Padrões de Projeto

1 Introdução

Comece imaginando uma empresa onde todos os sistemas são construídos de acordo com a vontade do programador. Suponha que ele tenha liberdade total para tomar qualquer tipo de decisão.

Mesmo que essa empresa tenha os melhores programadores, se for descoberto que todos os sistemas possuem um erro - ainda que seja um erro simples e igual em todos - será preciso resolvê-lo de forma diferente em cada um devido à implementação de cada sistema.

A empresa seria obrigada a gastar tempo e dinheiro para elaborar e implementar as diferentes soluções para o mesmo problema. Para evitar esse tipo de situação, foi criado o conceito de ***design pattern*** (**padrão de projeto**).

O padrão de projeto é uma **técnica de modelagem** para o sistema que apresenta uma solução consolidada para o problema comum a diversos tipos de aplicativos. Assim, a solução é criada uma única vez e aplicada em todos os aplicativos, ao invés de se criar uma solução para cada aplicativo.

Isso pode parecer um pouco formal e desanimador, mas, na verdade, padrão de projeto é uma **maneira conveniente de reutilização entre sistemas de código orientado a objetos**, bem como uma maneira de **otimizar a comunicação** e o conhecimento entre desenvolvedores através de termologia padronizada.

Essa é a razão pela qual os *Design Patterns* não se aplicam exclusivamente ao Java, podendo ser utilizados em qualquer Linguagem Orientada a Objetos, como: PHP, C++, Smalltalk, C#.

1.1 Origens

Inspirações para padrões de projetos existem desde o fim dos anos 1970, sendo sua origem atribuída a **Chistopher Alexander**, professor de arquitetura da **U.C. Berkeley**, responsável por introduzir o conceito e alguns padrões de arquiteturas de projeto.

Seus trabalhos despertaram o interesse da comunidade orientada a objetos e, nas décadas seguintes, vários padrões de design de software foram apresentados. Porém, a referência mais importante relacionada aos padrões de projetos é o livro **Design Patterns: Elements of Reusable Object-Oriented dos autores Erich Gamma, Richard Helm, Ralph Johnson e John Vlissides**, conhecidos como **"Gang of Four" (GoF)**.

Devido a sua linguagem abrangente, o livro se tornou um *best-seller* e fonte de referência sobre padrões de projeto. Este texto fará referência aos padrões descritos no livro e também apresentará outros padrões úteis, que não foram abordados pelos autores.

1.2 Definindo Design Patterns

Existem várias defiçnões para *Design Patterns* na literatura. As mais citadas são:

"Padrões de projeto são soluções recorrentes para problemas que você vê mais e mais em um projeto."
Companion Smaltalk

"Padrões de projeto constituem um conjunto de regras que descrevem como realizar determinadas tarefas no domínio do desenvolvimento de software"
Pree, 1994

"Padrões de projeto se concentra mais na reutilização de arquiteturas recorrentes do projeto, enquanto *frameworks* se concentram no design e na implementação"
Coplien e Schmidt, 1995

"Um padrão aborda um problema recorrente do projeto e apresenta uma solução para ele"
Buschamann e Meunier, et al. 1996

"Padrões identificam e especificam abstração que estão em nível acima das classes e instâncias ou componentes"
Gamma, Helm, Johnson e Vlissides, 1993

Neste sentido podemos trabalhar com a seguinte definição:

O padrão de projeto é uma **técnica de modelagem** para o sistema que apresenta uma solução consolidada para o **problema comum a diversos tipos de aplicativos**. Assim, a solução é criada uma única vez e aplicada em todos os aplicativos, ao invés de se criar uma solução para cada aplicativo.

Diferente do que pode parecer, os padrões de projeto não são apenas padrões de design para objetos; eles também lidam com a **interação entre os objetos**. Uma visão possível, no caso desses padrões, seria considerá-los como padrões de comunicação.

Certos padrões não lidam apenas com a comunicação do objeto, mas também com as estratégias de herança e contenção. É a definição de simples, mas elegantes métodos de interação, que faz muitos padrões de projeto tão importantes.

Padrões de projeto podem existir em muitos níveis: do mais baixo, para solução de problemas específicos, até ser amplamente generalizado para grandes problemas. **Existem centenas de padrões na literatura para todo tipo de problema**.

Porém, não se pode escrever um padrão de projeto a partir do nada. Esses padrões são descobertos. O processo de procurar por determinado padrão é chamado de mineração de padrões (*pattern minig*).

1.3 *Antipattern*

Da mesma forma que os design patterns descrevem as boas práticas para a resolução de problemas, foi criado o conceito de antipattern para designar as práticas pouco recomendáveis.

Um antipattern é utilizado para denominar a solução muito usada para resolver certo tipo de problema, mas que gera outros problemas, seja de manutenibilidade, flexibilidade etc. Um antipattern pode ser resultado de um desenvolvedor com pouco conhecimento ou sem experiência suficiente na resolução de determinado problema, ou ainda por causa da aplicação de um bom pattern em contexto errado.

O termo foi cunhado com base no livro Design Patterns: Elements of Reusable Object-Oriented, e foi utilizado pela primeira vez no livro Antipatterns, onde foram definidas as suas características básicas:

- Um padrão repetitivo de ações, processos ou estrutura, que inicialmente parecem ser úteis, mas que com o tempo produzem mais efeitos nocivos que benefícios;
- Existe um Design Pattern ou Refactoring conhecido que substitui o antipattern.

No livro Antipatterns, também é apresentado alguns exemplos de padrões nocivos, que podem ser encontrados em Antipatterns[1].

[1] http://www.antipatterns.com/

Por mais que, à primeira vista, utilizar um antipattern seja uma boa escolha, sempre dê preferência pela aplicação dos design patterns porque a longo prazo seus benefícios são maiores. Agora você irá conhecê-los.

1.4 Padrões GOF

Os padrões definidos no livro Design Patterns: Elements of Reusable Object-Oriented são denominados padrões GoF. Quando o livro foi lançado, eles já haviam sido aplicados em vários aplicativos e estavam em nível médio de generalidade, onde poderiam facilmente ser aplicados em diversas áreas.

Para melhor organizá-los, o GoF os dividiu em três categorias:

- **Padrões de criação**: cria os objetos para o desenvolvedor, ao invés de ser preciso instanciar os objetos diretamente.
- **Padrões estruturais**: ajuda a compor grandes estruturas de objeto, tais como interfaces complexas e dados contábeis.
- **Padrões de comportamento**: ajuda a definir a comunicação entre os objetos do sistema e o controle do fluxo em um programa complexo.

Nas próximas seções serão apresentado cada padrão separado por categoria. Os diagramas UML apresentados baseiam-se nos diagramas mostrados no livro *Design Patterns: Elements of Reusable Object-Oriented*.

2 Padrões de Criação

Todos os padrões de criação lidam com as formas de criar instâncias de objetos. Isso é importante porque o programa não deve ficar dependendo de como os objetos são criados e organizados. Em muitos casos, a natureza exata do objeto pode variar de acordo com as necessidades do programa. Isso pode causar alguns problemas na hora de criar os objetos, como:

- Definir qual a classe concreta deve ser utilizada para criar o objeto.
- Definir como os objetos devem ser criados e como eles se relacionam com os demais objetos.

Seguindo o princípio de encapsulamento, essa complexidade pode ser isolada, como demonstrado nos padrões de projeto abaixo:

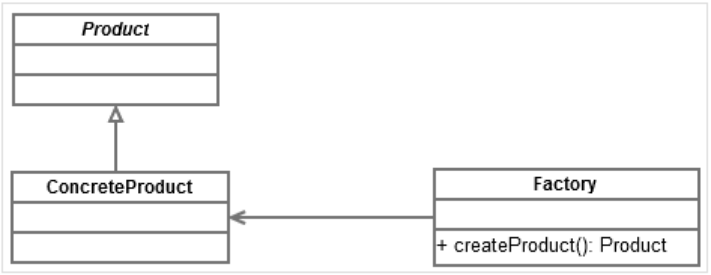
- **Simple Factory**: fornece uma classe que retorna outra classe entre as várias possíveis, de acordo com o parâmetro informado.
- **Factory Method**: define uma interface para a criação de um objeto, mas deixa as subclasses decidirem qual a classe a ser instanciada.
- **Abstract Factory**: fornece uma interface para a criação de uma família de objetos relacionados.
- **Builder**: separa a construção de um objeto complexo de sua representação, de modo que o mesmo processo possa criar representações diferentes.
- **Prototype**: permite a criação de cópias ou clones de objetos a partir de objetos existentes.
- **Singleton**: permite a criação de apenas uma única instância de uma classe e fornece uma forma de recuperá-la.

- **Multiton**: permite a criação de um número limitado de instâncias de uma classe, e fornece uma forma de recuperá-las.
- **Object Pool**: possibilita o reaproveitamento de objetos.

2.1 Padrão Factory

Não é um padrão GoF, mas é recorrente em programas orientado a objetos. Também conhecido como **Simple Factory**, esse padrão retorna uma classe, dentre a lista de possíveis, de acordo com os dados fornecidos a ele. Normalmente, todas as possíveis classes que ele retorna possuem uma classe-pai igual e métodos com a mesma assinatura, mas cada classe é otimizada para executar uma tarefa diferente. Esse padrão de **Simple Factor**y serve de introdução ao padrão **Factory Method** que será explicado à frente.

Para entender o funcionamento do padrão, veja o diagrama abaixo:



Note que há uma classe abstrata (ou interface) que serve de base para as classes derivadas. À direita está a classe *Factory*, que decide qual das subclasses irá retornar, dependendo dos argumentos que serão passados para o método.

Para exemplificar o padrão, suponha um sistema onde, dependendo do ambiente, as informações são salvas em um banco de dados diferente. Com o objetivo de padronizar a classe do banco, é implementada a interface abaixo:

```
1  public interface Banco {
2          public boolean salvar();
3          public boolean alterar();
4          public List<?> buscar();
5          public boolean excluir();
6  }
```

E, a partir dela, obtém-se o polimorfismo entre seus objetos.

Dessa forma, a classe **BancoOracle** ficará conforme o seguinte código:

```
1  public class BancoOracle implements Banco {
2          @Override
3          public boolean salvar() {
4                  //...
5                  return false;
6          }
7          @Override
8          public boolean alterar() {
9                  //...
10                 return false;
11         }
12         @Override
13         public List<?> buscar() {
14                 //...
15                 return null;
16         }
17         @Override
18         public boolean excluir() {
19                 //...
20                 return false;
21         }
22 }
```

Dessa forma, a classe **BancoMySQL** ficará conforme o seguinte código:

```
public class BancoMySQL implements Banco {
        @Override
        public boolean salvar() {
                //...
                return false;
        }
        @Override
        public boolean alterar() {
                //...
                return false;
        }
        @Override
        public List<?> buscar() {
                //...
                return null;
        }
        @Override
        public boolean excluir() {
                //...
                return false;
        }
}
```

Só assim é possível utilizar diretamente cada classe:

```
Banco banco = new BancoMySQL();
banco.salvar();
```

Porém, utilizando essa modelagem, cada novo banco de dados que for aplicado à aplicação irá requerer alteração em várias partes do sistema. Para eliminar isso, deve-se adicionar um intermediário (***Factory***) que será responsável pela escolha da classe a ser utilizada. O código ficará da seguinte forma:

```
public class BancoFactory {
        public static final int ORACLE = 0;
        public static final int MYSQL = 1;

        public Banco getBanco(int tipoBanco){
                if(tipoBanco == BancoFactory.ORACLE)
                        return new BancoOracle();
                else if(tipoBanco == BancoFactory.MYSQL)
                        return new BancoMySQL();
                else
                        throw new IllegalArgumentException("Banco não suportado");
        }
}
```

Assim, ao instanciar a classe, é informado o banco de dados utilizado:

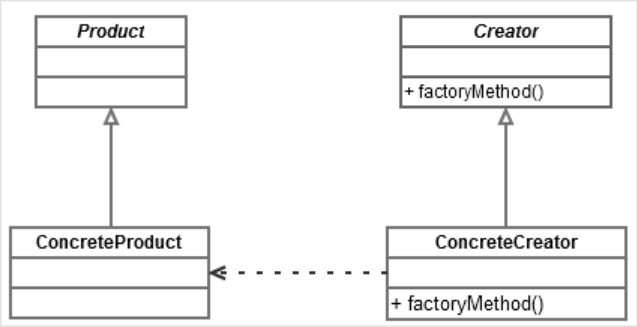
```
BancoFactory factory = new BancoFactory();
Banco banco = factory.getBanco(BancoFactory.MYSQL);
banco.salvar();
```

Com isso, ao implementar um novo banco, não será preciso se preocupar com o código existente.

2.2 Padrão Factory Method

Assim como o padrão *Simple Factory*, o padrão ***Factory Method*** fornece uma forma de **encapsular a escolha de uma classe específica** a ser utilizada na criação do objeto de determinado tipo; ou seja, objetos que possuem a mesma estrutura. O que diferencia os dois padrões é a definição de uma interface para o objeto criador, e as subclasses que decidem qual classe retornar, no **Factory Method**.

Para entender o funcionamento do padrão, veja o diagrama a seguir:



Note que há a classe abstrata *Create*, que fornecerá uma interface para a criação dos objetos, conhecida como **factoryMethod**. Todos os outros métodos, que forem definidos na classe abstrata Creator, serão criados somente para operar no Product, criado no ConcreteCreator. O Creator não cria Products; esse trabalho é feito por subclasses como ConcreteCreate.

Para exemplificar o padrão, suponha um sistema onde é necessário criar um log sobre vários tipos de operações. Para padronizar os logs, será preciso definir a interface abaixo:

```
1 public interface Logger {
2         public void log(String mensagem);
3 }
```

E, a partir dela, definir logs específicos, como para XML:

```
1 public class XMLLogger implements Logger {
2         @Override
3         public void log(String mensagem) {
4                 System.err.print("Erro em XML: " + mensagem);
5         }
6 }
```

Para a leitura de um arquivo texto:

```
1 public class ReadFileLogger implements Logger {
2         @Override
3         public void log(String mensagem) {
4                 System.err.print("Erro na leitura do arquivo: " + mensagem);
5         }
6 }
```

Para a escrita de um arquivo texto:

```
1 public class WriteFileLogger implements Logger {
2         @Override
3         public void log(String mensagem) {
4                 System.err.print("Erro na escrita do arquivo: " + mensagem);
5         }
6 }
```

E acesso ao Banco de Dados:

```
1 public class ConnectionLogger implements Logger {
2         @Override
3         public void log(String mensagem) {
4                 System.err.print("Erro na conexão com o banco de dados: " + mensagem);
5         }
6
7 }
```

Agora precisamos criar a classe abstrata **Creator**

```
1 public abstract class AbstractCreator {
2         public abstract Logger createLogger();
3         public Logger getLogger(){
4                 Logger logger = createLogger();
5                 return logger;
6         }
7
8 }
```

E um criador, para cada subclasse do **Logger**. A de XML:

```
public class XMLLoggerCreator extends AbstractCreator {
        @Override
        public Logger createLogger() {
                XMLLogger logger = new XMLLogger();
                return logger;
        }
}
```

Para a leitura de aquivo:

```
public class ReadFileLoggerCreator extends AbstractCreator {
        @Override
        public Logger createLogger() {
                ReadFileLogger logger = new ReadFileLogger();
                return logger;
        }
}
```

Para a escrita de arquivo:

```
public class WriteFileLoggerCreate extends AbstractCreator {
        @Override
        public Logger createLogger() {
                WriteFileLogger logger = new WriteFileLogger();
                return logger;
        }
}
```

E para a conexão com o Banco de Dados:

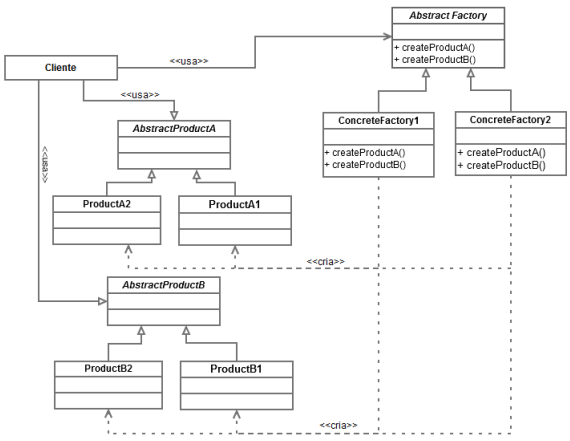
```
public class ConnectionLoggerCreate extends AbstractCreator {
        @Override
        public Logger createLogger() {
                ConnectionLogger logger = new ConnectionLogger();
                return logger;
        }
}
```

As classes podem ser utilizadas da seguinte forma:

```
AbstractCreator creator = new ConnectionLoggerCreate();

Logger logger = creator.createLogger();
logger.log("Erro no MySQL");
```

2.3 Padrão Abstract Method

É considerado um nível de abstração mais elevado do que o padrão **Factory Method**. Você pode usar esse padrão para retornar uma, das muitas classes de objetos relacionados - cada um dos quais pode retornar vários objetos diferentes. Em outras palavras, a **Abstract Method** é como uma fábrica que pode retornar diferentes tipos de objetos de vários grupos de classes. Até é possível decidir qual classe retornar usando **Simple Factory**. Veja o diagrama abaixo:



Repare no diagrama que a **Abstract Factory** trabalha como um modelo que deve ser seguido pelas demais **Factories**. Dessa forma, as fábricas podem ser reaproveitadas, reusadas ou alteradas com facilidade, pois todas terão a mesma estrutura.

Esse padrão é melhor utilizado quando o sistema tem que criar múltiplas famílias de produtos. Um exemplo clássico de uso para esse padrão é a criação de **toolkis** de interface de usuário. Por exemplo, suponha um sistema que irá funcionar em vários ambientes (sabendo que cada sistema operacional possui componentes específicos para definir a interface como janelas, botões, campos de textos etc.); então, para criar uma aplicação independente de plataforma, pode ser usada a **Abstract Factory**.

