Notas da Aula 01 e 02: Proposta de Projeto
Prof. Daniel Caetano

Objetivo: Propor o Projeto a ser desenvolvido na disciplina e selecionar competências dos alunos.

Introdução

Objetivo principal do curso de projetos é produzir o projeto de um software, na forma mais completa possível. O curso é dividido em duas partes: a primeira trata da análise de sistemas e a segunda trata do projeto propriamente dito. Estas fases não são, como pode parecer a princípio, desvinculadas. Existe uma íntima ligação entre elas e isto deverá ficar mais claro ao londo do curso.

Tanto a análise quanto o projeto serão desenvolvidos com base em uma metodologia mista, adotando elementos de diversas metodologias diferentes como RUP e ICON, e deverão resultar em um projeto orientado a objetos. A documentação será produzida segundo as regras da UML e, portanto, esta será vista/revisada neste curso.

Há duas possibilidades para este curso, com desafios diferentes: a primeira é de um único projeto para a turma toda e a segunda é um projeto diferente para cada grupo de 4 ou 5 alunos. No primeiro caso, as maiores dificuldades são no aspecto **gerencial**, ou seja, as tarefas de análise e projeto individuais passam a ser reduzidas... porém gerenciar "quem faz o quê" é tarefa do grupo (e não do professor) e nem sempre é algo simples. No segundo caso, a divisão de tarefas continua sendo uma atividade de obrigação grupo, mas é de monta bem mais simples (dado o menor número de pessoas). Por outro lado, como o número de pessoas para cada projeto é menor, a quantidade de **trabalho individual aumenta**.

Vale ressaltar que a cobrança da disciplina será basicamente sobre a documentação do software (análise e projeto) e não sobre o produto. Se a(s) equipe(s) desejar(em) implementar o produto, ótimo; isso impactará positivamente na nota de apresentação. Por outro lado, a nota do trabalho escrito será indiferente à existência da implementação.

Adicionalmente, as etapas a serem executadas para obter o projeto serão melhor detalhadas aula a aula, o que não significa que os grupos precisem esperar, caso já saibam o que devem fazer. Dependendo do(s) projeto(s) selecionado(s), o trabalho maior pode estar no começo, no meio ou no fim da disciplina. Entretanto, a aplicação individual de todos é de extrema importância para que um bom resultado seja obtido ao final da disciplina.

Haverá o preenchimento de uma ficha de competências, mas isso é meramente indicativo. Todos deverão trabalhar em quase todas as etapas da análise e projeto.

Um Problema para Modelar

Como na vida real, este será um problema razoavelmente complexo. Sendo assim, existirão os mais diversos tipos de atividades. Nesta aula veremos um esboço do problema a ser solucionado. O **professor** faz o papel de cliente, e os alunos todos serão responsáveis pelo produto.

História

Eu pretendo abrir uma nova pizzaria, voltada basicamente ao "delivery", ou seja, entrega em casa. Entretanto, pretendo que o novo negócio seja totalmente informatizado, para evitar situações indesejáveis como a entrega de uma pizza incorreta. Resumidamente, os **principais** (mas não todos) requisitos do sistema são:

a) Permitir venda de Pizza para os clientes atravás de terminal na pizzaria;
b) Permitir venda de Pizza para os clientes através do atendente de telefone;
c) O sistema deve vender bebdias também;
d) O sistema deve identificar o usuário pelo telefone ou CPF;
e) Permitir que o gerente crie novos tipos de Pizza ou refrigerantes no cardápio.
f) Permitir que o gerente adicione novos ingredientes, para poder criar novas Pizzas.
g) Permitir que o gerente indique preços aos ingredientes para que eles componham o preço da Pizza. Alternativamente, o gerente pode especificar o preço da Pizza diretamente.
h) O gerente pode desligar um tipo de Pizza, sem ter de apagá-lo do cardápio.
i) O gerente deve poder consultar o histórico de pedidos global ou por cliente.

Seria também interessante que o cliente pudesse realizar modificações em uma pizza pedida (como tirar cebolas ou queijo), mas este não é um requisito obrigatório.

Caso seja programado, o programa principal **deve** ser em **java** e o banco de dados **deve** ser o MySQL. O prduto deverá rodar em plataforma **Linux e Windows**.

Ficha de Competências

Essa ficha deve ser preenchida por cada aluno, completamente. As notas variam de 1 a 5, sendo que 1 representa o menor grau de conhecimento deste assunto e 5 representa o maior graude conhecimento sobre o assunto:

Nome: _____

1. Fazer gráficos e editar imagens.	1.( )	2.( )	3.( )	4.( )	5.( )
2. Criar e editar textos .	1.( )	2.( )	3.( )	4.( )	5.( )
3. Organizar informações.	1.( )	2.( )	3.( )	4.( )	5.( )
4. Bom gosto visual (cores, formas...).	1.( )	2.( )	3.( )	4.( )	5.( )
5. Criação e edição de páginas Web.	1.( )	2.( )	3.( )	4.( )	5.( )
6. Fazer diagramas e fluxogramas.	1.( )	2.( )	3.( )	4.( )	5.( )
7. Raciocinar logicamente.	1.( )	2.( )	3.( )	4.( )	5.( )
8. Trabalhar com redes de computadores.	1.( )	2.( )	3.( )	4.( )	5.( )
9. Trabalhar com programação visual.	1.( )	2.( )	3.( )	4.( )	5.( )
10. Desenvolver Bancos de Dados	1.( )	2.( )	3.( )	4.( )	5.( )
11. Desenvolver Interfaces Gráficas com o Usuário.	1.( )	2.( )	3.( )	4.( )	5.( )
12. Desenvolver Sistemas para a Web.	1.( )	2.( )	3.( )	4.( )	5.( )
13. Programar.	1.( )	2.( )	3.( )	4.( )	5.( )
14. Descrever e modelar negócios.	1.( )	2.( )	3.( )	4.( )	5.( )
15. Trabalhar com produção de documentação.	1.( )	2.( )	3.( )	4.( )	5.( )

Ficha de Competências

Essa ficha deve ser preenchida por cada aluno, completamente. As notas variam de 1 a 5, sendo que 1 representa o menor grau de conhecimento deste assunto e 5 representa o maior graude conhecimento sobre o assunto:

Nome: _____

1. Fazer gráficos e editar imagens.	1.( )	2.( )	3.( )	4.( )	5.( )
2. Criar e editar textos .	1.( )	2.( )	3.( )	4.( )	5.( )
3. Organizar informações.	1.( )	2.( )	3.( )	4.( )	5.( )
4. Bom gosto visual (cores, formas...).	1.( )	2.( )	3.( )	4.( )	5.( )
5. Criação e edição de páginas Web.	1.( )	2.( )	3.( )	4.( )	5.( )
6. Fazer diagramas e fluxogramas.	1.( )	2.( )	3.( )	4.( )	5.( )
7. Raciocinar logicamente.	1.( )	2.( )	3.( )	4.( )	5.( )
8. Trabalhar com redes de computadores.	1.( )	2.( )	3.( )	4.( )	5.( )
9. Trabalhar com programação visual.	1.( )	2.( )	3.( )	4.( )	5.( )
10. Desenvolver Bancos de Dados	1.( )	2.( )	3.( )	4.( )	5.( )
11. Desenvolver Interfaces Gráficas com o Usuário.	1.( )	2.( )	3.( )	4.( )	5.( )
12. Desenvolver Sistemas para a Web.	1.( )	2.( )	3.( )	4.( )	5.( )
13. Programar.	1.( )	2.( )	3.( )	4.( )	5.( )
14. Descrever e modelar negócios.	1.( )	2.( )	3.( )	4.( )	5.( )
15. Trabalhar com produção de documentação.	1.( )	2.( )	3.( )	4.( )	5.( )

Etapa 1: Metodologia de Pesquisa
Prof. Daniel Caetano

Objetivo: Apresentar os principais conceitos em metodologia de pesquisa, ou seja, como realizar uma pesquisa e como apresentar os resultados desta pesquisa.

Introdução

Para muitos, a palavra **"pesquisa"** parece não exigir maiores explicações. Segundo o dicionário Aurélio Buarque de Holanda Ferreira, **pesquisar é o ato de "buscar com diligência, inquirir, investigar".**

Entretanto, apesar do praticamente todas as pessoas saberem intuitivamente **"o que" é pesquisa**, o **"como" se faz pesquisa** nem sempre é tão óbvio. Muitas pessoas acreditam que pesquisar é simplesmente **pegar a maior quantidade de volumes possível** e lê-los todos, buscando algo de interessante. Em alguns casos isso pode até ser divertido ou interessante, mas não é necessariamente uma pesquisa eficiente. Além disso, **adquirir um dado conhecimento não é suficiente**; **é necessário registrá-lo, também**.

Para que se possa obter o **máximo de resultado e qualidade** de uma pesquisa, alguns conceitos devem ser compreendidos e **alguns critérios devem ser seguidos**. O objetivo destas aulas é, então, transmitir estes conceitos e critérios, que passam a constituir uma **metodologia de pesquisa**.

Assim, seguindo os conceitos e critérios apresentados, é possível organizar os pensamentos mais facilmente, colocar estas idéias no papel e até mesmo comunicar suas descobertas e avaliações adequadamente no meio científico.

1. Objetivo de uma Pesquisa

A primeira pergunta que muitos se fazem é, certamente, **"para que se faz pesquisa?".** Esta é uma pergunta que tem muitas respostas; Na realidade, dependendo do tipo de pesquisa, a razão pode ser diferente. De forma geral, entretanto, pode-se dizer que o objetivo de toda pesquisa é **produzir uma contribuição** - algumas vezes inovadora - para a ciência ou para a sociedade.

É possível ainda dizer que uma pesquisa deve **responder a uma pergunta que seja de relevância** para a comunidade científica, para a sociedade ou que ainda não tenha sido respondida anteriormente. Sob esta óptica, talvez a parte mais difícil de uma pesquisa seja **encontrar qual a pergunta correta**.

O **produto da pesquisa é, então, um documento** - artigos, livros, dissertações ou teses - que responda à esta pergunta, com todo o detalhe necessário para a compreensão da resposta, para que qualquer pessoa consiga entender não só a própria pergunta, mas também a resposta fornecida. Por outro lado, é preciso ter em mente que **detalhes desnecessários** apenas prejudicam a clareza do trabalho, sendo **dispensáveis**.

Finalmente, a produção deste documento é uma atividade realizada de forma cooperativa e coordenada. Independentemente da área de pesquisa, **existem regras** nas comunidade científica, que implica no uso de citações, por exemplo, sob pena de **processo por plágio**.

1.1. Como Descobrir a Pergunta?

Descobrir a pergunta a ser respondida pode ser a **etapa mais difícil**, mas é a primeira que deve ser cumprida, já que todo o estudo a ser realizado e material a ser produzido depende desta. Assim, é interessante reformular a pergunta de uma maneira um pouco diferente: **qual é o problema que desejo investigar ou resolver?**

A idéia é **selecionar um assunto ou problema de vida real**, que possa ser **estudado, analisado e resolvido**. Não basta, entretanto, ser um problema qualquer: deve ser um **problema relevante e possível** de ser estudado e/ou resolvido. No caso de teses de doutorado, os problemas são ainda mais restritos: devem ser assuntos/problemas que ainda não tenham sido estudados ou resolvidos e, em geral, é aí que reside a maior dificuldade.

1.2. Tipos de Pesquisa

A forma de realizar uma pesquisa depende também do tipo de pesquisa e da área do conhecimento ao qual a pesquisa pertence. Quanto à **Natureza**, existem dois tipos de pesquisa (SILVA e MENEZES, 2001):

1) Pesquisa básica é aquela que visa avançar na ciência básica, descobrir novos fenômenos importantes, mas sem preocupação direta com aplicações práticas. Isso não quer dizer que o resultado da pesquisa não terá uso prático; significa apenas que a pesquisa **não** está sendo conduzida com base na finalidade prática. Um exemplo deste tipo de pesquisa é uma pesquisa sobre a composição dos buracos negros ou sobre a quantificação do volume do universo.

