GAMMA et al., 2000).

#DEFINIÇÃO#

Extamente a solução falada antes. Vamos utilizar uma classe substituta à classe `BancoUsuarios` para controlar o acesso. Vamos então criar a classe *Proxy*. Para garantir que ela possa realmente substituir a classe original precisamos fazer com que a classe `proxy` estenda o comportamento da classe original:

```
public class BancoProxy extends BancoUsuarios {
    protected String usuario, senha;
    public BancoProxy(String usuario, String senha) {
        this.usuario = usuario;
        this.senha = senha;
    }
}
```

Pronto, como a classe `BancoProxy` entende o comportamento de `BancoUsuarios` nós podemos utilizar um `BancoProxy` em qualquer lugar onde um `BancoUsuarios` era esperado. Além disso, adicionamos aqui os campos de usuário e senha, que serão verificados nas operações. Vamos então sobrescrever os métodos que buscam informação no banco. Agora, como temos o usuário e a senha, podemos realizar a verificação. Caso o usuário tenha permissão de acesso nós realizamos a chamado ao método da classe `BancoUsuarios`, caso contrário, retornamos nulo:

```
public class BancoProxy extends BancoUsuarios {
    protected String usuario, senha;
    public BancoProxy(String usuario, String senha) {
        this.usuario = usuario;
        this.senha = senha;
```

```
    }
    @Override
    public String getNumeroDeUsuarios() {
        if (temPermissaoDeAcesso()) {
            return super.getNumeroDeUsuarios();
        }
        return null;
    }
    @Override
    public String getUsuariosConectados() {
        if (temPermissaoDeAcesso()) {
            return super.getUsuariosConectados();
        }
        return null;
    }
    private boolean temPermissaoDeAcesso() {
        return usuario == "admin" && senha == "admin";
    }
}
```

Pronto, com esta classe nós implementamos a segurança necessária sem precisar alterar o programa. Quando for necessário implementar um acesso seguro, utilizamos a classe `Proxy`, quando não, podemos utilizar a classe original sem nenhum problema. Vejamos por exemplo o seguinte código cliente:

```
public static void main(String[] args) {
      System.out.println("Hacker acessando");
      BancoUsuarios banco = new BancoProxy("Hacker", "1234");
      System.out.println(banco.getNumeroDeUsuarios());
      System.out.println(banco.getUsuariosConectados());
      System.out.println("Administrador acessando");
      banco = new BancoProxy("admin", "admin");
      System.out.println(banco.getNumeroDeUsuarios());
      System.out.println(banco.getUsuariosConectados());
}
```

Veja que utilizamos uma referência a um objeto do tipo `BancoUsuarios`, mas instanciamos um objeto do tipo `BancoProxy`. Isto mostra o que falamos antes sobre utilizar um proxy em qualquer lugar que um objeto original seria esperado. Caso houvesse um método que tivesse como entrada um objeto `BancoUsuarios` poderíamos sem nenhum problema utilizar um objeto `BancoProxy` para manter a segurança. O diagrama UML para este caso de uso do *Proxy* seria bem simples:

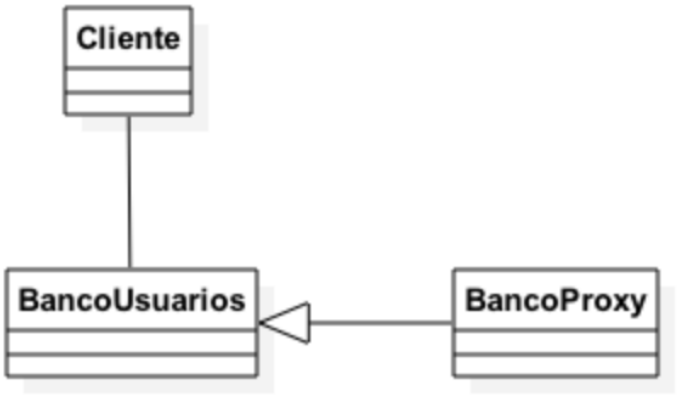

Figura 19 – O padrão de projetos Proxy

Como identificar a necessidade desses padrões

Sempre que um objeto se relacionar com outro, seja por associação, composição, herança, composição, realização ou dependência, é interessante observar a possibilidade de utilização de algum padrão estrutural. Esses padrões reduzem o acoplamento entre as classes e a propagação de feitos colaterais. Dessa forma, os testes e manutenções do software pode ser feito de maneira mais acertada em com poucas preocupações de publicação.

Considerações Finais

Neste estudo entendemos os padrões estruturais e como os mesmos podem ser aplicados aos projetos de software orientados a objetos. Esses padrões são fundamentais para a estabilidade do software e facilitam o teste e manutenção ao longo do ciclo de vida do produto.

UNIDADE 4 – PADRÕES COMPORTAMENTAIS

Objetivos de aprendizagem

- Os padrões de projetos comportamentais
- Os *padrões Chain of Responsibility, Command, Interpreter, Iterator, Mediator, Memento, Observer, State, Strategy, Template Method* e *Visitor*
- Como identificar a necessidade desses padrões

Os padrões de projetos comportamentais

Os padrões comportamentais atuam sobre como responsabilidades são atribuídas as entidades, ou seja, qual o comportamento das entidades. Estes padrões facilitam a comunicação entre os objetos, distribuindo as responsabilidades e definindo a comunicação interna. Padrões com escopo de classe utilizam herança para realizar a distribuição do comportamento. Um bom exemplo é o padrão Template Method, que fornece um algoritmo (comportamento) padrão e deixam as subclasses definirem alguns pontos da execução do algoritmo. Já os padrões de objetos vão compor os objetos para definir a comunicação, como o padrão Mediator, que define um objeto que realiza a comunicação muitos-para-muitos (LARMAN, 2007).

Os padrões Chain of Responsibility, Command, Interpreter, Iterator, Mediator, Memento, Observer, State, Strategy, Template Method e Visitor

Chain of Responsibility

Uma aplicação de e-commerce precisa conectar-se com múltiplos bancos de dados para prover aos seus usuários possibilidades de pagamento. Ao modelar uma forma de execução do pagamento, dado que precisamos selecionar um entre vários tipos de bancos, a primeira ideia que surge é utilizar uma estrutura de decisão para verificar, dado um parâmetro, qual o banco correto deve ser utilizado. Poderíamos utilizar os métodos fábricas para gerar o objeto correto para ser utilizado na nossa aplicação. Poderíamos criar estratégias diferentes para cada banco e escolher em tempo de execução. Em todas estas soluções, nós utilizamos uma forma de encapsular a estrutura de decisão por trás de uma interface, ou algo similar, para que as alterações fossem menos dolorosas. No entanto, continuamos utilizando as estruturas de decisão. Vamos analisar então o padrão *Chain of Responsibility*, que promete acabar com estas estruturas (FREEMAN e FREEMAN, 2007).

#DEFINIÇÃO#

Evitar o acoplamento do remetente de uma solicitação ao seu receptor, ao dar a mais de um objeto a oportunidade de tratar a solicitação. Encadear os objetos receptores, passando a solicitação ao longo da cadeia até que um objeto a trate (GAMMA et al., 2000).

#DEFINIÇÃO#

Dessa forma, percebemos como o *Chain of Responsibility* acaba com as estruturas de decisão. Ele cria uma cadeia de objetos e vai passando a responsabilidade entre eles até que alguém possa responder pela chamada. Vamos então iniciar construindo uma pequena enumeração para identificar os bancos utilizados no nosso sistema:

```
public enum IDBancos {
    bancoA, bancoB
}
```

Agora vamos construir a classe que vai implementar a cadeia de responsabilidades. Vamos exibir partes do código para facilitar o entendimento.

```
public abstract class BancoChain {
    protected BancoChain next;
    protected IDBancos identificadorDoBanco;
    public BancoChain(IDBancos id) {
        next = null;
        identificadorDoBanco = id;
    }
    public void setNext(BancoChain forma) {
        if (next == null) {
            next = forma;
        } else {
            next.setNext(forma);
        }
    }
}
```

A nossa classe possui apenas dois atributos, o identificador do banco e uma referência para o próximo objeto da corrente. No construtor inicializamos estes atributos. O método `setNext` recebe uma nova instância da classe e faz o seguinte, se o próximo for nulo, então o próximo na corrente será o parâmetro. Caso contrário, repassa esta responsabilidade para o próximo elemento. Assim, a instância que deve ser adicionada na corrente irá percorrer os elementos até chegar no último elemento. O próximo passo será criar o método para efetuar o pagamento.

```
public void efetuarPagamento(IDBancos id) throws Exception {
    if (podeEfetuarPagamento(id)) {
        efetuaPagamento();
    } else {
        if (next == null) {
            throw new Exception();
        }
        next.efetuarPagamento(id);
    }
}
```

A primeira parte do algoritmo de pagamento é verificar se o banco atual pode fazer o pagamento. Para isto é utilizado o identificador do banco, que é comparado com o identificador passado por parâmetro. Se o elemento atual puder responder a requisição é chamado o método que vai efetuar o pagamento de fato. Este método é abstrato, e as subclasses devem implementá-lo, com seu próprio mecanismo. Se o elemento atual não puder responder, ele repassa a chamado ao próximo elemento da lista. Antes disto é feita uma verificação, por questões de segurança, se este próximo elemento realmente existe. Caso nenhum elemento possa responder, é disparada uma exceção. Agora que definimos a estrutura da cadeia de responsabilidades, vamos implementar um banco concreto, que responde a uma chamada.

```
public class BancoA extends BancoChain {
    public BancoA() {
        super(IDBancos.bancoA);
    }
    @Override
    protected void efetuaPagamento() {
        System.out.println("Pagamento efetuado no banco A");
    }
}
```

O `Banco A` inicializa seu `ID` e, no método de efetuar o pagamento, exibe no terminal que o pagamento foi efetuado. A implementação dos outros bancos segue este exemplo. O `ID` é iniciado e o método de efetuar o pagamento exibe a saída no terminal. O cliente deste código seria algo do tipo:

```
public static void main(String[] args) {
    BancoChain bancos = new BancoA();
    bancos.setNext(new BancoB());
    try {
        bancos.efetuarPagamento(IDBancos.bancoA);
        bancos.efetuarPagamento(IDBancos.bancoB);
    } catch (Exception e) {
```

```
            e.printStackTrace();
        }
}
```