Primeiro é preciso definir uma interface, que irá padronizar todas as janelas da aplicação:

```
public interface Janela {
        public void setTitle(String title);
        public void print();
}
```

Com ela, são implementadas janelas para o Windows:

```
public class MSJanela implements Janela {
        @Override
        public void setTitle(String title) {
                // lógica para o ambiente Windows
        }
        @Override
        public void print() {
                // lógica para o ambiente Windows
        }
}
```

E MAC OS:

```
public class MacJanela implements Janela {
        @Override
        public void setTitle(String title) {
                // lógica para o ambienta Mac OS X
        }
        @Override
        public void print() {
                // lógica para o ambienta Mac OS X
        }
}
```

Também criar uma interface que irá padronizar a estrutura de todos os botões da aplicação:

```
public interface Botao {
        public void setText(String text);
        public void setClick();
}
```

E definir os botões para o ambiente Windows:

```
public class MSBotao implements Botao {
        @Override
        public void setText(String text) {
                // lógica para o ambiente Windows
        }
        @Override
        public void setClick() {
                // lógica para o ambiente Windows
        }
}
```

E MAC OS:

```
public class MacOSBotao implements Botao {
        @Override
        public void setText(String text) {
                // lógica para o Mac OSX
        }
        @Override
        public void setClick() {
                // lógica para o Mac OSX
        }
}
```

Agora é preciso criar as fábricas. Primeiro, para garantir que todas as **Factories** do sistema tenham o mesmo padrão, deve-se definir a **AbstractFactory**:

```
public interface AbstractFactory {
        public Janela createJanela();
        public Botao createBotao();
}
```

Agora é preciso criar **Factories** para os dois sistemas operacionais. Primeiro para o Windows:

```
public class MSFactory implements AbstractFactory {

        @Override
        public Janela createJanela() {
                MSJanela janela = new MSJanela();
                return janela;
        }

        @Override
        public Botao createBotao() {
                MSBotao botao = new MSBotao();
                return botao;
        }
}
```

Depois MAC OS:

```
public class MacOSFactory implements AbstractFactory {

        @Override
        public Janela createJanela() {
                MacJanela janela = new MacJanela();
                return janela;
        }

        @Override
        public Botao createBotao() {
                MacOSBotao botao = new MacOSBotao();
                return botao;
        }

}
```

Essas classes podem ser usadas da seguinte forma:

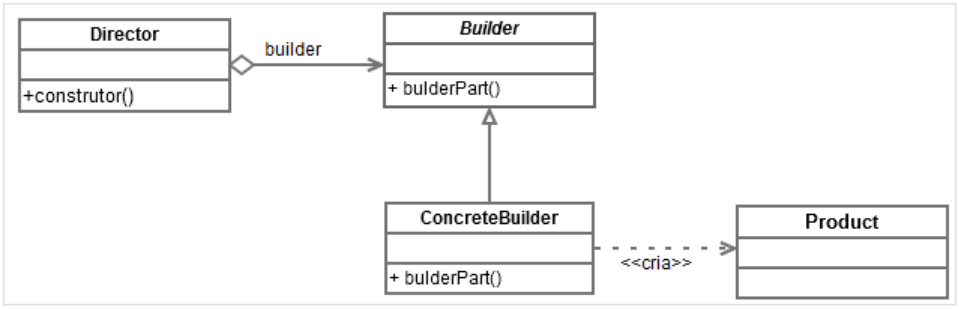
```
AbstractFactory factory = null;

if(Platform.currentPlatform() == "MACOSX"){
        factory = new MacOSFactory();
}
else{
        factory = new MSFactory();
}

Janela janela = factory.createJanela();
janela.setTitle("Nova janela!");
Botao botao = factory.createBotao();
botao.setText("OK!");
```

Assim a aplicação do código se torna independente da fábrica utilizada.

2.4 Padrão Builder

O padrão **Factory Method** retorna uma subclasse dentre as disponíveis, dependendo do argumento passado. Mas, supondo que não se queira apenas um algoritmo, mas uma representação totalmente diferente, dependendo do que é necessário apresentar, aplica-se o **padrão Builder**. Ele separa o processo de construção de um objeto, da sua definição, permitindo que objetos com implementações distintas sejam criados com a mesma representação.

Note, no diagrama abaixo, que há um diretor (**Director**) responsável por construir um **Builder** específico (**ConcreteBuilder**), de acordo com a estrutura que for especificada no **Builder**.



Para exemplificar o uso do padrão, suponha um sistema de uma fabricante de veículos. Os veículos terão uma estrutura padrão, mas cada veículo pode ter regras diferentes para as suas estruturas. Então, para encapsular a complexidade que cada veículo possa ter, as classes responsáveis pela criação dos veículos podem ser definidas de acordo com o tipo.

Para padronizar a estrutura do veículo, deve ser criada a seguinte interface:

```
1  public interface VeiculoBuilder {
2
3          public void builderEstrutura();
4          public void builderMotor();
5          public void builderRodas();
6          public void bulderPortas();
7
8          public Veiculo getVeiculo();
9  }
```

Para cada veículo a interface é implementada, e seus métodos lidam com as particularidades de cada um:

```
1  public class MotoBuilder implements VeiculoBuilder {
2          @Override
3          public void builderEstrutura() {
4                  // Regras do veículo
5          }
6          @Override
7          public void builderMotor() {
8                  // Regras do veículo
9          }
10         @Override
11         public void builderRodas() {
12                 // Regras do veículo
13         }
14         @Override
15         public void bulderPortas() {
16                 // Regras do veículo
17         }
18         @Override
19         public Veiculo getVeiculo() {
20                 // Regras do veículo
21                 return new Veiculo();
22         }
23 }
```

No caso do carro:

```
1  public class CarroBuilder implements VeiculoBuilder {
2          @Override
3          public void builderEstrutura() {
4                  // Regras do veículo
5          }
6          @Override
7          public void builderMotor() {
8                  // Regras do veículo
9          }
10         @Override
11         public void builderRodas() {
12                 // Regras do veículo
13         }
14         @Override
```

```
15          public void bulderPortas() {
16                  // Regras do veículo
17          }
18          @Override
19          public Veiculo getVeiculo() {
20                  // Regras do veículo
21                  return new Veiculo();
22          }
23  }
```

Agora, podem ser obtidas de algum local (banco de dados, arquivos, usuário etc.) as informações para criar o veículo e gerá-lo, utilizando o diretor:

```
1   public class GeradorVeiculos {
2           private VeiculoBuilder veiculoBuilder;
3
4           public GeradorVeiculos(VeiculoBuilder veiculoBuilder) {
5                   this.veiculoBuilder = veiculoBuilder;
6           }
7
8           public Veiculo gerarVeiculo(){
9
10                  this.veiculoBuilder.builderEstrutura();
11                  this.veiculoBuilder.builderMotor();
12                  this.veiculoBuilder.builderRodas();
13                  this.veiculoBuilder.bulderPortas();
14
15                  return veiculoBuilder.getVeiculo();
16          }
17  }
```

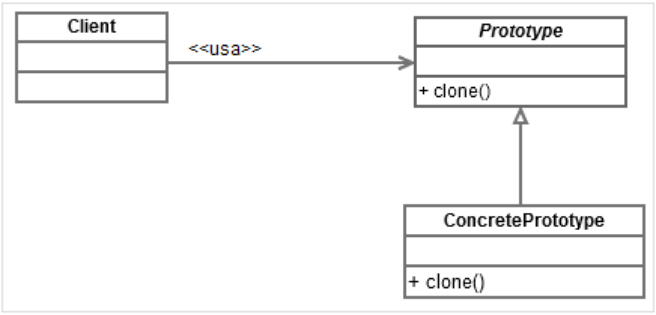
Desta forma, quando for necessário gerar um veículo, ele será escolhido de acordo com a interface **VeiculoBuilder**:

```
1   VeiculoBuilder veiculoBuilder = new CarroBuilder();
2   GeradorVeiculos gerador = new GeradorVeiculos(veiculoBuilder);
3   Veiculo veiculo = gerador.gerarVeiculo();
```

2.5 Padrão Prototype

Esse padrão possibilita a criação de novos objetos a partir da cópia de objetos existentes. É semelhante ao padrão Builder, sendo possível especi car quais componentes dos objetos existentes serão copiados para os novos objetos. O que difere, nesse padrão, é que os objetos que são criados por clonagem podem ser (e normalmente são) alterados para comportarem-se como desejado.

Para entender melhor o funcionamento, veja o diagrama abaixo:



Note que a classe abstrata **Prototype** servirá de modelo para as classes que permitem ser copiadas, e a classe **Client** pode criar novos objetos a partir das classes de nidas pela classe abstrata **Prototype**.

Para exemplificar o uso desse padrão, suponha um sistema em que seja necessário criar várias janelas com informações parecidas, mas existindo alguma informação diferente em cada uma.

Para atender essa necessidade, pode ser criada uma interface que permita a cópia de objetos:

```
public interface Prototype<T> {
        public T clone();
}
```

Ou utilizar a interface **Cloneable** do Java, que permite a implementação de clones de objetos.

Em ambos os casos, a interface deve ser implementada nas classes que possuirão a característica de cópia:

```
public class Janela implements Prototype<Janela> {
        private String titulo;
        private int altura;
        private int largura;

        public String getTitulo() {
                return titulo;
        }

        public void setTitulo(String titulo) {
                this.titulo = titulo;
        }

        public int getAltura() {
                return altura;
        }

        public void setAltura(int altura) {
                this.altura = altura;
        }

        public int getLargura() {
                return largura;
        }

        public void setLargura(int largura) {
                this.largura = largura;
        }

        @Override
                public Janela clone(){
                //lógica para criar a cópia
                return this;
        }
}
```

Como a interface **Cloneable** utiliza os recursos da API do Java, ao utilizá-la, a lógica para criar a cópia pode ser ignorada:

```
public class Janela implements Cloneable {
        private String titulo;
        private int altura;
        private int largura;

        public String getTitulo() {
                return titulo;
        }

        public void setTitulo(String titulo) {
                this.titulo = titulo;
        }

        public int getAltura() {
                return altura;
        }

        public void setAltura(int altura) {
                this.altura = altura;
        }

        public int getLargura() {
                return largura;
        }

        public void setLargura(int largura) {
                this.largura = largura;
        }
```

```
29
30          @Override
31          public Janela clone(){
32                  try {
33                          return (Janela) super.clone();
34                  } catch (CloneNotSupportedException e) {
35                          throw new IllegalArgumentException("Clone não suportado");
36                  }
37          }
38  }
```

Em ambos os casos, o usuário poderá copiar a configuração de um objeto Janela existente:

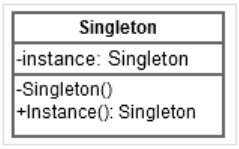
```
1  Janela janela1 = new Janela();
2  janela1.setTitulo("Janela 1");
3  janela1.setAltura(400);
4  janela1.setLargura(600);
5
6  Janela janela2 = janela1.clone();
7  janela2.setTitulo("Janela 2");
```

Dessa forma, a **janela2** terá as mesmas informações da janela1, e poderá alterar o que for preciso.

2.6 Padrão Singleton

Esse padrão é bastante popular e seu objetivo é bem simples. Ele permite a criação de uma e apenas uma única instância e fornece um modo de recuperá-la. Funcionamento

O seu funcionamento é simples:



Trata-se apenas de uma classe com um construtor privado e com o método **Instance**, que permite o acesso a uma instância da classe.

Existem muitas situações no uso das classes, onde só é necessária uma única instância durante todo o clico de vida do aplicativo. Por exemplo: suponha um sistema onde as configurações são obtidas através de um arquivo de propriedades e são armazenadas em um objeto:

```
1  public class ConfigManager {
2          private String serverName;
3
4          public ConfigManager() {
5                  // Carregamento do arquivo de propriedades
6          }
7
8          public String getServerName() {
9                  return serverName;
10         }
11
12         public void setServerName(String serverName) {
13                 this.serverName = serverName;
14         }
15 }
```

Não é desejado que no sistema exista mais de um objeto dessa classe ao mesmo tempo. Para garantir isso no Java, a maneira mais fácil é tornar o construtor privado e criar uma variável estática para armazenar uma instância. Assim você terá certeza que sempre irá existir apenas uma instância da classe. Para fornecer o acesso a essa instância, é criado o método **getInstance** com o retorno da instância:

```
1  public class ConfigManager {
2          private String serverName;
3          private static ConfigManager instance;
4
```

```
5           private ConfigManager() {
6                   // Carregamento do arquivo de propriedades
7           }
8
9           public static ConfigManager getInstance() {
10                  if(ConfigManager.instance == null)
11                          ConfigManager.instance = new ConfigManager();
12
13                  return ConfigManager.instance;
14          }
15
16          public String getServerName() {
17                  return this.serverName;
18          }
19
20
21          public void setServerName(String serverName) {
22                  this.serverName = serverName;
23          }
24  }
```

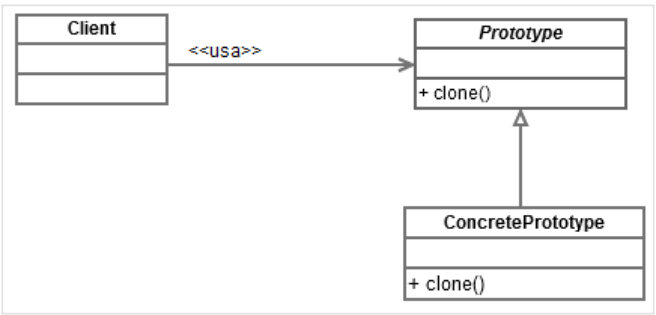
Dessa forma é possível acessar o objeto que contém as configurações do sistema, de qualquer lugar da aplicação:

```
1  String serverName = ConfigManager.getInstance().getServerName();
```

2.7 Padrão Multiton

Esse não é um padrão GoF, mas por ser muito semelhante ao **Singleton** será explicado. Ele segue o mesmo princípio de limitar o número de instâncias de uma classe na aplicação. Porém, diferente do **Singleton** que só permite uma instância, o padrão **Multiton** limita a instância a uma quantidade determinada pela classe e fornece um meio de recuperá-las.

Seu funcionamento também é muito simples:



Repare que a estrutura é muito semelhante ao padrão **Singleton**: o método construtor é privado, mas a diferença é que agora o atributo que fornece a instância da classe é uma coleção de instâncias, possibilitando a criação de um número fixo de instâncias.

Para exemplificar o padrão, suponha um sistema de segurança onde o usuário possui acesso às informações de um número limitado de câmeras. As informações das câmeras podem ser implementadas através de uma classe:

```
1  public class Camera {
2           private long hardwareId;
3           private String location;
4
5           public long getHardwareId() {
6                   return hardwareId;
7           }
8
9           public void setHardwareId(long hardwareId) {
10                  this.hardwareId = hardwareId;
11          }
12
13          public String getLocation() {
```

```
14                  return location;
15          }
16
17          public void setLocation(String location) {
18                  this.location = location;
19          }
20  }
```

Dessa forma, os usuários podem criar e acessar quantas câmeras quiser:

```
1  Camera cam1 = new Camera();
2
3  cam1.setHardwareId(1234);
4  cam1.setLocation("Left Store");
5
6  Camera cam2 = new Camera();
7
8  cam2.setHardwareId(5678);
9  cam2.setLocation("Center Store");
```

A intenção é limitar o número de câmeras criadas e poder acessá-las em qualquer ponto da aplicação, garantindo que exista apenas uma instância para cada câmera existente. O padrão **Multiton** pode ser aplicado para resolver esse problema:

```
1  public class Camera {
2          private long hardwareId;
3          private String location;
4
5          private static HashMap<Long, Camera> instance = new HashMap<Long, Camera>();
6
7          static{
8                  Camera cam1 = new Camera();
9
10                 cam1.setHardwareId(1234);
11                 cam1.setLocation("Left Store");
12
13                 Camera cam2 = new Camera();
14
15                 cam2.setHardwareId(5678);
16                 cam2.setLocation("Center Store");
17
18                 instance.put(1, cam1);
19                 instance.put(2, cam2);
20         }
21
22         private Camera(){
23
24         }
25
26         public static Camera getInstance(long hardwareId) {
27                 return Camera.instance.get(hardwareId);
28         }
29
30         public long getHardwareId() {
31                 return hardwareId;
32         }
33
34         public void setHardwareId(long hardwareId) {
35                 this.hardwareId = hardwareId;
36         }
37
38         public String getLocation() {
39                 return location;
40         }
41
42         public void setLocation(String location) {
43                 this.location = location;
44         }
45 }
```

Note que as câmeras estão sendo carregadas dentro da classe com valores fixos. Mas eles poderiam vir de outra fonte, como o banco de dados. Para acessar a informação de uma câmera, é passado o seu **hardwareId**:

```
1  Camera cam1 = Camera.getInstance(1234);
```

```
2
3  System.out.println(cam1.getLocation());
```

2.8 Padrão Object Pool

Esse padrão também não faz parte dos padrões GoF. Ele possibilita a reutilização dos objetos com o intuito de evitar a criação de instâncias em sistemas onde tal criação seja "cara".

Para entender o funcionamento veja o diagrama baixo:

Note que o construtor da classe também é privado, pois a ideia geral do padrão é que as instâncias da classe sejam reutilizadas e, para manter isso, é necessário bloquear a criações de novas instâncias fora da classe.

Dessa forma, a classe mantém uma coleção de objetos que só podem ser acessados quando o cliente utilizar o método **acquire**, que irá retornar o objeto e o removerá do **pool**. O cliente poderá utilizar esse objeto por um tempo limitado ou até devolvê-lo para o **pool**.

Esse padrão normalmente é aplicado em sistemas onde o desempenho é uma questão chave e o custo para criar objetos é muito alto.

Para exemplificar o uso do padrão, suponha um sistema onde a conexão com o banco de dados deva ser limitada. Esse limite pode ser implementado através do padrão **Object Pool**, onde primeiro deve-se criar a classe abstrata:

```
1  public abstract class ObjectPool <T> {
2          private long expirationTime;
3          private Hashtable<T, Long> locked, unlocked;
4
5          public ObjectPool() {
6                  expirationTime = 30000; // 30 segundos
7                  locked = new Hashtable<T, Long>();
8                  unlocked = new Hashtable<T, Long>();
9          }
10
11         protected abstract T create();
12
13         public abstract boolean validate(T o);
14
15         public abstract void expire(T o);
16
17         public synchronized T acquire() {
18                 long now = System.currentTimeMillis();
19                 T t;
20                 if (unlocked.size() > 0) {
21                         Enumeration<T> e = unlocked.keys();
22                         while (e.hasMoreElements()) {
23                                 t = e.nextElement();
24                                 if ((now - unlocked.get(t)) > expirationTime) {
25                                         // Objeto expirado
26                                         unlocked.remove(t);
27                                         expire(t);
28                                         t = null;
29                                 } else {
30                                         if (validate(t)) {
31                                                 unlocked.remove(t);
32                                                 locked.put(t, now);
33                                                 return (t);
34                                         } else {
35                                                 // falha da verificação
```

```
                                                unlocked.remove(t);
                                                expire(t);
                                                t = null;
                                        }
                                }
                        }
                }
                // nenhum objeto disponível, então cria um novo
                t = create();
                locked.put(t, now);
                return (t);
        }

        public synchronized void realese(T t) {
                locked.remove(t);
                unlocked.put(t, System.currentTimeMillis());
        }
}
```

A classe está diferente da estrutura padrão do Patterns Object Pool porque é preciso utilizar os recursos disponíveis em Java.

Veja que o método construtor dessa classe é público. Por ser uma classe abstrata, não pode ser instanciada diretamente. Então, os métodos construtores das classes descendentes devem ser protegidos:

```
public class ConnectionPool extends ObjectPool<Connection> {
        private String dsn;
        private String usr;
        private String pwd;

        protected ConnectionPool(String driver, String dsn, String usr, String pwd){
                super();
                try {
                        Class.forName(driver).newInstance();
                } catch (Exception e) {
                        e.printStackTrace();
                }
                this.dsn = dsn;
                this.usr = usr;
                this.pwd = pwd;
        }

        @Override
        protected Connection create() {
        try {
                        return (DriverManager.getConnection(dsn, usr, pwd));
                } catch (SQLException e) {
                        e.printStackTrace();
                        return (null);
                }
        }

        @Override
        public boolean validate(Connection o) {
                try {
                        return (!((Connection) o).isClosed());
                } catch (SQLException e) {
                        e.printStackTrace();
                        return (false);
                }
        }

        @Override
        public void expire(Connection o) {
                try {
                        ((Connection) o).close();
                } catch (SQLException e) {
                        e.printStackTrace();
                }
        }
}
```

Para usar a classe, faça da seguinte forma:

```
ConnectionPool pool = new ConnectionPool(
```

```
2  "com.mysql.jdbc.Driver", "jdbc:mysql://localhost:3306/mydb",
3  "root", "1234");
4
5  // Pegando a conexão
6  Connection con = pool.acquire();
7
8  // Usando a conexão
9  //...
10
11 // Devolvendo a conexão
12 pool.release(con);
```

3 Padrões estruturais

Um padrão estrutural descreve como as classes e objetos podem ser combinados para formarem estruturas maiores, diminuindo as dependências e complexidades entre os elementos do sistema e, consequentemente, diminuindo o custo de manutenção.