2) Pesquisa aplicada é aquela que tem um resultado prático visível, seja de importância econômica ou outra utilidade que não seja o próprio desenvolvimento do conhecimento. Um exemplo deste tipo de pesquisa é sobre o uso de resíduos agrícolas como adubo de solo.

É importante notar que existem outras classificações e tipos de pesquisa.

2. Como Fazer um Trabalho Científico?

Existem diversos tipos de trabalhos científicos, indo desde artigos até publicações para obtenção de grau acadêmico, sendo estes últimos o foco deste texto. Neste caso, existem **três categorias mais básicas**:

- Trabalho de Conclusão de Curso
- Dissertação de Mestrado
- Tese de Doutorado

O Trabalho de Conclusão de Curso serve para mostrar que o aluno tem conhecimento básico sobre técnicas de pesquisa, tem capacidade de aplicar aquilo que estudou em seu curso e sabe comunicar seus resultados de forma satisfatória. O portador do título Bacharel pode atuar em sua área como profissional capaz de assumir as responsabilidades correlacionadas.

A Dissertação de Mestrado serve para mostrar que o aluno domina as técnicas de pesquisa e investigação, foi capaz de produzir um resultado relevante para a sociedade ou comunidade científica e soube comunicar seus resultados de maneira efetiva. O portador do título de Mestre pode ministrar aulas e orientar alunos de conclusão de curso e iniciação científica.

A Tese de Doutorado serve para mostrar que o aluno tem pleno conhecimento das técnicas de pesquisa, ampliou a fronteira do conhecimento em seu campo e soube comunicar sua contribuição de forma efetiva. Com excelência em sua área de atuação, o doutor pode também orientar alunos de mestrado.

2.1. O que é Importante em um trabalho acadêmico e em um TCC?

Assim como na dissertação de mestrado e na tese de doutorado, o **TCC é o resultado da aplicação de uma metodologia**. Entretanto, diferentemente do mestrado e do doutorado, **a aplicação é mais importante que a inovação**. O TCC é o documento que **mostra a capacidade do aluno de aplicar - e ampliar - os conhecimentos desenvolvidos no curso como um todo**.

Além do caráter de avaliação do aluno, **o documento produzido** - como todo trabalho científico - **pode ser base para trabalhos futuros** e, portanto, enquadrando-se no aspecto colaborativo da produção científica. Por esta razão, o rigor na qualidade conceitural do trabalho precisa ser relativamente alto.

É importante ressaltar que a **forma** com que se comunica o trabalho, que deve ser **objetivo** e obedecer às **normas gramaticais**, com todos os procedimentos bem comentados.

2.2. Que cuidados o aluno deve tomar?

Primeiramente, é importante salientar que **não há regras que garantam sucesso**. Entretanto, existem algumas dicas que ajudam o desenvolvimento de um trabalho científico. Estas dicas seguem abaixo:

1) O aluno nunca deve deixar de anotar suas idéias. No momento em que se tem a idéia, ela pode parecer óbvia e que será facilmente lembrada no futuro, mas isso nem sempre é verdade e grandes idéias já foram perdidas por não terem sido anotadas.

2) O aluno é responsável pelo seu próprio trabalho. Em momento algum o aluno deve esperar que alguém faça algo por ele. Colegas prestam favor e o orientador orienta; mas a produção e a responsabilidade pela mesma é sempre do aluno.

3) O orientador não tem sempre a solução para seu problema. O aluno nunca deve ficar na esperança que o orientador vai surgir com alguma sugestão mágica que elimine todos os problemas. Muitas vezes o orientador sequer conhece o tema a fundo o suficiente. A função do orientador é direcionar na metodologia e, quando muito, realizar algum *brain-storm* com o aluno. Não é função do orientador fazer o trabalho para o aluno.

4) O aluno não deve tentar abraçar o mundo. Ninguém consegue explicar o mundo ou responder todas as perguntas do mundo em um único trabalho científico. Isso é verdade para uma dissertação, para uma tese e ainda mais para um trabalho de conclusão de curso. Perguntas dentro de seu tema e que continuem sem resposta ao final do trabalho, se forem realmente relevantes, podem ser apontadas como sugestões para futuras pesquisas no capítulo final do trabalho.

5) O aluno não deve nunca pensar que não há mais nada para pesquisar. Ainda que em alguns casos específicos isso possa ser verdade, estes casos são bastante raros. E é tão mais raro quanto menor for a graduação do aluno. É mais fácil "não haver nada mais a pesquisar", dentro de uma área específica, para um doutor do que para um aluno de graduação realizando um TCC.

6) O aluno não deve acreditar que já encontrará seu trabalho pronto. Se o aluno encontrar seu trabalho pronto, então cabe a ele ampliar o escopo deste trabalho, aprofundando-o ou ampliando-o ainda mais. Vale ressaltar aqui que os trabalhos "prontos" que o aluno por ventura encontre também serão encontrados pelo professor e isso implicará na recusa do trabalho.

7) Programa de computador não é trabalho científico. Apesar de servir como prova de conceito e possibilitar mostrar de forma concreta os resultados de algum estudo, o programa em si não é considerado o trabalho do aluno. É necessária a documentação do software desenvolvido, na forma de projeto e justificativa das decisões de projeto ou ainda na forma de um tutorial.

8) O aluno não deve deixar de mostrar a ligação entre teoria e prática. É muito importante - e torna o trabalho de mais valor - indicar como o conteúdo estudado e apresentado pode auxiliar na melhoria de vida das pessoas.

9) O aluno não deve esquecer o aspecto colaborativo. Escrever um trabalho que "esconde o jogo" não é uma boa prática. Tudo que é realizado como uma pesquisa ou trabalho acadêmico deve ser revelado de forma ampla e sem esconder nada, apresentando todos os detalhes e "truques".

10) O aluno deve manter o espírito crítico. Nunca se deve escrever um trabalho e "aceitar" que o mesmo está simplesmente perfeito. É necessário manter o espírito crítico não apenas quanto ao trabalho de outras pessoas, mas também quanto ao próprio. Eventuais críticas à metodologia, por exemplo, podem e devem ser feitas no capítulo de análise e conclusão, de preferência com sugestões de melhorias no método para futuras pesquisas.

2.3. Como o aluno deve se portar ao desenvolver um trabalho científico?

A atividade de pesquisa é uma busca racional pelo conhecimento. Assim, durante a atividade de pesquisa, o **aluno deve sempre buscar teorias explanatórias que possam ser**, dentro do possível, **generalizáveis**. Para isso, é importante **saber as implicações com relação às restrições que são feitas em seus modelos**.

É preciso também sempre **manter uma atitude receptiva**, mantendo a **mente aberta**. É preciso **perseverança e esforço para encontrar as "perguntas certas"** e tomar **cuidado com respostas prontas**. Um bom pesquisador **não deve buscar provar que sua teoria está correta, mas sim tentar derrubá-la**. É preciso tomar cuidado com "resultados prontos" e "frases feitas".

Para auxiliar neste processo, é interessante que o aluno mantenha sempre um **diálogo com seus colegas e orientador**, promovendo questionamentos sobre os assuntos estudados e teorias construídas.

2.4. O que o orientador espera do aluno?

Normalmente os alunos se perguntam o que é que seus orientadores desejam. Bem, os maiores desejos dos orientadores são, basicamente:

1) Que o aluno seja independente, ou seja, que pesquise e tenha idéias próprias.
2) Que o aluno documente bem seu trabalho, produzindo textos de boa qualidade.
3) Que o aluno seja o principal interessado no desenvolvimento do trabalho, não exigindo que o orientador fique caçando-o para que ele produza algo.

Para conquistar o orientador, é importante que o aluno **apresente resultados com freqüência**, ainda que sejam pequenos avanços. Qualquer **orientador gosta de estar**

informado sobre o estágio em que a pesquisa de seus alunos se encontra. Para isso, é interessante que ele receba **cópias preliminares do trabalho**, quando então ele pode fazer sugestões que auxiliarão no desenvolvimento do trabalho e de uma boa documentação.

2.5. O que não pode faltar no trabalho?

Primeiramente, é importante que o aluno ressalte **o que**, exatamente, será estudado e apresentado no trabalho. Isto deve ser feito na **Introdução** do trabalho, onde também deve ser apresentada **a motivação para o estudo**, lembrando aqui que motivação é algo como **"qual a relevância do estudo para a sociedade"** e não "qual a relevância do estudo para o aluno". Note que **o conteúdo trabalho não pode destoar do título**, já que o conteúdo do trabalho é avaliado primordialmente com relação ao título.

Ainda na introdução é interessante que o aluno **cite trabalhos que comprovem a importância do tema**, além de uma **descrição breve do que será visto em cada um dos capítulos** do trabalho, tornando clara a razão pela qual os capítulos são apresentados naquela ordem escolhida. Mas o aluno precisa tomar cuidado com o tamanho. Uma introdução de mais de três páginas é quase sempre inconveniente. O **objetivo da introdução é motivar o leitor** para o que vem em seguida, **não aborrecê-lo**. O ideal é uma introdução entre uma e duas páginas, no máximo.

Já nas **Conclusões**, é importante que o aluno **apresente resumidamente as informações que comprovam a relevância do tema** para a sociedade - que foi apresentada na introdução - assim como uma **opinião do aluno acerca do tema** e **como ele acredita que as informações por ele relatadas podem beneficiar trabalhos futuros**. A conclusão **não deve citar fatos que não estejam relatados no trabalho!**

4. O que é Avaliado no Trabalho Final: Dicas

A avaliação envolve inúmeros fatores, que vão **desde a escolha do tema até a qualidade de seu desenvolvimento**, **passando por erros de ortografia e gramática**. O ideal é que um trabalho científico, qualquer que seja, não possua qualquer tipo de erro.

Entretanto, existem alguns aspectos que **possuem maior peso** na avaliação. Podem ser citados:

1) Qualidade das referências;
2) capacidade de síntese;
3) capacidade crítica do aluno.

Enquanto as duas primeiras estão ligadas à apresentação de conteúdo já existente, a terceira refere-se, em geral, ao texto produzido pelo aluno. As próximas duas seções dão algumas dicas sobre a realização destas etapas.

4.1. Apresentado o Resultado da Pesquisa

Antes de mais nada, é preciso **expressar claramente o que foi pesquisado**. Um texto de uma pesquisa em que os leitores precisem recorrer a todos os autores originais para conseguir compreender o que ela diz é um texto mal feito. A principal dica aqui é que o **autor se coloque na posição do leitor**: como o leitor vai receber o que foi escrito? Lembre-se que o leitor pode ser leigo no assunto tratado!

Algumas dicas específicas seguem:

- Evitar produzir textos "colcha-de-retalhos", típicos de trabalhos da pior qualidade baseado na "técnica copiar-e-colar". O leitor experiente (como o professor) identifica rapidamente que esta "técnica" foi utilizada, devido às nítidas mudanças de estilo que ocorrem de um parágrafo para outro. Solução: tudo que o autor encontrar e quiser colocar em seu trabalho deve **ser re-escrito** com as próprias palavras do autor!

- Evitar falta de coerência entre as citações. Se elas dizem coisas opostas, o autor não deve inseri-las no texto como se elas concordassem. É importante manifestar conclusões conflitantes, quando elas existem, mas elas devem ser apresentadas de forma apropriada, deixando claro seu conflito. Uma forma de relatar informações conflitantes é, por exemplo: "Embora seja possível encontrar autores que digam que isso é X (FULANO, 19oo), há autores que discordam, dizendo que isso é Y (CICRANO, 19oo)".

- Evitar afirmações fora de contexto. Muitas vezes há informações de relativa importância espalhadas pelo texto que estão fora de seu lugar... ou simplesmente não são pertinentes ao trabalho realizado. No primeiro caso, elas devem ser realocadas para a posição correta e, no segundo, devem ser eliminadas.

- Evitar contar uma "história de guerra". O leitor nunca quer saber como o autor encontrou livros ou publicações, nem em que ordem isso foi feito. O leitor deseja uma síntese dos conteúdos em uma seqüência lógica e compreensível (e não seguindo a cronologia da pesquisa).

