Um pouco sobre diagramas de classes em UML

João Paulo Barros
Escola Superior de Tecnologia e Gestão
Instituto Politécnico de Beja

10 de Junho de 2008 – revisto em 11 de Maio de 2009

Este texto apresenta a parte que se considera mais importante dos diagramas de classes da UML 2.2, cuja especificação se encontra disponível no sítio Internet da Object Management Group (OMG) em `http://www.uml.org`. Enfatiza-se a ligação dos diagramas de classes ao código, nomeadamente utilizando a linguagem Java™. Actualmente, a especificação da UML tem $226 + 740 = 966$ páginas e muitas delas estão relacionadas com os diagramas de classes. Ainda assim, o que "realmente interessa" sobre diagramas de classes está aqui!

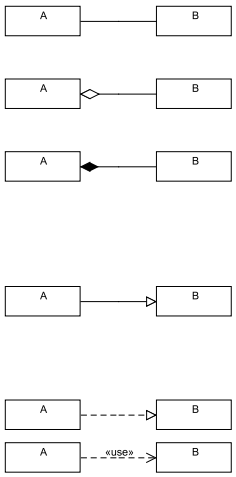
Os programas orientados pelos objectos podem (e muitas vezes devem) ser especificados por linguagens gráficas, visuais ou "diagramáticas"(do inglês *diagrammatic*). Ou seja, por desenhos ou diagramas. A razão está no facto, bem conhecido, das imagens poderem facilitar a visualização e compreensão de muitas coisas: problemas, estruturas, interacções, etc.. Muito provavelmente, o diagrama mais utilizado no desenvolvimento de programas orientados pelos objectos é o **DIAGRAMA DE CLASSES** parte da **UNIFIED MODELING LANGUAGE** (**UML**[1]).

diagrama de classes UML

Num diagrama de classes cada classe (de objectos pois claro) é representada por um rectângulo contendo o nome da classe. As linhas entre rectângulos indicam relações entre os objectos dessas classes.

As cinco **relações** (*Relationship*) de que iremos falar são classificadas da seguinte forma (ver exemplo na Fig. 1):

1. Associação (*Association*—▷*Relationship*). Vamos considerar apenas associações binárias. Cada extremo da associação pode assumir um de três valores
   (a) **Associação "simples"** (***AggregationKind=none***)(e.g. entre `Teacher` e `CurricularUnit`)
   (b) **Agregação partilhada** (***AggregationKind=shared***) (e.g. entre `Course` e `CurricularUnit`)
   (c) **Agregação composta** (***AggregationKind=composite***) (e.g. entre `CurricularUnit` e `Syllabus`)
2. Relação dirigida (*DirectedRelationship*—▷*Relationship*)
   (a) **Generalização/Herança** (***Generalization***—▷*DirectedRelationship)* (e.g. entre `LocalStudent` e `Student`)
   (b) Dependência (*Dependency*—▷*DirectedRelationship*)
      i. **Realização de interface** (***InterfaceRealization***—▷*Realization*—▷*Abstraction*—▷*Dependency*) (e.g. entre `Student` e `Comparable`)
      ii. **Utilização** (***Usage***—▷Dependency) (e.g. entre `Syllabus` e `Validator`)



[1] `http://www.uml.org`

A Fig. 1 apresenta exemplos das várias relações.

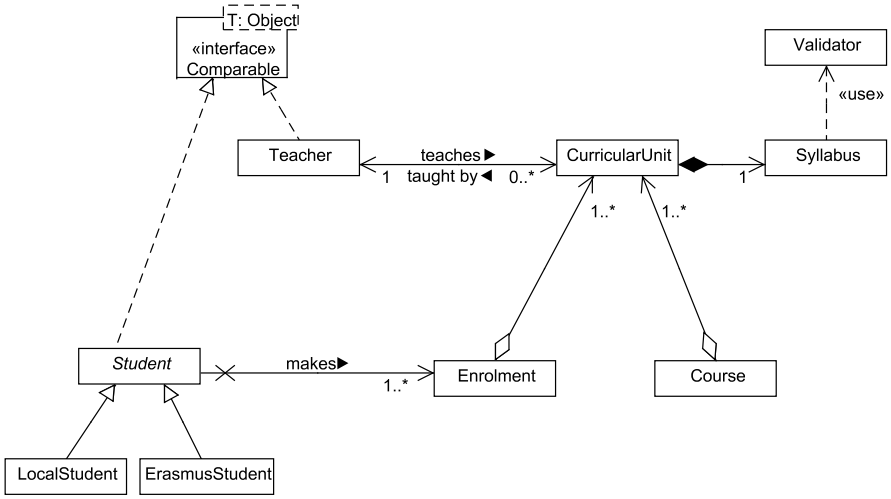


Figura 1: Relações em UML.