O diagrama UML deste exemplo seria:

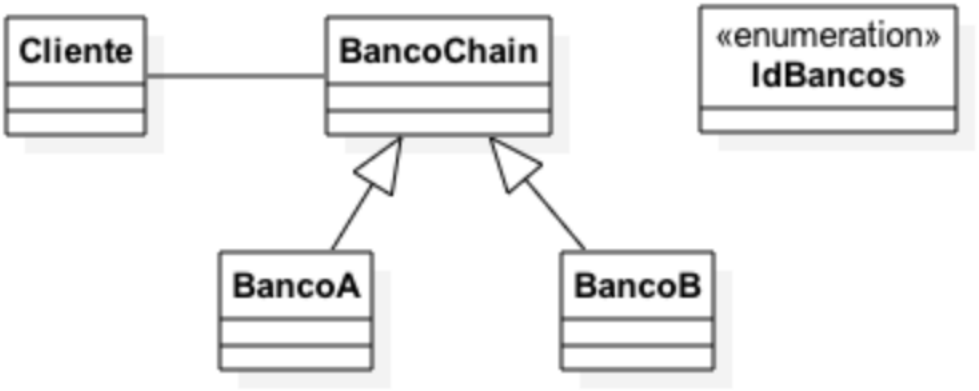

Figura 20 – O padrão de projetos Chain of Responsability

Command

Suponha uma loja que vende produtos e oferece várias formas de pagamento. Ao executar uma compra o sistema registra o valor total e, dada uma forma de pagamento, por exemplo, cartão de crédito, emite o valor total da compra para o cartão de crédito do cliente. Para este caso então vamos supor as seguintes classes para simplificar o exemplo: `Loja` e `Compra`. A classe `Loja` representa a loja que está efetuando a venda. Vamos deixar a classe `Loja` bem simples, como mostra o seguinte código:

```
public class Loja {
    protected String nomeLoja;
    public Loja(String nome) {
        nomeLoja = nome;
    }
    public void executarCompra(double valor) {
        Compra compra = new Compra(nomeLoja);
        compra.setValor(valor);
    }
}
```

Para focar apenas no que é necessário para entender o padrão, vamos utilizar a classe `Compra`, que representa o conjunto de produtos que foram vendidos com o seu valor total:

```
public class Compra {
    private static int CONTADOR_ID;
    protected int idNotaFiscal;
    protected String nomeLoja;
    protected double valorTotal;
    public Compra(String nomeLoja) {
        this.nomeLoja = nomeLoja;
        idNotaFiscal = ++CONTADOR_ID;
    }
    public void setValor(double valor) {
        this.valorTotal = valor;
    }
    public String getInfoNota() {
        return new String("NF: " + idNotaFiscal + "Valor: " + valorTotal);
    }
}
```

Pronto, agora precisamos alterar a classe Loja para que ela, ao executar uma compra saiba qual a forma de pagamento. Uma maneira interessante de fazer esta implementação seria adicionando um parâmetro a mais no método executar que nos diga qual forma de pagamento deve ser usada. Poderíamos então utilizar um Enum para identificar a forma de pagamento e daí passar a responsabilidade ao objeto específico. Veja o exemplo que mostra o código do método executar compra utilizando uma enumeração:

```
public void executarCompra(double valor, FormaPagamento formaPagamento) {
    Compra compra = new Compra(nomeLoja);
    compra.setValor(valor);
    if(formaPagamento == FormaPagamento.CartaoCredito){
        new PagamentoCartaoCredito().processarCompra(compra);
    } else if(formaPagamento == FormaPagamento.CartaoDebito){
        new PagamentoCartaoDebito().processarCompra(compra);
    } else if(formaPagamento == FormaPagamento.Boleto){
        new PagamentoBoleto().processarCompra(compra);
    }
}
```

O problema desta solução é que, caso seja necessário incluir ou remover uma forma de pagamento precisaremos fazer várias alterações, alterando tanto a enumeração quando o método que processa a compra. Veja também a quantidade de ifs aninhados, isso é um sintoma de um *design* mal feito. Poderíamos passar o objeto que faz o pagamento como um dos parâmetros, ao invés de utilizar a enumeração, assim não teríamos mais problemas com os ifs aninhados e as alterações seriam locais. Ok, está é uma boa solução, mas ainda não está boa, pois

precisaríamos de um método diferente pra cada tipo de objeto. A saída obvia então é utilizar uma classe comum a todos as formas de pagamento, e no parâmetro passar um objeto genérico! Essa é a ideia do Padrão *Command* (FREEMAN e FREEMAN, 2007).

#DEFINIÇÃO#

Encapsular uma solicitação como objeto, desta forma permitindo parametrizar cliente com diferentes solicitações, enfileirar ou fazer o registro de solicitações e suportar operações que podem ser desfeitas (GAMMA et al., 2000).

#DEFINIÇÃO#

Pela intenção vemos que o padrão pode ser aplicado em diversas situações. Para resolver o exemplo acima vamos encapsular as solicitações de pagamento de uma compra em objetos para parametrizar os clientes com as diferentes solicitações. Perceba no código com os vários ifs outra boa oportunidade para refatorar o código. Dentro de cada um dos if, a ação é a mesma: criar um objeto para processar o pagamento e realizar uma chamada ao método de processamento da compra. Então vamos primeiro definir a interface comum aos objetos que processam um pagamento. Todos eles possuem um mesmo método, processar pagamento, que toma como parâmetro uma compra e faz o processamento dessa compra de várias formas. A classe interface seria a seguinte:

```
public interface PagamentoCommand {
    void processarCompra(Compra compra);
}
```

Uma possível implementação seria a de processar um pagamento via boleto. Vamos apenas emitir uma mensagem no terminal para saber que tudo foi executado como esperado:

```
public class PagamentoBoleto implements PagamentoCommand {
    @Override
    public void processarCompra(Compra compra) {
        System.out.println("Boleto criado!" + compra.getInfoNota());
    }
}
```

Agora o método de execução da compra na classe Loja seria assim:

```
public void executarCompra(double valor, PagamentoCommand formaPagamento) {
    Compra compra = new Compra(nomeDaLoja);
    compra.setValor(valor);
    formaPagamento.processarCompra(compra);
}
```

O código cliente que usaria o padrão *Command* seria algo do tipo:

```
public static void main(String[] args) {
    Loja a = new Loja("A");
    a.executarCompra(999.00, new PagamentoCartaoCredito());
    a.executarCompra(49.00, new PagamentoBoleto());
    a.executarCompra(99.00, new PagamentoCartaoDebito());
    Loja b = new Loja("B");
    b.executarCompra(19.00, new PagamentoCartaoCredito());
}
```

Ao executar uma compra nós passamos o comando que deve ser utilizado, neste caso, a forma de pagamento utilizado. O diagrama UML seria algo do tipo:

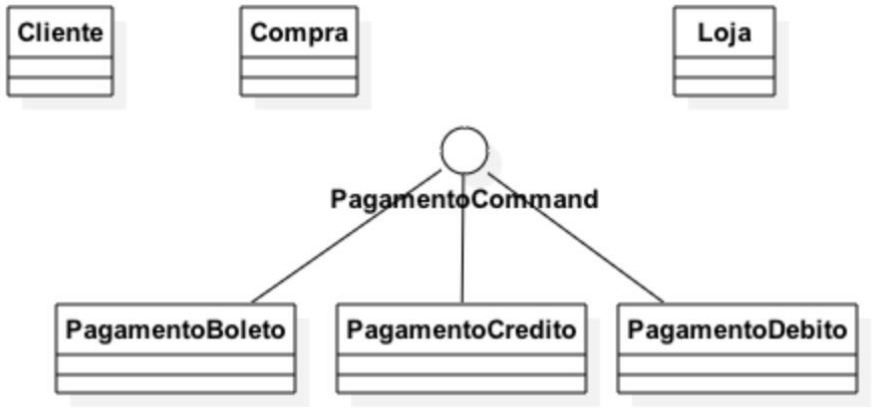

Figura 21 – O padrão de projetos Command

Interpreter

Reconhecer padrões é um problema bem complicado, no entanto, quando conseguimos formular uma gramática para o problema a solução fica bem mais fácil. Suponha que é preciso converter uma *String* representando um número romano em um inteiro que represente seu valor decimal. É fácil perceber que este problema trata-se de reconhecer determinados padrões. Percorrer a String e procurar cada um dos possíveis casos não é a melhor solução, pois dificultaria bastante a manutenção do código. Então vamos tentar formular o problema como uma gramática. Para simplificar, vamos tratar apenas números de quatro dígitos. Iniciando a definição da gramática, um número romano é composto por caracteres que representam números de quatro, três, dois ou um dígito:

```
numero romano ::= {quatro dígitos} {três dígitos} {dois dígitos} {um dígito}
```

Números de quatro, três, dois e um dígito são formados por caracteres que representam nove, cinco, quatro e um. Com estes caracteres é possível representar

qualquer um dos números em romanos:

```
quatro dígitos ::= um
três dígitos ::= nove | cinco {um} {um} {um} | quatro | um
dois dígitos ::= nove | cinco {um} {um} {um} | quatro | um
um dígito ::= nove | cinco {um} {um} {um} | quatro | um
```

O cinco em romano pode vir seguido por até três números um. E finalmente, os caracteres que representam nove, cinco, quatro e um são os seguintes, considerando apenas números de quatro dígitos:

```
nove ::= "CM", "XC", "IX"
cinco ::= "D", "L", "V"
quatro ::= "CD", "XL", "IV"
um ::= "M", "C", "X", "I"
```

Com estas regras podemos criar vários números romanos, por exemplo, CI é igual a 101, pelas regras definidas a construção seria:

```
número romano -> {três digitos} {um dígito} -> {um} {um dígito} -> C {um dígito} -> C {um} -> CI
```

Outro exemplo, CXCIV é 194, pelas regras seria:

```
número romano -> {três digitos} {dois dígitos} {um dígito} -> {um} {dois dígitos} {um dígito} -> C {dois dígitos} {um dígito} -> C {nove} {um dígito} -> C XC {um dígito} -> C XC {quatro} -> CXCIV
```

Uma vez definida a gramática e suas regras, é possível utilizar o padrão Interpreter para montar uma estrutura para interpretar os comandos.

