Diagramas de Classe

Sumário

- Colaborações
- Orientação aos Objectos
- Diagramas de Classe I — conceitos base
- Diagramas de Classe II — conceitos avançados
- Relações — conceitos avançados
- Diagramas de objectos
- *Packages*

Colaborações

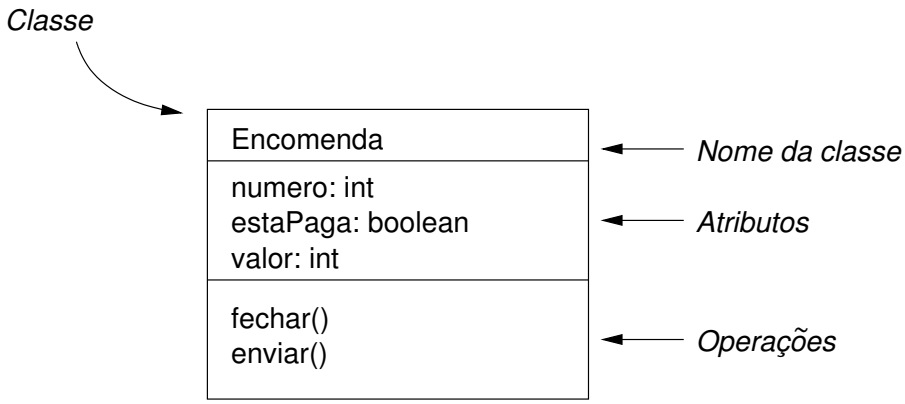
- Os diagramas de *use case* capturam os requisitos funcionais do sistema.
- Como construir um sistema que suporte esses *use cases*?
  - *Use cases* são realizados por colaborações.
- Colaborações representam interacções entre objectos.
  - Que objectos?
  - Que classes?
- As classes do sistema são representadas em *Diagramas de Classe*

Orientação aos Objectos

Classes

- A noção de classe é fundamental no paradigma OO
  - tipicamente uma classe representa uma **abstração** de uma entidade do mundo real.
- Cada classe descreve um conjunto de objectos com a mesma estrutura e comportamento:
  - Estrutura:
    - atributos
    - relações
  - Comportamento:
    - operações
- A organização do código em classes tem dois objectivos fundamentais:
  - facilitar a reutilização — através da reutilização de classes previamente desenvolvidas em novos sistemas;
  - facilitar a manutenção — o sistema deverá ser desenvolvido de forma a que a alteração de uma classe tenha o menor impacto possível no resto do sistema.

Representação de classes em UML



- Compartimentos pré-definidos
  - Nome da classe — começa com maiúsculas / substantivo
  - Atributos (de instância) — representam propriedades das instâncias desta classe / começam com minúsculas / substantivos
  - Operações (de instância) — representam serviços que podem ser pedidos a instâncias da classe / começam com minúsculas / verbos
- Compartimentos podem ser omitidos — isso não significa que não exista lá informação!

Alguns conceitos de OO (revisão)

- Identidade
  - Cada objecto (instância) tem identidade própria.
  - (Podem existir gémeos!)
- Classificação
  - Cada objecto pertence a uma classe.
  - A classe define a estrutura e comportamento do objecto.
- Herança/Especialização
  - Uma classe mais especifica pode ser definida a partir de uma classe mais geral.
  - A classe mais específica (sub-classe) *herda* o estado e comportamento da classe mais geral (super-classe).
- Polimorfismo — (*muitas formas*)
  - A mesma operação pode ter comportamentos diferentes em diferentes classes.
  - Uma forma de abstracção.
- Encapsulamento

  - Uma classe agrega um conjunto de dados e as operações sobre eles definidas.
- *Data Hiding*
  - O objecto pode esconder (parte d)a sua informação do exterior.
  - Deste modo garante que só ele a pode manipular o que facilita a manutenção de invariantes de dados e a manutenção do código.
- *Overriding*
  - Uma sub-classe pode redefinir o comportamento associado às operações que herda da super-classe.
- *Overloading*
  - Uma classe pode ter várias operações com o mesmo nome desde que as assinaturas sejam diferentes.
- Delegação
  - Cada objecto deve realizar uma tarefa muito bem e delegar noutros objectos tudo o que não seja a sua missão principal.

