

Aplicações de Padrões de Projetos

em Java

Vinicius Takeo Friedrich Kuwaki

Universidade do Estado de Santa Catarina

Seções

Introdução

Criacionais

Factory Method

Singleton

Estruturais

Composite

Comportamentais

Observer

Strategy

Padrão DAO

Introdução

- Aqui serão apresentados alguns padrões de projeto e suas implementações na linguagem de programação Java;
- No livro de Gamma et al. (1994), os autores apresentaram 23 padrões de projetos;
- Iremos implementar apenas alguns deles aqui;
- Mas o que são Padrões de Projetos?
- De acordo com os autores do livro, são soluções genéricas para problemas comuns no desenvolvimento de software;

- São divididos em três tipos:
 - Criacionais: Dedicados à criação de objetos;
 - Estruturais: Dedicados à composição de classes e objetos;
 - Comportamentais: Dedicados à comunicação entre os objetos.

Seções

Introdução

Criacionais

Factory Method

Singleton

Estruturais

Composite

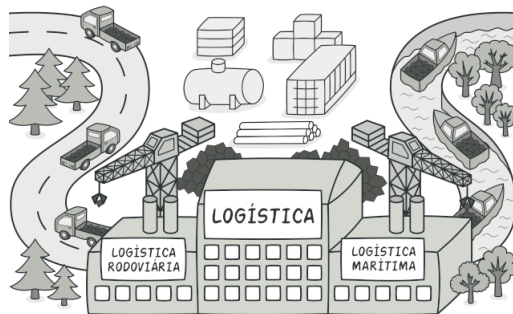
Comportamentais

Observer

Strategy

Padrão DAO

Padrão Factory Method



Fonte: Refactoring Guru

Factory Method - Definição

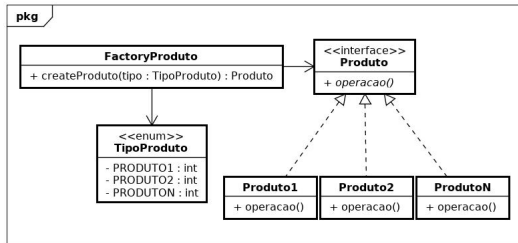
Definir uma interface para criar um objeto, deixando com que as subclasses decidam qual classe instanciar. O Factory Method deixa a classe adiar a instanciação as subclasses (GAMMA et al., 1994).

Factory Method - Objetivo

- Basicamente, o Factory Method tem como objetivo criar uma classe para instanciar objetos que descendem dessa classe ou interface;

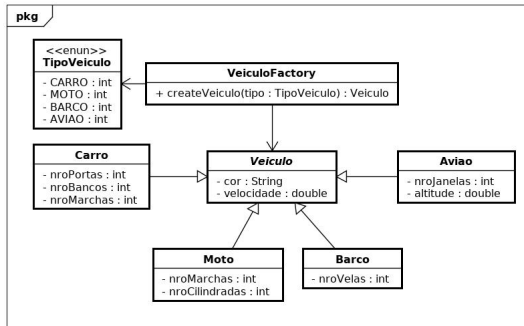
Factory Method - Estrutura

- A partir de uma classe Abstrata ou uma interface Produto;
- Teremos n classes que implementam ou estendem esse tal Produto;
- Como sabemos exatamente o número de classes que especializam esse Produto, podemos utilizar um enum para diferenciá-las;
- Por fim, teremos uma classe Factory para o Produto que possui um método cujo objetivo é instanciar cada um dos n diferentes tipos de Produtos;
- Vamos ver como isso funciona em um exemplo.



Factory Method - Exemplo

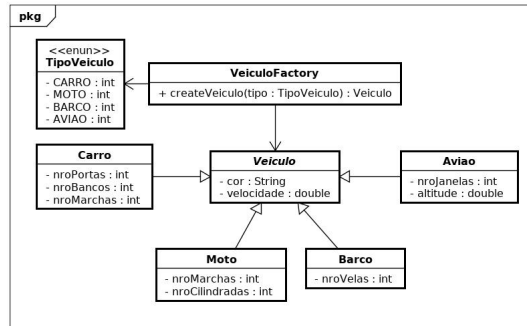
- Vamos criar um Factory Method chamado VeiculoFactory para instanciar Veículos;
- Faremos Veiculo ser uma classe abstrata (poderia ser uma interface também);
- Temos quatro tipos de Veiculo:
 - Avião;
 - Barco;
 - Carro;
 - Moto;
- Cada qual com seus atributos, getters e setters;



Factory Method - Exemplo

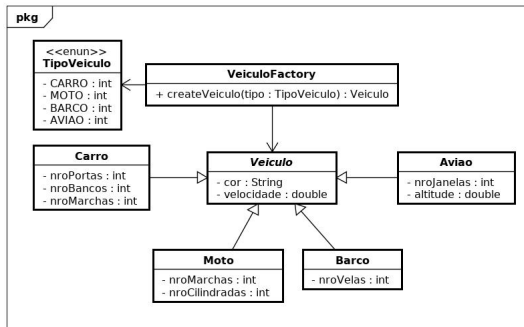
- As implementações dessas subclasses de Veiculo não serão mostradas, visto que o foco é a implementação do padrão Factory Method;
- Os códigos estão disponíveis no github;
- Precisaremos de um enum para diferenciar todas as classes que podem ser instanciadas pelo nosso Factory:

```
public enum TipoVeiculo {  
    CARRO, MOTO, BARCO, AVIAO;  
}
```



Factory Method - Exemplo

- Agora vamos implementar o `VeiculoFactory`;
- O método **`createVeiculo()`** recebe como parâmetro um valor do enum `TipoVeiculo`;
- Ele define o objeto a ser criado por meio de um switch;
- Ele retorna um novo objeto de acordo com o valor passado como parâmetro;
- Caso o valor passado não esteja no enum é lançada uma exceção *`IllegalArgumentException`*;
- Tal exceção não precisa ser tratada por quem utilizar o método!



Factory Method - Exemplo

- Acompanhe a implementação:

```
public class VeiculoFactory {  
    public Veiculo createVeiculo(TipoVeiculo tipo) {  
        switch (tipo) {  
            case AVIAO:  
                return new Aviao();  
            case BARCO:  
                return new Barco();  
            case CARRO:  
                return new Carro();  
            case MOTO:  
                return new Moto();  
            default:  
                throw new IllegalArgumentException("Tipo de veiculo inexistente");  
        }  
    }  
}
```

Factory Method - Exemplo

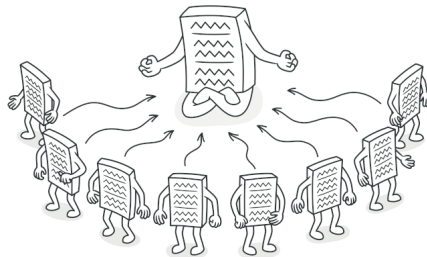
- Agora caso quisermos criar novos Veículos, delegamos tal construção a uma classe chamada VeiculoFactory;
- Veja como utilizar em um método **main()**:

```
public static void main(String [] args) {  
  
    VeiculoFactory factory = new VeiculoFactory();  
  
    Veiculo v1 = factory.createVeiculo(TipoVeiculo.CARRO);  
    Veiculo v2 = factory.createVeiculo(TipoVeiculo.BARCO);  
    Veiculo v3 = factory.createVeiculo(TipoVeiculo.MOTO);  
    Veiculo v4 = factory.createVeiculo(TipoVeiculo.AVIAO);  
  
}
```

Factory Method - Exemplo

- O padrão Factory Method é ideal para utilizar junto com o padrão Singleton que será apresentado mais a frente;
- Existem vários tipos de implementações na literatura;
- Mas todas mantêm o princípio de delegar a uma classe a decisão a respeito de qual tipo de objeto ela deve criar;

Padrão Singleton



Fonte: Refactoring Guru

Singleton - Definição

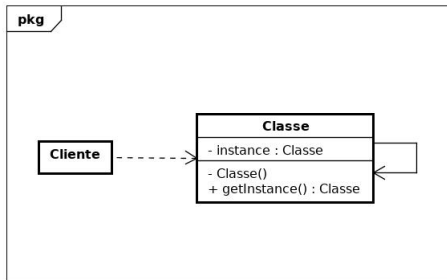
Garantir que uma classe tenha apenas uma instância e prover um ponto de acesso global a ela (GAMMA et al., 1994).