- Evitar "chover no molhado". Certamente, toda pesquisa tem um público alvo e este público alvo tem um certo conhecimento. Tentar explicar a Lei de Newton para um físico, a não ser em casos muito específicos, pode ser inadequado.

- Evitar usar advérbios de intensidade gratuitos. Muitos autores já fizeram afirmações adequadas se tornarem questionáveis ao acrescentar palavras como "impossível", "certamente", "sempre", etc.

4.2. Metodologia

Quando a pesquisa bibliográfica é apenas a fundamentação para o desenvolvimento de um trabalho experimental ou mesmo uma implementação de software, **é muito importante apresentar um capítulo chamado "metodologia".**

O capítulo **Metodologia é onde se descreve quais os procedimentos adotados** para a execução dos experimentos, avaliações ou mesmo a metodologia de projeto de software. O autor deve se conscientizar que no capítulo de metodologia **não devem aparecer os dados da pesquisa** que ele está desenvolvendo.

A idéia do capítulo "Metodologia" é exatamente **apresentar os procedimentos que serão realizados, independentemente dos dados que alimentarão estes procedimentos**. Ao invés de apresentar o código e diagramas de objetos na metodologia, é mais adequado apresentar a linguagem utilizada, método de avaliar o desempenho de uma solução etc.

A informação **dos dados utilizados para obtenção dos resultados deve estar presente**, mas em **outro local**. O local mais adequado pode variar:

1) Capítulo próprio: é bastante comum criar um capítulo chamado "Descrição" entre o de metodologia e o de resultados/análises.

2) Início do capítulo resultados/análises: usado quando a quantidade de informações de entrada é pequena frente ao tamanho da análise.

3) Anexo: dependendo da quantidade e forma dos dados, uma parte destes dado pode (e às vezes deve) estar presente nos anexos, a fim de não tornar a leitura do texto principal aborrecida com dados que nada acrescentarão ao leitor. Por exemplo: em um texto de uma pesquisa que envolva a implementação de um software, normalmente é mais adequado colocar descrição detalhada do código e informações sobre estruturas de dados nos anexos.

Um detalhe que convém ressaltar é que, quando se descreve a metodologia no trabalho final, já são conhecidos os resultados (dados). Desta forma, se algum resultado inesperado e/ou incoerente tiver sido encontrado, é interessante descrever como tratar estes tipos de "desvio" ainda no capítulo de metodologia.

5.Bibliografia

CÂMARA, G. *Notas de Aula do Curso Metodologia de Pesquisa.* Instituto Nacional de Pesquisas Espaciais, 2001.

SILVA, E.L; MENEZES, E.M. *Metodologia de Pesquisa e Elaboração de Dissertações*. Universidade Federal de Santa Catarina, 2001.

Etapa 1: Metodologia de Projeto
Prof. Daniel Caetano

Objetivo: Revisar os conceitos do Processo de Desenvolvimento de Software.

Bibliografia Básica:

- BEZERRA, E. **Princípios de Análise e Projeto de Sistemas com UML**. 2ed. Rio de Janeiro: Editora Campus Elsevier, 2007.

Introdução

A atividade de projetar um software é complexa, pois envolve uma enormidade de fatores do próprio software, do hardware, das interações entre as pessoas, do problema a ser resolvido, dentre muitos outros. O desempenho do profissional de projeto depende diretamente de sua experiência com projeto e quanto mais participar de atividades relacionadas, melhor este profissional estará preparado para lidar com estas complexidades.

Apenas para ilustrar a complexidade de desenvolvimento de um projeto real, serão apresentados resultados do Chaos Report, feito pelo Standish Group sobre projetos de desenvolvimento (CHAOS, 1994 apud BEZERRA, 2007):

- Porcentagem de projetos que terminaram dentro do prazo estimado: 10%
- Porcentagem de projetos que são descontinuados antes do fim: 25%
- Porcentagem de projetos acima do custo esperado: 60%
- Atraso médio nos projetos: 1 ano

É claro que sempre que iniciamos um projeto temos o desejo que este projeto seja concretizado com sucesso. Entretanto, como visto, o sucesso de um projeto envolve muitos fatores e as chances de fracassos são razoáveis. Os chamados *processos de desenvolvimento de software* ou *metodologias de projeto* trazem alguns métodos para auxiliar a lidar com a complexidade. Alguns destes processos conhecidos são o ICONIX, RUP, EUP, XP e OPEN (BEZERRA, 2007).

1. Etapas Típicas de uma Metodologia de Projeto

Praticamente todas as metodologias de projeto voltadas a grandes projetos (como a RUP) seguem quatro passos: análise, projeto, implementação e testes. Alguns autores acrescentam um quinto passo: a operação/vida útil/manutenção. Apesar de ser uma fase importante do ciclo de vida de um software, não iremos tratar desta fase nesta disciplina. Uma parte muito importante da análise, chamada *levantamento de requisitos*, que aqui será apresentada como uma etapa a mais, inicial.

Resumidamente, Levantamento de Requisitos é a fase em que se *aprende o problema*. Análise é a fase em que *soluções são propostas*. Projeto é a fase em que *uma solução é detalhada*. Implementação é a fase em que *se programa* e Testes é a fase em que *se busca "matar os últimos bugs*". Certamente não é comum que estas fases sejam seqüenciais, ou seja, uma começa quando a outra termina. Normalmente há uma certa sobreposição entre as fases e, em geral, ocorrem alguns ciclos de desenvolvimento (após os testes volta-se para a fase de análise/projeto a fim de solucionar problemas).

Veremos cada uma destas fases em maior detalhe a seguir.

2. Levantamento de Requisitos: Fase 1 da Engenharia de Requisitos

A primeira etapa do processo de desenvolvimento de software é aquela que permite com que os projetistas tomem profundo conhecimento do problema que precisam resolver. O objetivo fundamental desta fase é garantir que a visão do sistema por parte dos usuários e desenvolvedores é a mesma. Durante esta fase é elaborado um documento chamado "Documento de Especificação de Requisitos" que deve ser aprovado pelos clientes antes do projeto ter continuidade.

Nesta fase devem ser explicitados, basicamente, os *requisitos funcionais* (funcionalidades do sistema, como "o gerente deve poder adicionar uma pizza nova ao sistema, utilizando sua interface específica") e os *requisitos não-funcionais* (características de qualidade do sistema, como "confiabilidade: o sistema não pode perder informações de um pedido caso caia a energia elétrica"). Adicionalmente, devem ser explicitadas as características que alguns autores denominam de *requisitos normativos*, que definem restrições tecnológicas, regras de negócios, aspectos legais, etc.

É interessante também identificar a prioridade de cada requisito, considerando o valor que ele agrega ao sistema (interessante, desejável ou indispensável), bem como uma indicação de sua volatilidade (requisitos que poem mudar futuramente).

As metodologias de projeto mais usadas descrevem passos claros para que a análise de sistemas seja concluída com sucesso e que o Documento de Especificação de Requisitos seja completo. Os passos mais comuns são:

- **Análise de Viabilidade** - o sistema vai agregar valor ao cliente?

- **Passo I: Identificações** - qual o domínio do sistema: o que o cliente faz, como o sistema se insere em sua atividade, quem são os usuários, quais são as funções que o sistema precisa ter... lista de requisitos funcionais e não-funcionais.

- **Passo II: Análises** - classificação de requisitos em módulos coerentes, definição de prioridades das funções, resolução de conflitos.

- **Passo III: Documentação** - Criação da documentação incluindo todas as características detectadas e primeiros esboços de estrutura de sistema. Pode ser um único documento ou vários... variação do público alvo: cliente/desenvolvedores.

- **Passo IV: Validação** - Documento de Especificação de Requisitos passa pela validação do cliente. Resolução de conflitos.

A identificação de problemas nas fases iniciais de projeto é muito importante, pois quanto mais avançado o projeto está quando é descoberto um problema, maior é o custo para corrigi-lo.

É muito importante ressaltar que a maior parte dos sistemas que são descartados (muitas vezes com pouco ou nenhum uso) são aqueles em que a engenharia de requisitos falhou, ou seja, não identificou adequadamente todas as necessidades do sistema, tornando o sistema insuficiente e/ou ineficiente para o cliente, que preferiu manter o sistema antigo ou mesmo gastar mais dinheiro desenvolvendo um novo - com outra empresa, claro.

3. Análise de Sistemas: Fase 2 da Engenharia de Requisitos

Nesta etapa o objetivo é identificar os componentes do sistema e suas interações, com o objetivo de compreender como ele deve funcionar. Para isso, os *analistas* estudam detalhadamento o Documento de Especificação de Requisitos gerado na etapa anterior e propõem *modelos* para representar o sistema estudado.

Neste momento, ainda não há uma preocupação com aspectos tecnológicos da implementação. A única preocupação é representar adequadamente o problema, preocupando-se apenas com a estratégia de solução. Primeiro é preciso definir *o que* o sistema deve fazer, para depois definir *como* ele irá executar estas atividades.

Todos os modelos construídos nesta fase devem ser validados e verificados, para assegurar que as necessidades do cliente estão sendo atendidas (deve ser validado pelos usuários do sistema) e para garantir que o modelo atende aos requisitos definidos (verificando se o sistema está sendo construído corretamente), respectivamente. A atividade de validação inicia-se no levantamento de requisitos e finaliza na análise. A atividade de verificação começa na análise e termina no projeto.

O produto da fase de análise é um modelo de objetos que representa os sistema e também um modelo funcional do sistema (como ele deve funcionar).

Segundo Blaha e Humbaugh (2006, apud BEZERRA, 2007), a etapa de análise pode ser subdividida em duas partes: a análise de domínio e a análise de aplicação. A análise de domínio visa *identificar* os *objetos do mundo real* que devem ser processados pelo sistema, além de identificar as *regras* e *processos de negócio*. A etapa de análise de aplicação consiste

em identificar outros objetos que fazem sentido no domínio da aplicação, mas não para um especialista do negócio para o qual o sistema está sendo desenvolvido (como, por exemplo, um objeto 'janela de cadastro' ou algo do tipo).

4. Projeto de Sistemas

Nesta etapa, os esboços de estrutura de projeto são refinados e a estrutura principal do software é melhor definida. Nesta fase intensifica a verificação se os requisitos estão mapeados e representados adequadamente, respeitando os princípios de modularidade, para permitir a implementação parcial do sistema rapidamente (requisitos obrigatórios) e a adição de recursos posteriormente (requisitos importantes e desejáveis).

Nesta fase todos os detalhes necessários à implementação (como identificação de atributos, métodos e seus parâmetros, algoritmos dos métodos.) deve acontecer. A preocupação na etapa de projeto é em *como* implementara aquilo que foi definido na etapa de análise, objetivando a produção de uma *descrição computacional* do que o software deve fazer.

O projeto também pode ser dividido em duas atividades: o *projeto de alto nível* (arquitetura) e o *projeto de baixo nível* (detalhado). O projeto de arquitetura refere-se a distribuir as classes de objetos relacionadas em subsistemas (módulos) e distribuí-los pelos recursos de hardware disponíveis. Nesta etapa são normalmente utilizados os diagramas de implementação da UML.

É importante lembrar que esta estruturação merece cuidado especial para que o sistema seja composto de módulos adequados. É bastante útil, neste caso, seguir o conceito da arquitetura MVC (Model-View-Controller... Modelo-Visão-Controle), uma arquitetura que propõe que as estruturas de dados, a interface com o usuário e o modelo de negócio sejam implementados em camadas isoladas, de forma que elas tenham uma certa independência.

No projeto de baixo nível são definidos aspectos de colaboração de objetos para executar as funcionalidades necessárias, além de a definição da iterface com o usuário e projeto do banco de dados. Entram aqui também aspectos de concorrência de distribuição do sistema, mapeamento dos modelos para aterfatos de software e algoritmos a serem utilizados. Os diagramas mais comuns da UML aqui são o Diagrama de Classes, Diagrama de Casos de Uso, Diagrama de Interação, Diagrama de Estados e Diagramas de Atividades (BEZERRA, 2007).

