**UNIVERSIDADE DE SÃO PAULO**

**FACULDADE DE MEDICINA DE RIBEIRÃO PRETO**

# ABSTRACT FACTORY

## Padrão de Projeto - Criação

**Disciplina: Projeto de Software**

**LETÍCIA SILVA BARBOSA**

**NOILSON OLIVEIRA**

**Ribeirão Preto**
**2020**

**Apresentaremos neste trabalho o padrão de projeto Abstract Factory que é um padrão do tipo criação.**

## INTENÇÃO:

O Abstract Factory possui a intenção de fornecer uma interface para a criação de famílias de objetos relacionados ou dependentes sem especificar suas classes concretas.

## MOTIVAÇÃO:

Considerando um conjunto de *widgets* (*toolkit*) para construção de interfaces de usuários que suporte múltiplos estilos de interação (*look-and-feel*) como o Motif e o Presentation Manager. O uso de diferentes estilos de interação definem diferentes apresentações e comportamento para os *widgets* de uma interface de usuário, como por exemplo, barras de rolamento, janelas e botões. Com o objetivo de ser portátil entre vários estilos de aparência, uma aplicação não deve codificar rigidamente seus *widgets* para um determinado padrão, uma vez que, instanciando classes específicas de estilo de interação para os *widgets* pela aplicação toda, torna difícil a mudança de estilo no futuro.

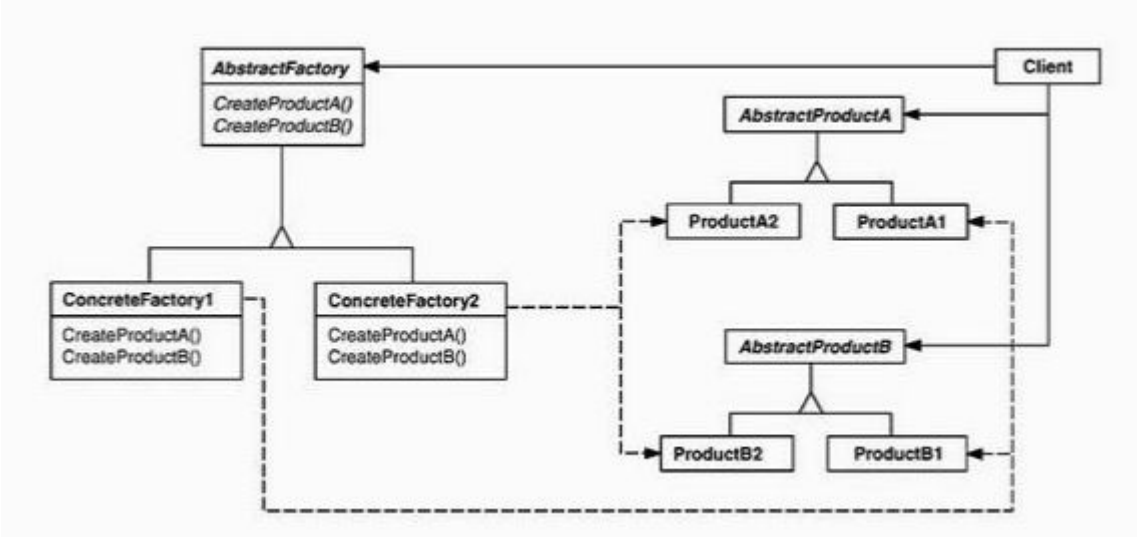
Para a solução deste problema é necessário a definição de uma classe abstrata WidgetFactory que declara uma interface para criação de cada tipo básico de *widget*. Possuindo também uma classe abstrata para cada tipo de *widget*, e subclasses concretas implementam os *widgets* para interação. Para retornar um novo objeto *widget* para cada classe abstrata de *widget* utiliza-se uma operação da interface WidgetFactory. Mesmo sem ter conhecimento das classes concretas que estão utilizando, os clientes chamam estas operações para obter instâncias de *widget*. Desta forma, os clientes não ficam dependentes do padrão de interação usado no momento.

