Projeto de Software

UNIDADE 5

FASE DE ELABORAÇÃO
Disciplina de Requisitos: Diagramas de Colaboração

Autores:

Profa. Dra Rosana Terezinha Vaccare Braga
Prof. Dr. Paulo Cesar Masiero

ICMC/USP

Professor da Disciplina:
Prof. Dr. Valter Camargo
Prof. Esp. Alexandre Pedroso Fernandes

1 - Disciplina de Projeto – Diagramas de Colaboração

Nas unidades anteriores foi abordada a análise orientada a objetos, sem aprofundar os detalhes de como os modelos produzidos seriam projetados de forma a serem implementados em uma linguagem de programação orientada a objetos. Em outras palavras, dados os requisitos do software a ser desenvolvido, os requisitos foram analisados detalhadamente e foram produzidos modelos que representam o quê o software deve fazer. Portanto, os artefatos produzidos durante a análise OO representam o quê o sistema faz, sem detalhes que comprometam a solução a uma tecnologia específica. A partir deste capítulo, enfocaremos os artefatos de projeto, que contém detalhes sobre como o sistema poderá ser implementado utilizando um computador. Para conseguir isso, será necessário detalhar as informações sobre as classes que comporão o sistema, incluindo o comportamento esperado de cada objeto e a colaboração entre os objetos.

No Processo Unificado, o projeto é conduzido utilizando basicamente dois tipos principais de diagramas UML: o Diagrama de Classes e os Diagramas de Colaboração. Os diagramas de classes serão apresentados em unidades posteriores, pois são construídos a partir de informações presentes nos diagramas de colaboração, vistos nesta unidade.

1.1 – Visão Geral da notação

A UML possui dois diagramas específicos para mostrar a interação entre objetos: o diagrama de colaboração e o diagrama de seqüência. O Diagrama de Seqüência é similar ao visto na Unidade 4, com a diferença que, ao invés de representar os atores e o Sistema, representam-se dois ou mais objetos trocando mensagens. Na Figura 1 é ilustrado como um diagrama de seqüência pode ser utilizado para mostrar a interação entre os objetos por meio de mensagens.

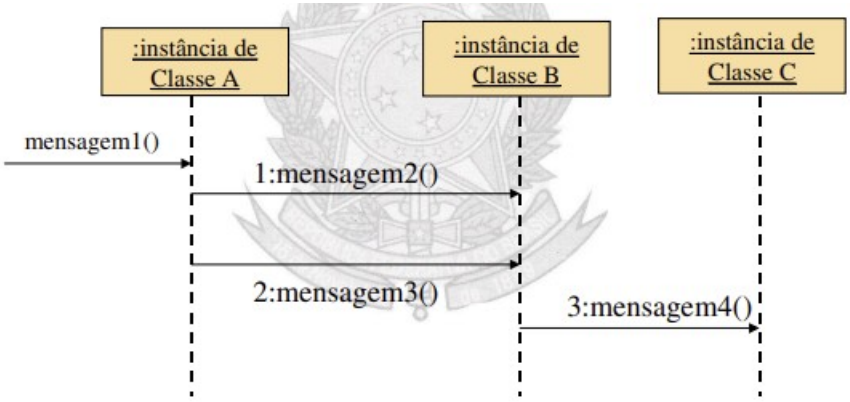


Figura 1 – Notação UML para Diagrama de interação: Diagrama de Seqüência

Note que embora tenha sido utilizada a mesma notação para representar os Diagramas de Seqüência do Sistema (Unidade 4), lá o processo encontrava-se na fase de requisitos (análise) e mostravam-se apenas os atores e o sistema, que era visto como uma caixa preta. Tratava-se, portanto, de uma visão externa do sistema. Já na fase de projeto, a mesma notação é utilizada com um enfoque diferente: deseja-se detalhar a interação entre os objetos, portanto uma visão interna do sistema é desejada neste momento. Para tanto, utilizam-se os diagramas de interação entre objetos.

Nas seções seguintes é mostrada a notação específica para o Diagrama de Colaboração. O mesmo diagrama de interação entre objetos, desenhado na forma de um Diagrama de Seqüência na Figura 1, pode ser desenhado utilizando a notação do Diagrama de Colaboração, como mostrado na Figura 2. Os diagramas de colaboração têm melhor capacidade de expressar informações contextuais, conforme será visto nas seções seguintes, e podem ser mais econômicos em termos de espaço.

