Artigo SQL Magazine 63 - Utilizando UML: Diagrama de Classes

Neste terceiro artigo da série, abordaremos o Diagrama de Classes, um dos mais importantes diagramas da UML.

Esse artigo faz parte da revista SQL Magazine edição 63. Clique aqui para ler todos os artigos desta edição

Projeto

Utilizando UML: Diagrama de Classes

Este é o terceiro artigo de uma série que aborda o assunto UML. Nesta série estamos demonstrando os conceitos descritos na versão 2.0 da UML. No último artigo, apresentamos o Diagrama de Casos de Uso, importante ferramenta utilizada nas etapas iniciais da Análise de Requisitos. Neste terceiro artigo da série, abordaremos o Diagrama de Classes, um dos mais importantes diagramas da UML.

Introdução

O principal objetivo da análise de sistemas é realizar um mapeamento prévio do comportamento requerido para os elementos de modelagem no sistema a serem implementados posteriormente nas fases de construção. Durante as etapas iniciais de um projeto, é comum realizarmos um refinamento nos detalhes e na precisão do "desenho do sistema" a fim de conseguir classes de análise que possam evoluir antes de serem detalhadas durante as atividades de especificação e implementação.

No presente artigo, falaremos sobre um importante diagrama da UML, o diagrama de classes, cujo principal objetivo é permitir a visualização e o relacionamento existente entre as classes obtidas nas etapas iniciais através da modelagem conceitual.

O Diagrama de Classes

O diagrama de classes é considerado por muitos autores como o mais importante e o mais utilizado diagrama da UML. Seu principal enfoque está em permitir a visualização das classes que irão compor o sistema com seus respectivos atributos e métodos, bem como em demonstrar como as classes do sistema se relacionam, se complementam e transmitem informações entre si. Este diagrama apresenta uma visão estática de como as classes estão organizadas, preocupando-se em definir a estrutura lógica das mesmas. O diagrama de classes serve como base para a construção da maior parte dos demais diagramas da UML.

Basicamente, o diagrama de classes é composto por suas classes e pelas associações existentes entre elas, ou seja, os relacionamentos entre as classes. Segundo Guedes em seu livro "UML – Uma Abordagem Prática", o objetivo do diagrama de classes é mostrar os relacionamentos existentes entre as classes que são abstraídas no projeto, e como esses relacionamentos colaboram para a execução de um processo específico.

Perspectivas de Construção de Diagramas de Classe

Existem três perspectivas que você pode usar quando projetar diagramas de classes:

- **Conceitual:** Se tomarmos a perspectiva conceitual, você projeta um diagrama que representa os conceitos do domínio que está sendo estudado. Estes conceitos serão naturalmente relacionados às classes que irão executá-los. Na verdade, um modelo conceitual deve ser projetado com pouca ou nenhuma preocupação com o software que poderá implementá-lo. Portanto, deve ser considerado independente da linguagem implementada. Esta perspectiva recebe o nome de perspectiva essencial.
- **Especificação:** Agora estamos examinando o software, mas estamos analisando as suas interfaces, não a sua implementação. O desenvolvimento orientado a objetos dispõe muita ênfase na diferença entre interface e implementação, mas isso é freqüentemente negligenciado na prática porque a noção que temos de classe em uma linguagem orientada a objetos combina interface com implementação.
- **Implementação:** Nesta visão, realmente temos classes e estamos pondo a implementação às claras. Esta é, provavelmente, a perspectiva usada com mais freqüência.

A compreensão das diversas perspectivas é crucial tanto para desenhar como para ler diagramas de classes. Infelizmente, as linhas entre as perspectivas não são rígidas, e a maioria dos analistas de sistemas não se preocupa em ter suas perspectivas classificadas quando eles estão desenvolvendo a modelagem de um sistema.

Classes, Atributos e Métodos

As classes, atributos e métodos são os principais elementos que compõem um diagrama de classes.
Classes costumam possuir atributos, que, como já foi explicado no artigo anterior desta série (ver **Nota 1**), armazena os dados dos objetos da classe, e métodos, que são as funções que uma instância da classe pode executar. Os valores dos atributos podem variar de instância para instância. Graças a essa característica, aliás, é possível identificar cada objeto individualmente, ao passo que os métodos são idênticos para todas as instâncias de uma classe específica.

Nota 1. Classes
No último artigo da série pudemos ver que uma classe descreve um conjunto de objetos com as mesmas propriedades (atributos), o mesmo comportamento (métodos), os mesmos relacionamentos com outros objetos e a mesma semântica.

Embora os métodos sejam declarados no diagrama de classes, identificando os possíveis parâmetros que são por eles recebidos e os possíveis valores por eles retornados, o diagrama de classes não se preocupa em definir as etapas que estes métodos deverão percorrer quando forem chamados, sendo esta função atribuída a outros diagramas, como o diagrama de seqüência, que se preocupa com a ordem temporal de execução dos métodos.
Uma classe, na linguagem UML, é representada por um retângulo com até três divisões, descritas a seguir (ver **Figura 1**):

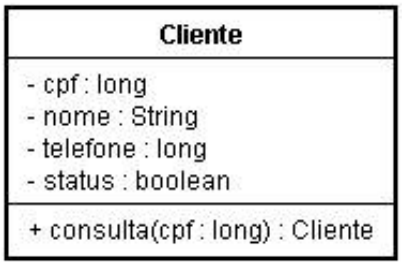


Figura 1. Exemplo de classe contendo atributos e métodos de um cliente.