Para cada estilo de interação vai existir uma subclasse concreta de WidgetFactory. Essas subclasses vão implementar as operações de criar o *widget* apropriado para o estilo de interação. Os clientes criam *widgets* exclusivamente através da interface de WidgetFactory e não possuem conhecimento das classes que implementam os *widgets* para um padrão em particular. Ou seja, eles apenas

precisam se comprometer com uma interface definida por uma classe abstrata, não uma determinada classe concreta.

Uma WidgetFactory também implementa e garante as dependências entre as classes concretas de *widgets.*

**DIAGRAMA:**




**DESCRIÇÃO DAS CLASSES**:

### AbstractFactory ou WidgetFactory

Pode ser uma classe abstrata ou uma interface, mas a classe abstrata é utilizada com maior frequência. Seu objetivo é declarar métodos de criação de objetos do tipo AbstractProduct, que são implementados por uma classe do tipo ConcreteFactory, que estende ou implementa a AbstractFactory.

### ConcreteFactory

Estende ou implementa a AbstractFactory. O objetivo dessa classe é implementar os métodos declarados em AbstractFactory, criando um objeto do tipo ConcreteProduct e retornando-o como um AbstractProduct. É comum existir mais de

uma classe do tipo ConcreteProduct assim como ocorre com ConcreteFactory. A quantidade de classes do tipo ConcreteFactory está diretamente ligada com a quantidade de classes do tipo ConcreteProduct.

### AbstractProduct

Pode ser uma classe abstrata ou uma interface, mas a classe abstrata é utilizada com maior frequência. Produto abstrato declara os métodos que são implementados por classes do tipo ConcreteProduct . ConcreteFactory cria internamente um objeto do tipo ConcreteProduct , mas esse objeto é retornado como um AbstractProduct. O Abstract Factory não sabe qual ConcreteProduct está sendo criado, mas sabe quais métodos do produto ele pode utilizar.

### ConcreteProduct

Estende ou implementa a classe AbstractProduct. Nessa classe são implementados os métodos declarados em AbstractProduct. Essa é a classe que faz uma instância concreta ser criada. Para cada ConcreteFactory, há pelo menos um ConcreteProduct.

### Client

Usa somente as interfaces declaradas pelas classes AbstractFactory e AbstractProduct.

## APLICABILIDADE:

O padrão Abstract Factory é indicado quando:

- O sistema tem que ser independente de como seus produtos são criados, compostos ou representados.
- O sistema tem que ser configurado como um produto de uma família de múltiplos produtos.
- Uma família de objetos/produtos for projetada para ser usada em conjunto e esta restrição deve ser garantida.
- Não queremos revelar suas implementações, apenas fornecer uma biblioteca de classes de produtos e suas interfaces.

**CONSEQUÊNCIAS DO USO:**

O uso do padrão Abstract Factory tem as seguintes consequências:

- **Ele isola as classes concretas.** Ajuda a controlar as classes de objetos criados por uma aplicação. Ele isola os clientes das classes de implementação por meio do encapsulamento da responsabilidade e o processo de criar objetos/produtos. No qual, a manipulação das instâncias é realizada através das suas interfaces abstratas. Os nomes das classes/produtos não aparecem no código do cliente, eles ficam isolados na implementação da fábrica concreta.

- **Ele torna fácil a troca de famílias de produtos.** A classe de uma fábrica concreta aparece somente quando é instanciada. Isso facilita mudar a fábrica concreta que uma aplicação usa. Pode utilizar diferentes configurações de produtos simplesmente trocando a fábrica concreta. Quando a fábrica abstrata cria uma família completa de produtos, toda família de produtos muda de uma só vez. Por exemplo, podemos mudar de *widgets* do Motif para *widgets* do Presentation Manager simplesmente trocando os correspondentes objetos/fábrica e recriando a interface.

- **Ela promove a harmonia entre produtos.** Quando objetos/produtos numa determinada família são projetados para trabalharem juntos, é fundamental que uma aplicação use objetos de apenas uma família de cada vez. Com o uso da AbstractFactory torna-se fácil assegurar isso.

- **É difícil de suportar os tipos de produtos.** É difícil estender fábricas abstratas para produzir novos tipos de produtos. Visto que, a interface de AbstractFactory fixa o conjunto de produtos que podem ser criados. Em consequência disso, para suportar novos tipos de produto há a exigência de estender a interface da fábrica, o que envolve a mudança da classe Abstract Factory e todas as suas subclasses.