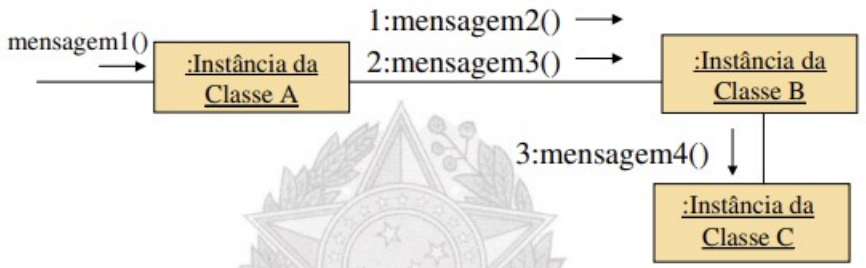


Figura 2 – Notação UML para Diagrama de interação: Diagrama de Colaboração

1.1.1 – Notação UML para mensagens entre dois objetos

Conforme mencionado anteriormente neste curso, no paradigma OO deve-se projetar o sistema em termos de objetos e da comunicação entre eles. A única forma de um objeto se comunicar com outro é enviando uma mensagem para esse outro objeto, que poderá responder a essa mensagem. Vimos também que um objeto, ao receber uma mensagem, executa um método para atender a essa mensagem. A notação da UML para comunicação entre objetos é ilustrada na Figura 3. Alguns detalhes dessa notação serão explicados nas seções seguintes.

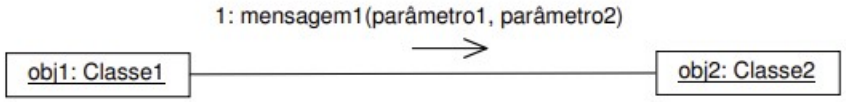


Figura 3 – Mensagem entre dois objetos na UML

O objeto chamado obj1 é uma instância da classe Classe1. O nome do objeto poderia ter sido omitido, portanto poderia ter sido colocado dentro do retângulo apenas :Classe1 (note que o sinal de dois pontos é necessário para indicar que trata-se de um objeto da classe, neste caso um objeto sem nome). Da mesma forma, obj2 é uma instância da classe Classe2. O objeto obj1 está enviando uma

mensagem, em particular a mensagem1, ao objeto obj2. Alguns parâmetros estão sendo passados ao objeto obj2 (parâmetro1 e parâmetro2), que são colocados entre parênteses, separados por vírgula.

Para poder responder a mensagem1, a Classe2 deve possuir um método mensagem1 em sua interface, ou seja, o método deve estar disponível e ser visível a Classe1.

Na Figura 4 é mostrado um exemplo mais concreto, em que o objeto linhaEmpto (da classe LinhaDoEmprestimo) envia uma mensagem ao objeto copiaLivro (da classe CopiaDoLivro) para solicitar que a situação da cópia seja modificada para “disponível”, após a devolução do livro. A classe CopiaDoLivro deve possuir um método para alterar o valor do atributo situação (o nome desse método é mudarSituacao e ele possui um parâmetro do tipo String) .

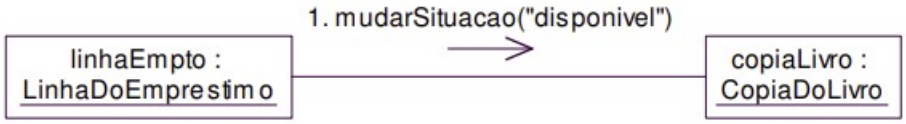


Figura 4 – Exemplo de mensagem entre dois objetos

1.1.2 – Ordem das mensagens

Em geral, um Diagrama de Colaboração mostra a comunicação entre vários objetos para realizar um certo comportamento desejado. Por exemplo, se um objeto não contém conhecimento suficiente para responder a uma determinada mensagem, ele pode recorrer a outros objetos que conheça, enviando-lhes mensagens para obter o conhecimento desejado.