Três níveis de modelação

- Conceptual
  - Representar conceitos no domínio de análise
  - Não existirá necessáriamente um mapeamento directo para a implementação
  - Pensar nas responsabilidades de cada abstracção/classe (cf. cartões CRC)
- Especificação
  - Identificar interfaces software
  - Substituir responsabilidades por operações e atributos que as satisfação
- Implementação
  - Definir mais concretamente as classes
  - Indicar tipos e visibilidades de cada atributos/operação
  - Provavelmente o nível mais utilizado
  - mas, provavelmente, o nível de especificação é o mais útil (onde se deve passar mais tempo)

Diagramas de Classe I — conceitos base

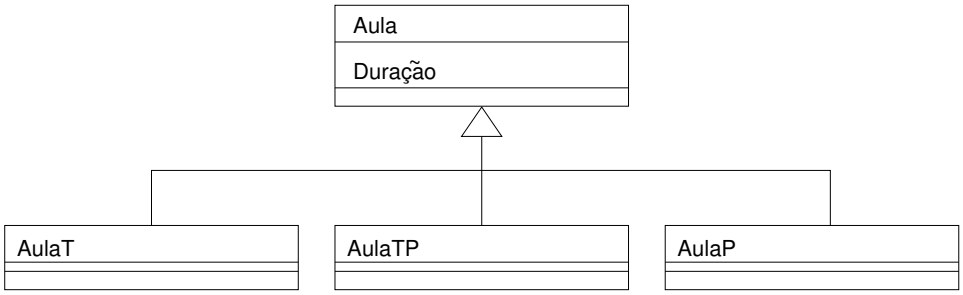
- Visão estática do sistema — indicam que tipos de objectos exitem no sistema e como este tipos se relacionam entre si.
- Três tipos de relações possíveis entre as classes:
  - Generalização/Especialização
    relação entre classe mais geral e classe mais específica
  - Dependência
    indica que uma classe depende de outra
  - Associação
    indica que existe algum tipo de ligação entre objectos das duas classes

Generalização/Especialização

- Indica a relação entre uma classe mais geral (super-classe) e uma classe mais específica (sub-classe).
- Noção de *is-a* — tipagem / substitubilidade
- Polimorfismo — duas sub-classes podem fornecer métodos diferentes para implementar

uma operação da super classe.

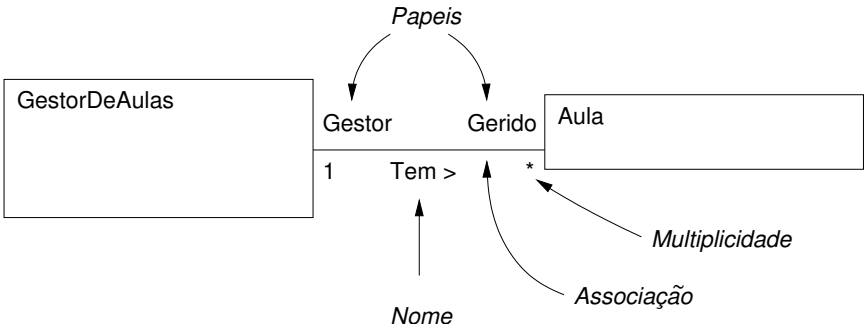
- *Overriding* — sub-classe pode alterar o método associado a uma operação declarada pela super-classe
- Herança simples vs. herança multipla
- Notação:



- Modos típicos de utilização:
  - *top-down* — depois de identificadas as classes que compõem o sistema, identificar sub-tipos das classes existentes.
  - *bottom-up* —depois de identificadas as classes que compõem o sistema, identificar o que é comum a um sub-grupo e elevar para classe mais geral.