Singleton - Objetivo

- Garantir que uma classe tenha uma instância única na aplicação;
- Ainda, permitir que qualquer classe da aplicação consiga acessar essa instância única;

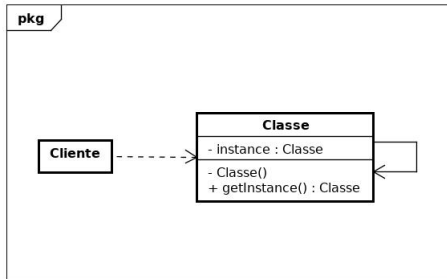
Singleton - Estrutura

- Suponha uma classe que representa algo único em um sistema;
- Essa classe presta serviços a uma ou mais classes do sistema;
- Sempre que essa classe for acessada por alguma outra, queremos que ela tenha o mesmo estado;
- Portanto, queremos que todos os clientes possam alterar o seu estado e as consequências sejam vistas por todos.



Singleton - Estrutura

- Observe que a classe singleton possui uma instância dela mesma;
- E que seu construtor é privado;
- O método **getInstance()** é o único público;
- É esse método que garante que todo objeto que queira uma instância dessa classe receba a mesma instância sempre;
- Vamos ver um exemplo melhor da sua aplicação.

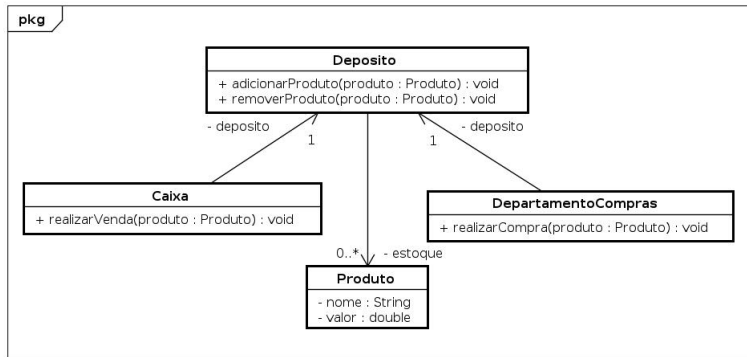


Singleton - Exemplo

- Suponha um sistema de supermercado que possua um módulo para controlar o seu depósito;
- Esse módulo pode ser acessado:
 - Pelos caixas do supermercado, que realizam vendas aos clientes e retiram produtos do depósito;
 - Pelo departamento de compras, que compra os produtos para a venda nos caixas e os armazena no depósito;
- Queremos que o sistema se comporte da seguinte forma:
 - Quando um caixa vende um produto, ele seja retirado do depósito;
 - Quando o departamento de compras adiciona um produto no depósito, qualquer caixa possa vendê-lo;

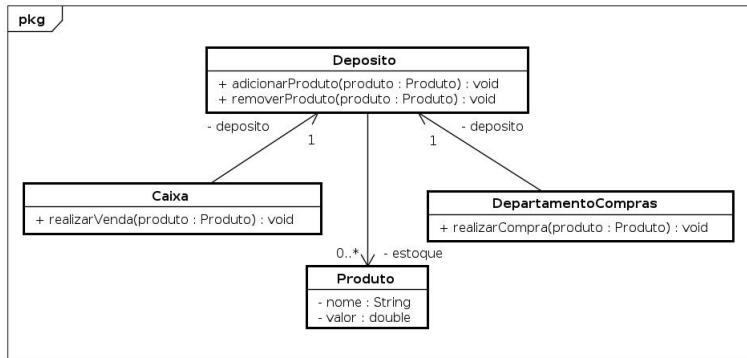
Singleton - Exemplo

- Em uma modelagem inicial desse sistema de supermercado, chegamos ao seguinte diagrama de classes:



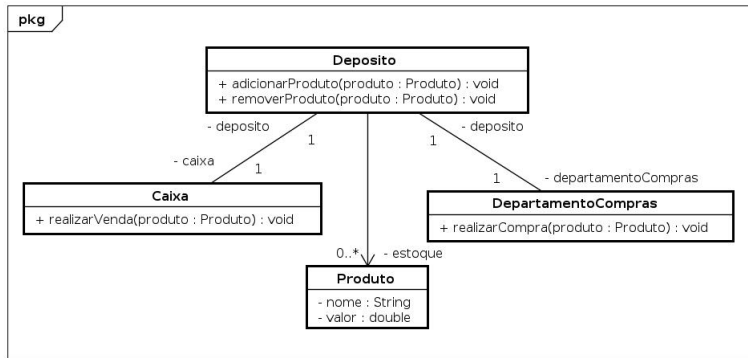
Singleton - Exemplo

- Porém, note o seguinte:
- Como o Deposito é um atributo de Caixa e de DepartamentoCompras, como ambos irão se comunicar?



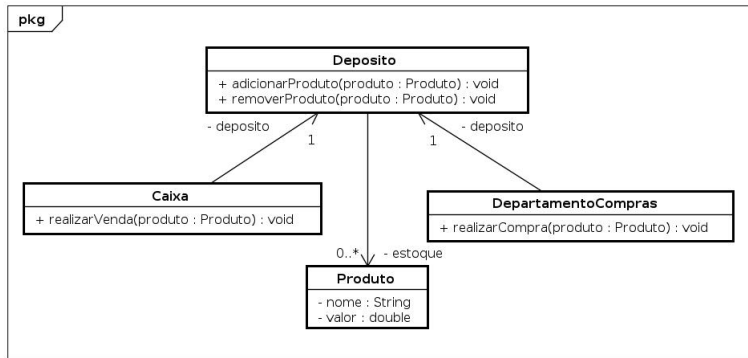
Singleton - Exemplo

- Talvez se a associação fosse sem navegação isso seria possível, assim o Caixa teria uma instância do Depósito, que teria uma instância do DepartamentoCompras e haveria apenas uma instância só de cada um, tal como no diagrama abaixo:



Singleton - Exemplo

- Essa solução não é nem um pouco prática pois, caso uma classe fosse administrar esse relacionamento triplo, ela precisaria ligar o Deposito com o Caixa, o Caixa com o Deposito, o Deposito com o DepartamentoCompras e o DepartamentoCompras com o Deposito;
- Vamos retornar a solução utilizando a associação de navegação:



Singleton - Exemplo

- Agora o **Caixa** e o **DepartamentoCompras** terão uma instância da classe **Deposito**;
- A implementação da classe Produto não será mostrada, pois ela segue a implementação normal de qualquer objeto do pacote de dados, que contém atributos e seus métodos de get() e set();
- Vamos diretamente para classe Deposito:
 - Ela vai possuir uma lista de produtos e os métodos de adicionar e remover;
 - Lançaremos uma exceção caso o produto a ser removido não esteja na lista;

Singleton - Exemplo

```
public class Deposito {  
    private List<Produto> estoque = new LinkedList<Produto>();  
  
    public void adicionarProduto(Produto produto) {  
        estoque.add(produto);  
    }  
  
    public void removerProduto(Produto produto) throws Exception {  
        if (estoque.contains(produto)) {  
            estoque.remove(produto);  
        } else {  
            throw new Exception("Produto indisponivel");  
        }  
    }  
}
```

Singleton - Exemplo

- Agora a classe Caixa, que realiza a venda do produto:

```
public class Caixa {  
  
    private Deposito deposito = new Deposito();  
  
    public void realizarVenda(Produto produto) throws Exception {  
        deposito.removerProduto(produto);  
    }  
  
}
```

- Agora a classe DepartamentoCompras, que realiza a compra de produtos para o mercado:

```
public class DepartamentoCompras {  
  
    private Deposito deposito = new Deposito();  
  
    public void realizarCompra(Produto produto) {  
        deposito.adicionarProduto(produto);  
    }  
  
}
```

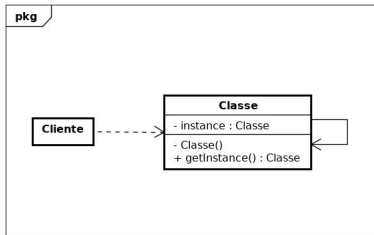
Singleton - Exemplo

- Observe que o **deposito** em ambas as classes são **objetos diferentes**, portanto, **não se comunicam**;
- Para garantir que o **caixa** e o **departamentoCompras** se comuniquem, temos que fazer com que o **deposito** adicionado a ambos seja o mesmo objeto (portanto, caixa e departamentoCompras teriam que ter um **setDeposito()**);
- Vamos aplicar o Singleton para resolver esse problema de forma mais robusta.