- **Descrição ou Nome da Classe:** A primeira divisão contém a descrição ou nome da classe, que no exemplo apresentado na **Figura 1** é *Cliente*;
- **Atributos da Classe:** A segunda divisão armazena os atributos e seus tipos de dados (o formato que os dados devem possuir para serem armazenados em um atributo). Neste caso, a classe *Cliente* contém o atributo '*cpf*' e '*telefone*', que são do tipo *long*, o atributo '*nome*' é do tipo *String* e o '*status*' do tipo *boolean*;
- **Métodos da Classe:** A terceira divisão contém a lista dos métodos da classe, que no caso da classe *Cliente* contém apenas o método '*consulta*' que recebe como parâmetro um valor do tipo *long* e retorna uma instância da classe *Cliente*;

Os símbolos de mais e menos na frente dos atributos e métodos representam a visibilidade dos mesmos, o que determina quais tipos de classe podem utilizar o atributo ou método em questão. Estes símbolos são conhecidos como qualificadores de acesso (ver **Tabela 1**).

Qualificador	Significado
public	Indica que o elemento da classe pode ser utilizado livremente por outras classes do mesmo ou de outro pacote.

protected	Indica que existe proteção e os detalhes da execução da classe não são apresentados.
private	Indica que apenas manipulações internas pode haver dentro da classe, não permitindo acesso nem mesmo por suas instâncias.
package	Indica que o conteúdo público da classe pode ser utilizado livremente por outras classes pertencentes ao mesmo pacote. (ler **Nota 2**)

Tabela 1. Qualificadores de acesso permitem determinar a acessibilidade do atributo ou método da instância.

Nota 2. Qualificador de Acesso '*package*'
O qualificador *package* é suportado pela linguagem de programação Java e não é descrito na especificação da UML. Muitos softwares para construção de diagramas UML, tal como *Umbrello* e o *Jude*, suportam sua utilização.

Não é obrigatório que uma classe apresente as três divisões, já que podem existir classes que não possuam atributos ou que não possuam métodos, ou ainda que seus atributos e métodos não precisem ser apresentados no diagrama, já que é recomendado apresentar somente os atributos relevantes ao diagrama para evitar, por exemplo, que o diagrama se torne poluído e com entendimento confuso. Desta forma, é possível encontrar classes com somente duas divisões ou mesmo com apenas uma, neste caso a que contém a descrição da classe, pois esta é obrigatória segundo a especificação da UML.

Relacionamentos

As classes costumam possuir relacionamentos entre si com o intuito de compartilhar informações que colaborem uma com as outras para permitir a execução dos diversos processos executados pelo sistema. As seções a seguir apresentam as diversas formas de relacionamento possíveis em um diagrama de classes da UML.

Associações

Uma associação descreve um vínculo que ocorre normalmente entre duas classes, chamado neste caso de associação binária, mas é perfeitamente válido que uma classe esteja vinculada a si mesma, caso conhecido como associação unária, ou que uma mesma associação seja compartilhada por várias classes, o que é conhecido como associação ternária ou N-ária, embora esta última seja o tipo de associação mais raro e complexo.

Em uma associação, determina-se que as instâncias de uma classe estão de alguma forma ligadas às instâncias das outras classes envolvidas na associação, podendo haver troca de informações entre elas e compartilhando métodos, ou mesmo que uma determinada instância de uma das classes origine uma ou mais instâncias das outras classes envolvidas na associação. Uma associação pode ainda identificar algum nível de dependência entre as classes que a compõem.

As associações representam o equivalente mais próximo dos relacionamentos realizados no MER – Modelo Entidade-Relacionamento (ver **Nota DevMan 1**), ou seja, seu objetivo é definir a maneira como as classes estão unidas entre e interagem entre si, compartilhando informações.

Nota DevMan 1. Semelhança do Diagrama de Classes ao MER

O diagrama de classes foi intencionalmente projetado para ser uma evolução do Modelo de Entidade-Relacionamento (MER) e pode ser utilizado para modelar a estrutura lógica das tabelas que irão compor o banco de dados. Nas entidades que compunham o MER, no entanto, eram definidos apenas os dados a serem preservados, que são representados nas classes pelos seus atributos. Porém, o diagrama de classes oferece ainda a possibilidade de definir as operações que podem ser aplicadas às tarefas, representadas pelos métodos.

Da mesma maneira que no diagrama de casos de uso, as associações são representadas por linhas conectando as classes envolvidas, podendo também possuir setas em suas extremidades para indicar a navegabilidade da associação. A navegabilidade representa o sentido em que as informações são transmitidas entre as classes envolvidas, embora isso não seja obrigatório, mesmo porque se não houver setas significa que as informações podem trafegar entre todas as classes da associação.

As associações podem também possuir títulos para determinar o tipo de vínculo estabelecido entre as classes. Da mesma forma como na navegabilidade, não é obrigatório definir uma descrição para a associação, porém é útil determinar um nome para quando ela não estiver implícita e é necessária alguma forma de esclarecimento. Neste caso, é recomendável determinar também a navegabilidade da associação para facilitar o sentido da leitura.

Associação Unária ou Reflexiva

Este tipo de associação ocorre quando existe o relacionamento de uma classe consigo mesmo. Um exemplo de associação unária pode ser observado na **Figura 2**.