Nesse sentido, a ordem em que as mensagens são enviadas é freqüentemente muito importante. Por isso, as mensagens são numeradas (ver Figuras 1 a 4). A Figura 5 mostra um outro exemplo em que a ordem das mensagens é importante. Por exemplo, a linhaDoEmpréstimo só pode ser criada (mensagem 3) após a obtenção do tipo do leitor (mensagem 2), pois o método criar() passa como parâmetro esse tipo de leitor. Outros conceitos e notações incluídos nesta figura serão explicados nas seções seguintes.

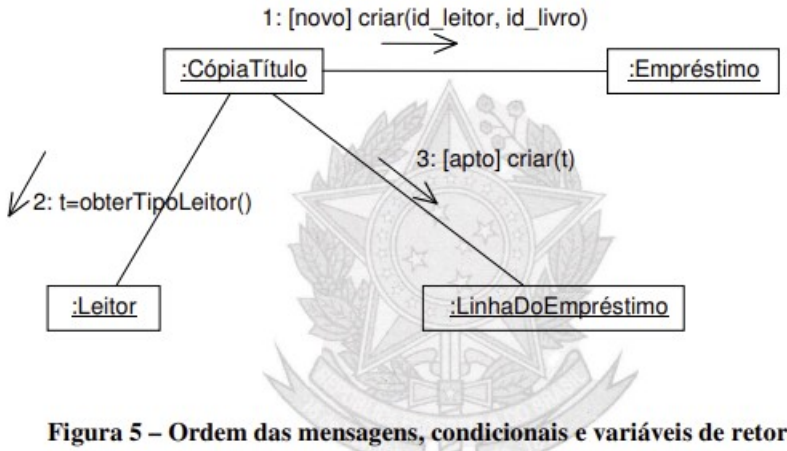


Figura 5 – Ordem das mensagens, condicionais e variáveis de retorno

1.1.3 – Variáveis de retorno

Em um Diagrama de Colaboração, pode ser necessário armazenar certos resultados de invocação de mensagens para serem utilizados posteriormente como, por exemplo, para serem passados como parâmetros em outras mensagens. Para isso, pode-se utilizar variáveis, como ilustrado na Figura 5. Na mensagem número 2, cujo nome é obterTipoLeitor(), obtêm-se o tipo do leitor (por exemplo, aluno, professor, etc.), que é atribuído a uma variável t. Esse tipo é passado como parâmetro para a criação da linhaDoEmprestimo (mensagem número 3). Nesse caso específico, linhaDoEmprestimo utiliza t para calcular a data de devolução do livro.

Um objeto de uma classe específica pode ser retornado como resultado da invocação de uma mensagem e pode ser atribuído a uma variável. Depois disso, esse objeto pode ser invocado diretamente pelo objeto que o atribuiu à variável, como exemplificado na Figura 6, em que o objeto c1 é obtido por e1 pela invocação da mensagem copia() ao objeto l1, e então e1 pode invocar a mensagem mudarSituacao desse objeto c1.

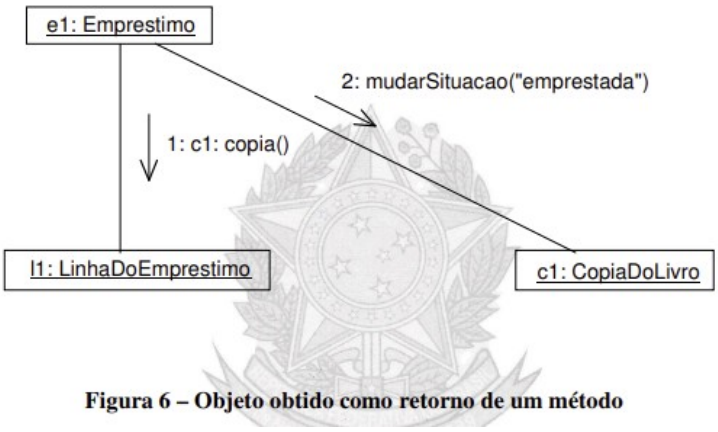


Figura 6 – Objeto obtido como retorno de um método

1.1.4 – Execução condicional de mensagem

Pode ser desejável que certas mensagens só sejam executadas em circunstâncias pré-definidas, por exemplo, um empréstimo só pode se concretizar se o leitor estiver apto a emprestar livros (não ultrapassou a quantidade máxima de livros permitidos e não está devendo nenhum livro à Biblioteca). Isso pode ser feito utilizando cláusulas ou predicados booleanos antes do nome da mensagem, como na Figura 5, em que a mensagem 1 tem o predicado [novo] como prefixo. Nesse caso, a mensagem só é executada caso a condição contida no predicado seja verdadeira. Analogamente, a mensagem 3 só será executada se a condição [apto] for verdadeira. Pode-se utilizar, dentro dos predicados, nomes de variáveis criadas anteriormente no diagrama de colaboração, ou passadas como parâmetros por outros métodos.

