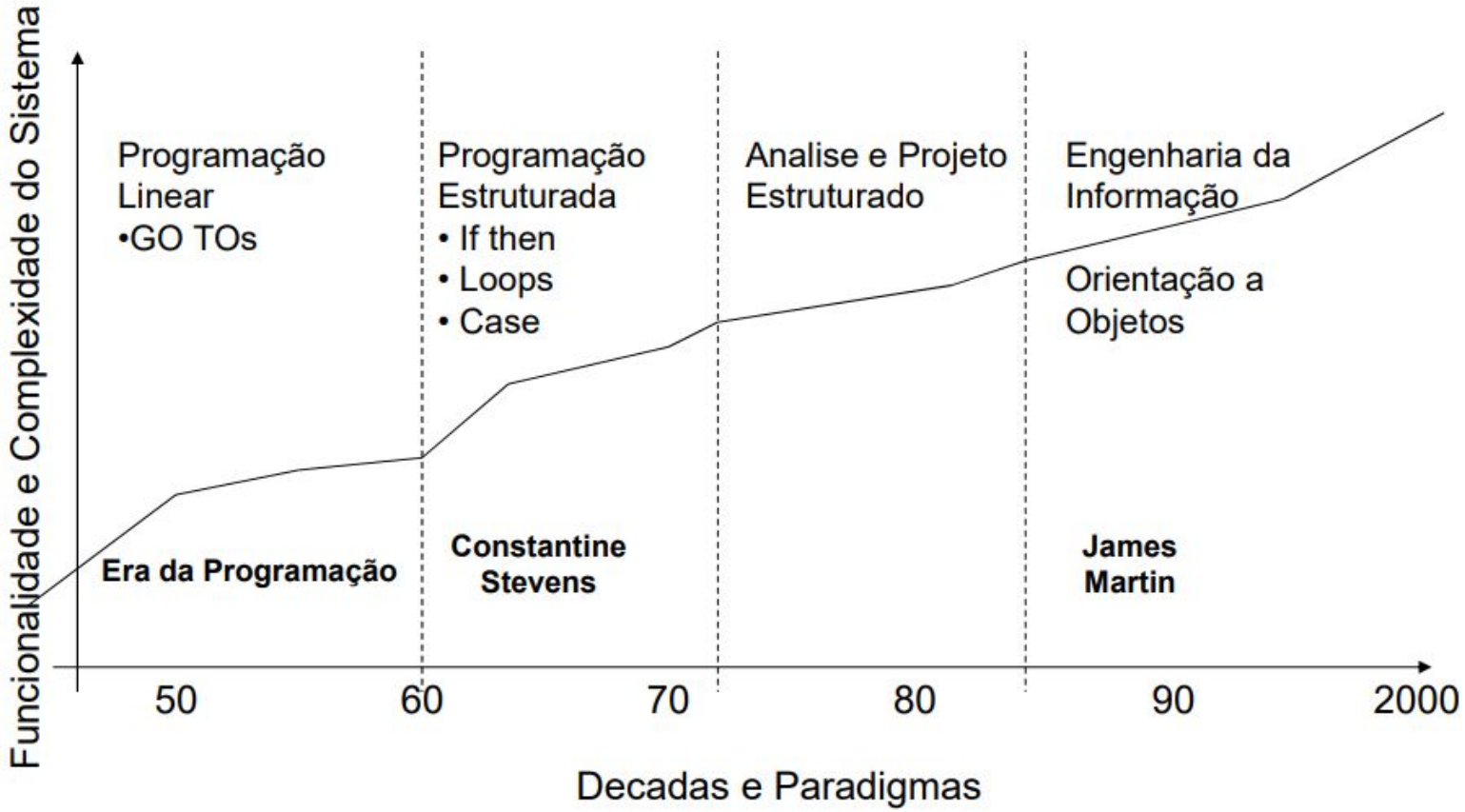
Desenvolvimento para dispositivos móveis

Introdução à Orientação a Objetos em Kotlin

Professor Msc. Fabio Pereira da Silva
E-mail: fabio.pereira@faculdadeimpacta.com.br

Evolução


Funcionalidade e Complexidade do Sistema
Programação Linear
•GO TOs
Programação Estruturada
• If then
• Loops
• Case
Analise e Projeto Estruturado
Engenharia da Informação
Orientação a Objetos
Era da Programação
Constantine Stevens
James Martin
50
60
70
80
90
2000
Decadas e Paradigmas

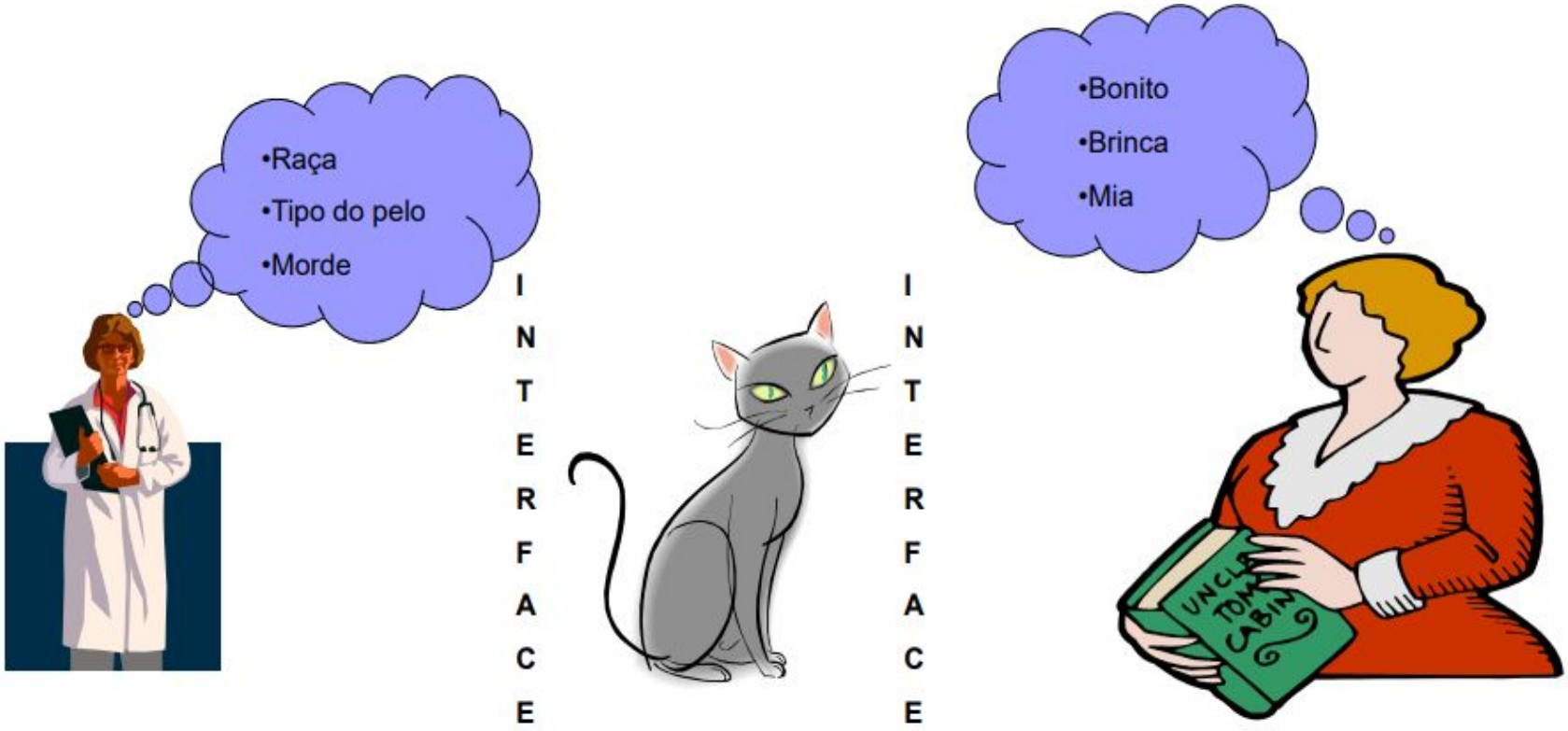
O que é a orientação a objetos

- A orientação a Objetos visa **abstrair** características de entidades concretas e abstratas do mundo real
- Criando padrões ou classificações que posteriormente serão transformados em objetos virtuais.

O que é a orientação a objetos

- Todo objeto é composto por:
- Características, também conhecidas como propriedades, atributos, status. As características de um objeto guardam seu estado.
- Comportamentos, também conhecidos como métodos, funções e ações. Os comportamentos são ações que o objeto pode executar.

O que é abstração?



Nota : Duas ou mais pessoas podem ver características distintas em um mesmo objeto. O objeto comporta ambas as características porém cada uma das pessoas visualiza aquilo que mais atendem a sua realidade, mediante aos seus próprios filtros internos. Isso chama-se **abstração.**

Introdução

- O **modelo de classes de projeto** é resultante de refinamentos no modelo de classes de análise.
- Esse modelo é construído em paralelo com o **modelo de interações**.
  - A construção do MI gera informações para a transformação do modelo de classes de análise no modelos de classes de projeto.
- O modelo de classes de projeto contém detalhes úteis para a **implementação** das classes nele contidas.

Introdução

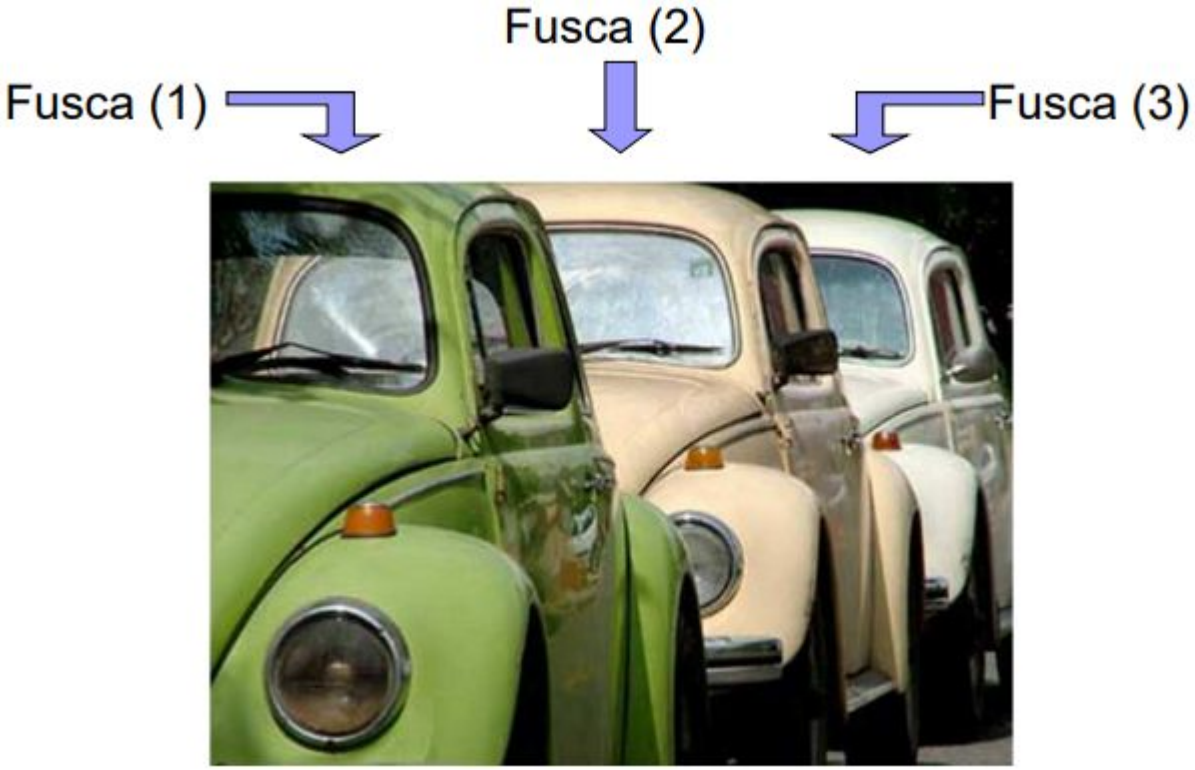
- Aspectos a serem considerados na fase de projeto para a modelagem de classes:
  - Estudo de novos elementos do diagrama de classes que são necessários à construção do modelo de projeto.
  - Descrever transformações pelas quais passam as classes e suas propriedades com o objetivo de transformar o modelo de classes análise no modelo de classes de projeto.
  - Adição de novas classes ao modelo
  - Especificação de atributos, operações e de associações
  - Descrever refinamentos e conceitos relacionados à herança, que surgem durante a modelagem de classes de projeto
    - classes abstratas, interfaces, polimorfismo e padrões de projeto.
  - Utilização de padrões de projeto (*design patterns*)

Classes e objetos

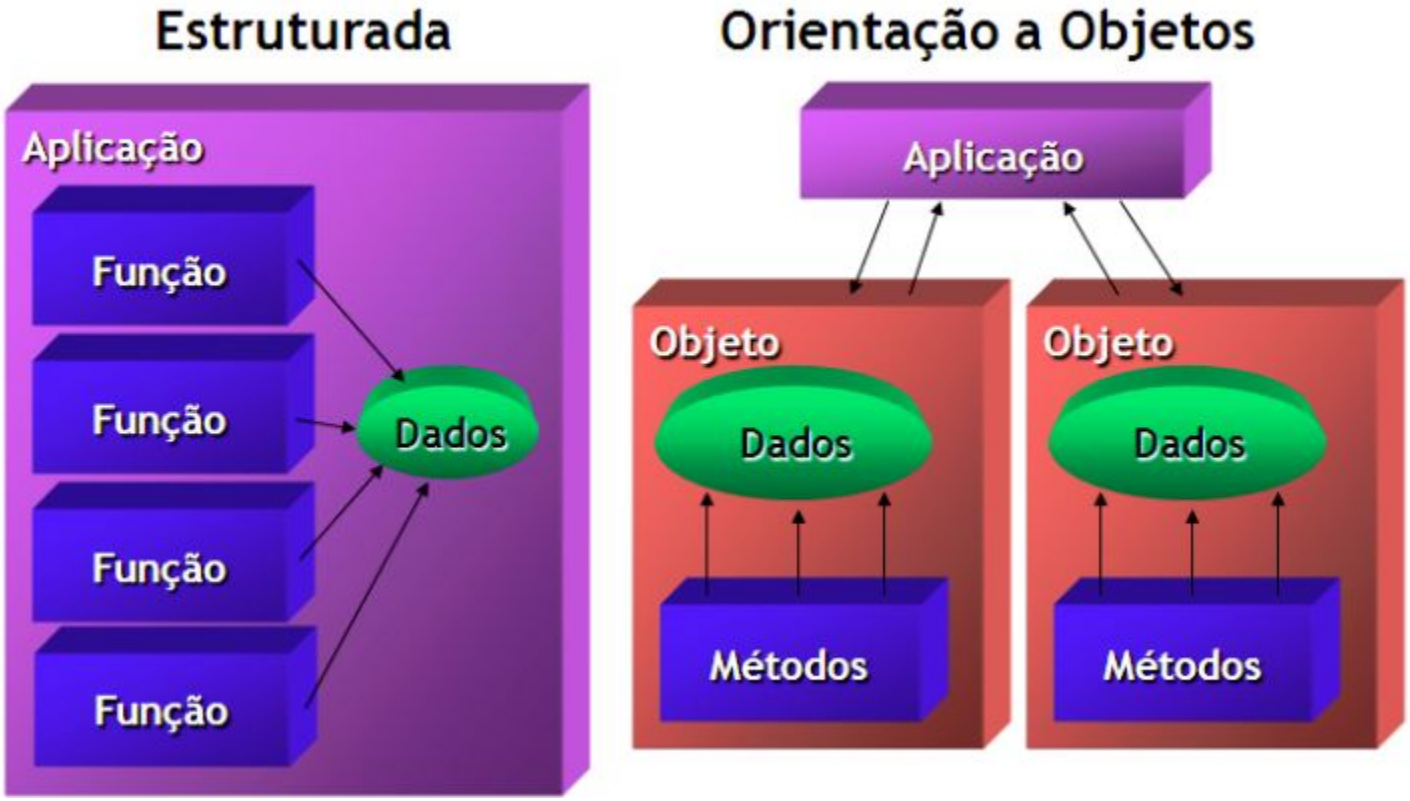
- Classe é um conjunto de objetos com características e comportamentos afins.