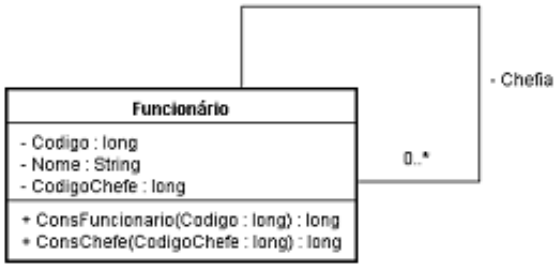


Figura 2. Associação Unária – Exemplo de classe Funcionário.

Podemos perceber no exemplo da classe *Funcionário* que a mesma possui os atributos *código*, *nome* e *código* do possível chefe do funcionário. O chefe do funcionário também é, por sua vez, um funcionário da empresa e, portanto, também constitui uma instância da classe *Funcionário*. Dessa forma, a associação chamada "*Chefia*" indica uma possível relação entre uma ou mais instâncias da classe *Funcionário*, ou seja, esta associação determina que um funcionário pode ou não chefiar outros funcionários.

É necessário existir uma associação da classe funcionário com ela mesma, o que força a classe a possuir um atributo *CodigoChefe* para armazenar o código do funcionário que é responsável pela instância do funcionário em questão. Desse modo, após consultar uma instância da classe funcionário, pode-se utilizar o atributo *CodigoChefe* da instância consultada para pesquisar por outra instância que possua um valor no atributo *Codigo* com o mesmo valor contido no atributo *CodigoChefe* do funcionário já pesquisado.

Outra informação existente na associação, além de seu próprio nome, é a informação representada pelo valor "0..*". Esta é conhecida como multiplicidade (ver **Nota 3**). No exemplo apresentado da classe Funcionário (ver Figura 2), a multiplicidade "0..*" indica que um determinado funcionário pode chefiar nenhum (0) ou muitos (*) funcionários, ou seja, um funcionário pode não chefiar ninguém ou pode chefiar um ou mais funcionários. Pode se perceber que só existe multiplicidade em uma das extremidades da associação, por padrão, quando não existe multiplicidade explícita entende-se que a multiplicidade é "1..1", significando que um e somente um objeto desta extremidade se relaciona com os objetos da outra extremidade. No exemplo apresentado na **Figura 2** isso significa que um funcionário pode ou não chefiar outros funcionários, mas um funcionário possui um e apenas um funcionário como chefe imediato. Alguns dos diversos valores de multiplicidade que podem ser utilizados em uma associação são apresentados na **Tabela 2**.

Nota 3. Definição de Multiplicidade
A multiplicidade é extremamente semelhante ao conceito de cardinalidade utilizado no MER. A multiplicidade procura determinar qual das classes envolvidas em uma associação fornece informações para as outras, além de permitir especificar o nível de dependência de uma classe para com as outras envolvidas na associação.

Multiplicidade	Significado
0..1	

	No mínimo zero (nenhum) e no máximo um. Indica que os objetos das classes associadas não precisam obrigatoriamente estar relacionados, mas se houver relacionamento indica que apenas uma instância da classe se relaciona com as instâncias da outra classe.
1..1	Um e somente um. Indica que apenas um objeto da classe se relaciona com os objetos da outra classe.
*0..**	No mínimo nenhum e no máximo muitos. Indica que pode ou não haver instâncias da classe participando do relacionamento
*	Muitos. Indica que muitos objetos da classe estão envolvidos no relacionamento.
*1..**	No mínimo um e no máximo muitos. Indica que há pelo menos um objeto envolvido no relacionamento, podendo haver muito envolvidos.
3..5	No mínimo três e no máximo cinco. Indica que existem pelo menos três instâncias envolvidas no relacionamento e que podem ser quatro ou cinco as instâncias envolvidas, mas não mais do que isso.

Tabela 2. Exemplos de multiplicidade.

Associação Binária

Associações binárias ocorrem quando são identificados relacionamentos entre duas classes. Este tipo de associação constitui-se na mais comum encontrada nos diagramas de classes.

Figura 3. Associação Binária. Exemplo de associação entre as classes Sócio e Dependente.

Na **Figura 3** podemos verificar que um objeto da classe *Sócio* pode se relacionar ou não com instâncias da classe *Dependentes*, conforme demonstra a multiplicidade 0..*, enquanto que, se existir um objeto da classe *Dependente*, ele terá que se relacionar obrigatoriamente com um objeto da classe *Sócio*, pois como não foi definida a multiplicidade na extremidade da classe *Sócio*, entende-se que a multiplicidade é 1..1. Podemos acrescentar uma descrição para esta associação, bem como definir a direção das informações por meio da utilização da navegabilidade. No exemplo apresentado na **Figura 4**, podemos ver o mesmo exemplo das classes *Sócio* e *Dependente* apresentado na **Figura 3**, adicionando um nome para a associação e definindo sua navegabilidade.

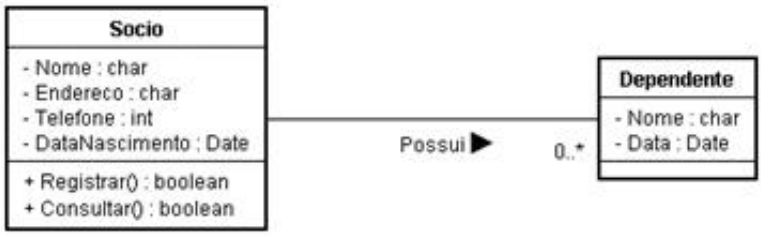


Figura 4. Associação Binária com Descrição e Navegabilidade.

