# Engenharia de Software

## Implementação

**Luís Morgado**

Instituto Superior de Engenharia de Lisboa
Departamento de Engenharia de Electrónica e Telecomunicações e de Computadores

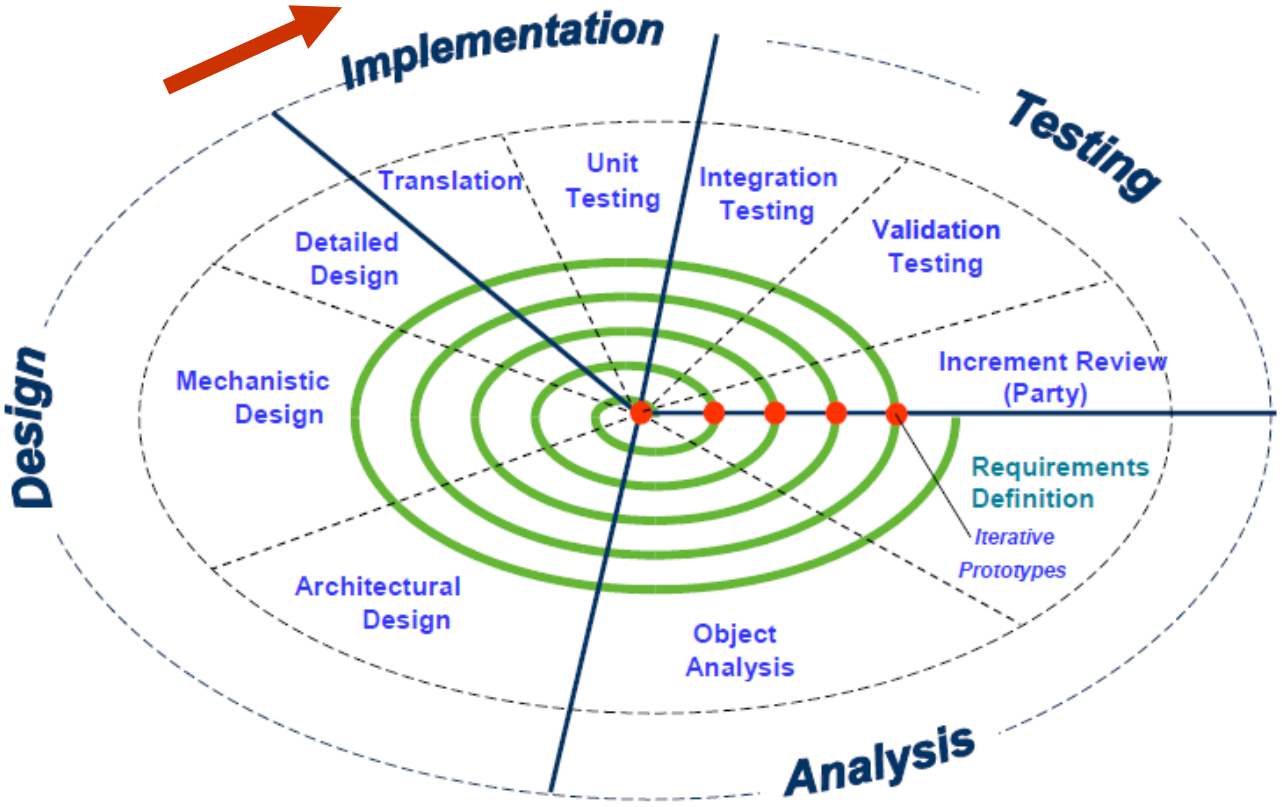
# Processo de Desenvolvimento

- **Análise**
- **Concepção**
- **Construção**
  - Transição projecto de arquitectura de software – implementação
  - Concretização do sistema por conversão da sua arquitectura em código fonte executável
- **Verificação**

# Transição Concepção - Construção

A transição entre a actividade de concepção (*design*) e a actividade de construção (implementação) requer um processo de conversão da arquitectura detalhada em código, o qual deve ser sistemático, consistindo essencialmente numa tradução de representações em linguagens distintas, nomeadamente de UML para uma ou mais linguagens de programação alvo

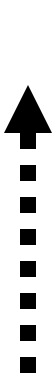
A conversão de UML para código fonte (*forward code engineering*) pode ser automática se a arquitectura do sistema tiver a informação necessária para especificar de forma precisa o código a gerar