Classes e objetos

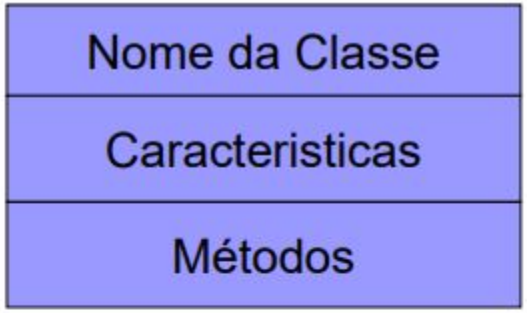
- Objetos: é um elemento concreto de um tipo de classe.



Paradigma Estruturado x Orientado a Objetos



Classes e objetos

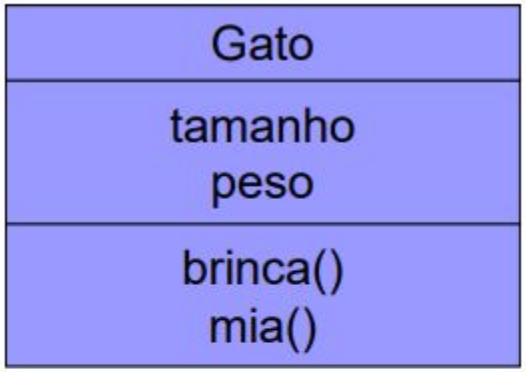


- A sintaxe para criar uma classe é :

```
<modificador> class <nome da classe> {
        <tipo> <nome da propriedade 1>;
        <tipo> <nome da propriedade 2>;
        ..
        <tipo> <nome da propriedade N>;

        <tipo de retorno> <nome do método 1> ( <parâmetros> ) {
                <Código a ser executado linha 1>;
                <Código a ser executado linha 2>;
                ...
        }

        <tipo de retorno> <nome do método N> ( <parâmetros> ) {
                <Código a ser executado linha 1>;
                <Código a ser executado linha 2>;
                ...
        }
}
```

- **O <modificador>** pode ser **public** de forma que todos vejam, ou **package** visível apenas pelas demais classes do pacote (para atribuir como package basta não colocar nada).

Classes e objetos




```
public class Gato {
        float tamanho;
        float peso;

        void brinca() {
                System.out.println("brincando...");
        }

        void mia() {
                System.out.println("miando...");
        }
}
```

Classes - Instanciando

- Instanciar uma classe é o ato de criar um objeto a partir dela.
- A classe contém os tipos das características e os comportamentos que os objetos vão possuir.
- Mas cada objeto possuirá conteúdos diferentes em suas características, e o comportamento de cada um irá variar conforme estes conteúdos.

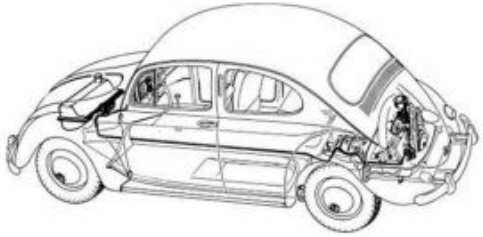
Classes - Instanciando

- A sintaxe para instanciar uma classe é:

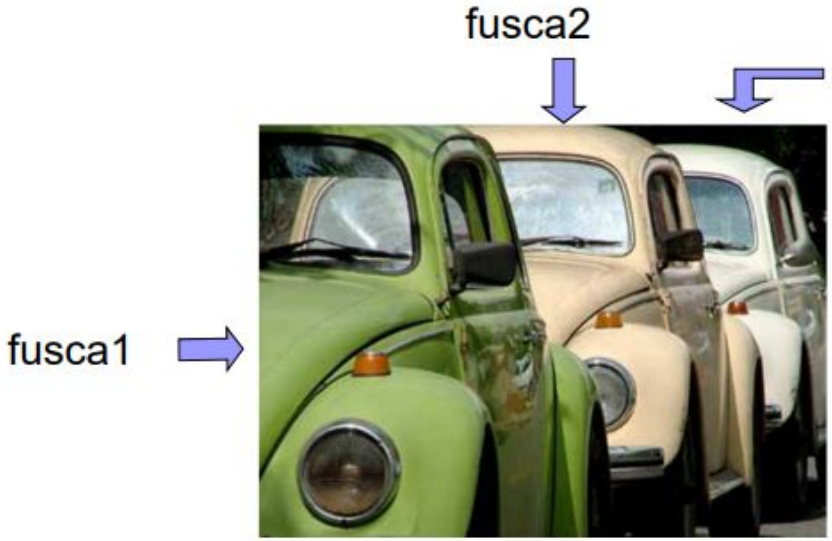
```
<nome da classe> <nome da variável> = new <nome da classe>(<parâmetros>);
```

- Exemplo: // Para instanciar um objeto do tipo Gato usaremos : Gato felix = new Gato();

Classes - Instanciando



Classe



- A partir de uma mesma classe é possível criar diversos objetos.

Classe - Instanciando

- A classe carro mostrada na página anterior é apenas a planta (o desenho), mostrando como os objetos instanciados serão e como se comportarão:

```
public class Carro {
        float autonomia;
        int maxKmHora;
        String Marca;
        String Modelo;
        int ano;
        int velocidade;

        public void acelerar() {
             System.out.println(" Acelerando .,. ");
              velocidade = velocidade + 10;
        }

        public void frear() {
             System.out.println(" Acelerando ... ");
              velocidade = velocidade – 10;
        }
}
```

Classe - Instanciando

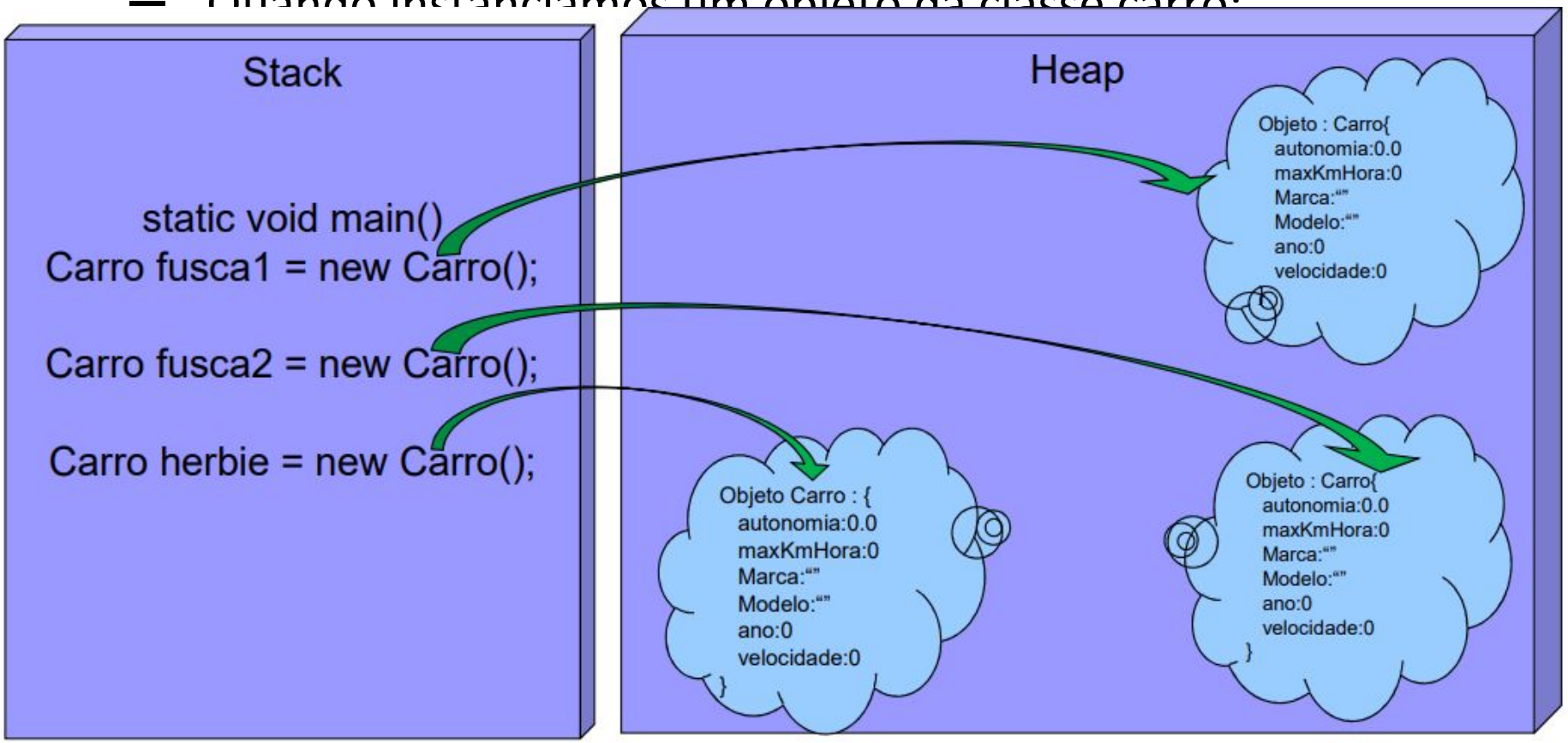
- Para se criar os objetos apartir da classe é preciso instanciá-los, conforme no exemplo abaixo: // Para instanciar um objeto do tipo Carro usaremos :
- Carro fusca1 = new Carro ();
- Carro fusca2 = new Carro ();
- Carro herbie = new Carro ();

Objetos

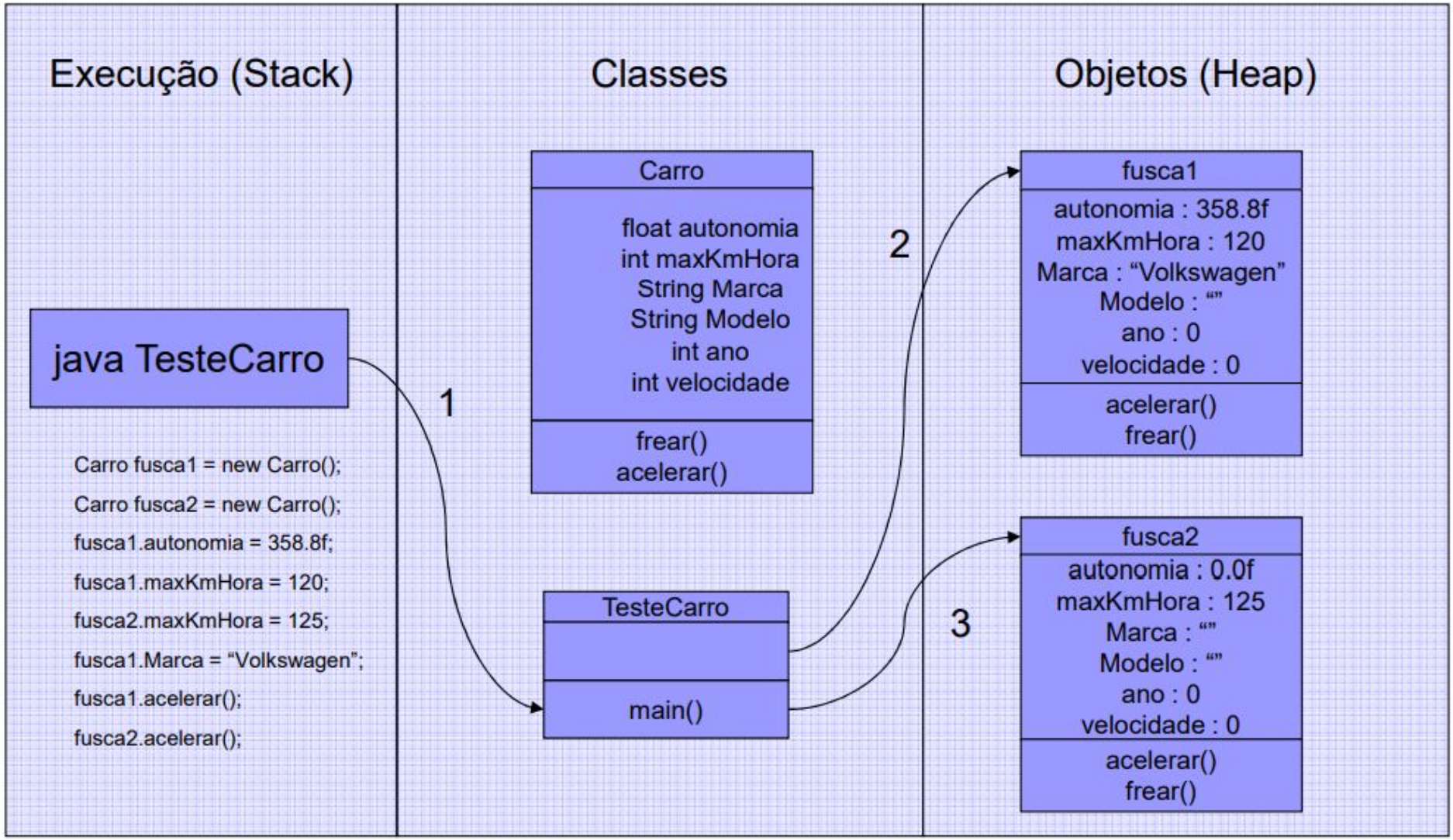
- A partir do momento em que a classe for instanciada, o java criará mais um objeto na memória, que poderá conter informações diferentes em suas características (propriedades).
- Para alterar o conteúdo de uma característica do objeto é preciso referenciar sua propriedade e atribuir uma nova informação.
- Lembre-se propriedades e características são a mesma coisa.