A utilização de uma descrição à associação apresentada na **Figura 4** permite compreender melhor o objetivo da associação e a definição da navegabilidade, além de auxiliar a multiplicidade a definir o sentido das informações e facilitar a leitura da associação. Como a navegabilidade da associação "Possui" aponta da classe *Sócio* para a classe *Dependente*, podemos ler a associação da seguinte maneira: "Uma instância da classe *Sócio* possui no mínimo nenhuma instância e no máximo muitas instâncias da classe *Dependente*, e uma instância da classe *Dependente* é possuída por uma e somente uma instância da classe *Sócio*".

Associação Ternária ou N-ária

Associações Ternárias ou N-árias são associações que conectam mais de duas classes. São representadas por um losango para onde convergem todas as ligações da associação (**ver Figura 5**).

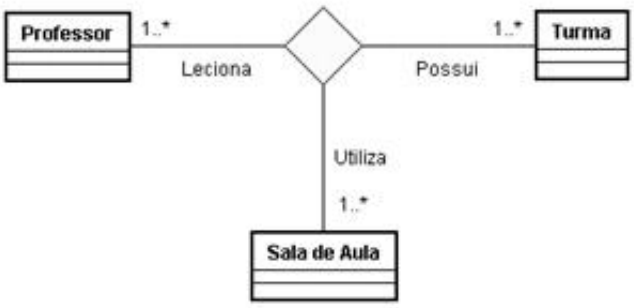


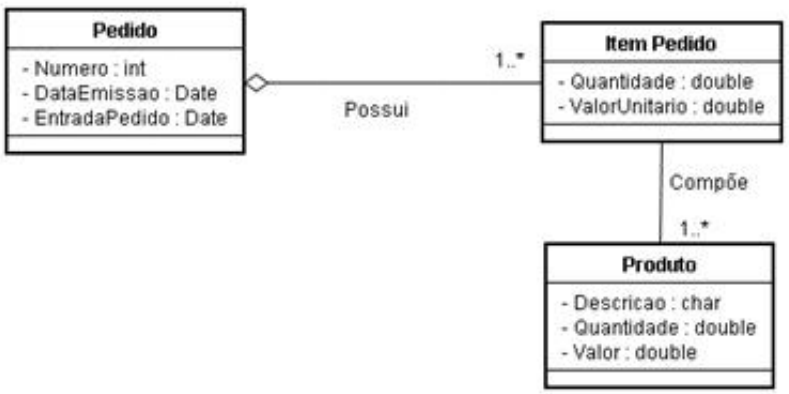
Figura 5. Exemplo de Associação Ternária ou N-ária.

Na **Figura 5** identificamos uma associação que demonstra um fato corriqueiro na maioria das universidades, onde um *professor* pode lecionar para muitas *turmas*, uma *turma* pode possuir muitos *professores* e utilizar muitas *salas de aula* e um *professor* lecionando para uma *turma* especifica pode utilizar mais de uma *sala de aula*.

Assim, podemos ler a associação apresentada na **Figura 5** da seguinte forma: "Um professor leciona para no mínimo uma turma e no máximo para muitas, uma turma possui no mínimo um professor e no máximo muitos, e um professor leciona para uma determinada turma em no mínimo uma sala ou no máximo muitas". As associações n-árias são úteis para demonstrar associações complexas, no entanto deve-se evitar utilizá-las, pois sua leitura é, por vezes, difícil de ser interpretada.

Agregação

Agregação é um tipo especial de associação onde se tenta demonstrar que as informações de um objeto (chamado objeto-todo) precisam ser complementadas pelas informações contidas em um ou mais objetos de outra classe (chamados objeto-parte). Este tipo de associação tenta demonstrar uma relação Todo/Parte entre os objetos associados. Objetos-parte não podem ser destruídos por um objeto diferente do objeto-todo e nem serem criados por outro que não seja objeto-todo. O símbolo de agregação difere do símbolo de associação por conter um losango na extremidade da classe que contém os objetos-todo.



A **Figura 6** demonstra um exemplo de agregação, onde existe uma classe *Pedido* que armazena os objetos-todo e uma classe *Item Pedido*, onde são armazenados os objetos-parte.

Figura 6. Agregação – no exemplo existe agregação apenas entre a classe *Pedido* e classe *Item Pedido*.

No exemplo apresentado na **Figura 6**, podemos perceber pela multiplicidade 1..* na associação *Compõe* na extremidade da classe *Produto*, que um objeto da classe *Produto* pode se referir a muitos objetos da classe *Item Pedido*. No entanto, um objeto da classe *Item Pedido* se refere a somente uma instância da classe *Produto*, a multiplicidade na extremidade da classe *Item Pedido* não foi especificada e portanto é 1..1.

No caso da agregação entre as classes *Pedido* e *Item Pedido*, é apresentada uma diferença em relação à associação binária entre a classe *Produto* e a classe *Item Pedido*. Um pedido pode tanto possuir apenas um item como vários, e é muito difícil determinar o número máximo de itens que um pedido pode ter, e mesmo que isso fosse possível, na imensa maioria das vezes o número máximo não seria atingido, o que faria com que diversos atributos reservados para itens do pedido fossem deixados em branco, ocupando um espaço desnecessário.

