Introdução à Linguagem UML[1]

Ricardo R. Gudwin
DCA-FEEC-UNICAMP
30/09/2010

Introdução

A Linguagem UML (Unified Modeling Language) é uma linguagem de modelagem que foi criada visando-se a criação de **modelos abstratos de processos**. Em princípio, não existem restrições quanto aos tipos de processos sendo modelados. Tanto podem ser processos do mundo real como processos de desenvolvimento de software ou ainda detalhes internos do próprio software. Assim, tanto podemos utilizar o UML para descrever o mundo real - por exemplo - a organização interna de uma empresa, como os detalhes internos que descrevem um sistema de software. A descrição de um processo envolve a determinação de duas classes básicas de termos:

- Os elementos estruturais que compõem o processo
- O comportamento que esses elementos desenvolvem quando interagindo

Assim, o UML tanto pode ser utilizado para a análise dos elementos ontológicos participantes de um processo como do comportamento destes elementos no processo. (o adjetivo *ontológico* diz respeito a algo que é porque é, simplesmente, sem entrar em detalhes de porque é ou como é).

Particularmente no tocante à engenharia de software, a linguagem UML pode ser utilizada para modelar todas as etapas do processo de desenvolvimento de software, bem como produzir todos os artefatos de software necessários à documentação dessas etapas.

Na verdade, o UML é a convergência de diversas outras linguagens de modelagem utilizadas em diferentes processos de desenvolvimento de software propostos até então.

Apesar de ser uma linguagem formal (ou seja, é definida na forma de uma gramática - no caso a partir de uma meta-linguagem de descricão), o UML é uma linguagem visual, baseada em diferentes tipos de diagramas. Neste capítulo, teremos a oportunidade de conhecer de maneira sumarizada alguns dos diagramas utilizados na linguagem UML. Em capítulos posteriores, iremos entrar em detalhes dos diversos diagramas necessários para a construção de um software.

Uma das características interessantes do UML é a existência de **mecanismos de extensão**, que permitem que o UML, como linguagem, possa ser estendido, resultando a criação de novos tipos de diagramas. Os mecanismos de extensão do UML são os chamados **Profiles**. Diferentes Profiles podem ser construídos utilizando-se **estereótipos**, os ***tagged values*** e as **restrições**.

A linguagem UML, por meio de seus diagramas, permite a definição e design de threads e processos, que permitem o desenvolvimento de sistemas distribuídos ou de programação concorrente. Da mesma maneira, permite a utilização dos chamados **patterns** (patterns são, a grosso modo, soluções de programação que são reutilizadas devido ao seu bom desempenho) e a descrição de **colaborações** (esquemas de interação entre objetos que resultam em um comportamento do sistema).

Um dos tipos de diagramas particularmente úteis para modelarmos processos são os chamados diagramas de atividades. Por meio deles, é possível especificarmos uma sequência de procedimentos que compõem um processo no mundo real. Diagramas de atividade podem portanto ser utilizados para descrever o processo de desenvolvimento de software (por exemplo, o processo Unificado).

Uma outra característica do UML é a possibilidade de efetuarmos uma descrição hierárquica dos

1 Este texto visa apresentar de maneira sumarizada os principais conceitos do UML, para alunos de Engenharia de Software da UNICAMP.

processos. Com isso, é possível fazermos um refinamento de nosso modelo, descrevendo o relacionamento entre diferentes níveis de abstração do mesmo.

Para implementar esse refinamento, utilizamos o expediente de definir abstratamente **componentes** de um determinado modelo por meio de suas **interfaces**. Assim, esses componentes são definidos somente em função de suas entradas e saídas, deixando a definição dos *internals* do componente para um nível de abstração posterior. Essa caracterísitca permite ao UML um **desenvolvimento incremental** do modelo.

Por fim, a semântica dos diagramas UML é determinada por uma linguagem de restrição chamada de OCL (Object Contraint Language), que determina de maneira não-ambígua a interpretação a ser dada a seus diagramas.

Todas essas características fazem da linguagem UML uma linguagem de modelagem moderna, eficiente e conveniente para o desenvolvimento de especificações e definições com relação ao sistema em desenvolvimento.

História do UML

As linguagens de modelagem começaram a aparecer na engenharia de software no final dos anos 70, com o desenvolvimento dos primeiros sistemas orientados a objeto. A partir daí, foram sucessivamente empregadas em diversos experimentos em diferentes abordagens orientadas a objeto. Diversas técnicas influenciaram estas primeiras linguagens: **os modelos entidade/relacionamento**, o **SDL** (Specification and Description Language) e diversas outras. Entretanto, apesar de sua conveniência, o número de linguagens de modelagem passou de pouco mais de 10 para mais de 50 até 1994. Junto a este fato, começaram a proliferar diferentes métodos de desenvolvimento, detonando uma verdadeira "guerra de métodos". Alguns dos métodos mais famosos são por exemplo o método Booch, o OMT, o método Fusion, o OOSE e o OMT-2. Uma característica dessa guerra de métodos é que todos os métodos tinham linguagens de modelagem que eram muito similares conceitualmente, entretanto empregando uma notação diferente. Algumas tinham características que faltavam em outras, que por sua vez tinham outras características que estas não tinham. Estava clara a necessidade de algum tipo de padronização. O desenvolvimento do UML começou no final de 1994, quando Booch e Rumbaugh passaram a trabalhar em conjunto, envidando esforços para integrar suas metodologias.

Uma primeira versão preliminar do UML (versão 0.8) surgiu em outubro de 1995, quando Jacobson se uniu ao grupo. Vale a pena ressaltar quem são Booch, Rumbaugh e Jacobson. Estes indivíduos, conhecidos na vulgata como ***los tres amigos***, foram as principais lideranças no pensamento científico voltado para a engenharia de software orientada a objetos durante a "guerra de métodos. O esforço que fizeram por tentar um diálogo acabou por permitir o fim da guerra dos métodos e a criação do UML. De fato, a partir dos esforços conjuntos de Booch, Rumbaugh e Jacobson (que fundaram, mais ou menos por essa data a Rational Software - empresa que norteou o desenvolvimento do UML), acabou resultando no lançamento público da versão 0.91 do UML, em outubro de 1996 (as versões anteriores não eram públicas). A partir desse esforço, uma RFP (Request for Proposal) foi realizada pela OMG buscando contribuições da comunidade para o estabelecimento de uma linguagem unificada. Antes de prosseguirmos, vale a pena esclarecer o que é uma **RFP** e o que é a **OMG**.

Antes de mais nada a RFP. Esse termo, RFP está relacionado com um mecanismo de criação colaborativa de padrões que vem sendo utilizado na Internet para a criação de normas e procedimentos relacionados à Internet. Assim, quando um grupo de indivíduos verifica que algum tipo de protocolo - ou qualquer coisa - que esteja funcionando na Internet precisa ser padronizado, esse grupo lança um RFP - um *Request for Proposal* - uma requisição de propostas. Essa RFP é um documento onde o grupo sumariza uma proposta para o protocolo ou padrão que se deseja desenvolver, e requisita comentários da comunidade internet sobre a proposta, para que ela seja

avaliada de maneira independente por múltiplas fontes e possa evoluir de maneira natural e democrática. Comentários e sugestões são agregados à proposta original e esta evolui até se transformar em uma norma ou padrão.

Sobre a OMG - a OMG - ou *Object Management Group*, é uma organização sem fins lucrativos, composta por representantes de diversas empresas que trabalham no desenvolvimento de software orientado a objetos, que se dedica ao esforço de desenvolver normas relacionadas ao desenvolvimento de software orientado a objetos. Dentre outras normas, a OMG é responsável pelo padrão CORBA, e diversos outros padrões desenvolvidos de maneira colaborativa na Internet.

Voltando agora à história do UML - a partir das diversas contribuições que resultaram da RFP, lançou-se o UML 1.0. Dentre as empresas que enviaram sugestões estavam nomes de peso tais como a Digital Equipment Corp., a HP, a i-Logix, o IntelliCorp, a IBM, a ICON Computing, a MCI Systemhouse, a Microsoft, a Oracle, a Rational Software, a Texas Instruments e a Unisys. Em janeiro de 1997, novas contribuições lançaram o UML 1.1. Dentre as empresas que colaboraram desta vez estavam a IBM, a ObjecTime, a Platinum Technology, a Ptech, a Taskon & Reich Technologies e a Softeam.

A partir da versão 1.1, a comunidade de desenvolvimento de software começa a fazer uma aderência maciça ao UML. Em novembro de 1997, o UML foi adotado como norma pela OMG, que estabelecu um RTF (Revision Task Force) para aperfeiçoar pequenos detalhes na linguagem. Em junho de 1999 o RTF libera a versão UML 1.3 e em setembro de 2001 é lançado o UML 1.4. Em março de 2003, publica-se a versão 1.5, que combina os detalhes da versão 1.4 com uma semântica de ações. Em julho de 2004, a versão 1.4.2 é criada, sendo simultaneamente publicada como a norma ISO/IEC19501. Em julho de 2005 publica-se a versão 2.0 da norma, que traz diversas inovações (dentre outras, substituindo o diagrama de colaboração pelo diagrama de comunicações - uma versão aperfeiçoada do mesmo). A versão 2.1 do UML nunca foi publicada como uma especificação formal, mas em agosto de 2007 disponibilizou-se a versão 2.1.1 e em novembro de 2007 a versão 2.1.2. A versão 2.2 da norma foi publicada em fevereiro de 2009 e em maio de 2010 publicou-se a versão 2.3, que é a versão mais atual da norma até a data da elaboração deste documento [1,2].

Elementos de Modelagem

Diagramas podem ser entendidos, de maneira simplificada, como grafos contendo **nós**, **caminhos** entre os nós e **textos**. Esses nós podem ser símbolos gráficos (figuras bidimensionais) tão complexos e estruturados quanto possível. Os caminhos, podem ser linhas cheias ou pontilhadas com possíveis decorações especiais nas pontas. Por fim, os textos podem aparecer em diferentes posições, sendo que conforme a posição em que aparecem, podem ter um significado diferente. O que torna tal grafo um diagrama é o relacionamento visual que existe entre os diferentes elementos que o compõem. Este relacionamento na verdade espelha algum tipo de relacionamento entre os elementos envolvidos. Por exemplo, o relacionamento de **conexão** entre elementos do diagrama, modela algum tipo de relação entre os elementos conectados. Relacionamentos de conexão são usualmente expressos por meio de linhas ligando figuras bidimensionais. O relacionamento de **inclusão** (por exemplo, a inclusão de um símbolo dentro da área correspondente a outro símbolo), denota a inclusão de um determinado elemento em outro. Por fim, a **proximidade visual** entre símbolos (por exemplo, um símbolo estar perto de outro símbolo ou figura bidimensional dentro de um diagrama), denota algum tipo de relação entre esses símbolos.

