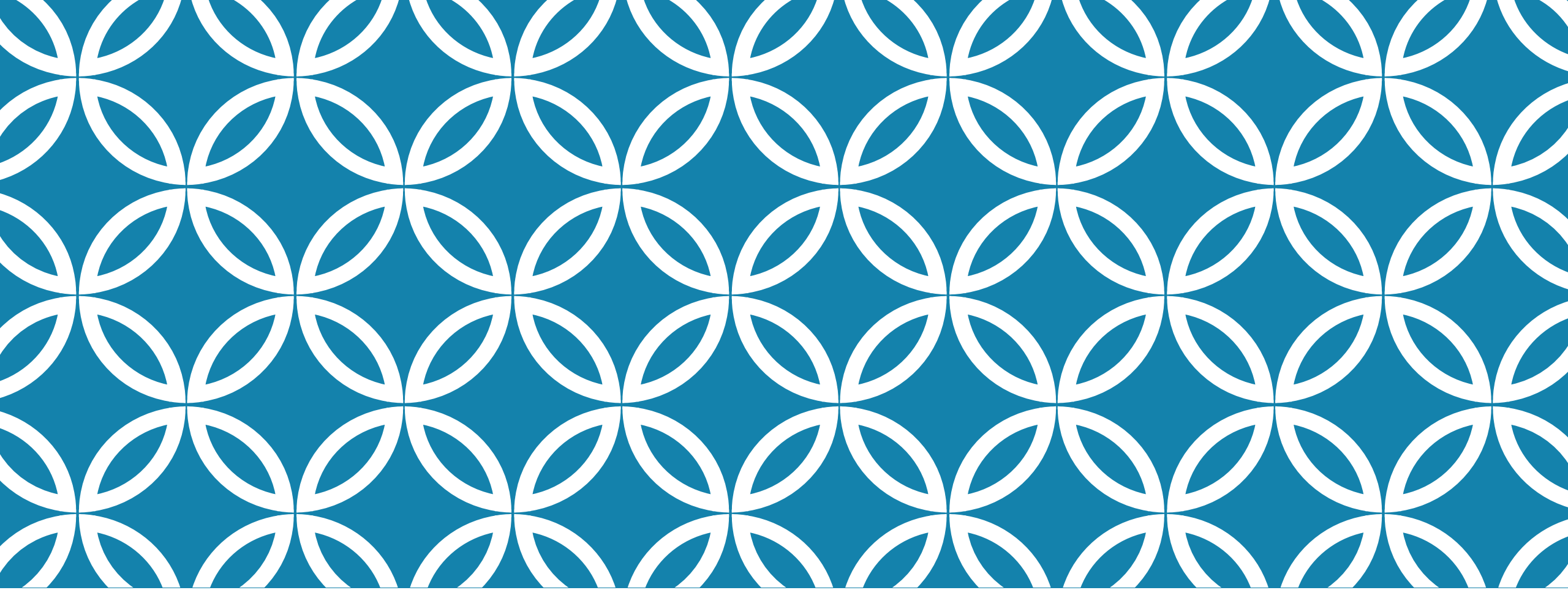


PADRÕES DE PROJETOS

Julho/2019

SOBRE MIM...

30 anos (12 de TI)

Técnologo em Processamento de Dados pela Fatec (2009)

Mestre em Engenharia de Software pela USP (2014)

Doutorando em Engenharia de Software na USP

MBA em Gestão de Pessoas na Uniara

~~Desenvolvedor/Coordenador/Gestor/Instrutor e Mentor Boot Camp na EDS/HP/HP~~E/DXC

Líder de Soluções na Embraer

Professor na Uniara (SI, Eng Computação, Eng Agro, Desen Jogos)

Professor EAD (Analise e Desenvolvimento de Sistemas)

17 Certificações (Java, Oracle, .NET, Itil, Iso, Cobit, BPM, Scrum, Six Sigma...)

SOBRE VOCÊS

- Nome
- Idade
- Formação
- Experiência Profissional

ROTEIRO

Revisão UML

Introdução: Por que padrões?

Introdução a Design Patterns: Padrões do GoF

- Padrões Criacionais
- Padrões Estruturais
- Padrões Comportamentais

Injeção de dependência

CONTATO E EXEMPLOS

E-mail : felipedallilo@hotmail.com

Git com exemplos das aulas: **felipeddallilo/padroesdeprojetos**

AVALIAÇÃO

Crie um repositório no github: **nomecompleto_padroesprojeto.**

Crie um projeto para cada exercício proposto (com o título do projeto igual ao do slide).

Implemente os exercícios da sala e suba no repositório.

Ao finalizar todos os exercícios, comente no meu github com o endereço do seu repositório

PREMISSAS

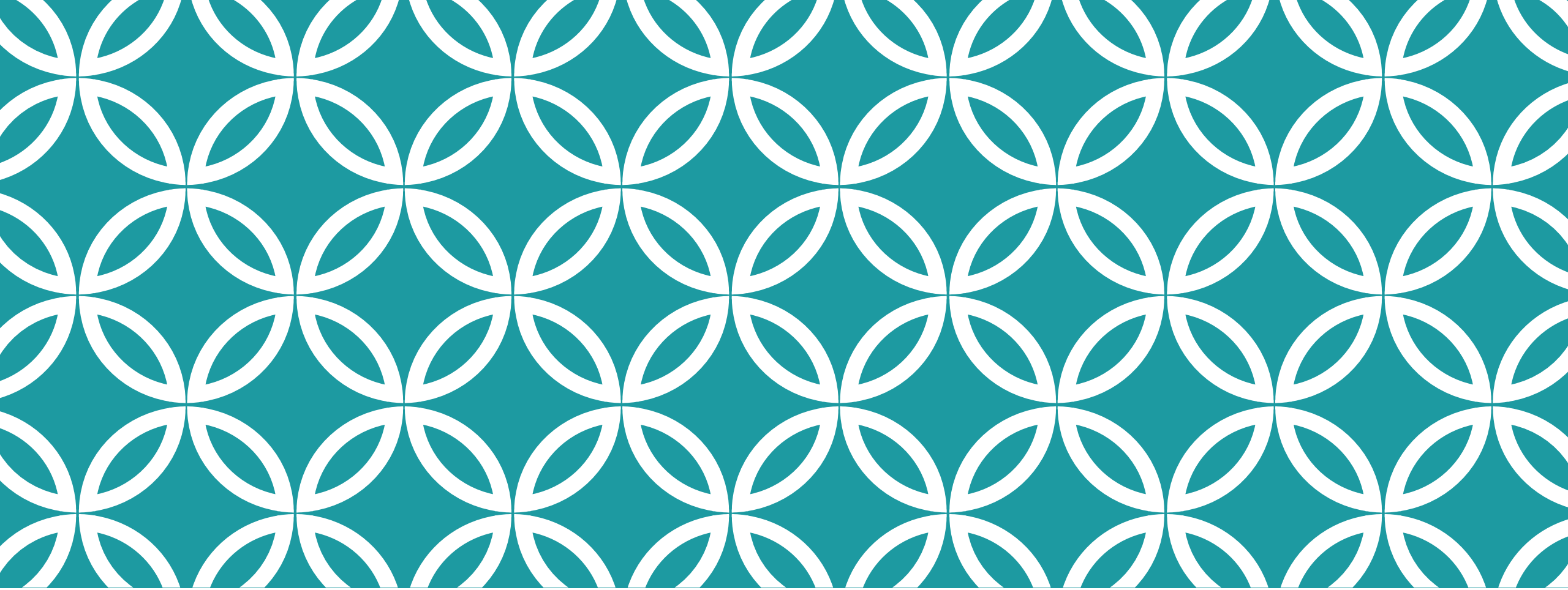
Eclipse/Netbeans

Usuário no Github

DICAS

Efetue o Download da ferramenta StarUML (ou qualquer similar para diagrama de classes) e modele a solução no padrão antes de desenvolver.

Suba no Git a modelagem junto com o projeto.



REVISÃO

UML

A UML - Linguagem de Modelagem Unificada (do inglês, UML - Unified Modeling Language) é uma linguagem-padrão para a elaboração da estrutura de projetos de software.

Ela poderá ser empregada para:

- Visualização
- Especificação
- Construção
- Documentação de artefatos

Permite representar um sistema de forma padronizada (com intuito de facilitar a compreensão pré-implementação).

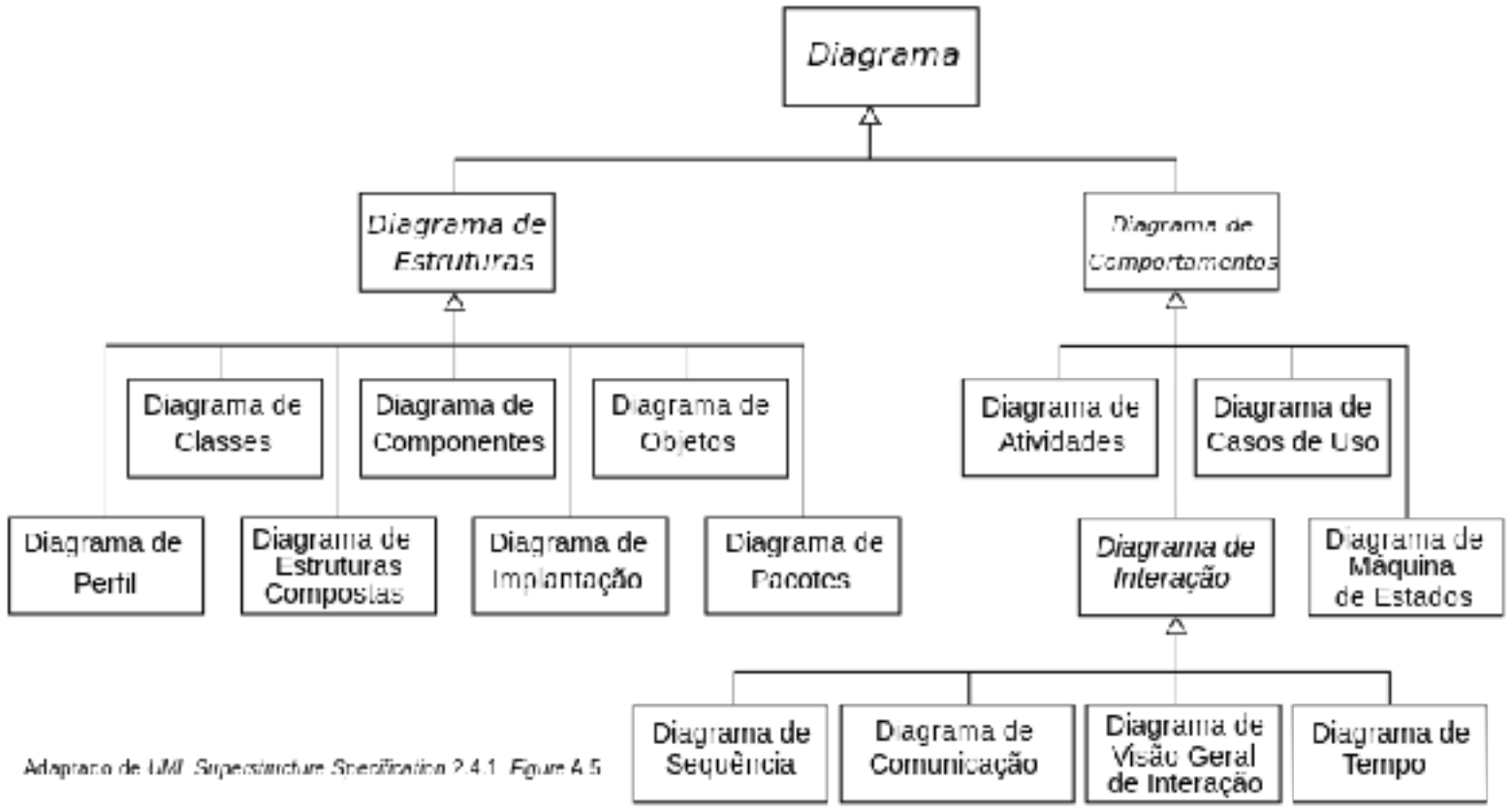
É uma linguagem muito expressiva, abrangendo diversas visões para o desenvolvimento e implantação de sistemas.

UML

Não é uma metodologia de desenvolvimento, apenas auxilia a visualizar seu desenho e a comunicação entre os objetos.

Basicamente, a UML permite que desenvolvedores visualizem os produtos de seus trabalhos em diagramas padronizados.

UML – VISÃO GERAL



Adaptado de *UML Superstructure Specification* 2.4.1, *Figure* 4.5

UML

Diagramas estruturais - Tratam o aspecto estrutural tanto do ponto de vista do sistema quanto das classes. Existem para a representação de seu esqueleto e estruturas. No UML existem 6 diagramas estruturais.

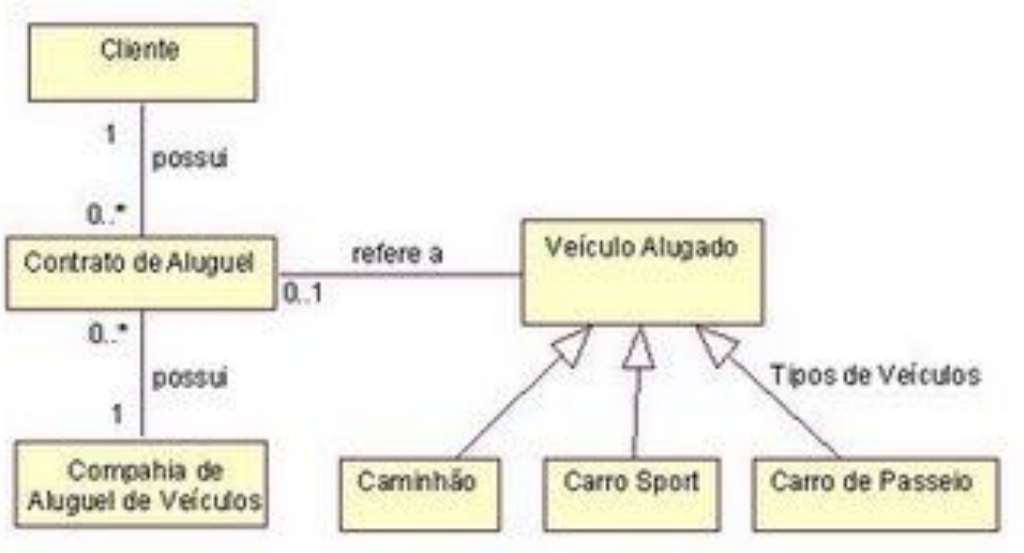
Diagramas comportamentais - Utilizado para visualizar, especificar, construir e documentar aspectos dinâmicos de um devido sistema, como representação das suas partes que passam por alteração. No UML existem 5 diagramas comportamentais

UML – DIAGRAMAS ESTRUTURAIS

- Diagrama de Classes
- Diagrama de Objetos
- Diagrama de Componentes
- Diagrama de Implementação ou instalação
- Diagrama de Pacotes
- Diagrama de Estrutura Composta

UML – DIAGRAMAS ESTRUTURAIS

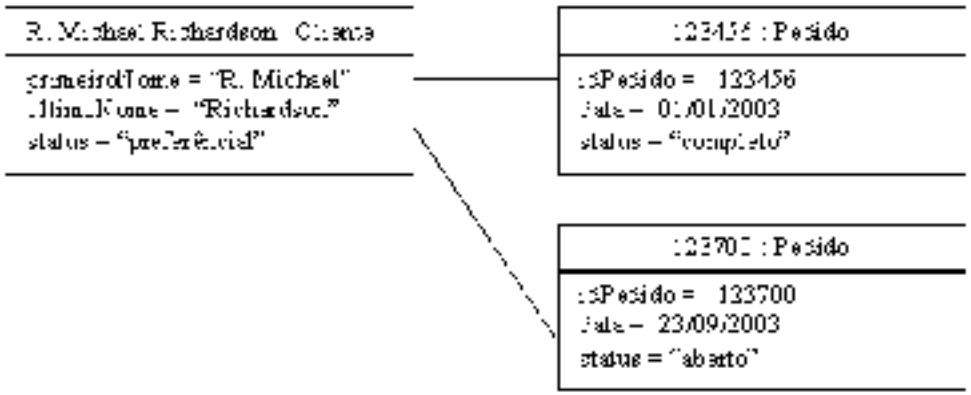
- **Diagrama de Classes**
- Diagrama de Objetos
- Diagrama de Componentes
- Diagrama de Implementação ou instalação
- Diagrama de Pacotes
- Diagrama de Estrutura Composta



- Considerado como o diagrama mais importante pois serve de referência/ligação com todos os demais.

UML – DIAGRAMAS ESTRUTURAIS

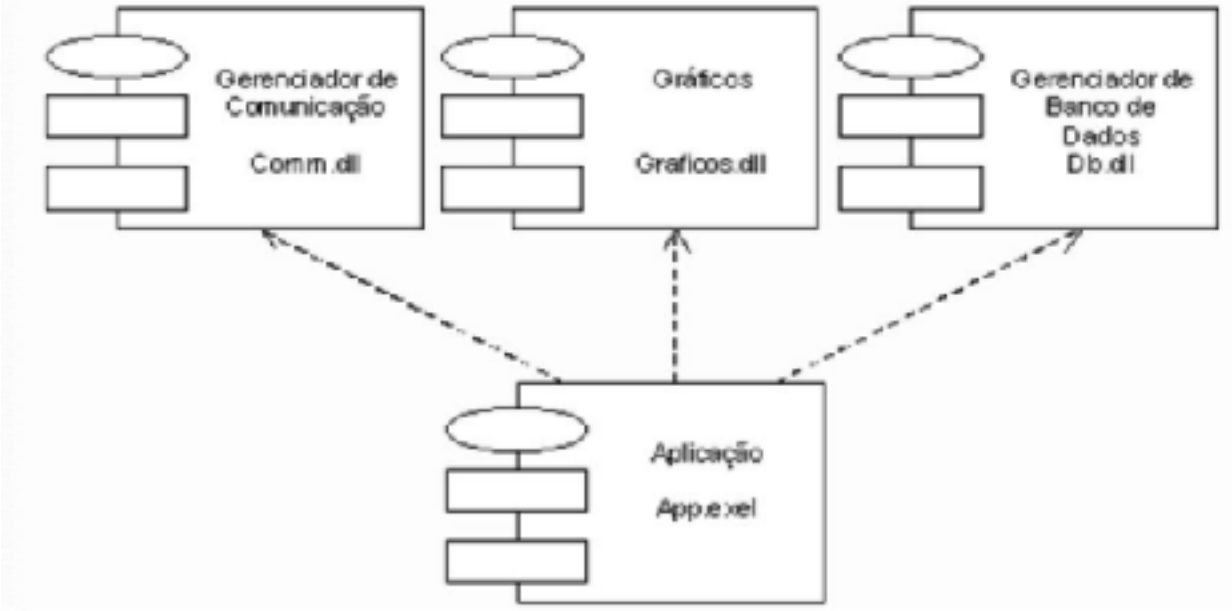
- Diagrama de Classes
- **Diagrama de Objetos**
- Diagrama de Componentes
- Diagrama de Implementação ou instalação
- Diagrama de Pacotes
- Diagrama de Estrutura Composta



- Representa os objetos gerados pelo diagrama de classe em um determinado momento de execução.