No exemplo apresentado na **Figura 6**, as informações da classe *Pedido* estão incompletas, possuindo apenas atributos que não se repetem, como por exemplo o número do pedido e a data em que este foi expedido. Os atributos que podem se repetir, no caso referente aos dados das instâncias da classe *Item Pedido*, como por exemplo a quantidade do item solicitado e seu valor unitário, devem ser armazenados em uma classe dependente da classe *Pedido*. Dessa forma, sempre que uma instância da classe *Pedido* for pesquisada, todas as instâncias da classe *Item Pedido* relacionadas à instância da classe *Pedido* pesquisada deverão ser apresentadas. Da mesma forma, sempre

que um objeto *Pedido* for excluído, todos os objetos da classe *Item Pedido* a ele relacionados devem também ser excluídos.

Composição

Uma associação do tipo Composição constitui-se em uma variação da associação de Agregação. Ela tenta representar um vínculo mais forte entre os objetos-todo e os objeto-parte, procurando demonstrar que os objetos-parte têm de pertencer exclusivamente a um único objeto-todo com que se relacionam. Em uma composição, um mesmo objeto-parte não pode se associar a mais de um objeto-todo.

O símbolo de composição diferencia-se graficamente do símbolo da agregação por utilizar um losango preenchido. Da mesma forma que na agregação, o losango deve ficar ao lado do objeto-todo (ver **Figura 7**).

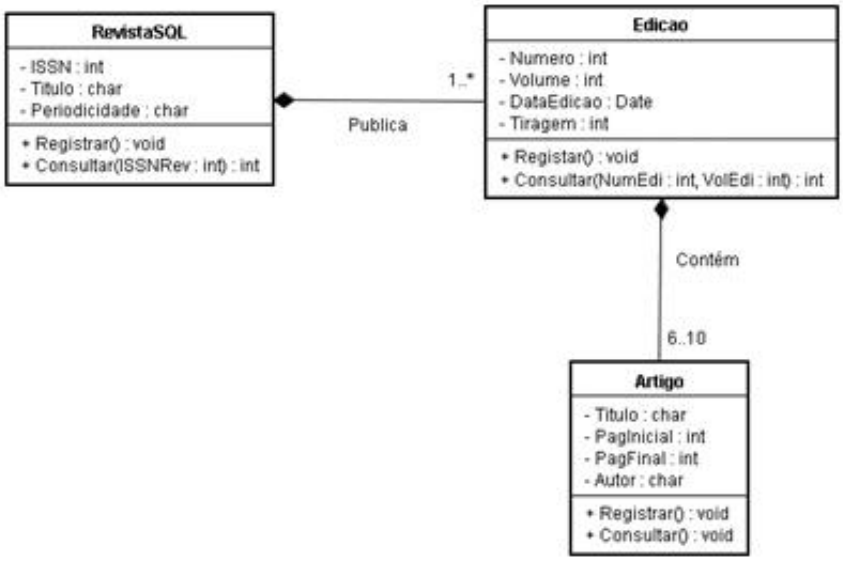


Figura 7. Composição

Observando a **Figura 7**, percebe-se que um objeto da classe *RevistaSQL* se refere a no mínimo um objeto da classe *Edição*, podendo se referir a muitos objetos desta classe, e que cada instância da classe *Edição* se relaciona única e exclusivamente a uma instância específica da classe *RevistaSQL*, não podendo relacionar-se com nenhuma outra.
Percebemos no exemplo apresentado que um objeto da classe *Edição* deve se relacionar a no mínimo 6 objetos da classe *Artigo*, podendo se relacionar com até 10 objetos desta classe. Esse tipo de informação se torna válida como documentação e serve como uma forma de validação, que impede que uma revista seja publica com menos de 6 ou mais de 10 artigos. No entanto, um objeto da classe *Artigo* se refere unicamente a um objeto da classe *Edição*. Isso também é uma forma de documentação, já que uma edição de uma revista só deve publicar trabalhos inéditos, assim é lógico que não é possível a um mesmo objeto da classe *Artigo* relacionar-se a mais de um objeto da classe *Edição*.
Isto não ocorre no exemplo de agregação apresentado na **Figura 6**, porque pode ocorrer de mais de um pedido possuir itens de pedido com características iguais, por exemplo, pode haver dois pedidos com um mesmo item de pedido referindo-se a 10 unidades de um livro de linguagem de programação Java. O simples fato de um produto se referir a mais de um item de pedido já implica que os objetos itens de pedido podem se repetir.

Especialização/Generalização

Este é um tipo especial de relacionamento muito similar à associação de mesmo nome utilizado no diagrama de casos de uso (**Nota 4**). Seu objetivo é identificar classes ancestrais, chamadas gerais, e classes herdeiras, chamadas especializadas. Este tipo de relacionamento permite também demonstrar a ocorrência de métodos polimórficos (**Nota 5**) nas classes especializadas do sistema.

Nota 4. Diagrama de Casos de Uso

No último artigo da série abordamos o diagrama de casos de uso. Sugere-se fortemente a sua leitura.

Nota 5. Conceito de métodos polimórficos

O conceito de criação de método polimórficos permite através da assinatura dos métodos (parâmetros que são recebidos), que um mesmo método possua em uma classe várias implementações de uma operação de formas diferentes. A mesma operação (com o mesmo nome) tem várias ("poli") formas ("morfismo") de ser chamada pela classe herdeira da classe ancestral. Este conceito na orientação a objetos é realizado através da sobrecarga (*overload*) de métodos.