Singleton - Exemplo

- Na classe Deposito, vamos definir um atributo estático do mesmo tipo da classe e iniciá-lo como null;
- Vamos também deixar o construtor privado:

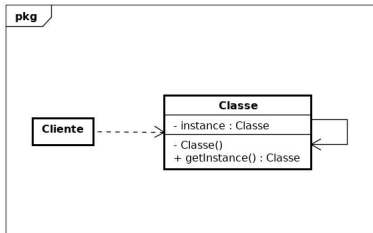
```
public class Deposito {  
    private static Deposito instance = null;  
    private Deposito() {  
    }  
}
```



Singleton - Exemplo

- Vamos criar um método chamado **getInstance()** para pegar a instância referenciada pela variável estática;
- Nesse método, caso a instância do objeto não tenha sido criada, ele chamará o construtor;
- Ao final, ele retorna sempre o valor da variável *instance*.

```
public static Deposito getInstance() {  
    if (instance == null) {  
        instance = new Deposito();  
    }  
    return instance;  
}
```



Singleton - Exemplo

- Agora que o padrão Singleton foi aplicado a classe Deposito, vamos fazer as classes Caixa e DepartamentoCompras utilizar o Singleton;
- Agora, ao invés de instanciarmos um novo deposito, vamos utilizar o método **getInstance()**;
- Note que o deposito não pode mais ser instanciado porque seu construtor é privado.

```
public class Caixa {  
  
    private Deposito deposito = Deposito.getInstance();  
  
    public void realizarVenda(Produto produto) throws Exception {  
        deposito.removerProduto(produto);  
    }  
  
}
```


Singleton - Exemplo

```
public class DepartamentoCompras {  
    private Deposito deposito = Deposito.getInstance();  
  
    public void realizarCompra(Produto produto) {  
        deposito.adicionarProduto(produto);  
    }  
}
```

- Agora as duas classes já se comunicam utilizando o mesmo objeto deposito;
- Vamos criar uma classe chamada **Principal** para realizar um teste;
- Primeiro, criaremos um método que recebe um caixa e um produto para realizar a venda;
- Como o método que realiza a venda lança uma exceção, teremos uma cláusula **catch** para capturar essa exceção e exibir a mensagem;

Singleton - Exemplo

```
public static void realizarVenda(Caixa caixa, Produto produto) {  
    try {  
        caixa.realizarVenda(produto);  
        System.out.println("Venda realizada!");  
    } catch (Exception e) {  
        System.out.println("Venda nao realizada! - " + e.getMessage());  
    }  
}
```

Singleton - Exemplo

- Agora vamos declarar o método **main()** na classe Principal que irá instanciar um departamentoCompras e três caixas, além de um produto água:

```
public static void main(String[] args) {  
    DepartamentoCompras d = new DepartamentoCompras();  
  
    Caixa c1 = new Caixa();  
    Caixa c2 = new Caixa();  
    Caixa c3 = new Caixa();  
  
    Produto agua = new Produto("Água", 1.5);  
}
```

Singleton - Exemplo

- Faremos com que o departamentoCompras compre 5 águas para serem vendidas nos caixas:

```
for (int i = 0; i < 5; i++) {  
    d.realizarCompra(agua);  
}
```

- E tentaremos vender 6 delas:

```
realizarVenda(c1, agua);  
realizarVenda(c1, agua);  
realizarVenda(c2, agua);  
realizarVenda(c3, agua);  
realizarVenda(c2, agua);  
realizarVenda(c1, agua);  
  
}
```

Singleton - Exemplo

- Terminados os métodos da classe Principal, vamos executar o código;
- Observe a saída gerada pelo console:

```
Venda realizada!  
Venda realizada!  
Venda realizada!  
Venda realizada!  
Venda realizada!  
Venda nao realizada! – Produto indisponivel!
```

Singleton - Exemplo

- Sem a aplicação do Padrão Singleton, teríamos que fazer com que o método **main()** utilizasse setters nas classes Deposito, Caixa e DepartamentoCompras para que o mesmo resultado fosse obtido;
- Singletons são muito usados em conjunto com outros padrões que necessitam de uma instância única do objeto, tal como o padrão DAO utilizado na persistência de dados;

Seções

Introdução

Criacionais

Factory Method

Singleton

Estruturais

Composite

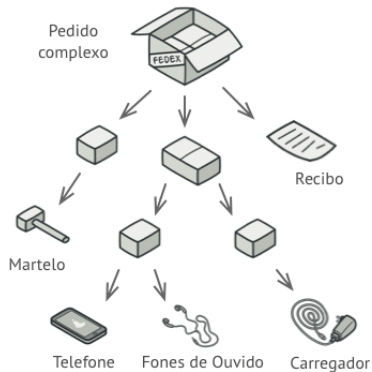
Comportamentais

Observer

Strategy

Padrão DAO

Padrão Composite



Fonte: Refactoring Guru

Composite - Definição

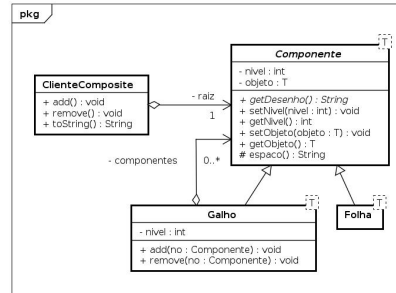
Compor objetos em estruturas de árvores representando hierárquias de parte-todo. Composite permite que o cliente trate objetos individuais e composições de objetos informalmente (GAMMA et al., 1994).

Composite - Objetivo

- O padrão Composite define uma maneira prática para representar grupos hierárquicos;
- São várias as estruturas hierárquicas presentes no nosso dia a dia:
 - Uma empresa, onde um gerente gerencia chefes que possuem subordinados;
 - Uma turma de estudantes, onde os alunos são subordinados ao professor;

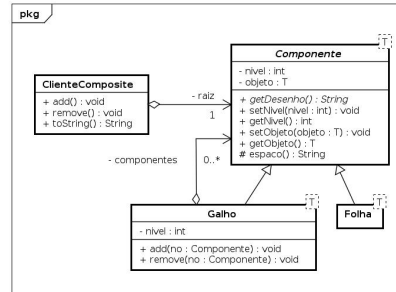
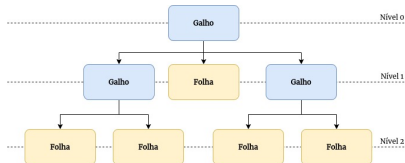
Composite - Estrutura

- No material original do GoF (GAMMA et al., 1994), não foi definido um modelo geral para o padrão;
- Entretanto, é mais conveniente criar um modelo genérico a ser seguido;
- Observe o diagrama ao lado;
- Nossa estrutura hierárquica será composta de **Componentes**;
- Esses componentes podem ser **Galhos** ou **Folhas**;



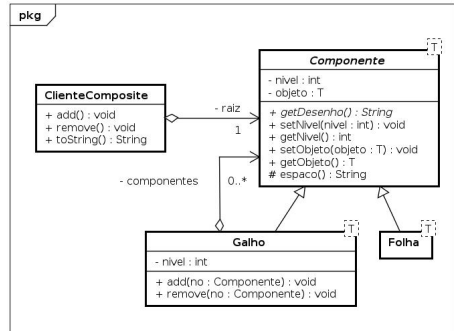
Composite - Estrutura

- Galhos contêm folhas e podem conter outros galhos;
- Já as folhas são finais, isto é, não possuem mais níveis abaixo delas;