Objetos (Stack e Heap)

- Comportamento dos objetos e variáveis de referência na memória
- Quando instanciamos um objeto da classe carro:



Objetos


Execução (Stack)
Classes
Objetos (Heap)
java TesteCarro
Carro fusca1 = new Carro();
Carro fusca2 = new Carro();
fusca1.autonomia = 358.8f;
fusca1.maxKmHora = 120;
fusca2.maxKmHora = 125;
fusca1.Marca = "Volkswagen";
fusca1.acelerar();
fusca2.acelerar();
1
Carro
float autonomia
int maxKmHora
String Marca
String Modelo
int ano
int velocidade
frear()
acelerar()
TesteCarro
main()
2
fusca1
autonomia : 358.8f
maxKmHora : 120
Marca : "Volkswagen"
Modelo : ""
ano : 0
velocidade : 0
acelerar()
frear()
3
fusca2
autonomia : 0.0f
maxKmHora : 125
Marca : ""
Modelo : ""
ano : 0
velocidade : 0
acelerar()
frear()

Princípios fundamentais

Abstração e princípios fundamentais

- **Abstração**: Transformar aquilo que observamos na realidade para o meio computacional.
- **Estado**: Atributos (Características)
- **Operações**: Métodos (Comportamentos)
- **Identidade**
  - Dois objetos com estados e operações precisamente idênticos não são iguais
- **Operações podem mudar os valores dos atributos, assim mudando o estado do objeto.**

Refletindo

- Cite 4 atributos de um aluno
  - ??????????
  - ??????????
  - ??????????
  - ??????????
- Cite 3 métodos de um aluno
  - ??????????
  - ??????????
  - ??????????

Especificação de classes de fronteira

- Não devemos atribuir a essas classes responsabilidades relativas à lógica do negócio.
  - Classes de fronteira devem apenas servir como um ponto de captação de informações, ou de apresentação de informações que o sistema processou.
  - A única inteligência que essas classes devem ter é a que permite a elas realizarem a comunicação com o ambiente do sistema.
- Há diversas razões para isso:
  - Em primeiro lugar, se o sistema tiver que ser implantado em outro ambiente, as modificações resultantes sobre seu funcionamento propriamente dito seriam mínimas.
  - Além disso, o sistema pode dar suporte a diversas formas de interação com seu ambiente (e.g., uma interface gráfica e uma interface de texto).
  - Finalmente, essa separação resulta em uma melhor ***coesão***.

Especificação de classes de entidade

- A maioria das classes de entidade normalmente permanece na passagem da análise ao projeto.
  - Na verdade, classes de entidade são normalmente as primeiras classes a serem identificadas, na análise de domínio.
- Durante o projeto, um aspecto importante a considerar sobre classes de entidade é identificar quais delas geram objetos que devem ser persistentes.
  - Para essas classes, o seu mapeamento para algum mecanismo de armazenamento persistente deve ser definido (Capítulo 12).
- Um aspecto importante é a forma de representar associações, agregações e composições entre objetos de entidade.
  - Essa representação é função da navegabilidade e da multiplidade definidas para a associação, conforme visto mais adiante.

Especificação de classes de entidade

- Outro aspecto relevante para classes de entidade é modo como podemos identificar cada um de seus objetos unicamente.
  - Isso porque, principalmente em sistemas de informação, objetos de entidade devem ser armazenados de modo persistente.
  - Por exemplo, um objeto da classe Aluno é unicamente identificado pelo valor de sua matrícula (um atributo do domínio).
- A manipulação dos diversos atributos identificadores possíveis em uma classes pode ser bastante trabalhosa.
- Para evitar isso, um ***identificador de implementação*** é criado, que não tem correspondente com atributo algum do domínio.
  - Possibilidade de manipular identificadores de maneira uniforme e eficiente.
  - Maior facilidade quando objetos devem ser mapeados para um SGBDR

Especificação de classes de controle

- Com relação às classes de controle, no projeto devemos identificar a real utilidade das mesmas.
  - Em casos de uso simples (e.g., manutenção de dados), classes de controle não são realmente necessárias. Neste caso, classes de fronteira podem repassar os dados fornecidos pelos atores diretamente para as classes de entidade correspondentes.
- Entretanto, é comum a situação em que uma classe de controle de análise ser transformada em duas ou mais classes no nível de especificação.
- No refinamento de qualquer classe proveniente da análise, é possível a aplicação de padrões de projeto (*design patterns*)

Especificação de classes de controle

- Normalmente, cada classe de controle deve ser particionada em duas ou mais outras classes para controlar diversos aspectos da solução.
  - Objetivo: de evitar a criação de uma única classe com baixa coesão e alto acoplamento.
- Alguns exemplos dos aspectos de uma aplicação cuja coordenação é de responsabilidade das classes de controle:
  - produção de valores para preenchimento de controles da interface gráfica,
  - autenticação de usuários,
  - controle de acesso a funcionalidades do sistema, etc.

Especificação de classes de controle

- Um tipo comum de controlador é o ***controlador de caso de uso***, responsável pela coordenação da realização de um caso de uso.
- As seguintes responsabilidades são esperadas de um controlador de caso de uso:
  - Coordenar a realização de um caso de uso do sistema.
  - Servir como canal de comunicação entre objetos de fronteira e objetos de entidade.
  - Se comunicar com outros controladores, quando necessário.
  - Mapear ações do usuário (ou atores de uma forma geral) para atualizações ou mensagens a serem enviadas a objetos de entidade.
  - Estar apto a manipular exceções provenientes das classes de entidades

Encapsulamento

- Um objeto, em um programa, “encapsula” todo o seu estado e o comportamento;
- Os dados e as operações são agrupados e a sua implementação é escondida, protegida dos usuários;

Visibilidade e Encapsulamento

- Os três qualificadores de visibilidade aplicáveis a atributos também podem ser aplicados a operações.

  + representa visibilidade pública

  # representa visibilidade protegida

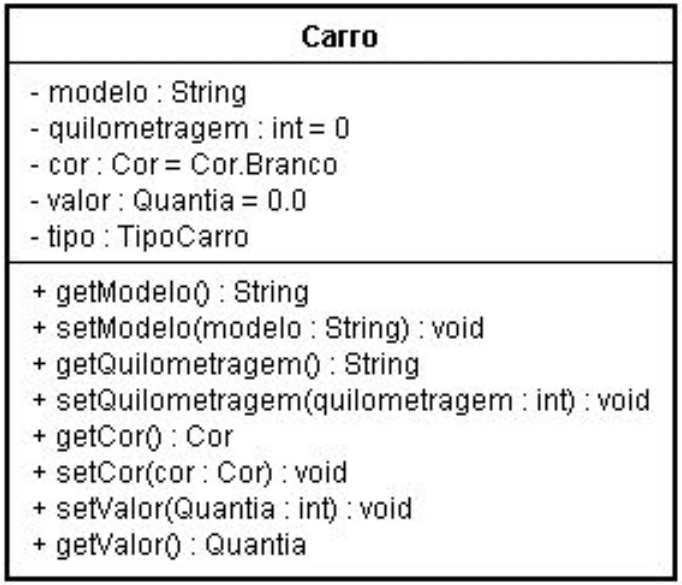
  - representa visibilidade privativa

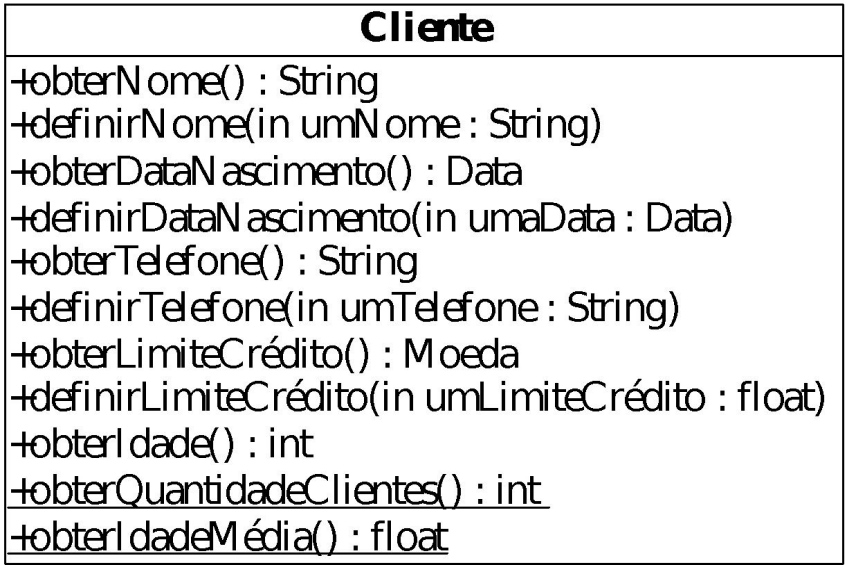
- O real significado desses qualificadores depende da linguagem de programação em questão.

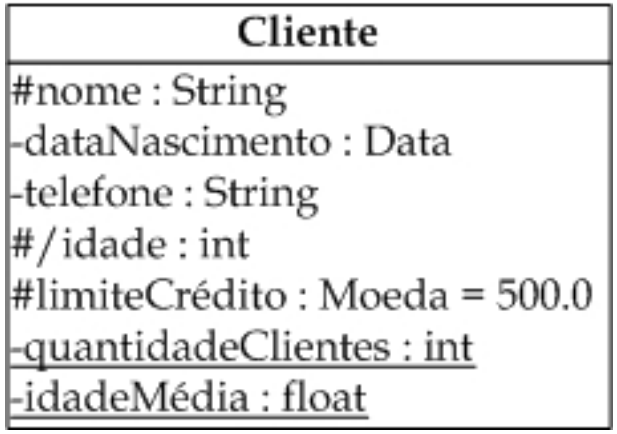
- Usualmente, o conjunto das operações públicas de uma classe são chamadas de **interface** dessa classe.

  - Note que há diversos significados para o termo interface.

Sintaxe para atributos e operações







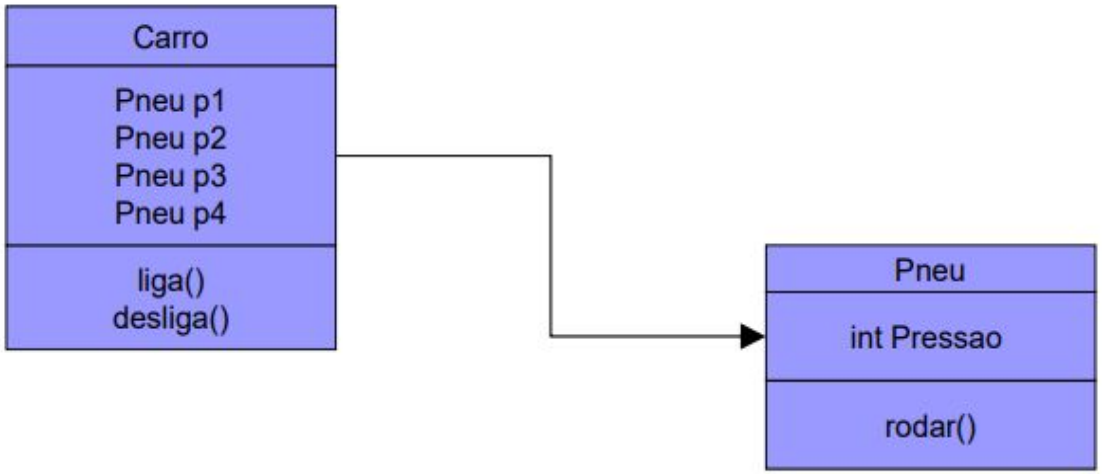
Obs.: classes utilitárias podem ser utilizadas como tipos para atributos. (e.g., Moeda, Quantia, TipoCarro, Endereco)

Projeto de métodos

- Métodos de construção (criação) e destruição de objetos
- Métodos de acesso (getX/setX) ou propriedades
- Métodos para manutenção de associações (conexões) entre objetos.
- Outros métodos:
  - Valores derivados, formatação, conversão, cópia e clonagem de objetos, etc.
- Alguns métodos devem ter uma operação inversa óbvia
  - e.g., habilitar e desabilitar; tornarVisível e tornarInvisível; adicionar e remover; depositar e sacar, etc.
- Operações para desfazer ações anteriores.
  - e.g., padrões de projeto GoF: Memento e Command

Associação

- Associação ocorre quando uma classe possui atributos do tipo de outra classe.



Nota : Neste caso estamos dizendo que carro possui pneu (4 pneus)

Associação

- A associação pode ser representada em Java da seguinte forma

```
public class Pneu {
        int Pressao;

        void roda() {
                System.out.println("Pneu em movimento");
        }
}
```

```
public class Carro {
        Pneu p1;
        Pneu p2;
        Pneu p3;
        Pneu p4;

        void liga() {
                System.out.println("Carro ligado");
        }

        void desliga() {
                System.out.println("Carro desligado");
        }
}
```

Descreva todas as classes, atributos e associações do domínio a seguir:

Elenque as classes, atributos e associações para um sistema de venda de passagens aéreas web e mobile, equivalente ao módulo de compra de passagem por um cliente, levando em consideração as situações abaixo. Cada classe de entidade deve conter pelo menos um atributo e / ou uma operação.

Um cliente pode ser passageiro de muitos voos. No entanto, uma passagem refere-se a um cliente específico. A empresa mantém um cadastro de todos os clientes que já foram passageiros de algum voo;

Um voo pode ter muitos passageiros; no entanto, cada passagem refere-se exclusivamente a um voo específico;

O cliente efetua o pagamento da passagem somente por meio de cartão de crédito ou boleto bancário;

Um voo pode fazer escalas em diversos aeroportos, ou seja, pode ter diversos destinos, e um aeroporto pode ser o destino de muitos voos. As empresas mantêm um cadastro de todos os aeroportos para onde oferece voos;

Um aeroporto pode ser a origem ou o destino de muitas escalas, no entanto, um determinado aeroporto só pode ser a origem ou o destino de uma determinada escala, nunca os dois ao mesmo tempo;

Um aeroporto está localizado em uma cidade específica, mas uma cidade pode possuir muitos aeroportos.