#DEFINIÇÃO#

Dada uma linguagem, definir uma representação para sua gramática juntamente com um interpretador que usa a representação para interpretar sentenças dessa linguagem.

#DEFINIÇÃO#

Ou seja, dada a linguagem, números romanos, construir uma representação para a gramática dela junto com um interpretador para essa gramática. A estrutura do padrão é muito parecida com a do padrão *Composite* , é definido inicialmente uma classe abstrata que será a base de todas as classes interpretadoras. Nela é construída a interface básica e o método de interpretar. Este método recebe como

atributo um contexto, que é uma classe que vai armazenar as informações de entrada e saída. Vamos então definir a classe contexto para o nosso problema:

```
public class Contexto {
    protected String input;
    protected int output;
    public Contexto(String input) {
        this.input = input;
    }
    //get e set omitidos
}
```

Não se preocupe com os *getters* e *setters*, eles serão necessários quando formos definir o método de interpretação. O que merece a atenção nesta classe são os atributos de input, que é a *String* em formato romano, e o *output*, que é o inteiro que vai armazenar o valor. Vamos analisar a classe interpretadora em partes. Primeiro vamos dar uma olhada no método de interpretação:

```
public abstract class NumeroRomanoInterpreter {
    public void interpretar(Contexto c) {
        if (c.getInput().length() == 0) {
            return;
        }
        if (c.getInput().startsWith(nove())) {
            adicionarValorOutput(c, 9);
            consumirDuasCasasDoInput(c);
        } else if (c.getInput().startsWith(quatro())) {
            adicionarValorOutput(c, 4);
            consumirDuasCasasDoInput(c);
        } else if (c.getInput().startsWith(cinco())) {
            adicionarValorOutput(c, 5);
            consumirUmaCasaInput(c);
        }
        while (c.getInput().startsWith(um())) {
            adicionarValorOutput(c, 1);
            consumirUmaCasaInput(c);
        }
    }
    private void consumirUmaCasaInput(Contexto c) {
        c.setInput(c.getInput().substring(1));
    }
    private void consumirDuasCasasDoInput(Contexto c) {
        c.setInput(c.getInput().substring(2));
```

```
    }
}
```

Este método recebe o contexto e a ideia é fazer o seguinte: comparar os primeiros caracteres da *String* com os caracteres que representam nove, quatro, cinco e um. Quando um destes padrões for encontrado é retirado da *String*, para que não seja repetido, e o seu valor é adicionado ao valor de output do contexto. Um detalhe que deve ser observado é que os valores que representam nove ou quatro possuem dois caracteres, assim é necessário retirar dois caracteres da string de *input*. Feito o método de interpretação é necessário definir as strings que vão representar os caracteres nove, cinco, quatro e um. Além disso, existe outro método não definido, o `multiplicador()`. Este método vai retornar qual o valor relativo do número, por exemplo, se for um número romano de quatro dígitos, o método retornará 1000, se for um de três retornará 100, etc.

```
public abstract class NumeroRomanoInterpreter {
    public abstract String um();
    public abstract String quatro();
    public abstract String cinco();
    public abstract String nove();
    public abstract int multiplicador();
}
```

Estes métodos serão definidos nas subclasses. Lembra das definições das regras da gramática? Cada regra daquela vai se tornar uma classe derivada da classe interpretadora. Nela vão ser definidas as strings de um, quatro, cinco e nove. Também será definido o multiplicado relativo de cada uma. Vejamos então como exemplo a definição da classe que representa números de um dígito:

```
public class UmDigitoRomano extends NumeroRomanoInterpreter {
    @Override
    public String um() {
        return "I";
    }
    @Override
    public String quatro() {
        return "IV";
    }
    @Override
    public String cinco() {
        return "V";
    }
    @Override
    public String nove() {
```

```
        return "IX";
    }
    @Override
    public int multiplicador() {
        return 1;
    }
}
```

Nesta classe definimos os números de um dígito, ou uma casa decimal, em caracteres romanos: I, IV, V e IX. O valor do multiplicador será 1. Como outro exemplo, vejamos a classe que representam números de dois dígitos:

```
public class DoisDigitosRomano extends NumeroRomanoInterpreter {
    @Override
    public String um() {
        return "X";
    }
    @Override
    public String quatro() {
        return "XL";
    }
    @Override
    public String cinco() {
        return "L";
    }
    @Override
    public String nove() {
        return "XC";
    }
    @Override
    public int multiplicador() {
        return 10;
    }
}
```

De maneira análoga, nesta classe são definidos os números de duas casas decimais: X, XL, L e XC. O valor do multiplicador será 10. As outras classes seguirão da mesma maneira. Então vejamos como ficaria o cliente do interpreter deste exemplo:

```
public static void main(String[] args) {
    ArrayList interpretadores = new ArrayList();
    interpretadores.add(new QuatroDigitosRomano());
```

```
    interpretadores.add(new TresDigitosRomano());
    interpretadores.add(new DoisDigitosRomano());
    interpretadores.add(new UmDigitoRomano());
    String numeroRomano = "CXCIV";
    Contexto c = new Contexto(numeroRomano);
    for (NumeroRomanoInterpreter numeroRomanoInterpreter : interpretadores)
{
        numeroRomanoInterpreter.interpretar(c);
    }
    System.out.println(numeroRomano + "=" +
Integer.toString(c.getOutput()));
}
```

Primeiro criamos uma lista onde vamos inserir todos os interpretadores, depois, vamos iterar sobre essa lista chamando os métodos de interpretação e passando um mesmo contexto como parâmetro. No final, podemos verificar o resultado pela saída no terminal. Experimente executar outras instâncias de números romanos e observar o fluxo de chamadas para entender melhor como o padrão foi aplicado. O diagrama UML para este caso é o seguinte:

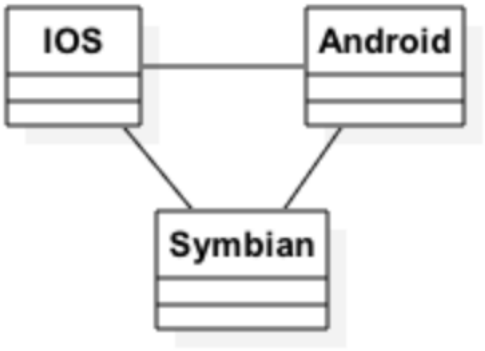

Figura 22 – O padrão de projetos Interpreter

Iterator

Imagine que você trabalha em uma empresa de TV a Cabo. Você recebeu a tarefa de mostrar a lista de canais que a empresa oferece. Ao procurar os desenvolvedores dos canais você descobre que existe uma separação entre os desenvolvedores que cuidam dos canais de esportes e os que cuidam dos canais de filmes. O problema começa quando você percebe que, apesar de ambos utilizarem uma lista de canais, os desenvolvedores dos canais de filmes utilizaram Matriz para representar a lista de canais e os desenvolvedores dos canais de esportes utilizaram `ArrayList`. Você não quer padronizar as listas pois todo o resto do código dos sistemas que cada equipe fez utiliza sua própria implementação (`ArrayList` ou Matriz). Como construir o programa que vai exibir o nome dos canais? A solução mais simples é pegar a lista de canais e fazer dois loops, um para percorrer o ArrayList e outro para percorrer a Matriz, e em cada loop exibir o

nome dos canais (FREEMAN e FREEMAN, 2007). A impressão dos canais seria algo desse tipo:

```
ArrayList<Canal> arrayListDeCanais = new ArrayList<Canal>();
Canal[] matrizDeCanais = new Canal[5];
for (Canal canal : arrayListDeCanais) {
    System.out.println(canal.nome);
}
for (int i = 0; i < matrizDeCanais.length; i++) {
    System.out.println(matrizDeCanais[i].nome);
}
```

No entanto é fácil perceber os problemas desta implementação sem ao menos ver o código, pois, caso outra equipe utilize outra estrutura para armazenar a lista de canais você deverá utilizar outro loop para imprimir. Pior ainda é se você precisar realizar outra operação com as listas de canais terá que implementar o mesmo método para cada uma das listas. Ok, então vamos ver agora uma boa solução para esse problema.

#DEFINIÇÃO#

Fornecer um meio de acessar, sequencialmente, os elementos de um objeto agregado sem expor sua representação subjacente (GAMMA et al., 2000).

#DEFINIÇÃO#

Então utilizando o padrão Iterator nós poderemos acessar os elementos de um conjunto de dados sem conhecer sua implementação, ou seja, sem a necessidade de saber se será utilizado `ArrayList` ou Matriz. No nosso exemplo os objetos agregados seriam as listas de canais (`ArrayList` e Matriz). Inicialmente vamos criar uma interface comum a todos os objetos agregados, ou seja uma lista genérica:

```
public interface AgregadoDeCanais {
    IteradorInterface criarIterator();
}
```

Por ser genérica a classe não possui nenhum detalhe da implementação da lista, ou seja, não possui uma `ArrayList` ou uma Matriz de Canais. A interface define apenas que todas as classes agregadas devem implementar um método de criação de *Iterator*. Na nossa classe concreta nós utilizamos a lista de dados com uma implementação própria e implementamos o método de criação de Iterator, veja a seguir o exemplo da classe de canais que utiliza o ArrayList:

```
public class CanaisEsportes implements AgregadoDeCanais {
```

```
    protected ArrayList<Canal> canais;
    public CanaisEsportes() {
        canais = new ArrayList<Canal>();
        canais.add(new Canal("A"));
        canais.add(new Canal("B"));
    }
    @Override
    public IteradorInterface criarIterator() {
        return new IteradorListaDeCanais(canais);
    }
}
```

O método de criação do *Iterator* retorna um iterador de Lista, que tem como conjunto de dados o `ArrayList` canais. Na classe que utiliza uma matriz para guardar os canais, o método de criação do Iterator retorna um iterador de Matriz.

```
@Override
public IteradorInterface criarIterator() {
    return new IteradorMatrizDeCanais(canais);
}
```