UML – DIAGRAMAS ESTRUTURAIS

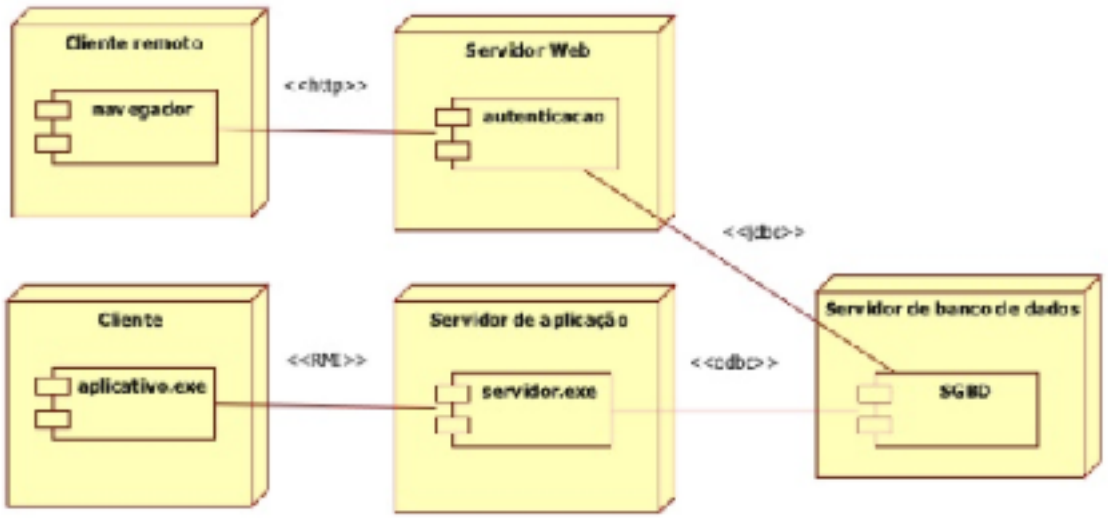
- Diagrama de Classes
- Diagrama de Objetos
- **Diagrama de Componentes**
- Diagrama de Implementação ou instalação
- Diagrama de Pacotes
- Diagrama de Estrutura Composta



- Este diagrama mostra a composição e relacionamento de um componente

UML – DIAGRAMAS ESTRUTURAIS

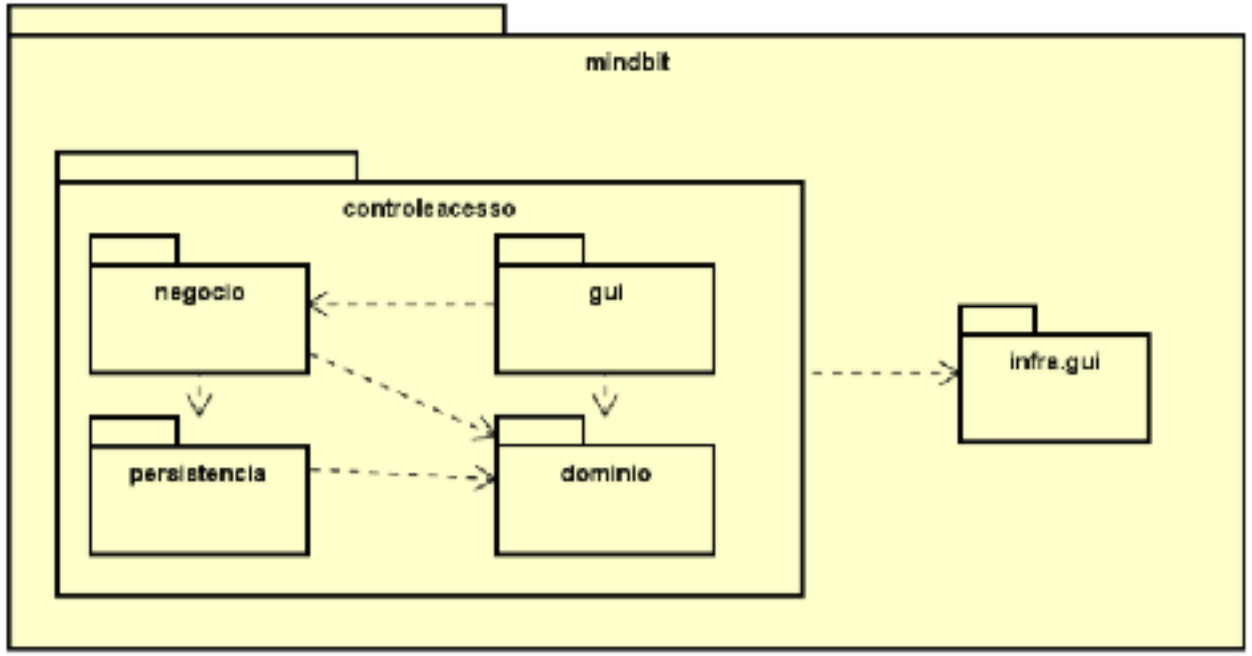
- Diagrama de Classes
- Diagrama de Objetos
- Diagrama de Componentes
- **Diagrama de Implementação ou instalação**
- Diagrama de Pacotes
- Diagrama de Estrutura Composta



- Consiste na organização do conjunto de elementos de um sistema para a sua execução

UML – DIAGRAMAS ESTRUTURAIS

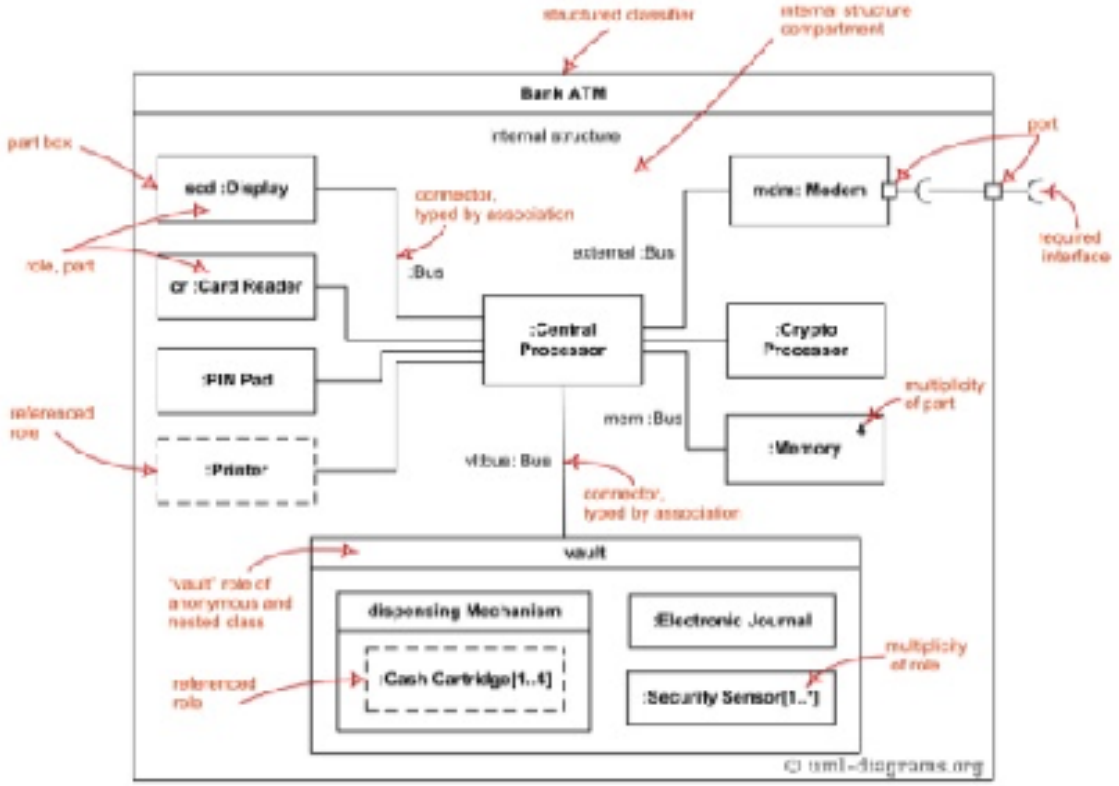
- Diagrama de Classes
- Diagrama de Objetos
- Diagrama de Componentes
- Diagrama de Implementação ou instalação
- **Diagrama de Pacotes**
- Diagrama de Estrutura Composta



- Diagrama representa os Pacotes do sistema e o relacionamento entre os mesmos.

UML – DIAGRAMAS ESTRUTURAIS

- Diagrama de Classes
- Diagrama de Objetos
- Diagrama de Componentes
- Diagrama de Implementação ou instalação
- Diagrama de Pacotes
- **Diagrama de Estrutura Composta**



- É utilizado para modelar Colaborações, de entidades interpretadas por instâncias que coopera... ...e para executar uma determinada função.

UML – DIAGRAMAS COMPORTAMENTAIS

Diagrama de Caso de Uso

Diagram de Sequencia

Diagrama de Comunicação

Diagrama de Estados

Diagrama de Atividade

UML – DIAGRAMAS COMPORTAMENTAIS

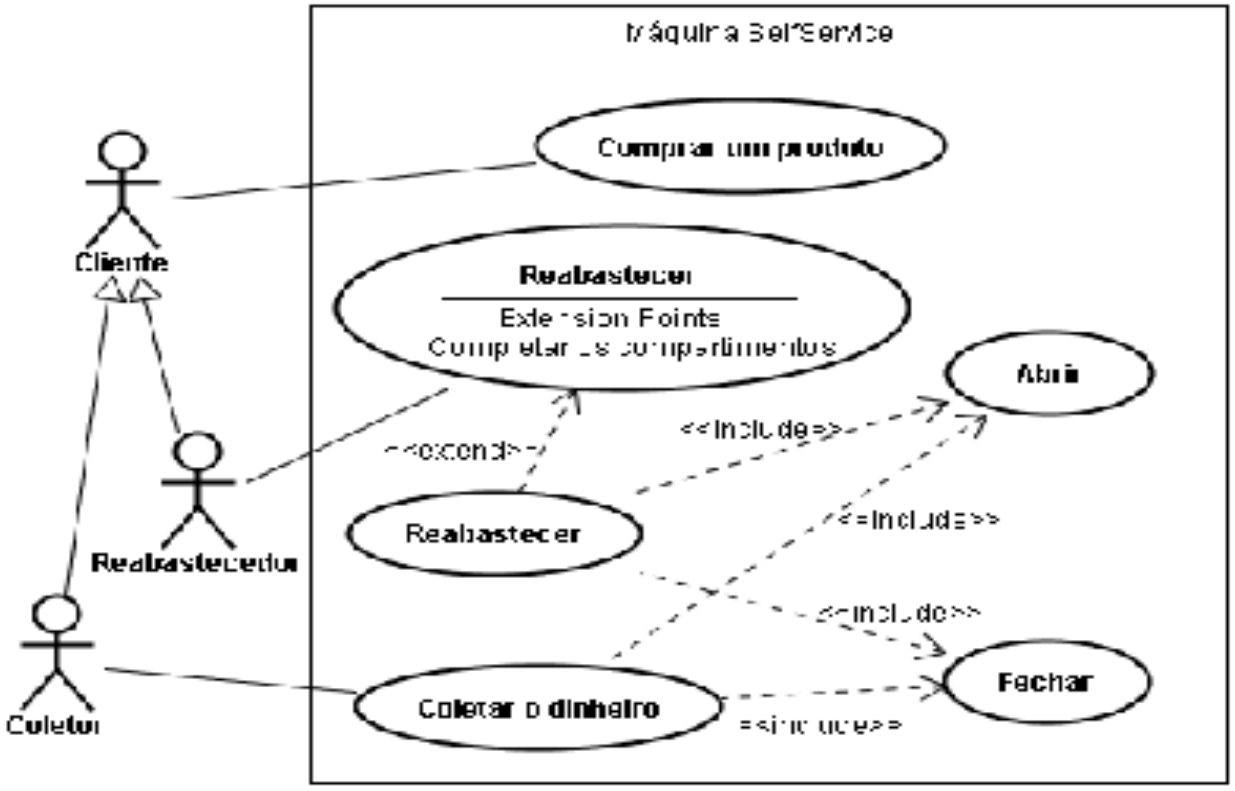
Diagrama de Caso de Uso

Diagram de Sequencia

Diagrama de Comunicação

Diagrama de Estados

Diagrama de Atividade



-Representa a relação entre os atores e os casos de uso de um sistema.

UML – DIAGRAMAS COMPORTAMENTAIS

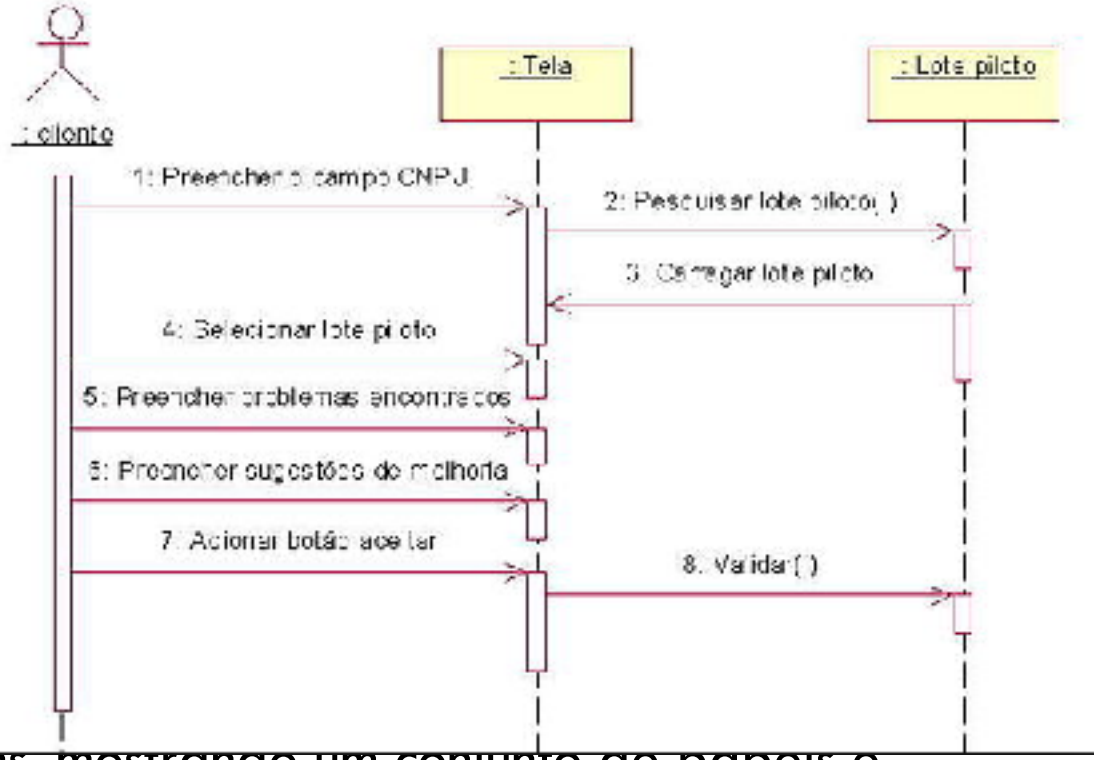
Diagrama de Caso de Uso

Diagram de Sequencia

Diagrama de Comunicação

Diagrama de Estados

Diagrama de Atividade



- Da ênfase à ordenação temporal de mensagens, mostrando um conjunto de papéis e as mensagens enviadas e recebidas pelos papéis, ilustrando uma visão dinâmica de um sistema.