Associação

- Notação:



- Indica que objectos de uma estão ligados a objectos de outra — define uma relação entre os objectos
- Noção de nevagabilidade (cf. diagramas E-R)
- Por omissão representam navegação bidireccional — mas pode indicar-se o sentido da navegabilidade utilizando setas nos extremos da associação.
- Três anotações possíveis:
  - nome — descreve a natureza da relação (pode ter direcção)
  - papeis — indica o papel que cada classe desempenha na relação definida pela asso-

ciação (usualmente utilizado como alternativa ao nome)

– multiplicidade — quantos objectos participam na relação:

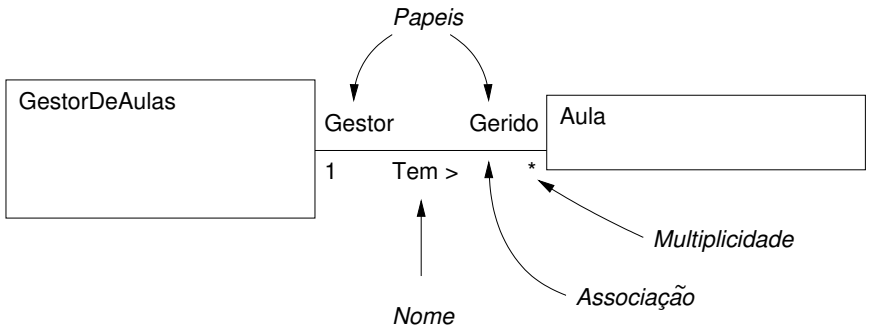
$*$ — zero ou mais objectos

$1..*$ — um ou mais objectos

n — n objectos

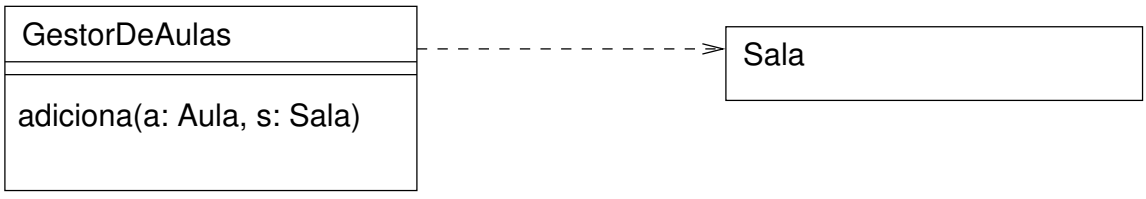
1 — um objecto / objecto obrigatório

$0..1$ — zero ou um objectos / objecto opcional



Dependência

- Notação:



- Indica que a definição de uma classe está dependente da definição de outra.
- Utiliza-se normalmente para mostrar que instâncias de origem utilizam, de alguma forma, instâncias de destino (por exemplo: um parâmetro de um método)
- Uma alteração no destino (quem é usado) pode alterar a origem (quem usa)
- <<include>> e <<extend>> são estereótipos de dependências.

Visibilidade de atributos e operações

O nível de visibilidade (acesso) que se pretende para cada atributo/operação é representado com as seguintes anotações:

– privado — só acessível ao objecto a que pertence (cf. encapsulamento)

\# protegido — acessível a instâncias das sub-classes (atenção: em Java fica também acessível a instâncias de classes do mesmo *package*!)

pacote/*package* — acessível a instâncias de classes do mesmo *package* (nível de acesso por omissão)

\+ público — acessível a todos os objectos no sistema (que *conhecam* o objecto a que o atributo/operação pertence!)

Diagramas de Classe II — conceitos avançados

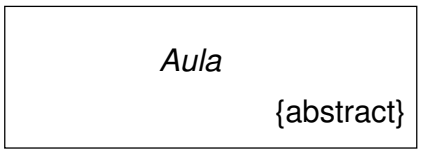
- Os conceitos já apresentados são usualmente suficientes para uma primeira abordagem à modelação a nível conceptual.
- Para além dos conceitos base já apresentados, o UML permite a representação de outros conceitos do paradigma da orientação aos objectos, necessários para passarmos a modelos de especificação e de implementação.