Seguidamente, apresentamos os vários tipos de relação que iremos utilizar.

1 Associação

A associação é a forma mais comum de compor objectos: um objecto tem outro. Nada mais simples! Bom, na verdade, não é assim tão simples e por duas razões:

1. Um objecto pode *ter* ou *estar associado* a outro sem que este seja *parte do* primeiro. Por exemplo, dizemos que um professor tem ou está associado a unidades curriculares, mas estas não são parte do professor.
2. Em rigor, um objecto *tem o nome* de outro objecto e não o próprio objecto. Tal permite que um objecto pertença a mais do que um outro objecto!

associação

Ao primeiro tipo em que um objecto tem outro sem que esse seja parte dele chamamos *associação simples* ou simplesmente **ASSOCIAÇÃO**, e já está.

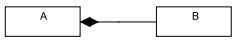
agregação

Ao segundo tipo de associação, quando podemos dizer convictos que um objecto é parte de outro, chamamos **AGREGAÇÃO**. Se tivermos dúvidas devemos evitar a agregação e especificar uma associação "simples", ou seja, sem agregação. Se considerarmos que se trata de uma agregação podemos ainda utilizar uma de duas formas:

Agregação partilhada Um objecto da classe `A` contém (o nome de) um objecto da classe `B`, mas sem exclusividade, ou seja, outros objectos podem conter o nome do mesmo objecto da classe `B`. Representa-se colocando um losango sem preenchimento (vazio) no lado do objecto (o composto) que contém o outro (o componente).

Agregação composta Em cada instante apenas um objecto da classe A contém (o nome de) um objecto da classe B. Representa-se colocando um losango com preenchimento (a cheio) no lado do objecto (o COMPOSTO) que contém o outro (o COMPONENTE). O composto tem a responsabilidade pela existência e armazenamento do composto.

composto
componente



Regra geral, as associações são implementadas no código (por exemplo em Java™) colocando nomes de objectos dentro de outros. Por vezes, diferentes associações são implementadas por igual código. Tal significa, que a diferença pode só estar presente no diagrama mas não no código. Tal é especialmente frequentemente para a associação simples e a agregação partilhada. Este facto não deve surpreender pois o diagrama de classes é um modelo mais abstracto (com menos detalhes) do que o código. Ou seja, o código pode e deve ser visto como um modelo mais detalhado e portanto menos abstracto do que o diagrama de classes.

Seguem-se os "esqueletos" de código Java™ que especificam as associações apresentadas na Fig. 1.

```
class Teacher implements Comparable<Teacher>
{
   private List<CurricularUnit> currUnits; // association
   ...
}


abstract class Student implements Comparable<Student>
{
   private List<Enrolment> es; // association
   ...
}

class CurricularUnit
{
   private Teacher teacher; // association
   private Syllabus syllabus; // composite aggregation
   ...
}

class Enrolment
{
   private List<CurricularUnit> units; // shared aggregation
   ...
}

class Course
{
   private List<CurricularUnit> units; // shared aggregation
   ...
}
```

1.1 Navegabilidade

Nas associações é possível indicar a navegabilidade entre os objectos, ou seja, se um objecto tem acesso ao outro, tipicamente devido a ter um atributo desse ou-

tro objecto. A associação entre a classe `Student` e a classe `Enrolment`, na Fig. 1, é um exemplo dos dois tipos de notação: a cruz e a seta aberta. Naturalmente, a seta indica que os objectos da classe `Student` podem alcançar objectos da classe `Enrolment`. A cruz do lado da classe `Student` indica que os objectos da classe `Enrolment` não podem alcançar os da classe `Student`.

1.2 Multiplicidade

Ainda podemos adicionar mais informação nas associações. A linguagem UML admite os seguintes indicadores de multiplicidade[2].

