

Programação orientada a objetos

Interfaces (realização)

Relacionamentos entre classes

- Classes podem se relacionar entre si, definindo um vínculo entre os objetos dessas classes.




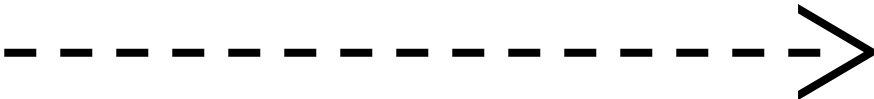


Exemplos

- Um **cliente** possui um **endereço**.
- Uma **empresa** é composta por **funcionários**.
- Uma **moto** é um tipo de **veículo**.
- Um **restaurante** possui **pratos**.
- Uma **correspondência** possui um **remetente** e um **destinatário**.

Relacionamentos entre classes

- **Associação:** conexão entre classes.
- **Agregação e composição:** especialização de uma associação onde um todo é relacionado com suas partes (relacionamento “parte-de”).
- **Dependência:** um objeto depende de alguma forma de outro (relacionamento de utilização).
- **Herança (generalização):** um dos princípios da orientação a objetos, permite a reutilização, uma nova classe pode ser definida a partir de outra já existente.
- **Realização:** um contrato que a classe segue (obrigação).

Relacionamentos entre classes

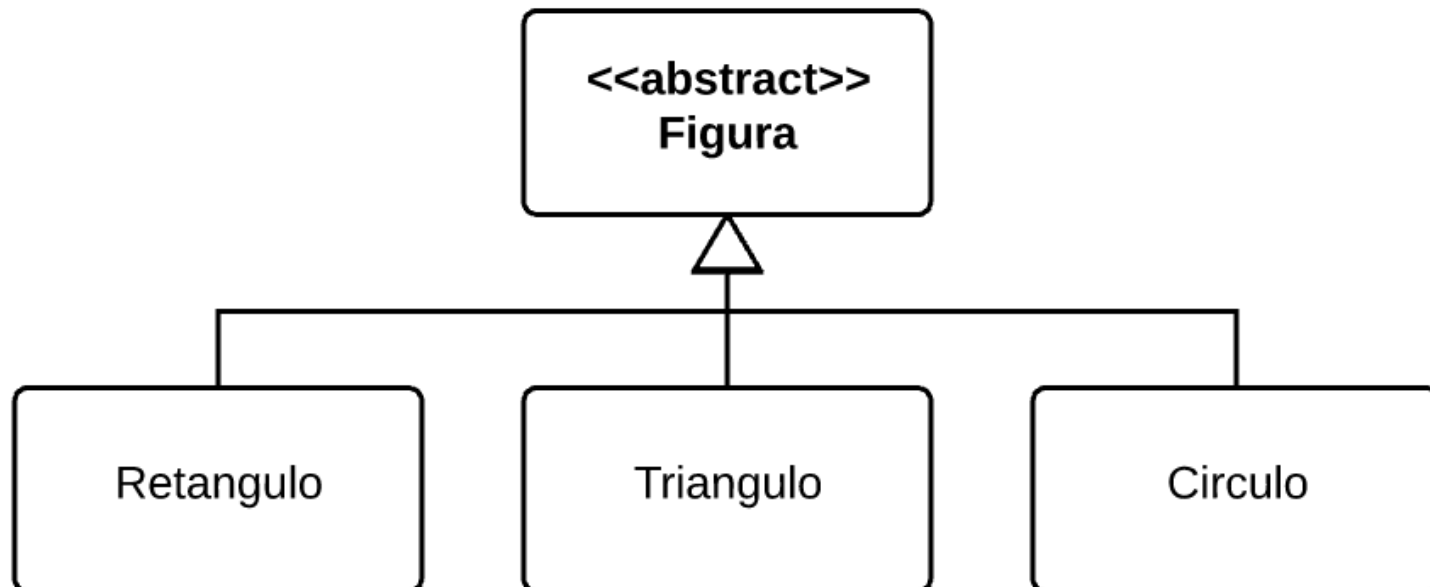
- Associação: 
- Agregação: 
- Composição: 
- Dependência: 
- Herança (generalização): 
- Realização: 

Interfaces

- Uma interface define um contrato ao qual uma classe pode assinar.
- Este contrato estabelece todos os métodos que esta classe deverá implementar e fornecer aos seus clientes.
- Quando uma classe assina o contrato (implementa a interface) deve implementar todos os métodos definidos nele.