Operações e variáveis de classe

- Variáveis de classe são variáveis globais a todas as instâncias de uma classe.
- Métodos de classe são métodos executados directamente pela classe e não por uma das suas instâncias (logo, não têm acesso directo a variáveis/métodos de instância).
- São representados tal como variáveis/métodos de instância, mas sublinhados.
- Deve evitar-se abusar de operações e variaveis de classe.

Classes e operações abstractas

- Nem sempre ao nível da super-classe é possível saber qual deverá ser o método associado a uma operação.
- Quando se está a utilizar uma hierarquia de classes para representar sub-tipos, pode não fazer sentido permitir instâncias da super classe.
- Uma operação abstracta é uma operação que não tem método associado na classe em que está declarada.
- Uma classe abstracta é uma classe da qual não se podem criar instâncias e que pode conter operações abstractas.
- Classes concretas (não abstractas) não podem conter métodos abstractos!
- Notação: em *itálico* ou através da propriedade {abstract}.

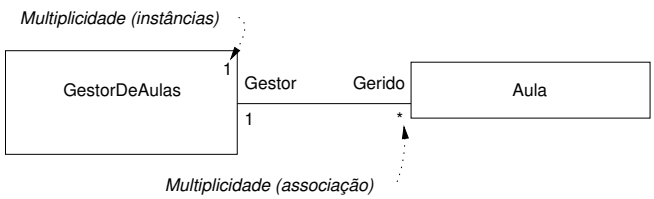


Classes root e leaf

- Classes etiquetadas com a propriedade {root} não podem ser generalizadas.
- Por exemplo, se o modelo apresenta classes pertencentes ao ambiente de desenvolvimento que irá ser utilizado, não será viável generalizar tais classes.
- Classes etiquetadas com a propriedade {final} não podem ser especializadas (classes final no Java).
- Por exemplo, se o sistema contém uma classe que fornece serviços de encriptação, por motivos de segurança não é desejável que os métodos associados às operações dessa classe possam ser redefinidos (isto também pode ser controlado ao nível das operações).

Multiplicidade de instâncias

- Tal como para as associações, também podemos definir a multiplicidade das instâncias de uma classe (qual o número válido de instâncias uma dada classe).
- Notação:



Declaração de atributos

- Sintaxe completa para a declaração de um atributo:

  `[visibilidade] nome [multiplicidade][:tipo][=valor inicial][{propriedades}]`

- Propriedades possíveis:
  - changeable — é possível alterar o atributo (valor por omissão)
  - addOnly — é apenas possível adicionar valores ao atributo
  - frozen — o valor do atributo não pode ser alterado (final no Java)
- Um exemplo:

```
- nome [0..1]: String="" {changeable}
```

Declaração de operações

- Sintaxe completa para a declaração de uma operação:

  `[visibilidade] nome[(parâmetros)][:tipo do resultado][{propriedades}]`

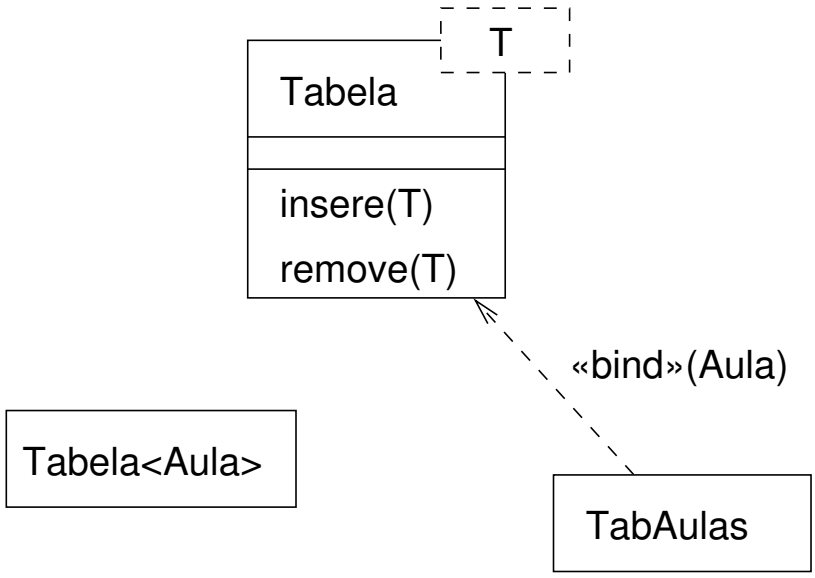
- Parâmetros:

  `[direcção] nome[:tipo][=valor por omissão]`

- Direcções: in (parâmetro de entrada); out (parâmetro de saída); inout (parâmetro de entrada e saída).
- Propriedades possíveis:
  - leaf — não pode ser redefinido por uma sub-classe
  - isQuery — não altera o estado do objecto
  - sequential — operação executada de forma sequencial (sem *threads*)
  - guarded — operação executada recorrendo a *threads*, mas apenas uma *thread* activa em cada momento;
  - concurrent — operação executada de forma concorrente.
- Um exemplo:

```
- nome [0..1]: String="" {changeable}
```

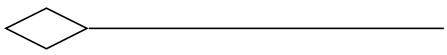
Classes parametrizadas

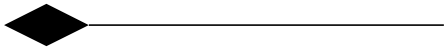


Relações — conceitos avançados

Agregação

- Relação todo-parte (uma associação *normal* é uma associação entre iguais).
- Importante quando se começa a refinar a estrutura do modelo.



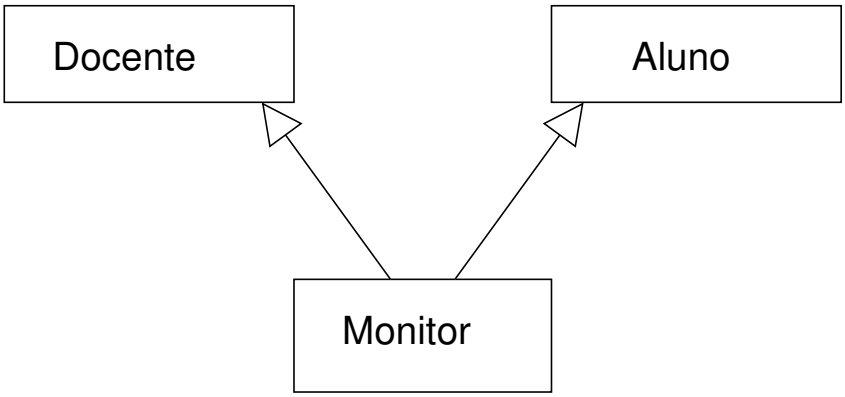


Composição

- Versão mais forte de agregação;
- Parte existe *dentro* do todo (se o todo for destruído a parte também o é).

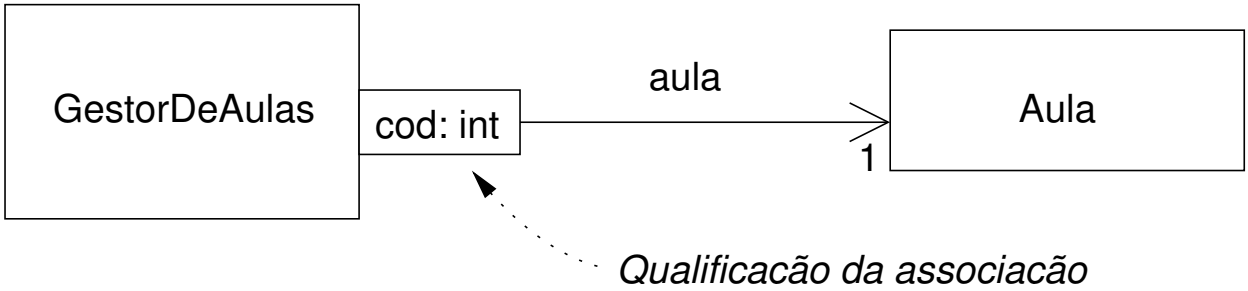
Herança múltipla (revisitada)

- Herança múltipla não existe no Java.