Operadores lógicos e relacionais podem ser utilizados para combinar várias condições, por exemplo [naoEstaEmAtraso] and [nroLivros < maximoPermitido] (Figura 7). Pode-se desviar o fluxo da execução das mensagens para um método ou outro, simulando um comando condicional, colocando-se [condição] em um deles e [not condição] no outro (Figura 8).

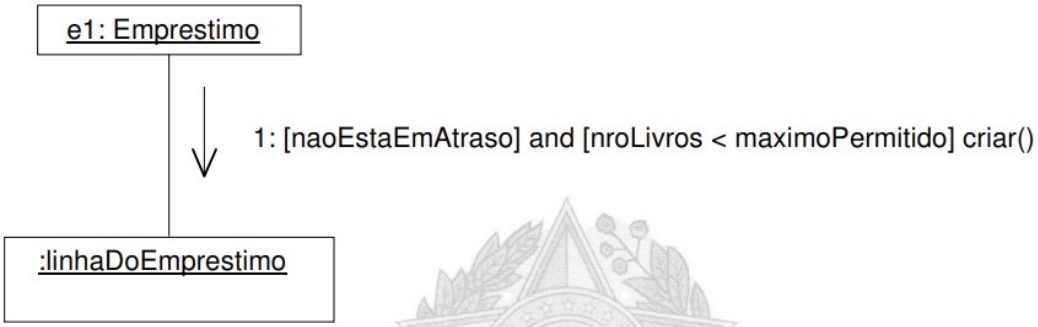


Figura 7 – Operadores e condicionais

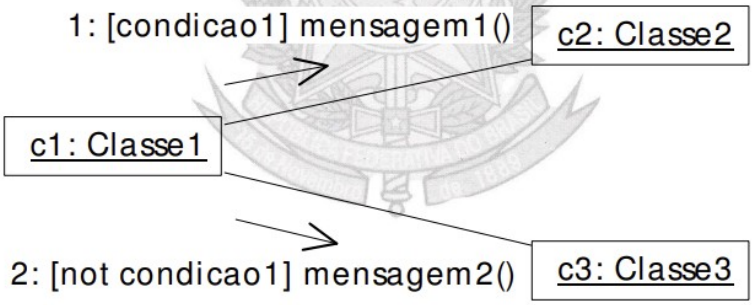


Figura 8 – Condicionais opostas

1.1.5 – Repetição de mensagem

Em alguns casos, uma mensagem pode precisar ser enviada várias vezes ao objeto, por exemplo em um laço. Isso pode ser denotado pelo uso de palavras chave tais como para cada, repita, etc., conforme ilustrado na Figura 9. Deve-se notar que a mensagem1 é enviada várias vezes, sempre ao objeto c2. Caso não seja possível dizer exatamente quantas vezes a mensagem é enviada, pode-se usar o símbolo *.