5. Implementação de Sistemas

A implementação é a etapa que transforma a especificação produzida no projeto em um produto real. Em resumo, é a tradução dos diagramas gerados nas fases anteriores para uma linguagem de programação escolhida.

A atividade de programação é complexa e a boa programação exige prática, uma vez que envolve a aplicação de conceitos matemáticos, científicos e de engenharia. Além disso, um bom conhecimento sobre padrões de projeto e implementação podem ser úteis, evitando a perda de tempo com o desenvolvimento de algoritmos para os quais já se conhece uma solução muito boa (ou até mesmo a melhor solução possível).

O primeiro passo para a implementação é a escolha de um paradigma de programação e uma linguagem. O paradigma refere-se à programação estruturada, orientada a objetos, etc. Este paradigma não precisa ser o mesmo selecionado para o projeto, mas é interessante que seja. A escolha de paradigma delimita entre algumas linguagens, e uma escolha dentre elas precisa ser feita, se tal escolha já não foi feita nas etapas anteriores, de análise e de projeto. A escolha da linguagem deve ser feita levando em considerações características do projeto e atividades a serem implementadas; a escolha da linguagem pode seguir restrições impostas na análise de requisitos ou, quando não houver este tipo de restrição, deve ser feita com base na característica base da aplicação.

Na maioria dos casos existe uma linguagem mais adaptada ao tipo de aplicação sendo desenvolvida. Algumas destas relações são apresentadas a seguir:

-C/C++: para alto desempenho e propósitos gerais;
-Pascal/Delphi: para aplicações de rápido desenvolvimento e bom desempenho;
-Visual BASIC: para desenvolvimento de protótipos;
-LISP: para programas de inteligência artificial;
-FORTRAN: para programas de cálculo matemático puro e complexo;
-Java: para aplicações multiplataforma;
-...

6. Teste de Sistemas

Teste de Software é o processo pelo qual se verifica o quão correto, completo e seguro está um software, propiciando uma forma de avaliar a qualidade do mesmo.

O ato de testar é a realização de uma comparação crítica entre o que o software faz e o que ele deveria fazer. Assim, o teste ideal seria aquele que fosse capaz de avaliar todas as possibilidades e combinações de funções de um software. Entretanto, é matematicamente comprovado que há tipos de faltas impossíveis de serem identificados por processos automáticos e, portanto, a identificação destas ficam delegadas à inspeção humana. Como tal

inspeção também pode ser falha, a grande maioria dos autores sugere que não existe software isento de faltas.

Apesar de custosa, é importante a realização dos testes, já que é muito mais custoso para um desenvolvedor quando seu sistema falha nas mãos do cliente. Danos irreparáveis podem ser causados.

Os testes podem ser desenvolvidos desde as primeiras fases do projeto (na análise, por exemplo). Na fase de projeto propriamente dita, deve-se inclusive projetar os testes para cada parte do software, testes estes que serão realizados nesta fase de testes. Dependendo do software, até mesmo ferramentas de teste (test-suites) são desenvolvidas paralelamente.

O testes podem ser de três tipos:

- Caixa Branca - o aplicador dos testes tem acesso ao funcionamento interno do componente sendo testado;
- Caixa Preta - o aplicador dos testes **não** tem acesso ao funcionamento interno do componente sendo testado;
- Caixa Cinza - o aplicador dos testes **não** tem acesso ao funcionamento interno do componente sendo testado, mas **tem** acesso ao estado deste componente após cada operação.

Bibliografia

BEZERRA, E. **Princípios de Análise e Projeto de Sistemas com UML**. 2ed. Rio de Janeiro: Editora Campus Elsevier, 2007.

KRUCHTEN, P. **The Rational Unified Process**: an introducion. Addison-Wesley, 1998.

KONTONYA, G; SOMMERVILLE, I. **Requirements Engineering**: Processes and Techniques. Wiley, 1998.

DEITEL, H.M; DEITEL, P.J. **Java**: como programar - Sexta edição. São Paulo: Pearson-Prentice Hall, 2005.

MOLINARI, L. **Testes de Software**: Produzindo Sistemas Melhores e Mais Confiáveis. Editora Érica, 2005.

MOREIRA FILHO, T. **Projeto e Engenharia de Software**: Teste de Software. Alta Books, 2002.

Etapa 1: Metodologia de Projeto
Tópicos 3c e 3d: Elaboração de Cronograma e Análise de Requisitos
Prof. Daniel Caetano

Introdução

Os primeiros passos no processo do software são a elaboração de um cronograma e a elaboração de um documento de análise de requisitos. Estes dois tópicos serão cobertos nesta aula.

1. Elaboração de Cronograma

O cronograma é uma parte fundamental da gerência de um projeto, para acompanhar se o mesmo está seguindo de acordo com o esperado e se seu término ocorrerá, adequadamente, no prazo. Por esta razão, apesar de não ser uma atividade exatamente de projeto, iremos abordá-lo neste curso.

Um cronograma nada mais é do que um diagrama temporal que apresenta em que períodos cada uma das atividades serão cumpridas. Entretanto, o primeiro passo é identificar quais serão as atividades incluídas no cronograma. Para isso, será apresentada a metodologia WBS, que signigica "**Work Breakdown Structure**" ou, em português, EAP: Estrutura Analítica de Projeto.

1.1. Work Breakdown Structure / Estrutura Analítica de Projeto

A idéia da WBS/EAP é construir um **diagrama em árvore** que contenha na raiz o resultado final a ser obtido. O segundo nível contém os grandes blocos das partes que, juntas, componham totalmente o resultado final. No terceiro nível, cada atividade do segundo nível será dividida em várias partes que componham totalmente cada uma delas.

Esta é a regra mais importante do WBS/EAP: cada bloco de um nível superior deve estar ligado a vários blocos no nível seguinte (mais baixo), e a soma das partes à quais ele está ligado devem compor a totalidade do escopo definido por ele. Esta regra chama-se "**regra do 100%**", em que os "filhos" de um elemento precisam, juntos, representar este elemento pai.

Por exemplo: se em um nível há a parte "cadeira", ela deve estar associada a blocos "pés", "assento" e "encosto" no nível mais baixo. Os blocos do nível mais baixo não se sobrepõem e, em conjunto, constituem o elemento ao qual estão ligados no nível superior.

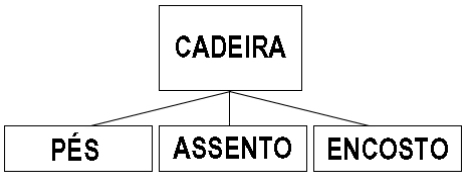


Figura 1: Exemplo de um WBS/EAP simples

Um detalhe importante é que o **WBS/EAP deve representar produtos** de atividades e não atividades em si. Assim, "Documentação" é um bom bloco de WBS/EAP, mas "Criação dos Documentos" não. É o que se chama de "Product Breakdown Structure.

É importante ressaltar que os **elementos de um mesmo nível** do WBS/EAP devem ser **mutuamente exclusivos**. Não se pode ter sobreposição entre eles.

Finalmente, o **nível de detalhe** a que se deve chegar depende do período das reuniões da equipe. Os "produtos" na camada mais inferior do diagrama devem ser tais que seja **possível completá-lo nos períodos entre as reuniões**. Se chegar a um nível de detalhe em que eles não possam ser subdivididos ainda mais, melhor. Adicionalmente, é interessante que sejam "produtos" que podem ser desenvolvidos sem dados adicionais aos que já foram obtidos/definidos na reunião anterior.

Em algum momento pode-se perceber que é possível criar diagramas bastante largos e pouco profundos, ou ainda bastante profundos e pouco largos. Como uma regra geral, **não se deve criar mais que 7 sub-elementos** para cada elemento do nível anterior. Isso costuma garantir que a árvore fique balanceada.

É importante lembrar que o WBS/EAP **não é** o cronograma e nem segue qualquer tipo de ordem cronológica, ele é apenas um meio de identificar as partes de um trabalho/produto que precisam ser concluídas.

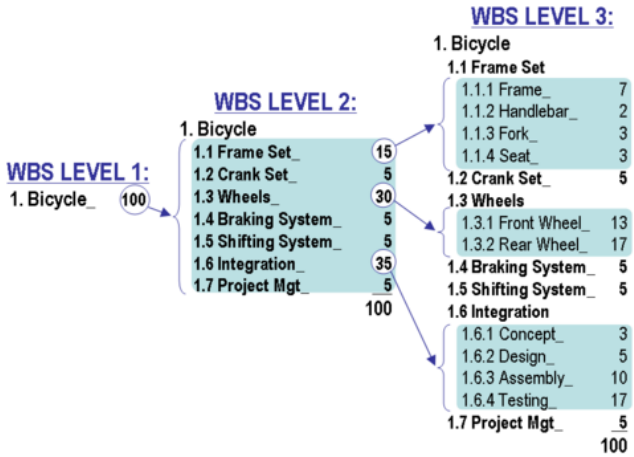


Figura 1: Exemplo de um WBS mais complexo, com indicação de porcentagens

1.2. Do WBS/EAP ao Cronograma

Com o WBS/EAP pronto, é necessário organizar as partes definidas no WBS/EAP em uma seqüência adequada à sua execução. Neste ordenamento, algumas partes precisam ser seqüenciais, outras podem ocorrer em paralelo.

Esta seqüência é a base do cronograma e os tempos estimados podem ser calculados de acrdo com os valores apresentados no WBS/EAP.

2. O que é Engenharia de Requisitos?

Após a decisão de que um sistema de software é necessário para auxiliar na resolução de algum problema, o primeiro passo para o desenvolvimento do mesmo é a **elaboração de um Documento de Requisitos**.

Um Documento de Requisitos é o material que descreve detalhadamente todas as características que o sistema de software deve possuir, sejam elas funcionais (coisas que o sistema tem de fazer) **ou não** (hardware onde ele deve ser executado, tempos de resposta, etc).

O **processo** da elaboração do Documento de Requisitos é **complexo** e, em alguns casos, longo e difícil. Isso ocorre porque tal documento deve ser **elaborado em conjunto pelos *stakeholders***, ou seja, todos aqueles envolvidos no projeto, incluindo contratantes do serviço, usuários, futuros administradores do sistema, projetistas, etc. Usualmente, cada uma destas partes tem conhecimento e domínio sobre uma das categorias de conhecimento necessárias para o correto funcionamento do sistema a ser desenvolvido, mas muitas vezes têm dificuldades em explicitar as reais necessidades do sistema com relação à área de conhecimento que representa. Em palavras simples: **o cliente nem sempre sabe dizer exatamente o que quer e como quer**.

Muitas **metodologias** têm sido desenvolvidas para auxiliar os analistas e projetistas de software a realizarem essa etapa com sucesso, sendo a **mais conhecida atualmente o UP** (Unified Process, nova nomenclatura da RUP, o Rational Unified Process). O processo de elaboração de um Documento de Requisitos, seguindo uma metodologia padronizada ou não, constitui o que se convenciono chamar de "Engenharia de Requisitos". **O processo da engenharia de requisitos é composto basicamente por quatro etapas: Identificações, Análises, Documentação e Validação.**

Neste curso não será analisada uma metodologia específica, mas sim os conceitos por trás de cada uma destas etapas. Tais conceitos regem todas as metodologias e são suficientes não só para nortear o processo de elaboração de bons documentos de requisitos, mas também para facilitar a compreensão e aprendizado futuro de metodologias específicas.

2.1. Antes da Engenharia de Requisitos

Como já dito anteriormente, o **processo da Engenharia de Requisitos pode ser longo e complexo**. Em muitos casos, isso significa também que o processo é **oneroso**. Desta forma, ao iniciar este tipo de processo, **é importante analisar se tal sistema é viável** e, ainda mais importante, **ter a certeza de que** ele **é**, de fato, **necessário**.

Esta etapa, anterior ao início do processo de engenharia de requisitos, é denominada de **Estudo de Viabilidade** e, usualmente, é feita através de **entrevistas e reuniões**. Tais entrevistas e reuniões visam verificar se existem **condições técnicas e administrativas** de se implementar o sistema em questão, além da **verificação de possibilidade de integração** do novo sistema com os já existentes.