Nessa categoria estão os padrões:

- **Adapter Pattern**: permite que um objeto seja correspondente à interface de outro objeto. **Bridge Pattern**: separa a representação de um objeto da sua implementação, de forma que possa produzir e variar objetos separadamente.
- **Composite Pattern**: agrupa um conjunto de objetos que podem ser manipulados sem distinção.
- **Decorator Pattern**: permite adicionar funcionalidades aos objetos dinamicamente.
- **Façade Pattern**: cria uma interface que representa as interfaces de um subsistema.
- **Flyweight Pattern**: permite o compartilhamento de objetos de forma eficiente para economizar espaço, quando os objetos são utilizados em grande quantidade.
- **Proxy Pattern**: controla as requisições de um objeto mais complexo da mesma interface.
- **MVC Pattern**: divide a aplicação em três camadas: modelo, controle e interface.

Agora conheça cada padrão em detalhes.

3.1 Padrão Adapter

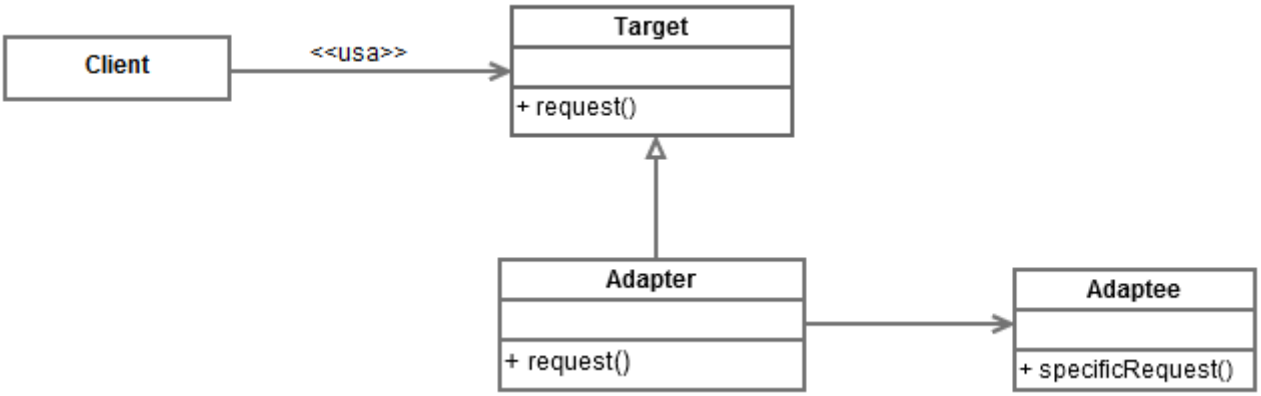
O padrão **Adapter** representa uma técnica para compatibilizar interfaces de objetos distintos. O conceito de um **adapter** é muito simples: Você escreve uma classe que tem a interface desejada e, em seguida, a faz se comunicar com a classe que tem uma interface diferente.

Pense no adapter como um adaptador de tomada, ou seja, ele permite que duas interfaces diferentes se comuniquem.

Há duas formas de criar um adapter: com herança e com composição de objetos. Na primeira, a classe que deve ser adaptada é herdada, e na classe descendente são criados os métodos necessários para permitir que a classe se comunique com a interface desejada. No segundo caso, uma classe é criada e, nela, o objeto é adaptado para se comunicar com a interface desejada. Essas abordagens são conhecidas, respectivamente, como adaptadores de classes e adaptadores de objetos.

Para entender melhor o funcionamento do padrão, veja o diagrama abaixo:

Note que o **Adapter** funciona como uma ponte entre a classe que será Adaptada (**Adaptee**) e a estrutura da classe destino (**Target**).



Para exemplificar o padrão, suponha que você irá realizar a manutenção em um sistema, cujo cadastro de funcionários está muito lento. Então, para otimizar esse cadastro, deve ser criada uma nova classe. Por exemplo, com a seguinte estrutura:

```
1  public class Employee {
2          private int nrFuncionario;
3          private String nome;
4          private String departamento;
5          private double salario;
6
7          public int getNrFuncionario() {
8                  return nrFuncionario;
9          }
10
11         public void setNrFuncionario(int nrFuncionario) {
12                 this.nrFuncionario = nrFuncionario;
13         }
14
15         public String getNome() {
16                 return nome;
17         }
18
19         public void setNome(String nome) {
20                 this.nome = nome;
21         }
22
23         public String getDepartamento() {
24                 return departamento;
25         }
26
27         public void setDepartamento(String departamento) {
28                 this.departamento = departamento;
29         }
30
31         public double getSalario() {
32                 return salario;
33         }
34
35         public void setSalario(double salario) {
36                 this.salario = salario;
37         }
38
39         public boolean create(){
40                 //implementação
41                 return true;
42         }
43
44         public boolean update(){
45                 //implementação
46                 return true;
47         }
48
49         public boolean delete(){
50                 //implementação
51                 return true;
52         }
53
```

```
54          public ResultSet search(String filter){
55                  //implementação
56                  return null;
57          }
58  }
```

Porém, a antiga classe de cadastro de funcionário possuía a seguinte estrutura:

```
1   public class Funcionario {
2           private int nrFuncionario;
3           private String nome;
4           private String departamento;
5           private double salario;
6
7           public int getNrFuncionario() {
8                   return nrFuncionario;
9           }
10
11          public void setNrFuncionario(int nrFuncionario) {
12                  this.nrFuncionario = nrFuncionario;
13          }
14
15          public String getNome() {
16                  return nome;
17          }
18
19          public void setNome(String nome) {
20                  this.nome = nome;
21          }
22
23          public String getDepartamento() {
24                  return departamento;
25          }
26
27          public void setDepartamento(String departamento) {
28                  this.departamento = departamento;
29          }
30
31          public double getSalario() {
32                  return salario;
33          }
34
35          public void setSalario(double salario) {
36                  this.salario = salario;
37          }
38
39          public boolean salvar(Funcionario f){
40                  //implementação
41                  return true;
42          }
43
44          public boolean atualizar(Funcionario f){
45                  //implementação
46                  return true;
47          }
48
49          public boolean excluir(int nrFuncionario){
50                  //implementação
51                  return true;
52          }
53
54          public ResultSet listaFuncionarios(){
55                  //implementação
56                  return null;
57          }
58
59          public ResultSet buscaFuncionariosPorNome(String nome){
60                  //implementação
61                  return null;
62          }
63
64          public ResultSet buscaFuncionario(int nrFuncionario){
65                  //implementação
66                  return null;
67          }
68  }
```

Para diminuir o impacto que a mudança da classe irá causar ao código, crie o seguinte adaptador:

```
public class NovoFuncionario extends Funcionario {
        private Employee employee;

        public NovoFuncionario(){
                if(employee == null)
                employee = new Employee();
        }

        public int getNrFuncionario() {
                return this.employee.getNrFuncionario();
        }

        public void setNrFuncionario(int nrFuncionario) {
                this.employee.setNrFuncionario(nrFuncionario);
        }

        public String getNome() {
                return this.employee.getNome();
        }

        public void setNome(String nome) {
                this.employee.setNome(nome);
        }

        public String getDepartamento() {
                return this.employee.getDepartamento();
        }

        public void setDepartamento(String departamento) {
                this.employee.setDepartamento(departamento);
        }

        public double getSalario() {
                return this.employee.getSalario();
        }

        public void setSalario(double salario) {
                this.employee.setSalario(salario);
        }

        public boolean salvar(Funcionario f){
                this.employee.setNrFuncionario(f.getNrFuncionario());
                this.employee.setNome(f.getNome());
                this.employee.setDepartamento(f.getDepartamento());
                this.employee.setSalario(f.getSalario());
                return this.employee.create();
        }

        public boolean atualizar(Funcionario f){
                this.employee.setNrFuncionario(f.getNrFuncionario());
                this.employee.setNome(f.getNome());
                this.employee.setDepartamento(f.getDepartamento());
                this.employee.setSalario(f.getSalario());
                return this.employee.update();
        }

        public boolean excluir(int nrFuncionario){
                this.employee.setNrFuncionario(nrFuncionario);
                return this.employee.delete();
        }

        public ResultSet listaFuncionarios(){
                return this.employee.search(null);
        }

        public ResultSet buscaFuncionariosPorNome(String nome){
                return this.employee.search("nome LIKE '%" + nome + "%'");
        }

        public ResultSet buscaFuncionario(int nrFuncionario){
                return this.employee.search("nrFuncionario = " + String.valueOf(nrFuncionario));
        }
}
```

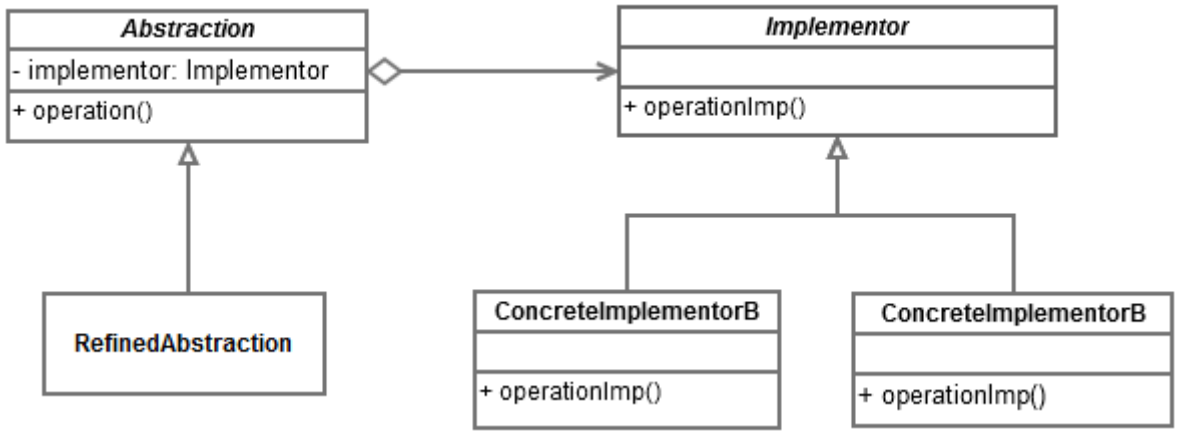
Assim, o adaptador poderá ser utilizado como se estivesse sendo utilizada a classe antiga:

```
Funcionario f = new NovoFuncionario();

f.setNome("Carlos");
```

3.2 Padrão Bridge

À primeira vista, o padrão Bridge se parece muito com o padrão Adapter, no sentido que uma classe é utilizada para converter um tipo de interface para outro. No entanto, o padrão Adapter destina-se a adaptar uma ou mais classes para uma estrutura em particular, enquanto que o padrão Bridge é projetado para separar a interface da classe, de sua implementação, para que seja possível alterar e substituir a implementação sem ter que alterar o código do cliente.

Para entender melhor o funcionamento desse padrão, veja o diagrama baixo:



Observe que a abstração e a implementação são definidas em classes separadas. Mas, diferente do que se pode pensar, onde a classe abstrata é estendida pela implementação, no padrão Bridge é considerada uma abstração a representação de um elemento com o qual o aplicativo-cliente tem interesse em interagir e a implementação e representação das principais operações que a abstração deve realizar.

Desta forma, no diagrama acima, a classe Abstraction define uma estrutura e um comportamento padrão e contém uma instância da implementação. Enquanto a classe Implementor representa as funcionalidades utilizadas pela abstração. Já a classe RefinedAbstraction estende a classe Abstraction e define um comportamento adicional. E as classes ConcreteImplementor implementam a classe abstrata Implementor.

Para exemplificar o padrão, suponha um sistema onde você precise estender as funcionalidades de uma lista – uma lista simples, bastando ter a capacidade de adicionar e remover Strings.

No padrão Bridge um elemento é definido em duas partes: uma abstração e uma implementação. A implementação é quem faz o trabalho real. No exemplo em execução, ela armazenará e recuperará os elementos da lista. O comportamento geral da lista será definido na interface abaixo:

```
public interface ListImplementor {
        public void addItem(String item);
        public void addItem(String item, int position);
        public void removeItem(String item);
        public int getNumberOfItems();
        public String getItem(int index);
        public boolean isSupportsOrdering();
}
```

Com a interface criada, é possível implementar tipos específicos de lista, como uma lista ordenada:

```
public class OrderedListImpl implements ListImplementor {
        private ArrayList<String> items = new ArrayList<String>();

        @Override
```

```
5           public void addItem(String item) {
6                   if(!items.contains(item))
7                   items.add(item);
8           }
9
10          @Override
11          public void addItem(String item, int position) {
12                  if(!items.contains(item))
13                  items.add(position, item);
14          }
15
16          @Override
17          public void removeItem(String item) {
18                  if(items.contains(item))
19                  items.remove(items.indexOf(item));
20          }
21
22          @Override
23          public int getNumberOfItems() {
24                  return items.size();
25          }
26
27          @Override
28          public String getItem(int index) {
29                  if(index < items.size())
30                          return items.get(index);
31                  return null;
32          }
33
34          @Override
35          public boolean isSupportsOrdering() {
36                  return true;
37          }
38  }
```

Já a abstração representa as operações da lista para o sistema. Inclusive, é possível definir uma interface para padronizar as classes abstratas:

```
1   public interface ListAbstraction {
2           public void setImplementor(ListImplementor impl);
3           public void add(String item);
4           public void add(String item, int position);
5           public void remove(String item);
6           public String get(int index);
7           public int count();
8   }
```

Pode ser implementada da seguinte forma:

```
1   public class BaseList implements ListAbstraction {
2           private ListImplementor implementor;
3
4           @Override
5           public void setImplementor(ListImplementor impl) {
6                   this.implementor = impl;
7           }
8
9           @Override
10          public void add(String item) {
11                  this.implementor.addItem(item);
12          }
13
14          @Override
15          public void add(String item, int position) {
16                  if(this.implementor.isSupportsOrdering())
17                          this.implementor.addItem(item, position);
18          }
19
20          @Override
21          public void remove(String item) {
22                  this.implementor.removeItem(item);
23          }
24
25          @Override
26          public String get(int index) {
27                  return this.implementor.getItem(index);
```

```
28          }
29
30          @Override
31          public int count() {
32                  return this.implementor.getNumberOfItems();
33          }
34  }
```

Repare que todas as operações são delegadas para o objeto da interface ListImplementor, que representa a implementação da lista. Dessa mesma forma, os recursos dessa classe podem ser estendidos ou implementar uma nova classe com base na interface.

Para utilizar essas classes, deve-se fazer da seguinte forma:

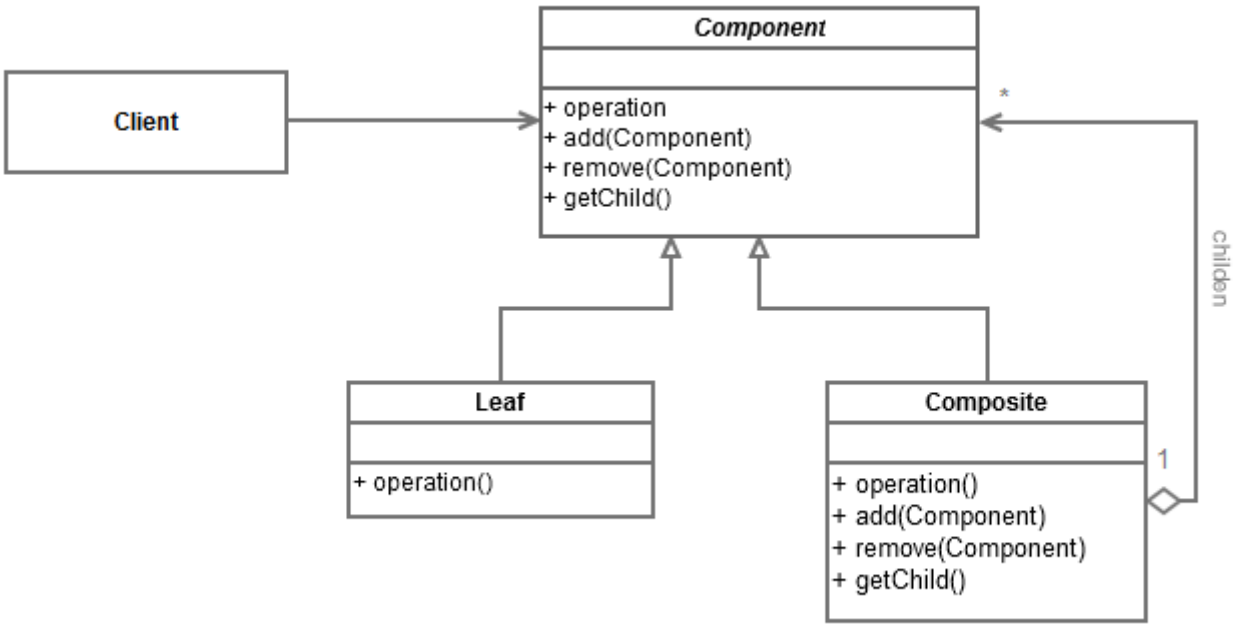
```
1  ListImplementor list = new OrderedListImpl();
2
3  ListAbstraction baseList = new BaseList();
4
5  baseList.setImplementor(list);
6  baseList.add("Treinaweb");
```

3.3 Padrão Composite

Composite Pattern é um padrão clássico de estrutura de objetos. Seu objetivo é agrupar objetos que fazem parte de uma relação parte-todo como, por exemplo, o sistema de diretórios e arquivos. Mas qualquer objeto presente no Composite pode ser tratado com um objeto primitivo.

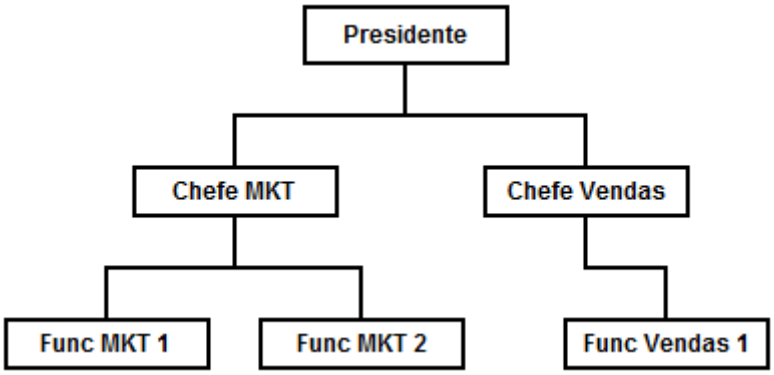
Tipicamente o Composite trabalha com sistemas de nós que são chamados de árvore. Em uma árvore de objetos é possível ter nós do tipo galho, que poderão conter outros nós; e nós do tipo folha, que não podem conter outros nós.

Para entender o funcionamento do padrão, veja o diagrama abaixo:



Observe que a classe abstrata Component irá definir os métodos disponíveis para todas as partes da estrutura da árvore. Nessa estrutura, a classe Composite contém métodos que permitem adicionar e remover instâncias de componentes da sua coleção. As classes adicionadas podem ser outro Composite ou uma Leaf. Como mostra no diagrama, a classe Leaf também implementa a classe Component, mas diferente da classe Composite, ela não contém referências de outros componentes. Uma Leaf representa o nível mais baixo em uma estrutura.