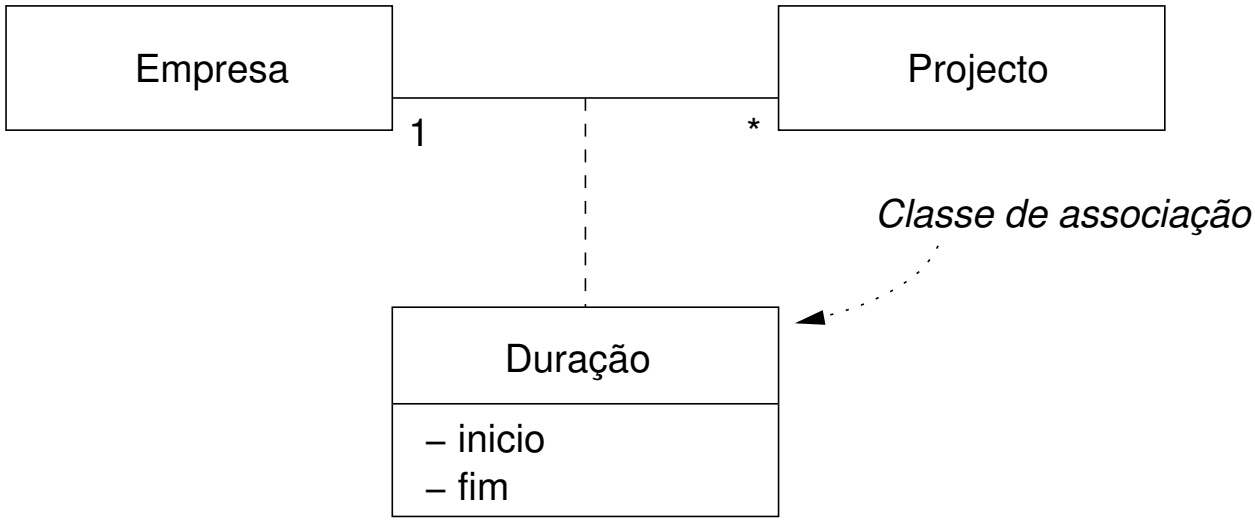
Qualificação de associações

- Em relações 1 para * é comum os valores serem acedidos através de uma chave.
- Isto pode ser representado em UML utilizando uma qualificação.
- A qualificação identifica a chave de uma associação.



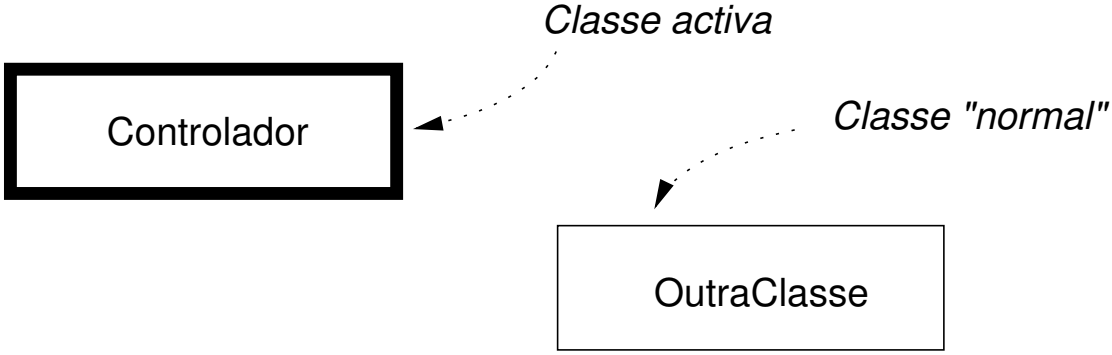
Classes de associação

- Permite adicionar atributos e operações às associações.
- Não directamente expressável em Java.