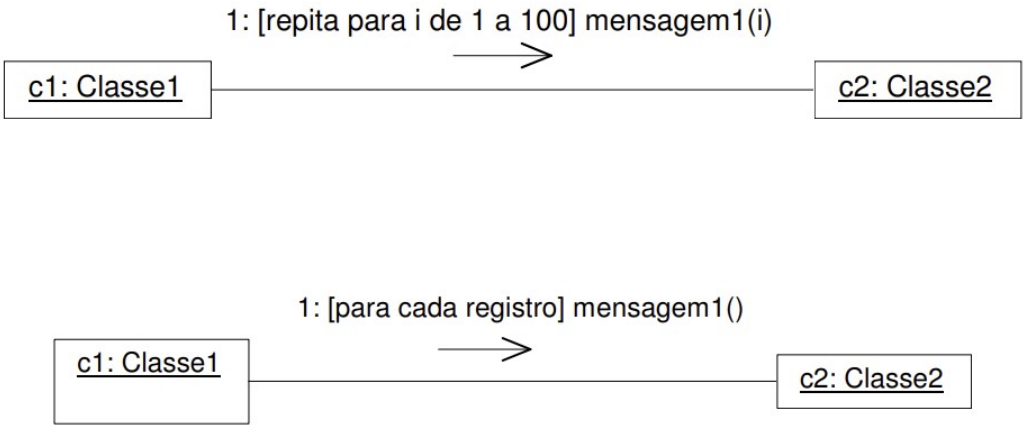


Figura 9 – Repetição

1.1.6 – Mensagem para coleção de objetos

Uma mensagem pode ser enviada a vários objetos, ou seja, para uma coleção de objetos, que denominamos um multiobjeto. Por exemplo, para obter os títulos dos livros emprestados pelo leitor em uma certa data, deve-se percorrer todas as linhas de empréstimo do empréstimo em questão, obtendo de cada uma delas o título do livro correspondente.

Para denotar um multiobjeto em UML desenha-se um retângulo com várias sombras, como mostrado na Figura 10. Deve-se notar que a mensagem obterTituloDoLivro é enviada uma única vez para cada um dos objetos linhaDoEmprestimo que pertencem à coleção de linhas de empréstimo do empréstimo atual.

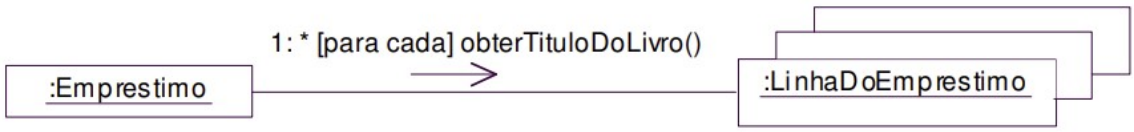


Figura 10 – Mensagem para coleção de objetos

A Figura 11 mostra um exemplo mais elaborado, no qual cada título de livro obtido é armazenado em uma outra coleção, neste caso uma coleção de strings cujo nome é titulosEmprestados. Ela é inicialmente criada (coleção vazia), por um método bastante comum em diagramas de colaboração, o criar. Para percorrer a coleção de linhas do empréstimo, usa-se um outro método comum na modelagem de diagramas de colaboração, chamado próximo, para obter o próximo elemento da coleção, que é atribuído à variável linha. O objeto linha pode então ser invocado para responder o título do livro, que é finalmente adicionado à coleção titulosEmprestados. Isso se repete enquanto houverem linhas de empréstimos a serem percorridas na coleção.

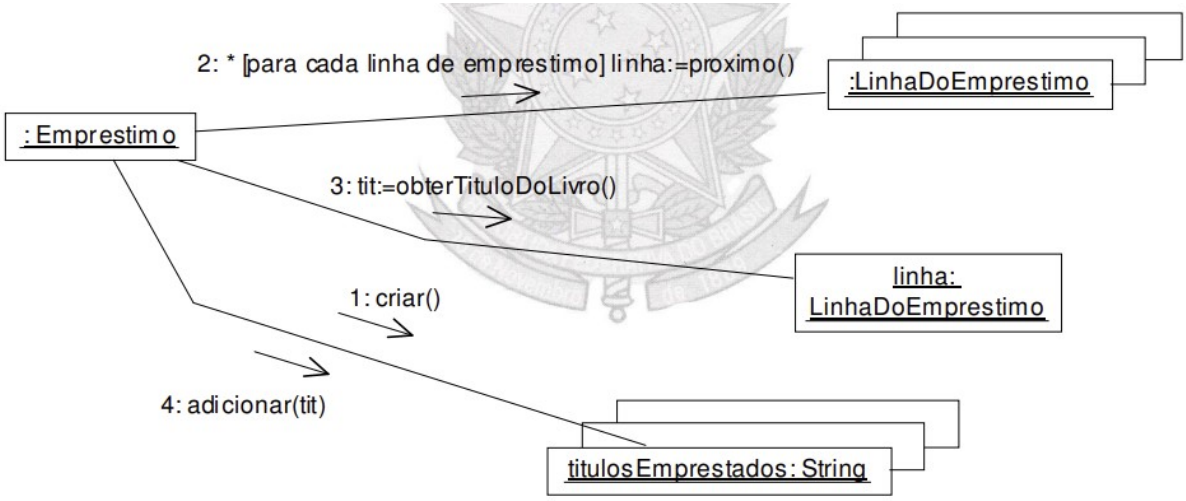


Figura 11 – Como percorrer uma coleção de objetos

Uma observação importante é que o JUDE Community não possui a opção para criar multiobjetos.