Os elementos de um diagrama podem ser: **ícones**, **símbolos bidimensionais**, **caminhos** e **strings**. Os **ícones** são figuras gráficas de tamanho e formato fixo que não podem ser expandidas para ter algum tipo de conteúdo. Podem aparecer como participantes de um símbolo bidimensional, como terminadores de caminhos ou simplesmente isoladamente. Os **símbolos bidimensionais** são figuras bidimensionais que podem ter tamanho variável e que podem ser expandidos de modo a conter

outros elementos, tais como listas de strings ou outros símbolos bidimensionais. Podem ser divididos em compartimentos de tipo similar ou diferente. Os **caminhos** são sequências de linhas cujas extremidades estão conectadas a outros elementos. Caminhos podem ter terminadores. Por fim, os **strings** são textos apresentando diversas informações cujo significado depende de onde aparecem.

Estereótipos

Estereótipos são tipos especiais de Strings que servem para modificar a semântica de elementos de um diagrama. Assim, os estereótipos permitem que se definam novos elementos de modelagem em termos de elementos já conhecidos. Em princípio, estereótipos podem ser aplicados a qualquer elemento de modelagem. A notação de um estereótipo é na forma «estereótipo». Com o uso de estereótipos, novos tipos de diagramas podem ser criados dentro da linguagem UML. Veja alguns exemplos de estereótipos na figura 1 abaixo.

Nestes casos, as classes "Janela Principal", "Controle Janela", "Dados Iniciais" e "Usuário" são marcadas pelos estereótipos boundary, control, entity e actor. Essas classes, ditas classes estereotipadas, alternativamente podem ser representadas por símbolos gráficos diferenciados para representar esses elementos modificados. Observe que estereótipos podem ser aplicados a qualquer elemento de modelagem, não somente classes. Com isso, relacionamentos, componentes, etc… podem ser estereotipados também. Relacionamentos estereotipados podem adquirir uma notação diferente, tais como linhas tracejadas, etc, ou ainda ter simplesmente seu estereótipo declarado na forma de texto.

Observe que cada elemento UML pode ter no máximo um único estereótipo. Assim, não é possível aplicarmos estereótipos a elementos que já sejam estereotipados. Alguns tipos de elementos contém um conjunto de estereótipos previamente definidos. Por exemplo, classes podem assumir estereótipos do tipo «actor», ou associações entre casos de uso podem assumir os estereótipos «extends» ou «include». O uso de estereótipos pode ser utilizado para modificar o modo de geração de código. Assim, pode-se usar um ícone diferente para modelar tipos de entidades em etapas específicas do processo de modelagem, tal como faremos por exemplo na fase de análise. Estereótipos podem ser utilizados em qualquer caso em que uma extensão é necessária para modelar alguma característica particular que os elementos padrões não evidenciam por si só. Na figura 2 acima, temos exemplos de diferentes visualizações de estereótipos.

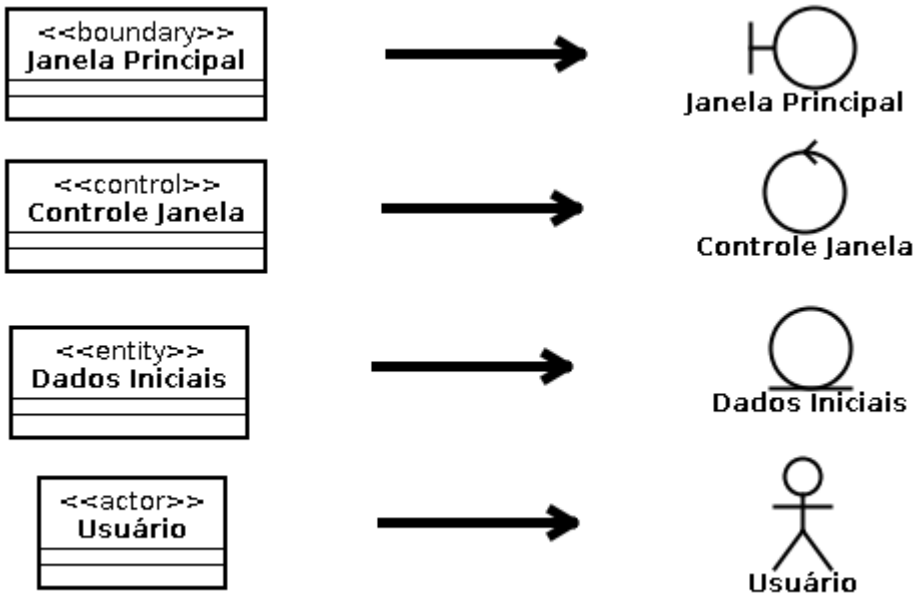

Figura 1: Exemplos de Estereótipos

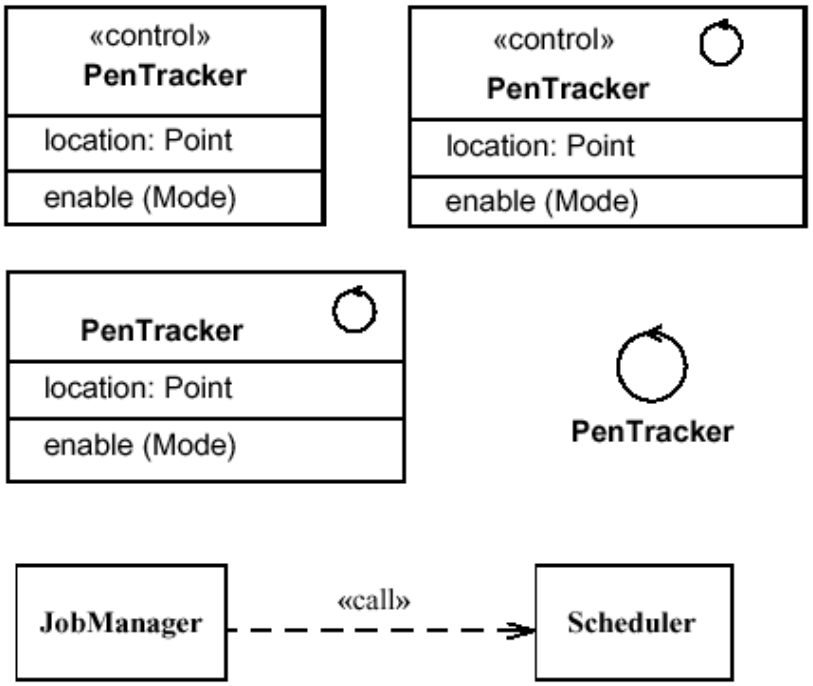

Figura 2: Exemplos de Diferentes Visualizações para Estereótipos

Notas

Notas são descrições de texto que complementam a informação contida em algum diagrama. As notas devem estar ancoradas a um elemento por meio de uma linha pontilhada. Um exemplo é dado na figura 3.

Pacotes

Pacotes são agrupamentos de elementos de modelagem em um único elemento, permitindo uma descrição do modelo em múltiplos níveis. Pacotes podem ser aninhados assim dentro de outros pacotes, de maneira recursiva, da mesma maneira que diretórios de arquivos em sistemas computacionais. Em princípio, qualquer elemento UML pode ser agrupado em um pacote. Pacotes podem também se relacionar com outros pacotes. Dentre os possíveis relacionamentos, os mais comuns são os relacionamentos de dependência e de generalização. Exemplos de pacotes podem ser vistos na figura 4 .

Sub-Sistemas

Sub-sistemas são um tipo de pacote específico, denotados pelo estereótipo «subsystem». Na verdade, subsistemas representam uma unidade comportamental em um sistema físico, ou seja, unidades que funcionam de maneira coesa e que podem ser vistas externamente como uma única entidade. Essa unidade pode oferecer interfaces e ter operações. Uma característica dos subsistemas é que seu conteúdo pode ser particionado em elementos de especificação e realização.

A especificação de um sub-sistema consiste na definição de operações sobre o sub-sistema, ao mesmo tempo que se definem outros elementos de especificação associado, tais como casos de uso ou máquinas de estado.

Subsistemas podem ou não instanciáveis. Sub-sistemas não instaciáveis servem meramente como unidades de especificação para o comportamento dos elementos nele contidos.

Um subsistema pode possuir compartimentos, que permitem a distribuição dos elementos que o