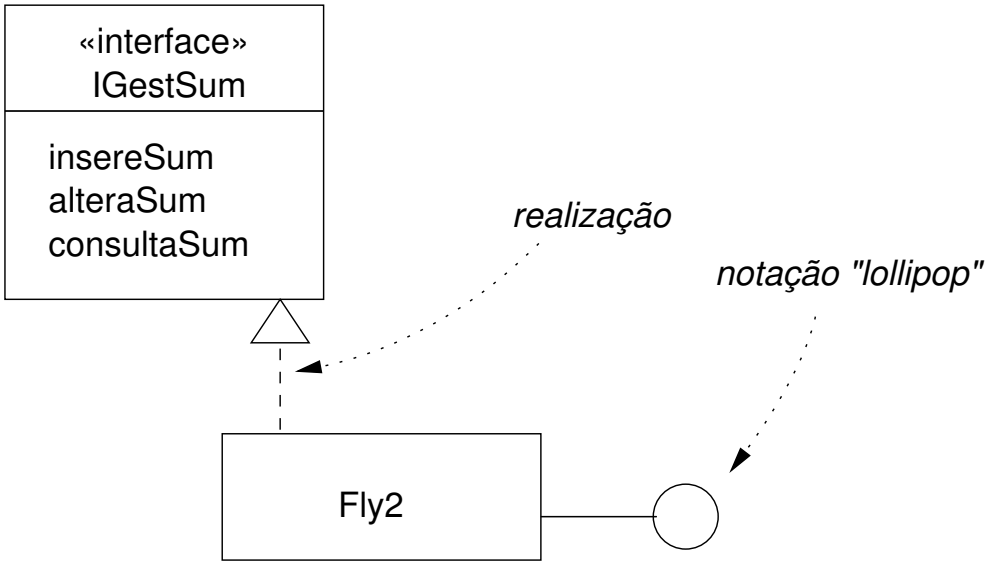
Classes activas

- Normalmente os objectos tem um comportamento reactivo (só apresentam actividade em resposta a mensagens recebidas).
- Classes activas permitem modelar objectos que possuem controlo de fluxo próprio (cf. *threads* em Java).
- Representam-se com uma linha mais espessa ao desenhar a classe.



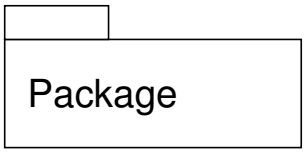
Interfaces e realização

- Um dos conceitos mais importantes tendo em vista o encapsulamento.
  - Acesso ao estado dos objectos faz-se apenas através dos métodos que este disponibiliza — noção de interface.
- Uma interface define um conjunto de operações (abstractas) — um serviço.
- Quando uma classe declara que realiza uma dada interface, compromete-se a implementar essas operações (caso contrário tem que obrigatóriamente ser abstracta).
- As dependências entre classes devem tanto quanto possível ser *isoladas* recorrendo a interfaces
- A modelação deve ser orientada às interfaces em vez de ser orientada às classes.
- No entanto é necessário perceber que a noção de interface é meramente sintáctica.
- Duas representações possíveis: estereótipos e notação *lollipop*:



Packages

- Permitem agrupar/organizar os diagramas/código.
- Permitem definir diferentes vistas do sistema:
  - diferentes versões dos sistemas
  - separar diferentes subsistemas
- Cada pacote define um contexto diferente:
- Notação:



- Estereótipos:
  - facade — apenas uma vista (parcial) de outro pacote (não contém elementos próprios)
  - framework — pacote consistindo essencialmente de padrões
  - stub — pacote substituto de um outro pacote contendo apenas *stubs* (útil para trabalho em equipa distribuido)
  - subsystem — pacote contendo parte do sistema
  - system — o sistema!