Entretanto, a **verificação mais importante** a ser feita é quanto à **utilidade do sistema**, ou seja, se ele contribui, de fato, com os objetivos do cliente. Se um sistema é útil ao cliente, diz-se que ele **agrega valor**, e portanto justifica os investimentos necessários.

Nem sempre é fácil identificar as informações necessárias para responder a estas questões. Normalmente **quem possui** essas **respostas são os usuários e administradores** dos departamentos aos quais o sistema de software vai servir. **Alguns pontos** que se pode tentar **elucidar** são:

- Como o novo sistema irá contribuir com os objetivos do cliente?
- Como o serviço é feito pelos sistemas atuais? **Como um novo sistema poderia resolver os problemas do sistema atual?**
- Existe possibilidade do novo sistema integrar-se aos sistemas atuais?
- Quais são as alternativas que o cliente possui, caso o sistema não seja viável?

Este estudo deve possibilitar a **criação de um documento com restrições financeiras, tecnológicas e temporais** mais **explícitas**, além de já **definir** alguns dos **requisitos fundamentais do sistema**, como seu **objetivo principal**.

É importante lembrar que muitas vezes um novo sistema influencia e modifica a rotina de trabalho dentro de uma empresa e pode haver **resistência** a ele **por parte de algumas pessoas**, ainda que este sistema seja positivo. **É preciso levar em consideração** também essa questão ao analisar as informações.

2.2. Passo I: Identificações

A **primeira etapa** para o desenvolvimento de um bom sistema é **compreender o problema a ser resolvido**. Em outras palavras, isso significa compreender o domínio do sistema: o que o cliente faz, como o sistema a ser desenvolvido se insere em sua atividade.

Em seguida, deve-se **identificar** quem serão **os usuários** do sistema e consultá-los acerca das funções que o sistema deve desempenhar sob a óptica de cada um deles, resultando disso uma lista de **requisitos funcionais e não-funcionais**.

A identificação dos requisitos não é exatamente uma tarefa **simples**, mas pode ser realizada de diversas maneiras, como **entrevistas e questionários, discussão de cenários de uso, workshops, etc**. Entretanto, convém lembrar que cada uma das modalidades apresentam vantagens e desvantagens.

Além disso, **os clientes e usuários** podem ter **dificuldades em explicitar o que pretendem do sistema**, ou ainda podem apresentar **requisitos absurdos ou conflitantes**. Em muitos destes casos pode ser necessário complementar os estudos com a criação de protótipos e mesmo com estudos "in loco", ou seja, observando o dia-a-dia do cliente para identificar as suas necessidades reais sem que ele precise explicitá-las.

Finalmente, após uma clara definição dos requisitos, devem ser **verificados os possíveis problemas que podem surgir para atendê-los**. Devem ser **criadas soluções** para tais problemas e **apresentá-las para os clientes e usuários**. A **negociação** é um processo muito importante nesta etapa e o resultado final deve ser definido consensualmente.

2.3. Passo II: Análises

Uma vez que os requisitos estejam claramente definidos, é interessante **classificá-los em módulos**, facilitando uma visão de requisitos globais do sistema. Uma **definição de prioridades** também é importante, ressaltando quais **requisitos** são **obrigatórios**, quais são **importantes**, quais são **desejáveis, etc.**

Conflitos que surjam na etapa de análise **devem ser discutidos com os envolvidos no projeto**, incluindo clientes e usuários, **e solucionados o mais brevemente possível**. A cada definição feita, clientes e usuários devem ser consultados com relação à conformidade e consistência.

É importante notar que muitas vezes o fim da etapa de identificação e o processo de análise ocorrem com sobreposição, ou seja, ocorrem simultaneamente, cada uma das etapas servido de realimentação na execução de outras. Muitas vezes são necessárias algumas iterações (em ciclos) envolvendo as identificações e o processo de análise.

A necessidade destes **ciclos** pode existir por diversas razões, como **mudanças nas atividades** do cliente, **adição de necessidades** específicas dos clientes e usuários, ainda que não sejam diretamente relacionadas aos objetivos globais do sistema em si. Mais uma vez, a **negociação** é fundamental.

2.4. Passo III: Documentação

Finalizada as duas primeiras partes do processo de engenharia de requisitos, o terceiro passo é a **elaboração do Documento de Requisitos propriamente dito**.

O documento de requisitos pode ser **único** para todos os envolvidos **ou** pode ter **várias versões (requisitos de sistema, requisitos do usuário e projeto da aplicação)**. No caso de existirem vários Documentos de Requisitos, **uma versão "oficial" e única** deve também ser produzida, chamado de **Documento de Especificação de Requisitos**, contendo seções específicas para cada público alvo, desde clientes e gestores até engenheiros de teste e manutenção.

A **vantagem** de um **documento único** é a **facilidade de manutenção e consistência**, já que com **vários documentos** eles precisam apresentar informações coerentes. Entretanto, a utilização de documentos distintos para especificar requisitos de esferas distintas permite que **a linguagem utilizada** em cada um deles seja mais **adequada a seu público alvo**. Este é um aspecto bastante positivo quando os públicos têm domínios de conhecimento bastante distintos. Num controle de estoque, por exemplo, o documento que apresenta os requisitos do usuário pode descrevê-los segundo termos mais comuns aos profissionais da área de estoque, enquanto o documento de

requisitos de sistema e projeto podem tratar mais diretamente em linguagem comum aos envolvidos na elaboração e implementação do software em si.

Independente da forma que seja dada ao Documento de Requisitos, é fundamental que eles **contemplem os requisitos funcionais e não-funcionais de forma explícita**. Se existir mais de um Documento de Requisitos, cada um deles deve apresentar os requisitos funcionais e não-funcionais que lhe diz respeito.

Mas o que são requisitos funcionais e não-funcionais, explicitamente?

- **Requisitos funcionais**: uma descrição detalhada, completa e consistente de todas as funções que o sistema deve fornecer.
- **Requisitos não funcionais**: uma descrição completa das restrições sobre as quais o sistema deve operar, tempos de resposta, tolerância a falhas, etc.

Documento de Requisitos do Usuário

Primeiramente, é importante ressaltar que quando se fala em "requisitos do usuário" deve-se pensar no sentido amplo da palavra "usuário", referindo-se a todas as pessoas que terão contato com o sistema, seja para adicionar informações, consultá-las ou alterá-las.

Quando se cria um documento específico para este tipo de requisitos, é interessante que ele seja **em linguagem natural simples, com fórmulas e diagramas também simples**.

Esta abordagem traz consigo alguns **problemas**, como a **dificuldade em expressar de forma clara e com exatidão os requisitos**, além da dificuldade em **separar os requisitos do usuário entre funcionais e não-funcionais**.

Neste documento é **importante diferenciar** o que são **requisitos obrigatórios e desejáveis**, usando-se expressões como "**o sistema deve**" e "**é interessante que**", respectivamente, por exemplo.

Além disso, por ser utilizada uma linguagem simples, evitando termos técnicos, é importante utilizar recursos de realce de texto (itálico, sublinhado, negrito, etc) para ressaltar aspectos mais importantes, que não devem deixar de serem lidos pelo usuário.

Documento de Requisitos do Sistema

A **função** deste documento é similar à do **Documento de Requisitos do Usuário**. Entretanto, seu **público alvo** é outro: **os técnicos** que lidarão diretamente com **o projeto e implementação** do sistema.

Assim, não é mais necessário evitar **termos técnicos**, recorrendo sempre que possível a **linguagens algorítmicas e diagramas especiais** para especificar requisitos sempre que a linguagem natural for insuficiente para fazê-lo de forma adequada.

Diagramas como os de **Caso de Uso** devem ser usado amplamente, para evitar problemas de interpretação por parte dos diversos membros da equipe de desenvolvimento, sendo também **importante manter consistência com o documento de Requisitos do Usuário**.

Documento de Projeto da Aplicação

Este é um documento **também destinado à equipe técnica** e tem função de **apresentar uma visão geral de arquitetura do sistema** a ser projetado.

Não é um documento que **deve conter detalhes**: deve apresentar a **estrutura global**, **funcionalidades dos componentes e** permitir uma compreensão da **interação entre eles**.

Dentre as funções deste documento está apresentar aos programadores e projetistas uma visão do funcionamento do sistema, para facilitar a incorporação e novos projetistas e programadores à equipe de desenvolvimento.

Documento de Especificação de Requisitos

Existem diversas formas de apresentar um Documento de Especificação de Requisitos (abrevie como SRS, *Software Requirements Specification*, nunca como DER! DER é outra coisa!). A mais comum é aquela definida no **IEEE/ANSI 830, de 1993**.

Esta norma define cinco seções básicas, alguma delas subdivididas:

- Introdução
    - Os objetivos do documento e sistema;
    - a abrangência do documento e sistema;
    - definições e abreviações utilizadas;
    - referências utilizadas;
    - explicação das seções seguintes.
- Descrição
    - Panorama do produto a ser projetado;
    - funcionalidades do produto;
    - peculiaridades dos usuários;
    - restrições a que o sistema está sujeito;
    - considerações e dependências a que a implementação está sujeita.
- Requisitos Específicos
- Apêndices
- Índice

2.5. Passo IV: Validação

Criado o Documento de Especificação de Requisitos, é importante **verificar sua** validade, ou seja, sua **conformidade com as necessidades dos usuários e clientes**, considerando que sua viabilidade técnica já foi garantida durante o processo de elaboração do SRS.

Deve-se **buscar conflitos e inadequações às necessidades**, para que estas sejam corrigidas antes do início do projeto, já que, em alguns casos, mudanças nos requisitos podem implicar em grandes mudanças na estruturação do projeto, o que costumeiramente significa aumento de custos.

Os requisitos propostos devem atender completamente a todas as necessidades dos usuários e, de forma alguma, podem ser conflitantes. Neste aspecto, é interessante poder rastrear os responsáveis (clientes/usuários) responsáveis por cada requisito, tornando possível uma rápida resolução de eventuais conflitos. Além disso, os requisitos apresentados precisam estar em consonância com a realidade financeira e tecnológica disponível.

3. Modelo de um Documento de Especificação de Requisitos

Documento elaborado pelo professor Doutor Juliano Lopes de Oliveira:
http://www.caetano.eng.br/aulas/fb/proj3/modelo-especificacao-requisitos.pdf

4. Exemplo de um Documento de Especificação de Requisitos

Documento de especificação do software Whiteboard, elaborado por diversos membros, para a Fapesp:
http://www.caetano.eng.br/aulas/fb/proj3/whiteboard-requisitos-v2-0-3.pdf

Bibliografia

KRUCHTEN, P. **The Rational Unified Process - An introducion**. Addison-Wesley, 1998.

KONTONYA, G; SOMMERVILLE, I. **Requirements Engineering: Processes and Techniques**. Wiley, 1998.

DEITEL, H.M; DEITEL, P.J. **Java: como programar** - Sexta edição. São Paulo: Pearson-Prentice Hall, 2005.

ALMEIDA, R; BORBA, A.S.P. **Estudo do RUP** - Disponível em: < http://www.cin.ufpe.br/~phmb/RUP/MaterialDeEnsino/AnaliseEProjetoDaArquitetura.ppt > visitado em 29 de Março de 2007.

Etapa 2: Orientação a Objetos e UML
Tópico 4: Noções de Orientação a Objetos
Prof. Daniel Caetano

Objetivo: Revisar os conceitos de Orientação a Objetos e uma breve visão de um Diagramas de Classes.

Bibliografia:

- BEZERRA, E. **Princípios de Análise e Projeto de Sistemas com UML**. 2ed. Rio de Janeiro: Editora Campus Elsevier, 2007.
- JACOBSON, I; CHRISTERSON, M; JONSSON, P; ÖVERGAARD, G. **Object-oriented software engineering: a use case driven approach.** Essex, England: Addison-Wesley Longman Ltd, 1992.

- COAD, P; YOURDON, E. **Análise baseada em objetos.** Editora Campus, 1992.

