Introdução

Polimorfismo significa "muitas formas", é o termo definido em linguagens orientadas a objeto, como por exemplo Java, C# e C++, que permite ao desenvolvedor usar o mesmo elemento de formas diferentes. Polimorfismo denota uma situação na qual um objeto pode se comportar de maneiras diferentes ao receber uma mensagem. No Polimorfismo temos dois tipos:

- Polimorfismo Estático ou Sobrecarga
- Polimorfismo Dinâmico ou Sobreposição

O Polimorfismo Estático se dá quando temos a mesma operação implementada várias vezes na mesma classe. A escolha de qual operação será chamada depende da assinatura dos métodos sobrecarregados.

O Polimorfismo Dinâmico acontece na herança, quando a subclasse sobrepõe o método original. Agora o método escolhido se dá em tempo de execução e não mais em tempo de compilação. A escolha de qual método será chamado depende do tipo do objeto que recebe a mensagem.

Nos próximos tópicos veremos um pouco mais sobre os dois tipos de polimorfismo discutidos acima e exemplos de como o Polimorfismo é implementado e utilizado na prática em diferentes contextos.

Sobrescrita de Métodos e Sobrecarga

A Sobrescrita de Métodos pode ser classificada como polimorfismo de inclusão. Quando um método sobrescreve um método herdado de uma classe, temos uma sobrescrita de método. Este método de sobrescrita tem que ser idêntico ao método da classe herdada, ou seja, eles precisam ter o mesmo nome, valor de retorno e argumentos. Portanto, temos que uma classe filha fornece apenas uma nova implementação para o método herdado e não um novo método. Por exemplo, temos uma superclasse Forma e as suas subclasses Triangulo e Circulo. Digamos que a superclasse Forma tem um método chamado calculaArea(). Cada uma das subclasses Triangulo e Circulo definirá o seu próprio método calculaArea(). Dependendo do tipo de objeto que for criado teremos a execução do método dessa subclasse.

Se a classe filha fornecer um método de cabeçalho ou assinatura parecida com a do método herdado (difere ou no número ou no tipo dos argumentos) então não se trata mais de redefinição, trata-se de uma sobrecarga, pois criou-se um novo método. Uma chamada ao método herdado não mais será interceptada por esse novo método de mesmo nome. O método tem o mesmo nome, mas é ligeiramente

diferente na sua assinatura (o corpo ou bloco de código não importa), o que já implica que não proporciona o mesmo comportamento do método da superclasse. Por exemplo, se temos um método soma que aceita um inteiro como parâmetro e um outro método soma que recebe dois parâmetros inteiros. O método que será chamado depende dos argumentos sendo passados na chamada deste método, portanto, se chamarmos soma(1,3) o segundo método será chamado.

O tipo de polimorfismo de Sobrecarga permite a existência de vários métodos de mesmo nome, porém com assinaturas levemente diferentes, ou seja, variando no número e tipo de argumentos. Ficaria a cargo do compilador escolher de acordo com as listas de argumentos os procedimentos ou métodos a serem executados.

Sobrecarga de Métodos é comumente usada nos construtores de uma classe Java.

Utilizando Polimorfismo em Padrões de Projetos

Boa parte dos padrões de projeto de software baseia-se no uso de polimorfismo, por exemplo: Abstract Factory, Composite, Observer, Strategy, Template Method, etc.

A partir de agora será visto como funciona o Abstract Factory, um padrão de projeto muito utilizado que faz uso do polimorfismo.

Padrões de Projeto representam soluções comprovadas para problemas recorrentes em desenvolvimento de software. A ideia original surgiu em 1979, na Arquitetura e Engenharia Civil com Christopher Alexander, arquiteto, que queria melhorar o processo de projeto de edifícios e áreas urbanas. No entanto, na Engenharia de Software, quatro autores se basearam em Christopher Alexander para criar Padrões de Projeto de software. Em 1994 os quatro autores, também chamados de Gang of Four, descreveram 23 padrões em seu livro "Padrões de Projeto: Soluções Reutilizáveis de Software Orientado a Objetos". O livro já vendeu mais de 500 mil cópias e tem praticamente uma atualização por ano.

Conforme os autores, Erich Gamma, et. al, um padrão de projeto é definido como: “Descrição de uma solução para resolver um problema genérico de projeto em um contexto específico. [...] Um padrão de projeto dá nome, abstrai e identifica os aspectos-chave de uma estrutura de projeto comum para torná-la reutilizável”.