Composite - Estrutura

- Precisaremos também de uma classe para administrar onde e como adicionar e remover esses componentes de nossa árvore;
- A classe ClienteComposite realizará esse papel;
- Vamos ver um exemplo menos abstrato desse padrão de projeto:



Composite - Exemplo

- Vamos representar a estrutura hierárquica de um seriado de televisão;
- Esse seriado terá temporadas e episódios;
- A classe *Série* terá dois atributos:
 - Nome: String;
 - Ano: int;
- A classe *Temporada* terá apenas um:
 - Número: int;
- E a classe *Episódio* terá:
 - Nome: String;
 - Número: int;
- Todas terão seus respectivos métodos **toString()**:

Composite - Exemplo

- Método **toString()** da Classe Série:

```
public String toString() {  
    return this.nome + " — " + this.ano;  
}
```

- Método **toString()** da Classe Temporada:

```
public String toString() {  
    return "Temporada " + numero;  
}
```

- Método **toString()** da Classe Episódio:

```
public String toString() {  
    return numero + " — " + nome;  
}
```

- A implementação dessas classes não será mostrada aqui, visto que são simples e não é o foco da aula;

- Vamos apenas implementar um método **equals()** para classe Temporada, pois ele será necessário na classe CompositeSerie;

```
public boolean equals(Object o) {  
    Temporada t = (Temporada) o;  
    if (t.getNumero() == numero) {  
        return true;  
    }  
    return false;  
}
```


Composite - Exemplo

- Vamos definir a classe Abstrata Componente que será implementada pelas classes Galho e Folha;
- Assim no diagrama ela terá dois atributos: nivel e objeto;
- Além de um método abstrato **getDesenho()**, que será utilizado para exibirmos a estrutura hierárquica no console, motivo pelo qual seu retorno é do tipo String;

```
public abstract class Componente<T> {  
  
    private int nivel;  
    private T objeto;  
  
    public abstract String getDesenho();  
}
```

- Os getters e setters, embora não contidos nos slides, devem ser implementados também;

Composite - Exemplo

- E por fim, vamos implementar o método **espaco()**;
- Para exibir a árvore de forma hierárquica, esse é o método mais importante;
- Pois é ele quem vai determinar a indentação na String, de acordo com o nível;
- O método retornará n identações para concatenarmos à String de retorno do método **toString()**;
- Utilizaremos a classe `StringBuilder` para concatenar os n espaços;
- Faremos isso por meio de um laço de repetição;
- O laço irá parar de acordo com o nível em que a folha se encontra;

```
protected String espaco() {  
    StringBuilder espaco = new StringBuilder();  
    for (int i = 0; i < getNivel(); i++) {  
        espaco.append("\t");  
    }  
    return espaco.toString();  
}
```

Composite - Exemplo

- Vamos implementar a classe **Folha**;
- Ela vai estender a classe abstrata Componente;
- Seu construtor recebe o objeto T e o adiciona a seu atributo *objeto*:

```
public class Folha<T> extends Componente<T> {  
    public Folha(T objeto) {  
        setObjeto(objeto);  
    }  
}
```

Composite - Exemplo

- Agora, vamos implementar o método abstrato **getDesenho()** na classe Folha;
- O método **getDesenho()** retornará o espaço concatenado com o método **toString()**;
- Para isso, ele faz a chamada do método **espaco()** da superclasse, somado a chamada do método **toString()** do próprio objeto;

```
@Override  
public String getDesenho() {  
    return espaco() + getObjeto().toString() + "\n";  
}
```

Composite - Exemplo

- Agora vamos implementar a classe **Galho**;
- Ela estende a classe abstrata **Componente** e terá uma lista de componentes, assim como especificado no diagrama anteriormente:

```
public class Galho<T> extends Componente<T> {  
    private List<Componente> componentes = new LinkedList<Componente>();
```

- Teremos um construtor vazio e um que recebe um objeto T, que seta o atributo T da superclasse, tal como no construtor da classe Folha;

```
public Galho() {  
}  
  
public Galho(T objeto) {  
    setObjeto(objeto);  
}
```

Composite - Exemplo

- Também iremos criar o método **adiciona()**;
- É ele quem irá setar o nível de seus filhos (folhas ou galhos) a cada inserção, incrementando em 1 o nível do filho, a partir do seu próprio, antes de adicioná-lo a sua lista;

```
public void adiciona(Componente componente) {  
    componente.setNivel(getNivel() + 1);  
    this.componentes.add(componente);  
}
```

Composite - Exemplo

- Como essa nossa estrutura de árvore é “pseudo-recursiva” já que imita uma estrutura recursiva, o método **getDesenho()** da classe Galho fará o percurso recursivo (indo das folhas até o nível mais alto) através da iteração de um **for**;
- Percorrendo todos os componentes da sua lista e para cada um chamando seu método **getDesenho()** e concatenando-o;

```
@Override
public String getDesenho() {

    StringBuilder desenho = new StringBuilder();
    desenho.append(espaco());
    desenho.append(getObjeto().toString() + "\n");

    for (Componente componente : componentes) {
        desenho.append(componente.getDesenho());
    }

    return desenho.toString();
}
```

Composite - Exemplo

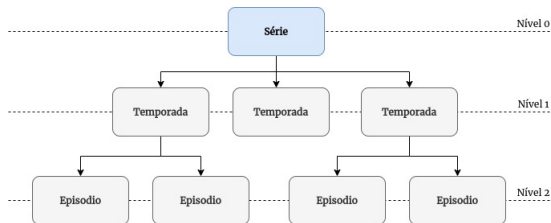
- Agora, vamos implementar a classe **CompositeSerie**, que irá administrar as inserções em nossa estrutura hierárquica;
- Essa classe terá apenas um atributo Componente do tipo Galho usando Serie como tipo genérico:

```
public class CompositeSerie {  
    private Componente<Serie> raiz = new Galho<Serie>();  
}
```


Composite - Exemplo

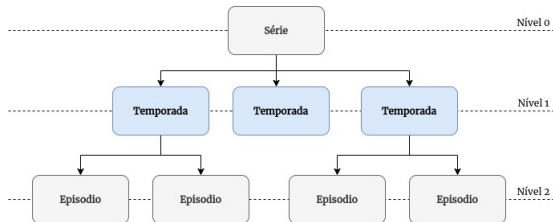
- Teremos um método **adicionarSerie()** que irá setar esse atributo, utilizando do método **setObjeto()** da classe Componente, adicionando uma Série passada como parâmetro:

```
public void adicionarSerie(Serie serie) {  
    this.raiz.setObjeto(serie);  
}
```



Composite - Exemplo

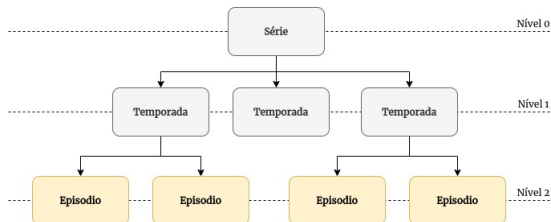
- Já o método **adicionarTemporada()** adiciona um novo objeto do tipo Galho, com a temporada passada como parâmetro;
- É necessário fazer um Casting na raiz para poder usar o método **adiciona()** da classe Galho;



```
public void adicionarTemporada(Temporada temporada) {  
    Galho<Serie> serie = (Galho<Serie>) (raiz);  
    serie.adiciona(new Galho<Temporada>(temporada));  
}
```

Composite - Exemplo

- O método **adicionarEpisodio()** como trata do último nível, precisa saber em qual dos objetos do nível anterior iremos adicionar;
- Para isso teremos que percorrer todos os componentes do nível das temporadas e encontrar onde o episódio deve ser adicionado;
- Como a Temporada é nosso último nível, ela será uma Folha.