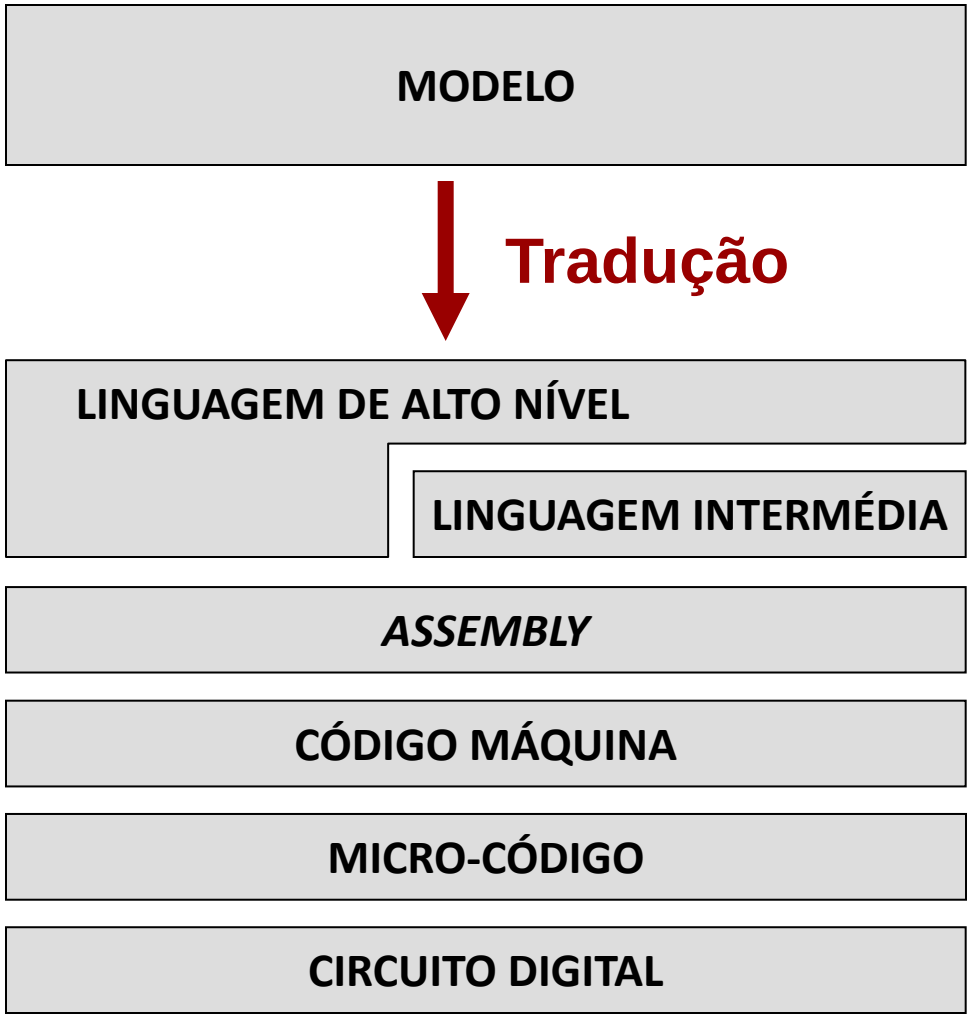


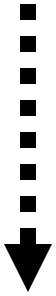

[Douglass, 2007]

# Níveis de Especificação de Software

***Engenharia de código inversa*** (*Reverse engineering*)
Processo de conversão de código fonte em elementos do modelo





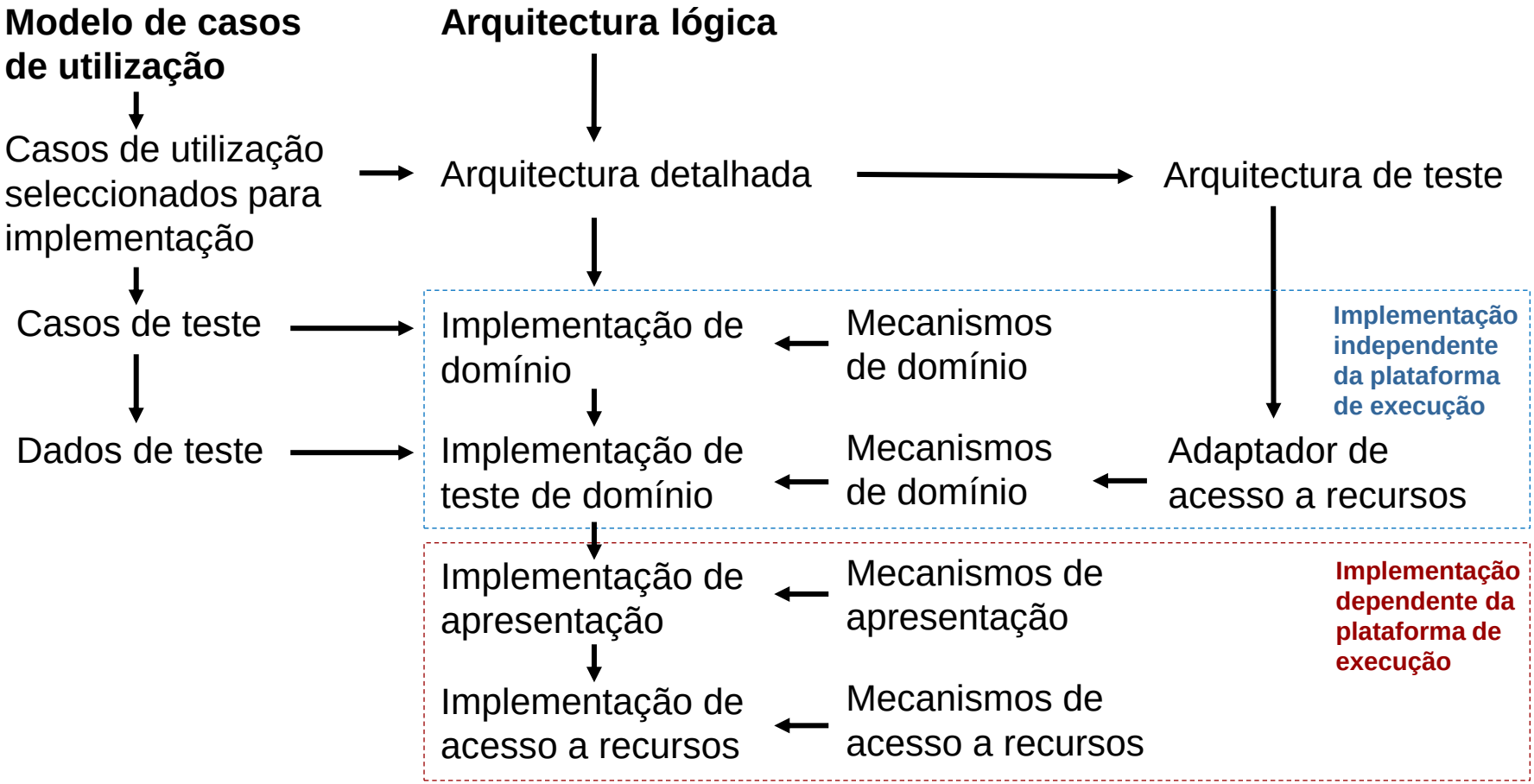



***Engenharia de código directa*** (*Forward engineering*)
Processo de geração de código fonte para um ou mais elementos do modelo

**Conversão para plataforma de execução específica**

# Processo de Implementação

## Implementação modular e incremental




**A implementação de domínio deve ser independente de plataformas de execução**

A dependência de plataformas de execução deve ser abstraída pela implementação de apresentação e de acesso a recursos, de modo a garantir a flexibilidade e facilidade de teste e manutenção

# Conversão Modelo – Implementação

**Relação entre elementos elaborados em diferentes actividades de desenvolvimento**

| Análise | Concepção | Implementação |
|---|---|---|
| Objecto de Análise | Classe ou Mecanismo | Uma ou mais classes |
| Comportamento de um objecto | Operação | Método |
| Atributo (classe) | Atributo (classe) | Variável estática da classe |
| Atributo (instância) | Atributo (instância) | Variável da instância |
| Associação | Associação (detalhada) | Referência |
| Interacção entre objectos | Mensagem/Evento | Chamada de um método |
| Caso de utilização | Caso de Utilização (detalhado)<br>Realização de Caso de Utilização | Sequência de chamadas de métodos |
| Subsistema | Subsistema | Ficheiro ou conjunto de ficheiros |