Portanto, os padrões de projetos são especialmente bons para descrever como e por que resolver problemas não funcionais facilitando o reuso de soluções arquiteturais que já deram certo no passado.

O Abstract Factory proporciona uma interface para criar famílias de objetos relacionados ou dependentes sem especificar suas classes concretas.

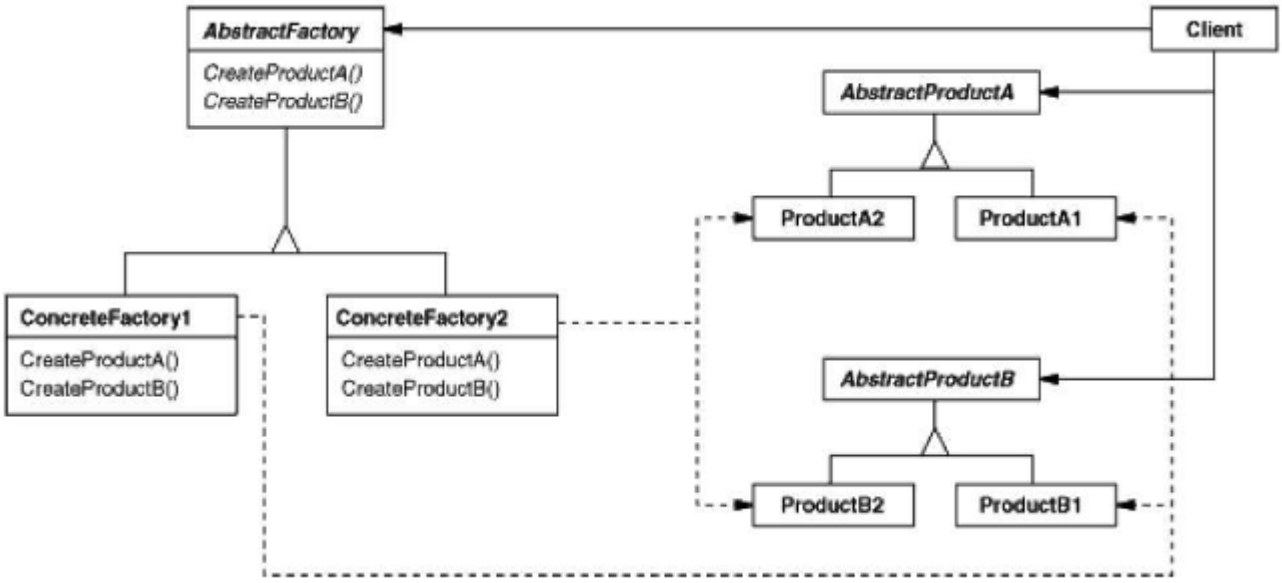

Figura 1: Exemplo de implementação do Abstract Factory

Abaixo segue um exemplo de implementação do Abstract Factory.

Listagem 1: Implementação da fábrica abstrata e das fábricas concretas

```
interface GUIFactory

        public Menu createMenu();

}



class WinFactory implements GUIFactory {



        public Menu createMenu() {

                return new WinMenu();

        }
```

```
}


class LinuxFactory implements GUIFactory {

        public Menu createMenu() {

                return new LinuxMenu();

        }

}
```

Na listagem 1 tem-se a fábrica abstrata “GUIFactory” que é implementada pelas fábricas concretas WinFactory e LinuxFactory que vão criar a mesma coisa, ou seja, um menu, no entanto cada sistema tem seu tipo de menu com seu próprio tamanho, tipo, design, etc.

Abaixo tem-se a implementação dos menus para cada sistema.

Listagem 2: Implementação do produto abstrato e dos produtos concretos

```
interface Menu {

        public void paint();

}



class WinMenu implements Menu {

        public void paint() {

                System.out.println("Eu sou um WinMenu");

        }

}
```

```
class LinuxMenu implements Menu {

        public void paint() {

                System.out.println("Eu sou um LinuxMenu");

        }

}
```

A interface Menu cria o produto abstrato que deve ser implementado pelos produtos concretos WinMenu e LinuxMenu. Podemos notar que temos aqui as classes WinMenu e LinuxMenu com os métodos paint() que sobrescrevem o método paint() da interface Menu.

No código abaixo temos a execução do padrão de projeto implementado acima.