Pronto, conseguimos encapsular os diferentes conjuntos de dados numa interface comum, agora qualquer nova lista que apareça precisa apenas implementar a interface que agrega canais. Vamos ver agora como será a implementação dos iteradores. Da mesma maneira que criamos uma interface comum aos agregados vamos criar uma interface comum aos iteradores, assim podemos garantir que todo iterador tenha o mínimo de operações necessárias para percorrer o conjunto de dados.

```
public interface IteradorInterface {
    void first();
    void next();
    boolean isDone();
    Canal currentItem();
}
```

De maneira bem simples esta interface segue a recomendação do GoF (LARMAN, 2007). Ou seja todo iterador possui um método que inicia o iterador (`first`), avança o iterador (`next`), verifica se já encerrou o percurso (`isDone`) e o que retorna o objeto atual (`currentItem`). A implementação desses métodos será feita no iterador concreto, levando em consideração o tipo do conjunto de dados. Vamos mostrar primeiro a implementação do iterador do ArrayList. Apesar de já existir o Iterador de um ArrayList nativo do Java, vamos criar o nosso próprio Iterator.

```
public class IteradorListaDeCanais implements IteradorInterface {
    protected ArrayList<Canal> lista;
    protected int contador;
    protected IteradorListaDeCanais(ArrayList<Canal> lista) {
        this.lista = lista;
        contador = 0;
    }
    public void first() {
        contador = 0;
    }
    public void next() {
        contador++;
    }
    public boolean isDone() {
        return contador == lista.size();
    }
    public Canal currentItem() {
        if (isDone()) {
            contador = lista.size() - 1;
        } else if (contador < 0) {
            contador = 0;
        }
        return lista.get(contador);
    }
}
```

A implementação do iterador é bem simples também. Os métodos alteram o contador do iterador, que marca qual o elemento está sendo visitado e, no método que retorna o objeto nós verificamos se o contador está dentro dos limites válidos e retornamos o objeto corrente. A implementação do iterador de matriz também é bem simples e segue a mesma estrutura. Veja o código a seguir:

```
public class IteradorMatrizDeCanais implements IteradorInterface {
    protected Canal[] lista;
    protected int contador;
    public IteradorMatrizDeCanais(Canal[] lista) {
        this.lista = lista;
    }
    @Override
    public void first() {
        contador = 0;
    }
```

```
    @Override
    public void next() {
        contador++;
    }
    @Override
    public boolean isDone() {
        return contador == lista.length;
    }
    @Override
    public Canal currentItem() {
        if (isDone()) {
            contador = lista.length - 1;
        } else if (contador < 0) {
            contador = 0;
        }
        return lista[contador];
    }
}
```

Agora nós conseguimos encapsular também uma maneira de percorrer a lista de dados. Se um novo conjunto de dados for inseridos nós poderemos reutilizar iteradores (caso a estrutura do conjunto de dados seja a mesma), ou criar um novo iterador que implemente a interface básica dos iteradores. O código cliente seria bem mais simples, veja:

```
public static void main(String[] args) {
    AgregadoDeCanais ce = new CanaisEsportes();
    System.out.println("Canais de Esporte:");
    for (IteradorInterface it = ce.criarIterator(); !it.isDone();
it.next()){
        System.out.println(it.currentItem().nome);
    }
    AgregadoDeCanais cf = new CanaisFilmes();
    System.out.println("Canais de Filmes:");
    for (IteradorInterface it = cf.criarIterator(); !it.isDone();
it.next()){
        System.out.println(it.currentItem().nome);
    }
}
```

O nosso código utiliza apenas as classes Interfaces, pois assim ficamos independentes de implementações concretas (obedecendo ao princípio de Design Orientado a Objetos) (LARMAN, 2007).

Mediator

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 (FREEMAN e FREEMAN, 2007). Obviamente seria uma péssima solução criar vários métodos para cada plataforma. Analise o diagrama abaixo:

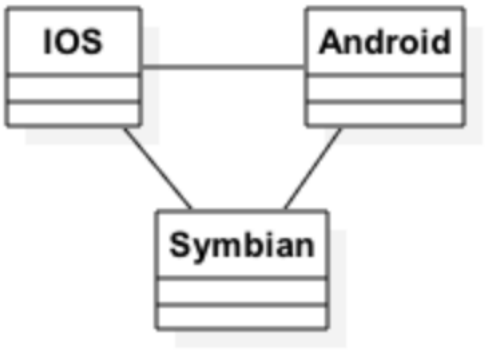

Figura 23 – O padrão de projetos Mediator

Imagine que agora o aplicativo vai incluir a plataforma *BlackBerry OS*, precisaríamos criar os métodos de comunicação com todas as outras plataformas 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 manutenabilidade. 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*.

#DEFINIÇÃO#

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 (GAMMA et al., 2000).

#DEFINIÇÃO#

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:

```
public abstract class Colleague {
    protected Mediator mediator;
    public Colleague(Mediator m) {
        mediator = m;
    }
    public void enviarMensagem(String mensagem) {
        mediator.enviar(mensagem, this);
    }
    public abstract void receberMensagem(String mensagem);
}
```

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:

```
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 destes 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:

```
public static void main(String[] args) {
    MensagemMediator mediador = new MensagemMediator();
    AndroidColleague android = new AndroidColleague(mediador);
    IOSColleague ios = new IOSColleague(mediador);
    SymbianColleague symbian = new SymbianColleague(mediador);
    mediador.adicionarColleague(android);
    mediador.adicionarColleague(ios);
    mediador.adicionarColleague(symbian);
    symbian.enviarMensagem("Oi, eu sou um Symbian!");
    android.enviarMensagem("Oi Symbian! Eu sou um Android!");
    ios.enviarMensagem("Olá todos, sou um iOs!");
}
```

O diagrama UML para este exemplo seria o seguinte:

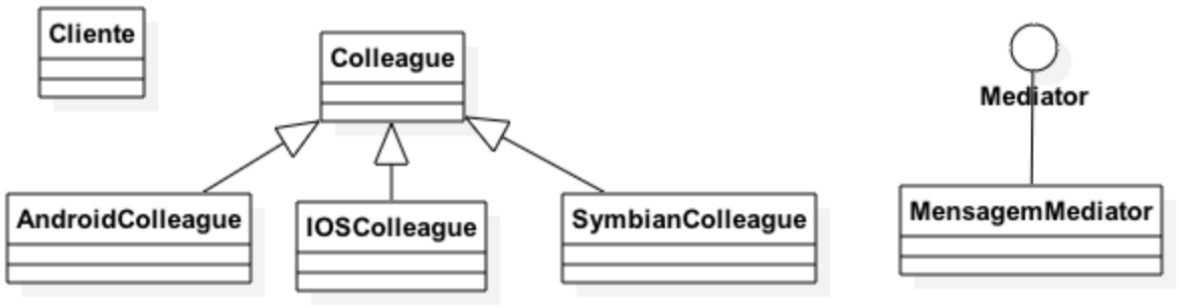

Figura 24 – O padrão de projetos Mediator

Memento

Qualquer bom editor, seja de video, texto, imagens, etc. oferece uma maneira de desfazer ações, recuperando estados anteriores. Como é possível modelar a arquitetura do sistema de maneira que seja possível salvar estados dos elementos? Para exemplificar vamos pensar em editor de texto que precisa manter o controle apenas do texto que é digitado, um notepad por exemplo. Uma primeira saída poderia ser criar um objeto que representasse o texto com suas informações e guardar estes objetos em uma lista. O problema desta implementação está em como recuperar os estados sem ferir o encapsulamento da classe? Pois, ao restaurar o objeto precisaríamos de, no mínimo, getters para acessar as informações do objeto e poder copiá-las para o objeto a ser restaurado. O padrão Memento oferece uma maneira simples de evitar o problema da quebra de encapsulamento e manter o controle das alterações feitas em um objeto (FREEMAN

e FREEMAN, 2007). Vejamos então o padrão:

#DEFINIÇÃO#

Sem violar o encapsulamento, capturar e externalizar um estado interno de um objeto, de maneira que o objeto possa ser restaurado para esse estado mais tarde (GAMMA et al., 2000).

#DEFINIÇÃO#

Pela intenção do padrão podemos facilmente notar sua aplicabilidade. O estado interno do objeto seria, para o exemplo acima, o texto que está sendo digitado pelo usuário. Assim, o padrão *Memento* permitiria capturar o estado do texto par que depois ele possa ser reutilizado. Vamos então ao código do problema. Vamos iniciar com a classe `Memento`. Ela simplesmente mantém a String que representa o texto e oferece um getter para esta `String`, permitindo que ela seja recuperada mais tarde.