# Conversão Modelo – Implementação

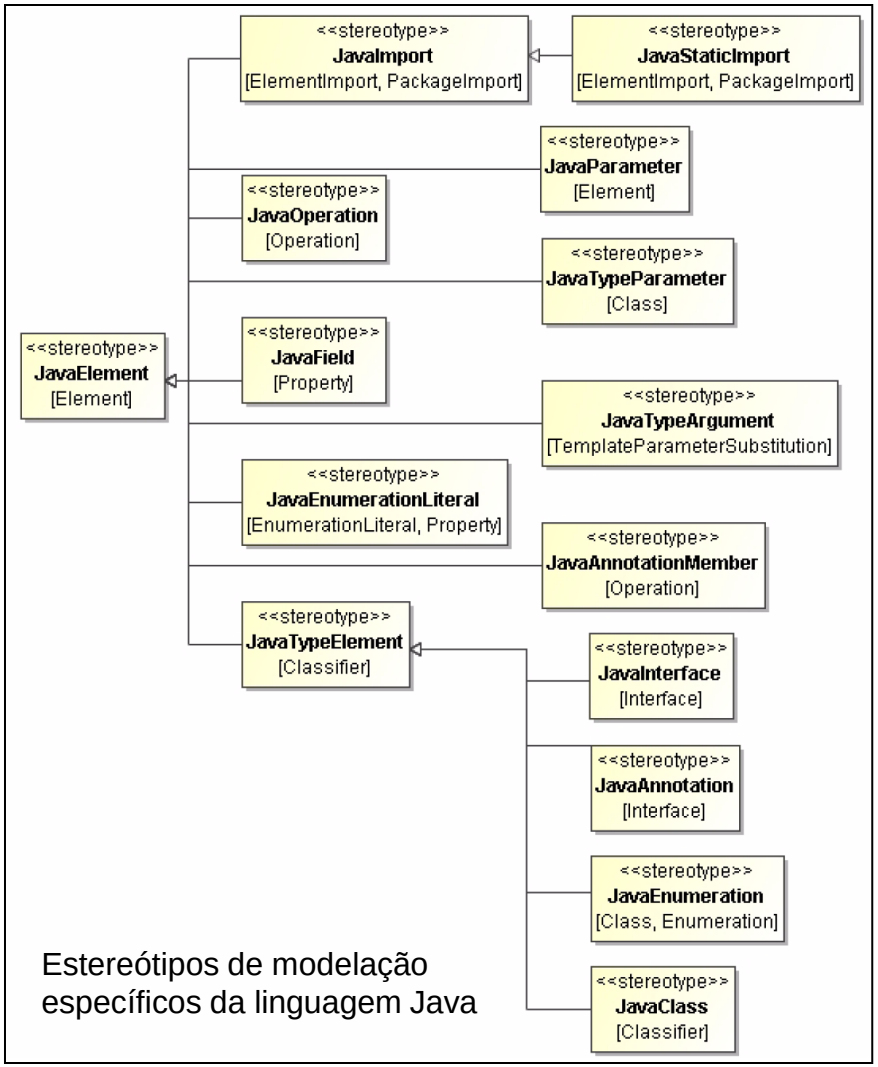
## Perfis de Modelação

Os diferentes níveis de modelação são caracterizados por elementos com semântica específica

Nas ferramentas de modelação, esses elementos são organizados em perfis de modelação (*profiles*), que podem ser utilizados no contexto da arquitectura detalhada para incluir informação específica relevante para a conversão para a linguagem alvo

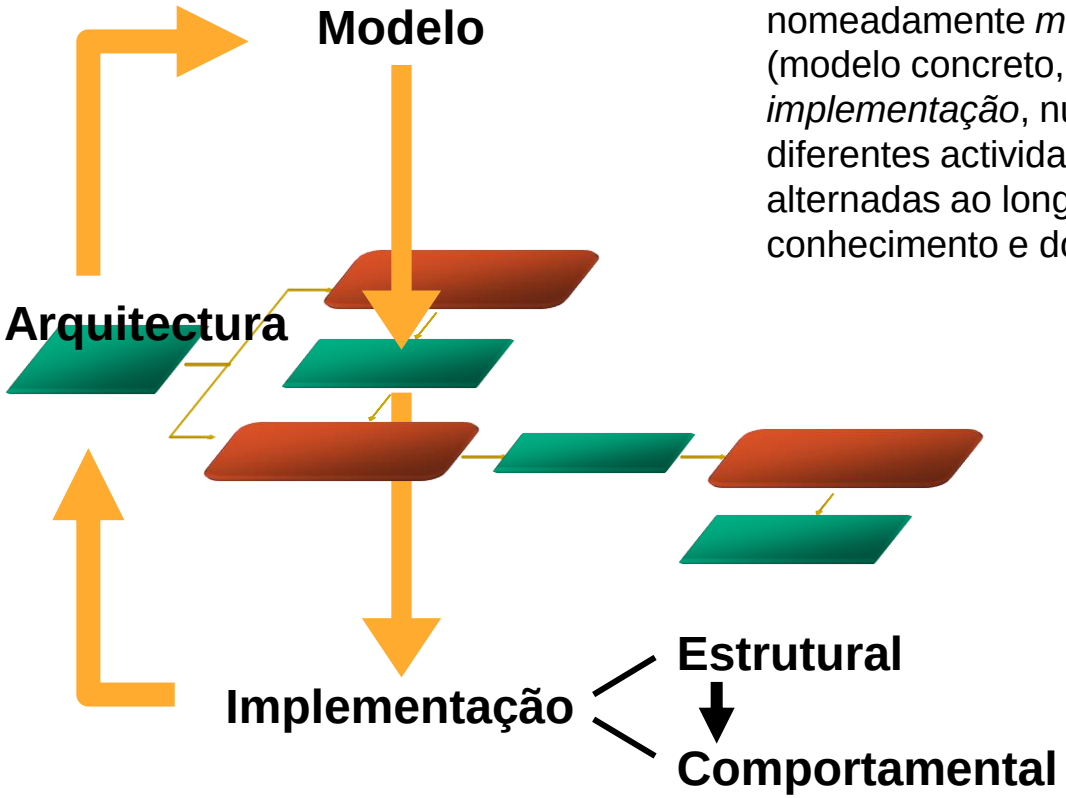
Os elementos específicos são representados em UML sob a forma de estereótipos

**Exemplo**: Perfil Java da ferramenta *MagicDraw*




Estereótipos de modelação específicos da linguagem Java

# Etapas de Implementação




O *processo de desenvolvimento* de software consiste na criação da organização de um sistema de forma progressiva, através de diferentes níveis de abstracção, nomeadamente *modelo* (conceptual), *arquitectura* (modelo concreto, orientado para a implementação) e *implementação*, num processo iterativo em que as diferentes actividades de desenvolvimento são alternadas ao longo do tempo em função do conhecimento e do nível de detalhe envolvido