Indicador	Multiplicidade
`0..1`	zero ou um
`1`	um
`0..*` ou apenas *	zero ou mais
`1..*`	um ou mais
`n..*`	n ou mais (com $n > 1$)
`n`	apenas n (com $n > 1$)
`0..n`	zero a n (com $n > 1$)
`1..n`	um a n (com $n > 1$)
`n..m`	n a m (com $n > 1$, $m > 1$ e $n < m$)

As letras `n` e `m` representam números inteiros. Por exemplo, `1..n` significa que são permitids todas as seguintes especificações: `1..1`, `1..2`, `1..3`, etc.. Note que `1..1` é igual a `1` e que o valor após o `..` tem de ser maior ou igual do que está antes de `...` Por exemplo, `3..2` não é permitido

2 Relações *é-um* (em inglês *is-a*)

A generalização/herança e a realização de interfaces, correspondem ao tipo de relação mais característico da programação orientada pelos objectos. É nelas que se baseia a ligação dinâmica (*dynamic binding*) para suporte ao polimorfismo o qual permite aumentar a coesão das classes através da ocultação automática de *if*s ou *switch*es para escolher entre diversos tipos.

generalização/herança
realização
é um

Quer a **GENERALIZAÇÃO/HERANÇA** (**`extends`** em Java™) quer a **REALIZAÇÃO** (**`implements`** em Java™) significam ***é um*** (*is-a*). Por exemplo: cada `Student` e cada `Teacher` é um `Comparable`; um `LocalStudent` é um `Student`; um `ErasmusStudent` é um `Student`. Também podemos dizer que a classe `ErasmusStudent` herda da classe `Student`, ou ainda que a classe `Student` generaliza a classe `ErasmusStudent`. Herança é ainda o termo mais comum.

3 *tem-um* versus *é-um*

Devemos optar por uma associação/agregação ou uma generalização/herança? Numa perspectiva de alto-nível (independente do código), a diferença deve ser vista através do "teste" **é um** *is-a* versus **tem um** *has-a*: ou um `A` **tem um** `B`, ou um `A` **é um**

[2] *in* UML Superstructure Specification, v2.1.2, 7.3.32 MultiplicityElement (from Kernel)

B. Por exemplo, um avião **tem um** motor, mas dificilmente (apesar de verdadeiro!) diríamos um avião **é um** motor com mais coisas. Por outras palavras, devemos pensar no que faz mais sentido no *mundo real*.

Numa perspectiva de baixo-nível, quer a generalização/herança quer a associação/agregação permitem aumentar as funcionalidades dos objectos de uma classe. Aí a grande diferença está no facto da generalização/herança permitir a definição de métodos polimórficos.

Em caso de dúvida e se não necessitarmos de polimorfismo, devemos optar por associações/agregações em lugar de generalizações/heranças.

Seguem-se os "esqueletos" de código Java™ que especificam as generalizações apresentadas na Fig. 1.

```
class Teacher implements Comparable<Teacher>
{
   private List<CurricularUnit> units;
   ...
}

abstract class Student implements Comparable<Student>
{
   private List<Enrolment> enrolments;
   ...
}

class LocalStudent extends Student
{
   ...
}

class ErasmusStudent extends Student
{
   ...
}
```

4 Utilização

A utilização (*usage*) é uma forma de dependência no código (ou seja, na implementação). Dizemos que um objecto da classe `A` depende de outro da classe `B` quando necessita de algo nesse objecto para a sua definição. Pode ser vista como dependência de compilação: se a classe `B` não existir, a classe `A` não compila.

Note que também as outras relações implicam dependência entre os objectos. Assim, a relação de utilização deve ser utilizada apenas quando nenhuma das outras se aplica. Tipicamente existem duas situações em que existe utilização podendo não existir associação/agregação ou generalização/herança:

1. Quando o objecto da classe `A` utiliza um objecto da classe `B` como variável local de um método, mas não o guarda como atributo. Então `A` utiliza `B`.
2. Quando o objecto da classe `A` utiliza um objecto da classe `B` que recebeu como parâmetro mas não o guarda como atributo.

5 Relações e classes nos diagramas

A Fig. 2 apresenta os diagramas para as várias relações.