Introdução

Apesar de nem sempre ser tão clara, segundo Bezerra (2007), a necessidade de realizar a modelagem e projeto de software é das mais importantes conforme aumenta sua complexidade. O exemplo usado por Bezerra (2007) é o da casa de cachorro. Provavelmente ninguém precisa de um projeto para construí-la. Basta "algumas ripas de madeira, alguns pregos, uma caixa de ferramenta e certa dose de amor por seu cachorro". Entretanto, quando se pretende construir uma casa ou um edifício, as coisas se complicam o suficiente para ninguém pensar em construí-las sem um projeto; na realidade, é até mesmo proibido realizar tais obras sem um projeto.

O objetivo principal do projeto é gerenciar a complexidade do problema reduzindo-os a modelos, usando o princípio das *abstrações*, que removem todas as características "não importantes" do problema, evidenciando apenas os dados relevantes. Os modelos auxiliam a comunicação de todos envolvidos no desenvolvimento, possibilitando que todos enxerguem o objeto a ser construído sob a mesma óptica.

Além disso, a realização do projeto permite antever problemas antes de partir para o trabalho de implementação propriamente dito, permitindo que problemas sejam corrigidos sem desperdício de recursos. Em outras palavras, o projeto evita que se construa uma parede para depois ter que derrubá-la.

Neste curso deve ser produzido um projeto baseado no paradigma orientado a objetos, usando a Unified Modeling Language para especificação. Esta aula introduz alguns dos conceitos necessários.

1. Orientação a Objetos

Orientação a objetos é um conceito de representação da realidade em que os componentes são objetos e não funções ou estruturas de dados. Em outras palavras, se nos modelos estruturados os componentes eram definidos de acordo com características intrínsecas à implementação, nos modelos orientados a objetos estes componentes se baseiam objetos (entidades) do mundo real.

- O mundo real é composto de objetos que interagem entre si.
- Um modelo orientado a objetos é composto de objetos que interagem entre si.

Da teoria de sistemas, temos que um sistema é um conjunto de entidades que interagem entre si a fim de produzir um resultado comum. Assim, é natural o uso de "objetos programa" a fim de compor um sistema computacional.

1.1. Como São os Objetos?

No mundo real, objetos podem ser animados ou inanimados, mas qualquer um deles possui carcterísticas que podem ser classificadas como atributos ou comportamentos.

Exemplos de objetos: átomos, veículos, vias, pessoas...

Isso faz com que exista uma diferença semântica muito pequena entre modelo e a realidade que ele representa, proporcionando maior clareza.

Vantagens principais:

- Concepção do sistema mais simples: a transição da *realidade* para o *modelo* é facilitada.
- Compreensão do modelo é simples: como o *modelo* é mais próximo da *realidade*, a compreensão do modelo por quem compreende o problema real é quase automática.
- Gerenciamento do sistema mais simples: assim como na realidade, os objetos são estáveis na solução de um problema, ou seja, os objetos mudam muito pouco; quando é necessário resolver problemas ligeiramente diferentes, modificamos a forma com que os objetos interagem e não os objetos em si.

Mas afinal, o que são objetos em programação?

Em programação (e, de certa forma também na vida real), um objeto é um ente caracterizado por um conjunto de operações e um estado, caracterizados por *métodos* e *campos*, podendo ainda ser compostos por outros objetos.

Note que a propriedade de um objeto poder ser composto de outros objetos também atende à teoria de sistemas, já que uma entidade que faz parte de um sistema pode ser ela mesma um *subsistema*. Da mesma forma, é uma característica que está em perfeito acordo com a realidade, visto que usualmente um objeto é composto de outros objetos (ex.: uma geladeira é composta de porta, prateleiras, caixa, motor, fios...). Os próprios seres vivos são compostos de elementos chamados *células*.

Note que um objeto é uma estrutura similar à uma "estrutura de dados"; porém, além de "dados", um objeto pode armazenar também "funções". Em um objeto os dados são chamados de *atributos* e as funções são chamadas de *métodos*.

Inicialmente estaremos trabalhando com "objetos de análise". Isso significa que esboçaremos o que objeto faz e o que ele armazena, mas não "como faz" ou "como armazena". Observe que estaremos respondento, então, à pergunta "O quê?" e não à pergunta "Como?". Deixemos o "Como" para a etapa de projeto.

Exemplos de objetos do mundo real:

TV	- Liga	- Canal
	- Desliga	- Volume
	- Muda canal	- Estado(ligada/desligada)
Carro	- Liga	- Cor
	- Desliga	- Velocidade
	- Acelera	- Quilometragem (odômetro)
	- Breca	- Portas

1.2. Conceitos da Modelagem e da Programação Orientadas a Objetos

Além dos objetos, os modelos orientados a objetos também se baseiam em outros conceitos, como os de classes, mensagens e associações, além das propriedades dos objetos.

CLASSES

Uma classe pode ser considerada como um "molde" de um objeto, sendo uma descrição de como um objeto pode ser criado. Uma forma interessante de explicar é que uma classe está para um objeto assim como a planta de uma casa está para a casa. Uma outra maneira de explicar é que se o objeto é um bolo, então a classe seria uma combinação entre a forma e a receita do bolo.

Exemplo:
Classe: Carro
Objetos: Carro vermelho, Carro azul, Ferrari etc.

MENSAGENS

Objetos são capazes de executar operações. Entretanto, estas operações não são ativadas de maneira aleatória. É preciso que um objeto receba um estímulo para executar uma operação. Este estímulo é chamado de *mensagem*. Em outras palavras, uma mensagem é a forma como um objeto se comunica com outro (ou estimula a outro). Essas mensagens normalmente são padronizadas, constituindo uma *interface*. Pense em uma língua comum que os objetos precisam saber para poder se comunicar. Para um ser humano entender um pedido de outro (ou seja, receber uma mensagem), é necessária uma *interface* comum.

ASSOCIAÇÕES

Como já foi visto anteriormente, um objeto pode ser composto de outros objetos diferentes. Quando objetos mais simples se unem para formar um objeto mais complexo, dizems que houve uma *associação de objetos.*

Exemplo: Carro = chassi + motor + acessórios + etc.

1.3. Principais Propriedades da Orientação a Objetos

As principais caracterísitas das classes de objetos constituem também as fundações do modelo orientado a objetos. Estas características são: encapsulamento, polimorfismo e a herança.

ENCAPSULAMENTO

É a propriedade que permite que um objeto seja tratado como uma "caixa preta". O interior do objeto, ou seja, "como" ele realiza as tarefas é invisível para os clientes daquele objeto. Os clientes só podem se comunicar com um objeto através da *interface* deste objeto, sendo que a interface de um objeto nada mais é do que a definição de quais mensagens ele "sabe" responder.

Exemplo: o carro é um objeto que pode ser tratado como uma caixa preta; uma pessoa pode dirigir sem saber como funciona o motor do carro.

Note que essa propriedade permite que pensemos em termos de "classe de análise" antes de pensarmos em "classe de projeto". Nas "classes de análise" são definidas, basicamente, as interfaces dos objetos. Posteriormente, nas "classes de projeto", é que existirá a preocupação em como fazer tais objetos funcionarem a partir da interface estabelecida.

POLIMORFISMO

Polimorfismo é uma propriedade que permite que um objeto que conheça uma determinada *interface*, pode se comunicar, isto é, trocar mensagens com qualquer outro objeto que respeite aquela *interface*, independentemente de qual seja o tipo do objeto com quem está se comunicado. Em outras palavras, dois objetos que conheçam uma mesma *interface* podem se comunicar, independentemente de quais sejam suas classes.

Exemplo: Se Carro Azul e Caminhonete Vermelha possuem a mesma interface, que é conhecida por João, então:

Objeto	Ação	Objeto		Objeto	Ação	Objeto
João	Dirige	Carro azul	=>	João	Dirige	Caminhonete Vermelha

Trocando em miúdos, se carro e caminhonete possuem a mesma interface de operação que é conhecida por João, então João saberá operar tanto o carro quanto a caminhonete, mesmo que o objeto caminhonete tenha sido inventado muito tempo depois da criação do objeto João.

HERANÇA

Herança é a propriedade que nos permite criar uma nova classe especificando que ela "é uma" outra classe também. Por exemplo, se temos a classe "Pessoa", podemos criar a classe "Trabalhador" dizendo que *Trabalhador é uma Pessoa.* Assim, um objeto da classe Trabalhador vai também possuir todos os atributos e métodos de um objeto da classe Pessoa (como *nome*, por exemplo).

De forma mais rigorosa, podemos dizer que herança é a propriedade que permite que, ao especializar uma classe, os objetos da nova classe preservem todos os comportamentos e atributos dos objetos da classe original, ou seja, os comportamentos e atributos são *herdados*. Em outras palavras, a nova classe (mais especializada) continua a respeitar a *interface* estabelecida pela classe original.

Exemplo:

classe: Pessoa	ação: dirige	**classe: Veículo**
objeto: joao		***subclasse: Carro*** (é um Veículo)
objeto: carla		*objeto: carroAzul*
		subclasse: Caminhonete (é um Veículo)
		objeto: caminhoneteVermelha

Se objetos da classe Pessoa conhecem a interface da classe Veículo, conhecem também a interface das classes Carro e Caminhonete, que são classes **especializadas** da classe Carro original. Assim, se *joao* e *carla* são objetos da classe Pessoa e *carroAzul* é um objeto da classe Carro e *caminhoneteVermelha* é um objeto da classe Caminhonete, então

tanto objeto *joao* quanto o objeto *carla* podem interagir com carroAzul e caminhoneteVermelha.

Muitas linguagens, incluindo C++ e Java, utilizam a propriedade da Herança para implementar o Polimorfismo. O Java inclui também o tipo "interface" para esta finalidade.

2. Algumas Observações sobre Programação Orientada a Objetos

1) Não é preciso usar uma linguagem orientada a objetos para programar de forma orientada a objetos. É importante lembrar que tudo que é feito no computador é convertido pelo compilador em código de máquina. Assim, em essência, é possível realizar Programação Orientada a Objetos até mesmo em Assembly.

2) Um programa orientado a objetos não tem necessariamente uma interface com o usuário (UI) orientada a objetos. O método de programação e a interface com o usuário são duas coisas bastante distintas. Para que a interface seja OO, é necessário que o programador implemente todas as características citadas anteiormente para os elementos da interface, o que não é tão simples. Como um exemplo, podemos citar a interface do Windows, que é programada em linguagem OO, mas sua operação não respeita os princípios da OO.

Por outro lado, apesar disso, é possível dizer que o uso de uma linguagem orientada a objetos facilita a implementação de um projeto orientado a objetos, e a razão para isso é a proximidade semântica de modelos.

Com proximidade do modelo e o código, é mais fácil também definir uma mudança no sistema fazendo uma análise do projeto e, a partir dele, ir diretamente ao código daquele componente ou módulo para realizar as mudanças necessárias.

Nem tudo são flores, entretanto. O uso de Orientação a Objetos tem implicações de desempenho, seja em termos de memória ou de processamento.

2.1. Eficiência em Softwares Orientados a Objetos

1) Em geral, os programas OO são menos eficientes que programas bem estruturados, sob o ponto de vista de custo computacional, ou seja, na sua velocidade de execução e consumo de memória. Por outro lado, nestas condições, os códigos estruturados tendem a ser relativamente mais complexos.

2) É possível dizer também, em geral, que os programas OO são muito mais eficientes do que programas estruturados, sob o ponto de vista de tempo de desenvolvimento. Isto é uma conseqüência da maior abstração e proximidade do modelos de projeto e implementação com relação à realidade.

3) Em programas OO a correção de bugs é mais simples, pois é mais fácil testar componentes (que quase não dependem do "mundo exterior") do que testar funções (que podem ser altamente dependentes do "mundo exterior"). Entenda "mundo exterior" por *variáveis globais* e outros elementos similares.

4) Os programas OO são modulares por natureza, já que os próprios objetos acabam por constituir módulos atômicos. Por esta razão, o reaproveitamento de código é facilitado.

3. Um Diagramas de Classe UML Simplificado

Para rememorar os principais elementos de um diagrama de classes UML vamos analisar um diagrama simplificado.