UML – DIAGRAMAS COMPORTAMENTAIS

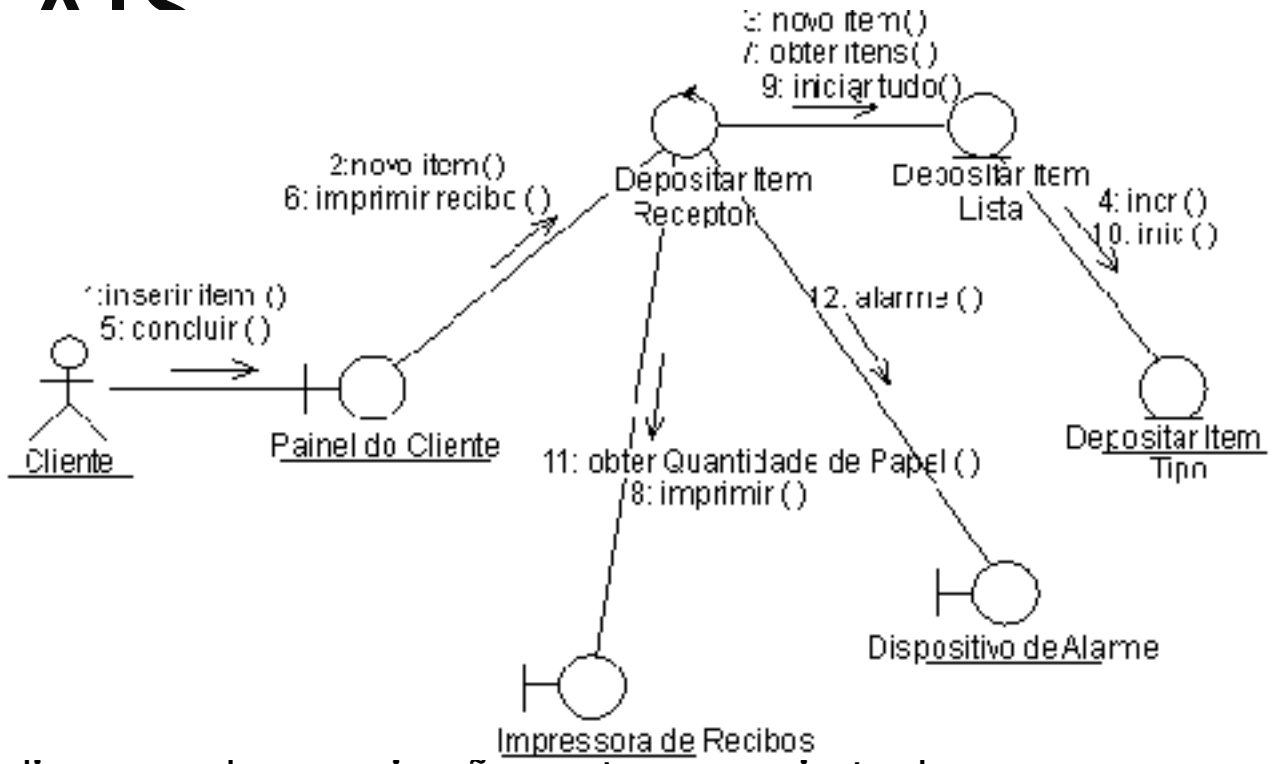
Diagrama de Caso de Uso

Diagram de Sequencia

Diagrama de Comunicação

Diagrama de Estados

Diagrama de Atividade



- Um diagrama de comunicação é um diagrama de
dos objetos que enviam e recebem mensagens. Um diagrama de comunicação mostra um conjunto de papéis, as conexões existentes entre esses papéis e as mensagens enviadas e recebidas pelas instâncias que representam os papéis.

UML – DIAGRAMAS COMPORTAMENTAIS

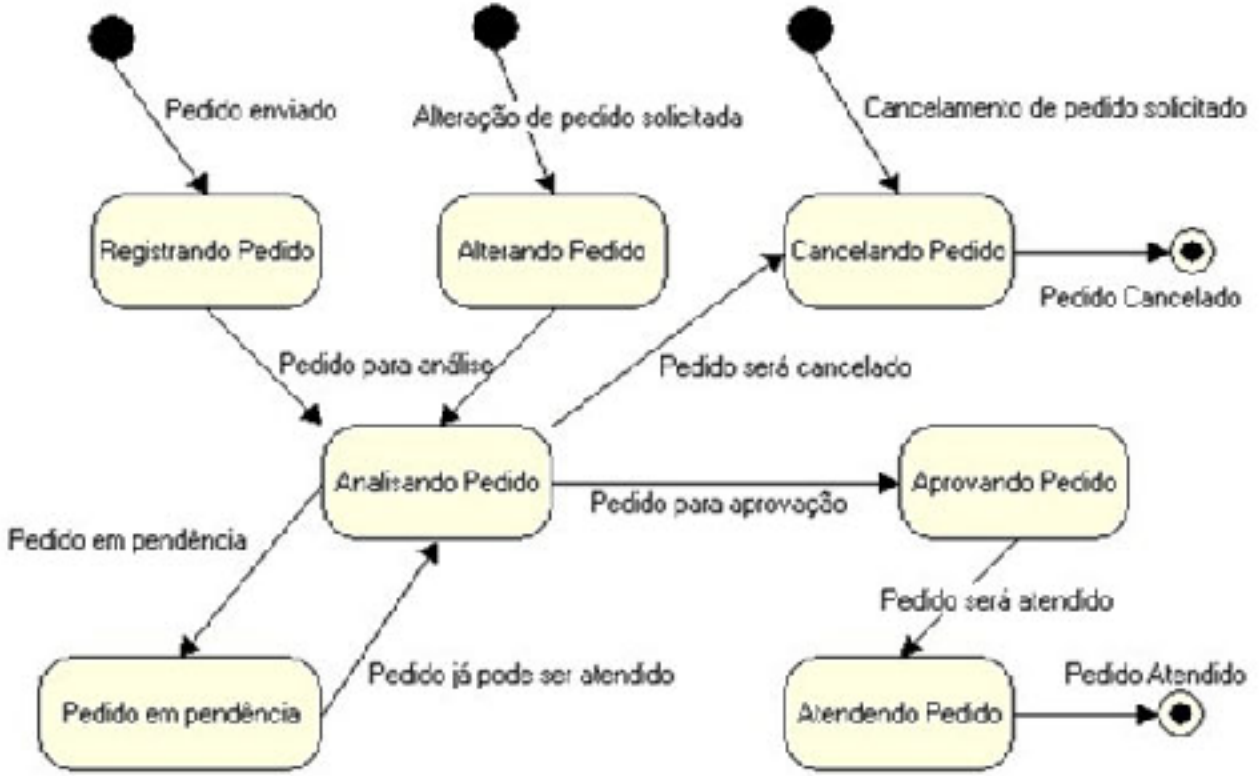
Diagrama de Caso de Uso

Diagram de Sequencia

Diagrama de Comunicação

Diagrama de Estados

Diagrama de Atividade



- Um diagrama de estados mostr transições, eventos e atividades.

UML – DIAGRAMAS COMPORTAMENTAIS

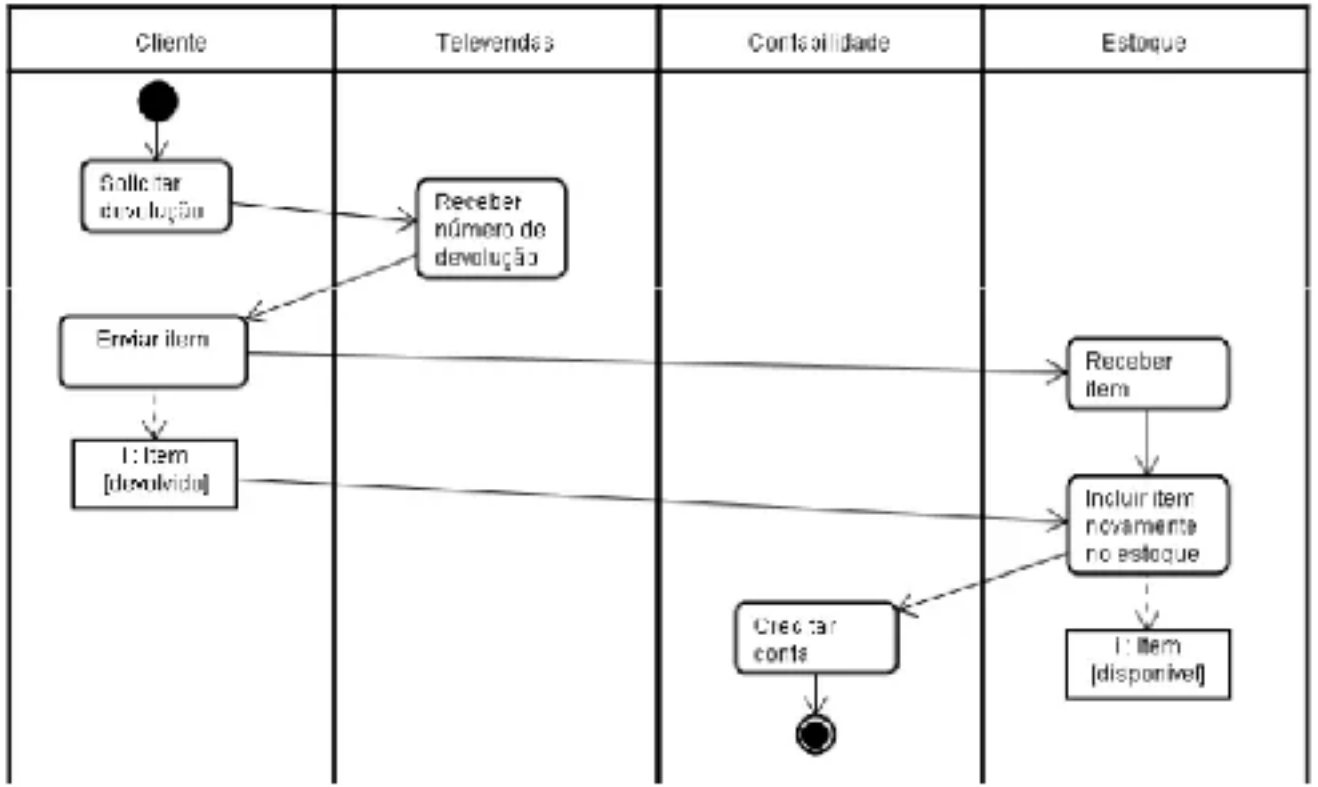
Diagrama de Caso de Uso

Diagram de Sequencia

Diagrama de Comunicação

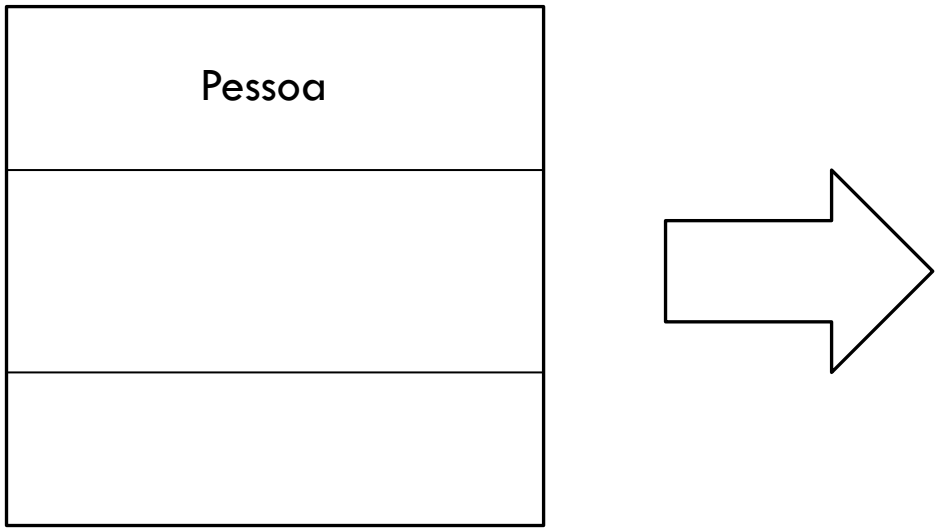
Diagrama de Estados

Diagrama de Atividade



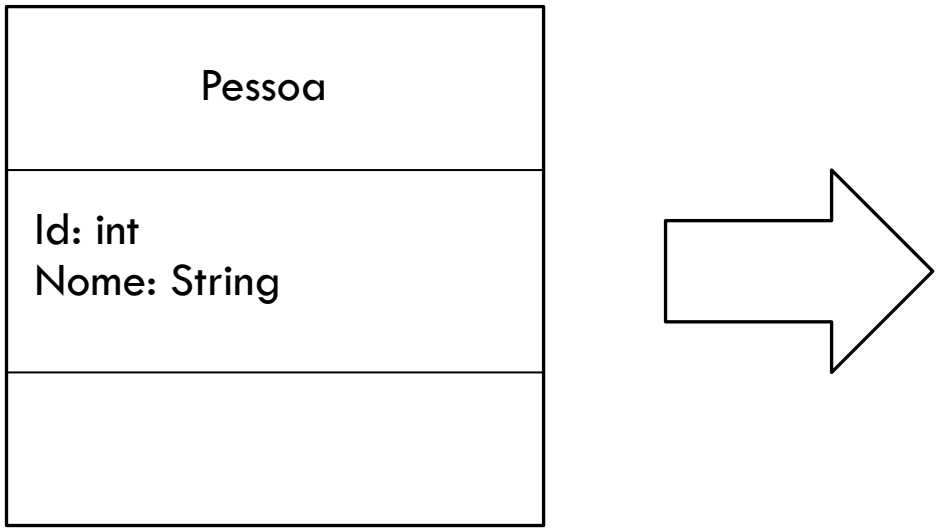
- Um diagrama de atividades mostra o
sistema.

DIAGRAMA DE CLASSES



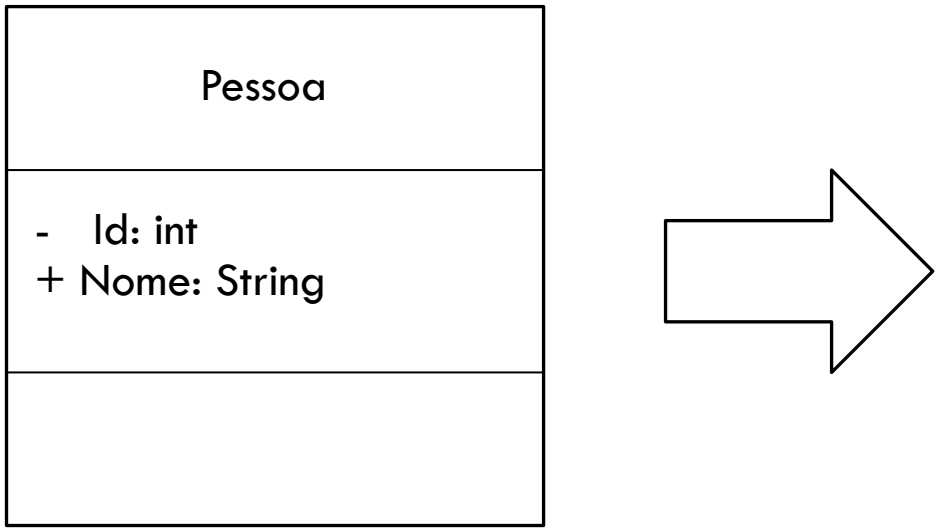

```
public class Pessoa {

}
```

DIAGRAMA DE CLASSES




```java
public class Pessoa {

    int id;
    String nome;
}
```

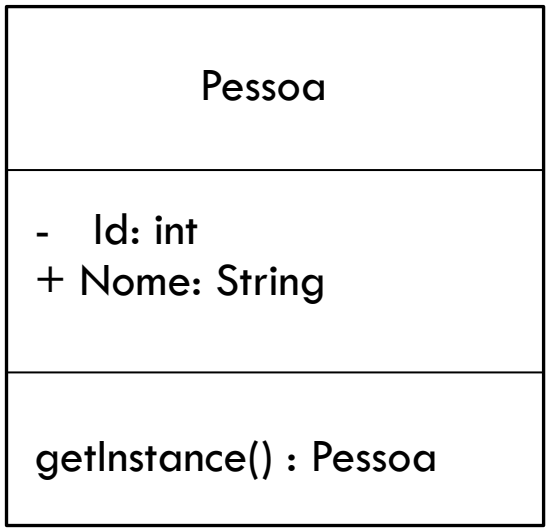
DIAGRAMA DE CLASSES

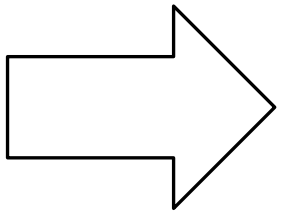



```
public class Pessoa {

    private int id;
    public String nome;
}
```

para modificador de acesso protected

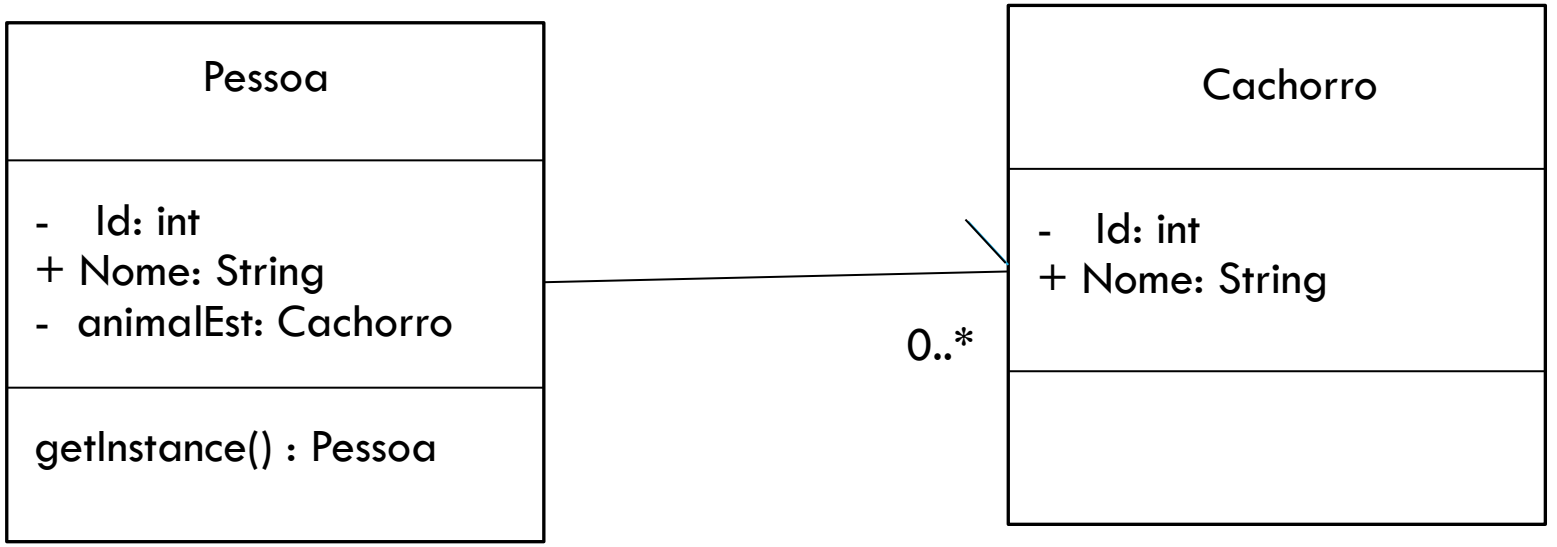
DIAGRAMA DE CLASSES





```
public class Pessoa {

    private int id;
    public String nome;

    public Pessoa getInstance(){
        return this;
    }
}
```

DIAGRAMA DE CLASSES (ASSOCIAÇÃO)