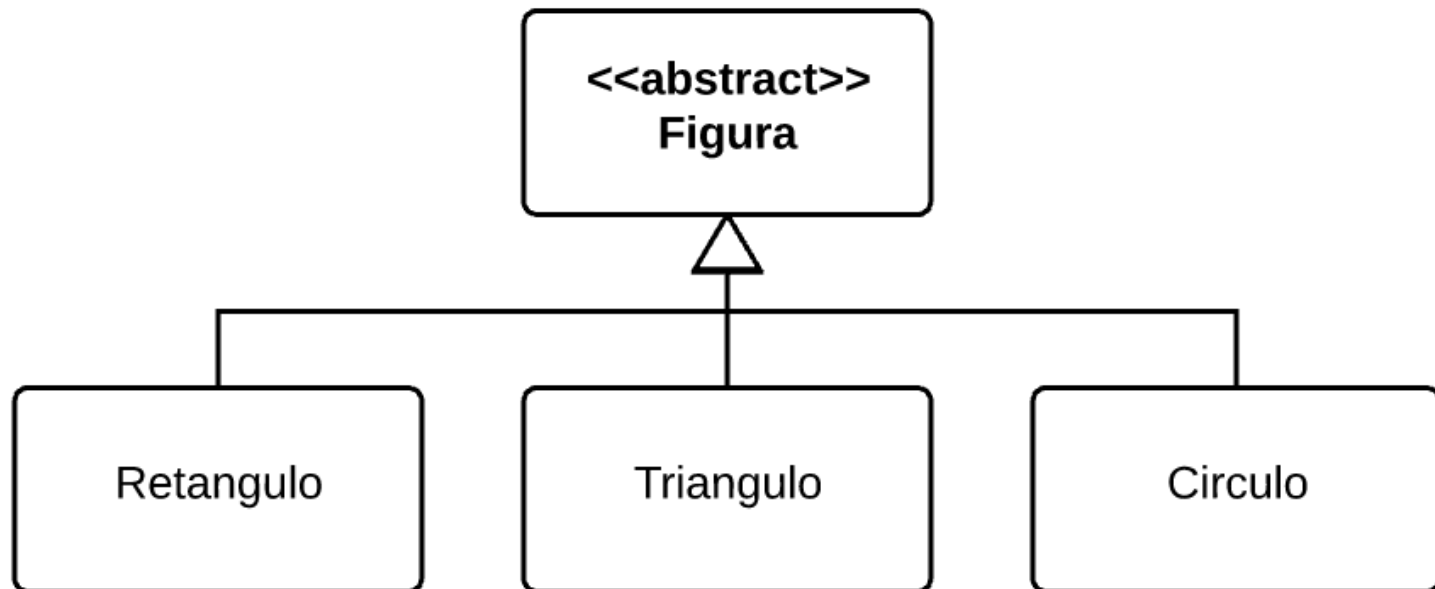
Interfaces

- **Exemplo:** a classe abstrata **Figura** define o atributo **cor** às suas subclasses, bem como os métodos abstratos **area()** e **perimetro()**.