A implementação deve ser realizada em duas etapas principais:

- **Implementação estrutural**
  - Realização de código relativo à estrutura do sistema (*código estrutural*)
- **Implementação comportamental**
  - Realização de código relativo ao comportamento do sistema (*código comportamental*)

# Implementação Estrutural

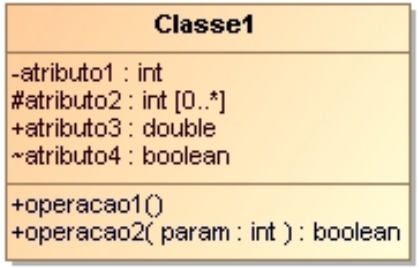
A implementação estrutural consiste na concretização dos modelos estruturais da arquitectura de um sistema, nomeadamente das partes e das relações entre partes

- **Implementação de partes**:
  - **Classes**: representam tipos de objectos
  - **Interfaces**: representam conjuntos de funcionalidades realizadas por diferentes partes
  - **Módulos**: representam mecanismos coesos, compostos por diferentes partes que interagem para realizar uma responsabilidade específica
  - **Subsistemas**: representam agregados de classes e módulos relacionados entre si, nomeadamente, pelo tipo de responsabilidades que concretizam
- **Implementação de relações entre partes**
  - As relações entre partes são concretizadas em função da semântica específica definida, por exemplo, herança ou composição, e do tipo de partes envolvidas, podendo envolver a definição de atributos para concretização de relações estruturais
- **Implementação de atributos**
  - Os atributos das partes são concretizados de acordo com as suas caraterísticas, por exemplo, tipo e visibilidade
- **Implementação de operações**
  - As operações são concretizadas em termos da sua forma estática, ou seja, a assinatura dos respectivos métodos, a implementação da lógica das operações deve ser realizada após a implementação estrutural dos módulos ou subsistemas respectivos

# Conversão UML - Java

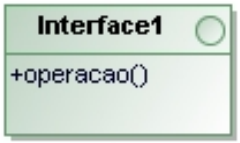
## Classificadores base

Classe:




```
public class Classe1
{
    private int atributo1;
    protected int[] atributo2;
    public double atributo3;
    boolean atributo4;

    public void operacao1()
    {
    }

    public boolean operacao2(int param)
    {
        return false;
    }
}
```

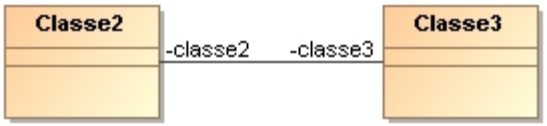
Interface:




```
public interface Interface1
{
    void operacao( );
}
```

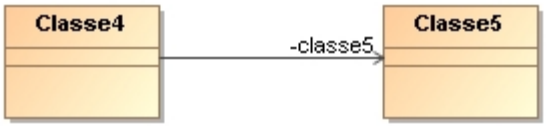
# Conversão UML - Java

## Relações entre classes



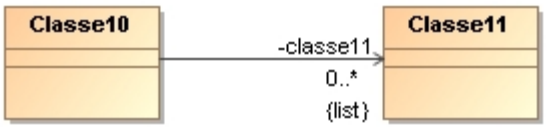

```
public class Classe2
{
    private Classe3 classe3;
}
```

```
public class Classe3
{
    private Classe2 classe2;
}
```




```
public class Classe4
{
    private Classe5 classe5;
}
```

```
public class Classe5
{

}
```

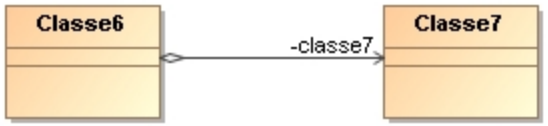



```
public class Classe10
{
    private java.util.List<Classe11> classe11;
}
```

# Conversão UML - Java

## Relações entre classes

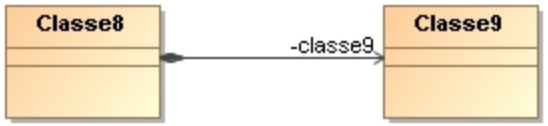
Agregação:




```
public class Classe6
{
    private Classe7 classe7;
}
```

```
public class Classe7
{

}
```

Composição:

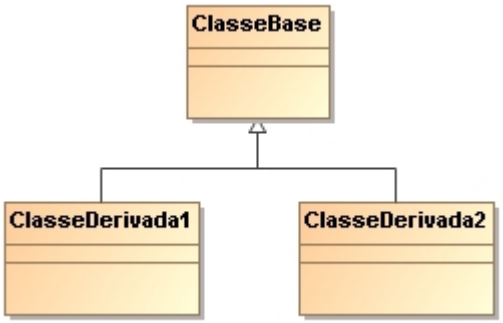



```
public class Classe8
{
    private Classe9 classe9 = new Classe9();
}
```

```
public class Classe9
{

}
```

# Conversão UML - Java

## Herança (generalização/especialização)

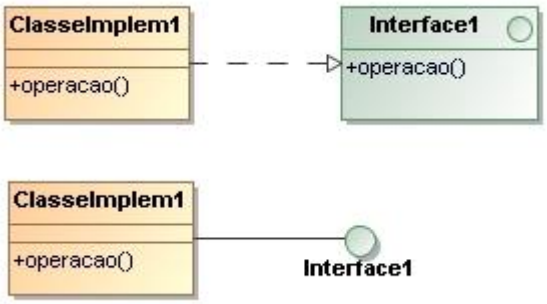



```java
public class ClasseBase
{

}
```

```java
public class ClasseDerivada1 extends ClasseBase
{

}
```

```java
public class ClasseDerivada2 extends ClasseBase
{

}
```

## Realização de interfaces

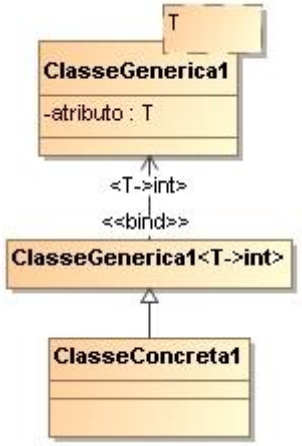



```java
public interface Interface1
{
    void operacao();
}
```

```java
public class ClasseImplem1 implements Interface1
{
    void operacao()
    {

    }
}
```