1.1.7 – Mensagem para o próprio objeto

Uma mensagem pode ser enviada de um objeto para ele mesmo (automensagem), denotando que o próprio objeto entende e processa a mensagem. Por exemplo, na Figura 12, o objeto l1, da classe Livro, recebe a mensagem ehDeConsulta, processa-a e armazena seu resultado na variável boolena cons, que depois é utilizada como condição para mudar ou não a situação da cópia do livro.

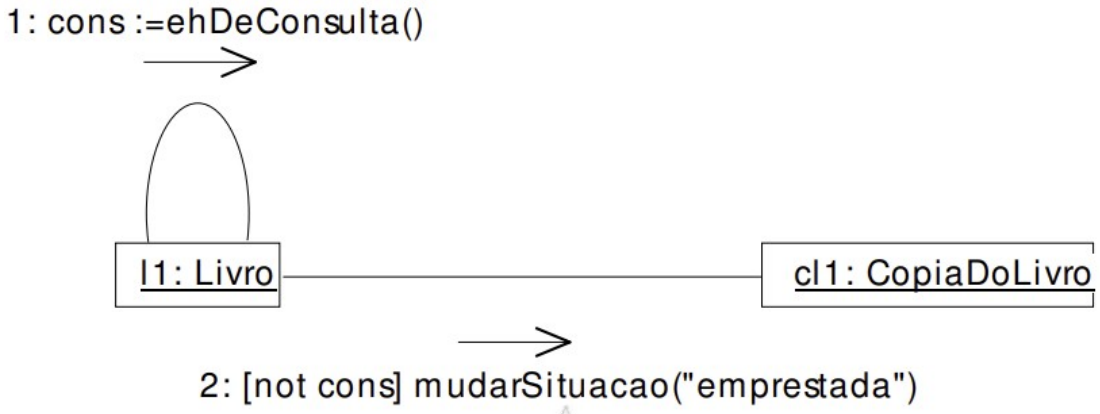
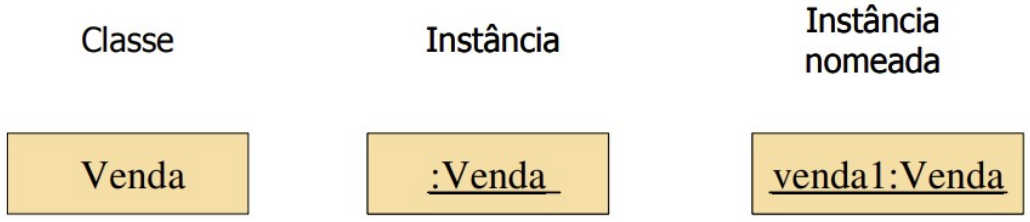


Figura 12 – Auto-Mensagem

1.1.8 – Classes x Instâncias

Na maioria das vezes, o diagrama de colaboração mostra troca de mensagens entre objetos das classes, ou seja, um objeto envia uma mensagem a um outro objeto que ele conhece. Porém, pode ser necessário enviar uma mensagem à classe propriamente dita, ao invés de a um objeto. Por exemplo, pode ser necessário saber o nome da classe, ou quais são seus atributos. Nesse caso, a notação é a mostrada mais à esquerda na Figura 13.



A notação utilizada nas seções anteriores foi a de instância, que pode ser tanto uma instância sem nome quanto uma instância nomeada. Em geral, uma sugestão básica é dar nome à instância sempre que for utilizá-la posteriormente no mesmo diagrama de colaboração, por exemplo, se for passar o objeto como parâmetro para invocação de um outro método. Outra utilidade em nomear a

instância é facilitar a compreensão do projeto OO quando ele tiver que ser comparado ao código-fonte produzido. Instâncias nomeadas provavelmente serão transformadas em variáveis pelo programador que implementará o projeto. Dessa forma, o programador não precisará inventar o nome da variável, ou seja, utilizará o nome de instância fornecido pelo diagrama de colaboração.