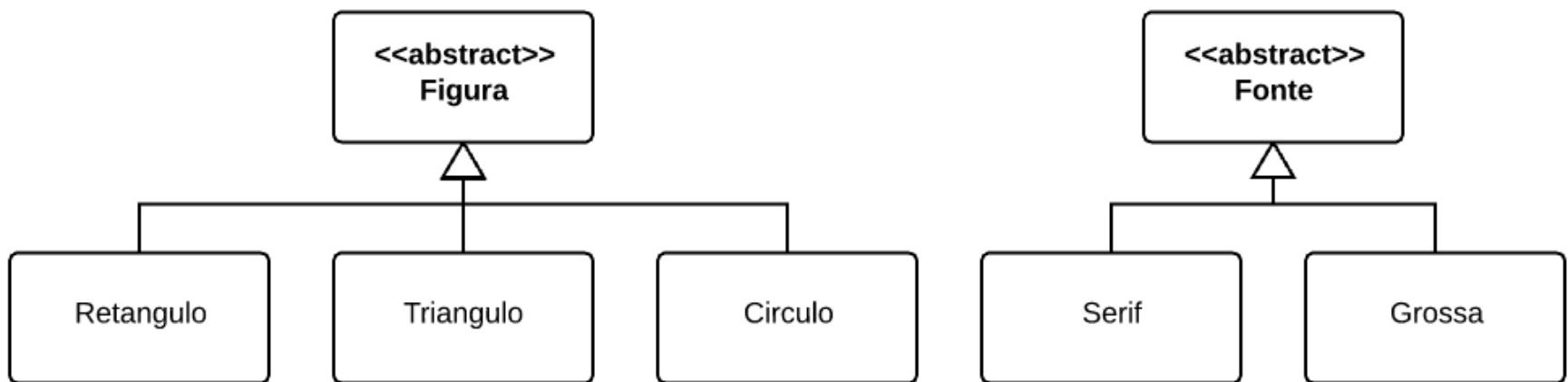
Interfaces

- **Exemplo:** a classe abstrata **Figura** define o atributo **cor** às suas subclasses, bem como os métodos abstratos **area()** e **perimetro()**.
- Se for necessário que cada figura implemente seu próprio método **desenhar()**, uma boa estratégia consiste em definir o método abstrato **desenhar** na classe **Figura**, garantindo que cada subclasse forneça sua implementação, aproveitando-se do polimorfismo.



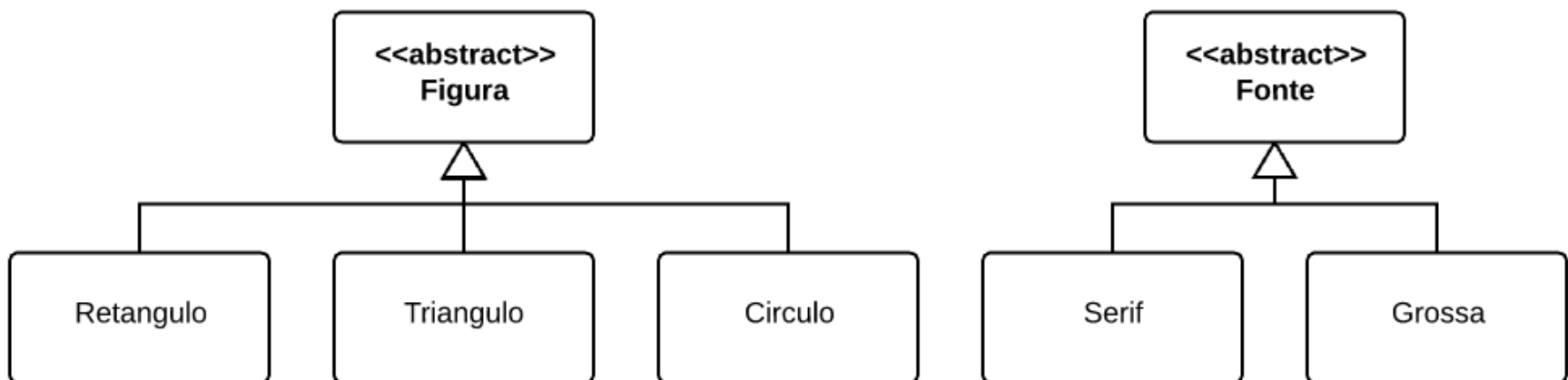
Interfaces

- Porém, o mesmo sistema contém as classes Fonte, Serif e Grossa, que definem as fontes de texto de um elemento gráfico. Estas classes também devem implementar seus métodos **desenhar()**.
- Logo, o método de desenho não é exclusivo das figuras, portanto não fazem parte da sua classe.
- Qual a forma adequada de estruturar o sistema? As fontes devem estender Figura? As figuras devem estender Fonte? As figuras e as fontes devem estender uma nova classe?



Interfaces

- A herança deve ser usada estritamente quando o relacionamento entre as classes responde a uma relação “é um”.
- Neste caso, uma figura **não é uma** fonte, e uma fonte **não é uma** figura.
- Para resolver esse problema, o ideal seria termos uma forma de apenas definir que as figuras e as fontes devem implementar o método **desenhar()**.
 - INTERFACES!



Interfaces

- Uma interface faz isso, define um contrato onde as classes que a realizam devem implementar os seus métodos.
- A interface **Desenhavel** pode ser definida da seguinte forma:

```
public interface Desenhavel {  
    void desenhar();  
}
```

- Toda a classe que realizar (ou implementar) esta interface, deverá apresentar seu código para o método **desenhar()**.