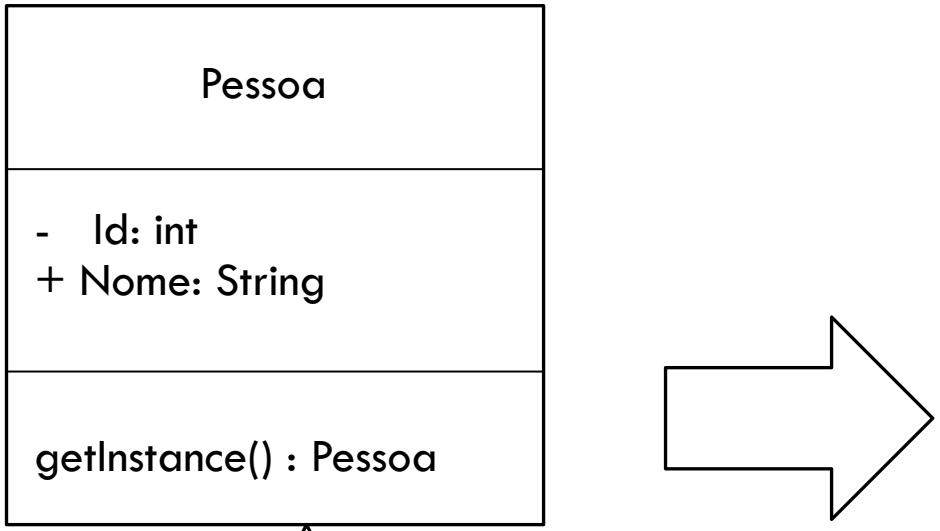



```
public class Pessoa {

    private int id;
    public String nome;


    public Pessoa getInstance(){
        return this;
    }
}
```

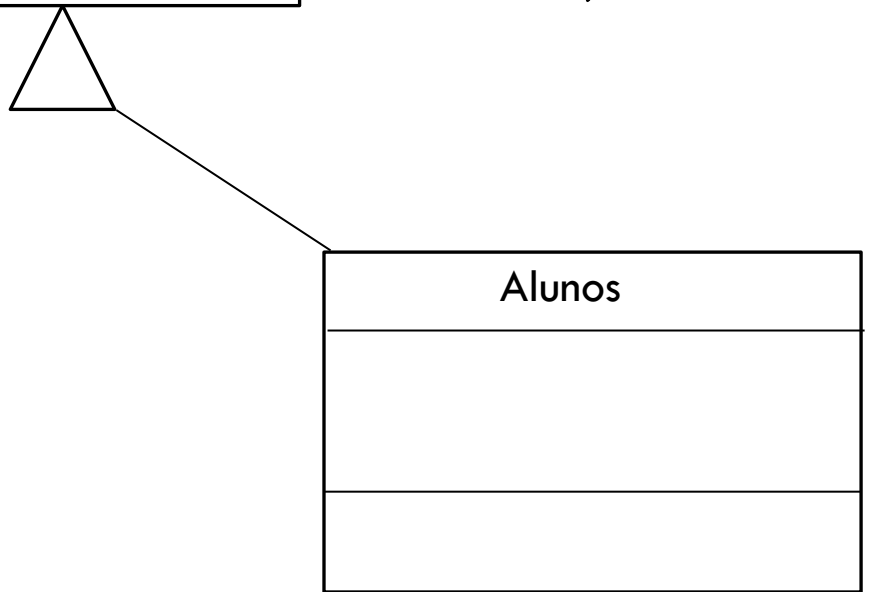
```
public class Cachorro {
    private int id;
    public String nome;

}
```

Indicador	Significado
0..1	Zero ou um
1	Somente um
0..*	Zero ou mais
*	Zero ou mais
1..*	Um ou mais
3	Somente três
0..5	Zero a cinco
5..15	Cinco a quinze

DIAGRAMA DE CLASSES

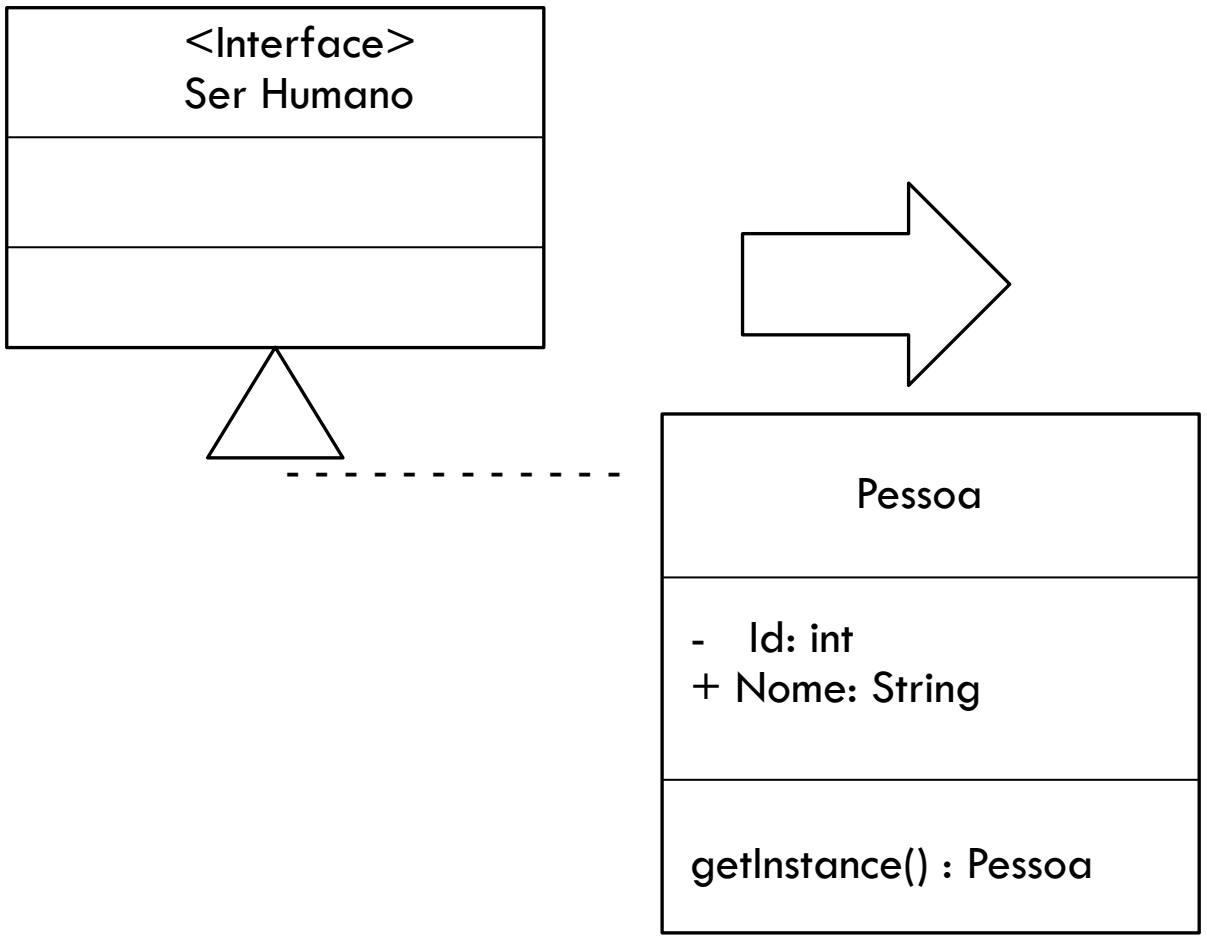



```java
public class Pessoa {

    private int id;
    public String nome;

    public Pessoa getInstance(){
        return this;
    }
}
```




```java
public class Aluno extends Pessoa {

}
```

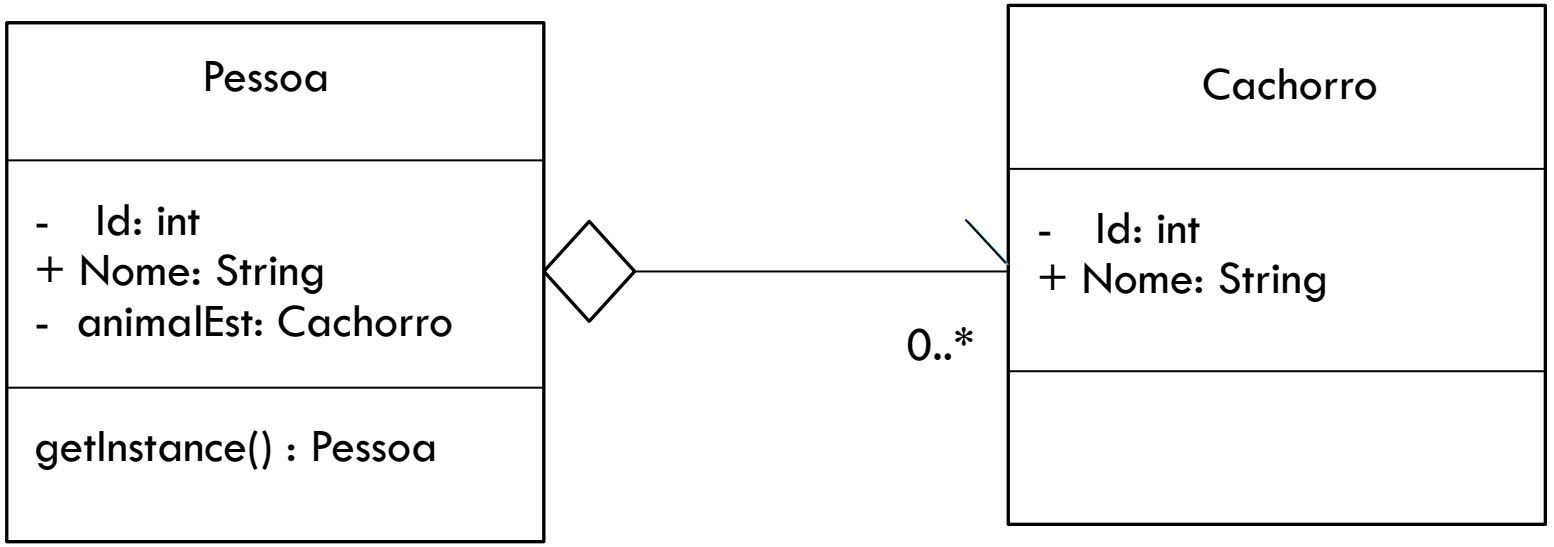
DIAGRAMA DE CLASSES




```java
public interface SerHumano {

}
```

```java
public class Pessoa {

    private int id;
    public String nome;

    public Pessoa getInstance(){
        return this;
    }
}
```

DIAGRAMA DE CLASSES (AGREGAÇÃO BÁSICA)

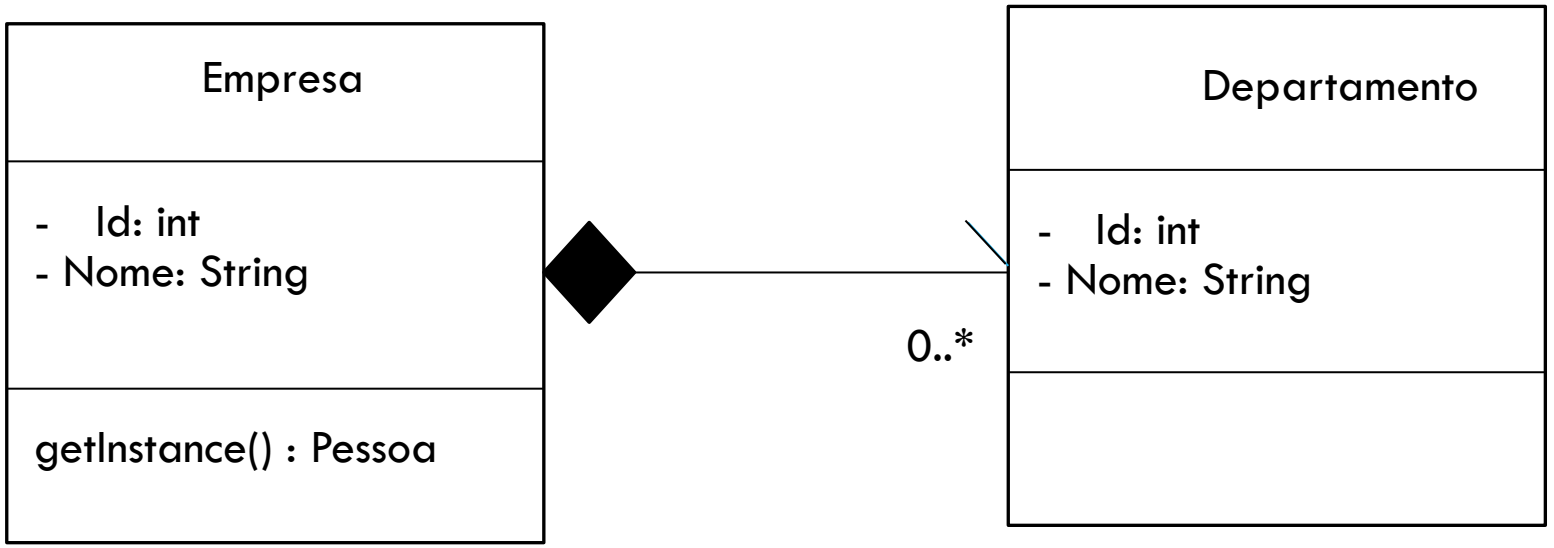



```java
public class Pessoa {

    private int id;
    public String nome;
    private List<Cachorro> animaisEstimacao;

    public Pessoa getInstance(){
        return this;
    }
}
```

```java
public class Cachorro {
    private int id;
    public String nome;

}
```

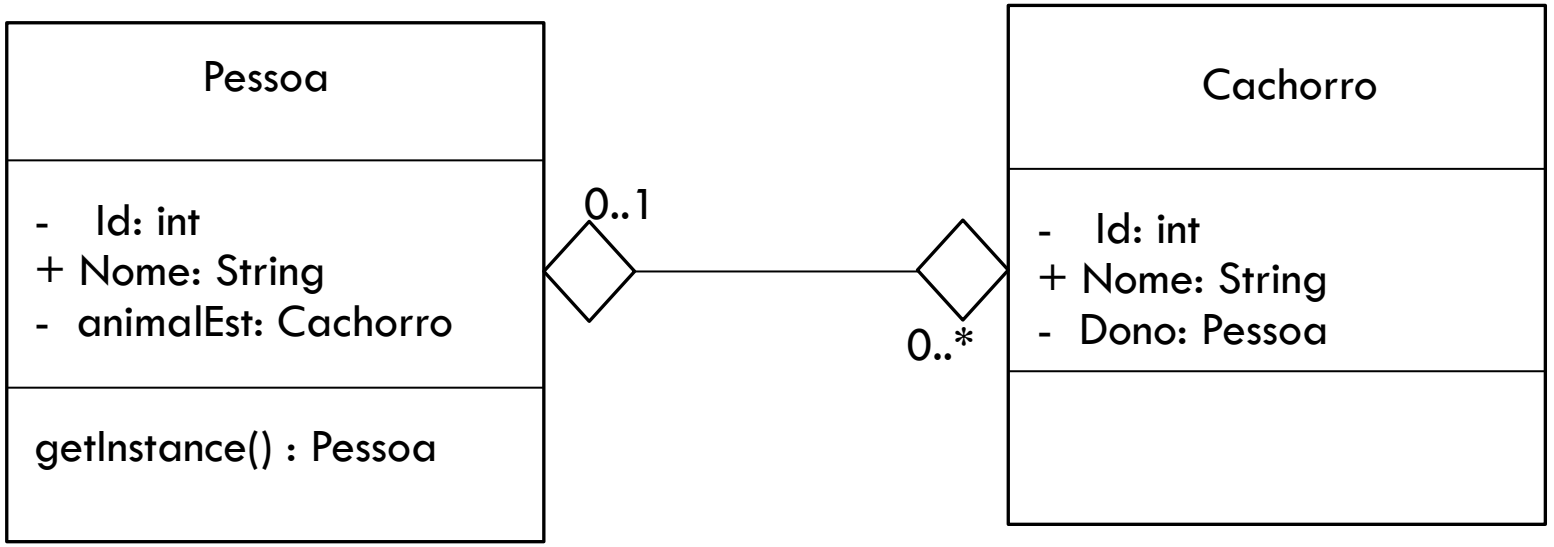
Indicador	Significado
0..1	Zero ou um
1	Somente um
0..*	Zero ou mais
*	Zero ou mais
1..*	Um ou mais
3	Somente três
0..5	Zero a cinco
5..15	Cinco a quinze

DIAGRAMA DE CLASSES (AGREGAÇÃO COMPOSIÇÃO)



Indicador	Significado
0..1	Zero ou um
1	Somente um
0..*	Zero ou mais
*	Zero ou mais
1..*	Um ou mais
3	Somente três
0..5	Zero a cinco
5..15	Cinco a quinze

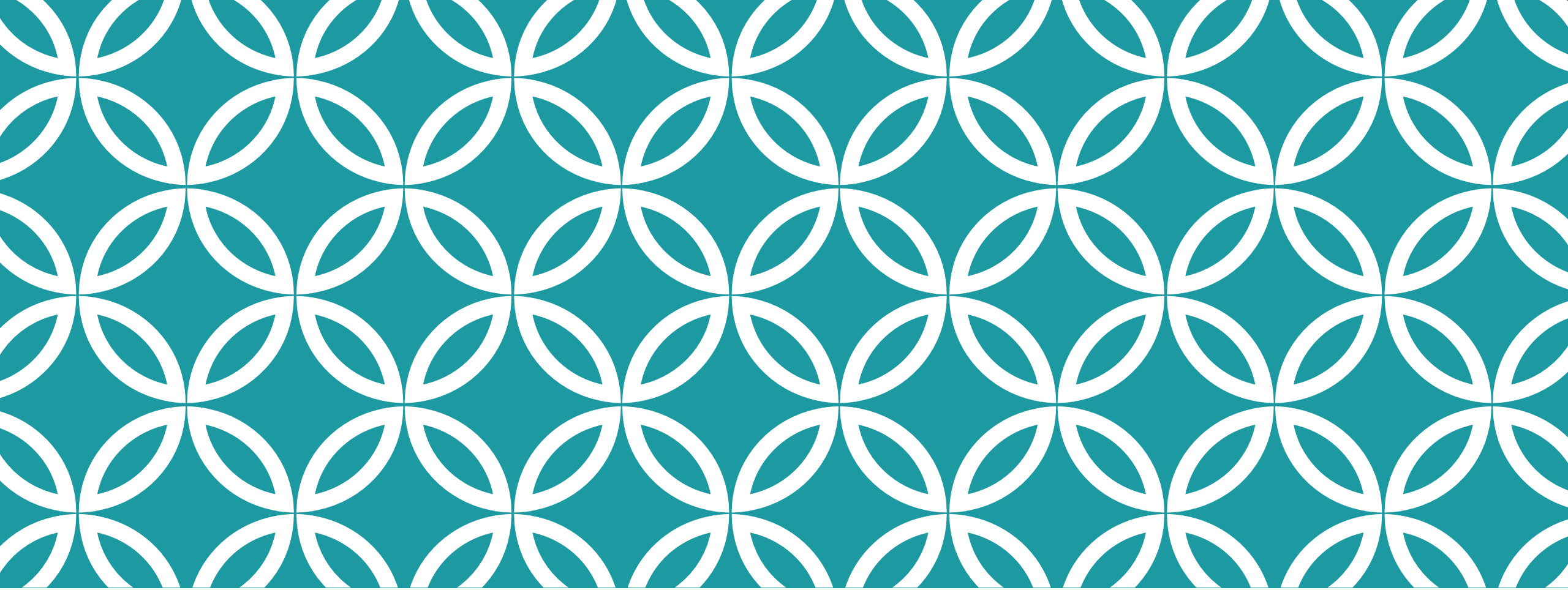
DIAGRAMA DE CLASSES (AGREGAÇÃO)



Indicador	Significado
0..1	Zero ou um
1	Somente um
0..*	Zero ou mais
*	Zero ou mais
1..*	Um ou mais
3	Somente três
0..5	Zero a cinco
5..15	Cinco a quinze

```
public class Pessoa {

    private int id;
    public String nome;
    private List<Cachorro> animalEstimcao;

    public Pessoa getInstance(){
        return this;
    }
}
```

```
public class Cachorro {
    private int id;
    public String nome;
    private Pessoa dono;
}
```



PADRÕES DE PROJETOS

HISTÓRIA

A idéia surgiu em 1977 quando Christopher Alexander publicou um catálogo com mais de 250 padrões para a arquitetura civil, que discutiam questões comuns da arquitetura, descrevendo em detalhe o problema e as justificativas de sua solução.