# Conversão UML - Java

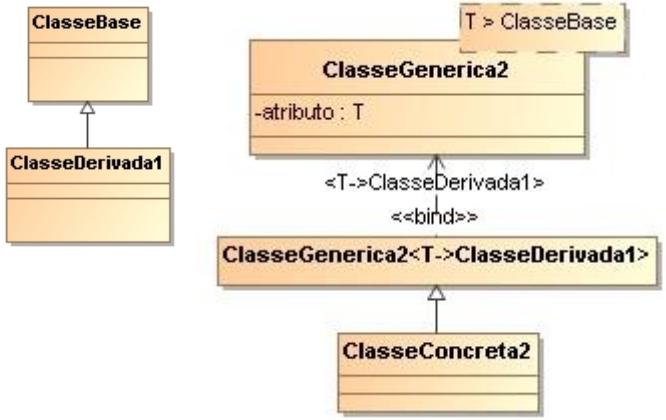
## Classes genéricas




```
public class ClasseGenerica1<T>
{
    private T atributo;
}
```

```
public class ClasseConcreta1 extends ClasseGenerica1<int>
{

}
```

## Classes genéricas com restrição de tipos base

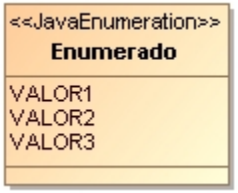



```
public class ClasseGenerica2<T extends ClasseBase>
{
    private T atributo;
}
```

```
public class ClasseConcreta2 extends ClasseGenerica2<ClasseDerivada1>
{

}
```

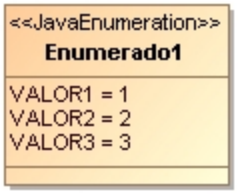
# Conversão UML - Java

## Enumerados




```
public enum Enumerado
{
    VALOR1,
    VALOR2,
    VALOR3
}
```

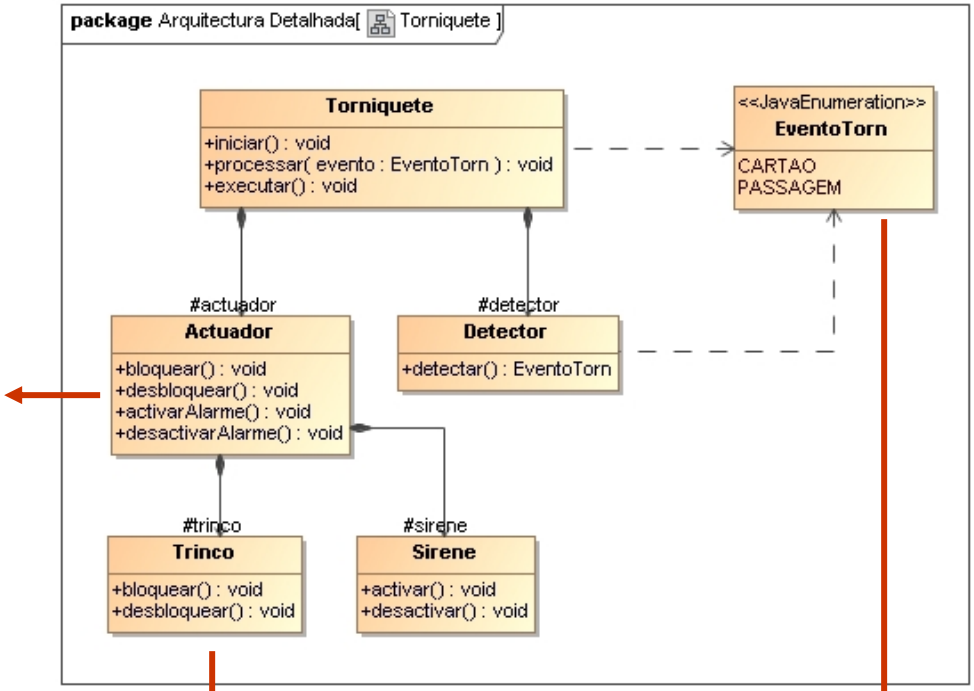
## Enumerados com valores específicos




```
public enum Enumerado1
{
    VALOR1(1),
    VALOR2(2),
    VALOR3(3)
}
```

# Exemplo: Implementação de Estrutura

```
public class Actuador
{
    protected Sirene sirene = new Sirene();
    protected Trinco trinco = new Trinco();

    public void bloquear() {
        trinco.bloquear();
    }

    public void desbloquear() {
        trinco.desbloquear();
    }

    public void activarAlarme() {
        sirene.activar();
    }

    public void desactivarAlarme() {
        sirene.desactivar();
    }
}
```




```
public class Trinco
{
    public void bloquear() {
    }

    public void desbloquear() {
    }
}
```

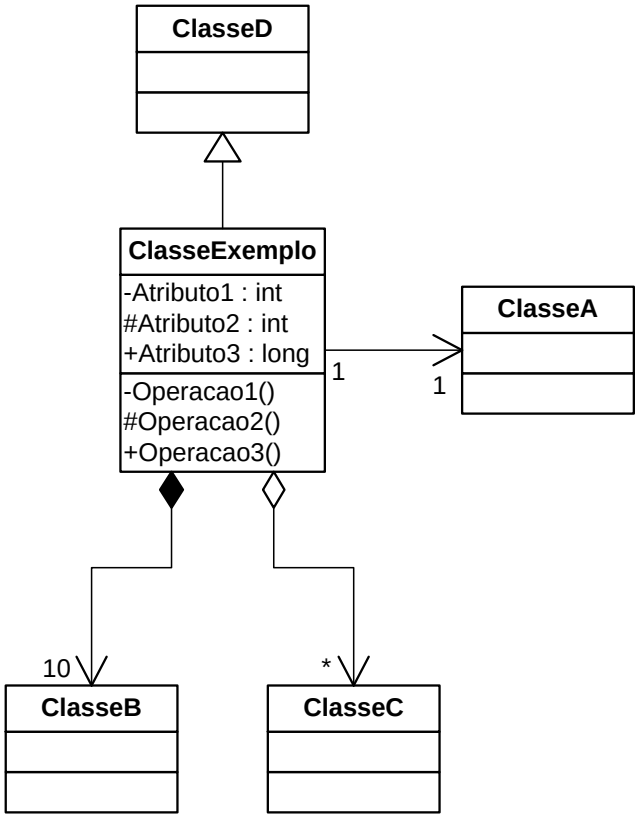
```
public enum EventoTorn
{
    CARTAO,
    PASSAGEM
}
```

## Implementação de estrutura

- É possível a tradução sem ambiguidade desde que o modelo seja preciso e tenha a informação detalhada para o efeito