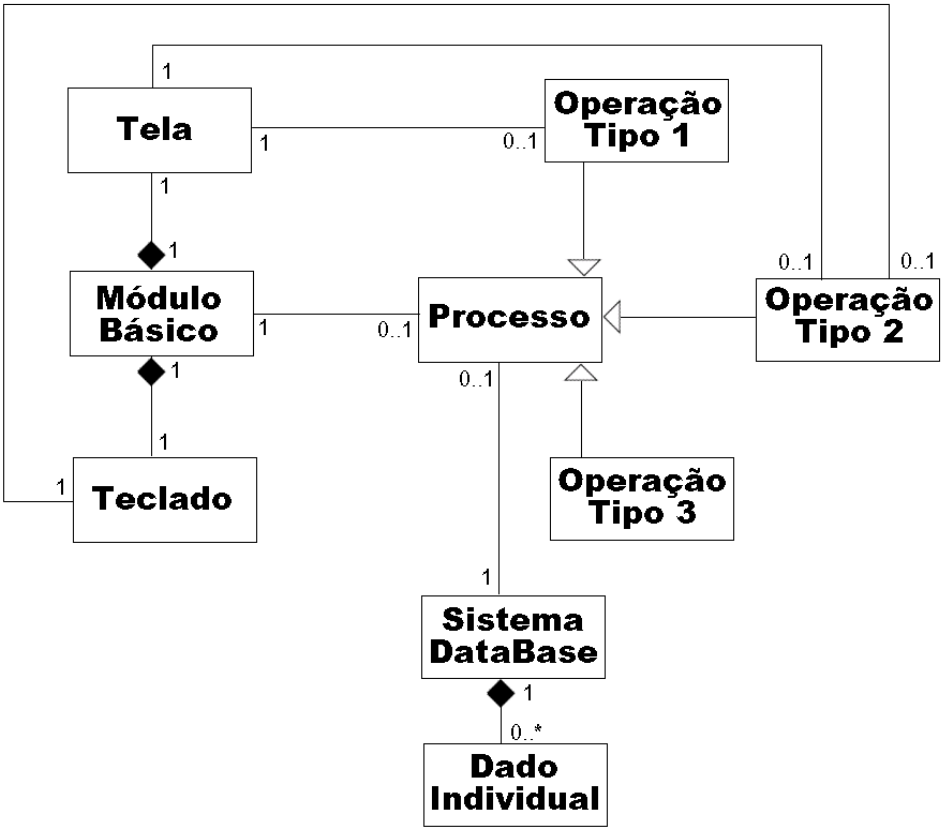


Figura 1: Diagrama de Classes Simplificado

Este é um diagrama básico de classes de um Sistema de Informações genérico. Neste diagrama temos uma arquitetura do tipo MVC, onde a Visão (Módulo Básico, Teclado e Tela) é separada do Modelo (Sistema Database e Dado Individual) e ambos são separados do Controle (Processo, Operação Tipo 1, Operação Tipo 2 e Operação Tipo 3).

Temos que o Módulo Básico é composto por Teclado e Tela, e o Sistema DataBase é composto por vários Dado Individual. Além disso, as Operações (Tipo 1, 2 e 3) são especializações da classe Processo.

BIBLIOGRAFIA

BEZERRA, E. **Princípios de Análise e Projeto de Sistemas com UML**. 2ed. Rio de Janeiro: Editora Campus Elsevier, 2007.

JACOBSON, I; CHRISTERSON, M; JONSSON, P; ÖVERGAARD, G. **Object-oriented software engineering: a use case driven approach.** Essex, England: Addison-Wesley Longman Ltd, 1992.

COAD, P; YOURDON, E. **Análise baseada em objetos.** Editora Campus, 1992.

Etapa 2: Orientação a Objetos e UML
Tópico 5: Introdução à UML
Prof. Daniel Caetano

Introdução

Como visto anteriormente, no início da história do desenvolvimento de software, tais softwares eram de complexidade bastante reduzida, envolvendo um número muito pequeno de pessoas entre análise, projeto e implementação.

Nestes tipos de software, o uso de pequenos fluxogramas ou mesmo algumas linhas de *pseudocódigo* eram suficientes para garantir o funcionamento adequado do sistema como um todo.

Entretanto, com a passagem do tempo e a evolução dos computadores, a realidade dos projetos de software mudaram drasticamente: enormes programas, com milhares de funcionalidades, passaram a exigir dezenas ou centenas de analistas, projetistas e programadores. Neste novo panorama, as antigas técnicas de modelagem de software tornaram-se inviáveis. Em vista disso, novos paradigmas de desenvolvimento surgiuram, em especial a modelagem orientada a objetos.

No entanto, não havia uma linguagem visual padronizada capaz de explicitar entes e relações como:

- Classes e objetos (e seus comportamentos e campos)
- Associações
- Mensagens
- Encapsulamento
- Polimorfismo
- Herança

Por esta razão, muitos cientistas de projeto começaram a buscar formas de representar tais elementos e a idéia para a modelagem de sistema global foi tomando forma: modelar "caixas pretas" que fornecem "serviços" aos "clientes".

- "Caixas pretas" seriam as classes ou objetos que realizam tarefas em nosso sistema.
- "Serviços" são todas as mensagens que a classe ou objeto pode reconhecer.
- "Clientes" são todos que podem trocar *mensagens* com as classes e objetos.

Com isso, a descrição do funcionamento interno de cada "caixa preta" (ou seja: cada classe) seria feita em separado e, em tese, não deve impor restrições ao projeto do sistema global.

1. Histórico da UML

DÉCADA DE 80

- Crescimento do uso de análise e desenvolvimento de projetos baseados em objetos nas empresas.

- Propostas de metodologias Orientadas a objetos por:

  - Booch, Grady
  - Rumbaugh, James
  - Jacobson, Ivar

DÉCADA DE 90

- Após divulgação de vários trabalhos em separado, Rumbaugh e Booch resolvem se unir, constituindo uma empresa chamada "Rational Software Corporation", em 1994, e unificam suas metodologias em uma única.

- Alguns anos depois, já em 1996, Jacobson se associou à Rational Software Corporation e contribuiu com a metodologia da empresa, que criou uma metologia unificada.

- Ainda em 1996, pouco depois da associação de Jacobson à Rational Software, foram divulgados para a comunidade as primeiras especificações do que foi chamado de UML: Unified Modeling Language.

- Nesta mesma época, grandes empresas como a IBM, Microsoft, Oracle e outras também já haviam percebido a necessidade de algo como a UML e trabalhavam neste tipo de desenvolvimento.

- No fim de 1996, o Object Management Group (OMG), uma organização sem fins lucrativos que promove a padronização de tecnologias orientadas a objeto, solicitou idéias da comunidade para a criação de uma linguagem gráfica para especificação de projetos orientados a objeto.

- A Rational Software, juntamente com as maiores empresas da área de desenvolvimento de software apresentaram, então, à OMG o que seria a primeira versão oficial da UML.

- Em 1997 a OMG oficializou a UML 1.1.
- Em 2003 ocorreu uma pequena revisão do padrão, e foi oficializada a UML 1.5.
- Em 2006 foi oficializado o padrão UML 2.

Atualmente o padrão está na revisão 2.1.1 (agosto de 2007) e o padrão 2.2 está em fase de definição.

2. O que é a UML?

Como já dito, UML significa "Unified Modeling Language". Alguns livros definem como "Universal Modeling Language", mas essa nomenclatura não é recomendada.

Em poucas palavras:

"UML é uma linguagem gráfica que permite que sejam projetados sistmas de software orientados a objetos".

A UML talvez não tivesse tanta importância, se não tivesse se tornado uma notação padrão da indústria. Isso significa que grande parte notoriedade da UML, além de permitir que praticamente qualquer modelagem orientada a objetos seja representada, é sem dúvida alguma devido ao fato de que, hoje, em 2006, a grande maioria das empresas de médio e grande porte a utiliza em seus projetos.

A razão para esta grande aceitação tem raízes no fato de que a UML não foi um padrão desenvolvido "do nada", mas sim baseado nas notações mais populares para cada tipo de atividade, tendo sido criados novas formas de notação apenas nos casos em que não existia nada previamente conhecido e utilizado pela indústria.

Finalmente, a UML possui uma características bastante desejável, que é a sua flexibilidade. Dizemos que a UML é *extensível*, uma vez que ela permite que novos diagramas e representações sejam criados à medida em que se tornam necessários, nas revisões do padrão que ocorrem periodicamente.

Existem basicamente 13 tipos de diagrama na UML: 6 diagramas de estrutura e 7 diagramas de comportamento.

Diagramas de estrutura são, obviamente, aqueles que definem a estruturação de um sistema, quais são as partes e quais delas se inter-relacionam.

Diagramas de comportamento são aqueles que definem o funcionamento de cada elemento estrutural do projeto, bem como os mecanismos de inter-relação entre os elementos para atingir determinadas tarefas.

No curso de projeto são estudados apenas alguns destes diagramas - os mais utilizados -, mas todos os outros podem ser verificados na bibliografia. Os diagramas a serem estudados são:

- Diagramas de Casos de Uso (comportamento)
- Digramas de Classes (estrutura)
- Diagramas de Seqüência (comportamento)
- Diagramas de Atividades (comportamento)

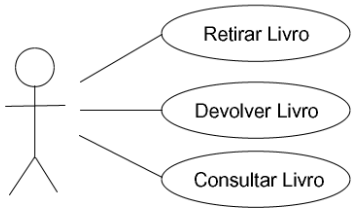
Destes, o mais importante de todos estes é o Diagrama de Classes, pois será utilizado sempre, como uma "coluna vertebral" do projeto e implementação do software.

2.1. Diagrama de Casos de Uso

Os diagramas de casos de uso são o primeiro passo na transformação de um documento de requisitos em um projeto. Embora simples, eles servem para explicitar as funções que o software precisa exercer frente a seus possíveis tipos de usuário.

Ele é composto basicamente por atores e funções. Os atores são desenhos de pessoas e as funções ficam em elipses. Se houver mais de um ator ou a função do ator não for óbvia, deve ser especificado um "nome" para esse ator (como "cliente" ou "balconista", por exemplo).

O diagrama de casos de uso também auxilia na fase de testes, pois é com base nos casos de uso que devemos planejar os principais testes do sistema global.



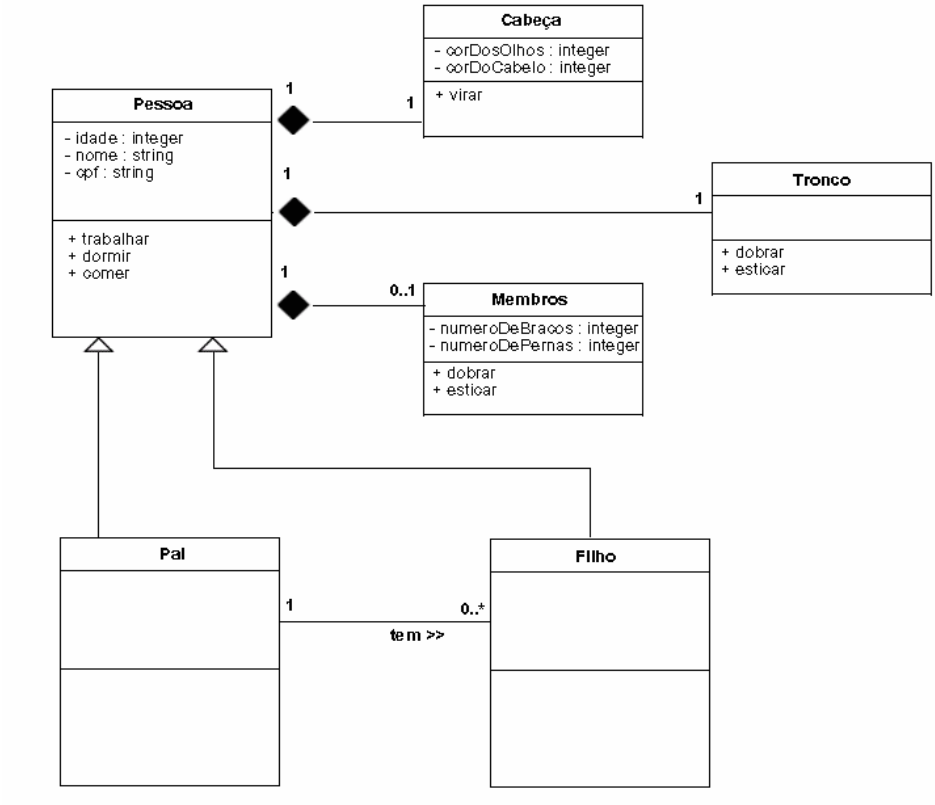
2.2. Diagrama de Classes

Os diagramas de classes são diagramas que, como o próprio nome diz, fornecem informação sobre a estruturação das classes. Neste diagrama você encontra todos os métodos e atributos de cada classe, qual classe é especializada de outra, qual classe tem relacionamento com outra, bem como a cardinalidade de tais relacionamentos.