```
public class TextoMemento {
    protected String estadoTexto;
    public TextoMemento(String texto) {
        estadoTexto = texto;
    }
    public String getTextoSalvo() {
        return estadoTexto;
    }
}
```

Além do `Memento`, existe outra figura importante, o `Caretaker`. O `Caretaker` vai guardar todos os Memento, permitindo que eles sejam restaurados. Como a aplicação é de um editor de texto os Memento devem ser recuperados de maneira LIFO, *Last in First out*, assim o último memento adicionar será o primeiro a ser recuperado. Vejamos o código a seguir:

```
public class TextoCareTaker {
    protected ArrayList<TextoMemento> estados;
    public TextoCareTaker() {
        estados = new ArrayList<TextoMemento>();
    }
    public void adicionarMemento(TextoMemento memento) {
        estados.add(memento);
    }
    public TextoMemento getUltimoEstadoSalvo() {
        if (estados.size() <= 0) {
            return new TextoMemento("");
```

```
        }
        TextoMemento estadoSalvo = estados.get(estados.size() - 1);
        estados.remove(estados.size() - 1);
        return estadoSalvo;
    }
}
```

A partir do `Caretaker` é possível armazenar e recuperar um estado. No método que retorna o último estado salvo é necessário fazer uma verificação se existe algum estado a ser retornado, caso contrário é retornado um memento vazio. Uma pequena observação: retornar nulo exigiria uma verificação que poderia facilmente ser esquecida, causando vários problemas na execução do programa. O ideal seria disparar uma exceção, mas como estamos apenas exemplificando, retornar um objeto que não tenha informações é mais simples. Lembre-se: se você precisa retornar nulo, é melhor repensar no seu método. Agora vamos analisar a classe que representa o Texto:

```
public class Texto {
    protected String texto;
    TextoCareTaker caretaker;
    public Texto() {
        caretaker = new TextoCareTaker();
        texto = new String();
    }
    public void escreverTexto(String novoTexto) {
        caretaker.adicionarMemento(new TextoMemento(texto));
        texto += novoTexto;
    }
    public void desfazerEscrita() {
        texto = caretaker.getUltimoEstadoSalvo().getTextoSalvo();
    }
    public void mostrarTexto() {
        System.out.println(texto);
    }
}
```

A classe texto possui uma interface que permite escrever um texto, desfazer a operação de escrita e exibir o texto no terminal. Ao escrever um novo texto, primeiro o estado é salvo, então a alteração é feita. Ao desfazer a escrita é solicitado ao `Caretaker` que pegue o último estado salvo, a partir deste estado é possível pegar o texto e restaurá-lo. A utilização do padrão seria algo do tipo:

```
public static void main(String[] args) {
    Texto texto = new Texto();
    texto.escreverTexto("Primeira linha do texto\n");
```

```
    texto.escreverTexto("Segunda linha do texto\n");
    texto.escreverTexto("Terceira linha do texto\n");
    texto.mostrarTexto();
    texto.desfazerEscrita();
    texto.mostrarTexto();
    texto.desfazerEscrita();
    texto.mostrarTexto();
    texto.desfazerEscrita();
    texto.mostrarTexto();
    texto.desfazerEscrita();
    texto.mostrarTexto();
}
```

O diagrama UML para este exemplo seria:

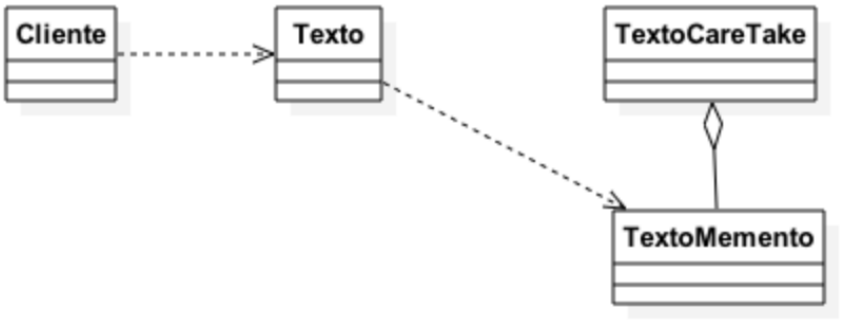

Figura 25 – O padrão de projetos Memento

Observer

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:

```
public class Dados {
    int valorA, valorB, valorC;
    public Dados(int a, int b, int c) {
        valorA = a;
        valorB = b;
        valorC = c;
    }
}
```

Como exemplo, vamos considerar que é necessário representar dados em uma tabela, que simplesmente exibe os números, 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 (FREEMAN e FREEMAN, 2007). Vamos ver então como o padrão *Observer* pode ajudar.

#DEFINIÇÃO#

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 (GAMMA et al., 2000).

#DEFINIÇÃO#

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:

```
public class DadosSubject {
    protected ArrayList&lt;DadosObserver&gt; observers;
    protected Dados dados;
    public DadosSubject() {
        observers = new ArrayList&lt;DadosObserver&gt;();
    }
    public void attach(DadosObserver observer) {
        observers.add(observer);
    }
    public void detach(int indice) {
        observers.remove(indice);
    }
}
```

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("Valor A: " + dados.getState().valorA
                + "Valor B: " + dados.getState().valorB
```

```
                + "Valor 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:

```
public class PorcentoObserver extends DadosObserver {
    public PorcentoObserver(DadosSubject d) {
        super(dados);
    }
    @Override
    public void update() {
        int soma = d.getState().valorA + d.getState().valorB + d.getState().valorC;
        DecimalFormat formatador = new DecimalFormat("#.##");
        String a = formatador.format((double) d.getState().valorA/soma);
        String b = formatador.format((double) d.getState().valorB/soma);
        String c = formatador.format((double) d.getState().valorC/soma);
        System.out.println("Valor A: " + a + "Valor B: " + b + "Valor C: " + c+
"%");
    }
}
```

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:

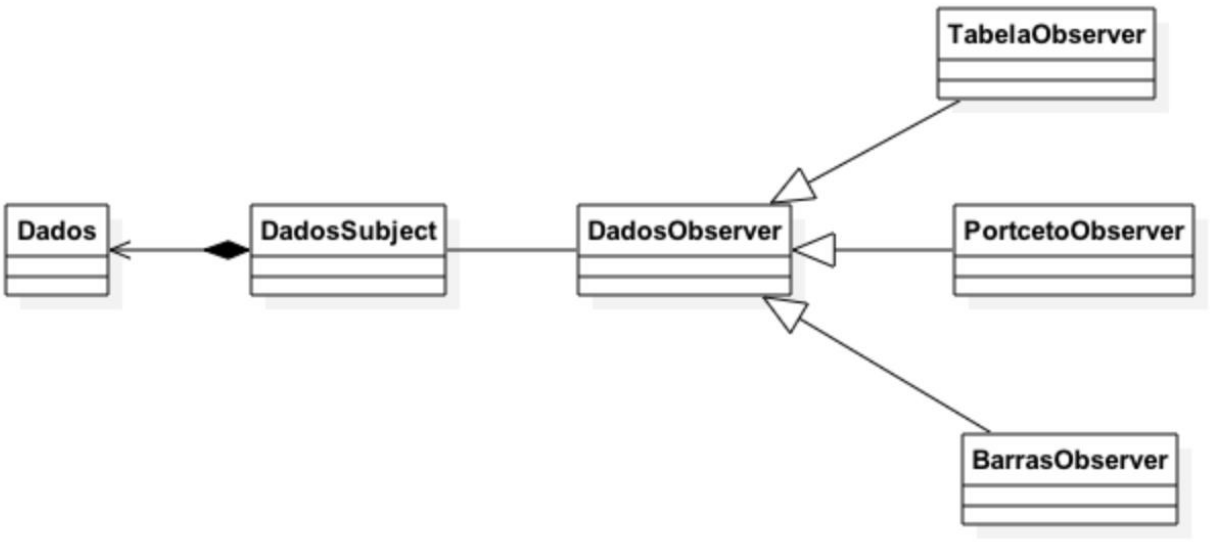

Figura 26 – O padrão de projetos Observer

State

A troca de estados de um objeto é um problema bastante comum. Tome como exemplo o personagem de um jogo, como o Mario. Durante o jogo acontecem várias trocas de estado com o Mario, por exemplo, ao pegar uma flor de fogo o Mario pode crescer, se estiver pequeno, e ficar com a habilidade de soltar bolas de fogo. Desenvolvendo um pouco mais o pensamento temos um conjunto grande de possíveis estados, e cada transição depende de qual é o estado atual do personagem (FREEMAN e FREEMAN, 2007). Como falado anteriormente, ao pegar uma flor de fogo podem acontecer quatro ações diferentes, dependendo de qual o estado atual do Mario:

- Se Mario pequeno -> Mario grande e Mario fogo
- Se Mario grande -> Mario fogo
- Se Mario fogo -> Mario ganha 1000 pontos
- Se Mario capa -> Mario fogo

Todas estas condições devem ser checadas para realizar esta única troca de estado. Agora imagine os vários estados e a complexidade para realizar a troca destes estados: Mario pequeno, Mario grande, Mario flor e Mario pena.

- Pegar Cogumelo:
    - Se Mario pequeno -> Mario grande
    - Se Mario grande -> 1000 pontos
    - Se Mario fogo -> 1000 pontos
    - Se Mario capa -> 1000 pontos

- Pegar Flor:
    - Se Mario pequeno -> Mario grande e Mario fogo
    - Se Mario grande -> Mario fogo
    - Se Mario fogo -> 1000 pontos
    - Se Mario capa -> Mario fogo

- Pegar Pena:
    - Se Mario pequeno -> Mario grande e Mario capa
    - Se Mario grande -> Mario capa
    - Se Mario fogo -> Mario fogo
    - Se Mario capa -> 1000 pontos

- Levar Dano:
    - Se Mario pequeno -> Mario morto
    - Se Mario grande -> Mario pequeno

- Se Mario fogo -> Mario grande
- Se Mario capa -> Mario grande

Com certeza não vale a pena investir tempo e código numa solução que utilize várias verificações para cada troca de estado. Para não correr o risco de esquecer de tratar algum estado e deixar o código bem mais fácil de manter, vamos analisar como o padrão *State* pode ajudar.

#DEFINIÇÃO#

Permite a um objeto alterar seu comportamento quando seu estado interno muda. O objeto parecerá ter mudado de classe (GAMMA et al., 2000).

#DEFINIÇÃO#

Pela intenção podemos ver que o padrão vai alterar o comportamento de um objeto quando houver alguma mudança no seu estado interno, como se ele tivesse mudado de classe. Para implementar o padrão será necessário criar uma classe que contém a interface básica de todos os estados. Como definimos anteriormente o que pode causar alteração nos estados do objeto Mario, estas serão as operações básicas que vão fazer parte da interface.