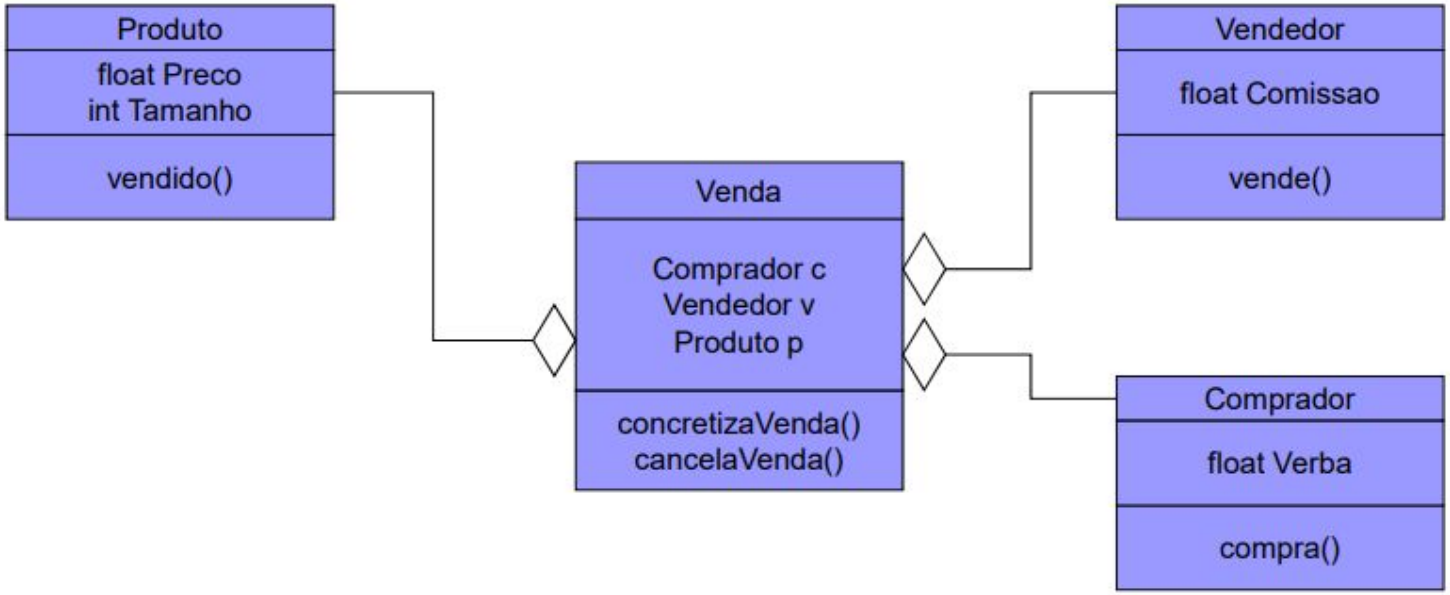
O conceito de dependência

- O ***relacionamento de dependência*** indica que uma classe depende dos serviços (operações) fornecidos por outra classe.
- Na análise, utilizamos apenas a ***dependência por atributo*** (ou estrutural), na qual a classe dependente possui um atributo que é uma referência para a outra classe.
- Entretanto, há também as ***dependências não estruturais***:
  - Na **dependência por variável global**, um objeto de escopo global é referenciado em algum método da classe dependente.
  - Na **dependência por variável local**, um objeto recebe outro como retorno de um método, ou possui uma referência para o outro objeto como uma variável local em algum método.
  - Na **dependência por parâmetro**, um objeto recebe outro

Agregação

- Ocorre quando uma classe usa outras classes em suas operações. As classes utilizadas participam da classe principal, mas a classe principal não contém estas classes utilizadas como sendo partes suas.

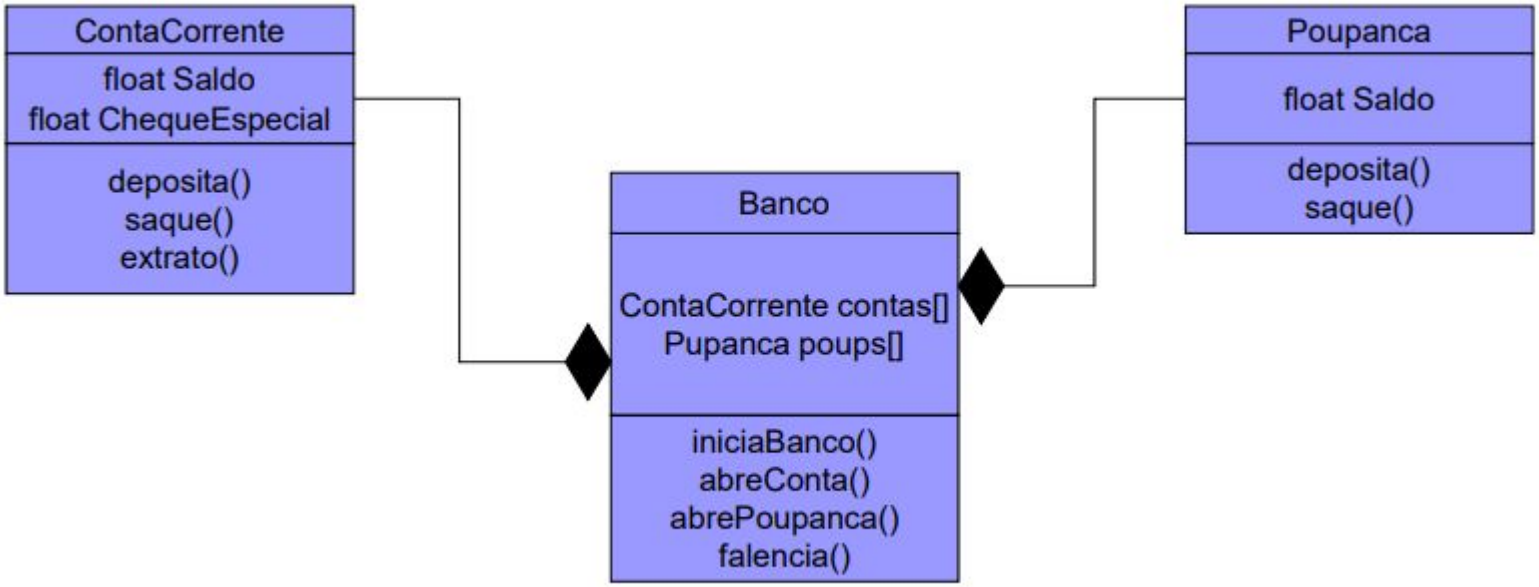
Agregação



Nota : Neste caso **Venda** é o objeto definido como sendo o **todo.** E este objeto somente pode existir caso os demais objetos que o compõem também existam.

Composição

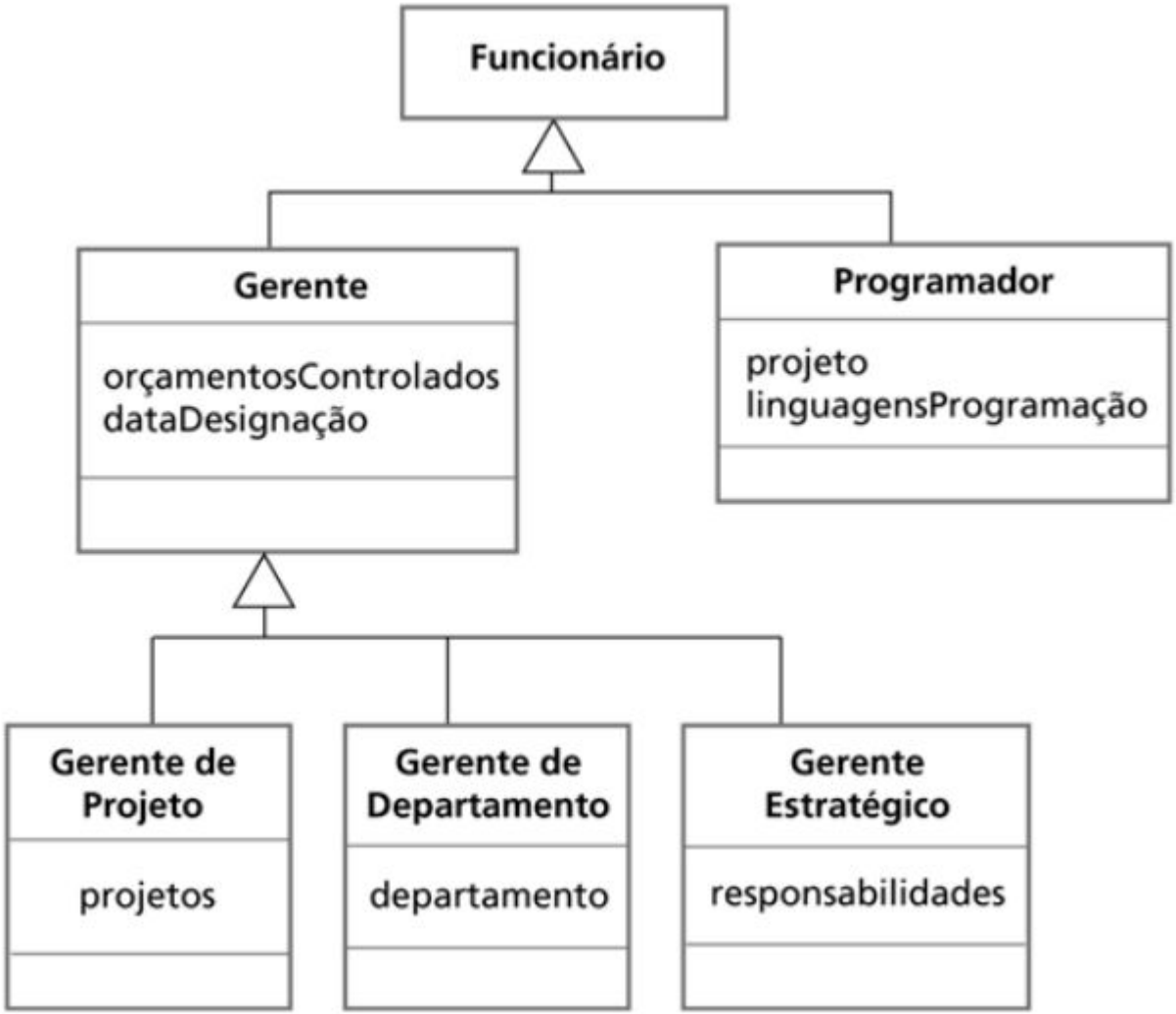
- Semelhante a agregação, a composição também é um conjunto onde há uma classe representando o todo e classes satélites funcionando como partes. Sua principal diferença ocorre que quando o objeto todo deixar de existir os seus objetos partes deverão deixar de existir também



Herança

- Herança é a capacidade de uma subclasse de ter acesso as propriedades da superclasse(também chamada classe base) relacionada a esta subclasse. Dessa forma os atributos e métodos de uma classe são propagados de cima para baixo em um diagrama de classe.
- Neste caso dizemos que a subclasse herda as propriedades e métodos da superclasse Os construtores da superclasse (classe base) não são herdados pela subclasse. A utilização da herança é um importante fator para a “reutilização de código”.

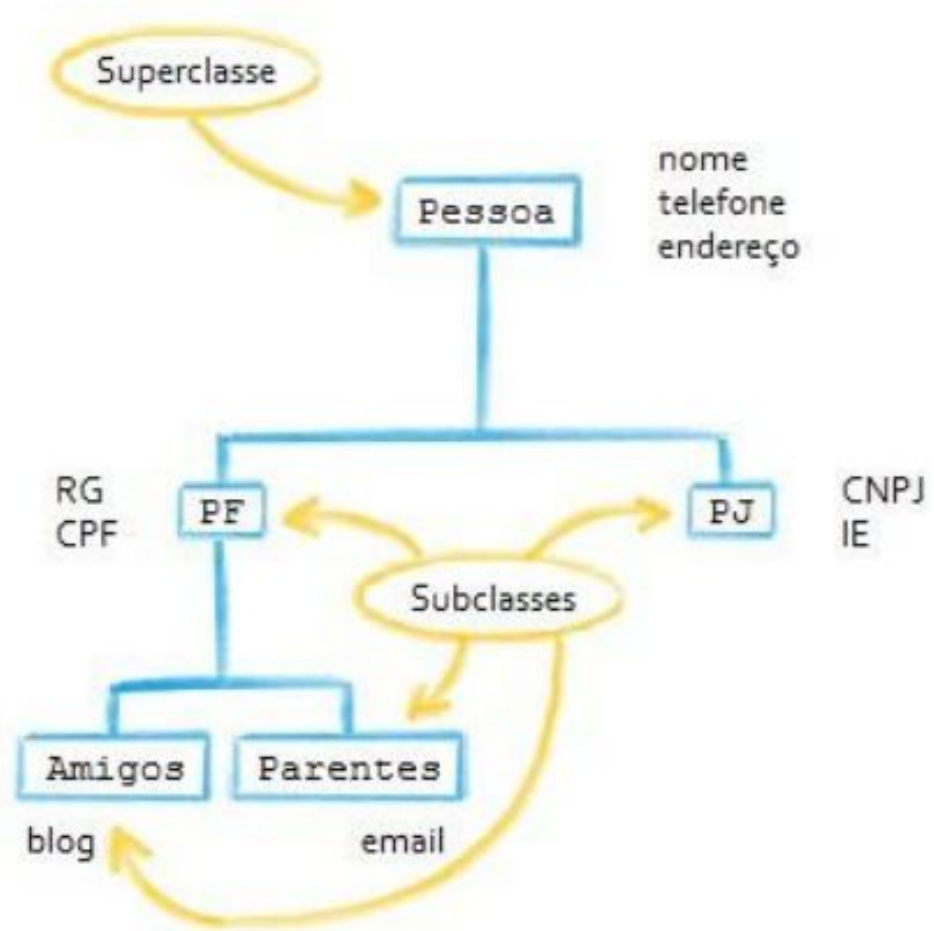
Herança


Funcionário
Gerente
orçamentosControlados
dataDesignação
Programador
projeto
linguagensProgramação
Gerente de Projeto
projetos
Gerente de Departamento
departamento
Gerente Estratégico
responsabilidades

Tipos de herança

- Com relação à quantidade de superclasses que certa classe pode ter.
  - ***herança múltipla***
  - ***herança simples***
- Com relação à forma de reutilização envolvida.
  - Na ***herança de implementação***, uma classe reusa alguma implementação de um "ancestral".
  - Na ***herança de interface*****,** uma classe reusa a interface (conjunto das assinaturas de operações) de um "ancestral" e se compromete a implementar essa interface.

Herança


Superclasse
Pessoa
nome
telefone
endereço
RG
CPF
PF
PJ
CNPJ
IE
Subclasses
Amigos
Parentes
blog
email

Classes abstratas

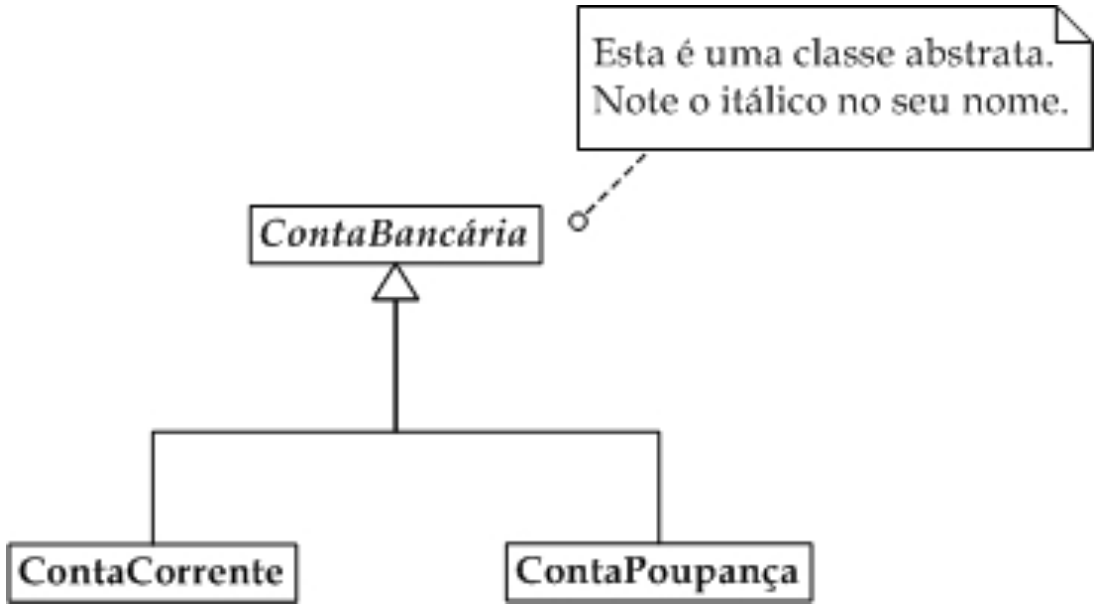
- Usualmente, a existência de uma classe se justifica pelo fato de haver a possibilidade de gerar instâncias a partir da mesma.
  - Essas classes são chamadas de **classes concretas**.
- No entanto, podem existir classes que não geram instâncias "diretamente".
  - Essas classes são chamadas de **classes abstratas**.
- Classes abstratas são usadas para organizar hierarquias gen/spec.
  - Propriedades comuns a diversas classes podem ser organizadas e definidas em uma classe abstrata a partir da qual as primeiras herdam.
- Também propiciam a implementação do **princípio do polimorfismo**.