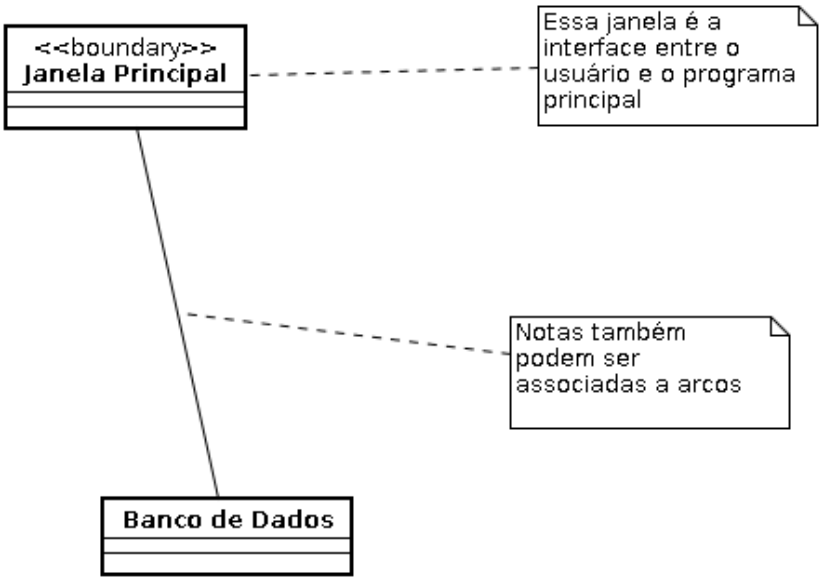

Figura 3: Exemplo de Nota

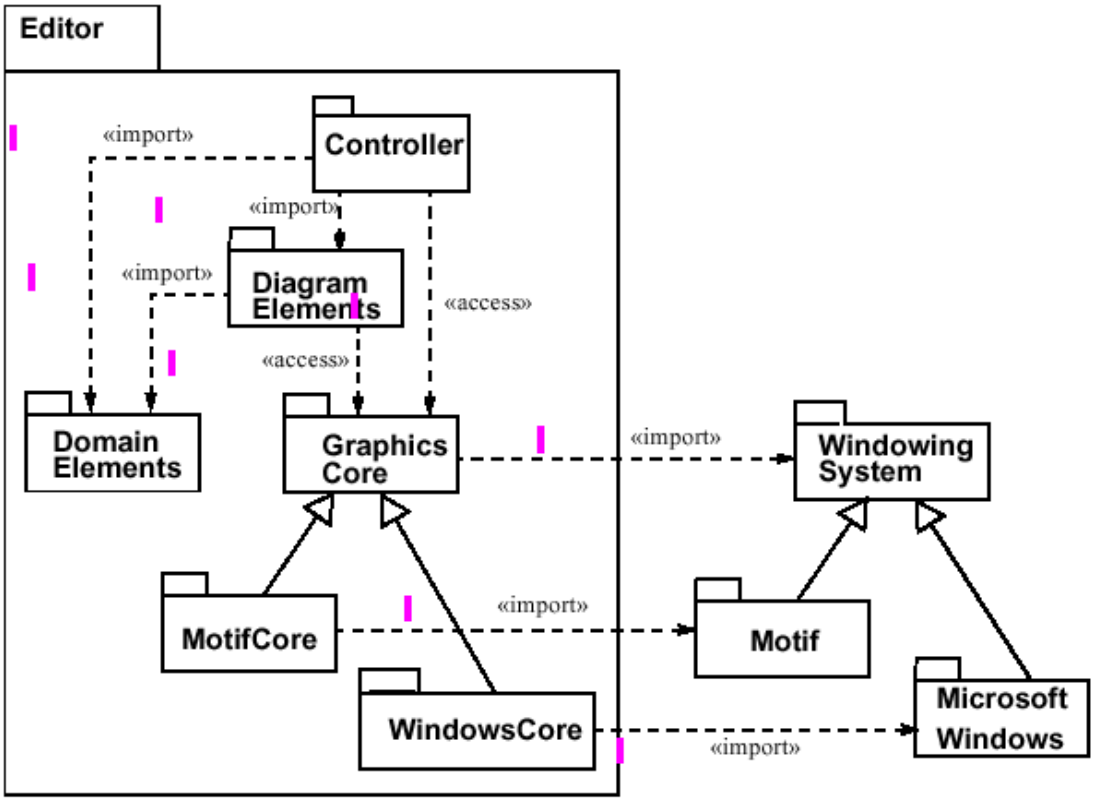

Figura 4: Exemplo de Pacotes

compõem em espaços reservados, tais como na figura 5.

Podem também possuir interfaces, que permitem especificar um conjunto de operações para o acesso ao sub-sistema, como na figura 6.

A norma UML permite diversas diferentes notações para subsistemas. Exemplos dessas notações estão nas figuras 7, 8 e 9 acima.

É possível fazer-se um mapeamento entre as partes de especificação e realização de um sub-sistema,

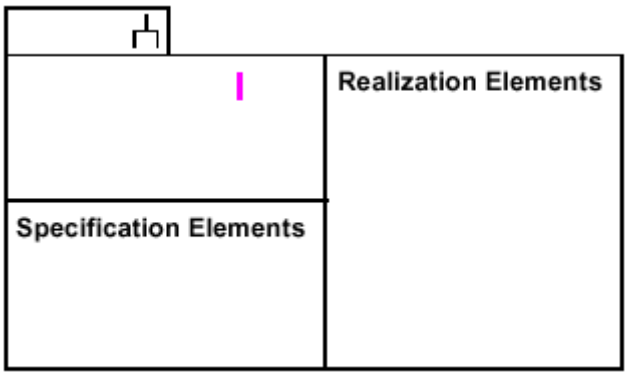

Figura 5: Exemplo de Subsistema com Compartimentos

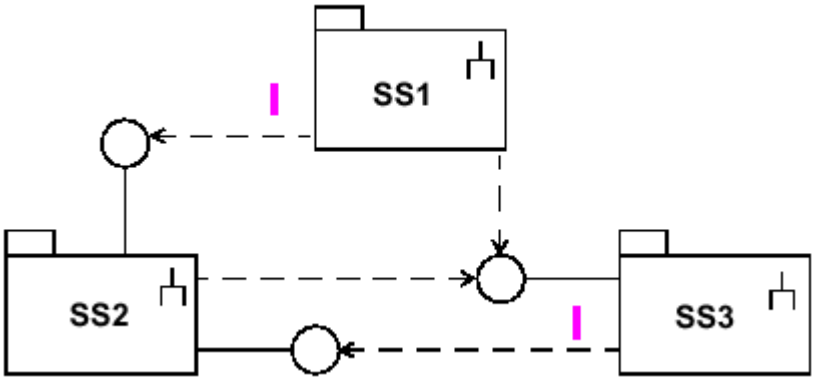

Figura 6: Exemplo de Subsistemas com Interfaces

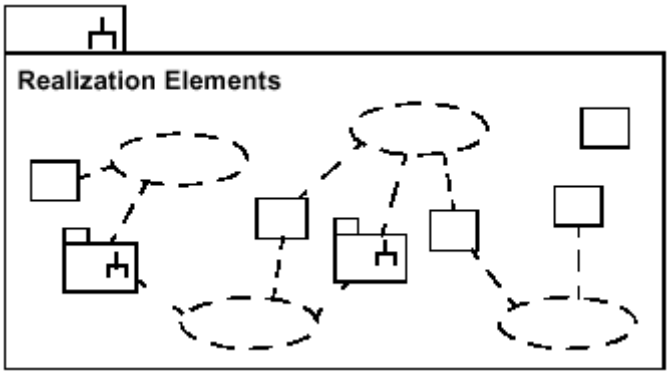

Figura 7: Exemplo de Diferentes notações para Subsistemas

usando os três compartimentos de um sub-sistema, conforme a figura 10.

Assim, os elementos de realização estão mapeados nos elementos de especificação e nas operações declaradas do sub-sistema. Da mesma forma, é possível fazermos um mapeamento do sub-sistema em uma interface. Entretanto, somente os elementos de realização com relevância são declarados, e esse mapeamento pode ser expresso de diferentes maneiras.

Restrições (*Constraints*)

Restrições são relacionamentos semânticos entre elementos de modelagem que especificam condições e proposições que necessitam ser mantidas. Algumas delas são pré-definidas, mas de maneira geral restrições também podem ser definidas pelo usuário. A figura 11 a seguir apresenta alguns exemplos de restrições:

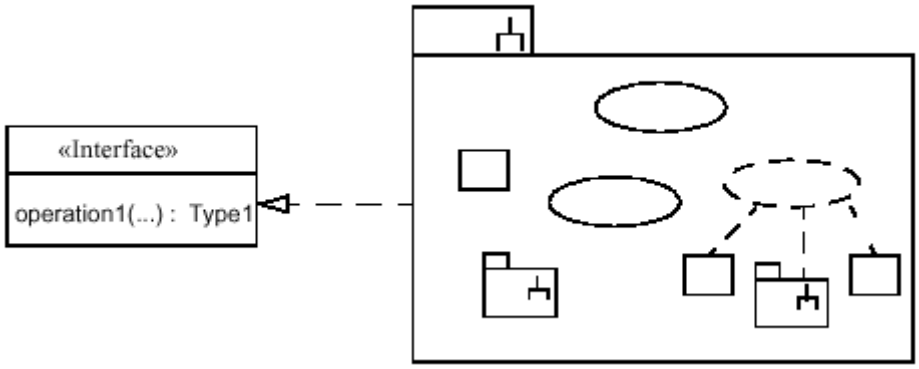

Figura 8: Exemplo de Notação de Subsistema

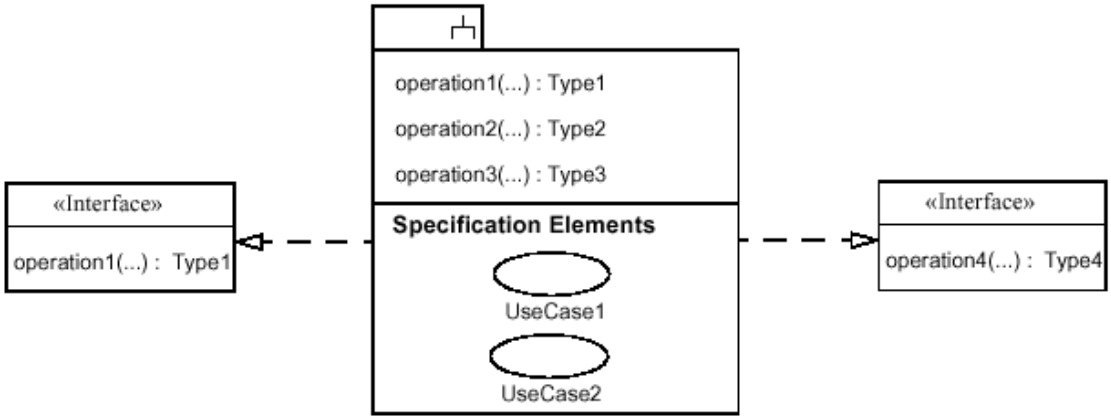

Figura 9: Exemplo de Subsistema com 2 Compartimentos

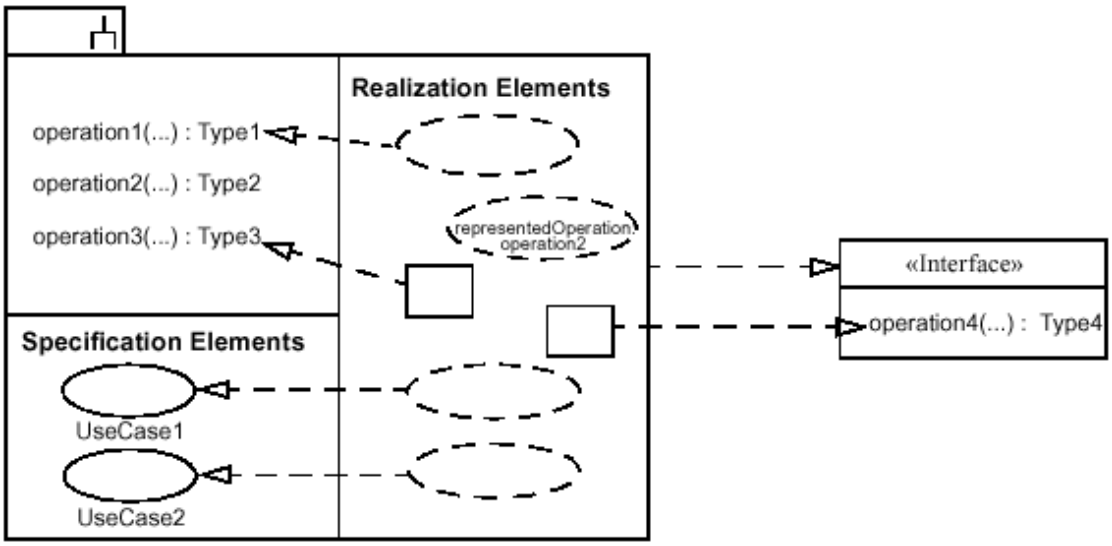

Figura 10: Exemplo de Mapeamento entre Elementos de Subsistemas

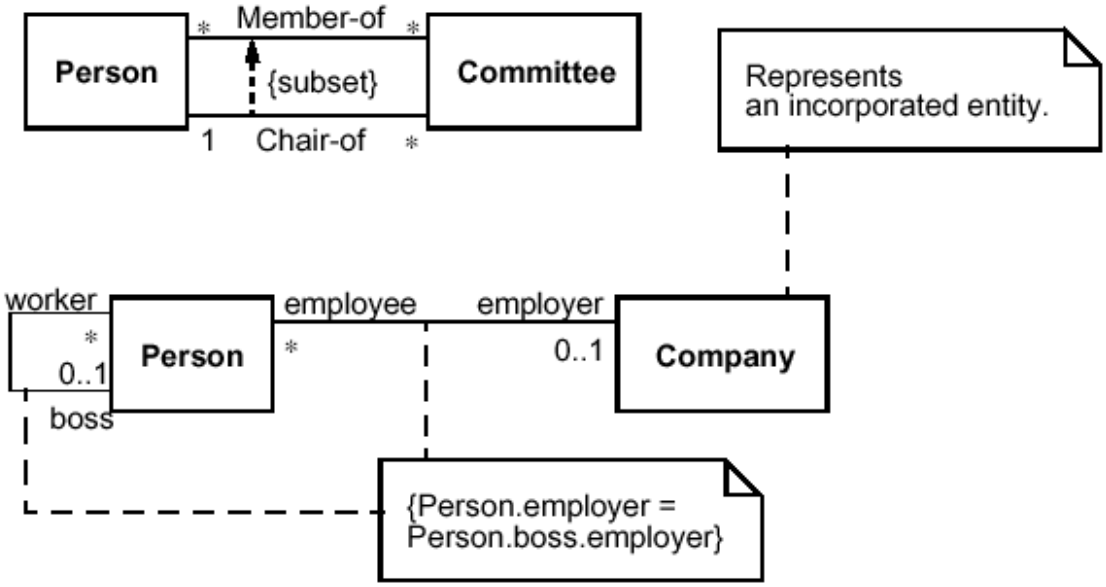

Figura 11: Exemplo de Restrições (Constraints)

Os Diagramas UML

Os diagramas UML podem, de uma maneira geral, se servir de todos estes expedientes de modelagem: estereótipos, tagged values, notas, pacotes, sub-sistemas e restrições. Detalhes dos diferentes diagramas serão introduzidos mais a frente, quando de sua utilização nas diversas etapas de modelagem. Uma visão geral do "jeitão" que marca cada tipo de diagrama poderá ser obtido a partir das atividades propostas a seguir. Os diagramas em questão são os seguintes:

- Diagramas Estruturais
    - Diagramas Estruturais Estáticos (Diagramas de Classes)
    - Diagramas de Componentes
    - Diagramas de Distribuição (*Deployment*)
- Diagramas Comportamentais
    - Diagramas de Casos de Uso
    - Diagramas de Sequência
    - Diagramas de Comunicação (Colaboração)
    - Diagramas de Estado

Diagramas Estruturais Estáticos

Diagramas estruturais estáticos denotam a estrutura estática de um modelo em particular. Em diagramas deste tipo, representamos:

- Coisas que existem (tais como classes, tipos e objetos)
- A Estrutura Interna Dessas Coisas
- Relacionamentos entre essas coisas

Com isso, diagramas estruturais estáticos não mostram informações temporais relacionadas com os conceitos representados, somente informações estruturais. Existem basicamente dois tipos fundamentais de diagramas estruturais estáticos:

- Diagramas de Classes
- Diagramas de Objetos

Diagramas de classes mostram conjuntos de classes e tipos relacionados entre si. Podem ainda representar templates e classes instanciadas (objetos), alémde relacionamentos (associações e

generalizações), bem como o conteúdo destas classes (atributos e operações)

Diagramas de objetos mostram instâncias compatíveis com algum diagrama de classes em particular e o relacionamento estrutural entre elas. A diferença básica que existe entre um diagrama de classes e um diagrama de objetos é que em diagramas de objetos preponderem as instâncias de classes, ao passo que em diagramas de classes preponderam as classes em si. Fundamentalmente, em termos formais ambos os diagramas são semelhantes. Uma diferença prática é que em diagramas de objetos podem haver diversas instâncias de uma mesma classe, pois o intuito é representar as instâncias, ao passo que em diagramas de classes o intuito é representar a organização das classes.

Diagramas de Classes

Formalmente falando, diagramas de classes são grafos de elementos do tipo *Classifier* conectados por diversos tipos de relacionamentos estáticos. Podem ainda conter pacotes e outros tipos de elementos gerais. Em princípio, um diagrama de classes representa uma visão do modelo estrutural estático, que pode ser entendidio como a união de todos os diagramas de classe e de objetos, da mesma maneira que podemos projetar uma figura tridimensional em diversos planos bi-dimensionais.

O tipo *Classifier* pode constituir-se de *Classes*, *Interfaces* e *DataTypes*. *Classes* são descritores para um conjunto de objetos com estrutura, comportamento e relacionamentos similares. Seu modelo diz respeito à **intensão** de uma classe, ou seja, as regras que a definem enquanto uma classe.

Exemplos de representações de classes podem ser vistas na figura 12.

Como se pode depreender dos diferentes exemplos, o símbolo gráfico utilizado para representar uma classe é uma caixa, possivelmente dividida em compartimentos. Esses compartimentos são utilizados em diferentes situações, dependendo se a classe pertence a um modelo de análise, design ou implementação. Tais compartimentos possuem nomes default, mas podem ser nomeados também caso seja necessário.

O primeiro compartimento, de cima para baixo é chamado de compartimento do nome, contendo o nome da classe em questão e opcionalmente um estereótipo, um conjunto de propriedades (tagged-values) ou um ícone referente ao estereótipo.

Os compartimentos seguintes são chamados de compartimentos de listas, podendo acomodar listas de atributos, operações ou outros.

Compartimento de Atributos

O compartimento de atributos (normalmente o primeiro compartimento depois do compartimento do nome), é utilizado para mostrar os atributos de uma classe. A sintaxe padrão para a descrição dos atributos nesse compartimento é a seguinte:

```
visibility name [ multiplicity ] : type-expression = initial-value { property-string }
```

A visibilidade corresponde a um flag (+, # ou -) correspondendo a:

- + público
- # protegido
- - privado

A multiplicidade é usada para representar arrays (por exemplo [3] ou [2..*] ou ainda [0..7].

Os atributos de classe (também chamados de atributos estáticos) devem aparecer sublinhados.

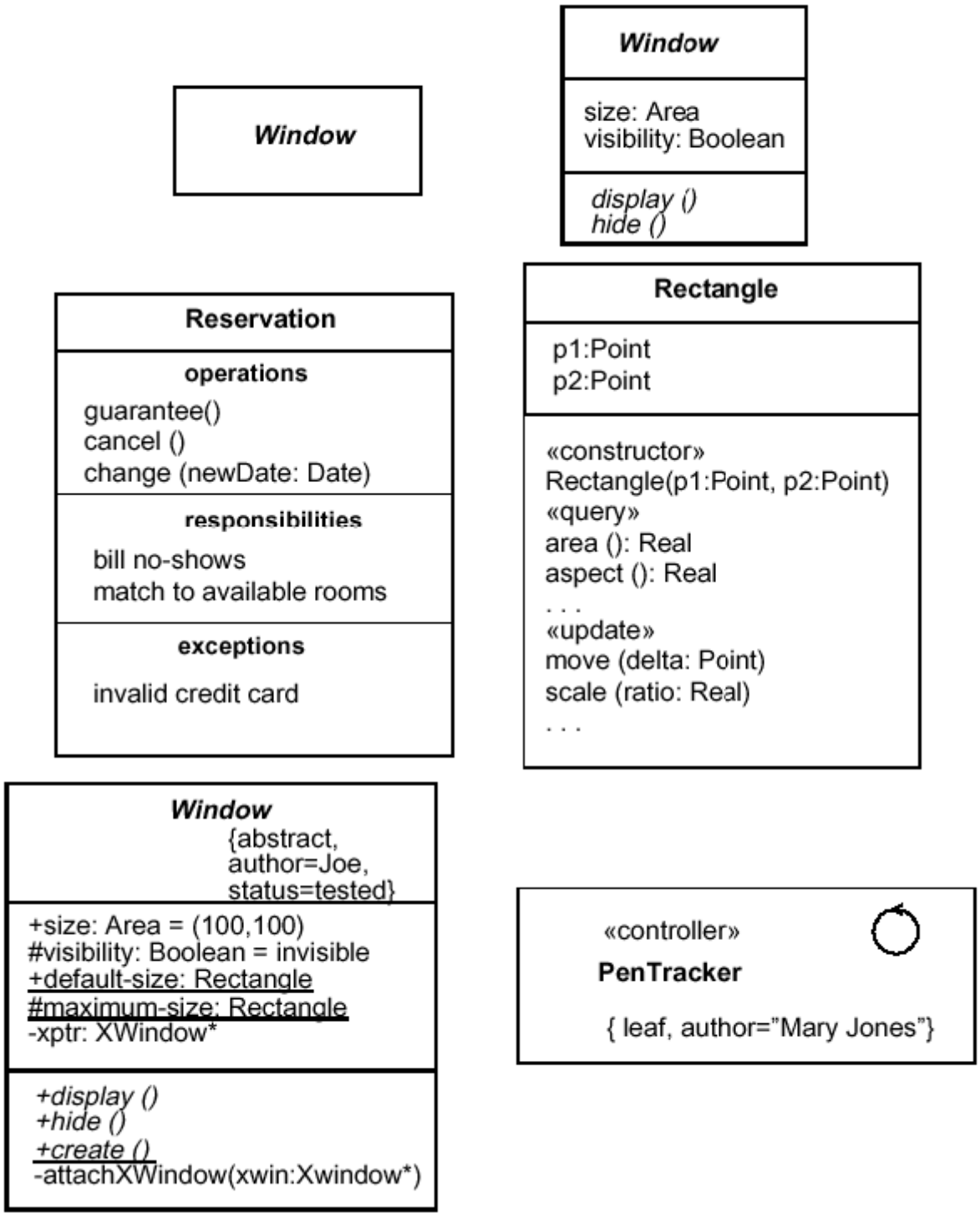

Figura 12: Exemplos de Diferentes Notações de Classes

Compartimento de Operações

O compartimento de operações mostram as operações definidas para uma classe e/ou os métodos supridos por uma classe. Sua sintaxe padrão é a seguinte:

```
visibility name ( parameter-list ) : return-type-expression { property-string }
```

onde visibilidade é semelhante ao usado para os atributos.