Assim como no diagrama de casos de uso, a especialização/generalização ocorre quando existem duas ou mais classes com características muito semelhantes. Assim, para evitar ter que declarar atributos e/ou métodos idênticos como uma forma de reaproveitar código, cria-se uma classe geral onde são declarados os atributos e métodos comuns a todas as classes envolvidas no processo e então se declaram classes especializadas ligadas à classe geral, que herdam todas as suas características podendo possuir atributos e métodos próprios.

Além disso, métodos podem ser redeclarados em uma classe especializada, possuindo o mesmo nome, mas comportando-se de forma diferente, não sendo, portanto necessário modificar o código-fonte do sistema em relação às chamadas de métodos das classes especializadas, porque o nome do método não mudou, apenas foi redeclarado em uma classe especializada e só se comporta de maneira diferente quando for chamado por objetos desta classe. Este conceito na orientação a objetos recebe o nome de sobrecarga de métodos. O símbolo de especialização/generalização é o mesmo do diagrama de casos de uso (ver **Figura 8**).

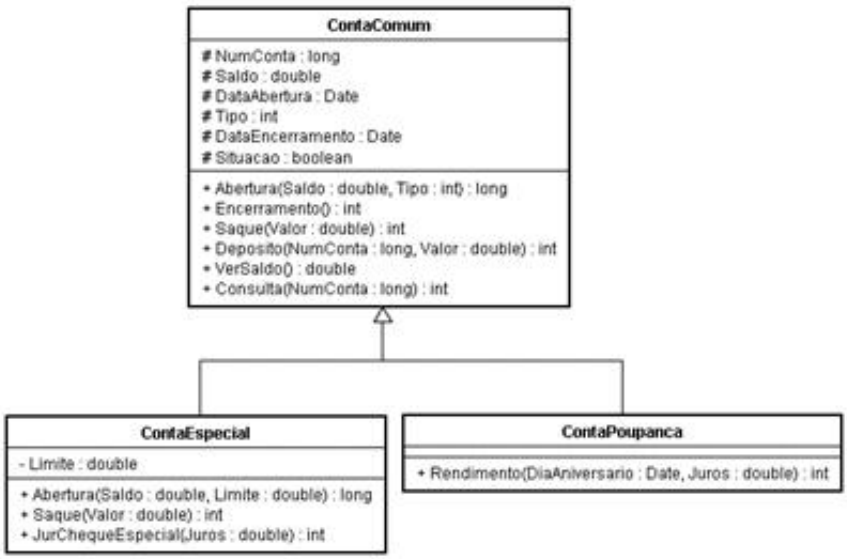


Figura 8. Exemplo de Especialização/Generalização no Diagrama de Classes.

Analisando a **Figura 8**, verificamos a existência de uma classe geral chamada *ContaComum* que possui os atributos número da conta, o saldo da conta, a data que esta foi aberta, o tipo da conta (se é comum, especial ou poupança, um atributo será herdado), a possível data de encerramento da conta e a situação da conta (se está ativa ou não). A classe *ContaComum* possui ainda os seguintes métodos:

- Abertura – para abrir uma conta, que neste caso atua como um método construtor;
- Encerramento – para encerrar uma conta já existente, que não é um método destrutor, porque este método só altera a situação da conta e possivelmente do cliente para inativa;
- Saque – para retirar valores da conta;
- Depósito – para investir valores na conta;
- VerSaldo – para verificar quanto está depositado na conta;
- Consulta – para verificar se uma determinada conta existe;

Existem ainda duas classes especializadas que herdam as características da classe geral (*ContaComum*), que são as classes *ContaEspecial* e *ContaPoupança*. Além dos atributos e métodos herdados da classe *ContaComum*, a classe *ContaEspecial* possui ainda o atributo limite que determina quanto o cliente pode sacar além de seu saldo e os métodos *Abertura*, *Saque* e *JurChequeEspecial*. Os dois primeiros métodos são uma redeclaração dos métodos *Abertura* e *Saque* da classe *ContaComum* (ver **Nota 6**), porque estes precisam incluir o limite da conta. O último método calcula o valor de juros a ser cobrado pelo uso do limite.

Nota 6. Sobrescrita de Métodos

A orientação a objetos nos permite sobrescrever métodos. Diferentemente da sobrecarga (*overload*), quando podemos criar várias versões diferentes do mesmo método, a sobrescrita (*override*) permite criarmos uma nova implementação para um método existente em uma classe ancestral, substituindo-o. As classes derivadas (herdeiras) irão herdar essa nova implementação ao invés da antiga. Não podemos modificar a assinatura do método, tendo que manter a quantidade e os tipos dos parâmetros como também o tipo de retorno do método.

A classe *ContaPoupança* possui, além dos atributos e métodos herdados da classe *ContaComum*, o método *rendimento*, cuja função é calcular os rendimentos a serem acrescidos ao saldo da conta sempre que esta fizer aniversário.

Dependência

O relacionamento de dependência não costuma ser encontrado com muita freqüência nos diagramas de classes. Este relacionamento, como o próprio nome já diz, identifica um certo grau de dependência de uma classe em relação à outra, isto é, sempre que ocorrer uma modificação na classe da qual uma outra classe depende, esta deverá também sofrer esta modificação. O relacionamento de dependência é representado por uma reta tracejada entre duas classes contendo uma seta apontando a classe da qual a classe posicionada na outra extremidade do relacionamento depende de alguma forma (ver **Figura 9**).