Classes abstratas (cont)

Na UML, uma classe abstrata pode ser representada de duas maneiras alternativas:

- Com o seu nome em *itálico*.
- Qualificando-a com a propriedade *{abstract}*

Exemplo:



Operações abstratas

Uma classe abstrata possui ao menos uma ***operação abstrata***, que corresponde à especificação de um serviço que a classe deve fornecer (sem método).

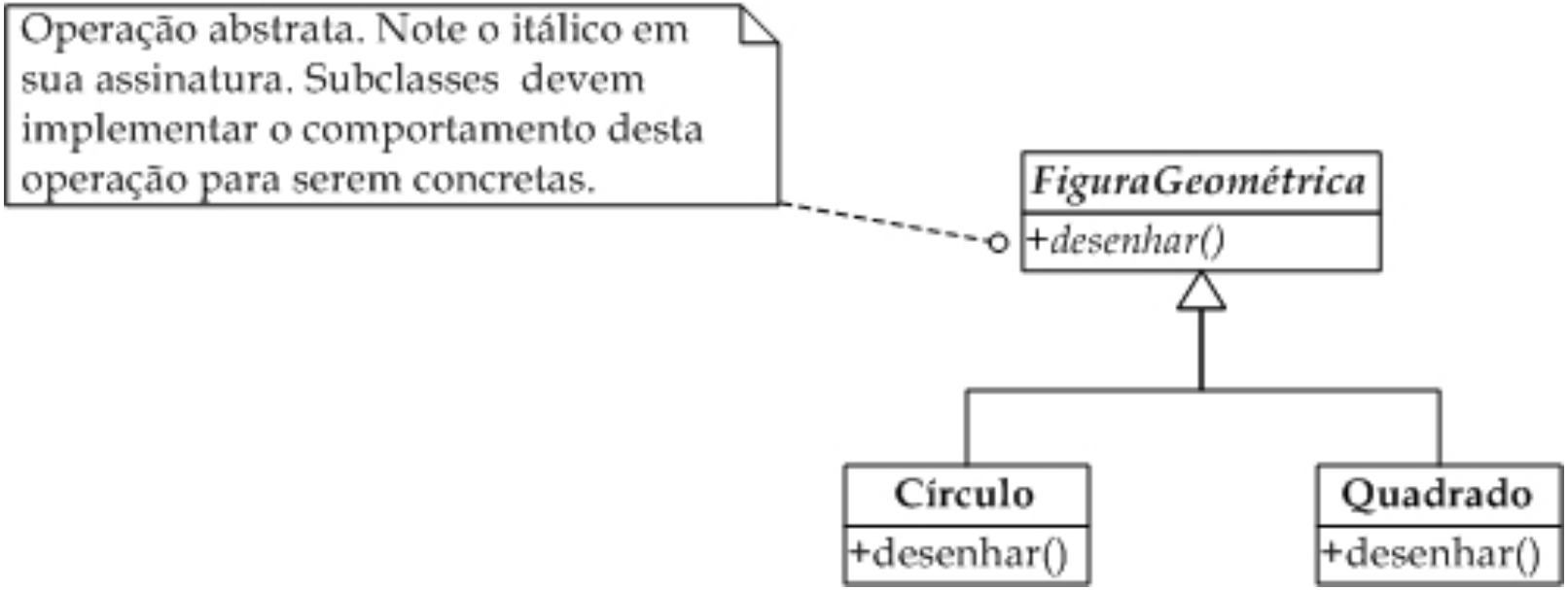
- Uma classe qualquer pode possuir tanto operações abstratas, quanto operações concretas (ou seja, operações que possuem implementação).
- Entretanto, uma classe que possui pelo menos uma operação abstrata é, por definição abstrata, abstrata.

Uma operação abstrata definida com visibilidade pública em uma classe também é herdada por suas subclasses.

Quando uma subclasse herda uma operação abstrata e não fornece uma implementação para a mesma, esta classe também é abstrata.

Operações abstratas (cont)

Na UML, a assinatura de uma operação abstrata é definida em itálico.



Operações polimórficas

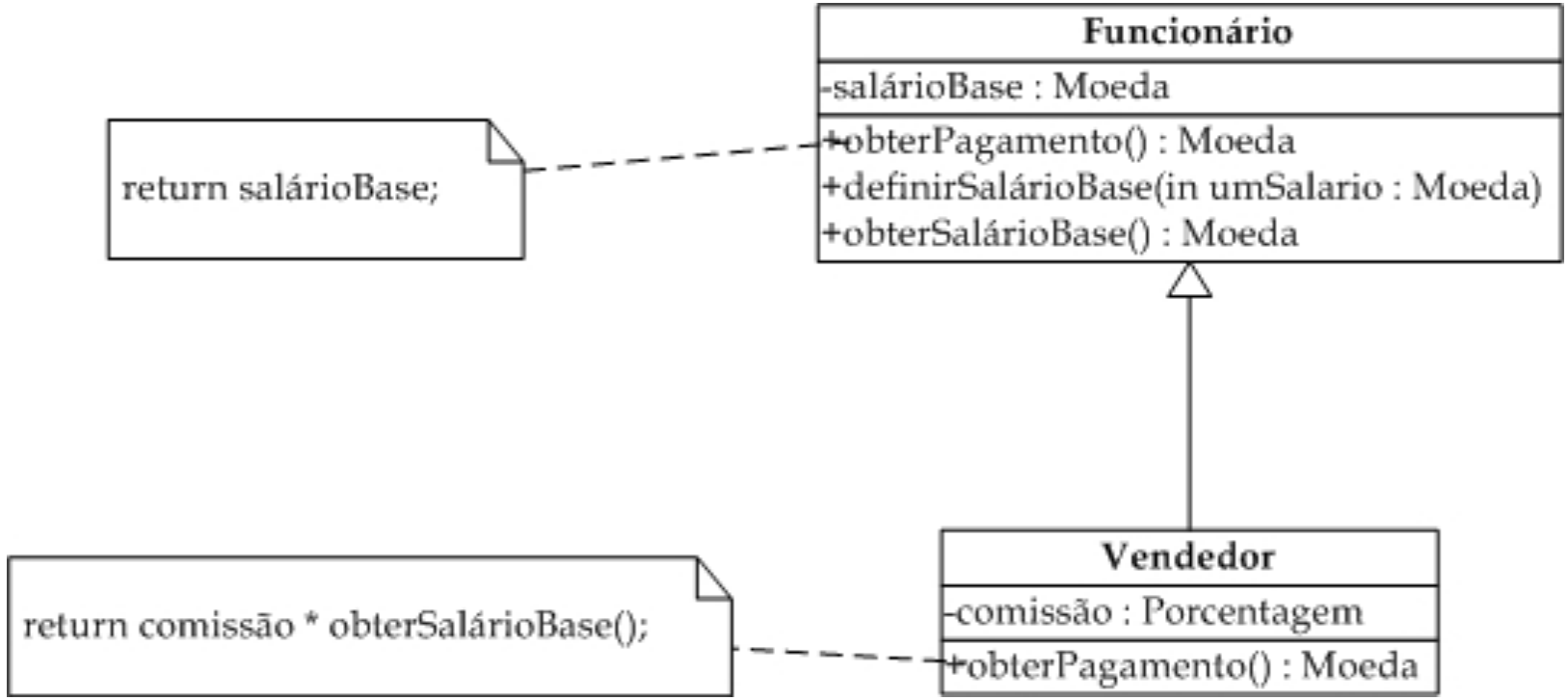
- Uma subclasse herda todas as propriedades de sua superclasse que tenham visibilidade pública ou protegida.
- Entretanto, pode ser que o comportamento de alguma operação herdada seja diferente para a subclasse.
- Nesse caso, a subclasse deve redefinir o comportamento da operação.
  - A assinatura da operação é reutilizada.
  - Mas, a implementação da operação (ou seja, seu ***método***) é diferente.
- Operações polimórficas são aquelas que possuem mais de uma implementação.

Operações polimórficas (cont)

- Operações polimórficas possuem sua assinatura definida em diversos níveis de uma hierarquia gen/spec.
  - A assinatura é repetida na(s) subclasse(s) para enfatizar a redefinição de implementação.
  - O objetivo de manter a assinatura é garantir que as subclasses tenham uma interface em comum.
- Operações polimórficas facilitam a implementação.
  - Se duas ou mais subclasses implementam uma operação polimórfica, a mensagem para ativar essa operação é a mesma para todas essas classes.
  - No envio da mensagem, o remetente não precisa saber qual a verdadeira classe de cada objeto, pois eles aceitam a mesma mensagem.
  - A diferença é que os *métodos* da operação são diferentes em

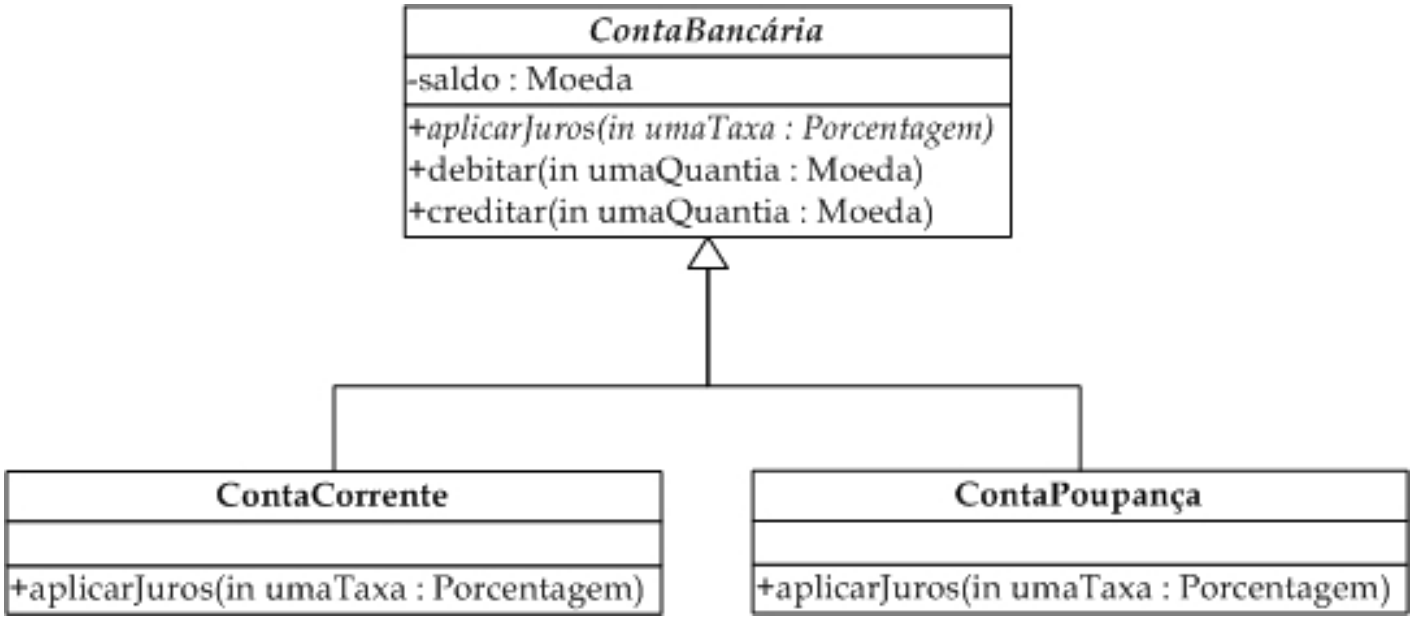
Operações polimórficas (cont)

A operação obterPagamento é polimórfica.



Operações polimórficas (cont)

Operações polimórficas também podem existir em classes abstratas.



Interfaces

- Uma ***interface*** entre dois objetos compreende um conjunto de **assinaturas de operações** correspondentes aos serviços dos quais a classe do objeto cliente faz uso.
- Uma interface pode ser interpretada como um ***contrato de comportamento*** entre um objeto cliente e eventuais objetos fornecedores de um determinado serviço.
  - Contanto que um objeto fornecedor forneça implementação para a interface que o objeto cliente espera, este último não precisa conhecer a verdadeira classe do primeiro.

Interfaces (cont.)

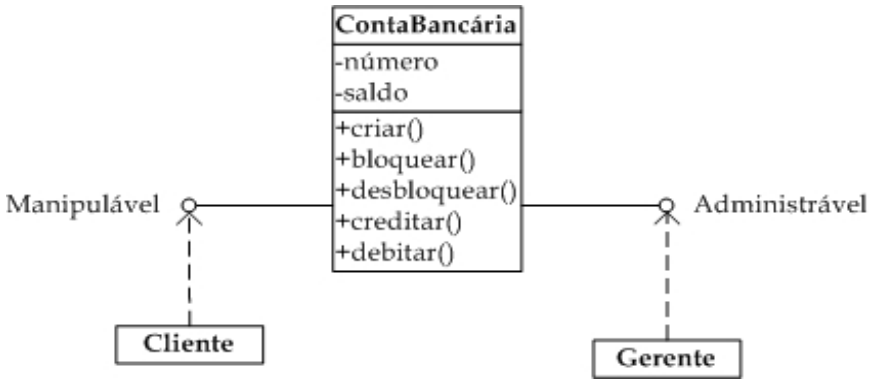
- Interfaces são utilizadas com os seguintes objetivos:
  - 1. Capturar semelhanças entre classes não relacionadas sem forçar relacionamentos entre elas.
  - 2. Declarar operações que uma ou mais classes devem implementar.
  - 3. Revelar as operações de um objeto, sem revelar a sua classe.
  - 4. Facilitar o desacoplamento entre elementos de um sistema.
- Nas LPOO modernas (Java, C#, etc.), interfaces são definidas de forma semelhante a classes.
  - Uma diferença é que todas as declarações em uma interface têm visibilidade pública.
  - Adicionalmente, uma interface não possui atributos, somente declarações de assinaturas de operações e (raramente) constantes.