# Conversão UML – C#

## Exemplo

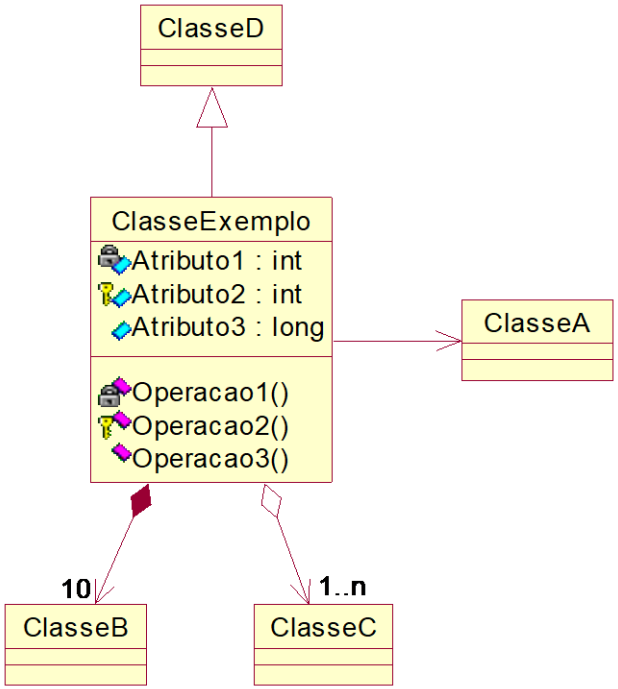



*Visio for Enterprise Architects 2005 Beta*
(*Visual Studio 2005 TeamSystem Beta*)

```
public class ClasseExemplo : ClasseD
{
     private int Atributo1;
     protected int Atributo2;
     public long Atributo3;

     private ClasseA the_A;
     private ClasseB [] the_B = new ClasseB[10];
     private System.Collections.ArrayList the_C;

     public void Operacao3()
     {
     }

     protected void Operacao2()
     {
     }

     private void Operacao1()
     {
     }
}
```

# Conversão UML – C++

## Exemplo




*IBM Rational Rose*

```
class ClasseExemplo : public ClasseD
{
  private:
     int Atributo1;

  protected:
     int Atributo2;

  public:
     long Atributo3;

  private:
     void Operacao1 ();

  protected:
     void Operacao2 ();

  public :
     void Operacao3 ();

  private:
      ClasseA *the_ClasseA;
      ClasseB the_ClasseB[10];
      UnboundedSetByReference<ClasseC> the_ClasseC;
 };
```

# Implementação Comportamental

A implementação comportamental consiste na concretização das operações associadas às partes dos sistema, tendo por base os modelos de interacção e de dinâmica

Diagramas UML que suportam a implementação comportamental:

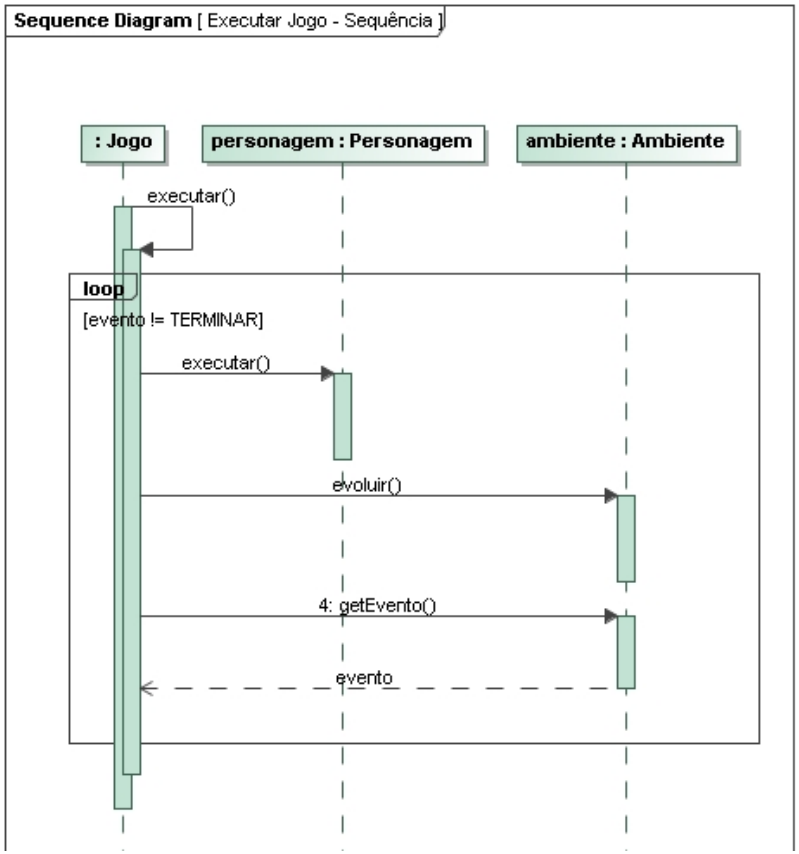
- **Diagramas de interacção**
  - Especificam a forma como as partes interagem entre si e com o exterior
  - Suportam a definição da lógica das operações, em particular no que se refere ao fluxo de controlo e ao encadeamento de interacções entre as partes
- **Diagramas de actividade**
  - Especificam os encadeamentos de acções e actividades que ocorrem nas partes
  - Suportam a definição da lógica das operações, em particular no que se refere ao encadeamento de acções a realizar
- **Diagramas de transição de estado**
  - Especificam estados e transições entre estados das partes
  - Suportam a definição de comportamentos complexos que requerem manutenção de estado

# Conversão UML - Java

## Diagramas de interacção

Os diagramas de interacção são um suporte para a definição da lógica das operações, em particular no que se refere ao fluxo de controlo e ao encadeamento de interacções entre as partes, no entanto, no processo de conversão para uma linguagem alvo pode ser necessário a decisão acerca de aspectos que não estão explícitos no modelo, nomeadamente, no que se refere a declaração de variáveis e atributos ou ao detalhe de instruções de fluxo de controlo, pelo que, no caso geral, não é possível a tradução automática

### Exemplo




```
public class Jogo
{
    private Personagem personagem;
    private Ambiente ambiente;

    private void executar()
    {
        Evento evento;
        do {
            personagem.executar();
            ambiente.evoluir();
            evento = ambiente.getEvento();
        } while (evento != Evento.TERMINAR);
    }
}
```

Neste exemplo, entre outros aspectos, a decisão acerca do tipo de ciclo (neste caso *do-while*) só pode ser realizada com a compreensão que a variável *evento*, envolvida na decisão, apenas está disponível após a última interacção do ciclo