Christopher Alexander descobriu que diminuindo o foco, ou seja, procurando estruturas que resolvam problemas similares, ele pode discernir similaridades entre projetos de alta qualidade. Ele chamou essas similaridades de “padrões”.

PADRÕES DE PROJETOS

Em Engenharia de Software, um padrão de projeto (do inglês design pattern) é uma solução geral para um problema que ocorre com frequência dentro de um determinado contexto no projeto de software, não sendo um projeto fechado e sim a solução de um problema seguindo as melhores práticas.

PADRÕES DE PROJETOS

Um padrão de projeto define:

- ✓ seu nome,
- ✓ o problema,
- ✓ quando aplicar esta solução e
- ✓ suas consequências.

Os padrões de projeto:

- ✓ Facilitam a reutilização de soluções de desenho
- ✓ Estabelecem um vocabulário comum de desenho

PADRÕES DE PROJETOS

- Primeiros conceitos de padrões para Ciência da Computação (Kent e Ward em 1987)
- Livro "*Design Patterns: Elements of Reusable Object-Oriented Software*" por GoF (Gangue of Four) em 1995.

MOTIVAÇÃO

Dessa forma, resumidamente pode-se entender como padrão de projeto, como a solução recorrente para um problema em um contexto, mesmo que em projetos e áreas distintas. Observe que os termos chaves dessa definição são:

CONTEXTO → **PROBLEMA** → **SOLUÇÃO**

MOTIVAÇÃO

O mais importante sobre os padrões é que eles são **soluções aprovadas**, ou seja, foram aplicados por desenvolvedores em vários projetos !

DESING PATTERNS

Projetar soluções de software com maior qualidade

Dependência com Orientação a Objetos (Extensibilidade e Reusabilidade) e requisitos funcionais e não funcionais bem elaborados.

Os padrões de projetos tornam mais fácil a reutilização de soluções e arquiteturas bem sucedidas para construir softwares orientados a objetos de forma flexível e fácil de manter

DESIGN PATTERNS

Problemas apresentados anteriormente que deveriam ser lições aprendidas para novos projetos.

Gasto de tempo e recurso em busca de soluções que em tese já haviam sido encontradas

CARACTERÍSTICAS

Características obrigatórias que devem ser atendidas pelos padrões:

1 . Devem possuir um nome (facilita comunicação entre stakeholders), que descreva o problema, as soluções e conseqüências (para análise de viabilidade e soluções alternativas).

2. Todo padrão deve relatar de maneira clara a qual (is)problema(s) ele deve ser aplicado e em qual(is) condição (ões).

3. Solução descreve os elementos que compõem o projeto, seus relacionamentos, responsabilidades e colaborações.

QUANDO OS PADRÕES DE PROJETOS NÃO AJUDARÃO

Os padrões são um mapa, não uma estratégia !

QUANDO OS PADRÕES DE PROJETOS NÃO AJUDARÃO

Os catálogos geralmente apresentarão algum código-fonte como uma solução de exemplo para um problema, portanto eles não devem ser considerados como definitivos

QUANDO OS PADRÕES DE PROJETOS NÃO AJUDARÃO

Os padrões não ajudarão a determinar qual aplicação você deve estar escrevendo apenas como implementar melhor a aplicação assim que o conjunto de recursos e outras exigências forem determinados.

Os padrões ajudam com o que e como, mas não com por que ou quando.

ANTI PADRÕES

O conceito de utilizar os padrões de forma indiscriminada é conhecida como antipadrões (anti patterns). De acordo com Andrew Koenig, se um padrão representa a “melhor prática”, então um antipadrão representa uma “lição aprendida”.

Existem duas noções de anti padrões:

Aqueles que descrevem uma solução ruim para um problema que resultou em uma situação ruim;

Aqueles que descrevem como se livrar de uma situação ruim e como proceder dessa situação para uma situação boa.

ANTI PADRÕES

A utilização dos padrões proporciona um aumento na flexibilidade do sistema, entretanto pode deixa-lo mais complexo ou degradar a performance. Algumas perdas são toleráveis, mas subestimar os efeitos colaterais da adoção dos patterns, é um erro comum, principalmente daqueles que tomam o uso como um diferencial e não pela real necessidade.

COMO SOLUCIONAM PROBLEMAS

Como os objetos são os elementos chaves em projetos OO, **a parte mais difícil do projeto é a decomposição de um sistema em objetos**. A tarefa é difícil porque muitos fatores entram em jogo: encapsulamento, granularidade, dependência, flexibilidade, desempenho, evolução, reutilização e assim por diante. Todos influenciam a decomposição, **freqüentemente de formas conflitantes.**

Muito dos objetos participantes provêm do método de análise. Porém, projetos orientados a objetos acabam sendo compostos por objetos **que não possui uma contrapartida no mundo real.**

COMO SOLUCIONAM PROBLEMAS

As **abstrações que surgem durante um projeto são as chaves para torna-lo flexível**. Os padrões de projeto **ajudam a identificar abstrações menos óbvias** bem como os objetos que podem capturá-las. Por exemplo, objetos que representam processo ou **algoritmo não ocorrem na natureza, no entanto, eles são uma parte crucial de projetos flexíveis. Esses objetos são raramente encontrados durante a análise ou mesmo durante os estágios iniciais de um projeto**; eles são **descobertos mais tarde**, durante o processo de tornar um projeto mais flexível e reutilizável.

COMO SELECIONAR UM PADRÃO DE PROJETO

Escolher dentre os padrões existentes aquele que **melhor soluciona um problema** do projeto, sem **cometer o erro de escolher de forma errônea e torná-lo inviável, é uma das tarefas mais difíceis**. Em suma, a escolha de um padrão de projeto a ser utilizado, pode ser baseada nos seguintes critérios:

1. Considerar como os padrões de projeto solucionam problemas de projeto.
2. Examinar qual a intenção do padrão, ou seja, o que faz de fato o padrão de projeto, quais seus princípios e que tópico ou problema particular de projeto ele trata (soluciona).
3. Estudar como os padrões se relacionam.
4 .Estudar as semelhanças existentes entre os padrões.
5. Examinar uma causa de reformulação de projeto.
6. Considerar o que deveria ser variável no seu projeto, ou seja, ao invés de considerar o que pode forçar uma mudança em um projeto, considerar o que você quer ser capaz de mudar sem reprojetá-lo.

COMO USAR UM PADRÃO DE PROJETO

Depois de ter sido feita a escolha do(s) padrão (ões) a ser (em) utilizado(s) no projeto é necessária conhecer como utilizá-lo(s). Uma abordagem recomendada pela gangue dos quatros amigos para aplicar um padrão a um projeto é:

Ler o padrão por completo uma vez, para obter sua visão geral. Conhecer o padrão principalmente a sua aplicabilidade e conseqüências são importantes para que ele realmente solucione o seu problema;

Estudar seções Estrutura, Participantes e Colaborações. Assegurando-se de que compreendeu as classes e objetos no padrão e como se relacionam entre si;

Escolher os nomes para os participantes do padrão que tenham sentido no contexto da aplicação;

Definir as classes. Declarar as interfaces, estabelecer os seus relacionamentos de herança e definir as variáveis de instância que representam dados e referências a objetos. Identifique as classes existentes na aplicação que serão afetadas pelo padrão e modifique-as;

COMO USAR UM PADRÃO DE PROJETO

Defina os nomes específicos da aplicação para as operações no padrão. Os nomes em geral dependem da aplicação. Use as responsabilidades e colaborações associadas com cada operação como guia;

Implemente as operações para suportar as responsabilidades e colaborações presentes do padrão. A seção de Implementação oferece sugestões para guiá-lo na implementação.

Essas são apenas diretrizes que podem ser utilizadas até que seja obtido experiência e conhecimento necessário para desenvolver uma maneira de trabalho particular com os padrões de projeto. Os padrões de projeto não devem ser aplicados indiscriminadamente. Freqüentemente eles obtêm flexibilidade e variabilidade pela introdução de níveis adicionais de endereçamento indireto, e isso pode complicar um projeto e/ou custar algo em termos de desempenho. Um padrão de projeto deverá apenas ser aplicado quando a flexibilidade que ele oferece for realmente necessária.

PADRÕES DE PROJETOS

De acordo com o livro: "Padrões de Projeto: soluções reutilizáveis de software orientado a objetos", os padrões "GoF" são divididos em 23 tipos.

São 3 as classificações/famílias:

Padrões de criação: relacionados à criação de objetos, visam abstrair o processo de criação de objetos, ou seja, sua instanciação

Padrões estruturais: tratam das associações entre objetos, identificando a melhor forma de realizar o relacionamento entre as entidades.

Padrões comportamentais: identificam padrões de comunicação entre objetos, implementando-os, a fim de aumentar a flexibilidade na condução desta comunicação.

PADRÕES DE PROJETOS

Padrões de criação

- Abstract Factory
- Builder
- Factory Method
- Prototype
- Singleton

Padrões estruturais

- Adapter
- Bridge
- Composite
- Decorator
- Facade
- Flyweight
- Proxy

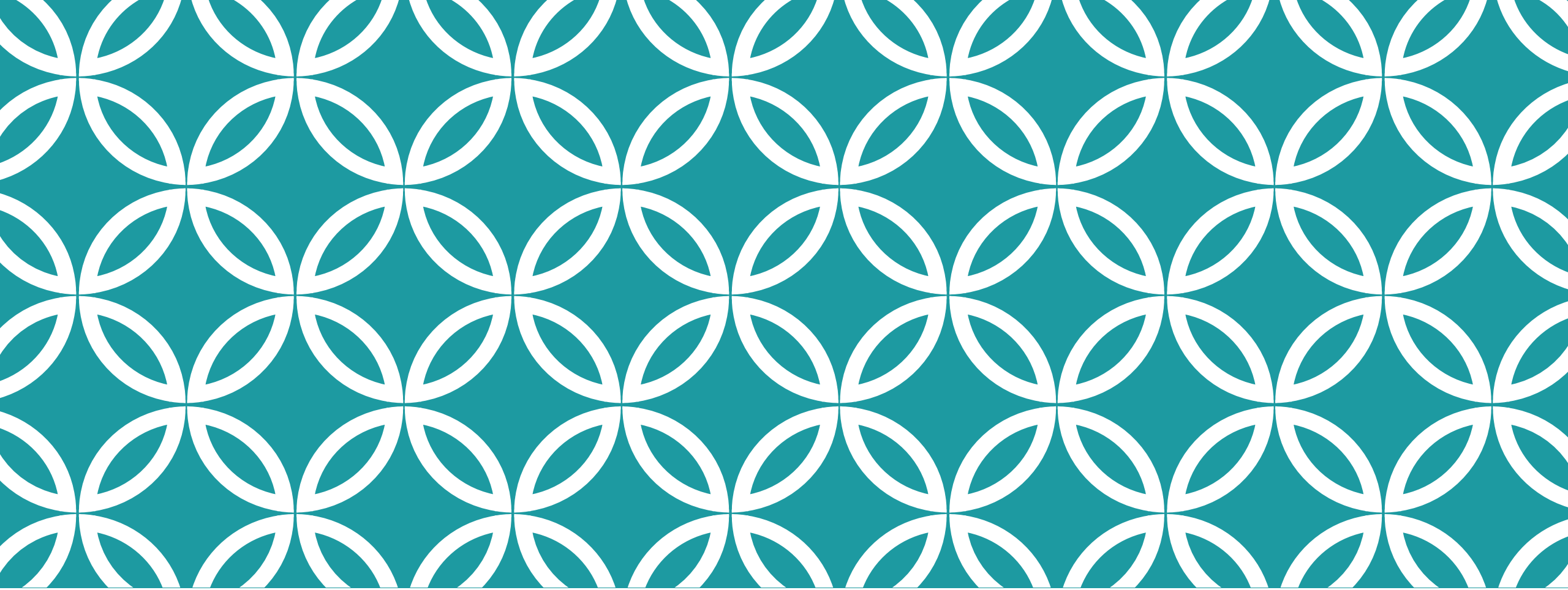
Padrões comportamentais

- Chain of Responsibility
- Command
- Interpreter
- Iterator
- Mediator
- Memento
- Observer
- State
- Strategy
- Template Method
- Visitor

PADRÕES DE PROJETOS

*Classificação dos 23 padrões segundo GoF**

		Propósito		
		1. Criação	2. Estrutura	3. Comportamento
Escopo	Classe	Factory Method	Class Adapter	Interpreter Template Method
	Objeto	Abstract Factory Builder Prototype Singleton	Object Adapter Bridge Composite Decorator Facade Flyweight Proxy	Chain of Responsibility Command Iterator Mediator Memento Observer State Strategy Visitor



PADRÕES DE CRIAÇÃO

PADRÕES DE CRIAÇÃO

Os padrões de criação são aqueles que abstraem e ou adiam o processo criação dos objetos.

Eles ajudam a tornar um sistema independente de como seus objetos são criados, compostos e representados.

Um padrão de criação de classe usa a herança para variar a classe que é instanciada, enquanto que um padrão de criação de objeto delegará a instanciação para outro objeto.

PADRÕES DE CRIAÇÃO

Há dois temas recorrentes nesses padrões. Primeiro todos **encapsulam conhecimento sobre quais classes concretas** são usadas pelo sistema. Segundo **ocultam o modo como essas classes são criadas** e montadas. Tudo que o sistema sabe no geral sobre os objetos é que suas classes são definidas por classes abstratas.

SINGLETON

Garantir que um **objeto terá apenas uma única instância**, isto é, que uma classe irá gerar apenas um objeto e que este estará disponível de forma única para todo o escopo de uma aplicação.

Algumas aplicações têm a necessidade de **controlar o número de instâncias** criadas de algumas classes, seja pela necessidade da própria lógica ou por motivos de performance e economia de recursos.

SINGLETON

O uso do padrão Singleton está condicionado a:

Quando for necessário manter num sistema, seja ele distribuído ou não, apenas uma instância de objeto e que o ponto de acesso para este seja bem conhecido (Ex. objeto responsável por um pool de impressão numa rede, gerenciador de janelas);

Quando a única instância tiver de ser extensível através de subclasses, possibilitando aos clientes usarem uma instância estendida sem alterar o seu código (visões polimórficas).

O desenvolvedores Delphi já utilizam o comportamento do padrão Singleton em suas aplicações, quando declaram variáveis globais na área de inicialização do projeto e depois reutilizam a instância dos objetos.TApplication, TCiipBoardsão exemplos de objetos que não teriam sentido existir mais de uma instância na aplicação e por isso assumem o comportamento do padrão Singleton.

SINGLETON

Os participantes são:

1- Singleton –define um método estático que permite aos clientes obterem o objeto único. Ele também será responsável pelo processo de criação do objeto.