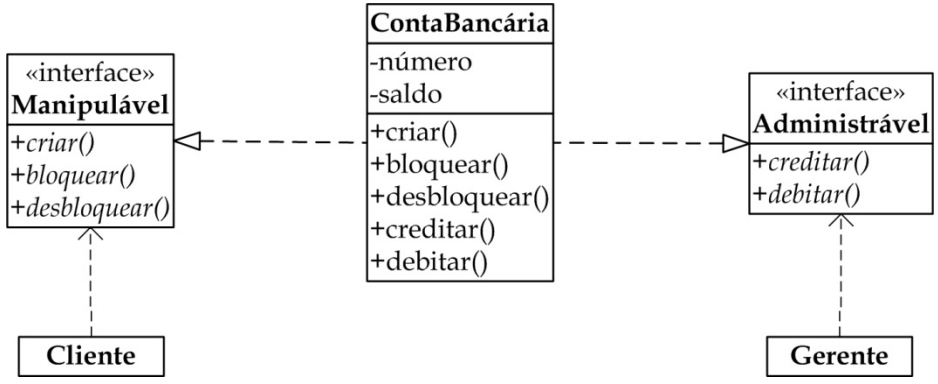
Interfaces (cont)

Notações para representar interfaces na UML:

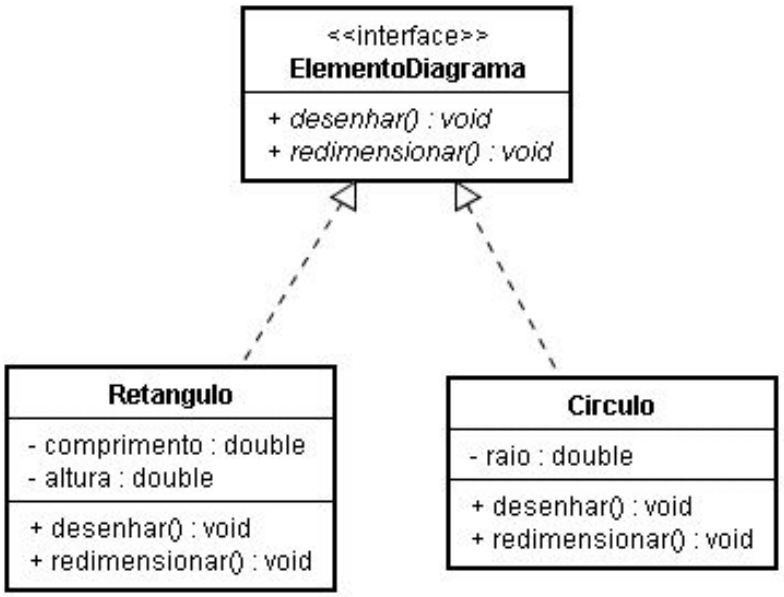
- A primeira notação é a mesma para classes. São exibidas as operações que a interface especifica. Deve ser usado o estereótipo <<interface>>.
- A segunda notação usa um segmento de reta com um pequeno círculo em um dos extremos e ligado ao classificador.

  Classes clientes são conectadas à interface através de um relacionamento de notação similar à do relacionamento de dependência.





Interface (cont)

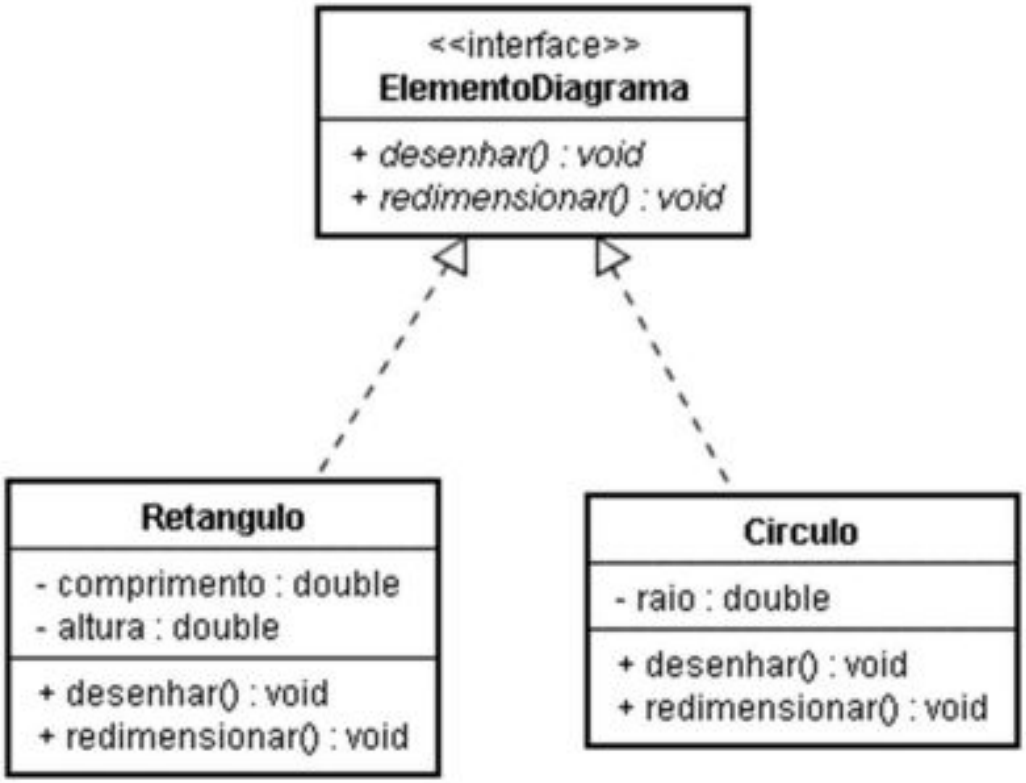



```
public interface ElementoDiagrama {
        double PI = 3.1425926; //static and final constant.
        void desenhar();
        void redimensionar();
}

public class Circulo implements ElementoDiagrama {

        …
        public void desenhar() {  /* draw a circle*/ }
        public void redimensionar() { /* draw a circle*/ }
}

public class Retangulo implements ElementoDiagrama {

        …
        public void desenhar() { /* draw a circle*/  }
        public void redimensionar() { /* draw a circle*/ }
```

Interface

- É um contrato sem implementação entre dois ou mais objetos. A interface serve como balizador para determinar quais métodos um objeto pode esperar do outro.
- Na interface não há atributos apenas assinaturas dos métodos.
- A interface é utilizada para reduzir o acoplamento, facilitando o reuso de classes.
- Todos os métodos na interface são abstract e public por padrão, colocar estes modificadores no código é uma redundância.

Interface


<<interface>>
ElementoDiagrama
+ desenhar() : void
+ redimensionar() : void
Retangulo
- comprimento : double
- altura : double
+ desenhar() : void
+ redimensionar() : void
Circulo
- raio : double
+ desenhar() : void
+ redimensionar() : void

Interface

- A interface deve ser criada utilizando-se:

```
<modificador> interface <nome da interface> {
        <tipo de retorno 1> <nome metodo 1> (<parametros>);
        <tipo de retorno 2> <nome metodo 2> (<parametros>);
        <tipo de retorno n> <nome metodo n> (<parametros>);
        ....
}
```

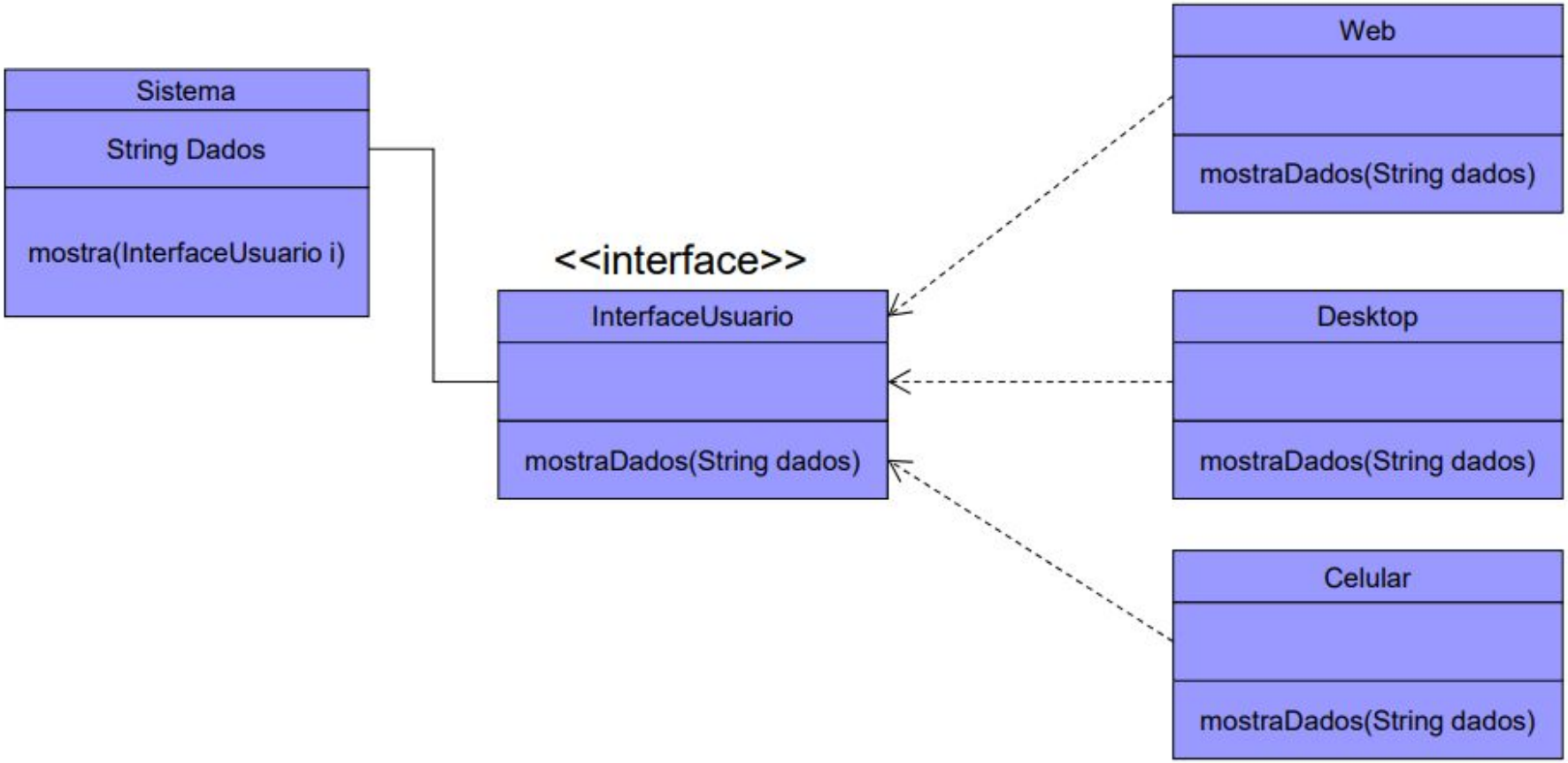
- Exemplo:

```
public interface Funcionario {
        public void recebeSalario(float valor);
        public void bateCartaoEntrada(Date horaAtual);
        public void bateCartaoSaida(Date horaAtual);
}
```

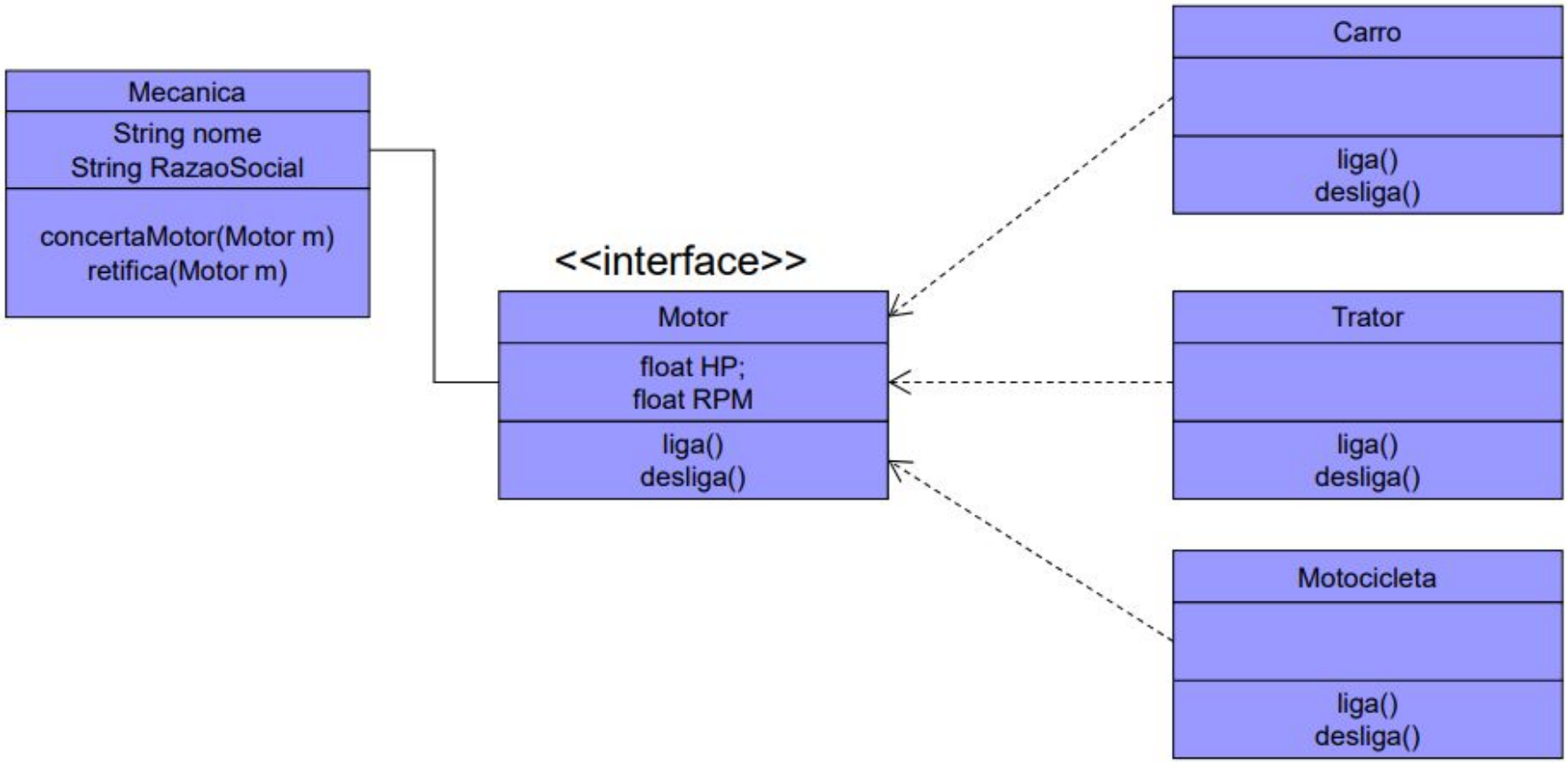
Interface

- Uma classe pode implementar uma ou mais interfaces, quando isso ocorre a classe deve implementar ou seja colocar código em todos os métodos recebidos da(s) interface(s).
- Se houver mais de uma interface implementada, então os nomes devem ser separados por virgula (,) Ao implementar mais de uma interface a classe precisa sobrescrever códigos em todos os métodos recebidos das interfaces implementadas