Figura 9. Exemplo de relacionamento de Dependência

No exemplo apresentado na **Figura 9**, notamos o relacionamento de dependência entre as classes *ItemCarrinho* e *CarinhoDeCompras*, que representam classes utilizadas em um sistema de vendas através da internet. A classe *ItemCarrinho* possui dependência com a classe *CarrinhoDeCompras*, porque a maioria dos eventos ocorridos na instância desta classe, senão todos, afetam de alguma forma as instâncias da classe *ItemCarrinho*.

Realização

Uma realização é um tipo de relacionamento especial que mistura características dos relacionamentos de generalização e dependência, sendo usada no diagrama de classes para identificar classes responsáveis por executar funções para classes que representam interfaces. Este tipo de relacionamento herda o comportamento de uma classe, mas não sua estrutura. O relacionamento de realização é representado por uma seta tracejada contendo uma seta vazia que aponta para a classe de interface, enquanto que na outra extremidade é definida a classe que realiza um comportamento pretendido pela classe de interface (ver **Figura 10**).

Figura 10. Exemplo de relacionamento de Realização

No exemplo apresentado pela **Figura 10**, observamos uma classe com o estereótipo de <>, representando uma página para submissão de artigos científicos. Entretanto, o comportamento desejado para a interface que é o de aceitar submissões de artigos não é realmente realizado por esta interface, ela apenas repassa as informações fornecidas para outra classe, a classe *ArtigosSubmetidos*, que é a real

responsável pela execução do método *RegistrarArtigos* (ver **Nota DevMan 2**).

Nota DevMan 2. Estereótipos
No último artigo da série pudemos ver que os estereótipos possibilitam um certo grau de extensibilidade aos componentes da UML, além de permitir a identificação de componentes que, embora semelhantes aos outros, possuam alguma característica que os diferencie, dando-lhes mais destaque no diagrama.

Classe Associativa

Classes associativas são classes produzidas quando da ocorrência de associações que possuem multiplicidade muitos (*) em todas as suas extremidades. As classes associativas são necessárias nesses casos porque não existe um repositório que possa armazenar as informações produzidas pelas associações já que todas as classes envolvidas apresentam multiplicidade *muitos*. Isto obriga que seu atributo-chave seja transmitido às outras classes envolvidas, e como todas possuem e mesma multiplicidade, nenhuma delas pode receber os atributos das outras. Assim, é preciso criar uma classe associativa para armazenar os atributos transmitidos pela associação, o que não impede que a classe associativa possua atributos próprios, além dos recebidos. Uma classe associativa é representada por uma seta tracejada partindo do meio da associação e atingindo uma classe (ver **Figura 11**).

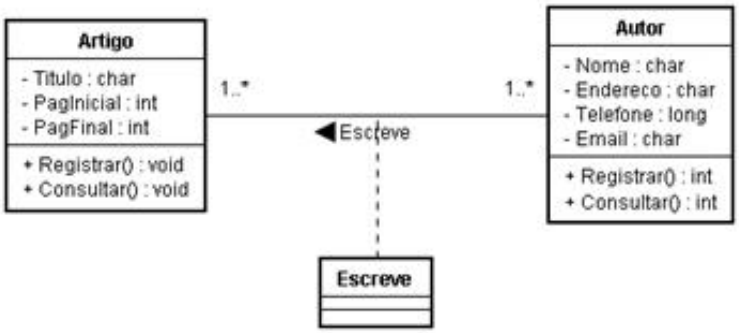


Figura 11. Classe Associativa

No exemplo apresentado na **Figura 11**, podemos notar que a classe *Autor* se relaciona com muitas instâncias da classe *Artigo*, e uma instância da classe *Artigo* se relaciona com muitas instâncias da classe *Autor*, ou seja, um autor pode escrever muitos artigos e um artigo pode ser escrito por muitos autores. Com isso, surge um questionamento: onde essa informação é armazenada? Como existe a multiplicidade *muitos* nas extremidades de ambas as classes da associação, não há como reservar atributos para armazenar as informações decorrentes da associação. Poderia ser reservado um certo número de atributos em uma das classes para armazenar esta informação, mas poderia haver um desperdício de espaço, já que não há como determinar um limite para a quantidade de autores que podem escrever um artigo e nem há como determinar quantos artigos um autor ainda vai escrever. Concluímos que é preciso criar uma classe para guardar essa informação. Note que a classe associativa representada na **Figura 11** não possui declaração de atributos. Ela simplesmente recebe os atributos-chave da classe *Autor* e da classe *Artigo* transmitidos de forma transparente pela associação, não sendo necessário, portanto, criar uma representação.

Uma variação do diagrama de classes apresentado na **Figura 11** poderia substituir a classe associativa por uma classe intermediária, que representaria da mesma forma a associação (ver **Figura 12**). Na verdade, segundo Guedes em seu livro "UML – uma abordagem prática", são duas formas de representar a mesma informação.

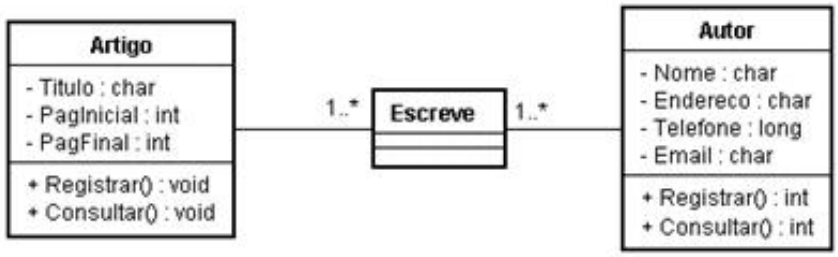


Figura 12. Classe Intermediária substituindo a classe associativa apresentada no exemplo da Figura 11.