Listagem 3: Teste de execução do padrão

```
class Aplicacao {

        public Aplicacao(GUIFactory factory) {

                Menu menu = factory.createMenu();

                menu.paint();

        }

}



class Principal {

        public static void main(String args[]) {

                chamar Application();
```

```
                //De preferencia leia de algum lugar e coloque a opção na
variável

                int tipoDeMenu = 0;

                if (tipoDeMenu == 0)

                        new Aplicacao(new WinFactory());

                else

                        new Aplicacao(new LinuxFactory());

        }

}
```

Acima podemos notar que dependendo do parâmetro que for passado para a classe Aplicacao podemos ter um retorno diferente, porém a chamada será sempre a mesma (paint).

Aqui temos um exemplo do polimorfismo ajudando na implementação do Padrão de Projeto Abstract Factory.

Utilizando Polimorfismo em Refatorações

Refatoração é uma (pequena) modificação no sistema que não altera o seu comportamento funcional, mas que torna o código mais fácil de ser entendido e menos custoso de ser alterado. A refatoração visa sempre dar uma maior simplicidade, flexibilidade e clareza para o código.

Alguns exemplos de refatoração são: Alterar nomes de variáveis, métodos e objetos para melhorar legibilidade do código, mudar parâmetros de métodos para que fiquem mais claros, eliminar duplicação de código, etc.

Um catálogo com mais de 72 refatorações pode ser encontrado no livro de Martin Fowler, "Refatoração: Aperfeiçoando o Projeto de Código Existente".

O polimorfismo também é usado em uma série de refatorações, como na refatoração “Substituir Comando Condicional por Polimorfismo“ que será mais detalhada abaixo.

A refatoração “Substituir Comando Condicional por Polimorfismo” move cada ramificação de um comando condicional para um método de sobrescrita em uma subclasse.

Essa refatoração deixa bastante claro um dos grandes objetivos do Polimorfismo que é evitar escrever um comando condicional explícito quando temos objetos cujo comportamento varia de acordo com os seus tipos. O grande ganho dessa refatoração é quando temos um grande número dessas condicionais que aparecem em muitos lugares do programa. Se quisermos adicionar um novo tipo, temos que encontrar e atualizar todos os comandos condicionais. Com subclasses precisamos apenas criar uma nova subclasse, isso reduz consideravelmente as dependências no sistema e torna-o muito mais fácil de atualizá-lo.

No exemplo abaixo tem-se uma condicional.

Listagem 4: Exemplo utilizando condicional para definir o comportamento

```
class TipoEmpregado {



        public int quantiaAPagar() {

                switch(lerTipo()) {

                        case TipoDeEmpregado.ENGENHEIRO:

                                return _salarioMensal;

                        case TipoDeEmpregado.VENDEDOR:

                                return _salarioMensal + _comissao;

                        case TipoDeEmpregado.GERENTE:

                                return _salarioMensal + _bonus;

                        default:

                                throw new RuntimeException("tipo incorreto
de empregado");
```

```
                }

        }

}
```

Todas essas condicionais, após aplicar a refatoração “Substituir Comando Condicional por Polimorfismo“, vira a implementação abaixo:

Listagem 5: Código condicional após aplicar a refatoração

```
//Essa classe que vai chamar o TipoDeEmpregado dependendo do tipo passado

class Empregado {

        int quantiaAPagar() {

                return _tipo.quantiaAPagar(this);

        }

}



class TipoDeEmpregado {



        abstract int quantiaAPagar(Empregado emp);

}



class Engenheiro extends TipoDeEmpregado {

        int quantiaAPagar(Empregado emp) {

                return emp.lerSalarioMensal();
```

```
        }

}


class Vendedor extends TipoDeEmpregado {

        int quantiaAPagar(Empregado emp) {

                return emp.lerSalarioMensal() + emp.lerComissao();

        }

}


class Gerente extends TipoDeEmpregado {

        int quantiaAPagar(Empregado emp) {

                return emp.lerSalarioMensal() + emp.lerBonus();

        }

}
```

A classe Empregado vai delegar para uma classe mais específica o trabalho a ser feito. As classes específicas Engenheiro, Vendedor e Gerente recebem o que antes era feito num comando condicional. Todas as classes mais específicas sobrescrevem a superclasse TipoDeEmpregado.

A essência do polimorfismo é que, em vez de perguntar a um objeto qual é o seu tipo e então chamar algum comportamento baseado nessa resposta, você simplesmente chama o comportamento. O objeto, dependendo do seu tipo, faz a coisa certa.

Conclusão

Podemos notar a importância do Polimorfismo para a redução de código, simplicidade, flexibilidade, etc. O polimorfismo é utilizado em diversas refatorações e muitos Padrões de Projetos, portanto entendê-lo é fundamental para qualquer desenvolvedor.