Composite - Exemplo

- Como esse algoritmo possui alguns detalhes, vamos passo a passo;
- Primeiro, assim como no método anterior precisamos fazer um Casting na raiz, para acessar o método **getComponentes()** que existe na classe Galho;
- É sob os componentes que vamos iterar e procurar a temporada na qual vamos adicionar o episódio;

```
public void adicionarEpisodio(Episodio episodio , Temporada temporada) {  
    Galho<Serie> serie = (Galho<Serie>) (raiz);  
  
    ...  
}
```

Composite - Exemplo

- Agora vamos iterar sob esses componentes;
- Verificando se o objeto dentro dele é a temporada que queremos;

```
public void adicionarEpisodio(Episodio episodio , Temporada temporada) {  
    Galho<Serie> serie = (Galho<Serie>) (raiz);  
    for (Componente componente : serie.getComponentes()) {  
        if (componente.getObjeto().equals(temporada)) {  
            ...  
        }  
    }  
}
```

Composite - Exemplo

- Se for a temporada que buscamos, vamos fazer um Casting nela para acessarmos o método **adiciona()**:
- E adicionaremos uma nova Folha contendo um Episodio;

```
public void adicionarEpisodio(Episodio episodio , Temporada temporada) {  
    Galho<Serie> serie = (Galho<Serie>) (raiz);  
    for (Componente componente : serie.getComponentes()) {  
        if (componente.getObjeto().equals(temporada)) {  
            Galho<Temporada> aux = (Galho<Temporada>) componente;  
            aux.adiciona(new Folha<Episodio>(episodio));  
        }  
    }  
}
```

- O método **toString()** apenas retorna a chamada do método **getDesenho()** do nosso atributo **série**;

```
public String toString() {  
    return raiz.getDesenho();  
}
```

- Com isso, finalizamos a implementação da classe **CompositeSerie**;

Composite - Exemplo

- Vamos criar uma classe qualquer contendo um método **main()** para testar;
- Criaremos uma nova instância da classe CompositeSerie, uma do tipo Serie e três do tipo Temporada;

```
CompositeSerie serie = new CompositeSerie();  
  
Serie s = new Serie("Friends", 1994);  
  
Temporada t1 = new Temporada(1);  
Temporada t2 = new Temporada(2);  
Temporada t3 = new Temporada(3);
```


Composite - Exemplo

- Vamos também criar 10 episódios, três para cada uma das duas primeiras temporadas (no construtor da classe Episodio indicamos o número do episodio) e quatro para a terceira;

```
Episodio e1 = new Episodio("The Pilot", 1);  
Episodio e2 = new Episodio("The One with the Sonogram at the End", 2);  
Episodio e3 = new Episodio("The One with George Stephanopoulos", 3);  
  
Episodio e4 = new Episodio("The One with Ross's New Girlfriend", 1);  
Episodio e5 = new Episodio("The One with the Breast Milk", 2);  
Episodio e6 = new Episodio("The One Where Heckles Dies", 3);  
  
Episodio e7 = new Episodio("The One with the Princess Leia Fantasy", 1);  
Episodio e8 = new Episodio("The One Where No One's Ready", 2);  
Episodio e9 = new Episodio("The One with the Jam", 3);  
Episodio e10 = new Episodio("The One at the Beach", 25);
```

- Note que o episódio *The One at the Beach* não é o 4º episódio da 3ª temporada, mas sim o 25º.

Composite - Exemplo

- Por fim, vamos adicionar esses objetos dentro do nosso padrão de projetos:

```
serie.adicionarSerie(s);  
  
serie.adicionarTemporada(t1);  
serie.adicionarEpisodio(e1, t1);  
serie.adicionarEpisodio(e2, t1);  
serie.adicionarEpisodio(e3, t1);  
  
serie.adicionarTemporada(t2);  
serie.adicionarEpisodio(e4, t2);  
serie.adicionarEpisodio(e5, t2);  
serie.adicionarEpisodio(e6, t2);  
  
serie.adicionarTemporada(t3);  
serie.adicionarEpisodio(e7, t3);  
serie.adicionarEpisodio(e8, t3);  
serie.adicionarEpisodio(e9, t3);  
serie.adicionarEpisodio(e10, t3);
```

- Chamando o método **toString()** da classe CompositeSerie, iremos chamar o método **getDesenho()** do primeiro elemento da estrutura:

```
System.out.println(serie);
```

Composite - Exemplo

- A seguinte saída é gerada:

Friends — 1994

Temporada 1

- 1 — The Pilot
- 2 — The One with the Sonogram at the End
- 3 — The One with George Stephanopoulos

Temporada 2

- 1 — The One with Ross's New Girlfriend
- 2 — The One with the Breast Milk
- 3 — The One Where Heckles Dies

Temporada 3

- 1 — The One with the Princess Leia Fantasy
- 2 — The One Where No One's Ready
- 3 — The One with the Jam
- 25 — The One at the Beach

Seções

Introdução

Criacionais

Factory Method

Singleton

Estruturais

Composite

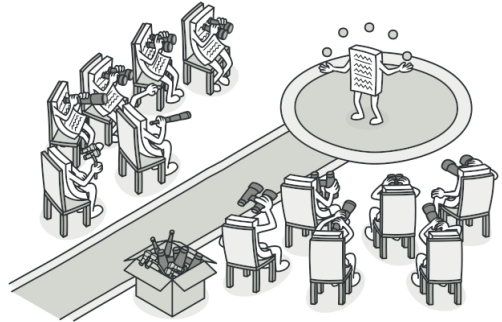
Comportamentais

Observer

Strategy

Padrão DAO

Padrão Observer



Fonte: Refactoring Guru

Observer - Definição

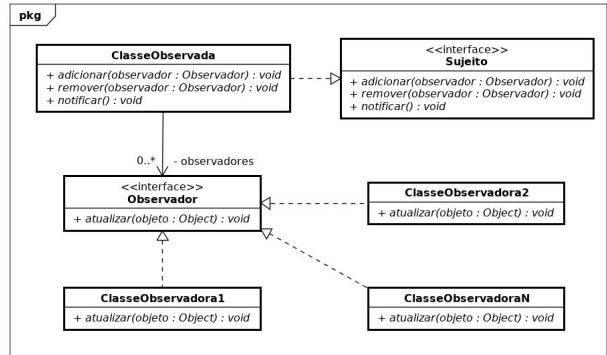
Definir uma dependência de um para muitos entre objetos para que quando o objeto mude de estado, todos os seus dependentes são notificados e atualizados automaticamente (GAMMA et al., 1994).

Observer - Objetivo

- O padrão observer define interfaces a serem seguidas para observadores “observarem” um determinado sujeito;
- Esse sujeito pode gerar, obter ou repassar informações para os observadores;
- Os observadores por sua vez podem processar essa informação, repassá-la, exibi-la, etc.