Para exemplificar o padrão, suponha um sistema em que seja necessário mapear a hierarquia de empregados de uma empresa. Naturalmente, no topo da lista vem o presidente, em seguida os chefes de departamento e, por fim, os funcionários. Seria criada uma estrutura como o diagrama abaixo:



Para criar essa estrutura, primeiro você deve definir uma classe abstrata para representar o componente, um funcionário:

```
1  public abstract class AbstractEmployee {
2          protected String nome;
3          protected long salario;
4          protected boolean leaf = true;
5          public abstract long getSalario();
6          public abstract String getNome();
7          public abstract boolean add(AbstractEmployee e)
8          throws NoSuchElementException;
9          public abstract void remove(AbstractEmployee e)
10         throws NoSuchElementException;
11         public abstract List<AbstractEmployee> subordinados();
12         public abstract AbstractEmployee getChild(String s);
13         public abstract long getSalarios();
14         public boolean isLeaf() {
15                 return leaf;
16         }
17 }
```

Um funcionário que não possua subordinado, terá esta estrutura:

```
1  public class Employee extends AbstractEmployee {
2
3          public Employee(String nome, long salario){
4                  this.nome = nome;
5                  this.salario = salario;
6          }
7
8          @Override
9          public long getSalario() {
10                 return salario;
11         }
12
13         @Override
14         public String getNome() {
15                 return nome;
16         }
17
18         @Override
19         public boolean add(AbstractEmployee e) throws NoSuchElementException {
20                 throw new NoSuchElementException("Employee não pode adicionar outro compomente");
21         }
22
23         @Override
24         public void remove(AbstractEmployee e) throws NoSuchElementException {
25                 throw new NoSuchElementException("Employee não pode remover outro compomente");
26         }
27
28         @Override
29         public List<AbstractEmployee> subordinados() {
30                 return null;
```

```
31          }
32
33          @Override
34          public AbstractEmployee getChild(String s) {
35                  return null;
36          }
37
38          @Override
39          public long getSalarios() {
40                  return salario;
41          }
42  }
```

Note que, como esse componente é uma folha, se os métodos add e remove forem utilizados será retornado um erro, e os métodos getChild e subordinados retornam nulo.

No caso de um chefe, a classe terá a seguinte estrutura:

```
1   public class Boss extends AbstractEmployee {
2
3   private List<AbstractEmployee> subordinados = new ArrayList<AbstractEmployee>();
4
5           public Boss(String nome, long salario){
6                   this.nome = nome;
7                   this.salario = salario;
8                   this.leaf = false;
9           }
10
11          @Override
12          public long getSalario() {
13                  return salario;
14          }
15
16          @Override
17          public String getNome() {
18                  return nome;
19          }
20
21          @Override
22          public boolean add(AbstractEmployee e) throws NoSuchElementException {
23                  this.subordinados.add(e);
24                  return true;
25          }
26
27          @Override
28          public void remove(AbstractEmployee e) throws NoSuchElementException {
29                  this.remove(e);
30          }
31
32          @Override
33          public List<AbstractEmployee> subordinados() {
34                  return this.subordinados;
35          }
36
37          @Override
38          public AbstractEmployee getChild(String s) {
39                  for (int i = 0; i < subordinados.size(); i++)
40                          if(subordinados.get(i).getNome().equals(s))
41                                  return subordinados.get(i);
42
43                  return null;
44          }
45
46          @Override
47          public long getSalarios() {
48                  long soma = salario;
49                  for(int i = 0; i < subordinados.size(); i++)
50                          soma += subordinados.get(i).getSalarios();
51                  return soma;
52          }
53  }
```

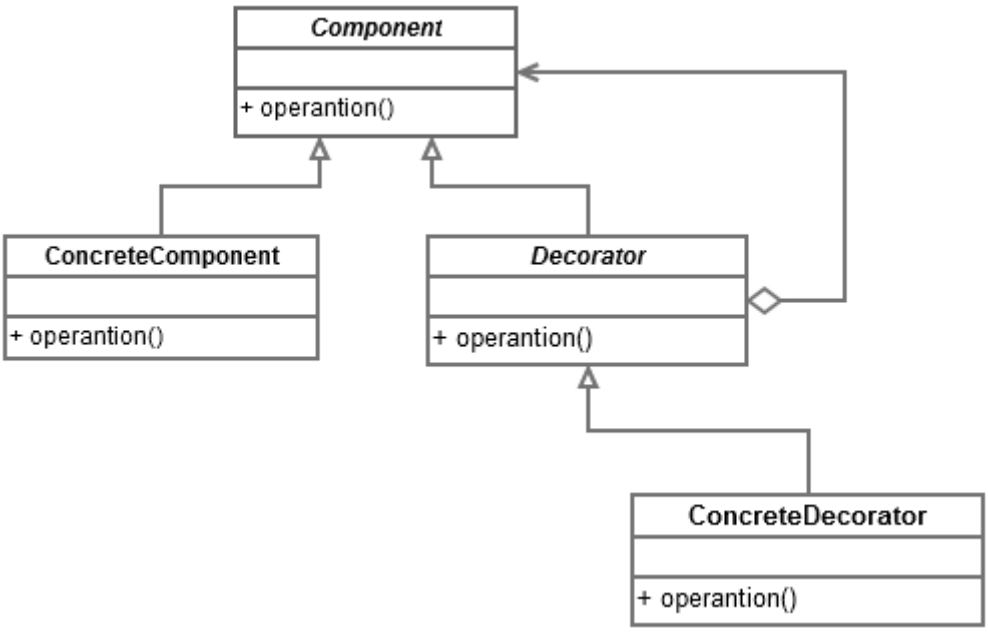
Note que, como essa classe se trata de um chefe, ela pode adicionar, remover e retornar subordinados. Observe também que no método que retorna os salários, é criado um arrays para somar o salário do chefe com os de seus subordinados.

Você pode criar a estrutura de funcionários da seguinte forma:

```
AbstractEmployee ceo, chefeMkt, chefeVendas, funMkt1, funMkt2, funVendas1;

ceo = new Boss("Presidente", 200000);
ceo.add(chefeMkt = new Boss("Chefe MKT", 100000));
ceo.add(chefeVendas = new Boss("Chefe Vendas", 100000));


chefeMkt.add(funMkt1 = new Employee("Funcionario MKT 1", 50000));
chefeMkt.add(funMkt2 = new Employee("Funcionario MKT 2", 50000));

chefeVendas.add(funVendas1 = new Employee("Funcionario Vendas 1", 50000));

long salariosTotal = ceo.getSalarios();
```

3.4 Padrão Decorator

O padrão Decorator fornece uma maneira de modificar o comportamento de objetos, sem a necessidade de criar uma classe derivada. Os comportamentos comuns a uma família de classes derivadas podem ser programadas em classes distintas para serem usadas sob demanda, permitindo a adição ou a remoção de comportamentos sem alterar uma classe.



Note que há uma classe abstrata que representa o comportamento-padrão de um componente. E que os comportamentos-padrão esperados por todos os decoradores são definidos em uma classe abstrata que estende a classe abstrata do componente.

Para exemplificar o padrão, suponha um sistema que grave as turmas de uma instituição. Nesse sistema, a classe Turma possui a seguinte estrutura:

```
public abstract class Turma {
        private int idturma;
        private String nome;

        //métodos getters e setters

        public abstract void save();

        public abstract void alterar();

        public abstract void excluir();
}
```

Uma turma básica poderia ser implementada desta forma:

```
public class TurmaBasica extends Turma {

        @Override
        public void save() {
                //implementação
        }

        @Override
        public void alterar() {
                //implementação
        }

        @Override
        public void excluir() {
                //implementação
        }
}
```

Agora, suponha que seja preciso adicionar algumas funcionalidades à turma. Para não alterar a classe que define a turma, as novas funcionalidades serão implementadas por um novo objeto decorador.

Mas, quando for necessário utilizar essas novas funcionalidades, não poderão ser usadas diretamente, pois não serão executadas. Primeiro você deve passar a informação da turma para o decorador que irá executar a nova tarefa. Assim, todo decorador deve possuir uma turma.

Para não afetar o código atual, os decoradores devem seguir a mesma estrutura das turmas. A classe abstrata que irá representar os decoradores da turma deve ter esta estrutura:

```
public abstract class DecoradorTurma extends Turma {
        private Turma turma;

        public DecoradorTurma(Turma turma){
                this.turma = turma;
        }

        public abstract void save();

        public abstract void alterar();

        public abstract void excluir();

        public Turma getTurma(){
                return this.turma;
        }
}
```

Agora você pode implementar um decorador:

```
public class TurmaTemporaria extends DecoradorTurma {
        Contrato contrato;

        public TurmaTemporaria(Turma turma) {
                super(turma);
        }

        @Override
        public void save() {
                // regras diferentes de salvar a turma
                this.getTurma().save();
        }

        @Override
        public void alterar() {
                // regras diferentes para alterar a turma
                this.getTurma().alterar();
        }

        @Override
        public void excluir() {
                // regras diferentes para alterar a turma
                this.getTurma().excluir();
        }

        public Contrato getContrato(){
```

```
27                  //Montar contrato
28                  return contrato;
29          }
30  }
```

Você pode usar o decorar da seguinte forma:

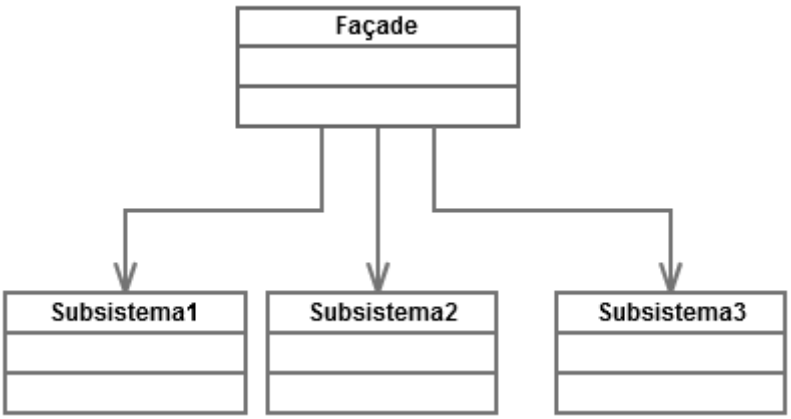
```
1  Turma turma = new TurmaBasica();
2
3  turma.setIdturma(1);
4  turma.setNome("Turma temporária");
5
6  TurmaTemporaria tempTurma = new TurmaTemporaria(turma);
7  tempTurma.save();
8  Contrato contato = tempTurma.getContrato();
9  ...
```

3.5 Padrão Façade

Esse padrão é utilizado para encapsular um conjunto de classes complexas em uma interface mais simples. Muitas vezes, os programas evoluem para um número sem-fim de subsistemas complicados e complexos.

O padrão Façade permite simplificar essa complexidade, fornecendo uma interface simplificada para esses subsistemas. Essa simplificação pode, em alguns casos, reduzir a flexibilidade das classes descendentes, mas geralmente fornece a todos uma função mais sofisticada.

Para entender o funcionamento desse padrão, veja o padrão abaixo:



Note que a classe do Façade conhece os subsistemas e executa as solicitações de cada um. Cada subsistema pode ser executado diretamente, ou pelo Façade.

Para implementar o padrão, suponha um sistema de e-commerce em que os subsistemas de estoque, financeiro e fiscal devam ser avisados assim que um pedido for realizado. O sistema está funcionando e, na hora de realizar um pedido, as ações são realizadas da seguinte forma:

```
1  Pedido pedido = new Pedido();
2  pedido.setNumero(1);
3  //...
4
5  Estoque estoque = new Estoque();
6  estoque.reserverProdutos(pedido);
7
8  Financeiro financeiro = new Financeiro();
9  financeiro.recebido(pedido);
10
11 Fiscal fiscal = new Fiscal();
12 fiscal.calcularImpostos(pedido);
```

A ideia do Façade é criar uma classe para encapsular essas ações, ficando da seguinte forma:

```
public class PedidoFacade {
        private Estoque estoque;
        private Financeiro financeiro;
        private Fiscal fiscal;

        public PedidoFacade(Estoque estoque, Financeiro financeiro, Fiscal fiscal){
                this.estoque = estoque;
                this.financeiro = financeiro;
                this.fiscal = fiscal;
        }

        public void efetivarPedido(Pedido pedido){
                this.estoque.reserverProdutos(pedido);
                this.financeiro.recebido(pedido);
                this.fiscal.calcularImpostos(pedido);
        }

        public void cancelarPedido(Pedido pedido){
                this.estoque.liberarEstoque(pedido);
                this.financeiro.devolverValores(pedido);
                this.fiscal.pedidoCancelado(pedido);
        }
}
```

Agora, quando um pedido for feito, basta chamar a classe PedidoFacade:

```
Pedido pedido = new Pedido();
pedido.setNumero(1);
//...

Estoque estoque = new Estoque();
Financeiro financeiro = new Financeiro();
Fiscal fiscal = new Fiscal();

PedidoFacade facade = new PedidoFacade(estoque, financeiro, fiscal);
facade.efetivarPedido(pedido);
```

3.6 Padrão Flyweight

O padrão Flyweight é utilizado para compartilhar objetos utilizados em grandes quantidades.

Às vezes, é preciso gerar um grande número de instâncias de pequenas classes para representar os dados. Isso acaba por colocar grande carga na memória da aplicação.

Uma forma de resolver esse problema é compartilhar objetos. Muitos desses pequenos objetos se diferenciam ligeiramente, enquanto que a maioria possui o estado e comportamento idênticos. Então, o compartilhamento reduz drasticamente o número de instâncias sem perder nenhuma funcionalidade.

O Flyweight usa a ideia de compartilhar objetos para aliviar a carga na memória.

Para entender o funcionamento do padrão veja o diagrama abaixo:

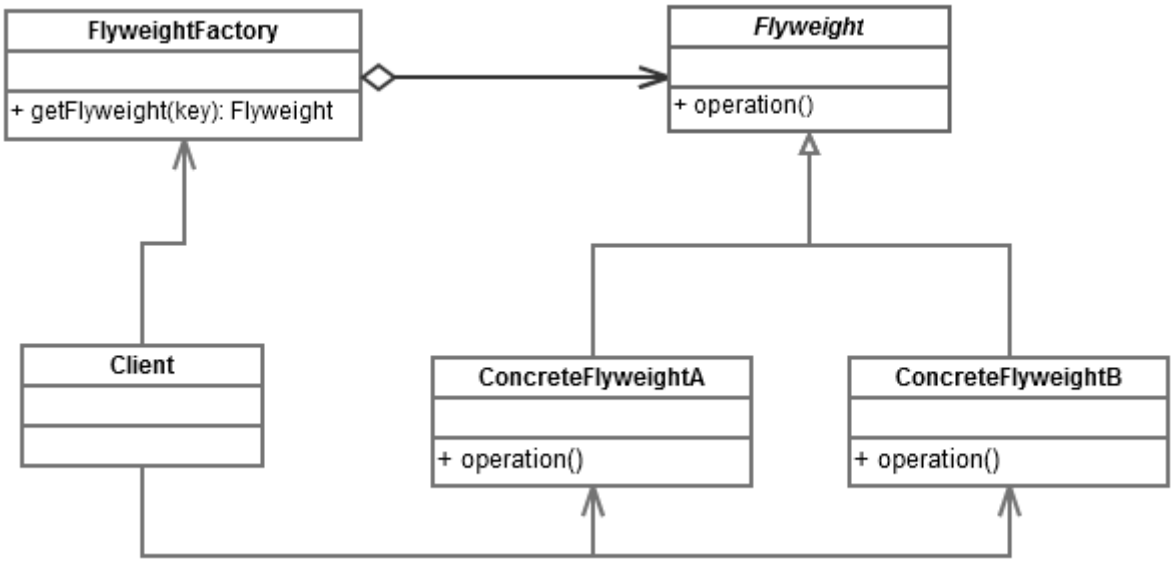
Observe que, para utilizar o padrão, há uma interface que fornece o comportamento-padrão de todos os objetos específicos. Uma fábrica é responsável pela criação e gerenciamento dos objetos.

Para exemplificar o padrão, suponha um sistema que lide com remessas, sendo que, para cada remessa, seja necessário pegar as informações do Estado, que pode ser utilizado várias vezes. Você pode utilizar o padrão para gerenciar a criação dos objetos Estados, reaproveitando instâncias, e criar somente quando for necessário.

Você pode definir uma interface para padronizar a estrutura dos estados:

```
public interface EstadoFlyweight {
        public void setId(int id);
        public int getId();
        public void setNome(String nome);
        public String getNome();
        public void setSigla(String sigla);
        public String getSigla();
}
```

Com isso, a classe Estado terá a seguinte estrutura:




```java
public class Estado implements EstadoFlyweight {
	private int id;
	private String nome;
	private String sigla;

	public Estado(String nome) {
		//busca as informações do estado pelo nome.
	}

	@Override
	public void setId(int id) {
		this.id = id;
	}

	@Override
	public int getId() {
		return this.id;
	}

	@Override
	public void setNome(String nome) {
		this.nome = nome;
	}

	@Override
	public String getNome() {
		return this.nome;
	}

	@Override
	public void setSigla(String sigla) {
		this.sigla = sigla;
	}

	@Override
	public String getSigla() {
		return this.sigla;
	}
}
```

Para controlar a criação e acesso dos objetos, implemente a seguinte classe:

```java
public class FactoryFlyweight {
	private static HashMap<String, Estado> estados = new HashMap<String, Estado>();

	public static Estado getEstado(String nome){
		if(!estados.containsKey(nome))
			estados.put(nome, new Estado(nome));

		return estados.get(nome);
	}
}
```

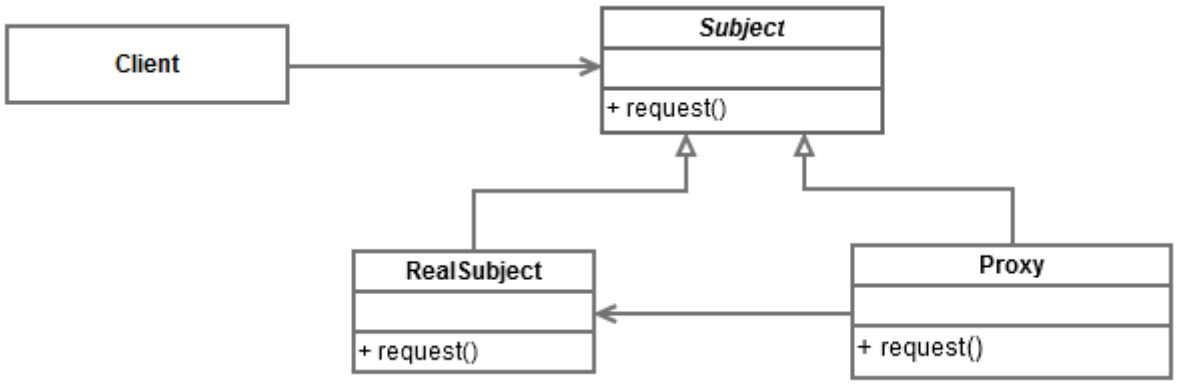
Com isso, os estados podem ser usados da seguinte forma:

```
Remessa remessa = new Remessa();
//...
remessa.setEstado(FactoryFlyweight.getEstado("São Paulo"));
```

3.7 Padrão Proxy

A ideia do padrão Proxy é fornecer um objeto substituto, que será chamado no lugar do objeto original. Esse conceito é utilizado em várias tecnologias como, por exemplo, o EJB – não gerenciado por um container JavaEE. Nesse caso, o EJB é acessado via um objeto proxy, que possui a mesma interface do original e que permite ter mais velocidade de acesso e segurança.