Cada elemento de parameter-list tem a seguinte sintaxe:

`kind name : type-expression = default-value`

onde **`kind`** deve ser **`in`**, **`out`**, ou **`inout`**

O UML define algumas propriedades como default. Exemplos de propriedades default incluem:

- **`{query}`** a operação não modifica o estado do sistema
- **`{concurrency = name}`**, onde **`name`** deve ser um dos nomes: **`sequential`**, **`guarded`**, **`concurrent`**
- **`{abstract}`** especifica operações não dotadas de implementação

Classes Conceituais e Classes de Implementação

Classes podem ser de dois tipos básicos: Classes Conceituais ou Classes de Implementação.

Classes Conceituais podem não incluir métodos, mas simplesmente prover especificações comportamentais para sua operação. Podem ter atributos e associações.

Classes de Implementação definem uma estrutura de dados física (para atributos e associações) e métodos de um objeto como implementados em linguagens tradicionais (C++, Java, etc). Normalmente dizemos que uma classe de implementação **realiza** uma classe conceitual, se ela provê todas as operações especificadas em uma classe conceitual. Dessa forma, é pssível que uma única classe de implementação realize diversas classes conceituais.

Veja o exemplo da figura 13. Nesse exemplo, as classes da esquerda são classes conceituais (devido à presença do estereótipo «type»).As classes da direita são classes de implementação (devido à presença do estereótipo «implementationClass»).

Interfaces

Interfaces são especificadores para um conjunto de operações externamente visíveis de uma classe, componente ou outro tipo de classifier (incluindo um sub-sistema) sem a especificação de sua estrutura interna. Desta forma, cada interface especifica somente uma parte limitada do comportamento de uma classe. Outra característica é que interfaces não possuem implementação. Da mesma forma, não possuem atributos, estados ou associações, mas somente operações. Interfaces podem ter relacionamentos do tipo generalização. Formalmente, são equivalentes a uma classe abstrata sem atributos e sem métodos, com somente operações abstratas.

A notação para interfaces no UML é a de uma classe sem compartimento de atributos, com o estereótipo «interface» ou simplesmente um círculo (visão estereotipada da interface). Uma dependência abstrata entre uma classe e uma interface pode ser representada por uma linha do tipo generalização tracejada ou por uma linha cheia ligada a um círculo representando a interface, conforme na figura 14.

Observe na figura que a interface Comparable está representada na parte inferior na forma de uma classe estereotipada e um pouco mais acima na forma de um círculo. A classe String, a esquerda, implementa a interface Comparable. Essa implementação está representada de duas maneiras na figura: primeiro por meio da generalização tracejada abaixo, e segundo por meio da associação (linha cheia) entre a classe String e a representação estereotipada da interface Comparable (na forma de um círculo, mais acima).

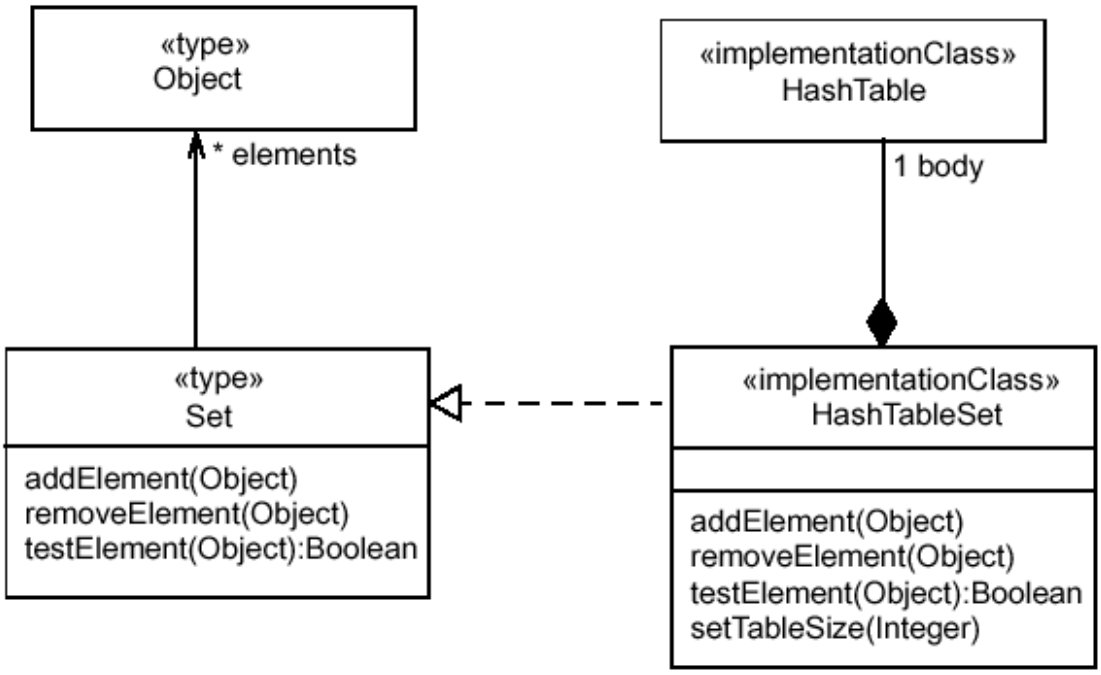

Figura 13: Classes Conceituais e Classes de Implementação

Classes e Pacotes

Cada diagrama de classes corresponde a um pacote na notação UML. Da mesma forma, mais de um pacote podem aparecer no mesmo diagrama. Em algumas situações, pode ser necessário referenciarmos classes em outros pacotes não disponíveis no diagrama corrente. Neste caso, a classe deve ser referenciada da seguinte forma:

```
Package-name :: class-name
```

Um exemplo desse uso está apresentado na figura a seguir:

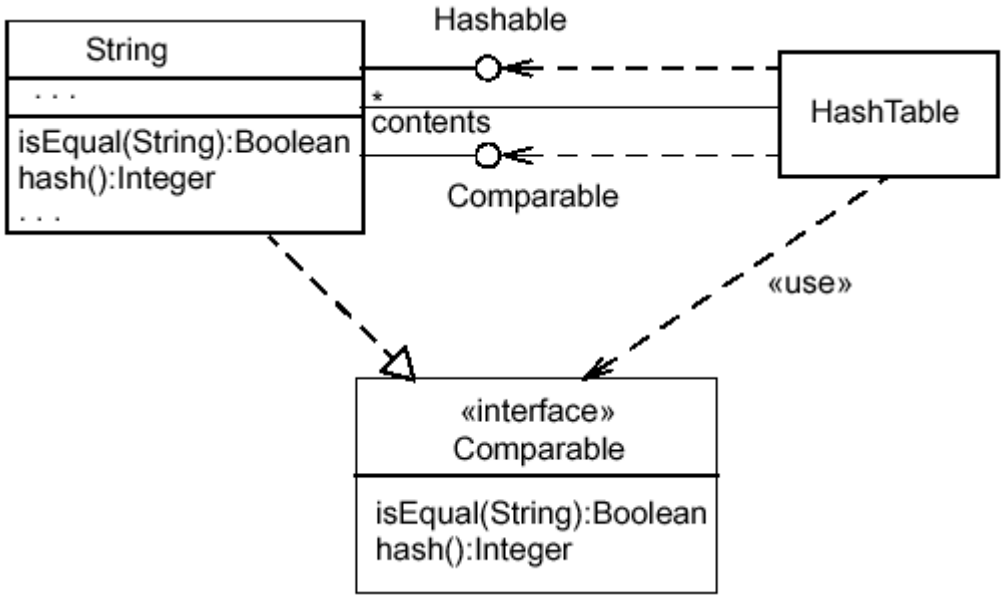

Figura 14: Exemplos de Interfaces

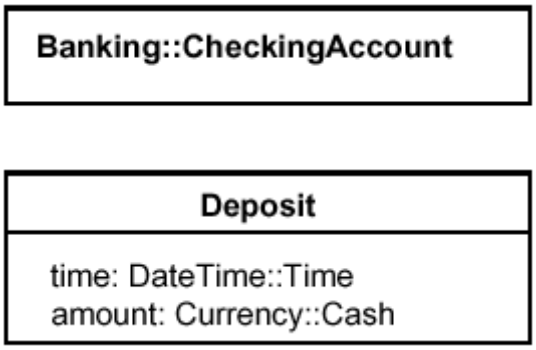

Figura 15: Exemplo de Classes Oriundas de Outros Pacotes

Da mesma maneira um elemento pode referenciar elementos em outros pacotes. Com isso, torna-se possível criar-se diversos níveis internos de pacotes. Quando se deseja referenciar um tipo de dependência entre pacotes, pode-se utilizar o estereótipo «access» sobre a dependência para indicar que o conteúdo de um pacote faz referência a elementos do outro pacote. A única restrição é que a visibilidade entre os pacotes seja condizente. Dependências estereotipadas com «import», ao contrário, liberam o acesso e automaticamente carregam os nomes com a visibilidade apropriada no espaço de nomes do pacote, fazendo a importação, o que que evita o uso de nomes de caminhos para referenciar as classes. Um exemplo de dependência estereotipada é mostrada na figura 16 a seguir:

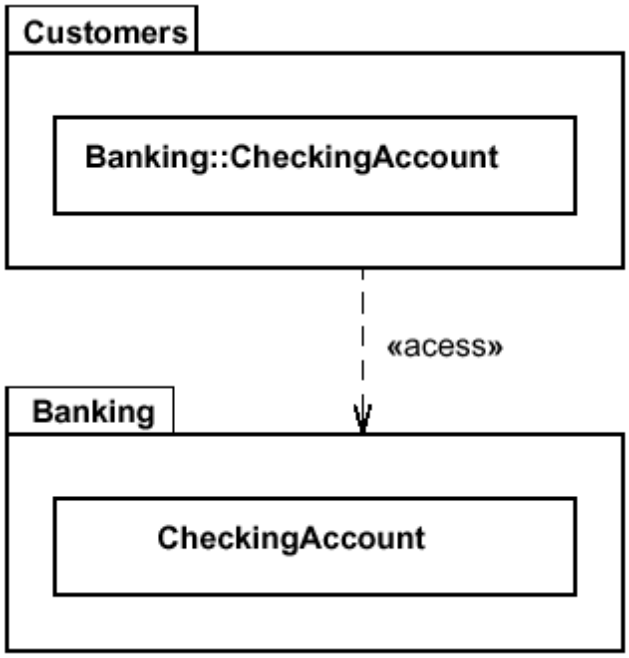

Figura 16: Exemplo de Dependência Estereotipada

Diagramas de Objetos

Diagramas de objetos são grafos de instâncias de classes, incluindo objetos e variáveis. Podemos dizer que sob certo aspecto um diagrama de classes não deixa de ser uma instância de um diagrama de classes mostrando detalhadamente o estado do sistema em um determinado instante de tempo. O formato do diagrama de objetos, como já dissemos, é semelhante ao do diagrama de classes, sendo diferenciado somente por seu uso. O uso de diagramas de objetos é normalmente limitado, sendo utilizado para mostrar exemplos de estruturas de dados em pontos estratégicos do sistema. Cada objeto em um diagrama de objetos representa uma instância particular de uma classe, tendo uma

identidade e um conjunto de valores de atributos que lhe são próprios.

A notação para objetos no UML é derivada da notação de classe, com a ressalva que o nome do objeto/classe aparece sublinhado. Assim, quando o compartimento de nome de uma classe aparece sublinhado, não se trata de uma classe, mas sim de uma instância desta, ou seja, de um objeto. A caixa de objetos pode ter dois compartimentos: o primeiro mostra o nome do objeto, seu estereótipo e propriedades:

`objectname : classname`

onde **`classname`** pode incluir o caminho completo do pacote onde se encontra a classe que o objeto referencia, por exemplo:

`display_window: WindowingSystem::GraphicWindows::Window`

ou seja, o objeto **`display_window`** é um objeto da classe **`Window`**, que fica no pacote **`GraphicWindows`** que fica no pacote **`WindowingSystem`**.

Caso haja herança múltipla, as classes devem ser separadas por vírgulas.

O segundo compartimento de um objeto (caso ele exista), pode mostrar os valores específicos dos atributos do objeto, na forma de uma lista, onde cada linha deve ser como segue:

`attributename : type = value`

Objetos compostos são representações de objetos de alto nível, ou seja, que contém outros objetos como partes. Na figura 17 a seguir, vemos diversos exemplos de objetos:

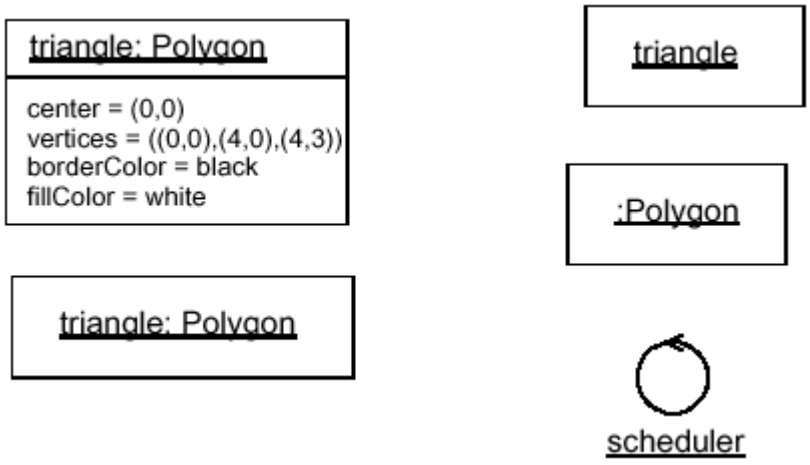

Figura 17: Exemplos de Objetos

A figura 18 a seguir mostra um exemplo de um objeto composto:

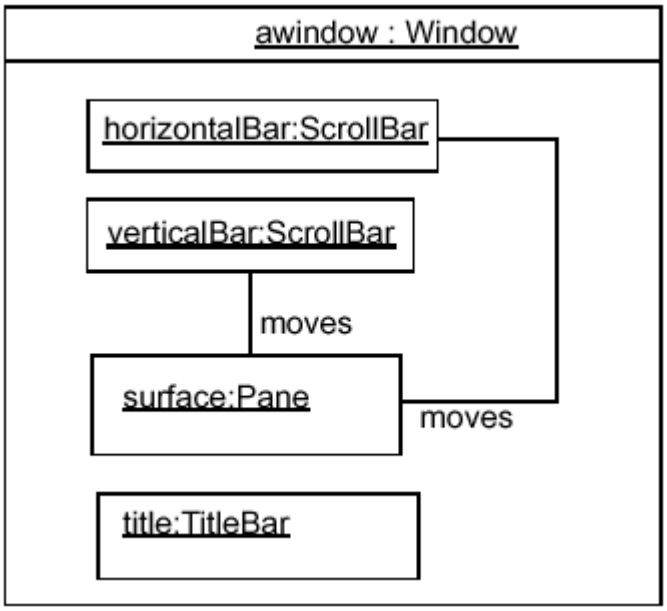

Figura 18: Exemplo de Objeto Composto

Relacionamentos entre Classes e Objetos

Classes e objetos podem estar conectados por algum tipo de relacionamento. A linguagem UML prevê três tipos básicos de relacionamentos:

- Associações
- Generalizações
- Dependências

As associações podem ainda ser sub-divididas em três sub-tipos básicos:

- Associações Simples
- Agregações
- Composições

Os relacionamentos podem ser binários, ternários ou de ordem superior. Relacionamentos binários são mostrados como linhas conectando dois símbolos do tipo *classifier*. Essas linhas podem ter uma variedade de decorações para diferenciar suas propriedades. Relacionamentos ternários ou de ordem superior podem ser exibidos como diamantes conectados por linhas a símbolos de *classifers*.

Associações Simples

Associações simples representam que existe alguma conexão entre dois elementos do tipo *classifier*, de tal forma que um deve manter alguma referência ao outro. Associações simples são representadas na forma de uma linha cheia conectando os dois *classifier*. Uma associação simples pode possuir ainda um nome e duas extremidades. O nome da associação é grafado na forma de um *String*, posicionado normalmente próximo ao centro da linha que representa a associação. As extremidades podem possuir ainda uma **navegabilidade**, uma **multiplicidade** e um **papel**. Observe a figura 19 a seguir:

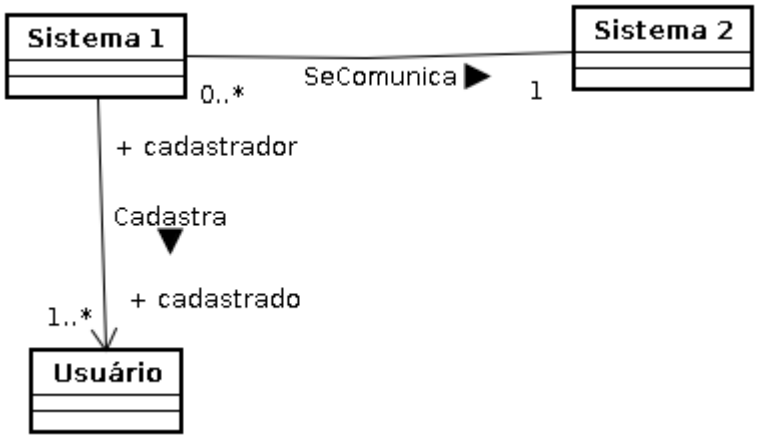

Figura 19: Exemplo de Associação Simples

A classe *Sistema1* está associada à classe *Sistema2* por meio de uma associação com nome *SeComunica*. Essa associação representa que um objeto do tipo *Sistema1* pode se comunicar com apenas 1 único (vide a multiplicidade na extremidade próxima a *Sistema2*) objeto do tipo *Sistema2*. Entretanto, um objeto de tipo *Sistema2* pode se comunicar com 0 ou mais objetos do tipo *Sistema1*. Nesse caso, como não existe nenhum tipo de decoração na ponta das linhas de *SeComunica*, dizemos que essa associação é bi-direcional. Na outra associação apresentada, entre *Sistema1* e *Usuário*, temos um exemplo de uma associação uni-direcional. Isso ocorre pois existe uma decoração chamada de navegabilidade na extremidade de *Cadastra* que se conecta a *Usuário*. A leitura da multiplicidade se faz da seguinte maneira: um objeto do tipo *Sistema1* pode cadastrar 1 ou mais objetos do tipo *Usuário*. Como a associação é unidirecional, não há associação na direção de *Usuário* a *Sistema1*. Observe ainda o símbolo > após o nome da associação (*Cadastra*). Esse símbolo indica a leitura que se deve fazer. Assim, é o *Sistema1* quem cadastra o *Usuário*, e não vice-versa. A figura apresenta ainda os papéis associados às extremidades da associação *Cadastra*. Nesse caso, o *Sistema1* é o cadastrador (papel público, pois aparece o +), e o *Usuário* é o cadastrado (também público).