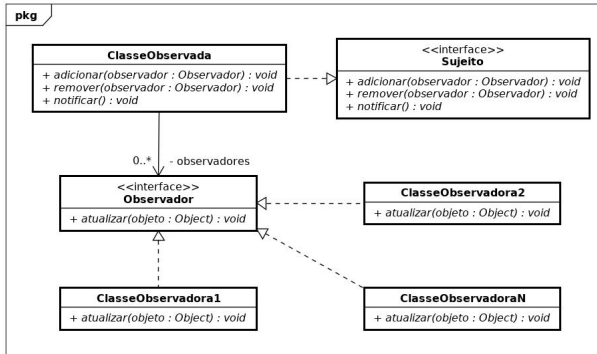
Observer - Estrutura

- Imagine uma classe que emite informações a outras;
- Um determinado sujeito, “notifica” seus observadores o que está acontecendo;
- Os observadores por sua vez, são “atualizados” pelo sujeito;



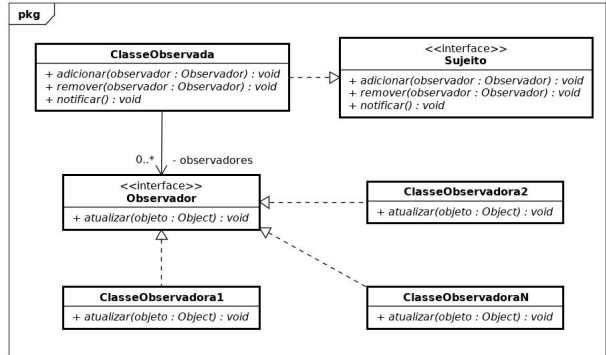
Observer - Estrutura

- O sujeito é uma interface que pede a implementação dos seguintes métodos:
 - **adicionar()**: cadastra um observador ao sujeito;
 - **remover()**: remove um observador;
 - **notificar()**: transmite a informação para todos os observadores cadastrados utilizando o método **atualizar()** de cada observador;



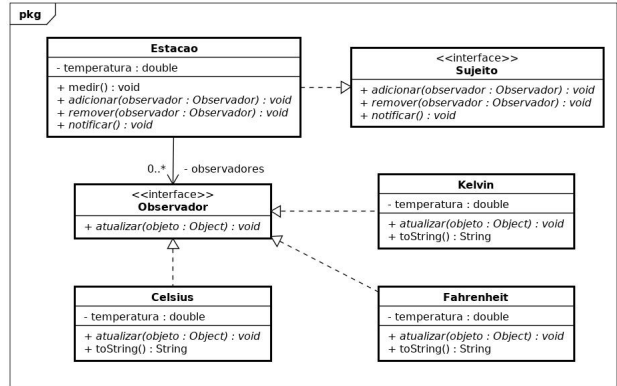
Observer - Estrutura

- Já o observador é outra interface que pede a implementação dos seguintes métodos:
 - **atualizar():** recebe as informações do sujeito. Esse método recebe um objeto, o observador deve conhecer que tipo de objeto o sujeito está enviando para realizar o devido casting;
- Vamos ver um exemplo desse padrão:



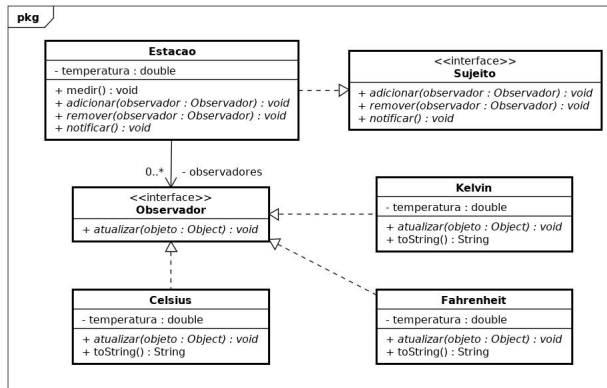
Observer - Exemplo

- Imagine uma classe que representa um estação metereológica;
- No qual três termômetros a observam;
- Cada um deles calcula a sua temperatura a partir da temperatura obtida pela estação, que é emitida em celsius;
- O observador Celsius apenas armazena a informação;



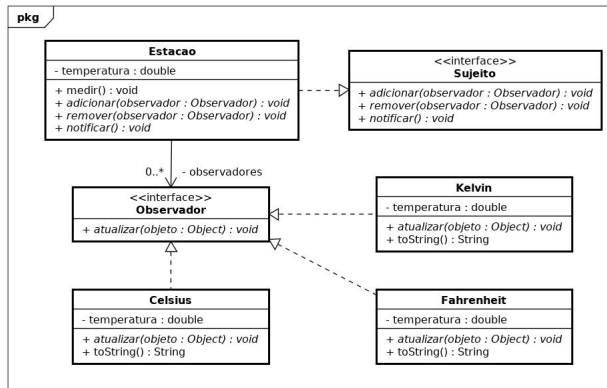
Observer - Exemplo

- Já os observadores Fahrenheit e Kelvin, utilizam uma fórmula para converterem para a suas respectivas temperaturas;
- A estação possui um método **medir()** que mede a temperatura (em graus celcius) calculando-a por meio da geração de um número aleatório e aplicando-o sobre uma fórmula para incrementá-lo ou decrementá-lo;



Observer - Exemplo

- Após realizar a medição, a estação notifica os seus observadores com os valores que foram medidos por ela;
- Cada observador realiza sua conversão dentro do método **atualizar()**;
- Os observadores também possuem um método **toString()** para poderem exibir suas devidas conversões;



Observer - Exemplo

- Vamos implementar primeiro a interface Observador;
- Ela possui apenas o método **atualizar()** que recebe um objeto;

```
public interface Observador {  
    public void atualizar(Object objeto);  
}
```

Observer - Exemplo

- Já a interface **Sujeito** possui três métodos
- Ela possui os métodos: **adicionar()** que recebe uma instância de alguma classe que implementa a interface **Observador** e a cadastra;
- O método **remover()** que remove um observador;
- E o método **notificar()** que irá notificar todos os **Observadores**;

```
public interface Sujeito {  
    public void adicionar(Observador observador);  
    public void remover(Observador observador);  
    public void notificar();  
}
```


Observer - Exemplo

- Agora criaremos a classe concreta do nosso sujeito: a Estação;
- Teremos uma List de Observadores, um double para representar a temperatura medida e uma instância da classe Random para gerar números aleatórios;

```
public class Estacao implements Sujeito {  
    private List<Observador> observadores = new LinkedList<Observador>();  
    private double temperatura;  
    private Random r = new Random();
```

- O construtor irá inicializar a variável temperatura com algum valor entre 0 e 30:

```
public Estacao() {  
    temperatura = r.nextInt(30);  
}
```

Observer - Exemplo

- Agora iremos implementar os métodos que a interface **Sujeito** pede:
- O método **adicionar()** apenas adiciona um Observador a lista;
- O método **remover()** remove um Observador;
- E o método **notificar()** itera sobre todos os Observadores chamando o método **atualizar()** de cada um, enviando as informações geradas pela estação, no caso, a temperatura;

```
public void adicionar(Observador observador) {  
    observadores.add(observador);  
}  
  
public void remover(Observador observador) {  
    observadores.remove(observador);  
}  
  
public void notificar() {  
    for (Observador o : observadores) {  
        o.atualizar(this.temperatura);  
    }  
}
```

Observer - Exemplo

- O método **medir()**, utiliza de uma fórmula para atualizar a temperatura da estação:
$$Temperatura = Temperatura + 2,5 * seno(x)$$
- Onde x será um valor aleatório;
- Tanto importa qual valor será gerado, pois essa função seno possui uma amplitude de 2.5, ou seja, o valor gerador pela função seno nunca será maior que 2.5 e nem menor que -2.5;
- Após medir, a estação notifica os Observadores;

```
public void medir() {  
    temperatura += 2.5 * Math.sin(r.nextInt(100));  
    notificar();  
}
```

Observer - Exemplo

- A classe Celsius apenas joga para seu atributo o valor da temperatura passada pelo Sujeito no método **atualizar()**;
- O método **toString()** apenas faz a formatação da temperatura para duas casas após a vírgula e retorna uma String contendo ela;

```
public class Celsius implements Observador {  
  
    private double temperatura;  
  
    public void atualizar(Object objeto) {  
        temperatura = (Double) objeto;  
    }  
  
    public String toString() {  
        return String.format("%.2f", temperatura) + " C ";  
    }  
  
}
```

Observer - Exemplo

- Já a classe Fahrenheit converte o valor recebido pela estação de celsius para fahrenheit, utilizando a fórmula:

$$F = C * 1.8 + 32$$

```
public class Fahrenheit implements Observador {  
  
    private double temperatura;  
  
    public void atualizar(Object objeto) {  
        double c = (Double) objeto;  
        temperatura = c * 1.8 + 32;  
    }  
  
    public String toString() {  
        return String.format("%.2f", temperatura) + " F ";  
    }  
}
```