Interfaces

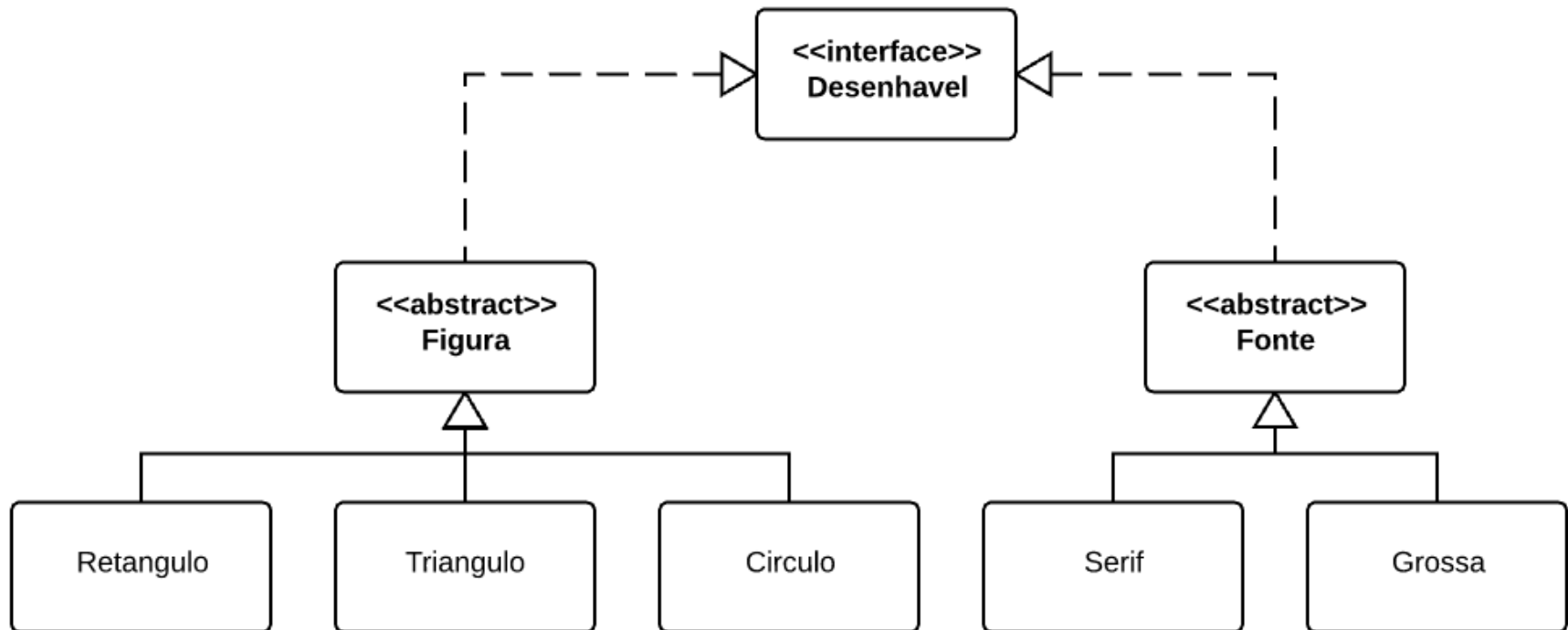
- Basta então fazer com que as classes Figura e Fonte realizem (ou implementem) a interface criada, fazendo com que suas subclasses tenham que implementar o método de desenho.

```
public abstract class Figura implements Desenhavel {  
    private String cor;  
  
    public abstract double area();  
    public double perimetro();  
  
    public void desenhar() {  
        //implementação aqui.  
    }  
}
```

```
public abstract class Fonte implements Desenhavel {  
  
    public void desenhar() {  
        //implementação aqui.  
    }  
}
```