Podem existir associações do tipo ou-exclusivo, chamadas também de associações XOR. Associações desse tipo indicam uma situação onde somente uma dentre diversas potenciais associações podem ser instanciadas em um determinado instante, para uma dada instância. Qualquer instância do classificador poderá participar de somente uma das associações indicadas. Este é um exemplo da aplicação de uma restrição a uma associação. Um exemplo de uma associação XOR é apresentado na figura 20 a seguir:

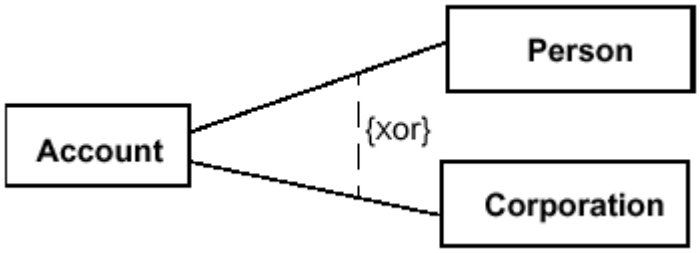

Figura 20: Exemplo de Associação do tipo XOR

Qualificadores

Qualificadores são atributos ou listas de atributos cujos valores servem para particionar o conjunto de instâncias associadas a uma instância através de uma associação. Assim, podemos entender que qualificadores são atributos de uma associação. Exemplo do uso de qualificadores são mostrados na figura 21 a seguir:

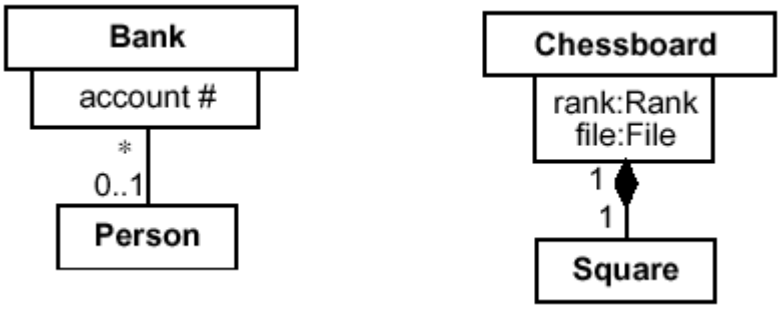

Figura 21: Exemplo de Qualificadores

Agregações e Composições

Agregações são um tipo especial de associação onde o elemento associado corresponde a uma parte do elemento principal. Composições são um tipo especial de agregação onde necessariamente a parte indicada deve existir. Um exemplo contendo uma agregação e uma composição é mostrado na figura 22 a seguir:

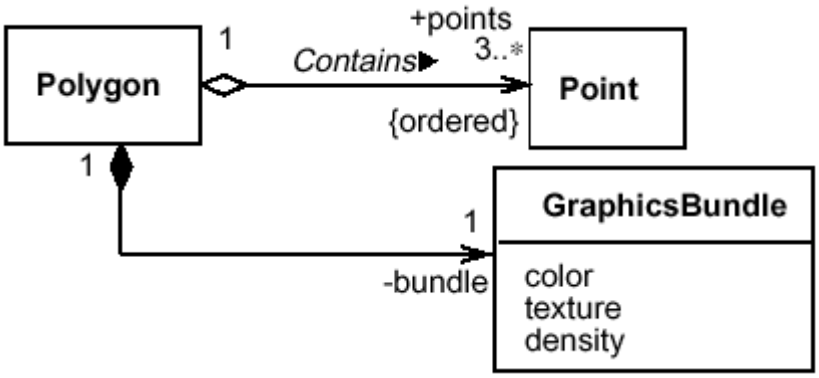

Figura 22: Exemplo de Agregação e Composição

Um objeto da classe *Polygon* pode conter diversos objetos da classe *Point*, entretanto terá somente um único objeto da classe *GraphicsBundle*. A diferença básica entre uma agregação e uma composição é que na agregação, o número de partes associadas à classe principal é variável e pouco importa. No caso de uma composição, esse número é definido, de tal forma que não faz sentido pensarmos o objeto da classe principal sem os objetos que o compõem. O melhor exemplo para uma composição é a idéia de uma *Mão*, que é formada pela composição de 5 objetos da classe *Dedo*.

Existem diversas maneiras de representar uma composição. A maneira da figura acima é uma delas. Outras maneiras são apresentadas na figura 23 abaixo.

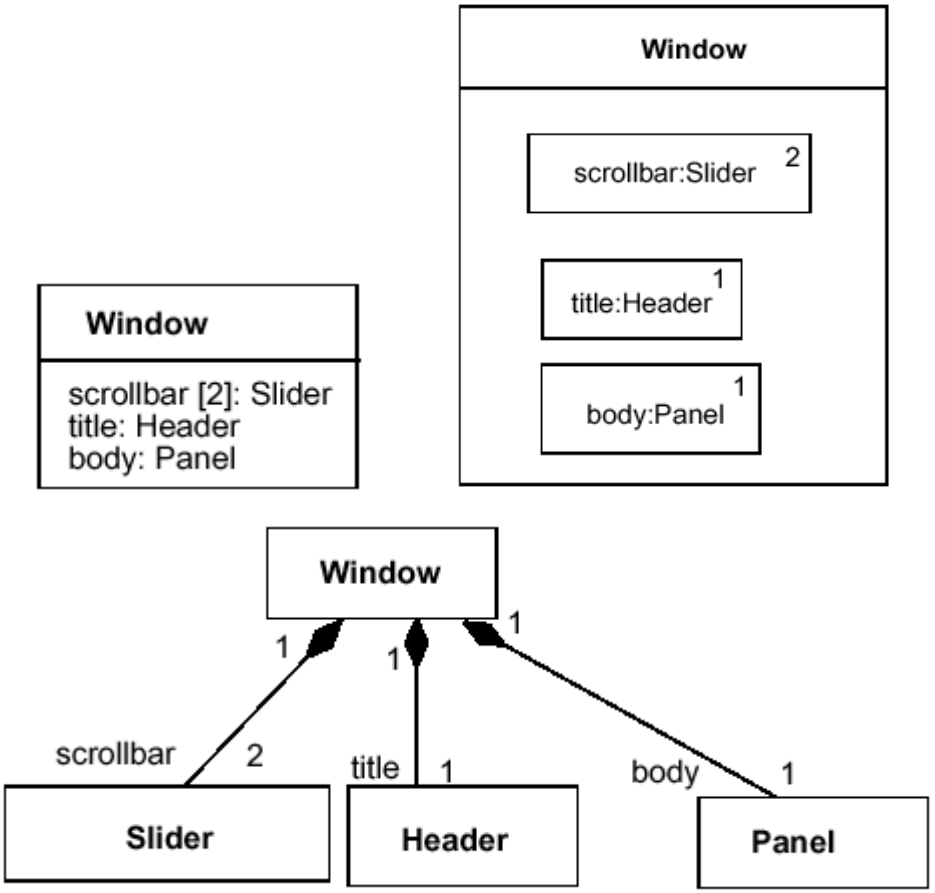

Figura 23: Exemplos de Composição

Classe de Associação

Quando uma associação necessitar uma representação diferenciada, por exemplo, tendo atributos ou operações associadas, podemos utilizar o conceito de uma classe de associação. Uma classe de associação é uma associação que ao mesmo tempo possui propriedades de uma classe (ou uma classe que tem propriedades de uma associação. Uma classe de associação corresponde a um único elemento, apesar de seu aspecto dual. Um exemplo de classe de associação é apresentado na figura 24 a seguir:

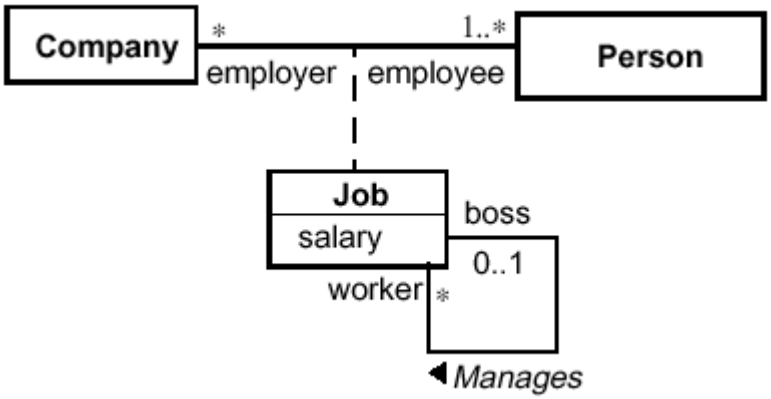

Figura 24: Exemplo de Classe de Associação

Associações N-árias

Associações n-árias são associações entre três ou mais classifiers (onde um mesmo classifier pode aparecer mais de uma vez). Neste caso, a multiplicidade em um papel representa o número potencial de instâncias de uma tupla na associação quando os outros N-1 valores são definidos. Associações n-árias não podem conter marcadores de agregação. Um exemplo de uma associação n-ária é apresentada na figura 25 a seguir:

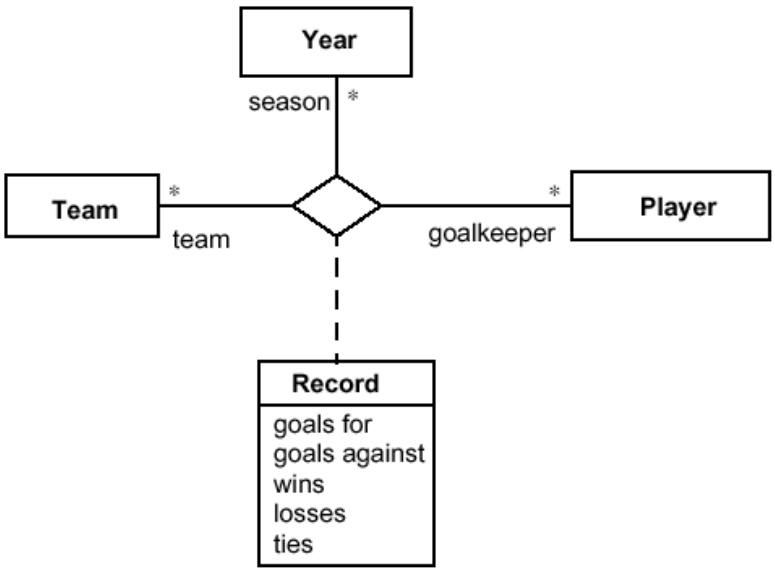

Figura 25: Exemplo de Associação N-ária

Generalizações

Generalizações são relacionamentos taxonômicos entre um elemento mais geral (o pai) e um elemento mais específico (o filho) que deve ser consistente com o primeiro elemento e que adiciona informações adicionais. Generalizações podem ser utilizadas para classes, pacotes, casos de uso e outros elementos. Exemplos de generalizações são apresentados na figura 26 a seguir:

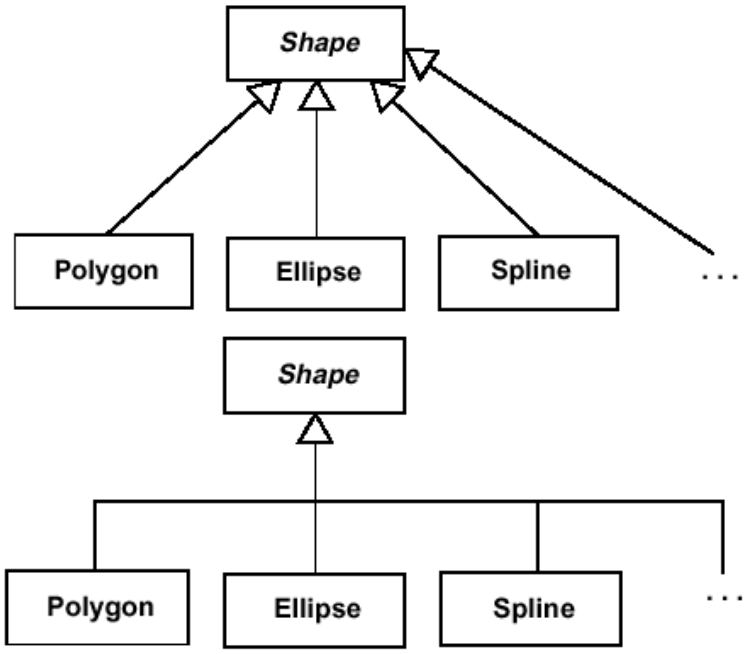

Figura 26: Exemplos de Generalizações (Herança)

A figura 27 a seguir apresenta generalizações com restrições e descrições:

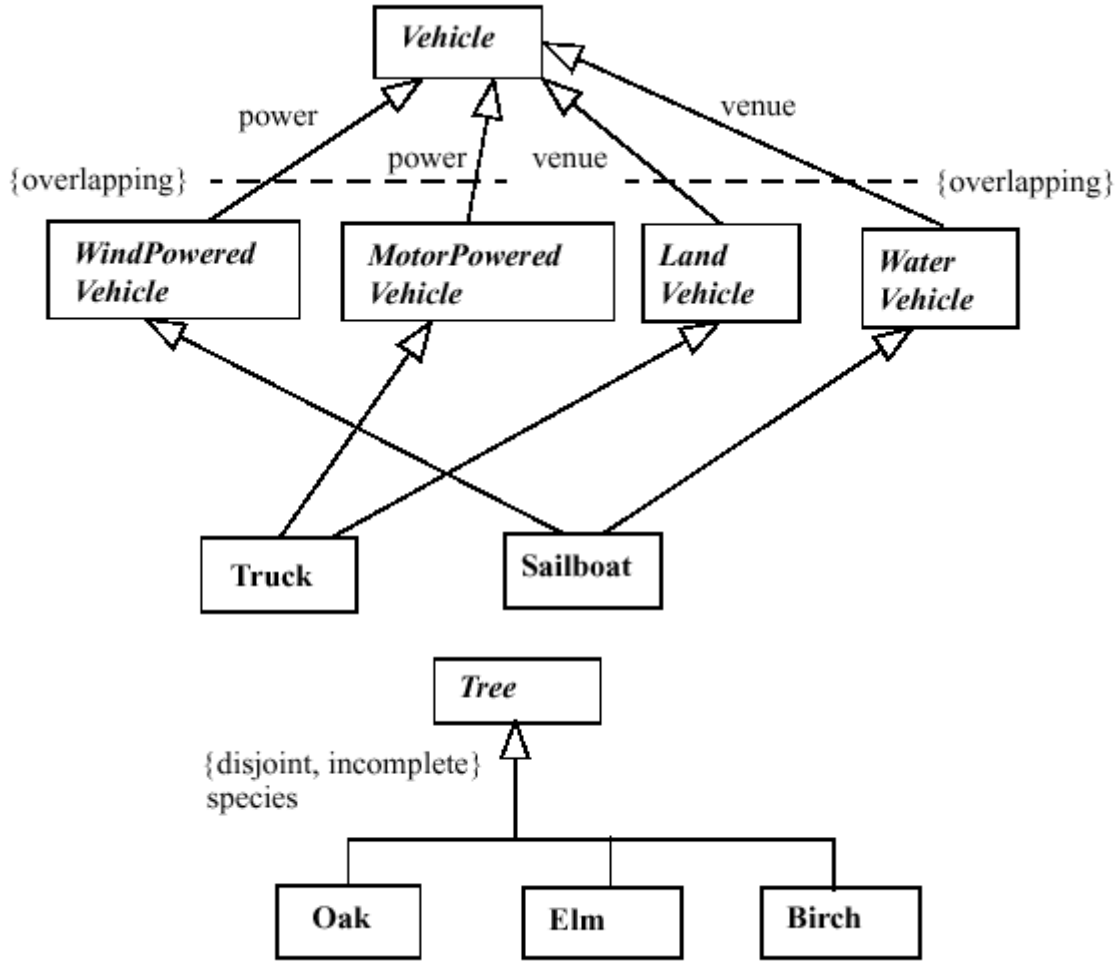

Figura 27: Exemplos de Generalização com Restrições e Descrições

Dependências

Dependências indicam um relacionamento semântico entre os dois elementos de modelagem (ou conjunto de elementos de modelagem). As dependências relacionam os elementos de modelagem por si só, não demandando um conjunto de instâncias para seu significado, e normalmente indicam situações em que uma mudança em um dos elementos pode demandar uma mudança no elemento que dele depende.

A linguagem UML estabelece ainda um conjunto de estereótipos padrões para dependências: access, bind, derive, import, refine, trace e use.

Outro recurso é a indicação de elementos derivados por meio de dependências, tais como na figura 28 a seguir:

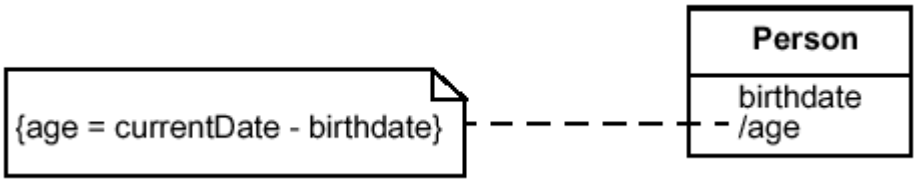

Figura 28: Exemplo de Dependência

Outros exemplos de dependências podem ser vistos na figura 29 a seguir:

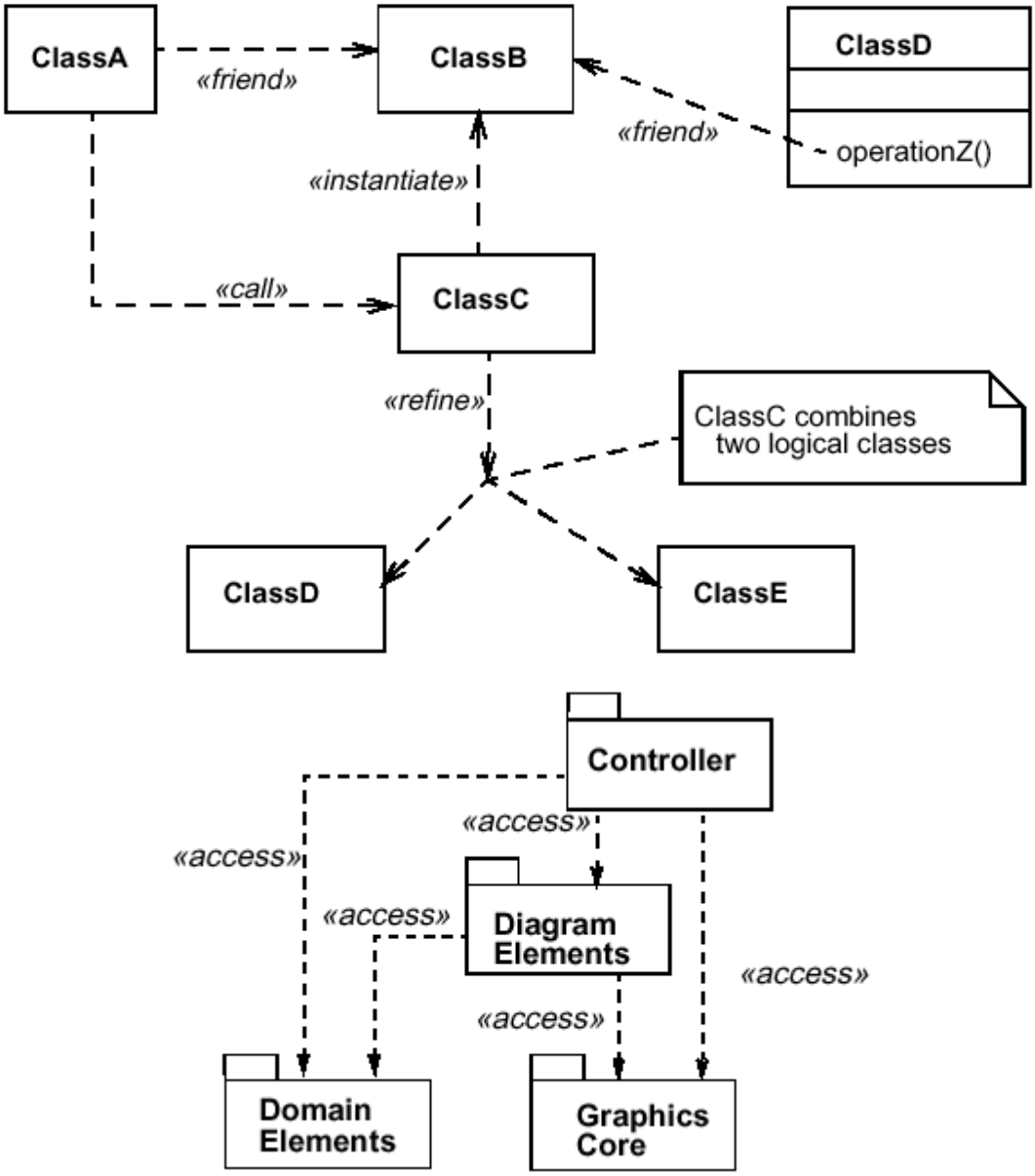

Figura 29: Outros Exemplos de Dependências

Bibliografia

[1] UML Infrastructure - Versão 2.3 - http://www.omg.org/spec/UML/2.3/Infrastructure/PDF/
[2] UML Superstructure - Versão 2.3 - http://www.omg.org/spec/UML/2.3/Superstructure/PDF/