```
public interface MarioState {
    MarioState pegarCogumelo();
    MarioState pegarFlor();
    MarioState pegarPena();
    MarioState levarDano();
}
```

Agora todos os estados do `Mario` deverão implementar as operações de troca de estado. Note que cada operação retorna um objeto do tipo `MarioState`, pois como cada operação representa uma troca de estados, será retornado qual o novo estado o `Mario` deve assumir. Vejamos então uma classe para exemplificar um estado:

```
public class MarioPequeno implements MarioState {
    @Override
    public MarioState pegarCogumelo() {
        System.out.println("Mario grande");
        return new MarioGrande();
    }
    @Override
    public MarioState pegarFlor() {
        System.out.println("Mario grande com fogo");
        return new MarioFogo();
```

```
    }
    @Override
    public MarioState pegarPena() {
        System.out.println("Mario grande com capa");
        return new MarioCapa();
    }
    @Override
    public MarioState levarDano() {
        System.out.println("Mario morto");
        return new MarioMorto();
    }
}
```

Percebemos que a classe que define o estado é bem simples, apenas precisa definir qual estado deve ser trocado quando uma operação de troca for chamada. Vejamos agora outro exemplo de classe de estado:

```
public class MarioCapa implements MarioState {
    @Override
    public MarioState pegarCogumelo() {
        System.out.println("Mario ganhou 1000 pontos");
        return this;
    }
    @Override
    public MarioState pegarFlor() {
        System.out.println("Mario com fogo");
        return new MarioFogo();
    }
    @Override
    public MarioState pegarPena() {
        System.out.println("Mario ganhou 1000 pontos");
        return this;
    }
    @Override
    public MarioState levarDano() {
        System.out.println("Mario grande");
        return new MarioGrande();
    }
}
```

Bem simples não? Novos estados são adicionados de maneira bem simples. Vejamos então como seria o objeto que vai utilizar os estados, o Mario:

```
public class Mario {
    protected MarioState estado;
    public Mario() {
        estado = new MarioPequeno();
    }
    public void pegarCogumelo() {
        estado = estado.pegarCogumelo();
    }
    public void pegarFlor() {
        estado = estado.pegarFlor();
    }
    public void pegarPena() {
        estado = estado.pegarPena();
    }
    public void levarDano() {
        estado = estado.levarDano();
    }
}
```

A classe `Mario` possui uma referência para um objeto estado, este estado vai ser atualizado de acordo com as operações de troca de estados, definidas logo em seguida. Quando uma operação for invocada, o objeto estado vai executar a operação e se atualizará automaticamente. Como exemplo de utilização, vejamos o seguinte código cliente:

```
public static void main(String[] args) {
    Mario Mario = new Mario();
    mario.pegarCogumelo();
    mario.pegarPena();
    mario.levarDano();
    mario.pegarFlor();
    mario.pegarFlor();
    mario.levarDano();
    mario.levarDano();
    mario.pegarPena();
    mario.levarDano();
    mario.levarDano();
    mario.levarDano();
}
```

Por este código é possível avaliar todas as transições e todos os estados. O diagrama UML a seguir resume visualmente as relações entre as classes:

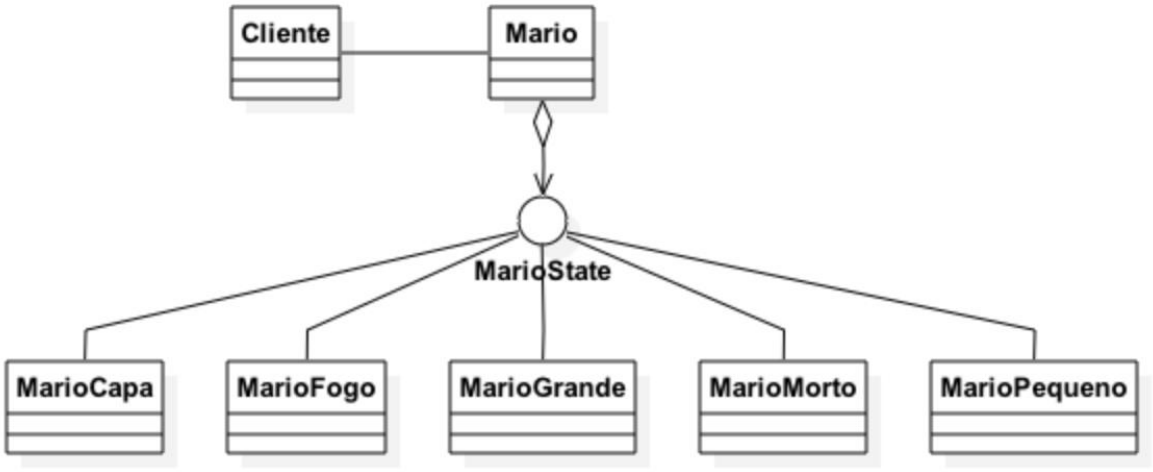

Figura 27 – O padrão de projetos State

Strategy

Suponha uma empresa, nesta empresa existem um conjunto de cargos, para cada cargo existem regras de cálculo de imposto, determinada porcentagem do salário deve ser retirada de acordo com o salário base do funcionário. Vamos as regras:

- O Desenvolvedor deve ter um imposto de 15% caso seu salário seja maior que R$ 2000,00 e 10% caso contrário;
- O Gerente deve ter um imposto de 20% caso seu salário seja maior que R$ 3500,00 e 15% caso contrário;
- O DBA deve ter um imposto de 15% caso seu salário seja maior que R$ 2000,00 e 10% caso contrário;

Até aí tudo bem, uma solução bem simples seria criar uma classe para representar um funcionário e dentro dele um campo para guardar seu cargo e salário. No método de cálculo de imposto utilizaríamos um switch para selecionar o cargo e depois verificaríamos o salário, para saber qual a porcentagem de imposto que deve ser utilizada. Vamos dar uma olhada no método que calcula o salário do funcionário aplicando o imposto:

```
public double calcularSalarioComImposto() {
        switch (cargo) {
        case DESENVOLVEDOR:
            if (salarioBase >= 2000) {
                return salarioBase * 0.85;
            } else {
                return salarioBase * 0.9;
```

```
            }
        case GERENTE:
            if (salarioBase >= 3500) {
                return salarioBase * 0.8;
            } else {
                return salarioBase * 0.85;
            }
        case DBA:
            if (salarioBase >= 2000) {
                return salarioBase * 0.85;
            } else {
                return salarioBase * 0.9;
            }
        default:
            throw new RuntimeException("Cargo não encontrado :/");
        }
    }
```

Este método é uma ótima prova do porque existem tantas vagas para desenvolvimento de software por aí. Pense na seguinte situação: Surgiu um novo cargo que precisa ser cadastro, este cargo deve utilizar as mesmas regras de negócio do cargo DBA. O que seria necessário para incluir esta nova funcionalidade? Um novo case com novos if e else. Fácil não? Imagine agora que depois de todo o trabalho para inserir todos os possíveis cargos de uma empresa, e uma regra muda? Seria awesome dar manutenção neste código não?

#DEFINIÇÃO#

Definir uma família de algoritmos, encapsular cada uma delas e torná-las intercambiáveis. *Strategy* permite que o algoritmo varie independentemente dos clientes que o utilizam (GAMMA et al., 2000).

#DEFINIÇÃO#

Ou seja, o padrão sugere que algoritmos parecidos (métodos de cálculo de Imposto) sejam separados de que os utiliza (Funcionário). Certo, e como fazer isso? Bom, a primeira parte é encapsular todos os algoritmos da mesma família. No nosso exemplo a família de algoritmos é a que calcula salários com impostos, então para encapsulá-las criamos uma classe interface (Java) ou abstrata pura (C++). Vamos lá então:

```
interface CalculaImposto {
    double calculaSalarioComImposto(Funcionario funcionario);
}
```

Uma vez definida a classe que encapsula os algoritmos vamos definir as estratégias concretas de cálculo de imposto, a seguir o código para calculo de imposto de 15% ou 10%:

```
public class CalculoImpostoQuinzeOuDez implements CalculaImposto {
    @Override
    public double calculaSalarioComImposto(Funcionario funcionario) {
        if (funcionario.getSalarioBase() > 2000) {
            return funcionario.getSalarioBase() * 0.85;
        }
        return funcionario.getSalarioBase() * 0.9;
    }
}
```

As outras estratégias seguem este mesmo padrão, então vamos partir agora para as alterações na classe Funcionário. Esta classe depende da classe `CalculoImposto`, ou seja, ela utiliza um objeto `CalculoImposto`. Mas, como eu vou utilizar um objeto interface? Simples, este objeto será instanciado em tempo de execução e, de acordo com o Cargo dele a estratégia de cálculo correta será utilizada. No construtor, de acordo com o cargo nós configuramos a estratégia de cálculo correta:

```
public Funcionario(int cargo, double salarioBase) {
        this.salarioBase = salarioBase;
        switch (cargo) {
        case DESENVOLVEDOR:
            estrategiaDeCalculo = new CalculoImpostoQuinzeOuDez();
            cargo = DESENVOLVEDOR;
            break;
        case DBA:
            estrategiaDeCalculo = new CalculoImpostoQuinzeOuDez();
            cargo = DBA;
            break;
        case GERENTE:
            estrategiaDeCalculo = new CalculoImpostoVinteOuQuinze();
            cargo = GERENTE;
            break;
        default:
            throw new RuntimeException("Cargo não encontrado :/");
        }
    }
```

E agora a única coisa que precisamos fazer para calcular o salário com

imposto é:

```
public double calcularSalarioComImposto() {
    return estrategiaDeCalculo.calculaSalarioComImposto(this);
}
```

Agora sim está simples.

Template Method

Suponha um player de música que oferece várias maneiras de reproduzir as músicas de uma playlist. Para exemplificar suponha que podemos reproduzir a lista de músicas da seguinte maneira:

- Ordenado por nome da música
- Ordenado por nome do Autor
- Ordenado por ano
- Ordenado por estrela (preferência do usuário)

Uma ideia seria utilizar o padrão *Strategy* e implementar uma classe que define o método de reprodução para cada tipo de reprodução da playlist. Esta seria uma solução viável, pois manteríamos a flexibilidade para implementar novos modos de reprodução de maneira bem simples. No entanto, observe que, o algoritmo para reprodução de uma playlist é o mesmo, independente de qual modo está sendo utilizado. A única diferença é a criação da playlist, que leva em consideração um dos atributos da música. Para suprir esta dificuldade vamos ver o padrão *Template Method*!

#DEFINIÇÃO#

Definir o esqueleto de um algoritmo em uma operação, postergando alguns passos para as subclasses. *Template Method* permite que subclasses redefinam certos passos de um algoritmo sem mudar a estrutura do mesmo (GAMMA et al., 2000).

#DEFINIÇÃO#

Perfeito para o nosso problema! Precisamos definir o método de ordenação da Playlist mas só saberemos qual atributo utilizar em tempo de execução. Vamos definir então a estrutura de dados que define uma música:

```
public class MusicaMP3 {
    String nome;
    String autor;
```

```
    String ano;
    int estrelas;
    public MusicaMP3(String nome, String autor, String ano, int estrela) {
        this.nome = nome;
        this.autor = autor;
        this.ano = ano;
        this.estrelas = estrela;
    }
}
```

Para escolher como a playlist deve ser ordenada vamos criar uma pequena enumeração:

```
public enum ModoDeReproducao {
    porNome, porAutor, porAno, porEstrela
}
```

Agora vamos escrever a nossa classe que implementa o método template para ordenação da lista:

```
public abstract class OrdenadorTemplate {
    public abstract boolean isPrimeiro(MusicaMP3 musica1, MusicaMP3
musica2);
    public ArrayList<MusicaMP3> ordenarMusica(ArrayList<MusicaMP3> lista) {
        ArrayList<MusicaMP3> novaLista = new ArrayList<MusicaMP3>();
        for (MusicaMP3 musicaMP3 : lista) {
            novaLista.add(musicaMP3);
        }
        for (int i = 0; i < novaLista.size(); i++) {
            for (int j = i; j < novaLista.size(); j++) {
                if (!isPrimeiro(novaLista.get(i), novaLista.get(j))) {
                    MusicaMP3 temp = novaLista.get(j);
                    novaLista.set(j, novaLista.get(i));
                    novaLista.set(i, temp);
                }
            }
        }
        return novaLista;
    }
}
```

Basicamente, definimos um método de ordenação, no caso o método `Bolha`,

e deixamos a comparação dos atributos para as subclasses. Essa é a ideia de utilizar o método template, definir o esqueleto e permitir a personalização dele nas subclasses. Veja o exemplo da classe a seguir que ordena as músicas por nome:

```
public class OrdenadorPorNome extends OrdenadorTemplate {
    @Override
    public boolean isPrimeiro(MusicaMP3 musica1, MusicaMP3 musica2) {
        if (musica1.nome.compareToIgnoreCase(musica2.nome) <= 0) {
            return true;
        }
        return false;
    }
}
```

Para implementar as outras formas de reprodução da lista bastam definir, na subclasse, o método que compara uma música com outra e diz se é necessário trocar. O exemplo a seguir define a ordenação baseado no nível de preferência do usuário (aquelas estrelinhas dos players de música)

```
public class OrdenadorPorEstrela extends OrdenadorTemplate {
    @Override
    public boolean isPrimeiro(MusicaMP3 musica1, MusicaMP3 musica2) {
        if (musica1.estrelas > musica2.estrelas) {
            return true;
        }
        return false;
    }
}
```

Pronto, definimos o algoritmo padrão e suas variações. Agora vamos ver a classe que manipula a playlist:

```
public class PlayList {
    protected ArrayList<MusicaMP3> musicas;
    protected OrdenadorTemplate ordenador;
    public PlayList(ModoDeReproducao modo) {
        musicas = new ArrayList<MusicaMP3>();
        switch (modo) {
        case porAno:
            ordenador = new OrdenadorPorAno();
            break;
        case porAutor:
```

```
            ordenador = new OrdenadorPorAutor();
            break;
        case porEstrela:
            ordenador = new OrdenadorPorEstrela();
            break;
        case porNome:
            ordenador = new OrdenadorPorNome();
            break;
        default:
            break;
        }
    }
  public void setModoDeReproducao(ModoDeReproducao modo) {
        ordenador = null;
        switch (modo) {
        case porAno:
            ordenador = new OrdenadorPorAno();
            break;
        case porAutor:
            ordenador = new OrdenadorPorAutor();
            break;
        case porEstrela:
            ordenador = new OrdenadorPorEstrela();
            break;
        case porNome:
            ordenador = new OrdenadorPorNome();
            break;
        default:
            break;
        }
    }
    public void adicionarMusica(String n, String a, String ano, int e) {
        musicas.add(new MusicaMP3(n, a, ano, e));
    }
    public void mostrarListaDeReproducao() {
        ArrayList<MusicaMP3> novaLista = new ArrayList<MusicaMP3>();
        novaLista = ordenador.ordenarMusica(musicas);
        for (MusicaMP3 musica : novaLista) {
            System.out.println(musica.nome + "-" + musica.estrelas);
        }
    }
}
```

Definimos métodos para inserir músicas e exibir a playlist e, de acordo com o parâmetro passado, criamos uma playlist. O código cliente ficaria da seguinte maneira:

```
public static void main(String[] args) {
    PlayList minhaPlayList = new PlayList(ModoDeReproducao.porNome);
    minhaPlayList.adicionarMusica("Everlong", "Foo Fighters", "1997", 5);
    minhaPlayList.adicionarMusica("Song 2", "Blur", "1997", 4);
    minhaPlayList.adicionarMusica("American ", "Bad Religion", "1993", 3);
    minhaPlayList.adicionarMusica("No Cigar", "Milencollin", "2001", 2);
    minhaPlayList.adicionarMusica("Ten", "Pearl Jam", "1991", 1);
    System.out.println("Lista por Nome de Musica");
    minhaPlayList.mostrarListaDeReproducao();
    System.out.println("Lista por Autor");
    minhaPlayList.setModoDeReproducao(ModoDeReproducao.porAutor);
    minhaPlayList.mostrarListaDeReproducao();
    System.out.println("Lista por Ano");
    minhaPlayList.setModoDeReproducao(ModoDeReproducao.porAno);
    minhaPlayList.mostrarListaDeReproducao();
    System.out.println("Lista por Estrela");
    minhaPlayList.setModoDeReproducao(ModoDeReproducao.porEstrela);
    minhaPlayList.mostrarListaDeReproducao();
}
```

Veja o quão simples foi alterar o modo como a lista é construída. No final, essa é a estrutura do projeto utilizando o *Template Method*:

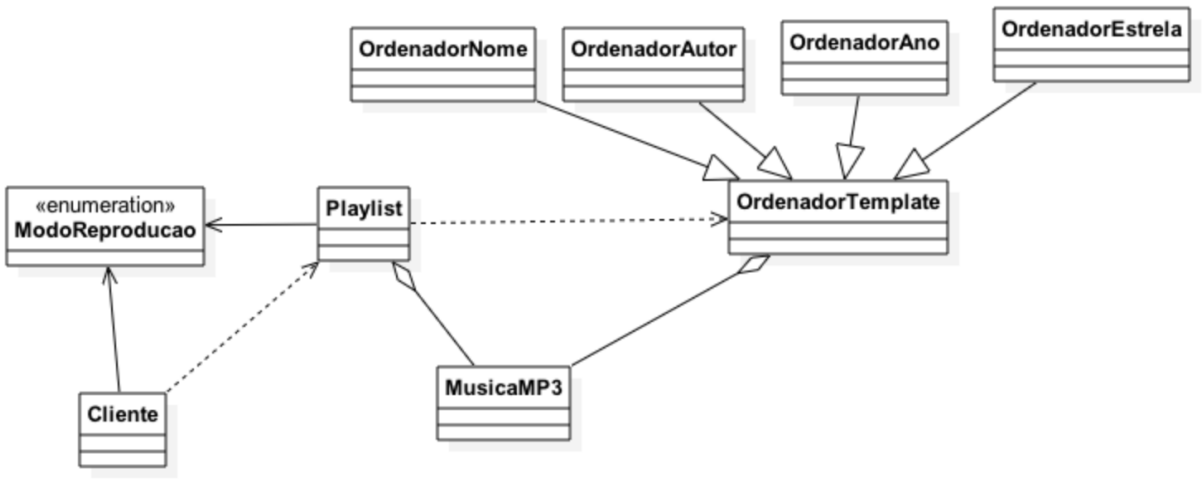

Figura 28 – O padrão de projetos Template Method

Visitor

É comum em projetos, onde você precisa criar sua própria estrutura de dados, que sejam necessários implementar várias operações sobre este conjunto de dados. Geralmente, esta responsabilidade é delegada para a própria classe que representa a estrutura. Para discutirmos melhor, suponha que em um projeto você precisa lidar com uma estrutura de dados muito complexa, uma árvore binária por exemplo. É comum que sejam implementados métodos para percorrer a árvore, como por exemplo: `in-order, pre-order e post-order`. O problema em implementar estes métodos diretamente na classe que representa a árvore é que é preciso ter um alto nível de certeza de que estas operações não mudarão, ou que serão utilizadas apenas nesta estrutura, pois o custo para realizar alterações seria muito grande, uma vez que o sistema dependerá da definição desta classe. Outro problema é que, várias outras estruturas de dados vão utilizar uma abordagem semelhante (FREEMAN e FREEMAN, 2007). Por exemplo, caso seja implementado uma árvore AVL, que utiliza os mesmos dados e as mesmas operações de percurso, não seria possível reutilizar este método. Vejamos então um pouco sobre o padrão *Visitor*, que vai nos ajudar bastante a resolver este problema.

#DEFINIÇÃO#

Representar uma operação a ser executada nos elementos de uma estrutura de objetos. *Visitor* permite definir uma nova operação sem mudar as classes dos elementos sobre os quais opera (GAMMA et al., 2000).

#DEFINIÇÃO#

Pela intenção já é possível ver como o padrão vai nos ajudar. A sua ideia é separar as operações que serão executadas em determinada estrutura de sua representação. Assim, incluir ou remover operações não terá nenhum efeito sobre a interface da estrutura, permitindo que o resto do sistema funcione sem depender de operações específicas. Vamos então começar definindo a estrutura de dados a ser utilizada. A seguinte classe representa o nó da árvore:

```
public class No {
    protected int chave;
    No esquerdo, direito;
    public No(int chave) {
        this.chave = chave;
        esquerdo = null;
        direito = null;
    }
    public String toString() {
        return String.valueOf(chave);
    }
    public int getChave() {
        return chave;
    }
    public No getEsquerdo() {
```

```
        return esquerdo;
    }
    public void setEsquerdo(No esquerdo) {
        this.esquerdo = esquerdo;
    }
    public No getDireito() {
        return direito;
    }
    public void setDireito(No direito) {
        this.direito = direito;
    }
}
```

Definimos então o nó básico da árvore, que contém a chave e os nós esquerdo e direito. Além disso o método `toString()` permite que a estrutura seja exibida na tela com o sysout. Agora vamos ver a estrutura árvore, que vai manter todos estes elementos.