# Conversão UML - Java

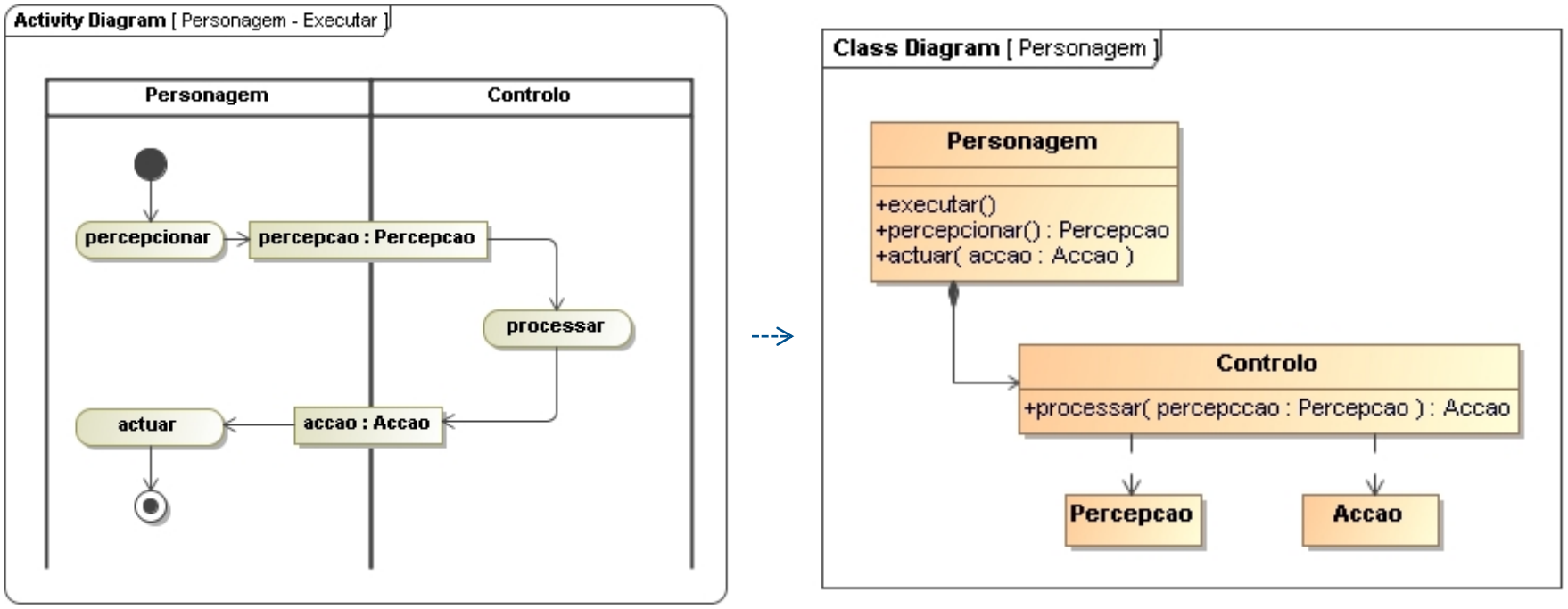
## Diagramas de actividade

Os diagramas de actividade especificam o fluxo de acções e actividades que ocorrem nas partes, indicando o encadeamento de acções a realizar e respectivas decisões

A conversão para uma linguagem alvo implica a definição das partes envolvidas, caso sejam especificadas partições (*swimlanes*), e a definição das operações que implementam as sequências de acções definidas

Numa primeira etapa, a tradução pode requerer a evolução dos modelos de estrutura para definir as acções realizadas pelas partes como operações dessas partes

### Exemplo

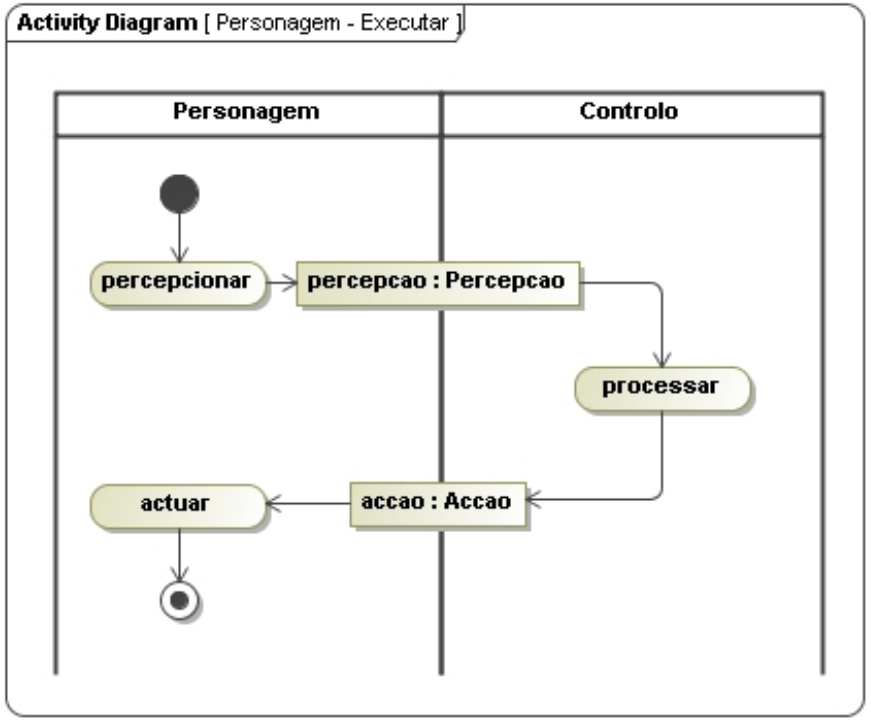


# Conversão UML - Java

## Diagramas de actividade

Para além da definição das partes envolvidas, caso sejam especificadas partições (*swimlanes*), a conversão de diagramas de actividade em código fonte consiste na implementação das operações que realizam os encadeamentos de acções especificados

### Exemplo




```
public class Personagem
{
    private Controlo controlo;

    public void executar()
    {
        Percepcao percepcao = percepcionar();
        Accao accao = controlo.processar(percepcao);
        actuar(accao);
    }

    private Percepcao percepcionar()
    {

    }

    private void actuar(Accao accao)
    {

    }
}
```