Observer - Exemplo

- E a classe Kelvin converte o valor recebido pela estação de celsius para kelvin, utilizando a fórmula:

$$K = C + 273.15$$

```
public class Kelvin implements Observador {  
  
    private double temperatura;  
  
    public void atualizar(Object objeto) {  
        double c = (Double) objeto;  
        temperatura = c + 273.15;  
    }  
  
    public String toString() {  
        return String.format("%.2f", temperatura) + "K";  
    }  
}
```

Observer - Exemplo

- Agora iremos instanciar uma estação e os três termômetros em um método main:

```
public static void main(String[] args) {  
    Estacao estacao = new Estacao();  
  
    Celsius celsius = new Celsius();  
    Fahrenheit fahrenheit = new Fahrenheit();  
    Kelvin kelvin = new Kelvin();
```

- Vamos cadastrar os três termômetros como observadores da estação:

```
estacao.adicionar(celsius);  
estacao.adicionar(fahrenheit);  
estacao.adicionar(kelvin);
```

Observer - Exemplo

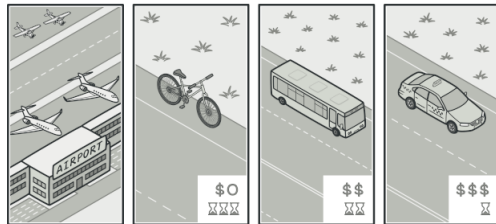
- Agora iremos fazer um laço de repetição para gerar 10 medições da estação e exibir no console os resultados obtidos pelos termômetros:

```
for (int i = 0; i < 10; i++) {  
    estacao.medir();  
  
    System.out.println("Temperatura em graus Celsius: " + celsius);  
    System.out.println("Temperatura em graus Fahrenheit: " + fahrenheit);  
    System.out.println("Temperatura em Kelvin: " + kelvin);  
    System.out.println();  
}
```


Observer - Exemplo

- Observe que a estação realizou a medição e notificou seus observadores;
- Que tiveram seus estados exibidos no console;

Padrão Strategy



Fonte: Refactoring Guru

Strategy - Definição

Definir uma família de algoritmos, encapsular cada um e torná-los intercambiáveis. Strategy deixa o algoritmo independente dos clientes que o utilizam (GAMMA et al., 1994).

Strategy - Objetivo

- O padrão Strategy separa a implementação de algoritmos das classes que o usam;
- Isso facilita a manutenção de software, testes unitários e etc.;

Strategy - Estrutura

- Imagine uma Classe que implementa alguma regra de negócio;
- Nela, um certo método implementa uma estratégia que deve ser alterada dependendo da regra de negócio;
- Vamos ver um exemplo menos teórico de um certo contexto antes e depois da aplicação do padrão:

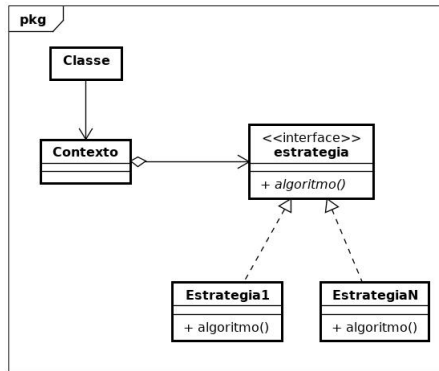
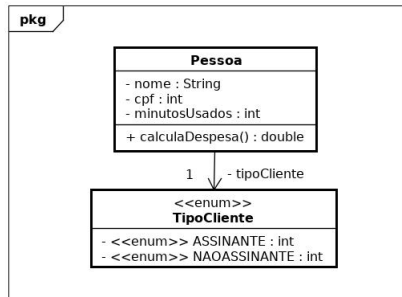


Figura 1: Padrão Strategy

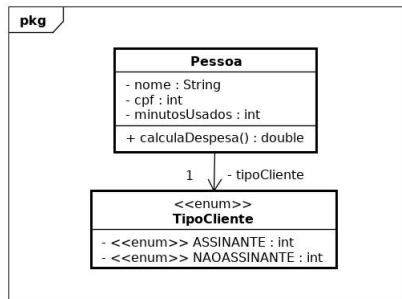
Strategy - Exemplo

- Imagine uma empresa que cobra de seus clientes por tempo de uso;
 - Assinantes de determinado serviço pagam 50 centavos por minuto de uso;
 - Já os não-assinantes pagam 70 centavos por minuto de uso;
- Vamos modelar a classe que representa essas pessoas;
- Cada Pessoa terá um nome, cpf, minutos usados e o seu tipo de cliente;
- Sendo o tipo de cliente assinante ou não-assinante;



Strategy - Exemplo

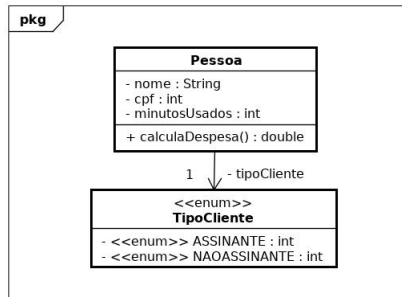
- Ainda, teremos o método `calcularDespesa()` dentro da classe `Pessoa`;
- Esse método vai variar de acordo com o tipo de cliente que essa pessoa é;
- Implementaremos aqui a regra de negócio da nossa situação;



Strategy - Exemplo

- Utilizaremos um enum para numerar esses dois tipos de clientes;

```
public enum TipoCliente {  
    ASSINANTE(1), NAOASSINANTE(2);  
  
    private int tipoCliente;  
  
    private TipoCliente(int tipoCliente) {  
        this.tipoCliente = tipoCliente;  
    }  
  
    public int getTipo() {  
        return this.tipoCliente;  
    }  
}
```

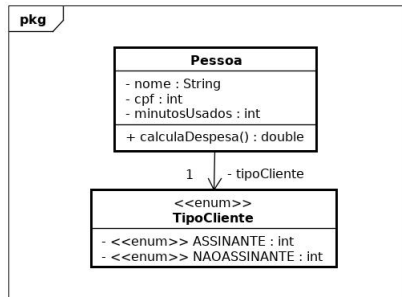


Strategy - Exemplo

- Para a classe Pessoa teremos os seguintes atributos:

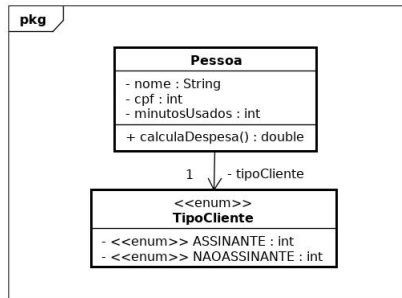
```
public class Pessoa {  
    private int cpf;  
    private String nome;  
    private TipoCliente tipoCliente;  
    private int minutosUsados;  
}
```

- A implementação dos getters e setters será omitida aqui;



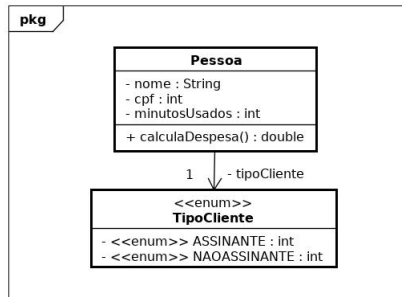
Strategy - Exemplo

- O importante aqui será a implementação do método **calculaDespesa()**;
- Para clientes assinantes, multiplicaremos o numero de minutos usados por 0.5 e retornaremos esse valor;
- Para os não-assinantes faremos o mesmo só que multiplicando por 0.7;



Strategy - Exemplo

```
public double calculaDespesa() {  
    switch (tipoCliente) {  
        case ASSINANTE:  
            return minutosUsados * 0.5;  
        case NAOASSINANTE:  
            return minutosUsados * 0.7;  
    }  
    return 0;  
}
```