Interface – Exemplo de desacoplamento de arquitetura



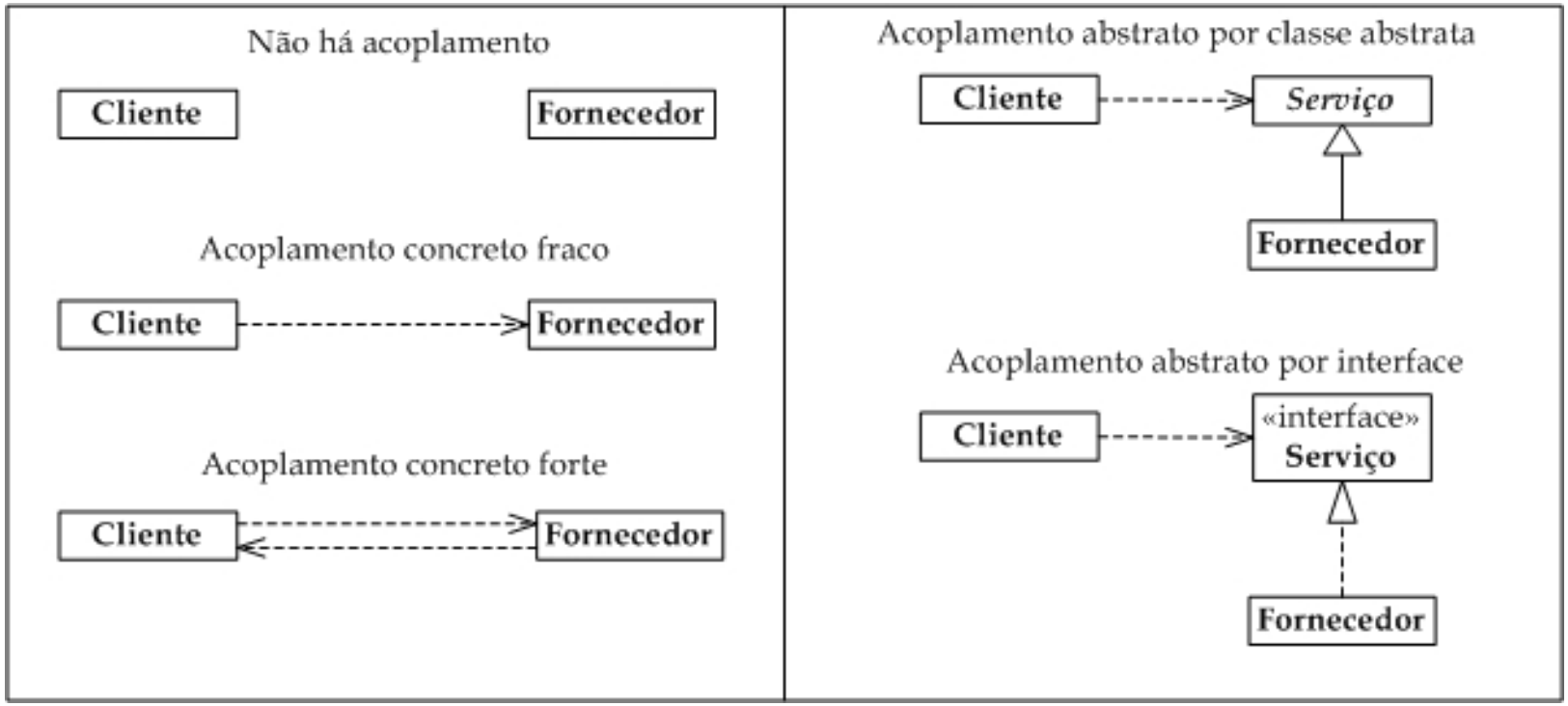
Interface – Exemplo de desacoplamento da classe Motor



Acoplamentos concreto e abstrato

- Usualmente, um objeto A faz referência a outro B através do conhecimento da classe de B.
  - Esse tipo de dependência corresponde ao que chamamos de ***acoplamento concreto***.
- Entretanto, há outra forma de dependência que permite que um objeto remetente envie uma mensagem para um receptor sem ter conhecimento da verdadeira classe desse último.
  - Essa forma de dependência corresponde ao que chamamos de ***acoplamento abstrato***.
  - A acoplamento abstrato é preferível ao acoplamento concreto.
- Classes abstratas e interface permitem implementar o acoplamento abstrato.

Acoplamentos concreto e abstrato (cont)



Padrões de projeto

- É da natureza do desenvolvimento de software o fato de que os mesmos problemas tendem a acontecer diversas vezes.
- Um ***padrão de projeto*** corresponde a um esboço de uma solução reusável para um problema comumente encontrado em um contexto particular.
- Estudar esses padrões é uma maneira efetiva de aprender com a experiência de outros.
- O texto clássico sobre o assunto é o de Erich Gamma et al.
  - Esses autores são conhecidos *Gang of Four*.
  - Nesse livro, os autores catalogaram 23 padrões.

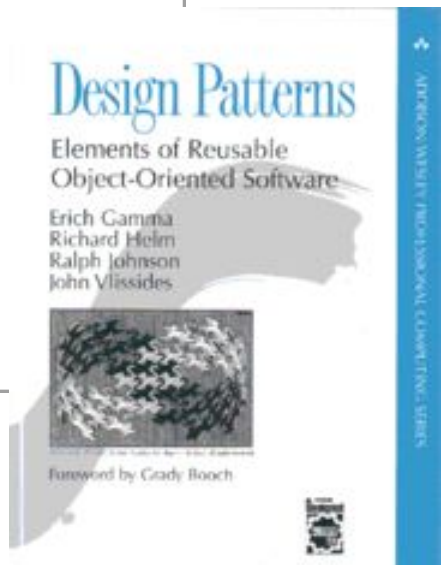
Padrões GoF

- Os padrões GoF foram divididos em três categorias:

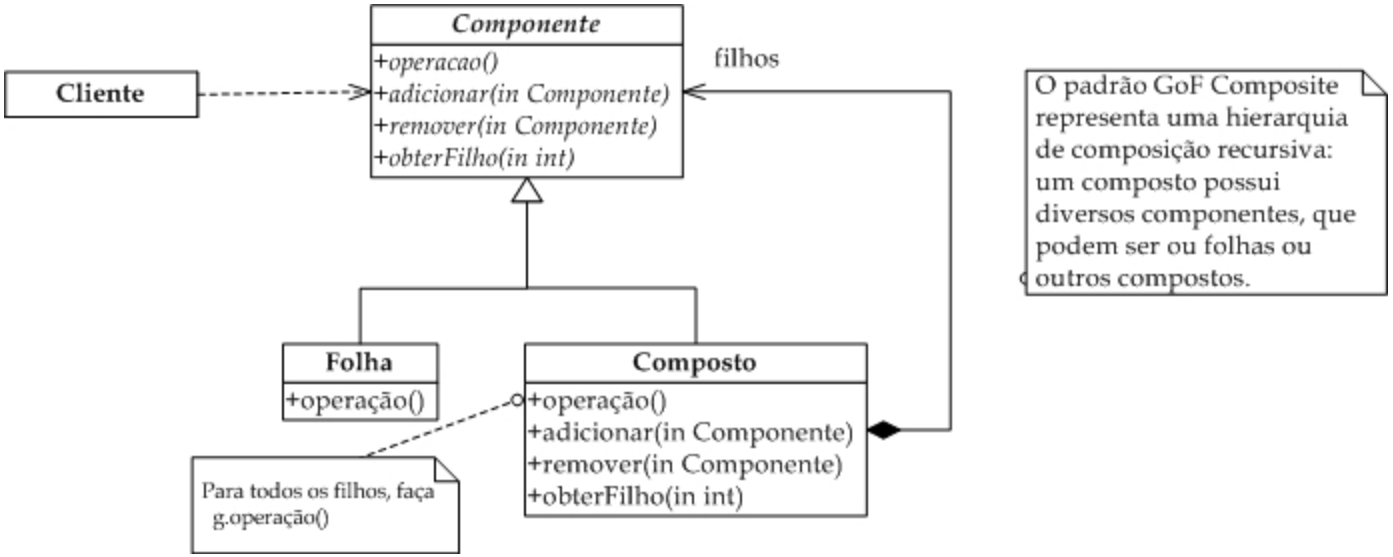
1. ***Criacionais***: procuram separar a operação de uma aplicação de como os seus objetos são criados.
2. ***Estruturais***: provêem generalidade para que a estrutura da solução possa ser estendida no futuro.
3. ***Comportamentais***: utilizam herança para distribuir o comportamento entre subclasses, ou agregação e composição para construir comportamento complexo a partir de componentes mais simples.

Padrões GoF

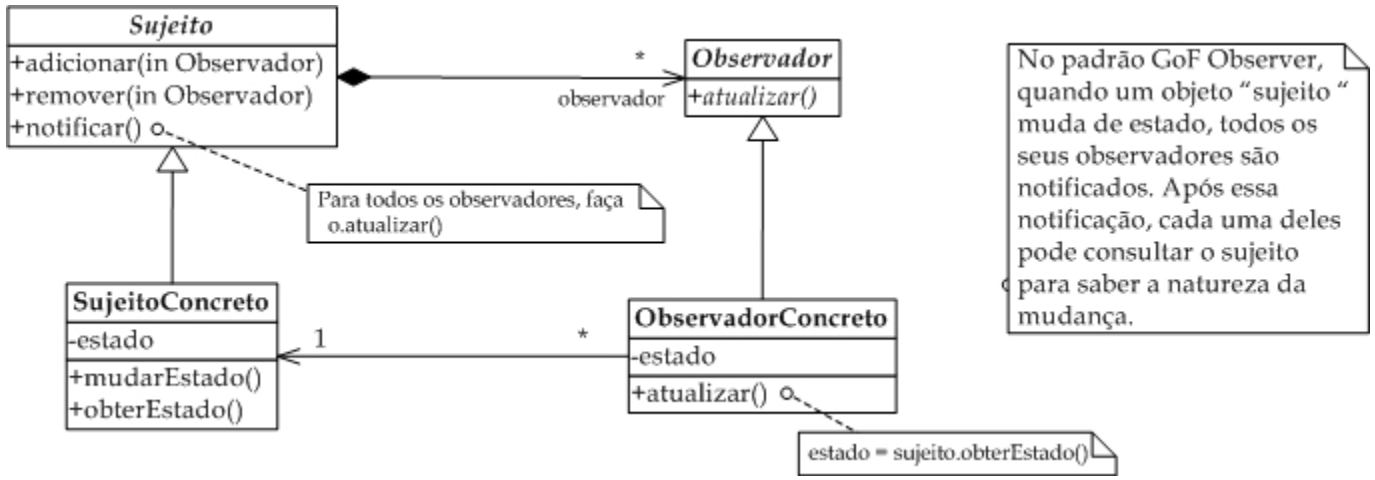
Criacionais	Estruturais	Comportamentais
Abstract Factory Builder **Factory Method** Prototype Singleton	Adapter Bridge **Composite** Decorator **Façade** Flyweight Proxy	Chain of Responsibility Command Interpreter Iterator **Mediator** Memento **Observer** State **Strategy** Template Method Visitor



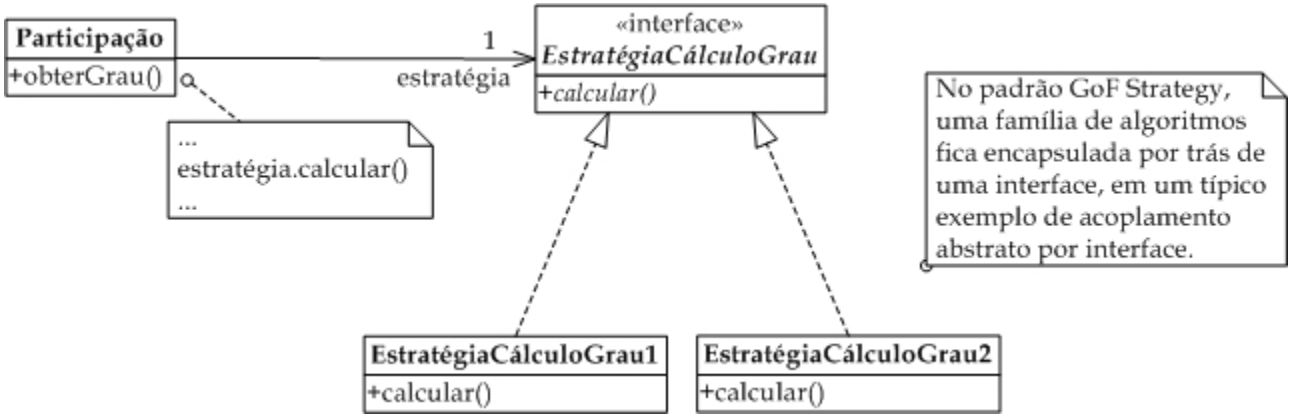
Composite


Componente
+operacao()
+adicionar(in Componente)
+remover(in Componente)
+obterFilho(in int)
filhos
Cliente
Folha
+operação()
Composto
+operação()
+adicionar(in Componente)
+remover(in Componente)
+obterFilho(in int)
Para todos os filhos, faça
g.operação()
O padrão GoF Composite representa uma hierarquia de composição recursiva: um composto possui diversos componentes, que podem ser ou folhas ou outros compostos.

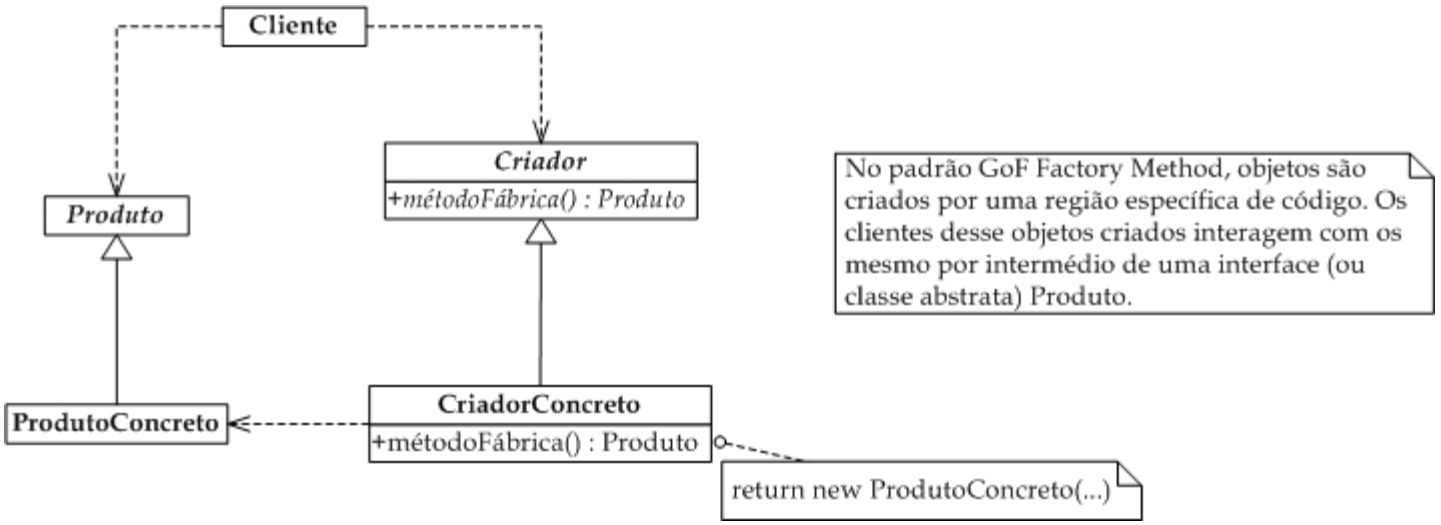
Observer


Sujeito
+adicionar(in Observador)
+remover(in Observador)
+notificar()
*
observador
Observador
+atualizar()
Para todos os observadores, faça
o.atualizar()
No padrão GoF Observer, quando um objeto "sujeito" muda de estado, todos os seus observadores são notificados. Após essa notificação, cada uma deles pode consultar o sujeito para saber a natureza da mudança.
SujeitoConcreto
-estado
+mudarEstado()
+obterEstado()
1
*
ObservadorConcreto
-estado
+atualizar()
estado = sujeito.obterEstado()