Para entender o funcionamento do padrão, veja o diagrama abaixo:



Note que há uma interface que irá definir a estrutura padrão para os objetos reais e para o proxy. O Proxy é utilizado para interceptar a conexão e depois enviar as chamadas para o objeto real.

Para exemplificar o padrão, suponha um sistema de cadastro de fornecedores que deverá ser auditado futuramente. Então, toda operação no fornecedor precisa ser registrada.

Para manter o sistema mais coeso, o registro das operações não deve ser implementado diretamente no objeto que representa o fornecedor. Essa lógica deve ser implementada em objetos intermediários.

Para preservar a estrutura dos objetos, deve ser criada a interface abaixo:

```
public interface Fornecedor {
        public boolean save();
        public boolean alter();
        public boolean delete();
        public ResultSet list();
}
```

E implementado o fornecedor-padrão:

```
public class FornecedorPadrao implements Fornecedor {

        //atributos e gettes e setters

        @Override
        public boolean save() {
                // lógica para salvar o fornecedor
                return true;
        }

        @Override
        public boolean alter() {
                // lógica para alterar o fornecedor
                return true;
        }

        @Override
        public boolean delete() {
                // lógica para excluir o fornecedor
```

```
20                  return true;
21          }
22
23          @Override
24          public ResultSet list() {
25                  // lógica de busca
26                  return null;
27          }
28  }
```

Implementando a mesma interface, podem ser criados objetos intermediários:

```
1   public class ProxyFornecedor implements Fornecedor {
2           private Fornecedor fornecedor;
3
4           public ProxyFornecedor(Fornecedor fornecedor){
5                   this.fornecedor = fornecedor;
6           }
7
8           @Override
9           public boolean save() {
10                  // lógica para registrar a operação
11                  return this.fornecedor.save();
12          }
13
14          @Override
15          public boolean alter() {
16                  // lógica para registrar a operação
17                  return this.fornecedor.alter();
18          }
19
20          @Override
21          public boolean delete() {
22                  // lógica para registrar a operação
23                  return this.fornecedor.delete();
24          }
25
26          @Override
27          public ResultSet list() {
28                  // lógica para registrar a operação
29                  return this.fornecedor.list();
30          }
31  }
```

Assim, na hora de realizar a operação é utilizado o objeto intermediário:

```
1   Fornecedor fornecedor = new FornecedorPadrao();
2   // atribuição dos dados
3
4   Fornecedor proxy = new ProxyFornecedor(fornecedor);
5   proxy.save();
```

3.8 Padrão MVC

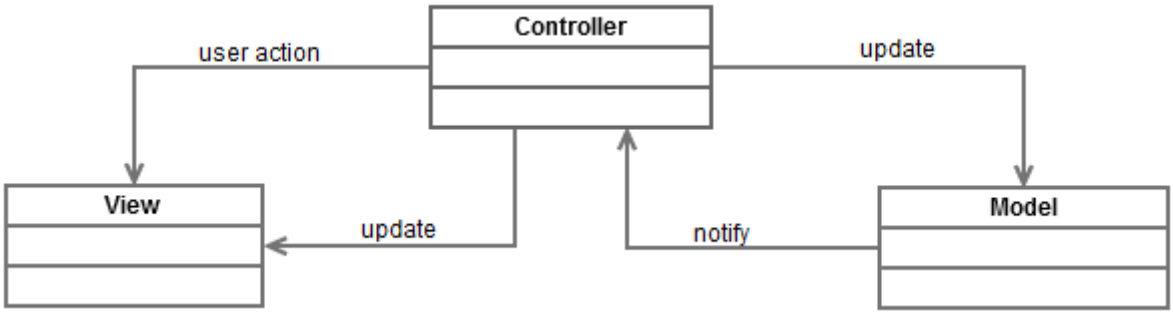
Esse não é um padrão GoF mas, por ser muito utilizado, será explicado. Ele separa a representação do dado da interação do usuário. Isso é feito separando a aplicação em três camadas: modelo (Model), interface (View) e controle (Controler). O modelo é a representação componentizada dos dados da aplicação e a lógica funcional. A interface fornece a representação visual dos dados da aplicação. Já o controle processa os eventos dirigidos pelos usuários, que podem resultar em resultados do sistema; ou seja, essa camada controla o fluxo da aplicação.

À primeira vista, esse padrão pode parecer um pouco complexo. Mas ele é bem simples, sendo, por isso, utilizado na maioria dos frameworks Java.

Para entender o funcionamento do padrão, veja o diagrama abaixo:

Como explicado acima, o padrão só possui três participantes principais que interagem entre si.

Quando um usuário interage com os componentes da camada view, os eventos são enviados para controle, que os processa. Se a requisição envolver alguma alteração em um modelo, o próprio controle pode manipular os dados do modelo, ou chamar uma operação específica para isso. Mas,



se a requisição só envolver alterações na camada view, o controle irá manipular isso e retornar o resultado.

Dependendo do modelo para apresentar os dados, a camada view não será diretamente alterada. Quando há mudanças no modelo, a camada view é notificada pelo controle, que consulta o modelo para adquirir as informações adicionais.

Como é possível notar, o controle atua como um intermediário entre a camada view e o modelo, notificando ou atualizando as informações das camadas.

Para exemplificar o padrão, suponha um sistema simples de gerência de funcionários. Para usar o padrão, primeiro deve ser definida uma classe abstrata para padronizar todos os modelos da aplicação:

```
1  public abstract class AbstractModel {
2
3          public abstract void save();
4          public abstract void alter();
5          public abstract void delete();
6          public abstract void select();
7
8  }
```

Que será utilizada para implementar o modelo Funcionario:

```
1  public class Funcionario extends AbstractModel {
2          private String nome;
3          private double salario;
4
5          public String getNome() {
6                  return nome;
7          }
8
9          public void setNome(String nome) {
10                 this.nome = nome;
11         }
12
13         public double getSalario() {
14                 return salario;
15         }
16
17         public void setSalario(double salario) {
18                 this.salario = salario;
19         }
20
21         @Override
22         public void save() {
23                 String row;
24                 row = this.nome;
25                 row += ";";
26                 row += this.salario;
27
28                 BufferedWriter writer = null;
29                 try{
30                         writer = new BufferedWriter(new FileWriter("C:\test\text.txt"));
31                         writer.write(row);
32                 }
33                 catch ( IOException e){
34                         e.printStackTrace();
35                 }
36                 finally{
37                         if (writer != null){
```

```
                                try{
                                        writer.flush();
                                        writer.close();
                                }
                                catch ( IOException e){
                                        e.printStackTrace();
                                }
                        }
                }
        }

        @Override
        public void alter() {
                String row;
                row = this.nome;
                row += ";";
                row += this.salario;

                BufferedWriter writer = null;
                try{
                        writer = new BufferedWriter(new FileWriter("C:\test\text.txt"));
                        writer.write(row);
                }
                catch ( IOException e){
                        e.printStackTrace();
                }
                finally{
                        if (writer != null){
                                try{
                                        writer.flush();
                                        writer.close();
                                }
                                catch ( IOException e){
                                        e.printStackTrace();
                                }
                        }
                }
        }

        @Override
        public void delete() {
                File file = new File("C:\test\text.txt");
                if(file.exists())
                file.delete();
        }

        @Override
        public void select() {
                BufferedReader read = null;
                try {
                        read = new BufferedReader(new FileReader("infilename"));
                        String str;
                        while ((str = read.readLine()) != null) {
                                process(str);
                        }
                } catch (IOException e) {
                        e.printStackTrace();
                }
                finally{
                        if (read != null){
                                try{
                                        read.close();
                                }
                                catch ( IOException e){
                                        e.printStackTrace();
                                }
                        }
                }
        }

        private void process(String str) {
                String[] args = str.split(";");
                if(args.length > 0){
                        this.nome = args[0];
                        this.salario = Double.parseDouble(args[1]);
                }
```

```
114         }
115
116 }
```

Agora você precisa criar o controle. Da mesma forma que no modelo, crie uma classe abstrata para padronizar os controles da aplicação:

```
1  public abstract class AbstractController {
2
3          public abstract void init(AbstractModel model);
4
5  }
```

Que será aplicada no controle funcionário:

```
1  public class FuncionarioController extends AbstractController {
2          private Funcionario modelFun;
3
4          @Override
5          public void init(AbstractModel model) {
6                  this.modelFun = (Funcionario) model;
7          }
8
9                  public Funcionario edit(String nome, double salario){
10                 this.modelFun.setNome(nome);
11                 this.modelFun.setSalario(salario);
12                 return this.modelFun;
13         }
14
15         public void setModel(Funcionario fun){
16                 this.modelFun = fun;
17         }
18
19 }
```

Note que a classe possui uma referência do modelo para poder manipulá-lo. Quando a View for criada, essas classes serão modificadas para também adicionarem uma referência dela. Mas, antes de criar a view, volte no modelo e adicione uma referência para o controle, da seguinte forma:

```
1  public class Funcionario extends AbstractModel {
2          private String nome;
3          private double salario;
4          private FuncionarioController controller;
5
6          public Funcionario(FuncionarioController controller){
7                  this.controller = controller;
8                  this.controller.setModel(this);
9          }
10
11 //Demais métodos da classe
12
13 }
```

Assim, quando o modelo for criado, deverá ser informado qual é o controle.

Agora crie a view. Da mesma forma que com os outros componentes, primeiro crie uma classe abstrata para padronizar as views da aplicação:

```
1  public abstract class AbstractView {
2
3          public abstract void display(AbstractModel model);
4
5  }
```

Que será implementada na view:

```
1  public class FuncionarioView extends AbstractView {
2
3          @Override
4          public void display(AbstractModel model) {
5                  Funcionario fun = (Funcionario) model;
6                  System.out.println("Nome do funcionário: "+ fun.getNome());
7                  System.out.println("Salário do funcionário: " + fun.getSalario());
8                  System.out.println("======================");
9          }
10
11 }
```

Antes de continuar a modificar a view, volte na classe AbstractController e modifique o código para:

```
public abstract class AbstractController {

        public abstract void init(AbstractModel model, AbstractView view);

}
```

Na classe FuncionarioController, adicione a referência à view, da seguinte forma:

```
public class FuncionarioController extends AbstractController {
        private Funcionario modelFun;
        private FuncionarioView viewFun;

        @Override
        public void init(AbstractModel model, AbstractView view) {
                this.modelFun = (Funcionario) model;
                this.viewFun = (FuncionarioView) view;
        }

        public Funcionario edit(String nome, double salario){
                this.modelFun.setNome(nome);
                this.modelFun.setSalario(salario);
                return this.modelFun;
        }

        public void setModel(Funcionario fun){
                this.modelFun = fun;
        }

        public void setView(FuncionarioView view){
                this.viewFun = view;
        }

        public void index(){
                this.viewFun.display(this.modelFun);
        }

}
```

Agora volte para a view e adicione mais alguns métodos:

```
public class FuncionarioView extends AbstractView {
        private FuncionarioController controller;

        public FuncionarioView(FuncionarioController controller){
                this.controller = controller;
                this.controller.setView(this);
        }

        @Override
        public void display(AbstractModel model) {
                Funcionario fun = (Funcionario) model;
                System.out.println("Nome do funcionário: "+ fun.getNome());
                System.out.println("Salário do funcionário: " + fun.getSalario());
                System.out.println("======================");
        }

        public void edit(){
                System.out.println("Entre com um novo nome....");
                BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
                try {
                        String nome = br.readLine();
                        System.out.println("Entre com um novo salário ....");
                        String number = br.readLine();
                        double salario = Double.parseDouble(number);
                        Funcionario newfun = controller.edit(nome, salario);
                        display(newfun);
                } catch (IOException e) {
                        e.printStackTrace();
                }catch(NumberFormatException e){
                        e.printStackTrace();
                }
        }

}
```

Essas classes podem ser testadas da seguinte forma:

```
FuncionarioController fc = new FuncionarioController();

// Criando modelo
Funcionario fun = new Funcionario(fc);
fun.setNome("Carlos Silva");
fun.setSalario(5600);

// Criando a view
FuncionarioView funView = new FuncionarioView(fc);

// exibindo
fc.index();
```

4 Padrões Comportamentais

Padrões comportamentais se preocupam com o fluxo de controle do sistema, mais especificamente a forma com que os objetos do sistema se comunicam. Algumas formas de organizar a comunicação de objetos do sistema podem produzir grandes benefícios em termos de eficiência e facilidade de manutenção desse sistema.

Nessa categoria estão os padrões:

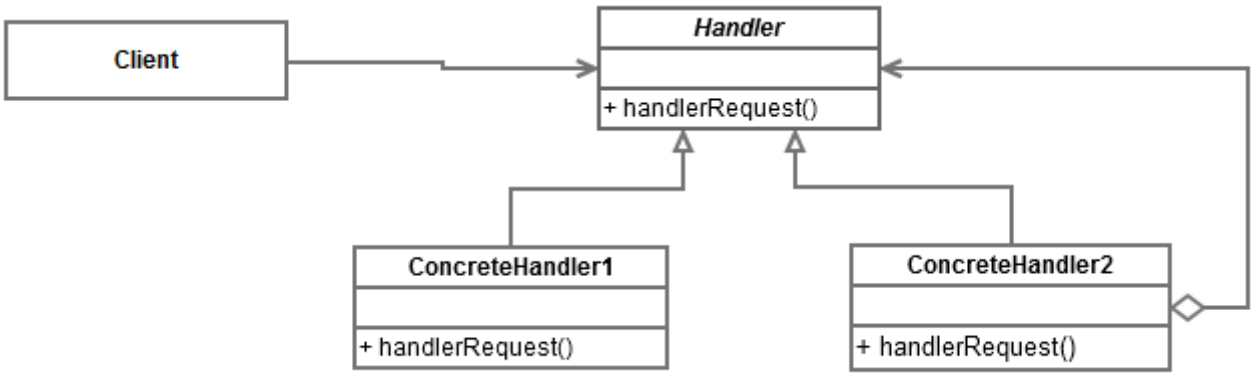
- **Chain of Responsibility Pattern**: permite o desacoplamento entre objetos, possibilitando que uma solicitação a um objeto seja delegada a outro objeto da cadeia, até que seja atendida.
- **Command Pattern**: permite controlar as chamadas de um objeto, encapsulando a solicitação como um objeto, possibilitando que as operações possam ser desfeitas, enfileiradas ou registradas.
- **Interpreter Pattern**: dada uma linguagem, define uma representação para sua gramática e um interpretador para interpretar as sentenças nessa linguagem.
- **Iterator Pattern**: fornece uma maneira de percorrer uma coleção sem a necessidade de expor a sua representação.
- **Mediator Pattern**: define como a comunicação entre os objetos pode ser simplificada, usando um objeto mediador que realizará toda a comunicação entre os objetos.
- **Memento Pattern**: permite que o estado do objeto seja salvo para ser restaurado depois.
- **Observer Pattern**: define uma dependência entre objetos para que, quando um objeto mudar de estado, todos os seus dependentes sejam notificados.
- **State Pattern**: permite que um objeto altere o seu comportamento quando mudar de estado.
- **Strategy Pattern**: permite que uma classe encapsule um algoritmo, de modo que ele possa ser utilizado na resolução de determinado problema.
- **Template Method Pattern**: fornece a definição de um algoritmo, postergando a implementação de alguns passos para as subclasses.
- **Visitor Pattern**: permite acrescentar operações em uma coleção de objetos, de forma não invasiva.
- **Object Null Pattern**: fornece um objeto substituto, na falta de determinado tipo de objeto.

Conheça cada padrão em detalhes.

4.1 Padrão Chain of Responsibily

Permite a um conjunto de classes lidar com uma solicitação, sem que nenhuma delas saiba as funcionalidades das outras classes. Isso é feito com o desacoplamento entre um objeto solicitante e o objeto solicitado, e criando apenas um acoplamento fraco entre todos os objetos, onde o único elo comum é o pedido passado entre eles.

Para entender o funcionamento do padrão, veja o diagrama abaixo:



Observe que o padrão possui uma interface que define o método a ser utilizado para enviar a solicitação para o próximo objeto. Na implementação dessa interface, os objetos mantêm uma referência para o próximo objeto do conjunto.

Para exemplificar o padrão, suponha um sistema de e-mail que trate a cada um de forma diferenciada, de acordo com o domínio.

Para padronizar as classes que lidarão com o e-mail, é preciso modelar a classe responsável por encadear o objeto. Faça da seguinte forma:

```
1  public abstract class EmailHandler {
2          protected EmailHandler next;
3
4          public void setNext(EmailHandler handler){
5                  this.next = handler;
6          }
7
8          public abstract void handleRequest(Email email);
9  }
```

Agora você criar os manipuladores, de acordo com o domínio do e-mail. Para o e-mail da empresa:

```
1  public class BusinessEmailHandler extends EmailHandler {
2
3          @Override
4          public void handleRequest(Email email) {
5                  if(!email.getFrom().endsWith("@empresa.com.br"))
6                          this.next.handleRequest(email);
7                  else{
8                          // lógica para lidar com os e-mails do trabalho
9                  }
10         }
11 }
```

Ou o Gmail:

```
1  public class GmailEmailHandler extends EmailHandler {
2
3          @Override
4          public void handleRequest(Email email) {
5                  if(!email.getFrom().endsWith("@gmail.com"))
6                          this.next.handleRequest(email);
7                  else{
8                          // lógica para lidar com os e-mails do gmail
9                  }
10         }
11 }
```

Você pode executar essas classes da seguinte forma:

```
EmailHandler business = new BusinessEmailHandler();
EmailHandler gmail = new GmailEmailHandler();

business.setNext(gmail);
gmail.setNext(null);

Email email = new Email();
//informações do e-mail

business.handleRequest(email);
```

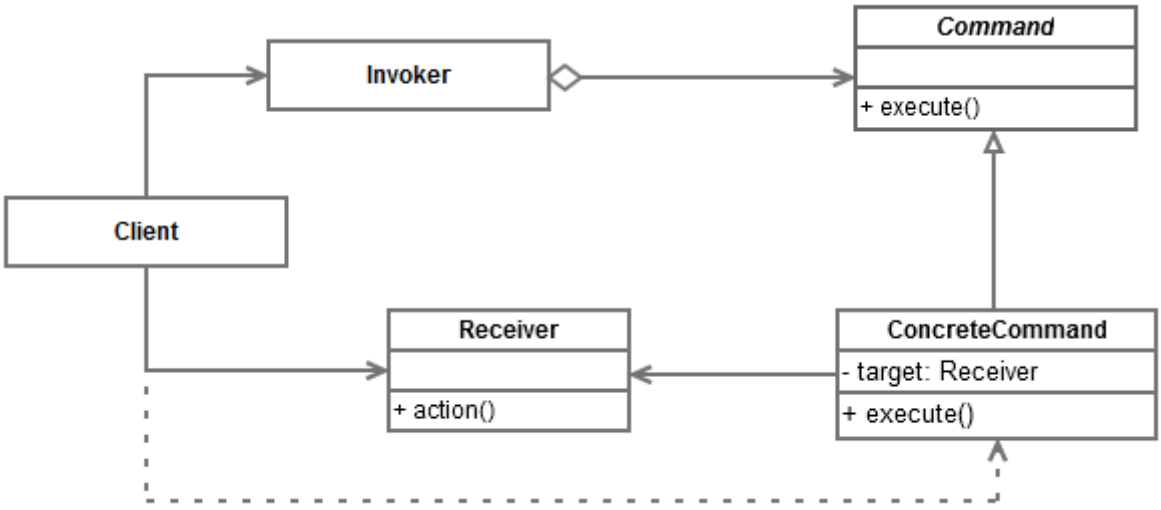
Nesse código, se o e-mail não for da empresa, será chamado o Handler do Gmail para lidar com ele; e pode ser assim, sucessivamente, se existissem mais Handlers.