```
public class ArvoreBinaria {
    No raiz;
    int quantidadeDeElementos;
    public ArvoreBinaria(int chaveRaiz) {
        raiz = new No(chaveRaiz);
        quantidadeDeElementos = 0;
    }
    public void inserir(int chave) {}
    public void remover(int chave) {}
    public void buscar(int chave) {}
    public void aceitarVisitante(ArvoreVisitor visitor) {
        visitor.visitar(raiz);
    }
}
```

A estrutura árvore vai então possuir o nó raiz da árvore e algumas outras informações sobre a árvore. Omitimos a implementação dos métodos da árvore para simplificar o exemplo. O detalhe importante desta classe é o método aceitarVisitante(), ele recebe um objeto `ArvoreVisitor` e passa a sua raiz para ele. Aí começa a implementação do padrão de verdade. A estrutura de dados vai possuir um método que recebe um objeto visitante. Deste método ela vai chamar o método visitar, do objeto visitante, e vai passar os seus dados para o objeto visitante. Dai em diante, o objeto visitante vai poder realizar as operações necessárias. Vamos então começar com a implementação de `ArvoreVisitor`:

```
public interface ArvoreVisitor {
    void visitar(No no);
}
```

Esta classe vai apenas definir a interface de visita de um nó. Todas as operações vão receber um objeto `No` e a partir dai vão implementar suas operações. Por exemplo, vejamos a implementação de um método de percurso `in-order`.

```
public class ExibirInOrderVisitor implements ArvoreVisitor {
    @Override
    public void visitar(No no) {
        if (no == null) {
            return;
        }
        this.visitar(no.getEsquerdo());
        System.out.println(no);
        this.visitar(no.getDireito());
    }
}
```

Seguindo a ideia de implementação de percurso `in-order` (LARMAN, 2007), primeiro é feita a visita ao nó esquerdo, em seguida mostramos no terminal o nó e ao fim é feita a visita ao nó direito. Com esta simples implementação temos o método de percurso `in-order` da árvore, sem precisar alterar a sua estrutura. Implementar várias outras operações fica muito simples ao utilizar este padrão. As implementações dos outros métodos de percurso ficam triviais. Uma implementação mais complexa, também fica mais simplificada. Por exemplo, caso seja necessário implementar um método que exibe os nós de uma maneira identada, de acordo com seu nível na árvore.

```
public class ExibirIndentadoVisitor implements ArvoreVisitor {
    @Override
    public void visitar(No no) {
        if (no == null) {
            return;
        }
        System.out.println(no);
        visitar(no.getEsquerdo(), 1);
        visitar(no.getDireito(), 1);
    }
    private void visitar(No no, int qtdEspacos) {
        if (no == null) {
            return;
```

```
        }
        for (int i = 0; i < qtdEspacos; i++) {
            System.out.print("-");
        }
        System.out.println(no);
        visitar(no.getEsquerdo(), qtdEspacos + 1);
        visitar(no.getDireito(), qtdEspacos + 1);
    }
}
```

Este método conta a quantidade de espaços para identação a partir de cada nível de recursão do método. A única restrição das classes visitantes é que implementem o método para visitar. Quaisquer outras operações que precisem de suporte podem ser implementadas. Um exemplo de cliente seria:

```
public static void main(String[] args) {
    ArvoreBinaria arvore = new ArvoreBinaria(7);
    arvore.inserir(15);
    arvore.inserir(10);
    arvore.inserir(5);
    arvore.inserir(2);
    arvore.inserir(1);
    arvore.inserir(20);

    System.out.println("Exibindo em ordem");
    arvore.aceitarVisitante(new ExibirInOrderVisitor());
    System.out.println("Exibindo pré ordem");
    arvore.aceitarVisitante(new ExibirPreOrdemVisitor());
    System.out.println("Exibindo pós ordem");
    arvore.aceitarVisitante(new ExibirPostOrderVisitor());
    System.out.println("Exibindo identado");
    arvore.aceitarVisitante(new ExibirIndentadoVisitor());
}
```

O diagrama UML que representa esta solução é o seguinte:

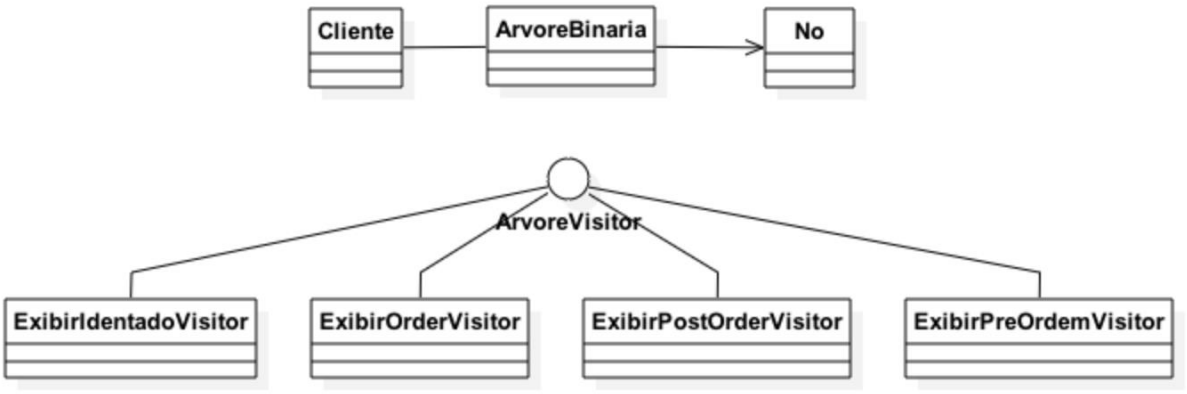

Figura 29 – O padrão de projetos Visitor

Como identificar a necessidade desses padrões

Os padrões comportamentais estão relacionados com o conceito de coesão. Dessa forma, sempre que houver a necessidade de escolher qual classe fica com qual responsabilidade é interessante recorrer a algum desses padrões. Eles auxiliam colocar atributos e métodos nas classes corretas e relacionando-os de forma correta.

Considerações Finais

Neste estudo entendemos os padrões comportamentais e como os mesmos podem ser aplicados aos projetos de software orientados a objetos. Esses padrões são fundamentais para a coesão do software e facilitam o entendimento e testes ao longo do ciclo de vida do produto.

UNIDADE 5 – CENÁRIOS DE UTILIZAÇÃO

Objetivos de aprendizagem

- Cenários comuns de utilização dos padrões de projetos
- Como evitar a febre dos padrões
- Boas práticas de desenho

Cenários comuns de utilização dos padrões de projetos

Independente do tipo do padrão é sempre importante entender bem o cenário de utilização dos mesmos. Em sistemas especialistas, baseados em regras *se-então*, esses padrões podem ser de difícil identificação de utilização. Geralmente os mesmos estão associados com bons projetos arquiteturais para prover a manutenabilidade do software ao longo do ciclo de vida.

Por exemplo, os padrões criacionais podem ser empregados para a criação de objetos como Pedidos, Faturas ou demais objetos que possam ter muitos atributos e que, no momento da sua criação, os mesmos precisem alguns códigos específicos para criação que o construtor da classe não possa fornecer. Já os padrões estruturais podem ser usados para reduzir muitos níveis de hierarquias prejudicando assim a abstração de métodos e atributos. Por fim, os padrões comportamentais pode ser usados para encapsulamento de comportamentos e reduzir o espalhamento de regras ao longo de várias camadas do software.

Como evitar a febre dos padrões

Uma fenômeno comum que acontece quando começamos a dominar a utilização dos padrões é conhecida como “febre dos padrões”. Em resumo, isso está relacionado com utilizar padrões de projetos sem necessidade. Padrões de projetos tem consequências e se bem aplicados os benefícios são maiores do que os benefícios. Da mesma forma, quando mal aplicados eles tendem a gerar problemas maiores do que benefícios. Por exemplo, quando utilizamos um padrão como o Builder na criação de objetos que um construtor seja o suficiente, adicionamos uma complexidade desnecessária e que dificulta a manutenção do software. Do ponto de vista arquitetural isso é chamado de *complexidade acidental*. Dessa forma, utilize os padrões caso seja necessário.

Boas práticas de desenho

Ao projetar software precisamos saber como criar as classes e principalmente como as mesmas se relacionam entre si. Dessa forma, a utilização de padrões podem ajudar nessa tarefa. Além disso, podemos lançar mão dos padrões de projeto para criar uma estrutura que possa acomodar facilmente modificações que possam (em sempre vão) surgir e se não bem projetado, haverá

sempre muito esforço para modificar o software evitando os efeitos colaterais.

Considerações Finais

Nesse estudo abordamos os padrões de projetos do GoF. Apesar desses padrões serem o suficiente para resolver a maior parte dos problemas que são conhecidos (e padronizados) em software. Adicionalmente, existem mais projetos sendo um deles os padrões GRASP, sigla para *General Responsibility Assignment Software Patterns (or Principles),* consistem de um conjunto de práticas para atribuição de responsabilidades a classes e objetos em projetos orientados a objeto. Existem também os padrões de projetos que foram criados para a tecnologia Java conhecido como J2EE *Blueprints*[1]. Utilize os padrões com sabedoria e sempre para melhorar a qualidade do software.

[1] https://www.oracle.com/technetwork/java/catalog-137601.html

Referências

LARMAN, Craig. Utilizando UML e Padrões: Uma Introdução à Análise e ao Projeto Orientados a Objetos e ao Desenvolvimento Iterativo - 3ª Ed., Bookman, 2007.

FREEMAN, Elisabeth; FREEMAN, Eric. Use a Cabeça! Padrões de Projetos - 2ª Ed., Alta Books, 2007.

GAMMA, Erich et al. Padrões de Projeto: Soluções Reutilizáveis de Software Orientado a Objetos. Bookman, 2000.