Restrições

Restrições são informações extras que definem condições a serem validadas durante a implementação dos relacionamentos entre as classes. As restrições podem ser utilizadas para a maioria do diagramas da UML, no entanto, são mais utilizados no diagrama de classes. As restrições em geral são descritas por textos limitados através de chaves.

Em um sistema de locações, por exemplo, podemos restringir que um sócio somente poderá realizar uma locação se não possuir nenhuma locação anterior pendente (ver **Figura 13**). Restrições podem também ser aplicadas para validar um atributo ou método de uma classe especifica, por meio do uso de Notas (ver **Figura 14**).

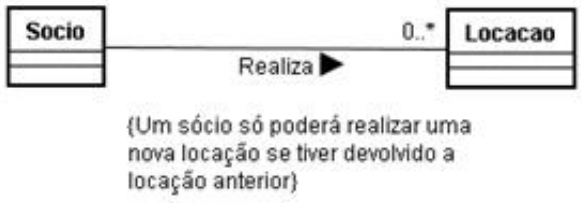


Figura 13. Exemplo de Restrições em uma Associação.

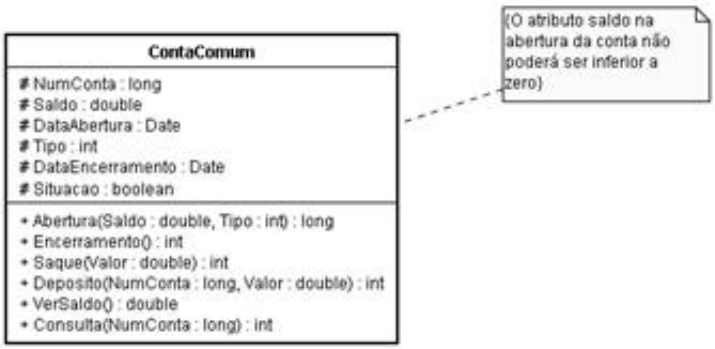


Figura 14. Exemplo de Restrições em uma Classe – note que é utilizada uma Nota para descrever a restrição.

Notas são adornos explicativos colocados em algum ponto contíguo ao elemento explicado, representando simples comentários, não alterando, portanto, o significado do diagrama no qual elas se encontram. As notas são representadas por retângulos com uma dobra no canto superior direito, ligam-se aos componentes do diagrama através de uma linha tracejada, chamada Âncora.
Restrições podem ser utilizadas também para representar o "ou" exclusivo (xor), quando instâncias de duas ou mais classes podem se relacionar com instâncias de uma outra classe específica, mas somente uma instância de uma das classes pode se relacionar com uma instância da classe específica, em detrimento das outras (ver **Figura 15**).

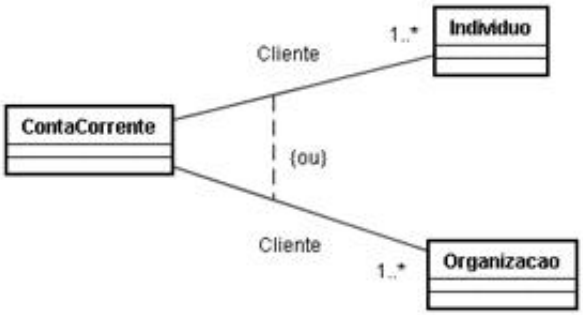


Figura 15. Exemplo de Restrição com Ou Exclusivo – uma conta pode ser possuída tanto por um Individuo como por uma Organização, mas uma conta não pode pertencer a ambos.

Conclusão

A análise orientada a objetos se preocupa com a criação de uma descrição do domínio a partir da perspectiva de uma classificação por objetos. Uma decomposição do domínio envolve a identificação dos conceitos, dos atributos e das associações que são considerados de interesse.

Foi apresentado neste artigo um dos mais, se não o mais, utilizado dos diagrama propostos pela UML, o diagrama de classes. Este diagrama, como vimos, supre com folga a necessidade do analista em visualizar a relação existente entre as classes de modelagem conceitual do sistema. Em seu próximo projeto, considere fortemente a possibilidade de inseri-lo na etapa de análise de sistema.

No próximo artigo da série, abordaremos o diagrama de seqüência, o diagrama da UML utilizado para representar a ordem temporal com que as mensagens são trocadas entre os objetos envolvidos em determinado processo.

Links

JUDE/Community - Free UML Modeling Tool

jude.change-vision.com

Object Management Group – UML

www.uml.org

Umbrello UML Modeller

uml.sourceforge.net/

Referências Bibliográficas

Livro: BOOCH, Grady; RUMBAUGH, James; JACOBSON, Ivar. UML: guia do usuário. Trad. Fábio Freitas. Rio de Janeiro: Campus, 2000.

Livro: GUEDES, Gilleanes T. A.. UML: uma abordagem prática. 2.ed. São Paulo: Novatec, 2006.

Livro: LIMA, Adilson da Silva: UML 2.0 – Do Requisito à Solução. 2.ed. São Paulo: Érica, 2007.

Paulo César Barreto Da Silva

Graduado em Análise de Sistemas pelo Centro Universitário Salesiano de São Paulo e Pós graduado pela Universidade Estadual de Campinas na área de Orientação a Objetos.

Publicado em 1899