4.2 Padrão Command

O padrão Command permite que um objeto possa encapsular um comando, que será chamado pelo cliente sem que este precise saber a real ação a ser executada pelo objeto, além de permitir que você altere tal ação sem afetar o cliente.

Esse padrão já é utilizado em algumas interfaces do Java como a KeyListener e MouseListener, entre outras. O fato é que o construtor da biblioteca do componente teclado, ou mouse, não sabe a ação que será executada quando uma tecla, ou o mouse, for pressionado, mas ele disponibiliza os métodos que serão executados nesses eventos, cabendo ao desenvolvedor programar a ação.

Para entender melhor o funcionamento do padrão, veja o diagrama abaixo:



Observe que há uma interface que irá definir os métodos que podem ser implementados pelos objetos. Um invocador (Invoker) é utilizado para chamar todos os métodos definidos pela interface Command. O Receiver é o objeto-alvo do comando, especificado nas classes ConcreteCommand. Mas também existe a possibilidade de o Receiver e o ConcreteCommand serem o mesmo objeto. Esse comportamento irá depender da forma em que o padrão for aplicado.

Para implementar o padrão, suponha um sistema de compra de ações onde o usuário possa comprar ou vender uma ação:

```
public class NegociarAcao {
        public void compra(){
                //lógica para comprar ações
        }

        public void venda(){
                //lógica para vender ações
        }
}
```

A compra e a venda das ações só pode ocorrer em horário específico. Para implementar, será preciso criar a interface abaixo para padronizar a execução do comando:

```
public interface OrdemCommand {
        public void execute();
}
```

Para controlar a compra, crie a seguinte classe:

```
public class OrdemCompra implements OrdemCommand {
        private NegociarAcao negociar;

        public OrdemCompra(NegociarAcao negociar){
                this.negociar = negociar;
        }

        @Override
        public void execute() {
                this.negociar.compra();
        }
}
```

Para a venda, crie esta classe:

```
public class OrdemVenda implements OrdemCommand {
        private NegociarAcao negociar;

        public OrdemVenda(NegociarAcao negociar){
                this.negociar = negociar;
        }

        @Override
        public void execute() {
                this.negociar.venda();
        }
}
```

Agora você deve implementar a classe que irá disparar o método execute, implementado pela classe de compra e venda:

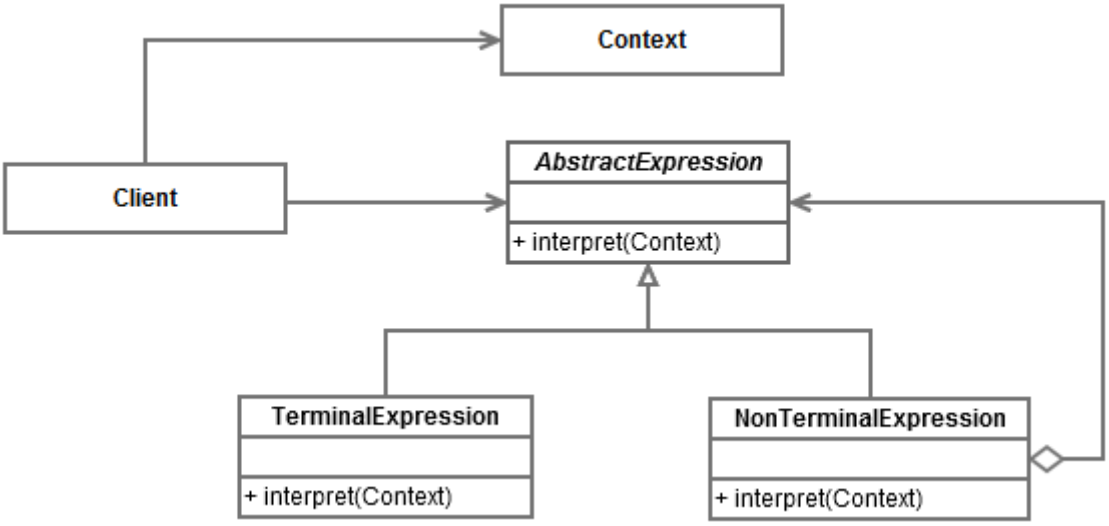
```
public class Agente {
        private List<OrdemCommand> ordens = new ArrayList<OrdemCommand>();

        public void add(OrdemCommand ordem){
                this.ordens.add(ordem);
        }

        public void executa(){
                for(OrdemCommand ordem : this.ordens)
                ordem.execute();
        }
}
```

Por fim, pode usar os comandos da seguinte forma:

```
NegociarAcao negociar = new NegociarAcao();
//criar as ordens de compra e venda
OrdemCommand ordemCompra = new OrdemCompra(negociar);
OrdemCommand ordemVenda = new OrdemVenda(negociar);

//adicioná-las ao agente
Agente agente = new Agente();
agente.add(ordemCompra);
agente.add(ordemVenda);

//Quando as ações puderem ser executadas
agente.executa();
```

4.3 Padrão Interpreter

Esse padrão propõe um modelo para que objetos possam interpretar a gramática de uma linguagem. Por ser bem específico, esse padrão é pouco utilizado, ficando restrito às expressões regulares e na linguagem Query do Framework Hibernate.



Para entender o funcionamento do padrão interpreter, veja o diagrama a seguir:

Note a interface que o cliente utiliza para interagir com as expressões. Duas classes são descendentes dessa interface: a TerminalExpression, destinada a interpretar os nós terminais da gramática; e a NonTerminalExpression, destinada a interpretar os nós não terminais da gramática. Também há a classe Context, que contém a expressão da gramática que será interpretada.

Para exemplificar o padrão, suponha um sistema que precise interpretar um numeral romano para um valor numérico.

Para pegar a expressão, crie a classe Contexto:

```
public class Contexto {
        private String input;
        private int output;

        public String getInput() {
                return input;
        }

        public void setInput(String input) {
                this.input = input;
        }

        public int getOutput() {
                return output;
        }

        public void setOutput(int output) {
                this.output = output;
        }
}
```

Para definir a estrutura dos interpretadores, crie a seguinte classe abstrata:

```
public abstract class Expressao {
        public void interpretador(Contexto contexto){
                if (contexto.getInput().length() == 0)
                        return;

                if (contexto.getInput().startsWith(nove()))
                {
                        contexto.setOutput(contexto.getOutput() + (9 * multiplo()));
                        contexto.setInput(contexto.getInput().substring(2));
                }
                else if (contexto.getInput().startsWith(cinco()))
                {
                        contexto.setOutput(contexto.getOutput() + (5 * multiplo()));
                        contexto.setInput(contexto.getInput().substring(1));
                }
                else if (contexto.getInput().startsWith(quatro()))
                {
                        contexto.setOutput(contexto.getOutput() + (4 * multiplo()));
```

```
19                          contexto.setInput(contexto.getInput().substring(2));
20                  }
21
22                  while (contexto.getInput().startsWith(um()))
23                  {
24                          contexto.setOutput(contexto.getOutput() + (1 * multiplo()));
25                          contexto.setInput(contexto.getInput().substring(1));
26                  }
27          }
28
29          protected abstract String nove();
30          protected abstract String cinco();
31          protected abstract String quatro();
32          protected abstract String um();
33          protected abstract int multiplo();
34  }
```

Para cada casa numérica dos numerais gregos, é preciso implementar uma classe que será descendente de Expressao. Primeiro, para a casa de milhar:

```
1   public class ExpressaoMilhar extends Expressao {
2
3           @Override
4                   protected String nove() {
5                   return " ";
6           }
7
8           @Override
9                   protected String cinco() {
10                  return " ";
11          }
12
13          @Override
14                  protected String quatro() {
15                  return " ";
16          }
17
18          @Override
19                  protected String um() {
20                  return "M";
21          }
22
23          @Override
24                  protected int multiplo() {
25                  return 1000;
26          }
27  }
```

Depois, para a casa das centenas:

```
1   public class ExpressaoCentena extends Expressao {
2
3           @Override
4           protected String nove() {
5                   return "CM";
6           }
7
8           @Override
9           protected String cinco() {
10                  return "D";
11          }
12
13          @Override
14          protected String quatro() {
15                  return "CD";
16          }
17
18          @Override
19          protected String um() {
20                  return "C";
21          }
22
23          @Override
24          protected int multiplo() {
25                  return 100;
26          }
```

```
}
```

Para a casa das dezenas:

```
public class ExpressaoDecena extends Expressao {

	@Override
	protected String nove() {
		return "XC";
	}

	@Override
	protected String cinco() {
		return "L";
	}

	@Override
	protected String quatro() {
		return "XL";
	}

	@Override
	protected String um() {
		return "X";
	}

	@Override
	protected int multiplo() {
		return 10;
	}
}
```

E para os valores até 9:

```
public class ExpressaoUm extends Expressao {

	@Override
	protected String nove() {
		return "IX";
	}

	@Override
	protected String cinco() {
		return "V";
	}

	@Override
	protected String quatro() {
		return "IV";
	}

	@Override
	protected String um() {
		return "I";
	}

	@Override
	protected int multiplo() {
		return 1;
	}
}
```

Agora você deve usar essas classes da seguinte forma:

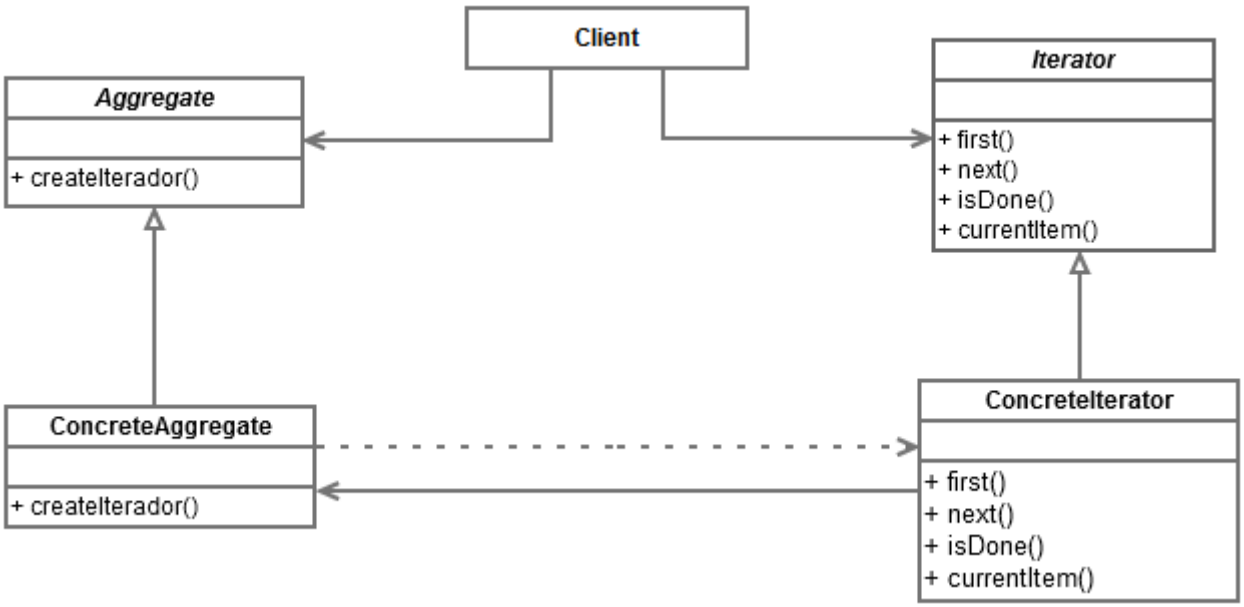
```
String romano = "MCMXXVIII";

Contexto contexto = new Contexto();
contexto.setInput(romano);

List<Expressao> tree = new ArrayList<Expressao>();
tree.add(new ExpressaoMilhar());
tree.add(new ExpressaoCentena());
tree.add(new ExpressaoDecena());
tree.add(new ExpressaoUm());

for(Expressao exp : tree){
exp.interpretador(contexto);
```

```
}

String resultado = String.valueOf(contexto.getOutput());
```

4.4 Padrão Iterador

O padrão Iterator é um dos mais simples e um dos mais utilizados padrões de design. Ele permite que uma lista ou coleção seja percorrida usando-se uma interface padrão sem ter a necessidade de saber as representações internas dos dados. Também é possível definir iteradores especiais, que executam algum processamento especial, ou retornar apenas elementos específicos.

Para entender o funcionamento do padrão, veja o diagrama abaixo:



Repare que a interface Iterator define os métodos-padrão de um Iterator, e a interface Aggregate define o padrão do método que irá criar o iterator, além de duas classes que implementam as interfaces.

Como esse padrão é utilizado em todas as listas do Java, é provável que você já o tenha utilizado. Mesmo assim, acompanhe este exemplo. Suponha um sistema onde seja necessário criar uma coleção de itens:

```
public class Itens {
        private Map<Integer, String> itens;
        private int index;

        public Itens(){
                itens = new HashMap<Integer, String>();
                index = 0;
        }

        public void add(String item){
                this.itens.put(index, item);
                index++;
        }

        public String remove(String item){
                return this.itens.remove(item);
        }

        public int getNumberOfItems(){
                return this.itens.size();
        }
}
```

Para implementar o Iterator nessa lista, primeiro você deverá defini-lo. Comece criando uma interface para padronizar a sua estrutura:

```
public interface Iterator<T> {
        public T first();
        public T next();
        public boolean isDone();
        public T currentItem();
}
```

Agora implemente-a na classe abaixo:

```
public class ConcreteIterator implements Iterator<String> {
        private Map<Integer, String> itens;
        private int index;

        public ConcreteIterator(Map<Integer, String> itens){
                this.itens = itens;
                index = -1;
        }

        @Override
                public String first() {
                return itens.get(0);
        }

        @Override
        public String next() {
                index++;
                return itens.get(index);
        }

        @Override
        public boolean isDone() {
                if(index > itens.size())
                        return true;
                else
                        return false;
        }

        @Override
        public String currentItem() {
                return itens.get(index);
        }
}
```

Para gerar o Iterador, crie a seguinte interface:

```
public interface Aggregate {
        public Iterator<String> getIterator();
}
```

Ela será implementada na classe Itens:

```
public class Itens implements Aggregate {
        private Map<Integer, String> itens;
        private int index;

        public Itens(){
                itens = new HashMap<Integer, String>();
                index = 0;
        }

        public void add(String item){
                this.itens.put(index, item);
                index++;
        }

        public String remove(String item){
                return this.itens.remove(item);
        }

        public int getNumberOfItems(){
                return this.itens.size();
        }

        @Override
        public Iterator<String> getIterator() {
                return new ConcreteIterator(itens);
        }
}
```

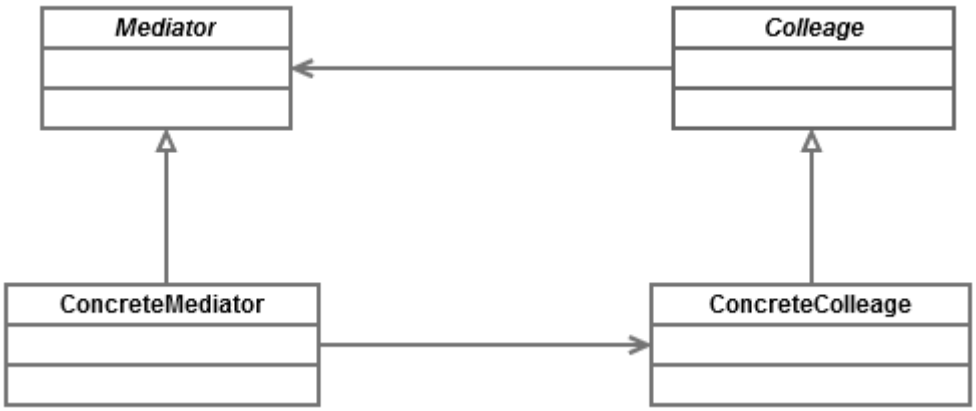
Ela pode ser utilizada da seguinte forma:

```
Itens itens = new Itens();

itens.add("Item 0");
itens.add("Item 1");
itens.add("Item 2");
itens.add("Item 3");
itens.add("Item 4");

Iterator<String> iterator;

iterator = itens.getIterator();
while(!iterator.isDone()){
        System.out.println(iterator.next());
}
```

4.5 Padrão Mediator

O padrão Mediator é utilizado para resolver o problema de sistemas que possuem muitas classes que se comunicam entre si, e de uma forma tão complexa que fica difícil entende-las e fazer qualquer manutenção no sistema. Nesse caso, o padrão centraliza as comunicações do sistema e reduz o acoplamento e encapsulamento entre os objetos.

Para entender o funcionamento do padrão, veja o diagrama abaixo:



Repare que a interface Mediator irá definir os métodos que os objetos poderão utilizar para recorrer ao Mediator e a interface Colleage, que define os métodos que um Mediator pode utilizar para informar aos objetos caso uma solicitação seja recebida pelo Mediator.

Para exemplificar o padrão, suponha um sistema de chat. Primeiro você deve definir a interface do Mediator:

```
public interface Mediator {
        public void send(String message, Usuario from);
}
```

Para o usuário, use uma classe abstrata:

```
public abstract class Usuario {
        private Mediator mediator;

        public Usuario(Mediator mediator){
                this.mediator = mediator;
        }

        public void send(String message){
                this.mediator.send(message, this);
        }

        public Mediator getMediator()
        {
                return mediator;
```

```
        }

        public abstract void receive(String message);
}
```

Agora implemente o Mediator:

```
public class MediatorCentral implements Mediator {
        private ArrayList<Usuario> usuarios;

        public MediatorCentral()
        {
                usuarios = new ArrayList<Usuario>();
        }

        public void addUsuario(Usuario usuario){
                this.usuarios.add(usuario);
        }

        @Override
        public void send(String message, Usuario from) {
                for(Usuario usuario: usuarios){
                        if(usuario != from)
                                usuario.receive(message);
                }
        }
}
```

Note que a mensagem é enviada a todos os objetos, menos para o que enviou a mensagem.