2 – Atribuição de Responsabilidades

2.1 – Tipos de responsabilidade

Na seção anterior foi apresentada em detalhes a notação dos diagramas de colaboração.

Sabemos que os objetos precisam se comunicar para atingir seus objetivos, ou seja, cumprir suas responsabilidades. Os Diagramas de colaboração mostram escolhas de atribuição de responsabilidade aos objetos. Mas como decidir quem é o melhor candidato para realizar cada uma das operações ou métodos do sistema?

Comecemos definindo o que é uma responsabilidade: é um contrato ou obrigação de um tipo ou classe. Pode-se também pensar que uma responsabilidade é um serviço fornecido por um elemento (classe ou subsistema). Nesse sentido, há dois tipos de responsabilidades básicas: Fazer e Saber.

A responsabilidade do tipo fazer implica na execução de algo, por exemplo, criar um objeto, calcular um valor, atribuir um valor a um atributo, iniciar ações em outros objetos (delegação), coordenar e controlar atividades em outros objetos. Já a responsabilidade de saber implica em conhecer dados privados encapsulados, conhecer objetos relacionados ou conhecer dados que podem ser derivados ou calculados.

Para projetar um bom diagrama de colaboração, decisões de projeto precisam ser tomadas pelo engenheiro de software. Nesse ponto, a qualidade do projeto depende de um bom entendimento das técnicas e princípios de atribuição de responsabilidade. Várias soluções para um mesmo problema podem ser encontradas por diferentes engenheiros de software. Como saber qual delas é melhor?

2.2 – Problemas causados quando a distribuição de responsabilidades é ruim

Para entender melhor os problemas causados por um mau projeto, considere o diagrama de colaboração para a operação “emprestarCopia”, mostrado na Figura 14. O que podemos notar de errado nesse diagrama?

Se analisarmos a interação entre os objetos, existe uma forte interação entre o objeto da classe Biblioteca e outros objetos do sistema. Pode-se dizer que toda a lógica está concentrada neste objeto, ou seja, ele coordena a operação, chamando métodos de várias classes.

O problema com esse tipo de solução é que o objeto biblioteca fica muito complexo, pois possui a responsabilidade de se comunicar com muitos objetos e acaba realizando tarefas que deveriam ser de responsabilidade de outros objetos. Por exemplo, é ele quem cria o

ItemDeEmprestimo, é ele quem associa o Empréstimo ao ItemDeEmprestimo, é ele quem associa o ItemDeEmprestimo à cópia do livro, etc.