Os benefícios que o padrão apresenta são:

1. Acesso controlado a instância única;

2. Espaço de nomes reduzido (diga não a proliferação de variáveis globais);

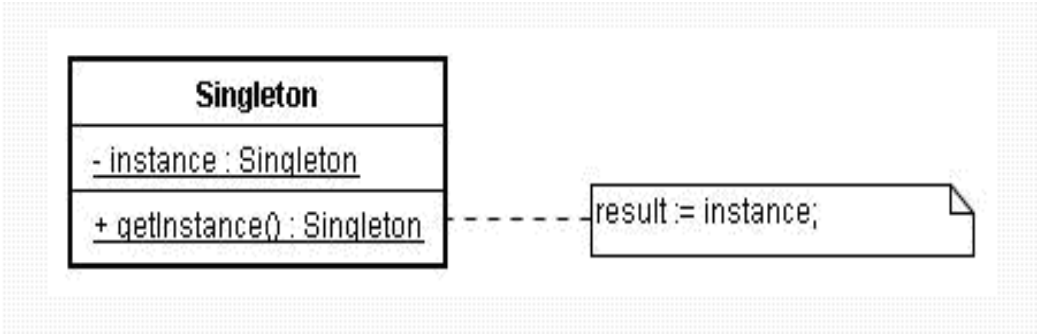
3. Permite um refinamento de operações e representações;

4. Permite um número variável de instâncias - O padrão possibilita que seja adotada a estratégica de criar mais de uma instância da classe, de forma controlada;

5. Mais flexível do que as operações de classe;

SINGLETON

Exemplo de Singleton:

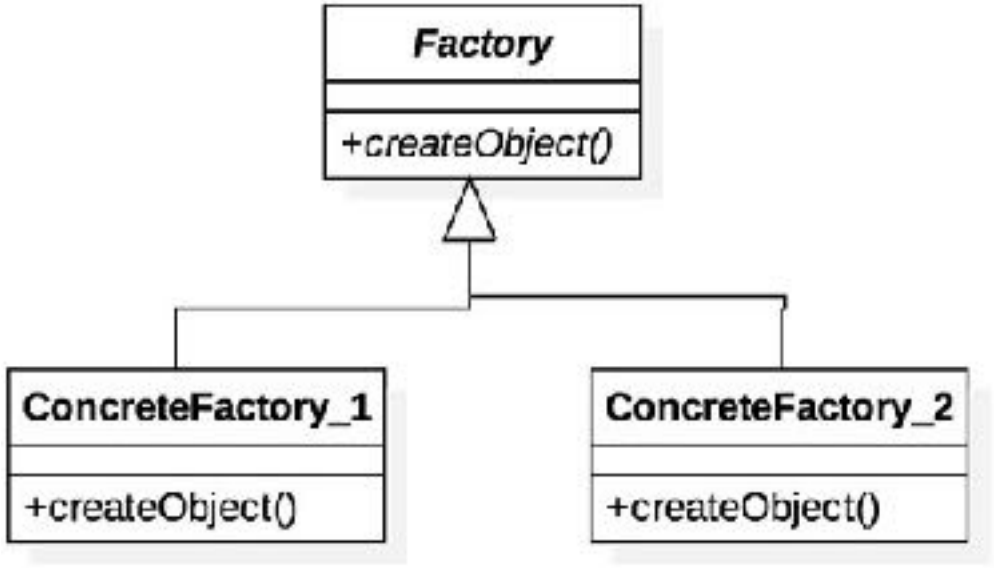


EXERCÍCIO SINGLETON

Implemente o padrão Singleton para fornecer apenas uma conexão com o banco de dados (não é necessário ser uma conexão real);

FACTORY METHOD

O Factory Method tem o objetivo de isolar código de criação (aquele que utiliza o **new**) das classes de negócio do projeto reduzindo o acoplamento do projeto.



FACTORY METHOD

Pontos positivos

O padrão é bastante difundido, em caso de outros desenvolvedores no projeto, não haverá a necessidade de explorar o código feito.

Remove o acoplamento forte que existia nas classes, separa a lógica de negócio própria delas e os códigos de criação de instâncias;

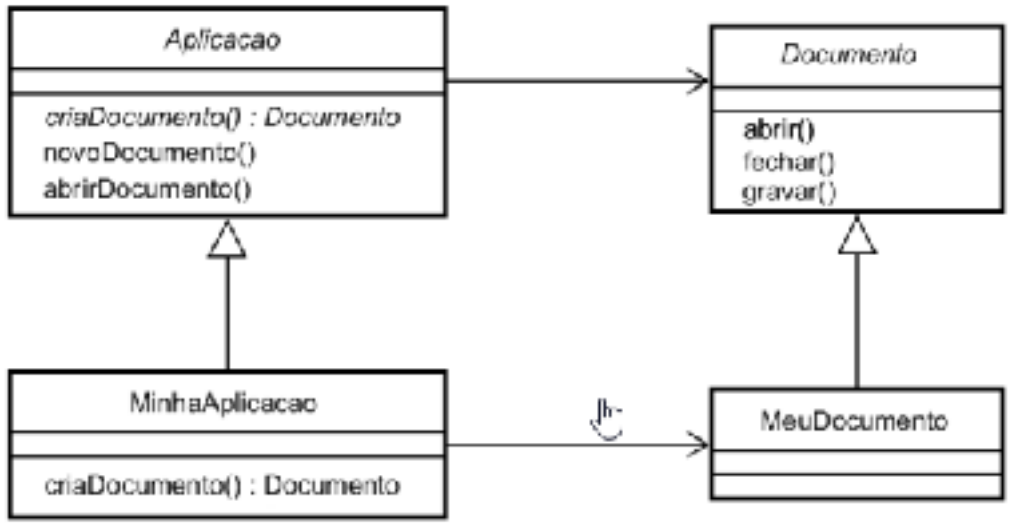
Com o baixo acoplamento, há apenas um local a ser alterado caso deseje mudar o construtor da classe gerada pela Fábrica.

Ponto negativo

Se as subclasses responsáveis pela instanciação das classes de domínio do problema for cobrir criações diretas de poucas classes e que são utilizadas poucas vezes, talvez o Factory complique mais do que ajude.

FACTORY METHOD

Exemplo de Factory Method:

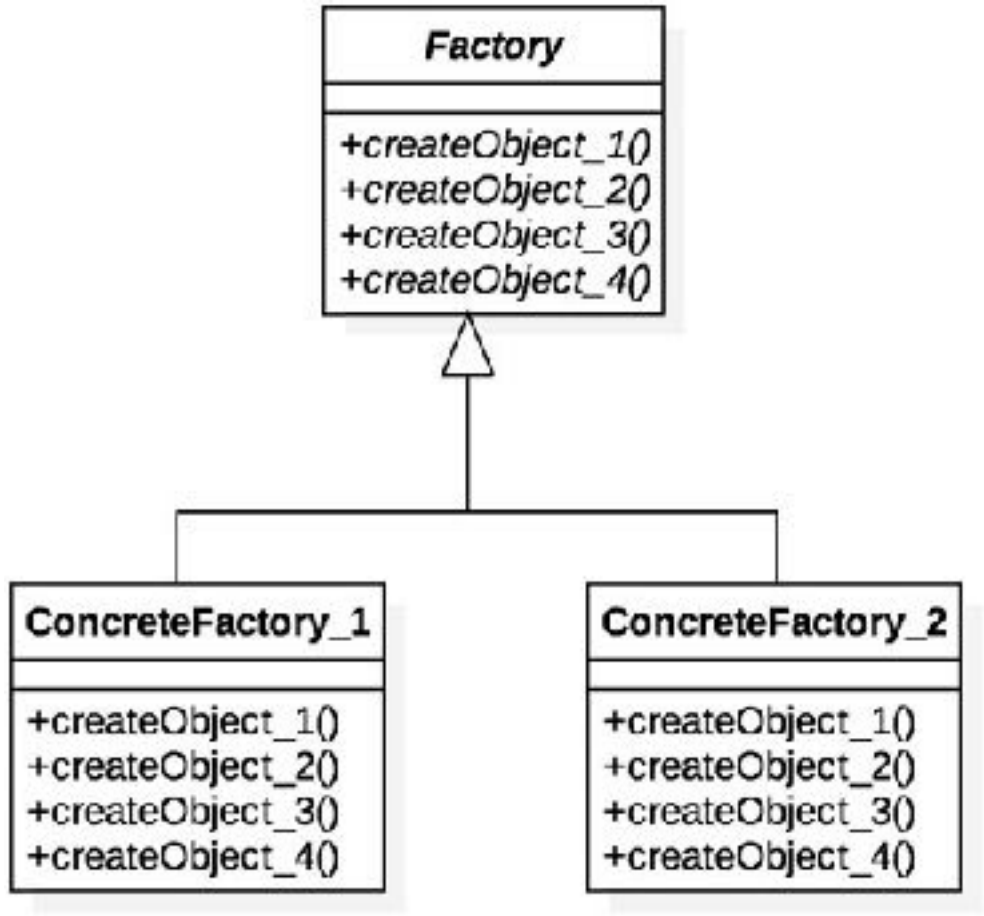


EXERCÍCIO FACTORY METHOD

Implemente o padrão Factory Method para a criação de um gerador de objetos que herdem da classa abstrata Animais (Gato, Cachorro, Peixe).

ABSTRACT FACTORY

Em Abstract Factory há uma família de métodos de criação. Outra diferença que é notável é que enquanto Factory Method é comumente utilizado como herança o Abstract Factory é comum como composição.



ABSTRACT FACTORY

Pontos positivos

Pelo padrão ser conhecido, não haverá a necessidade de explorar o código feito.

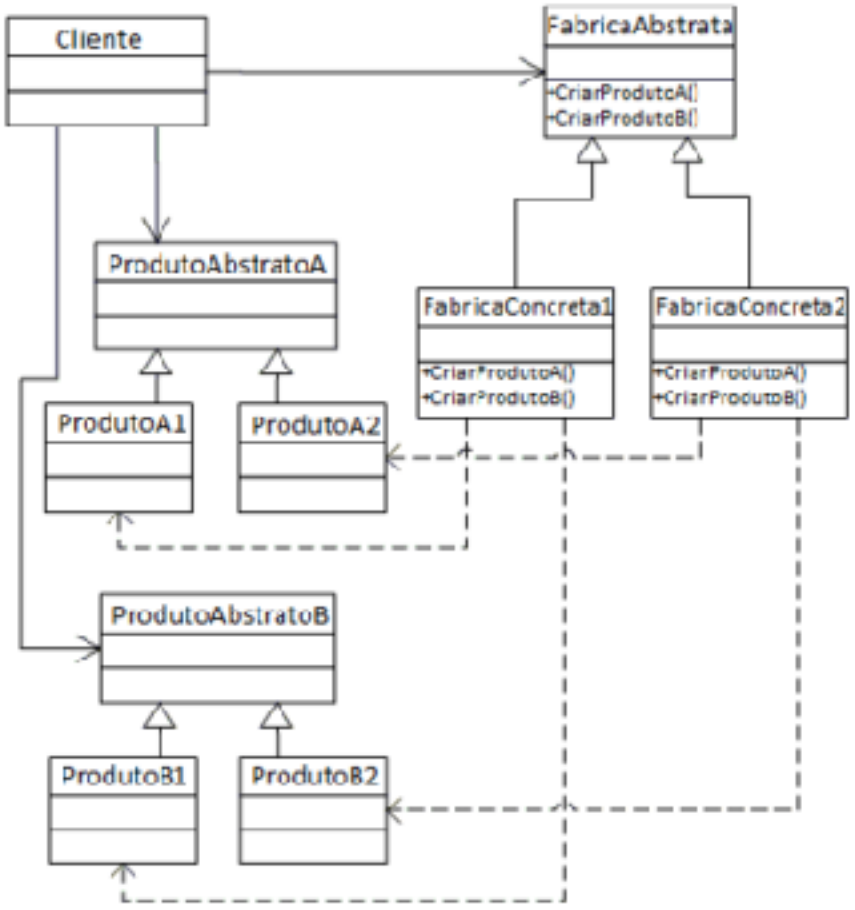
Encapsulamento do código de criação em pontos únicos do projeto removendo também com isso o forte acoplamento entre classes que não deveriam ter um alto nível de relacionamento umas com as outras;

Ponto negativo

Implementar um Abstract Factory quando um simples Simple Factory ou Factory Method seriam o suficiente, pode piorar a performance do projeto, ainda mais na leitura dele.

ABSTRACT FACTORY

Exemplo de Abstract Factory:

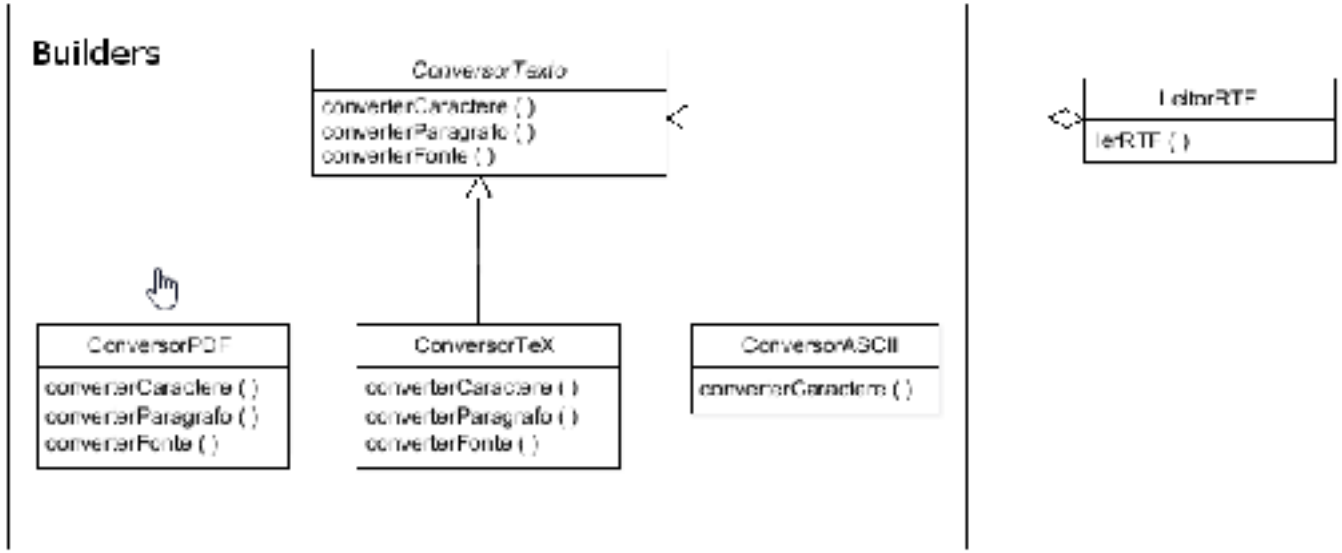


EXERCÍCIO ABSTRACT FACTORY

Implemente o padrão Abstract Factory para a criação de um gerador de objetos que herdem da classa abstrata Animais (Gato, Cachorro, Peixe, Papagaio) e classifique os animais em grupos de animais (Aquatico, Terrestre, Voador).

BUILDER

Separar a construção de um objeto complexo de sua representação de modo que o mesmo processo de construção possa criar diferentes representações.



BUILDER

Vantagens

Separar em pequenos passos a construção do objeto complexo.

Builder é bem menor que a dos factorys

Desvantagem

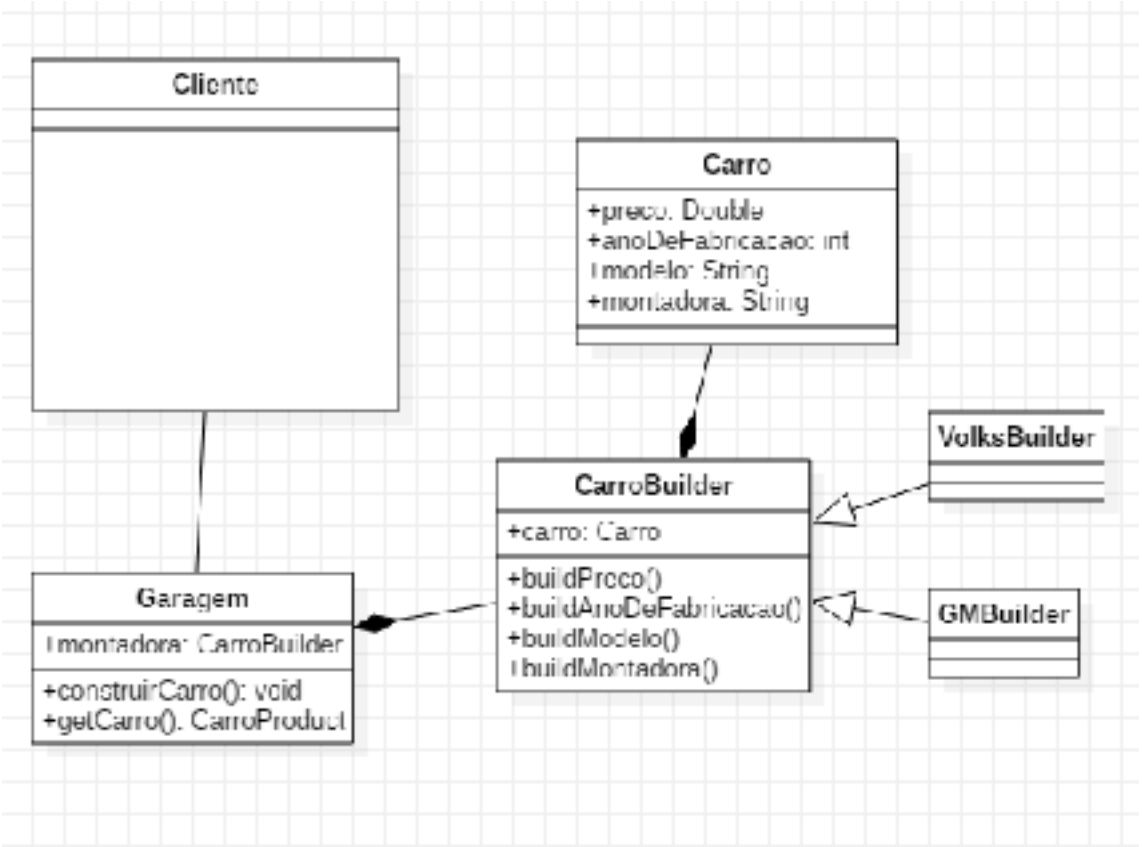
O padrão Builder, não existe o conceito de vários produtos ou de famílias de produtos, como nos outros dois padrões.

Builder foca na divisão de responsabilidades na construção do Produto.

O processo de criação e devolver o produto no final, definindo quais os passos devem ser executados

BUILDER

Exemplo de Builder

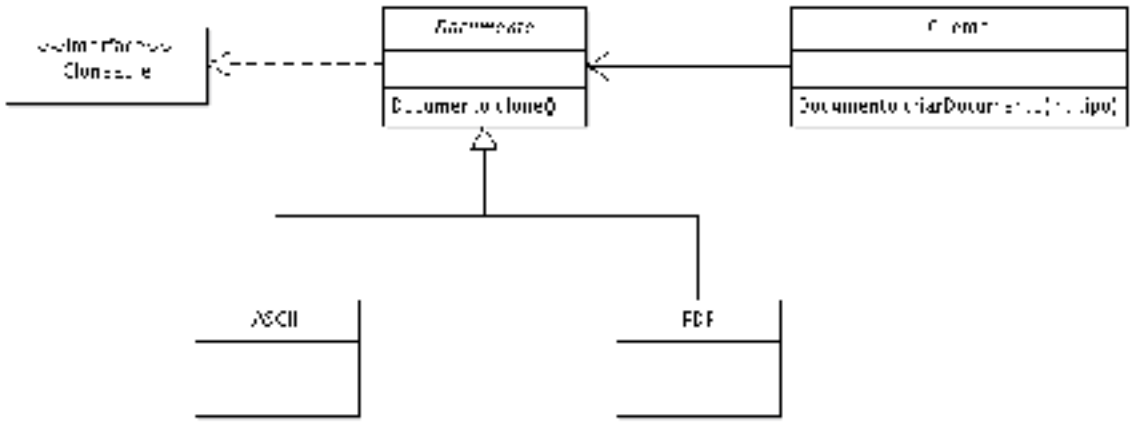


EXERCÍCIO BUILDER

Implemente o padrão Builder para a construção de uma Casa com variações de Casas (Padrão Simples, Elevado, etc...).