Por fim, implemente os usuários do chat:

```
public class UsuarioChat extends Usuario {

        public UsuarioChat(Mediator mediator) {
                super(mediator);
        }

        @Override
        public void receive(String message) {
                //Tratar mensagem recebida
        }
}
```

Você também poderia implementar outros objetos:

```
public class UsuarioChatMobile extends Usuario {

        public UsuarioChatMobile(Mediator mediator) {
                super(mediator);
        }

        @Override
        public void receive(String message) {
                // Tratar mensagem recebida
        }
}
```

E usar as classes da seguinte forma:

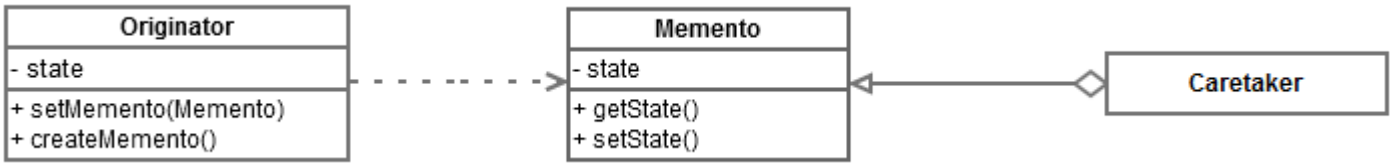
```
MediatorCentral mediator = new MediatorCentral();

UsuarioChat user1 = new UsuarioChat(mediator);
UsuarioChat user2 = new UsuarioChat(mediator);
UsuarioChatMobile userMobile1 = new UsuarioChatMobile(mediator);

mediator.addUsuario(user1);
mediator.addUsuario(user2);
mediator.addUsuario(userMobile1);

user1.send("Olá!");
```

4.6 Padrão Memento

O padrão Memento é utilizado para capturar o estado de um objeto em determinado momento para, posteriormente, poder restaurar o objeto para esse estado. Isso já é utilizado em sistemas com a

opção de desfazer uma ação.

Para entender o funcionamento, veja o diagrama abaixo:



Note a classe Memento, que armazena o estado do objeto Originator. De acordo com o padrão, apenas a classe Originator pode ter acesso ao objeto Memento porque será essa classe que irá criar o Memento com seu estado atual e restaurá-lo quando for necessário. Já a classe Caretaker é responsável pela guarda do Memento.

Ao aplicar esse padrão, alguns pontos devem ser analisados. Dependendo do projeto, alguns componentes com a funcionalidade de desfazer, podem não permitir que todas as ações sejam desfeitas. Por exemplo, uma ação de exclusão poderia ser irretornável; ou, para desfazer uma operação de inclusão, algumas regras devem ser respeitadas. De qualquer forma, antes de aplicar o padrão, às políticas que serão aplicadas, as ações desfeitas devem ser pensadas.

Para exemplificar o padrão, no exemplo em execução, suponha um simples cadastro de funcionários.

Primeiro você deve definir a classe Funcionario:

```
1  public class Funcionario {
2          private int nr;
3          private String nome;
4          private int idade;
5          private String sexo;
6          private String telefone;
7
8          public int getNr() {
9                  return nr;
10         }
11
12         public void setNr(int nr) {
13                 this.nr = nr;
14         }
15
16         public String getNome() {
17                 return nome;
18         }
19
20         public void setNome(String nome) {
21                 this.nome = nome;
22         }
23
24         public int getIdade() {
25                 return idade;
26         }
27
28         public void setIdade(int idade) {
29                 this.idade = idade;
30         }
31
32         public String getSexo() {
33                 return sexo;
34         }
35
36         public void setSexo(String sexo) {
37                 this.sexo = sexo;
38         }
39
40         public String getTelefone() {
41                 return telefone;
42         }
43
44         public void setTelefone(String telefone) {
```

```
                this.telefone = telefone;
        }
}
```

Será essa classe que armazenará o estado. Por isso, defina o Memento para ela:

```
public class MementoFuncionario {
        private Funcionario target;
        private int nr;
        private String nome;
        private int idade;
        private String sexo;
        private String telefone;

        public MementoFuncionario(Funcionario funcionario){
                this.target = funcionario;
                this.nr = funcionario.getNr();
                this.nome = funcionario.getNome();
                this.idade = funcionario.getIdade();
                this.sexo = funcionario.getSexo();
                this.telefone = funcionario.getTelefone();
        }

        public void undo(){
                int newRr;
                String newNome;
                int newIdade;
                String newSexo;
                String newTelefone;

                newRr = this.nr;
                newNome = this.nome;
                newIdade = this.idade;
                newSexo = this.sexo;
                newTelefone = this.telefone;

                this.nr = this.target.getNr();
                this.nome = this.target.getNome();
                this.idade = this.target.getIdade();
                this.sexo = this.target.getSexo();
                this.telefone = this.target.getTelefone();

                this.target.setNr(newRr);
                this.target.setNome(newNome);
                this.target.setIdade(newIdade);
                this.target.setSexo(newSexo);
        this.target.setTelefone(newTelefone);
        }
}
```

Com a classe Memento definida, volte à classe Funcionario e adicione o método que irá salvar seu estado:

```
public class Funcionario {
        //atributos e propriedades

        public MementoFuncionario salvarEstado(){
                return new MementoFuncionario(this);
        }
}
```

Para guardar o Memento, defina a classe Caretaker:

```
public class Caretaker {
        private MementoFuncionario memento;

        public MementoFuncionario getMemento() {
                return memento;
        }

        public void setMemento(MementoFuncionario memento) {
                this.memento = memento;
        }
}
```

Essas classes podem ser usadas da seguinte forma:

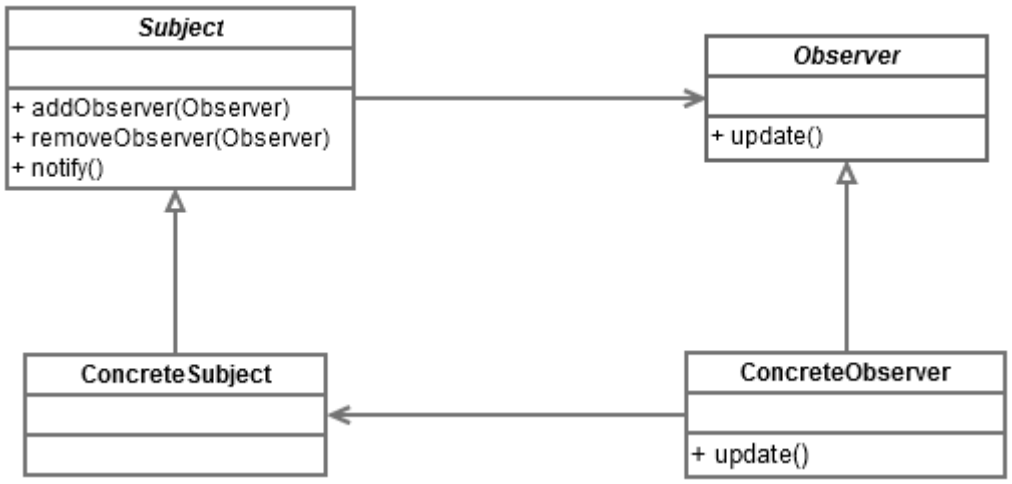
```
Funcionario fun = new Funcionario();

fun.setNr(23);
fun.setNome("Carlos Silva");
fun.setIdade(26);
fun.setSexo("Masculino");
fun.setTelefone("+55-11-1111-1111");

Caretaker caretaker = new Caretaker();

caretaker.setMemento(fun.salvarEstado());

fun.setNome("Carlos Silva Santos");
fun.setIdade(30);
fun.setTelefone("XX-XX-XXXX-XXXX");

caretaker.getMemento().undo();
```

4.7 Padrão Observer

O padrão Observer é utilizado para definir dependências entre objetos, de forma que quando um objeto mudar seu estado, todos os objetos dependentes sejam informados sobre a mudança.

Esse padrão é utilizado, por exemplo, em Frameworks: quando um arquivo de configuração é alterado, outros objetos são informados de tal mudança para poderem aplicá-la.

Para entender o funcionamento do padrão, veja o diagrama abaixo:



Observe a classe abstrata Subject, que define como um Observer pode interagir com os objetos Subjects. E a interface Observer, utilizada pelo Subject para comunicar aos objetos sobre a mudança de estado.

Para exemplificar o padrão, suponha um sistema de ações que informe a todos os interessados o valor da ação.

Para padronizar os objetos que serão notificados, defina a interface abaixo:

```
public interface Observer {
        public void atualizacao(Acao acao);
}
```

Agora você pode implementar os interessados nas informações sobre as ações, como o corretor:

```
public class Corretor implements Observer {

        @Override
        public void atualizacao(Acao acao) {
                //implementação
        }
}
```

Ou um Banco:

```
public class Banco implements Observer {

        @Override
        public void atualizacao(Acao acao) {
                //implementação
        }
}
```

Para padronizar as ações, defina a seguinte classe abstrata:

```
public abstract class Acao {
        private List<Observer> interessados = new ArrayList<Observer>();
        private double valor;

        public void addInterressado(Observer interessado){
                if(!this.interessados.contains(interessado))
                        this.interessados.add(interessado);
        }

        public void removeInteressado(Observer interessado){
                this.interessados.remove(interessado);
        }

        public double getValor() {
                return valor;
        }

        public void setValor(double valor) {
                this.valor = valor;
                for(Observer observer : interessados)
                observer.atualizacao(this);
        }

        public abstract void compra();
        public abstract void venda();
}
```

Que pode ser aplicada para as ações ordinárias:

```
public class AcaoOrdinaria extends Acao {

        @Override
        public void compra() {
                // Implementação
        }

        @Override
        public void venda() {
                // Implementação
        }
}
```

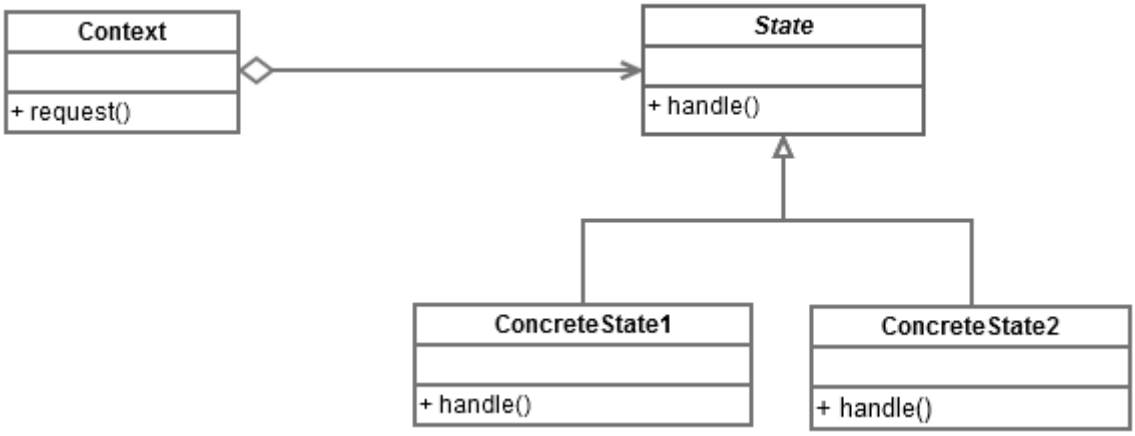
Essas classes podem ser utilizadas da seguinte forma:

```
Acao acoesOrdinarias = new AcaoOrdinaria();

acoesOrdinarias.addInterressado(new Corretor());
acoesOrdinarias.addInterressado(new Banco());

acoesOrdinarias.setValor(120.10);
acoesOrdinarias.setValor(121.00);
acoesOrdinarias.setValor(120.30);
```

4.8 Padrão State

O padrão State permite que o comportamento de um objeto seja alterado, de acordo com o seu estado. Por exemplo, em um objeto Pedido, o método cancelar pode variar de acordo com o estado do objeto. Se ele estiver como estado Entregue, o cancelamento implicará no processo de devolução de produtos, ou de dinheiro, ou cancelamento de nota fiscal e etc. Agora, se o estado for "Em análise de Crédito", o cancelamento acarretará apenas no cancelamento da análise e do pedido.

Para entender o funcionamento do padrão, veja o diagrama abaixo:



Observe o objeto Context, que terá seu comportamento condicionado ao estado. Esse objeto mantém uma referência ao estado atual, representado pela interface State. Essa interface terá todos os métodos, que possuem seu comportamento, condicionados ao estado. Para cada estado é definido um objeto com os métodos implementados para aquele estado.

Para exemplificar o padrão, suponha um sistema de banco que, de acordo com o estado da conta do cliente, conceda o empréstimo ou não.

Para padronizar os métodos, que serão diferentes de acordo com cada estado da conta, defina a seguinte interface:

```
1  public interface StatusConta {
2          public boolean emprestimo(double valor);
3  }
```

Agora defina a conta da seguinte forma:

```
1  public class Conta {
2          private int nr;
3          private double saldo;
4          private StatusConta status;
5
6          public int getNr() {
7                  return nr;
8          }
9
10         public void setNr(int nr) {
11                 this.nr = nr;
12         }
13
14         public double getSaldo() {
15                 return saldo;
16         }
17
18         public void setSaldo(double saldo) {
19                 this.saldo = saldo;
20         }
21
22         public void setStatus(StatusConta status) {
23                 this.status = status;
24         }
25
26         public void saque(double valor){
27                 this.saldo -= valor;
28                 if(this.saldo < 0)
29                         this.status = new ContaNegativa(this);
30         }
31
32         public void deposito(double valor){
33                 this.saldo += valor;
34                 if(this.saldo > 0)
35                 this.status = new ContaPositiva(this);
36         }
37
38         public boolean emprestimo(double valor){
```

```
39                  if(this.status.emprestimo(valor)){
40                          this.saldo += valor;
41                          return true;
42                  }
43                  else
44                          return false;
45          }
46  }
```

Agora você precisa definir os estados da conta. Como a finalidade é apenas didática, as contas poderão ter dois estados: contas positivas, veja:

```
1   public class ContaPositiva implements StatusConta {
2           private Conta conta;
3                   public ContaPositiva(Conta conta){
4                   this.conta = conta;
5           }
6
7           @Override
8           public boolean emprestimo(double valor) {
9                   if(conta.getSaldo() > 100 && valor < 1000)
10                          return true;
11                  else if(conta.getSaldo() > 1000)
12                          return true;
13                  else
14                          return false;
15          }
16  }
```

E contas negativas:

```
1   public class ContaNegativa implements StatusConta {
2           private Conta conta;
3                   public ContaNegativa(Conta conta) {
4                   this.conta = conta;
5           }
6
7           @Override
8           public boolean emprestimo(double valor) {
9                   if(conta.getSaldo() < -100 && valor < 1000)
10                          return true;
11                  else
12                          return false;
13          }
14  }
```

As classes podem ser utilizadas da seguinte forma:

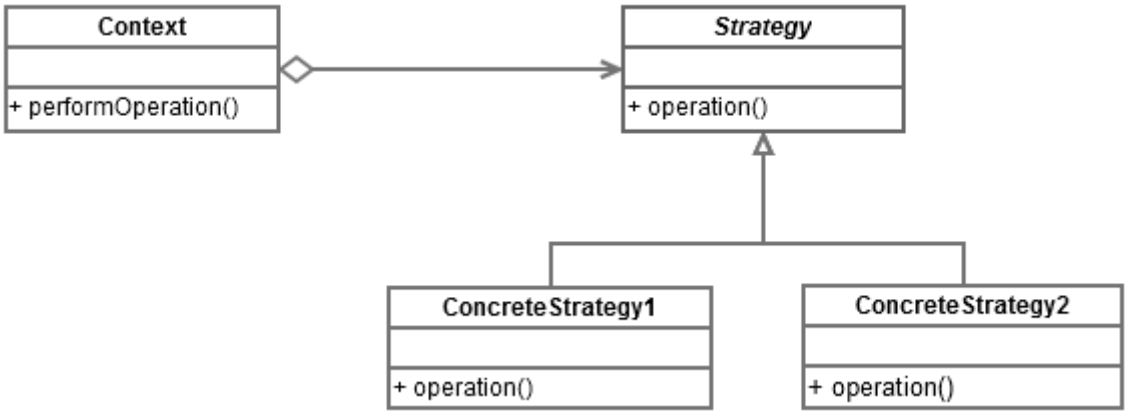
```
1   Conta conta = new Conta();
2   conta.setNr(1120);
3   conta.setSaldo(200);
4   conta.deposito(100);
5
6   if(conta.emprestimo(100))
7           System.out.println("Empréstimo concedido");
8   else
9           System.out.println("Empréstimo negado");
```

4.9 Padrão Strategy

O padrão Strategy consiste no encapsulamento de algoritmos relacionados, mas variados, que são utilizados para a resolução de um problema. Fica a cargo do objeto ou do cliente selecionar a melhor opção para resolver o problema. Por exemplo, se o problema for para recuperar o estado de um objeto, poderia ser implementado um algoritmo para cada local onde a informação pudesse ser recuperada: o banco de dado, a memória e etc.

Para entender o funcionamento do padrão, veja o diagrama abaixo:

Note que a estrutura do padrão Strategy é bem parecida com a estrutura do padrão State. A diferença é que no padrão Strategy, o método a ser executado é definido de acordo com a melhor opção para resolver o problema, e não pelo estado do objeto.



Para exemplificar o padrão, suponha um sistema onde seja necessário ordenar uma lista. Você já sabe que existem vários algoritmos de ordenação, então deverá escolher o algoritmo mais apropriado para a situação.

Para padronizar as diversas implementações dos algoritmos, defina a seguinte interface:

```
1 public interface SortStrategy {
2         public void Sort(List<String> lista);
3 }
```

Agora você pode implementar a interface para cada estratégia de ordenação, como a QuickSort:

```
1 public class QuickSort implements SortStrategy {
2
3         @Override
4         public void Sort(List<String> lista) {
5                 //implementação
6         }
7
8 }
```

A BubbleSort:

```
1 public class BubbleSort implements SortStrategy {
2
3         @Override
4         public void Sort(List<String> lista) {
5                 //implementação
6         }
7
8 }
```

A MergeSort:

```
1 public class MergeSort implements SortStrategy {
2
3         @Override
4         public void Sort(List<String> lista) {
5                 //implementação
6         }
7
8 }
```

Essas classes podem ser utilizadas da seguinte forma:

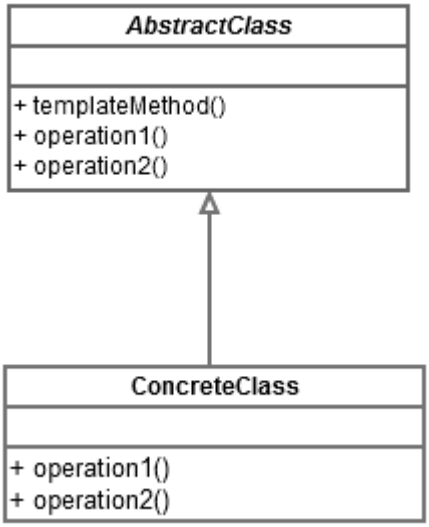
```
1  List<String> lista = new ArrayList<String>();
2
3  lista.add("Samual");
4  lista.add("Carlos");
5  lista.add("Maria");
6  lista.add("Ana");
7  lista.add("João");
8
9  MergeSort merge = new MergeSort();
10
11 merge.Sort(lista);
```

4.10 Padrão Template Method

O conceito do padrão Template Method é muito simples – e você irá perceber que usa esse padrão o tempo todo. Sempre que você cria uma classe-Pai e deixa para as classes-Filhas um ou mais métodos para serem implementados, você está usando a essência do padrão Template Method. Ele formaliza a ideia de definir uma ordem em que determinados passos devem ser realizados, além de permitir que esses passos possam ser realizados de formas diferentes. Em outras palavras, se a sua classe-base é uma classe abstrata, como muitas vezes acontece com os padrões de projeto, você está usando uma forma simples do padrão Template Method.