**IMPLEMENTAÇÃO:**

Existem algumas técnicas úteis para que o padrão Abstract Factory seja implementado:

1 - *Fábricas como Singletons*. Uma aplicação necessita apenas de uma instância de uma Concrete Factory por família de produto, assim ela é melhor implementada como um Singleton

2 - *Criando os produtos*. A classe AbstractFactory apenas declara uma interface para a criação de produtos, desse modo fica a cargo das subclasses de ConcreteProducts criar-los efetivamente. Uma maneira eficaz de fazer isso é definir um Factory Method para cada produto.
Para cada família de produtos será necessária uma nova subclasse de ConcreteFactory mesmo que possuam apenas diferenças pequenas

3 - *Definindo fábricas extensíveis*. O AbstractFactory normalmente define uma operação diferente para cada tipo de produto que pode produzir, os tipos de produtos estão codificados nas assinaturas das operações. Acrescentar um novo tipo de produto exige mudança na interface de AbstractFactory e de todas as classes dependentes.
Um projeto mais flexível é acrescentar um parâmetro as operações que criam objetos, especificando qual será o tipo do objeto a ser criado. Contudo, essa abordagem é menos segura.

**Exemplo de código:**

```
public class AbstractFactoryExample {

    public static void main(String[] args) {

        AbstractFactoryfabrica1 = new ConcreteFactory1();
        Cliente cliente1 = new Client(fabrica1);
        cliente1.executar();

        AbstractFactoryfabrica2 = new ConcreteFactory2();
        Cliente cliente2 = new Client(fabrica2);
        cliente2.executar();
    }
}


class Client {
    private AbstractProductA produtoA;
    private AbstractProductB produtoB;

    Client(AbstractFactory fabrica) {
        produtoA = fabrica.createProdutoA();
        produtoB = fabrica.createProdutoB();
    }
```

```
    void executar() {
        produtoB.interagir(produtoA);
    }
}

interface AbstractFactory{
    AbstractProductA createProdutoA();
    AbstractProductB createProdutoB();
}

interface AbstractProductA {

}

interface AbstractProductB {
    void interagir(AbstractProductA  a);
}

class ConcreteFactory1 implements AbstractFactory{

    @Override
    public AbstractProductA createProdutoA() {
        return new ProdutoA1();
    }
    @Override
    public AbstractProductB createProdutoB() {
        return new ProdutoB1();
    }
}

class ConcreteFactory2 implements AbstractFactory{

    @Override
    public AbstractProductA createProdutoA() {
        return new ProdutoA2();
    }
    @Override
    public AbstractProductB createProdutoB() {
        return new ProdutoB2();
    }
}

class ProdutoA1 implements AbstractProductA {

}

class ProdutoB1 implements AbstractProductB {
```

```
    @Override
    public void interagir(AbstractProductA  a) {
        System.out.println(this.getClass().getName() + " interage
com " + a.getClass().getName());
    }

}

class ProdutoA2 implements AbstractProductA {

}

class ProdutoB2 implements AbstractProductB {

    @Override
    public void interagir(AbstractProductA  a) {
        System.out.println(this.getClass().getName() + " interage
com " + a.getClass().getName());
    }

}
```

Saída do código:

```
run:
ProdutoB1 interage com ProdutoA1
ProdutoB2 interage com ProdutoA2
CONSTRUÍDO COM SUCESSO (tempo total: 4 segundos)
```

Neste exemplo a classe AbstractFactoryExample vai determinar o tipo concreto do objeto a ser criado. Contudo ela retorna apenas o ponteiro abstrato para o objeto concreto criado.

O Client não possui conhecimento do tipo concreto e acessam esses objetos através de sua interface abstrata.

Para adicionar novos tipos concretos pode ser feita modificando uma linha em arquivo, ou então como foi feito no código adicionando um novo AbstractFactory que cria objetos de um tipo concreto diferente, utilizando o mesmo ponteiro do anterior.

Então todos os objetos da fábrica são armazenados globalmente em um objeto Singleton, de modo que alterar as fábricas fica muito mais fácil.

## REFERÊNCIAS:

GAMMA,E. et al. **Padrões de Projeto: Soluções reutilizáveis de software orientado à objetos**. Porto Alegre: Bookman, 2000. p 95-104.