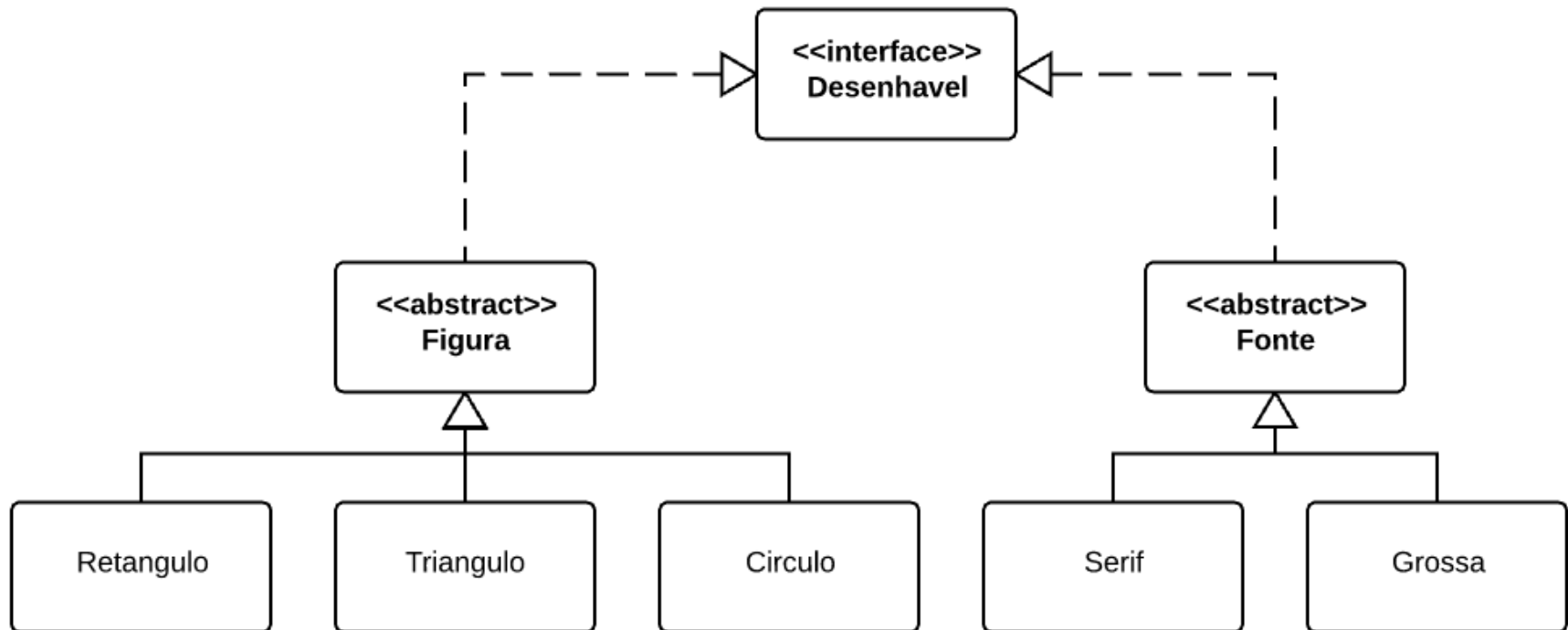
Interfaces

- Repare que, neste caso, as classes **Figura** e **Fonte** não são obrigadas implementar o método **desenhar()**, pois são abstratas (o que não impede sua implementação). No entanto, a obrigatoriedade de implementação é passada às suas subclasses concretas.



Interfaces

- Se a superclasse **Fonte** (por exemplo) implementar o método **desenhar()**, as classes **Serif** e **Grossa** não precisam fazê-lo, a não ser que queiram sobrescrever o método. Se a classe **Fonte** não implementar o método, isso deve ser feito em **Serif** e **Grossa**.



Interfaces

- Com as interfaces, podemos usufruir de uma capacidade ainda maior de polimorfismo. Se tivermos uma lista de figuras, podemos nos referir a elas como desenháveis e chamar os métodos definidos na interface (contrato) **Desenhavel**.

```
public void run() {  
    List<Figura> figuras = criaListaFiguras();  
  
    for(Figura f: figuras) {  
        f.desenhar();  
    }  
  
    for(Desenhavel d: figuras) {  
        d.desenhar();  
    }  
}
```

- Se futuramente novas classes forem incluídas no sistema como subclasses de **Figura**, a aplicação continuará funcionando normalmente, uma vez que a herança e a realização garantem a existência do método **desenhar()**.

Referências

CAELUM. **Apostila Java e Orientação a Objetos**. Curso FJ-11, 2016.

DEITEL, H. M. **Java: como programar**. H. M Deitel e P. J. Deitel - 8a ed. Porto Alegre: Prentice-Hall, 2010.

Leitura complementar

TutorialsPoint Java (<http://www.tutorialspoint.com/java>).