Por exemplo, em uma classe que monta relatórios, a estrutura padrão do relatório é definida em uma classe abstrata, com métodos para cabeçalho, conteúdo e rodapé; e em cada classe descendente as operações serão as mesmas, mas podendo variar o modo como as operações serão realizadas. Essa é a ideia do padrão: ter uma estrutura definida e permitir que cada passo possa ser implementado de maneira diferente.

Para entender o funcionamento do padrão, veja o diagrama abaixo.



Note que o padrão só possui duas classes. A classe abstrata terá o método modelo (Template Method), e os demais métodos abstratos serão definidos nas subclasses.

Para exemplificar o padrão, suponha um sistema de framework para jogos. A classe principal de cada jogo, que utilizar o framework, terá que importar a classe Game, que terá a seguinte estrutura:

```
1  public abstract class Game {
2          protected boolean active = false;
3
4          private void run(){
5                  active = true;
6                  load();
7                  while(active){
8                          update();
9                          print();
10                 }
11                 onload();
12         }
13
14         public void start(){
15                 run();
16         }
17
18         protected abstract void load();
19         protected abstract void update();
20         protected abstract void print();
21         protected abstract void onload();
22 }
```

Observe que a classe possui um método modelo (run), que não pode ser modificado pelas subclasses. Elas apenas implementam os métodos abstratos:

```
1  public class Jogo1 extends Game {
2
3          @Override
4          protected void load() {
5          // implementação
6          }
7
8          @Override
9          protected void update() {
10         // implementação
11         }
12
13         @Override
14         protected void print() {
15         // implementação
16         }
17
18         @Override
19         protected void onload() {
20         // implementação
21         }
22 }
```

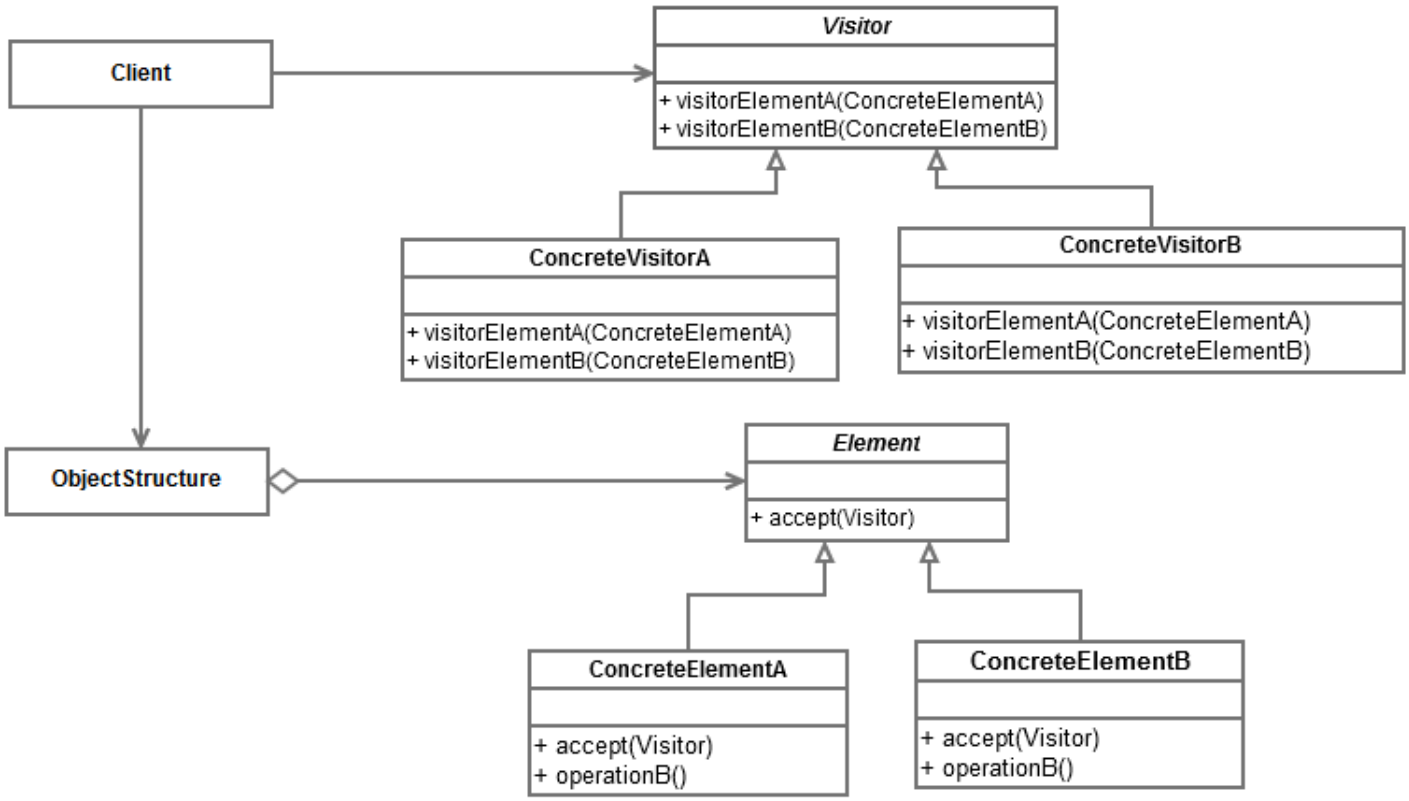
A classe pode ser utilizada da seguinte forma:

```
1  Game jogo = new Jogo1();
2  jogo.start();
```

4.11 Padrão Visitor

O padrão Visitor permite que sejam acrescentadas novas funcionalidades a um objeto sem a necessidade de alterá-lo, já que nele é criada uma classe externa para realizar a operação.

Para entender o funcionamento do padrão, veja o diagrama a seguir:



Note que há uma interface que define os métodos de visita para cada objeto, e que esses métodos são definidos nas subclasses. Também há uma interface Element, que define os objetos que o Visitor opera, além de um objeto que agrega os elementos.

Para exemplificar o padrão, suponha que você está desenvolvendo um sistema no qual os funcionários devam ser classificados de acordo com os cargos. Para padronizar os funcionários, você deve criar a seguintre classe abstrata:

```
1  public abstract class Funcionario {
2          private int num;
3          private String nome;
4          private double salario;
5
6          public int getNum() {
7                  return num;
8          }
9
10         public void setNum(int num) {
11                 this.num = num;
12         }
13
14         public String getNome() {
15                 return nome;
16         }
17
18         public void setNome(String nome) {
19                 this.nome = nome;
20         }
21
22         public double getSalario() {
23                 return salario;
24         }
25
26         public void setSalario(double salario) {
27                 this.salario = salario;
28         }
29
30 }
```

Que será herdada pela classe Chefe:

```
1  public class Chefe extends Funcionario {
2          private int senha;
3
4          public int getSenha() {
5                  return senha;
6          }
7
8          public void setSenha(int senha) {
9                  this.senha = senha;
10         }
11 }
```

E Atendente:

```
1  public class Atendente extends Funcionario {
2          private String ramal;
3
4          public String getRamal() {
5                  return ramal;
6          }
7
8          public void setRamal(String ramal) {
9                  this.ramal = ramal;
10         }
11 }
```

Todo funcionário está vinculado a um departamento. Você pode modelar essa agregação da seguinte forma:

```
1  public class Departamento {
2          private String nome;
3          private String telefone;
4          private List<Funcionario> funcionarios;
5
```

```
6           public Departamento(){
7                   this.funcionarios = new ArrayList<Funcionario>();
8           }
9
10          public String getNome() {
11                  return nome;
12          }
13
14          public void setNome(String nome) {
15                  this.nome = nome;
16          }
17
18          public String getTelefone() {
19                  return telefone;
20          }
21
22          public void setTelefone(String telefone) {
23                  this.telefone = telefone;
24          }
25
26          public void add(Funcionario funcionario){
27                  this.funcionarios.add(funcionario);
28          }
29
30          public List<Funcionario> getFuncionarios() {
31                  return funcionarios;
32          }
33
34  }
```

De tempos em tempos, os salários são reajustados e outras alterações podem ser realizadas em funcionários.

Os reajustes, ou qualquer outra tarefa de atualização dos funcionários, podem ser implementadas em classes especializadas. Para padronizar, pode ser criada uma interface:

```
1   public interface AtualizadorFuncionario {
2           public void atualiza(Chefe chefe);
3           public void atualiza(Atendente atendente);
4   }
```

E implementada a classe que irá atualizar um item específico do funcionário:

```
1   public class AtualizaSalario implements AtualizadorFuncionario {
2           private double reajusteChefe;
3           private double reajusteAtendente;
4
5           public double getReajusteChefe() {
6                   return reajusteChefe;
7           }
8
9           public void setReajusteChefe(double reajusteChefe) {
10                  this.reajusteChefe = reajusteChefe;
11          }
12
13          public double getReajusteAtendente() {
14                  return reajusteAtendente;
15          }
16
17          public void setReajusteAtendente(double reajusteAtendente) {
18                  this.reajusteAtendente = reajusteAtendente;
19          }
20
21          @Override
22          public void atualiza(Chefe chefe) {
23                  chefe.setSalario(chefe.getSalario() * this.reajusteChefe);
24          }
25
26          @Override
27          public void atualiza(Atendente atendente) {
28                  atendente.setSalario(atendente.getSalario() * this.reajusteAtendente);
29          }
30
31  }
```

Agora, suponha que seja preciso aplicar o atualizador em todos os funcionários, de acordo com os departamentos de uma lista. O código ficaria assim:

```
Chefe chefeMkv = new Chefe();
chefeMkv.setNum(1);
chefeMkv.setNome("Carlos");
chefeMkv.setSalario(4000.00);
chefeMkv.setSenha(123);

Atendente atendMkv1 = new Atendente();
atendMkv1.setNum(10);
atendMkv1.setNome("Maria");
atendMkv1.setRamal("349");
atendMkv1.setSalario(450.00);

Atendente atendMkv2 = new Atendente();
atendMkv2.setNum(11);
atendMkv2.setNome("João");
atendMkv2.setRamal("349");
atendMkv2.setSalario(450.00);

Departamento mkv = new Departamento();
mkv.add(chefeMkv);
mkv.add(atendMkv1);
mkv.add(atendMkv2);

Chefe chefeVendas = new Chefe();
chefeVendas.setNum(2);
chefeVendas.setNome("Henrique");
chefeVendas.setSalario(4000.00);
chefeVendas.setSenha(456);

Atendente atendVen1 = new Atendente();
atendVen1.setNum(12);
atendVen1.setNome("José");
atendVen1.setRamal("455");
atendVen1.setSalario(450.00);

Atendente atendVen2 = new Atendente();
atendVen2.setNum(13);
atendVen2.setNome("Ana");
atendVen2.setRamal("455");
atendVen2.setSalario(450.00);

Departamento vendas = new Departamento();
vendas.add(chefeVendas);
vendas.add(atendVen1);
vendas.add(atendVen2);

List<Departamento> departamentos = new ArrayList<Departamento>();
departamentos.add(mkv);
departamentos.add(vendas);

AtualizadorFuncionario atualizar = new AtualizaSalario();

for ( Departamento d : departamentos ) {
        for ( Funcionario f : d.getFuncionarios()) {
                if(f instanceof Chefe) {
                        Chefe c = (Chefe)f;
                        atualizar.atualiza(c);
                }else if(f instanceof Atendente){
                        Atendente a = (Atendente)f;
                        atualizar.atualiza(a);
                }
        }
}
```

A lista de departamentos poderia vir de outro local. Porém, note que se os funcionários forem atualizados dessa forma, poderia haver o problema de a listagem dos funcionários estar restrita; e, no código acima, é preciso testar o tipo de cada funcionário para selecionar o método correto de atualização. Se outro tipo de funcionário for adicionado, será necessário modificar a classe de atualização e todas as demais classes onde ela é utilizada.

Para solucionar esse problema, é possível passar os atualizadores para dentro da classe Departa-

mento e Funcionario, para que eles próprios chamem o método correto. Para padronizar isso, crie uma interface com o código abaixo:

```
public interface Atualiza {
        public void accept(AtualizadorFuncionario atualizador);
}
```

Para garantir que todas as subclasses de Funcionario utilizem essa interface, você deve implementá-la na classe Funcionario:

```
public abstract class Funcionario implements Atualiza {
        //getters e setters

}
```

Dessa forma, o método será implementado pelas subclasses:

```
public class Chefe extends Funcionario {
        private int senha;

        public int getSenha() {
                return senha;
        }

        public void setSenha(int senha) {
                this.senha = senha;
        }

        @Override
        public void accept(AtualizadorFuncionario atualizador) {
                atualizador.atualiza(this);
        }

}
```

E:

```
public class Atendente extends Funcionario {
        private String ramal;

        public String getRamal() {
                return ramal;
        }

        public void setRamal(String ramal) {
                this.ramal = ramal;
        }

        @Override
        public void accept(AtualizadorFuncionario atualizador) {
                atualizador.atualiza(this);
        }
}
```

Também implemente a interface na classe Departamento:

```
public class Departamento implements Atualiza {
        private String nome;
        private String telefone;
        private List<Funcionario> funcionarios;

        // getters e setters

        @Override
        public void accept(AtualizadorFuncionario atualizador) {
                for(Funcionario fun : funcionarios)
                        fun.accept(atualizador);
        }

}
```

Para aplicar um atualizador em todos os funcionários é mais simples. Por exemplo:

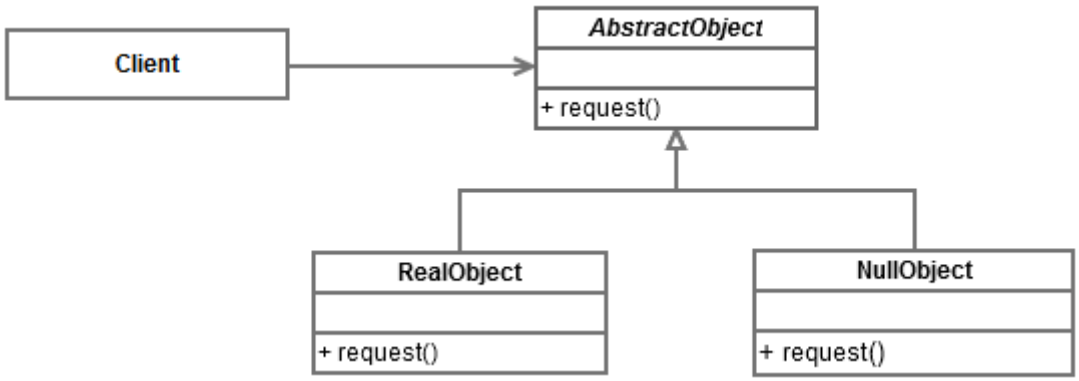
```
AtualizadorFuncionario atualizar = new AtualizaSalario();

for ( Departamento d : departamentos ) {
        d.accept(atualizar);
}
```

4.12 Padrão Null Object

O padrão Null Object descreve o uso de objeto quando o comportamento é nulo. Nesse caso, em vez de utilizar um objeto nulo, utilize um objeto cuja interface implemente o comportamento esperado, mas cujo método nada faz.

Esse padrão é muito útil quando aplicado com outros padrões. Por exemplo, no modelo do padrão Strategy foi demonstrado como criar estratégias para ordenar uma lista, mas para garantir que tudo esteja funcionando corretamente, você também pode querer criar a mais simples das estratégias, que poderia ser útil mais tarde. Nesse caso, a estratégia mais simples seria nenhuma estratégia. Ou seja: não fazer nada. No entanto, o padrão Strategy requer que os objetos implementem a interface da estratégia. É nesse ponto que o padrão Null Object se torna útil, já que ele é utilizado para implementar a estratégia que não faça nada.

Para entender o funcionamento do padrão, veja o diagrama abaixo:



Note que, no padrão, há uma classe abstrata que define a interface para todos os objetos. Entre esses objetos, que herdam a classe abstrata, está o Null Object. Repare que ele herda a classe abstrata, podendo ser utilizado em qualquer local que necessite de um objeto com a estrutura da classe abstrata, o que não seria possível se o valor fosse realmente um nulo.

Para exemplificar o padrão, suponha um sistema em que seja necessário ordenar uma lista. Nele, o usuário informa o tipo de ordenação que quer utilizar e ela é retornada. Para padronizar a classe de ordenação, crie a interface a seguir:

```
1  public interface SortList {
2          public void sort(List<Integer> list);
3  }
```

A ser implementada nos tipos de ordenação que a aplicação irá disponibilizar. Como QuickSort:

```
1  public class QuickSort implements SortList {
2
3          @Override
4          public void sort(List<Integer> list) {
5                  if (list == null || list.size() == 0){
6                          return;
7                  }
8                  int ini = 0, fim = 0;
9                  for(int ele : list){
10                         if(ele < ini)
11                         ini = ele;
12                         if(ele > fim)
13                         fim = ele;
14                 }
15
16                 quicksort(list, ini, fim);
17         }
18
19         private void quicksort(List<Integer> list, int ini, int fim){
20                 int i = ini, j = fim;
21                 int pivot = list.get(ini + (fim-ini)/2);
22                 while (i <= j) {
```

```
                        while (list.get(i) < pivot)
                        i++;
                        while (list.get(j) > pivot)
                        j--;

                        if (i <= j) {
                                exchange(list, i, j);
                                i++;
                                j--;
                        }
                }

                if (ini < j)
                        quicksort(list, ini, j);
                if (i < fim)
                        quicksort(list, i, fim);
        }

        private void exchange(List<Integer> list, int i, int j) {
                int temp = list.get(i);
                list.add(i, list.get(j));
                list.add(j, temp);
        }
}
```

E BubbleSort:

```
public class BubbleSort implements SortList {

        @Override
                public void sort(List<Integer> list) {
                bubbleSort(list);
        }

        public static void bubbleSort(List<Integer> list) {
                boolean sorted = false;

                for (int top = list.size() - 1; top > 0 && !sorted; top--) {
                        sorted = true;
                        for (int i = 0; i < top; i++) {
                                if (list.get(i) > list.get(i+1) ) {
                                        sorted = false;

                                        int temp = list.get(i);
                                        list.add(i, list.get(i+1));
                                        list.add(i+1, temp);
                                }
                        }
                }
        }
}
```

E até no objeto nulo:

```
public class NullSort implements SortList {

        @Override
        public void sort(List<Integer> list) {
                // Não faz nada
        }

}
```

Implemente uma fábrica que será responsável pela criação dos objetos de ordenação. O código ficará da seguinte forma:

```
public class FactorySort {
        public SortList getSort(String sortName){
                try {
                        if(Class.forName(sortName) != null)
                                return (SortList) Class.forName(sortName).getConstructor(String.
                                    class).newInstance();
                        else
                                return new NullSort();
                } catch (Exception e) {
                        e.printStackTrace();
```

```
10                          return new NullSort();
11                  }
12          }
13  }
```

Se o nome de um algoritmo de ordenação for informado, e for implementado na aplicação, o seu objeto será retornado. Caso contrário (e em caso de algum erro), será retornado o objeto nulo.

Essas classes podem ser utilizadas da seguinte forma:

```
1  List<Integer> list = new ArrayList<Integer>();
2  list.add(1);
3  list.add(30);
4  list.add(13);
5  list.add(11);
6  list.add(44);
7  list.add(3);
8
9  FactorySort factory = new FactorySort();
10
11 SortList ordernar = factory.getSort("MergeSort");
12 ordernar.sort(list);
```

Note que, como o algoritmo de ordenação Merge não foi informado no caso acima, será retornado o objeto nulo e, como ele possui a mesma interface de um objeto normal, nenhuma execução será lançada.