# Conversão UML - Java

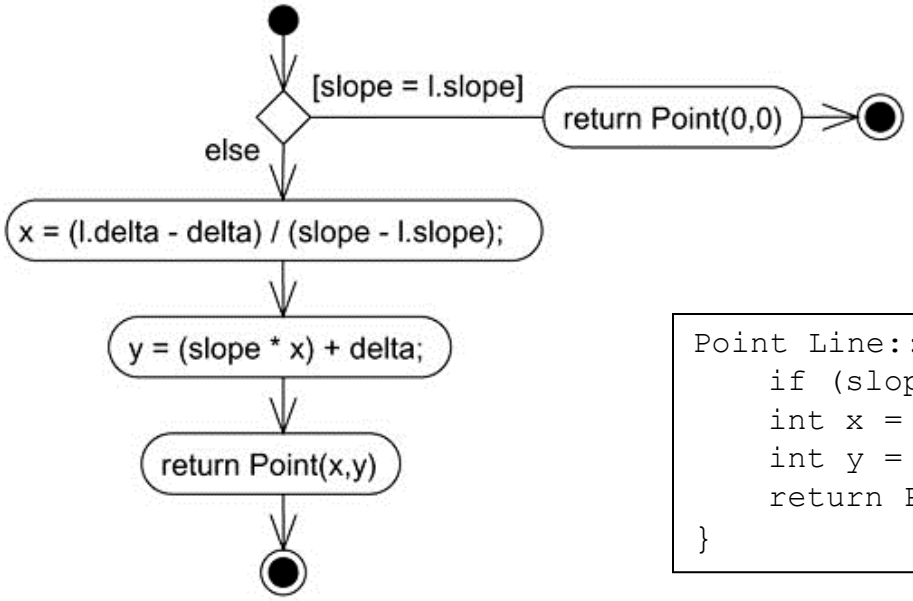
## Diagramas de actividade

### Modelação de operações

A utilização de diagramas de actividade para modelar uma operação é relevante quando o comportamento dessa operação é difícil de compreender se especificado directamente na linguagem alvo

Nessas situações, a elaboração de um diagrama de actividade poderá, por exemplo, salientar aspectos de um algoritmo que não seriam evidentes apenas pela observação do código na linguagem alvo

### Exemplo




```
Point Line::intersection (l : Line) {
    if (slope == l.slope) return Point(0,0);
    int x = (l.delta - delta) / (slope - l.slope);
    int y = (slope * x) + delta;
    return Point(x, y);
}
```

[UML User Guide, Booch *et al.* 1998]

# Conversão Implementação - Modelo

## Conversão de código para modelo

(**Engenharia de código inversa** – ***Reverse engineering***)

Processo de conversão de código fonte em elementos do modelo

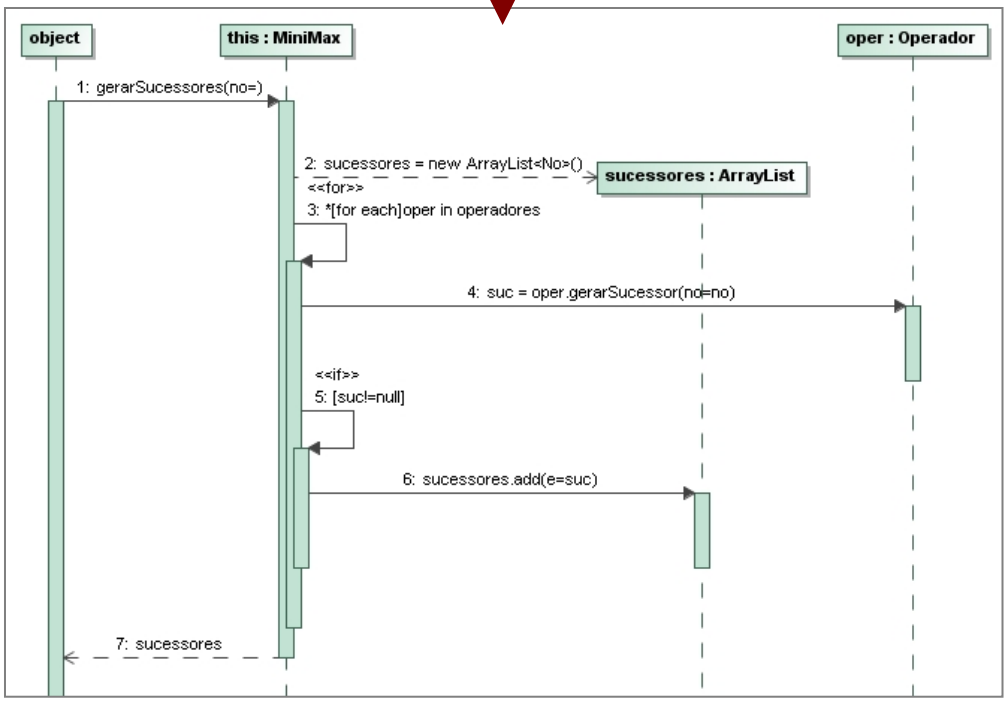
A conversão de implementação (código fonte) para modelos pode abranger a *perspectiva estrutural* e a *perspectiva comportamental*

No caso da *perspectiva comportamental*, é possível gerar automaticamente modelos de interacção, nomeadamente *diagramas de sequência*, a partir do código fonte de operações

É relevante para actualizar os modelos a partir de alterações realizadas no código fonte, de modo a sincronizar modelos e código

Também é relevante para analisar e compreender código pré-existente (*código legado*), antes de ser realizada alguma intervenção sobre esse código

```
private ArrayList<No> gerarSucessores(No no)
{
    No suc;
    ArrayList<No> sucessores = new ArrayList<No>();

    // Para todos os operadores gerar sucessor do nó
    for(Operador oper : operadores) {
        suc = oper.gerarSucessor(no);
        if(suc != null)
            sucessores.add(suc);
    }

    return sucessores;
}
```



# Bibliografia


[Pressman, 2003]

R. Pressman, *Software Engineering: a Practitioner's Approach*, McGraw-Hill, 2003.

[Gamma et al., 1995]

Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1995.

[Shaw & Garlan, 1996]

M. Shaw, D. Garlan, *Software Architecture: Perspectives on an Emerging Discipline*, Prentice-Hall, 1996.

[Vernon, 2013]

V. Vernon, *Implementing Domain Driven Design*, Addison-Wesley, 2013.

[Parnas, 1972]

D. Parnas, *On the Criteria to Be Used in Decomposing Systems into Modules*, Communications of the ACM 15-12, 1968.

[Kruchten, 1995]

F. Kruchten, *Architectural Blueprints - The "4+1" View Model of Software Architecture*, IEEE Software, 12-6, 1995.

[Booch *et al*., 1998]

G. Booch, J. Rumbaugh, I. Jacobson, *UML User Guide*, Addison-Wesley, 1998.

[Booch, 2004]

G. Booch, *Software Architecture*, IBM, 2004.

[Douglass, 2007]

B. Douglass, *Real Time UML*, OOP-2007, 2007.