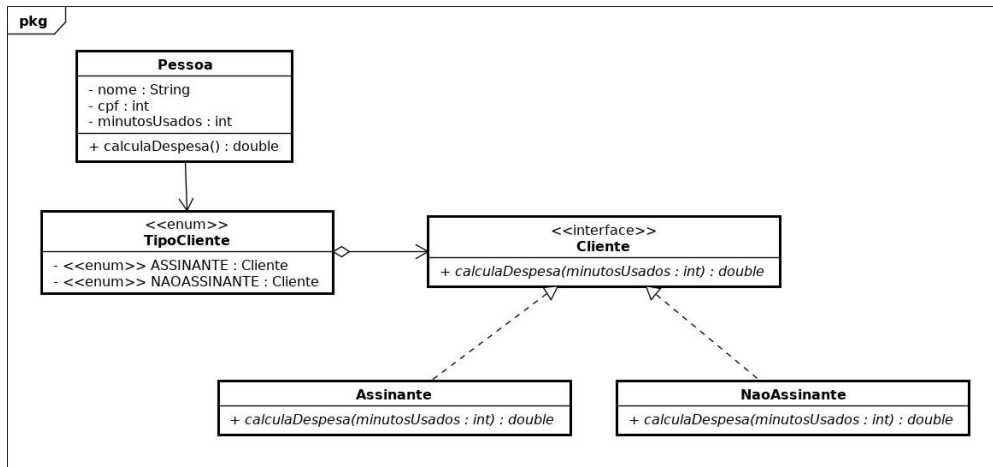
Strategy - Exemplo

- Observe nesse exemplo a dependência que causamos;
- A classe Pessoa agora sabe da regra de negócio;
- Sendo que a classe Pessoa devia apenas representar os objetos Pessoas, e não se importar com a cobrança de um determinado serviço;
- Além disso, se futuramente alterarmos a regra de negócio teremos que alterar a classe Pessoa;
- Entretanto, na verdade isso é algo que não tem nada a ver com a modelagem de uma pessoa;
- Tendo em vista esses pontos levantados, vamos aplicar o padrão Strategy em cima desse contexto;

Strategy - Exemplo

- Vamos criar uma interface Cliente;
- Essa interface irá calcular o valor da despesa, pedindo a implementação do método **calcularDespesa()** que fazia parte da Classe Pessoa;
- Teremos duas classes para implementar esse método com a sua respectiva regra de negócio:
 - Assinante;
 - NaoAssinante;
- Agora no lugar do nosso enum TipoCliente, podemos trocar a variável inteira do tipo para alguma implementação da interface Cliente através de polimorfismo;
- Observe o diagrama de classes que modela isso:

Strategy - Exemplo



Exemplo

- Na nossa interface Cliente teremos apenas a assinatura do método **calculaDespesa()**:

```
public interface Cliente {  
    public double calculaDespesa(int minutosUsados);  
}
```

Strategy - Exemplo

- Agora as classes Assinante e NaoAssinante irão realizar essa interface de acordo com a sua regra de negócio:

```
public class Assinante implements Cliente {  
    public double calculaDespesa(int minutosUsados) {  
        return minutosUsados * 0.5;  
    }  
}
```

```
public class NaoAssinante implements Cliente {  
    public double calculaDespesa(int minutosUsados) {  
        return minutosUsados * 0.7;  
    }  
}
```


Strategy - Exemplo

- O enum `TipoCliente` agora recebe no construtor algum objeto que implementa a interface `Cliente` (através de polimorfismo) ao invés de um inteiro para demarcar o seu tipo:

```
public enum TipoCliente {  
    ASSINANTE(new Assinante()), NAOASSINANTE(new NaoAssinante());  
  
    private Cliente tipo;  
  
    private TipoCliente(Cliente tipo) {  
        this.tipo = tipo;  
    }  
  
    public Cliente getTipo() {  
        return this.tipo;  
    }  
}
```

Strategy - Exemplo

- E agora na classe Pessoa, só precisamos mudar a implementação do método **calculaDespesa()**;
- Toda a implementação da regra de negócio que antes estava dentro desse método agora foi delegada pelo padrão Strategy para as implementações da interface Cliente;
- Então, como possuímos um TipoCliente nessa classe, que será uma instância de Assinante ou uma instância de NaoAssinante, basta chamarmos o método **calculaDespesa()** do enum TipoCliente:

```
public double calculaDespesa() {  
    return tipoCliente.getTipo().calculaDespesa(minutosUsados);  
}
```

Strategy - Exemplo

- Vamos instanciar em um método **main()** duas pessoas e definir o número de minutos gastos de cada uma como sendo 30;
- A primeira pessoa será um cliente assinante e a segunda será um cliente não-assinante:

```
Pessoa p1 = new Pessoa();  
p1.setNome("Pessoa 1");  
p1.setCpf(123);  
p1.setMinutosUsados(30);  
p1.setTipoCliente(TipoCliente.ASSINANTE);  
  
Pessoa p2 = new Pessoa();  
p2.setNome("Pessoa 2");  
p2.setCpf(456);  
p2.setMinutosUsados(30);  
p2.setTipoCliente(TipoCliente.NAOASSINANTE);
```

- Vamos exibir no console o valor da despesa de cada uma:

```
System.out.println(p1.calculaDespesa());  
System.out.println(p2.calculaDespesa());
```

- Observe a saída no console:

```
15.0  
20.0
```

Strategy - Exemplo

- Veja como agora a classe Cliente não se importa com a regra de negócio;
- E caso ela mude, por exemplo, agora os não-assinantes irão pagar o dobro dos assinantes, isso não será problema da classe Pessoa, mas sim do enum;
- O padrão Strategy é uma boa opção para eliminar vários `if's` e `else's` ou `switch cases's`;

Seções

Introdução

Criacionais

Factory Method

Singleton

Estruturais

Composite

Comportamentais

Observer

Strategy

Padrão DAO

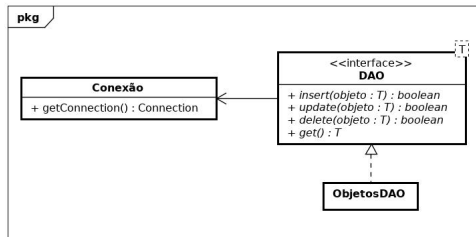
Use um Data Access Object (DAO) para abstrair e encapsular todo o acesso a fonte de dados. O DAO gerencia a conexão com a fonte de dados para obter os dados armazenados nela (ORACLE, 2002).

DAO - Objetivos

- O objetivo do DAO é promover uma interface para a persistência de dados;
- Que contenha as operações CRUD em alguma base de dados;
 - **Create;**
 - Criar novos objetos;
 - Inserir na base de dados;
 - **Read;**
 - Ler objetos;
 - Realizar consultas;
 - **Update;**
 - Atualizar objetos;
 - Alterar atributos de determinado objeto na base de dados;
 - **Delete;**
 - Remover objetos;
 - Excluir algo da base de dados;


DAO - Estrutura


- O padrão DAO provém uma interface de CRUD para objetos do tipo T;
- Uma classe qualquer, por exemplo ObjetosDAO irá implementar essa interface para um tipo de objeto em específico;
- A classe que implementar o DAO terá uma instância de uma Conexão com algum banco de dados;
- Dada essa Conexão ela fará as operações de CRUD.





DAO - Implementação

- Iremos implementar um DAO na Aula de Persistência em Banco de Dados;
- No momento ficaremos apenas com o conceito do padrão DAO;

 GAMMA, E. et al. **Design Patterns: Elements of Reusable Object-Oriented Software**. 1. ed. [S.l.]: Addison-Wesley Professional, 1994. ISBN 0201633612.

 GURU, R. **O Catálogo dos Padrões de Projeto**. <<https://refactoring.guru/pt-br/design-patterns/catalog>>.

 KUWAKI, V. T. F. Modelo de slides udesc lattex. In: . [S.l.]: Disponível em: <<https://github.com/takeofriedrich/slidesUdescLattex>>. Acesso em: 24 jan. 2020.

 MAKING, S. **Design Patterns**. <https://sourcemaking.com/design_patterns>.

 ORACLE. **Core J2EE Patterns - Data Access Object**. 2002. <<https://www.oracle.com/technetwork/java/dataaccessobject-138824.html>>.

Duvidas:
Vinicius Takeo Friedrich Kuwaki
vtkwki@gmail.com
github.com/takeofriedrich