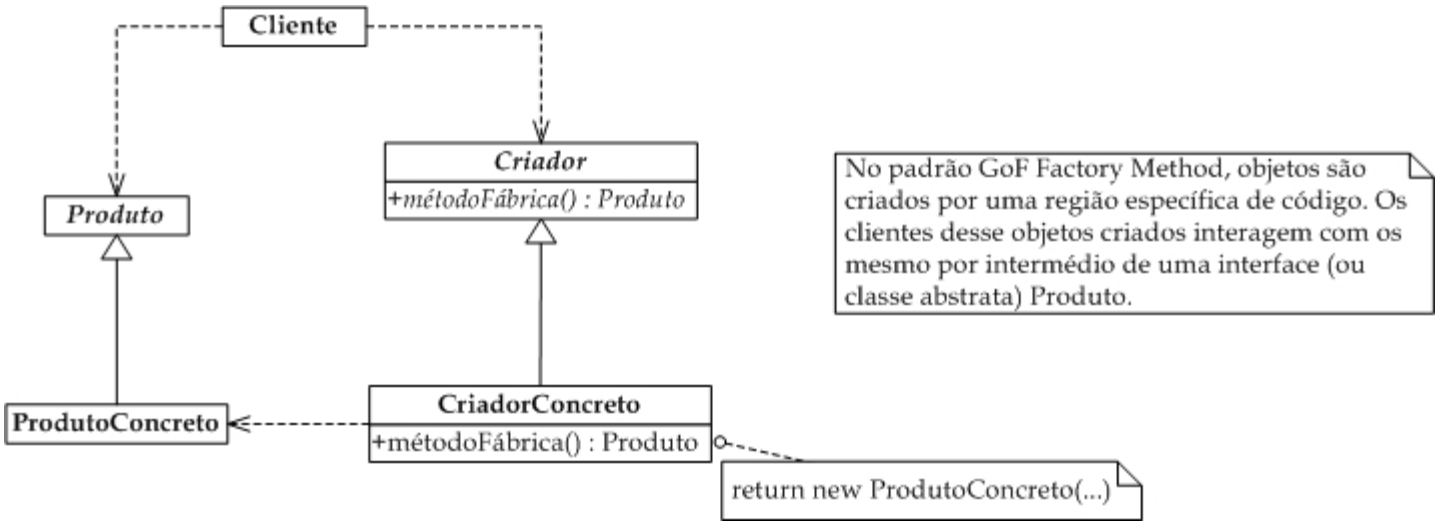
Strategy


Participação
+obterGrau()
1
estratégia
«interface»
EstratégiaCálculoGrau
+calcular()
...
estratégia.calcular()
...
No padrão GoF Strategy, uma família de algoritmos fica encapsulada por trás de uma interface, em um típico exemplo de acoplamento abstrato por interface.
EstratégiaCálculoGrau1
+calcular()
EstratégiaCálculoGrau2
+calcular()

Factory Method



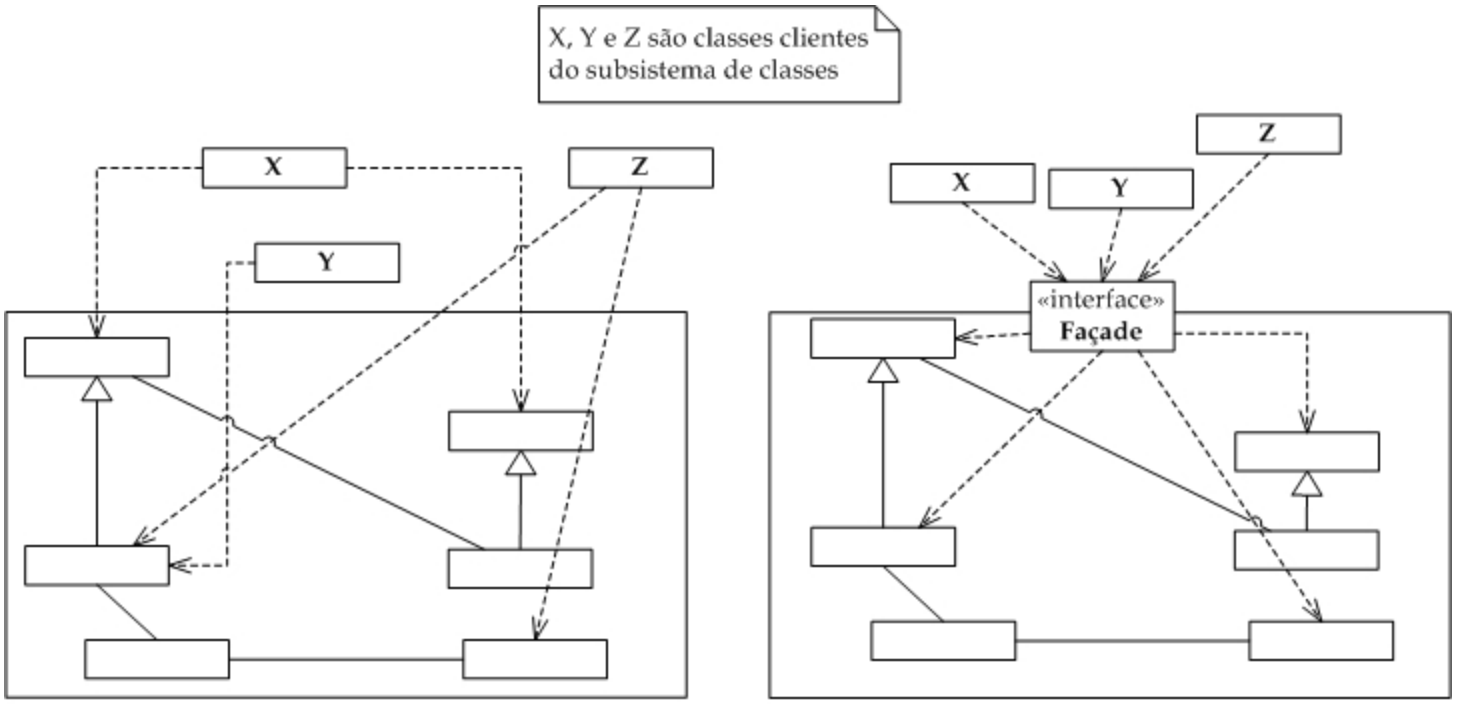
Factory Method



Mediator

- O padrão ***Mediator*** permite a um grupo de objetos interagirem, ao mesmo tempo em que mantém um acoplamento fraco entre os componentes desse grupo.
- A solução proposta pelo Mediator é definir um objeto, o *mediador*, para encapsular interações da seguinte forma: o resultado da interação de um subgrupo de objeto é passado a outro subgrupo pelo mediador.
- Dessa forma, os subgrupos não precisam ter conhecimento da existência um do outro e podem variar independentemente.
- *Objetos de controle* são exemplos de mediadores.

Façade



Solução sem o uso de um objeto Façade

Solução com o uso de um objeto Façade

Classes

- Para criar uma classe basta utilizar a palavra reservada class
- O construtor padrão é definido na mesma linha da classe

```
class Aluno(nome: String, idade: Int) {
val nome: String
val idade: Int
init {
// bloco de inicialização
this.nome = nome
this.idade = idade
}
override fun toString(): String {
return "Aluno: $nome, idade: $idade"
}
}
fun main(args: Array<String>) {
val a1 = Aluno(“Fabio", 28)
println(a1)
println("Aluno: ${a1.nome}, idade: ${a1.idade}")
}
```

Classes

Declarando uma classe no Kotlin

```
class Person(val name: String, private val birthYear: Int) {  init {
    println("Initializing Person with name $name")
  }

  constructor(name: String, birthYear: Int, isMarried: Boolean) : this(name, birthYear) {
    this.isMarried = isMarried
  }

  // Definindo uma propriedade com o get customizado
  val age: Int
    get() {
      return LocalDateTime.now().year - birthYear
    }

  var isMarried: Boolean = false
    private set // Atribuição apenas dentro da classe
}
```

Construtor Primário com Propriedades

Bloco de inicialização

Construtor Secundário sem propriedades

Propriedade com get customizado

Propriedade com set privado

Classes

Criando uma instância

```
val person1 = Person("Felipe", 1987)

val person2 = Person("Bruna", 1989, false)
```

Exercícios

- 1) Crie uma classe chamada Turma com os atributos ID Turma, nome da turma e quantidade de alunos. Passe 3 argumentos como parâmetros dessa função e exiba cada um dos atributos em console.
- 2) Cria uma classe chamada Aluno com os atributos ID, nome e notas. Passe 10 notas por meio da utilização de um ArrayList e imprima cada um dos valores em console.

Classes

- Não é preciso utilizar new para criar uma instância da classe
- O método toString foi sobrescrito para ser chamado sempre que a classe for convertida em String
- Mais: https://kotlinlang.org/docs/reference/classes.html

Herança

- Para utilizar herança basta utilizar : e o nome da classe mãe

```
open class Pessoa(nome: String, idade: Int) {
val nome: String
val idade: Int
init {// bloco de inicialização
this.nome = nome
this.idade = idade
}
open fun adicionarDesconto(desconto: Int) {
println("Desconto para a pessoa de $desconto")
}
override fun toString(): String {
return "Pessoa: $nome, idade: $idade"
}
}
class Aluno(nome: String, idade: Int): Pessoa(nome, idade) {
override fun adicionarDesconto(desconto: Int) {
println("Desconto para o aluno de $desconto")
}
}
fun main(args: Array<String>) {
val a1 = Aluno("Fabio", 28)
println(a1)
println("Aluno: ${a1.nome}, idade: ${a1.idade}")
a1.adicionarDesconto(10)
}
```

Herança

- Para que uma classe possa ser herdada ela deve ser marcada como open
  - Todas as classes são final por padrão
  - A mesma regra vale para métodos

- Mais: https://kotlinlang.org/docs/reference/classes.html

Exercícios

- 3) Inicialize a classe Carro contendo os atributos modelo do tipo de dado String, ano do tipo de dado inteiro e velocidade do tipo de dado inteiro. Na classe Carro, o atributo ano deve ser iniciado com o valor 1900. Na função main crie 3 instâncias da classe Carro, nomeadas como c0, c1 e c2. Inicialize o atributo ano de c2 com o valor 2020. Em seguida exiba o ano do carro acessado por meio do objeto c1 e o ano do carro acessado por meio do objeto c2.
- 4) Com base na resolução do exercício 3, inicialize todas as propriedades do carro, modelo, ano, velocidade para os objetos c0, c1 e c2 e exiba cada uma dessas propriedades iniciadas.
- 5) Com base no cenário de resolução do exercício 4, na função main, verifique se o ano do Carro de algum dos objetos é menor do que 2000. Se sim, exiba uma mensagem indicando que o motorista necessita realizar a venda do veículo.
- 6) Mude todas as propriedades, como no exemplo “private var modelo” e tente executar o exercício 5. Caso haja erro, solucione-o mudando as propriedades para public.

Data Classes

- Data Classes são classes que contém somente informações
- Em Kotlin uma Data Class é criada apenas com uma linha

```
data class Aluno(val nome: String)
```

- Essa linha cria a classe Aluno com:
  - Atributos do construtor
  - Getters e setters
  - Métodos equals, toString e copy

- Mais: https://kotlinlang.org/docs/reference/data-classes.html

Listas

- Lista ordenada: objeto ArrayList
- Lista mutável: função mutableListOf<T>(elementos) recomendado

```
fun main(args: Array<String>) {
// lista com ArrayList
var arrayListInt = ArrayList<Int>()
arrayListInt.add(1)
arrayListInt.add(2)
arrayListInt.add(3)
println(arrayListInt)
// mesma lista utilizando mutableListOf
var mutableListInt = mutableListOf<Int>(1,2,3)
println(mutableListInt)
// adicionar
mutableListInt.add(4)
println(mutableListInt)
}
```

- Mais:https://kotlinlang.org/docs/reference/collections.html

Enum

- Objeto com constantes predefinidas

```
//enum simples

enum class Status {ATIVO, INATIVO}

// enum com construtor

enum class Conceitos(val nota: String) {

APROVADO("A"),

RECUPERACAO("B"),

REPROVADO("C")

}

fun main(args: Array<String>) {

val status = Status.ATIVO

println(status)

val conceito = Conceitos.APROVADO.nota

println(conceito)

}
```

- Mais: https://kotlinlang.org/docs/reference/enum- classes.html

Higher-Order Functions e Lambdas

- Traz produtividade na escrita dos códigos

- Kotlin: paradigma funcional
  - Uma função pode receber como parâmetro e retorne outra função
  - Higher-order Functions

```
fun filtrar(list: List<Int>, filtro: (Int) -> (Boolean)): List<Int> {
val newList = arrayListOf<Int>()
for (item in list) {
if (filtro(item)) {
newList.add(item)
}
}
return newList
}
fun numerosPares(numero: Int) = numero % 2 == 0
fun numerosMaioresQue3(numero: Int) = numero > 3
fun main(args: Array<String>) {
var ints = listOf(1,2,3,4,5)
println(ints)
var list = filtrar(ints, ::numerosPares)
println(list)
list = filtrar(ints, ::numerosMaioresQue3)
println(list)
}
```

Higher-Order Functions e Lambdas

- filtro: (Int) -> (Boolean) Informa que a função enviada como parâmetro para filtro recebe um inteiro e retorna um booleano
- Para enviar a função utiliza-se a sintaxe ::funcao

Higher-Order Functions e Lambdas

- Lambda é uma sintaxe simplificada para enviar uma função como parâmetro
- A passagem e feita utilizando chaves

```
fun main(args: Array<String>) {
var ints = listOf(1,2,3,4,5)
println(ints)
// forma 1
var list = filtrar(ints, {numero:Int
      ->
      numerosPares(numero)})
println(list)
// forma 2
list = filtrar(ints,
      {numerosMaioresQue3(it)})
println(list)
// forma 3
list = filtrar(ints)
      {numerosMaioresQue3(it)}
println(list)
}
```

Funções anônimas

- Com lambda é possível enviar uma função sem nem mesmo cria-la

  – Funções anônimas

```
fun filtrar(list: List<Int>, filtro: (Int) -> (Boolean)): List<Int> {
val newList = arrayListOf<Int>()
for (item in list) {
if (filtro(item)) {
newList.add(item)
}
}
return newList
}
fun main(args: Array<String>) {
var ints = listOf(1,2,3,4,5)
println(ints)
var list = filtrar(ints) { it % 2 == 0}
println(list)
list = filtrar(ints) {it > 3}
println(list)
}
```

- Mais: https://kotlinlang.org/docs/reference/lambdas.html

Referências

- Princípios de Análise de Projetos de Sistemas com UML 2ª Edição
- https://developer.android.com/training/basics/firstapp?hl=pt-br
- https://www.sncticet.ufam.edu.br/2017/downloads/christianreis.pdf
- https://homepages.dcc.ufmg.br/~fernando/classes/android/slides/Class1.pdf
- https://pt.slideshare.net/AnaDoloresLimaDias/android-9149956
- https://developer.android.com/guide/components/activities/activity-lifecycle?hl=pt-br
- https://www.tutorialspoint.com/android/android_hello_world_example.htm