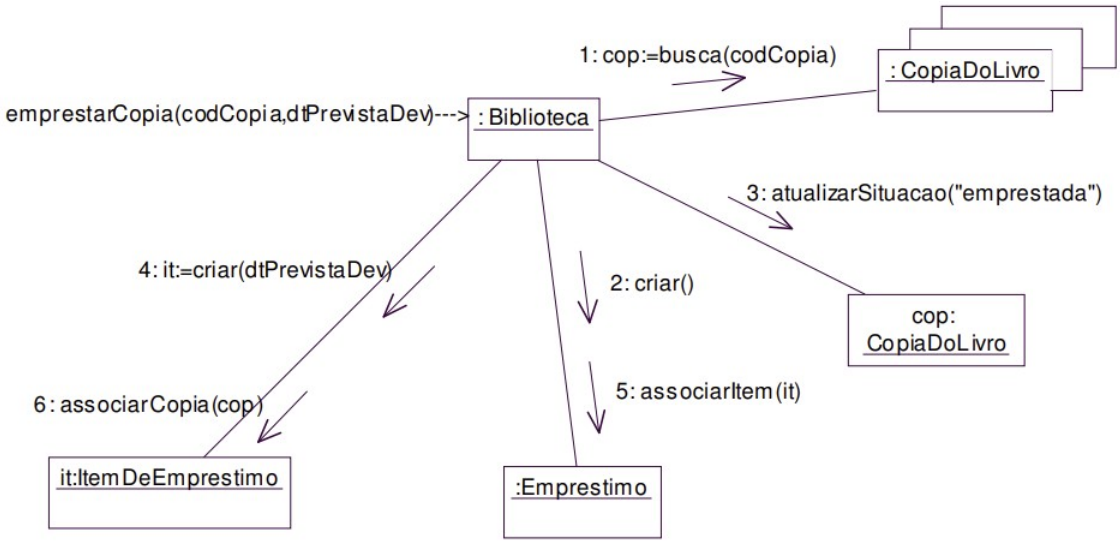


Figura 14 – Possível diagrama de colaboração para a operação emprestarCopia

Um projeto como esse pode ser muito difícil de entender e modificar, além de ter menor chance de ser reutilizado em outros desenvolvimentos, pois algumas classes possuem responsabilidades em excesso, muitas das quais não relacionadas ao propósito da classe em si. Para evitar esse problema, é necessário utilizar princípios bem aceitos na comunidade de orientação a objetos, conhecidos como princípios de atribuição de responsabilidades. Esses princípios podem nos guiar na atribuição de responsabilidades, levando a projetos melhores.

Referências

[Alexander 77] Christopher Alexander et. al., A Pattern Language, Oxford University Press, New York, 1977.
[Alexander 79] Christopher Alexander, The Timeless Way of Building, Oxford University Press, New York, 1979.
[Appleton 97] Appleton, Brad. Patterns and Software: Essential Concepts and Terminology, disponível na WWW na URL: http://www.cmcrossroads.com/bradapp/docs/patterns-intro.html
[Bauer, 2004] Bauer, Christian; King, Gavin. Hibernate in Action, Manning Publications.
[Beck 87] Beck, Kent; Cunningham, Ward. Using Pattern Languages for Object-Oriented Programs, Technical Report nº CR-87-43, 1987, disponível na WWW na URL: http://c2.com/doc/oopsla87.html
[Booch, 1995] Object Solutions : Managing the Object-Oriented Project (Addison-Wesley Object Technology Series by Grady Booch , Pearson Education; 1st edition (October 12, 1995)
[Buschmann 96] Buschmann, F. et at. A System of Patterns, Wiley, 1996. [Coad 92] Coad, Peter. Object-Oriented Patterns. Communications of the ACM, V. 35, nº9, p. 152-159, setembro 1992.
[Coad 95] Coad, P.; North, D.; Mayfield, M. Object Models: Strategies, Patterns and Applications, Yourdon Press, 1995.
[Coleman et al, 1994] Coleman, Derek et al. Object Oriented Development - the Fusion Method, Prentice Hall, 1994.
[Coplien 92] Coplien, J.O. Advanced C++ Programming Styles and Idioms. Reading-MA, Addison-Wesley, 1992.
[Coplien 95] Coplien, J.; Schmidt, D. (eds.) Pattern Languages of Program Design, Reading-MA, Addison-Wesley, 1995.
[Deitel, 2002] Deitel, Harvey M; Deitel Paul J. JAVA – como programar, Editora Bookman, 4a Edição.
[Gamma 95] Gamma, E.; Helm, R.; Johnson, R.; Vlissides, J. Design Patterns - Elements of Reusable Object-Oriented Software. Reading-MA, Addison-Wesley, 1995.
[Goodwill, 2002] Goodwill,J. Martering Jakarta Struts, Wiley Publishing, Inc.
[Krutchen, 2000] Krutchen, Philippe. The Rational Unified Process, An Introduction, Second Edition, Addison Wesley, 2000.
[Larman, 2004] Larman, Craig. Utilizando UML e Padrões, 2a edição, Bookman, 2004.
[Rational, 2000] RATIONAL, C. Unified Modeling Language. Disponível na URL: http://www.rational.com/uml/references, 2000.
[Rumbaugh, 1990] Object-Oriented Modeling and Design by James R Rumbaugh, Michael R. Blaha, William Lorensen, Frederick Eddy, William Premerlani, Prentice Hall; 1st edition (October 1, 1990)
[Waslawick, 2004] Waslawick, Raul. Análise e Projeto de sistemas de Informação Orientados a Objetos, Campus, 2004.