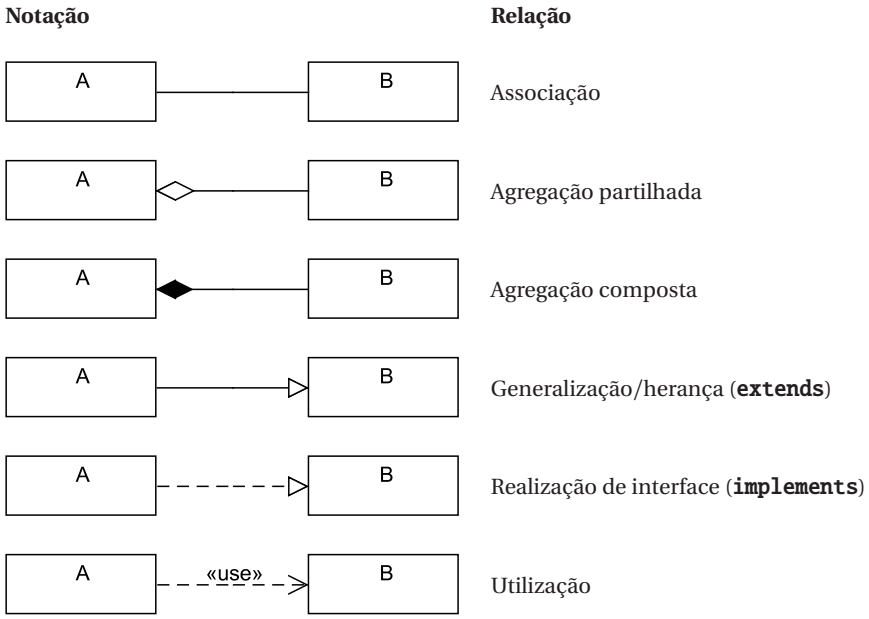


Figura 2: Notação para as relações apresentadas

Cada classe pode conter também os seus atributos e métodos. Para tal o respectivo rectângulo deve ser dividido em três partes. Por exemplo, se a classe `Student` contiver um atributo para o respectivo nome, outro para o número mecanográfico de aluno e os respectivos *getters*, mais um método **`private`** `somethingToDo(int n)`, tal pode ser representado como se exemplifica na Fig. 3.

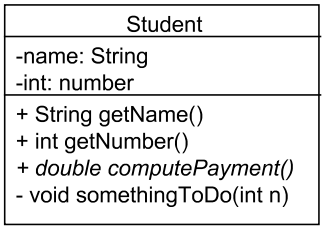


Figura 3: Uma classe com indicação de atributos (ambos privados) e três métodos: dois públicos e um privado

Os sinais antes dos atributos e métodos indicam a visibilidade dos mesmos:

Notação	Visibilidade
+	**`public`**
-	**`private`**
#	**`protected`**
~	*package visibility* (sem qualificador em Java™)

MUITO IMPORTANTE: os atributos que especificam associações **não** devem ser representados dentro da classe. Por exemplo, o atributo `syllabus` para objectos

da classe `CurricularUnit` não deve ser colocado no diagrama de classes porque já lá está sob a forma de notação gráfica, mais especificamente sob a forma de uma linha, com um losango a cheio numa das extremidades, que une as duas classes.

Como regra, representamos como atributos os de um tipo primitivo (por exemplo **`int`**, **`double`**) ou de classes já feitas e de muito comum utilização como a classe `String`. Os atributos que correspondam a outras classes por nós feitas devem ser representados de forma implícita utilizando a notação gráfica para as associações. Só no código todos terão que surgir: os que já estavam dentro da classe no diagrama e os que resultaram das especificações gráficas.

6 Leituras complementares

Para saber mais aqui ficam alguns links com mais informação sobre diagramas de classes, e não só, em UML 2:

1. Allen Holub , "Allen Holub's UML Quick Reference", disponível em `http://www.holub.com/goodies/uml`.
2. Scott W. Ambler, "UML 2 Class Diagrams", disponível em `http://www.agilemodeling.com/artifacts/classDiagram.htm`.
3. Scott W. Ambler, "UML 2 Class Diagram Guidelines", disponível em `http://www.agilemodeling.com/style/classDiagram.htm`.

7 Ferramentas

Existem MUITAS ferramentas para a linguagem UML. A wikipedia apresenta um boa lista em `http://en.wikipedia.org/wiki/List_of_UML_tools`.

Se se pretender apenas desenhar, a UMLet (`http://www.umlet.com/`) é especialmente simples e eficiente. Foi a única ferramenta utilizada para desenhar os diagramas neste texto.

Agradecimentos

A professora Isabel Sofia Brito e o aluno João Paulo Pereira Candeias apresentaram-me sugestões que contribuíram para melhorar este texto.