PROTOTYPE

Especificar tipos de objetos a serem criados usando uma instância protótipo e criar novos objetos pela cópia desse protótipo.



PROTOTYPE

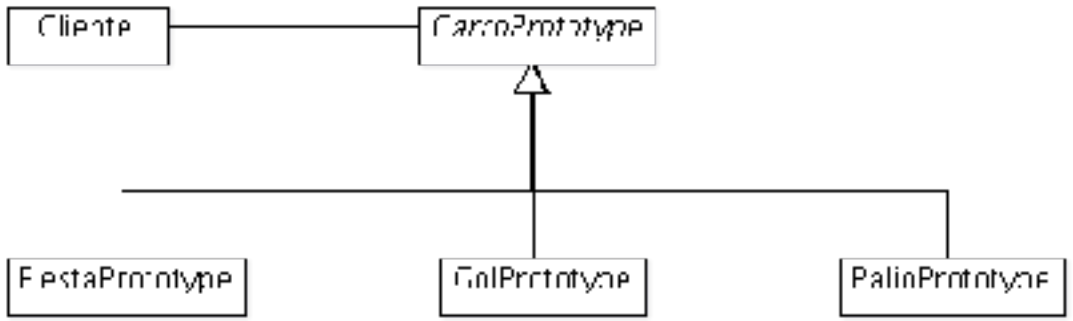
Permite que o cliente instancie vários protótipos, quando um deles não é mais necessário, basta removê-lo. Se é preciso adicionar novos protótipos, basta incluir a instanciação no cliente. Essa flexibilidade pode ocorrer inclusive em tempo de execução.

É preciso garantir que o método de cópia esteja implementado corretamente, para evitar que a alteração nos valores mude todas as instâncias.

O padrão Prototype leva grande vantagem quando o processo de criação de seus produtos é muito caro, ou mais caro do que uma clonagem.

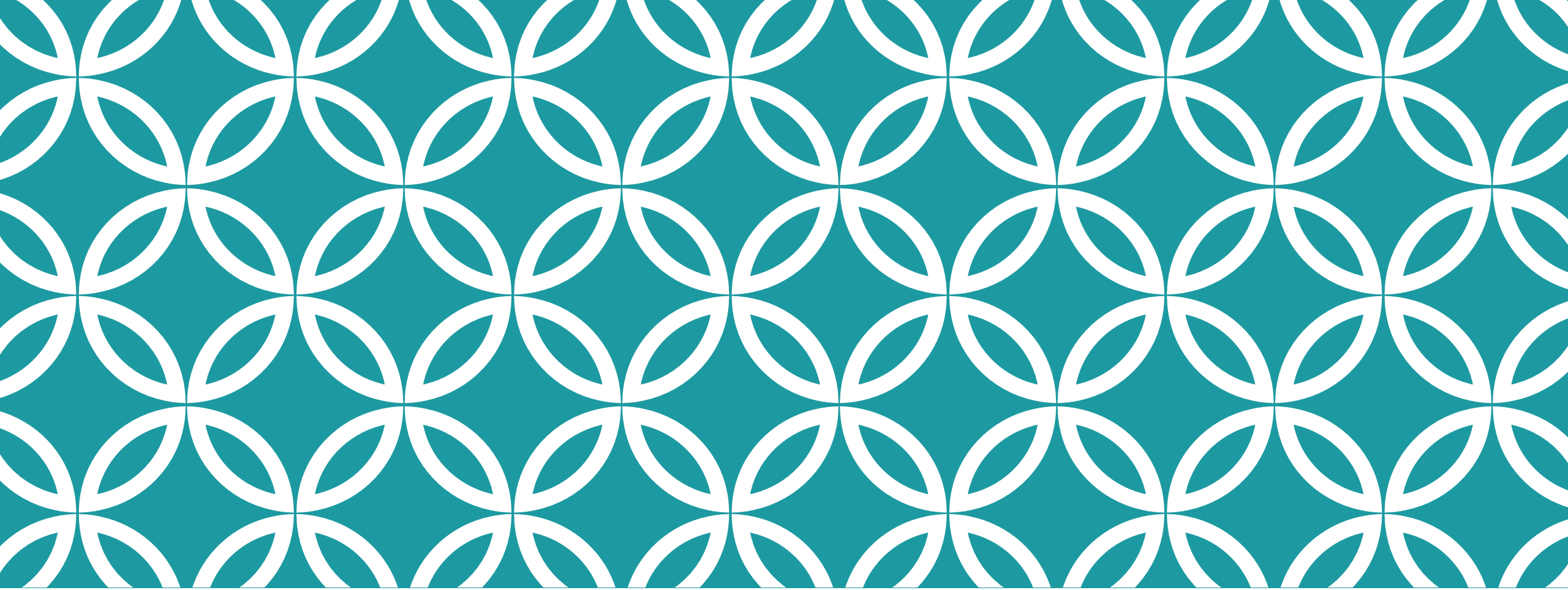
PROTOTYPE

Exemplo de Prototype



EXERCÍCIO PROTOTYPE

Crie uma classe Documento e gere Protótipos de documentos em classes (DocumentoPages, DocumentoPdf, DocumentoDocx).



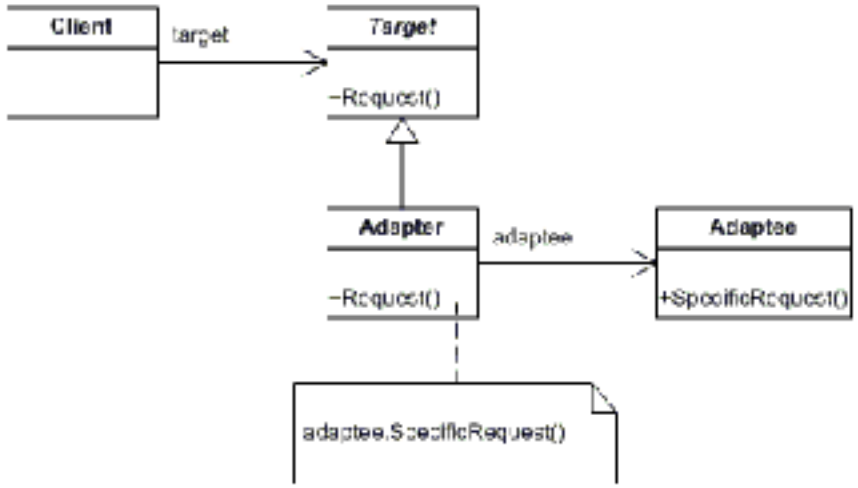
PADRÕES ESTRUTURAIS

PADRÕES ESTRUTURAIS

Os padrões estruturais vão se preocupar em como as classes e objetos são compostos, ou seja, como é a sua estrutura. O objetivo destes padrões e facilitar o design do sistema identificando maneiras de realizar o relacionamento entre as entidades, deixando o desenvolvedor livre desta preocupações inerentes a alto acoplamento (por exemplo).

ADAPTER

Adapter (também conhecido como Wrapper) converte uma interface de uma classe para outra interface que o cliente espera encontrar, permitindo que classes com interfaces incompatíveis trabalhem juntas.



ADAPTER

Vantagens:

O uso da composição de objetos

Vinculação do cliente a uma interface e não a uma implementação

Flexibilidade com o uso de diversos adaptadores.

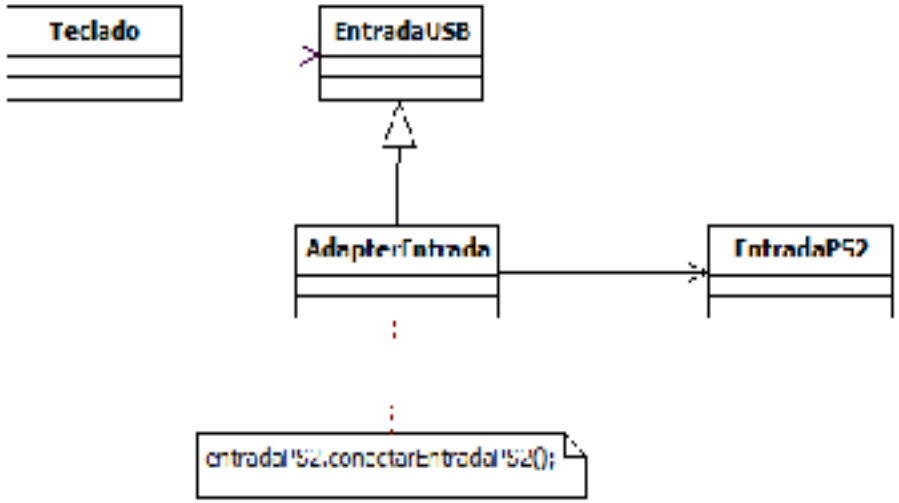
Desvantagens

É utilizado para uma adaptação, portanto, todas as arestas devem ser validadas.

Quando é necessário adaptar mais de uma classe, é necessário utilizar outro padrão de projeto.

ADAPTER

Exemplo de Adapter:



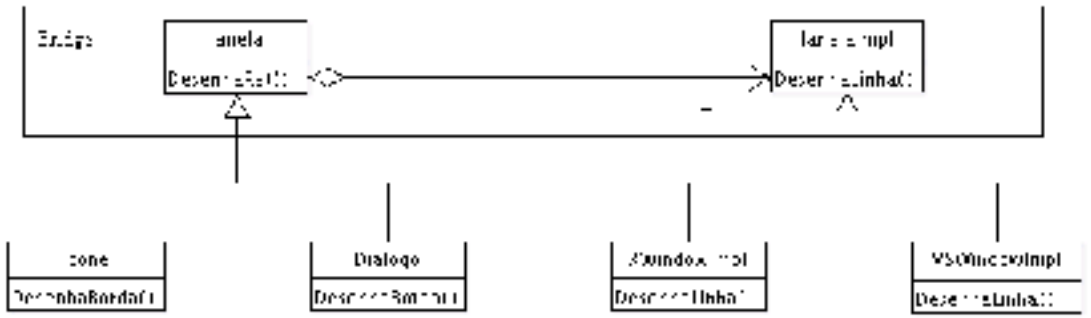
EXERCÍCIO ADAPTER

Crie o Padrão Adapter para as classes TomadaDoisPinos e TomadaTresPinos.

BRIDGE

"Desacoplar uma abstração da sua implementação, de modo que as duas possam variar independentemente."

Ou seja, o Bridge fornece um nível de abstração maior que o Adapter, pois são separadas as implementações e as abstrações, permitindo que cada uma varie independentemente.



BRIDGE

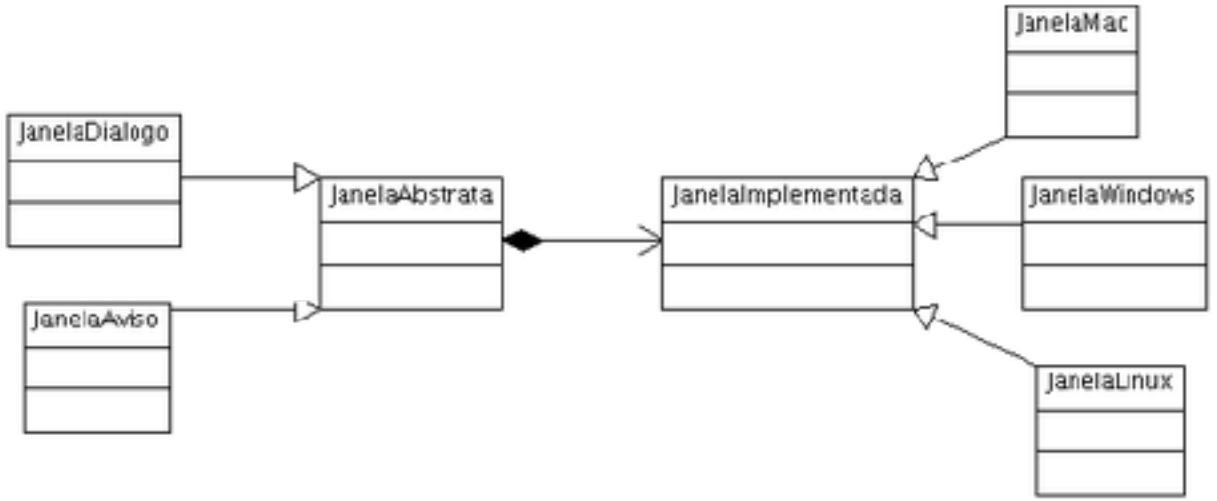
Bridge provê um excelente nível de desacoplamento dos componentes. Tanto novas abstrações como novas plataformas podem ser acomodadas pelo sistema sem grandes dificuldades, graças a extensibilidade do padrão.

Outro ponto que é comum a maioria dos padrões é a ocultação dos detalhes de implementação do cliente, que fica independente de qualquer variação ou extensão que precise ser feita.

Um ponto que merece um certo cuidado é sobre a instanciação dos objetos, pois vimos que é necessário especificar a abstração e utilizar uma implementação, assim o cliente precisa conhecer bem as classes, e o que elas realizam para saber exatamente o que, quando e como fazer.

BRIDGE

Exemplo Padrão de Projeto Bridge:

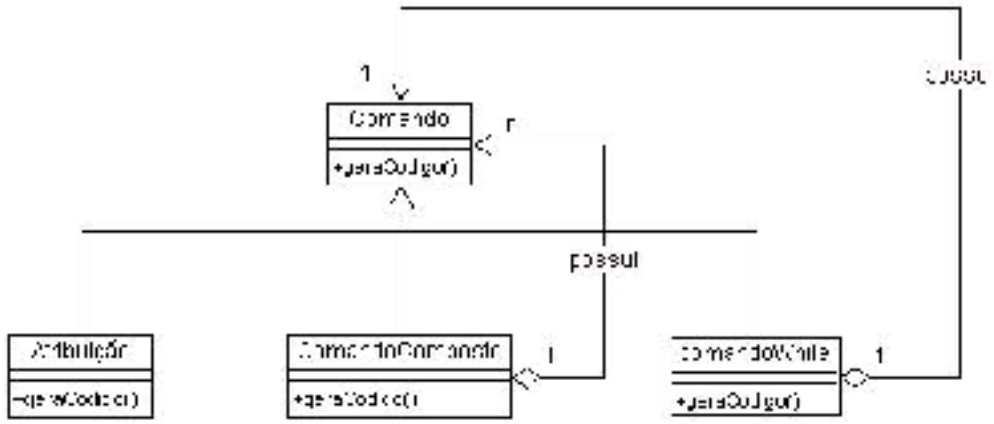


EXERCÍCIO BRIDGE

Crie uma classe Navegador e sua Implementação para Chrome, Firefox e Edge. Uma classe website deverá se comunicar com Navegador.

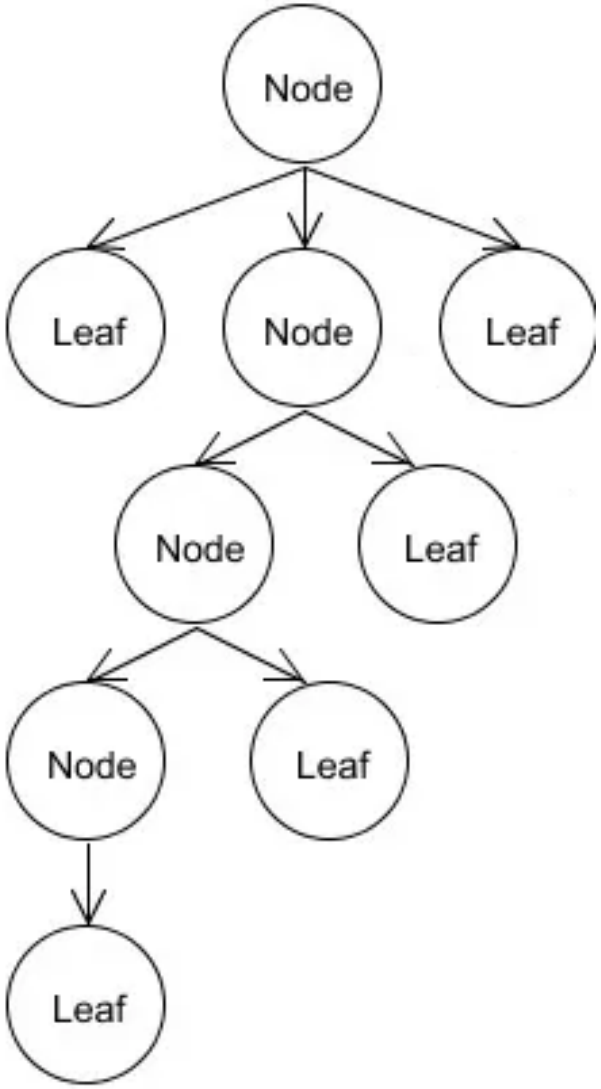
COMPOSITE

Utilizado quando sua Composição deve conter diversos objetos e ele mesmo, permitindo navegação entre os níveis. Como exemplo, um sistema de gerenciamento de arquivos onde existem os arquivos concretos (vídeos, textos, imagens, etc.) e arquivos pastas que armazenam outros arquivos.



COMPOSITE

Como uma estrutura de árvore temos os Nós e as Folhas. No padrão Composite os arquivos concretos são chamados de Folhas, pois não possuem filhos e os arquivos pasta são chamados de Nós, pois possuem filhos e fornecem operações sobre esses filhos.



COMPOSITE

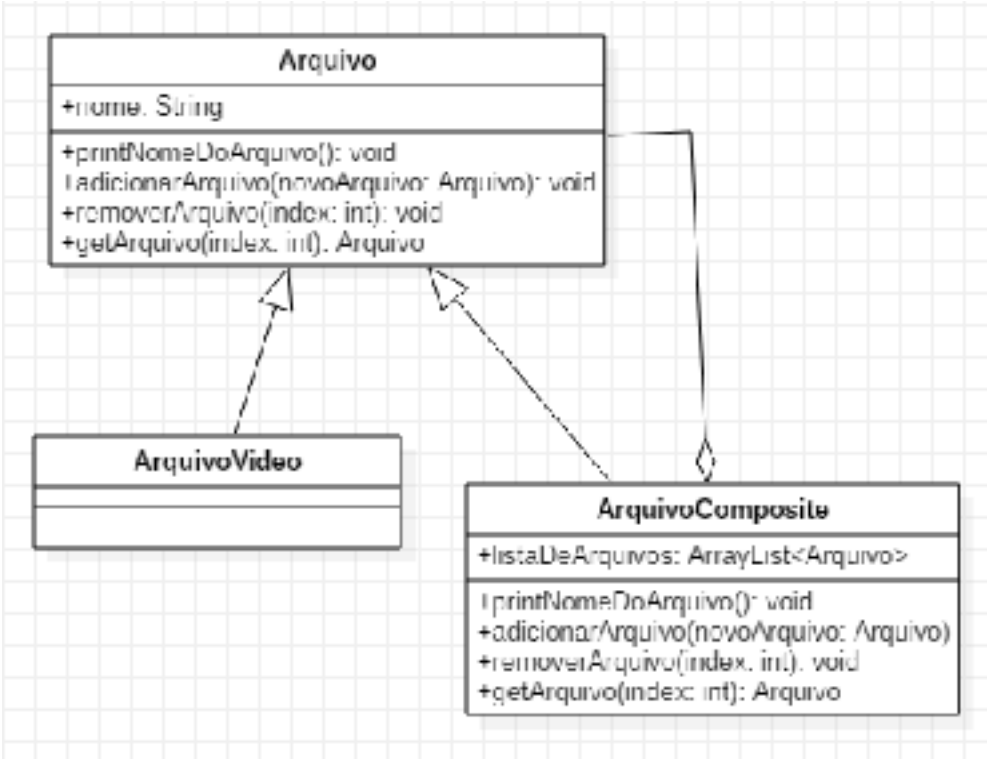
O mal tratamento destas exceções podem gerar problemas de segurança e ai surge uma outra forma de implementar o padrão, restringindo a interface comum dos objetos.

Para isto basta remover os métodos de gerenciamento de arquivos (adicionar, remover, etc) da classe base, assim apenas os arquivos pastas teriam estes métodos.

Em contrapartida o usuário do código precisa ter certeza se um dado objeto é Pasta para realizar um cast e chamar os métodos da pasta. Veja o método main que utiliza esta implementação:

COMPOSITE

Exemplo Composite:

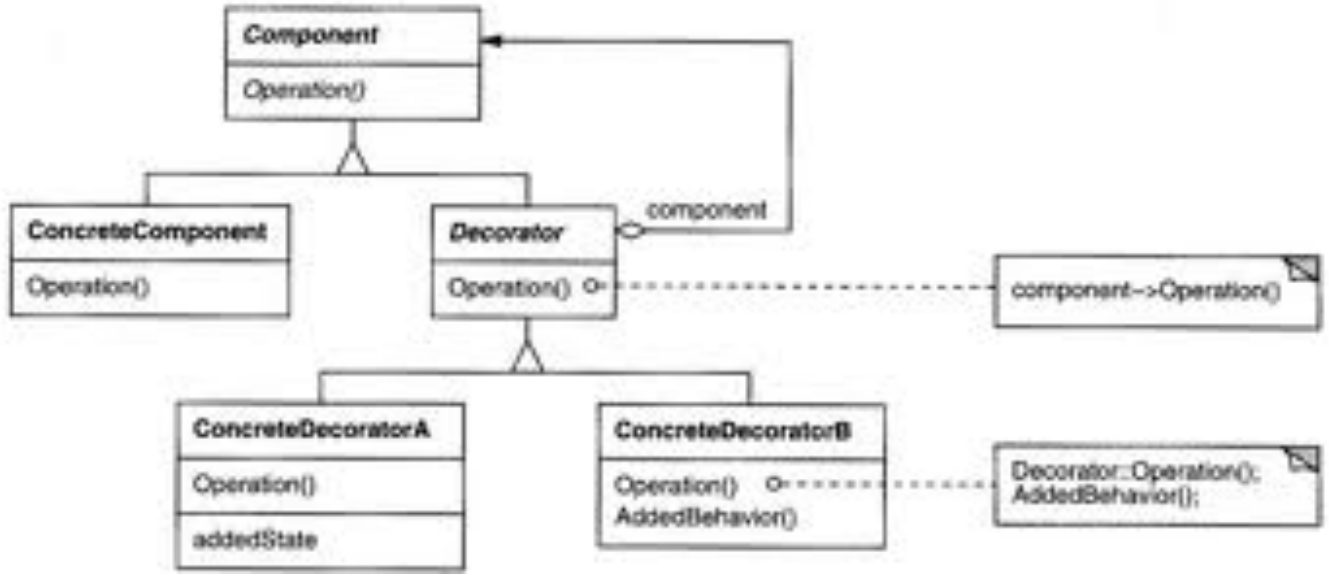


EXERCÍCIO COMPOSITE

Crie uma árvore genealógica onde os Pais não podem ter filhos, apenas as mães com as Classes Homem e Mulher.

DECORATOR

Os decoradores fornecem uma alternativa flexível de subclasse para estender a funcionalidade".



DECORATOR

O Padrão Decorator tem como característica o seguinte:

Os decoradores têm o mesmo supertipo que os objetos que eles decoram;

Você pode usar um ou mais decoradores para englobar um objeto;

Uma vez que o decorador tem o mesmo supertipo que o objeto decorado, podemos passar um objeto decorado no lugar do objeto original (englobado);

O decorador adiciona seu próprio comportamento antes e/ou depois de delegar o objeto que ele decora o resto do trabalho;

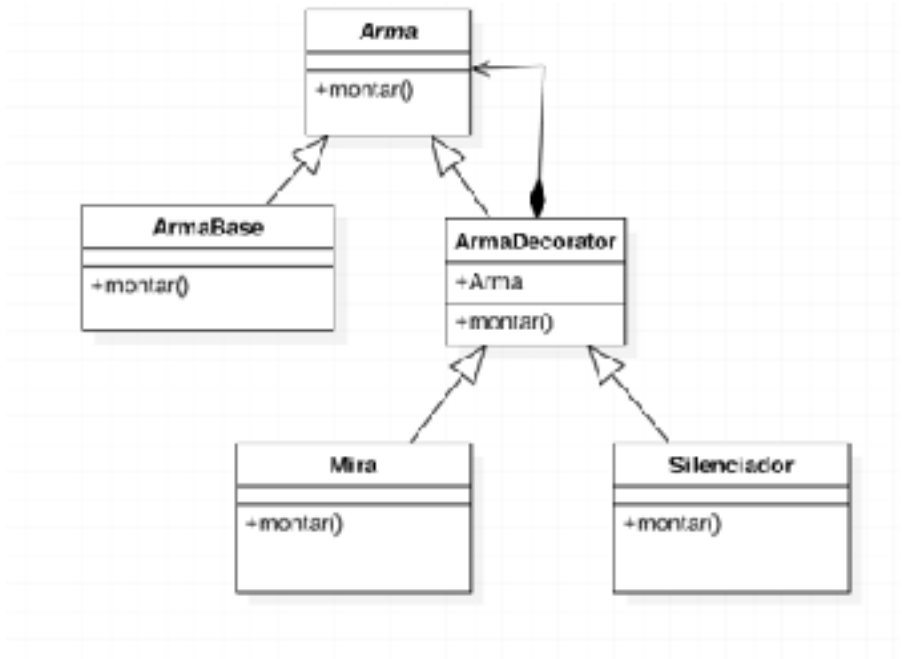
Os objetos podem ser decorados a qualquer momento, então podemos decorar os objetos de maneira dinâmica no tempo de execução com quantos decoradores desejarmos.

DECORATOR

O padrão Decorator usa a herança apenas para ter uma correspondência de tipo e não para obter o comportamento. Assim, quando compõe-se um decorador com um componente, adiciona-se um novo comportamento, nota-se que estamos adquirindo um novo comportamento e não herdando-o de alguma superclasse. Isso nos dá muito mais flexibilidade para compor mais objetos sem alterar uma linha de código, tudo em tempo de execução e não em tempo de compilação como ocorre com a herança. Todos esses benefícios nos são disponibilizados pelo uso do padrão Decorator. Uma desvantagem do padrão é que teremos inúmeras classes pequenas que pode ser bastante complicado para um desenvolvedor que está tentando entender o funcionamento da aplicação. Assim, precisamos avaliar esses casos e optar por uma solução que de repente não seja usando decoradores.

DECORATOR

Exemplo do Padrão Decorator:

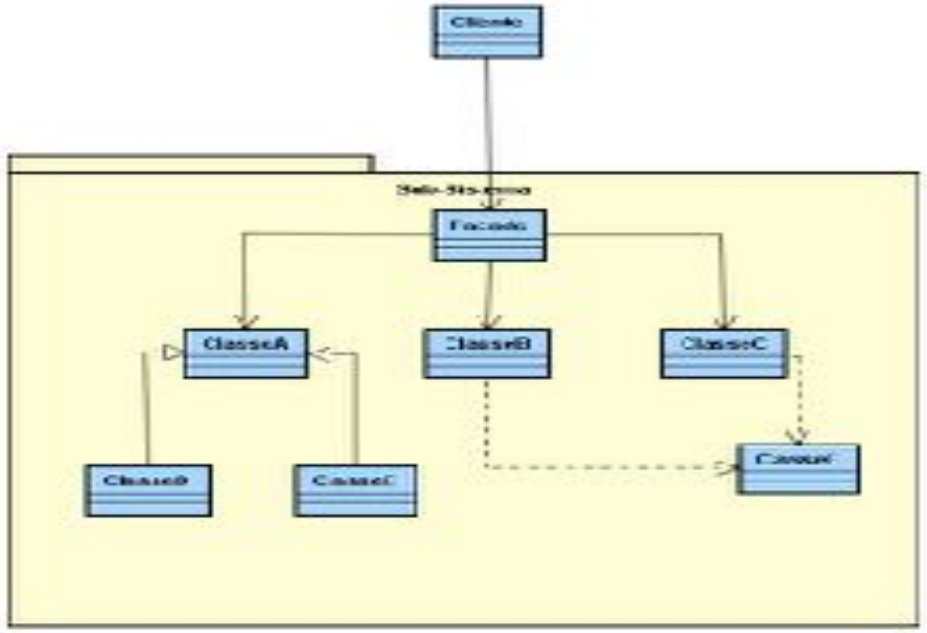


EXERCÍCIO DECORADOR

Crie uma classe Comida e classes que herdem esta (Burguer, Pizza, Massa) e Ingredientes que podem incrementar a Comida (Queijo, Bacon, Catupiry ...)

FACADE

Um Façade é um objeto que provê uma interface simplificada para um corpo de código maior, como por exemplo, uma biblioteca de classes.

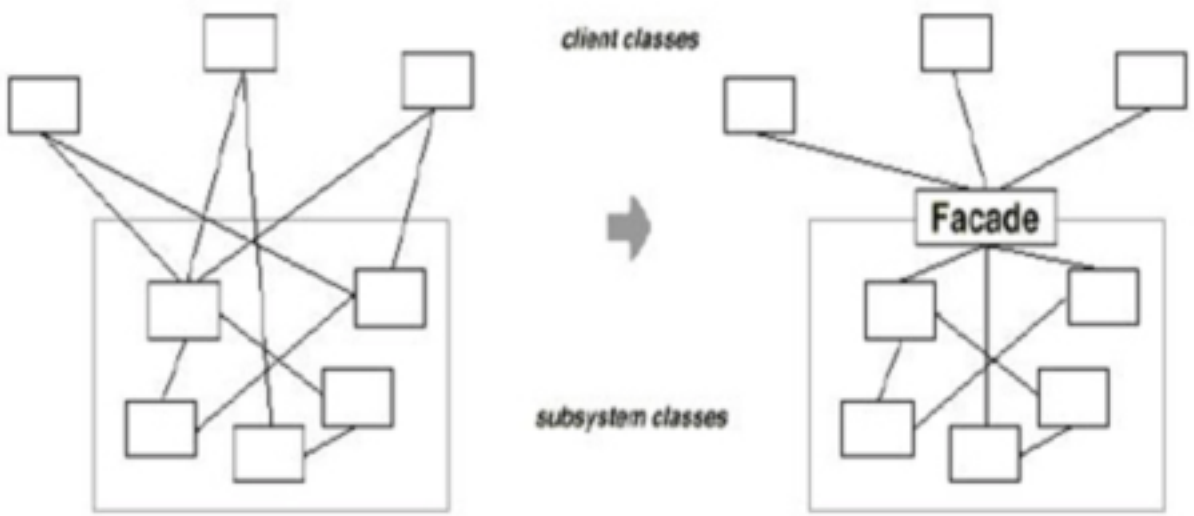


FACADE

O problema com essa centralização da complexidade é que a classe fachada pode crescer descontroladamente para abrigar uma conjunto grande de possibilidades. Nestes casos pode ser mais viável procurar outros padrões, como Abstract Factory para dividir as responsabilidades entre subclasses.

FACADE

Exemplo de Facade:



EXERCÍCIO FACADE

Ao efetuar uma compra, a mesma é paga pela Contabilidade, validada por Qualidade, embrulhada pela Logística, Movimentada pela Transportadora e Entregue pelos Correios. Implemente o Facade deste Processo apenas pela opção “Comprar”.

FLYWEIGHT

"Usar compartilhamento para suportar eficientemente grandes quantidades de objetos de granularidade fina." [1]

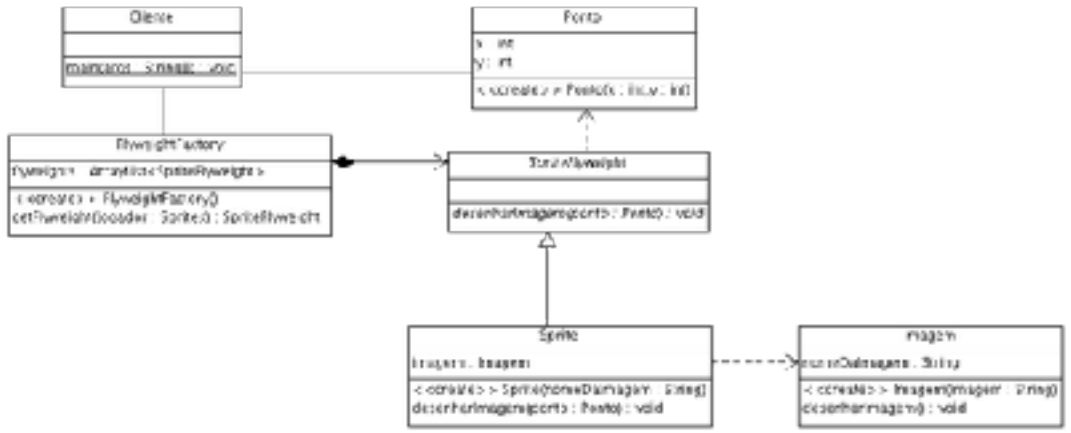
Pela intenção percebemos que o padrão Flyweight cria uma estrutura de compartilhamento de objetos pequenos. Para o exemplo citado, o padrão será utilizado no compartilhamento de imagens entre as entidades.

FLYWEIGHT

O ponto fraco do padrão é que, dependendo da quantidade e da organização dos objetos a serem compartilhados, pode haver um grande custo para procura dos objetos compartilhados. Então ao utilizar o padrão deve ser analisado qual a prioridade no projeto, espaço ou tempo de execução.

FLYWEIGHT

Exemplo de Flyweight:



EXERCÍCIO FLYWEIGHT

Você está gerenciando arquivos entre servidores, uma lista de arquivos estão disponíveis, onde não é necessário que o cliente receba este arquivo, apenas que trabalhe movimentando os arquivos entre servidores. Implemente com o padrão Flyweight.

PROXY

“Fornecer um substituto ou marcador da localização de outro objeto para controlar o acesso a esse objeto.”

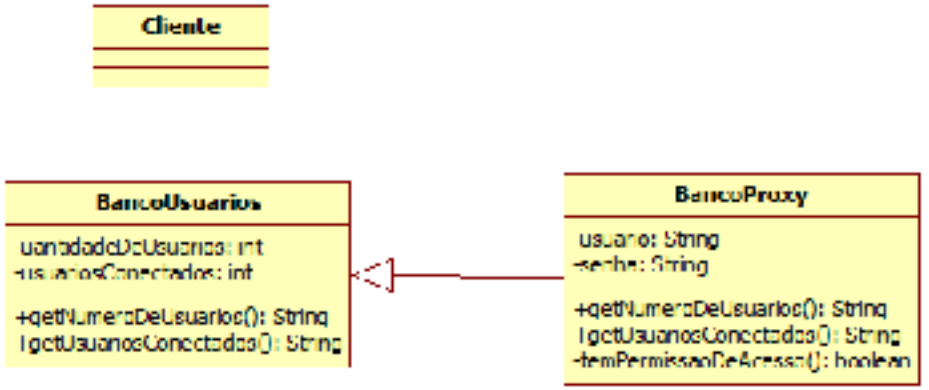
Ao implementar uma classe, pode ser desejável efetuar validações anteriores ou acessar através de outro local ou pacote, este padrão facilita esta comunicação.

PROXY

- **Protection Proxy**: Controlam o acesso aos objetos, por exemplo, verificando se quem chama possui a devida permissão.
- **Virtual Proxy**: mantem informações sobre o objeto real, adiando o acesso/criação do objeto em si.
- **Remote Proxy**: fornece um representante local para um objeto em outro espaço de endereçamento.
- **Smart Reference**: este proxy é apenas um substituto simples para executar ações adicionais quando o objeto é acessado.

PROXY

Exemplo de Proxy:



EXERCÍCIO PROXY

Você está implementando uma chamada ao site da Receita Federal para pagar um imposto.
pagarFisico(ValorDocumento, ValorImposto, Cpf)
pagarJuridico(ValorDocumento, ValorImposto, Cnpj)
Implemente um exemplo de Proxy para este caso.