As setas vazias indicam especialização e apontam da classe mais especializada para a mais genérica. Os losangos preenchidos representam agregação, indicando que a classe que contém o losango é composta pela classe na outra parte da ligação. Linhas diretas indicam que aquelas classes se relacionam, mas a relação não é nem de especialização nem de

composição (no exemplo, um pai "tem" um filho, mas o filho não é parte dele, nem filho é especializado de pai).

Em cada classe há 3 compartimentos: o mais do alto é o nome da classe. O do meio é o compartimento dos atributos e o de baixo é o compartimento dos métodos. Os símbolos + e - ao lado dos métodos e atributos indicam se esses métodos e atributos são públicos (visíveis fora da classe) ou se são privados (invisíveis fora da classe). Em geral os métodos são públicos e os atributos são privados, mas essa regra algumas vezes precisa ser quebrada. Os números nas ligações indicam a cardinalidade.



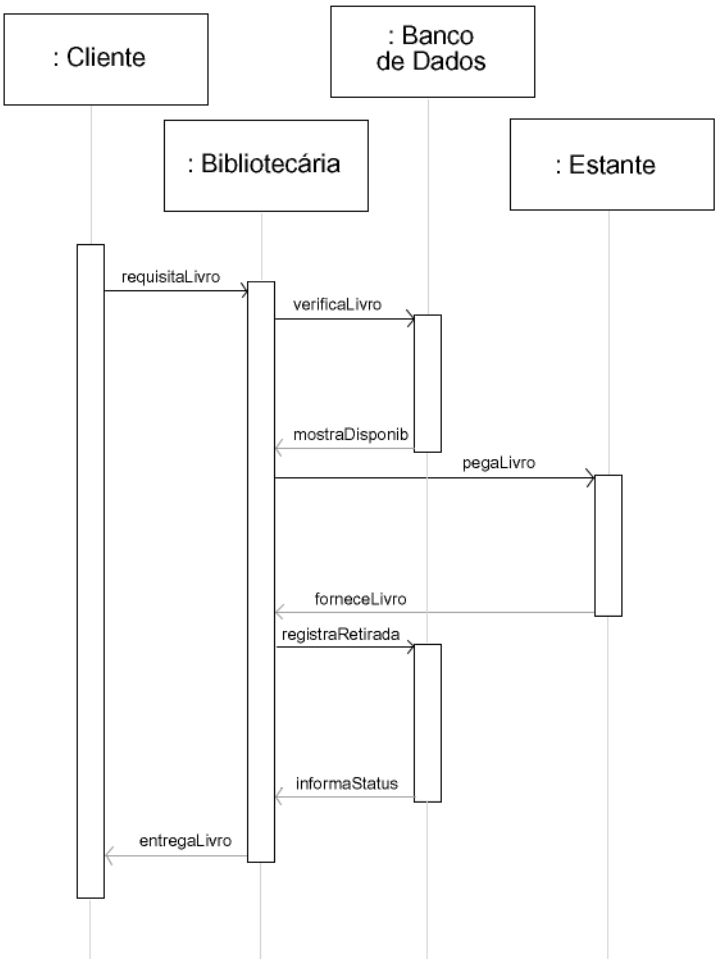
2.3. Diagrama de Seqüência

Os diagramas de seqüência têm função similar aos diagramas de comunicação, entretanto eles são criados quando é preciso dar mais ênfase à ordem em que as ações são tomadas e também o "tempo de vida" dos objetos de cada classe, numa dada operação.

Este diagrama é composto por retângulos que representam as classes dos objetos (com o nome precedido por dois pontos ":"). As linhas tracejadas ou claras na vertical indicam pontos em que o objeto está inativo ou não existe. Os retângulos estreitos verticais indicam o "tempo de vida" do objeto.

As setas ligando o tempo de vida de um objeto ao de outro indicam as operações executadas. As setas escuras representam operações ativamente solicitadas. As setas claras (ou tracejadas) indicam ações de resposta (ou reações).

Observando de cima para baixo é possível ter uma noção temporal da execução de cada ação.

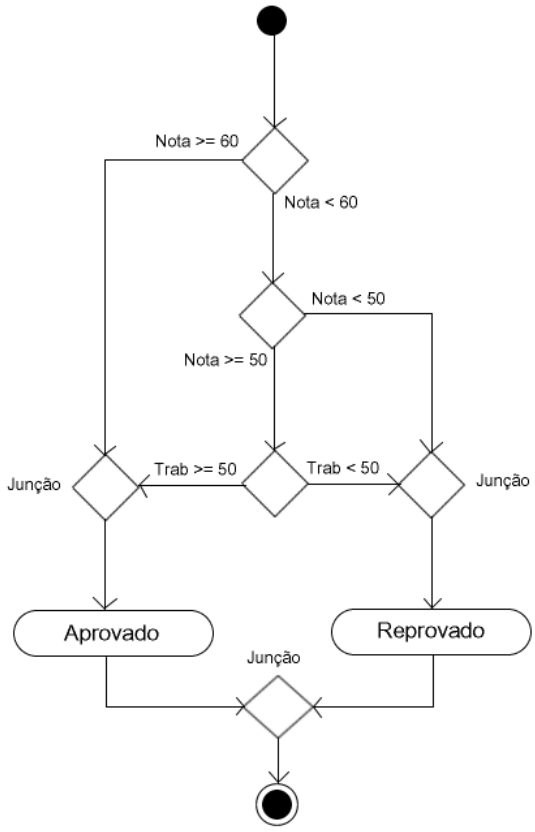


2.4. Diagrama de Atividades

Os diagramas de atividades descrevem algoritmos para realizar determinadas tarefas. Em essência, eles servem como uma descrição passo a passo do que deve ser feito em cada situação e uma de suas funções clássicas é descrever o que ocorre dentro um método de uma classe.

Neste diagrama, oriundo dos populares fluxogramas, o círculo preenchido indica o início do processo e o circulo vazado com um preenchido dentro indica o fim do processo.
Os blocos com laterais semi-circulares são as ações a serem desempenhadas (calcular isso, indicar aquilo, etc). Os losangos podem ter dois significados: ou eles indicam uma operação de seleção (uma seta entrando e duas saindo) e outros indicam uma junção (várias setas entrando e apenas uma saindo). As operações de seleção têm em suas setas de saída a indicação da condição do caminho, já as de junção não. As setas indicam o caminho do diagrama de atividades.

É importante observar que nem sempre um fluxograma deste tipo representa um desenvolvimento estruturado. Um diagrama estruturado deve apenas contar com operações de **seqüência**, **seleção** e **repetição**. Outras estruturas podem tornar o algoritmo "não-estruturado".



Biliografia

DEITEL, H.M; DEITEL, P.J. **Java: como programar** - Sexta edição. São Paulo: Pearson-Prentice Hall, 2005.

GUEDES, G.T.A. **UML Uma abordagem prática.** Novatec, 2004.

MEDEIROS, E. **Desenvolvendo software com a UML 2.0 - Definitivo**. Pearson Brasil, 2004.

WILLIAMS, C; TEXEL, P. P. **Use cases combined with Booch / OMT / UML.** Prentice Hall, 1997.

LIMA: A. S. **UML 2.0 - Do requisito à solução.** Editora Érica, 2004.

- Internet:
http://www.uml.org/
http://www.ibm.com/software/rational/uml

Etapa 2: Orientação a Objetos e UML
Tópico 6: Mecanismos Gerais da UML e ferramentas CASE
Prof. Daniel Caetano

Objetivo: Apresentar os mecanismos gerais da UML e apresentar os conceitos de ferramentas CASE.

Bibliografia Básica:

- DEITEL, H.M; DEITEL, P.J. **Java**: como programar - 6ed. São Paulo: Pearson-Prentice Hall, 2005.
- BEZERRA, E. **Princípios de Análise e Projeto de Sistemas com UML**. 2ed. Rio de Janeiro: Editora Campus Elsevier, 2007.

Introdução

Como foi visto anteriormente, o projeto de um sistema arbitrariamente grande envolve muitas atividades, muitos profissionais organizados em vários grupos que interagem entre si. A coordenação entre estes grupos depende muito de uma comunicação eficiente entre os mesmos, de forma que todos consigam visualizar o sistema da mesma forma.

Para atingir este objetivo, um grande facilitador é o uso de uma linguagem padronizada em todos os níveis de projeto, que deve ser adotada por todas as equipes encarregadas do mesmo. Se esta linguagem for extendida para uso em outros projetos, tanto quanto melhor, uma vez que haverá não só uma facilitação no reuso de trechos de projeto e implementação, mas também um melhor aproveitamento do conhecimento dos profissionais envolvidos, que não precisarão despender tempo em aprender uma nova linguagem de especificação de projeto a cada novo projeto a ser desenvolvido.

A linguagem de especificação de projeto mais aceita nos tempos atuais é a Linguagem de Modelagem Unificada (*Unified Modeling Language* ou, simplesmente, *UML*).

1. Mecanismos Gerais da UML

A UML é composta de três componentes fundamentais: blocos de construção, regras de associação destes blocos e mecanismos de uso geral, grande parte dos quais serão apresentados nesta aula. Estes mecanismos são: **estereótipos**, **notas explicativas**, **etiquetas valoradas**, **restrições**, **pacotes** e a **OCL**.

1.1. Estereótipos

Os estereótipos são mecanismos que permitem estender o significado de alguns elementos da UML. Existem estereótipos pré-definidos na UML e existem os estereótipos definidos pela equipe de projeto e desenvolvimento de um sistema.

Além disso, os estereótipos podem ser iconográficos ou textuais; na figura 1 temos exemplos de estereótipos iconográficos, sendo os dois primeiros predefinidos na UML e os outros dois foram definidos pela equipe.

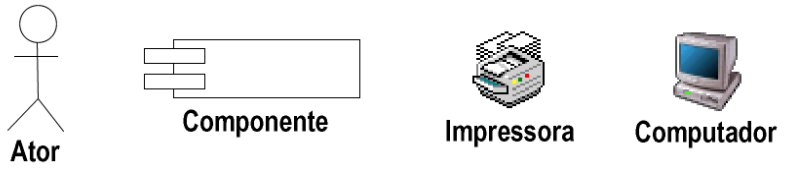


Figura 1: Exemplos de estereótipos - os dois primeiros são pré-definidos.

Os estereótipos textuais são escritos como textos entre aspas francesas: << entity >>, << interface >>, << control >> ...

1.2. Notas Explicativas

As notas explicativas servem para esclarecer alguma parte de um diagrama. Podem ter seu conteúdo em texto live ou OCL. Uma nota **não** modifica o significado de algum elemento do UML, apenas esclarece possíveis dúvidas. Sua representação é um retângulo com uma "orelha" dobrada, ligado ao elemento que explica por uma linha tracejada, com o texto escrito na parte interna do retângulo, como no exemplo da figura 2.

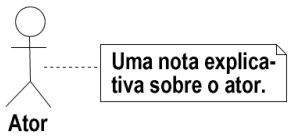


Figura 2: Uma nota explicativa.

1.3. Etiquetas Valoradas (*tagged values*)

As etiquetas valoradas servem para definir propriedades adicionais a um elemento de um diagrama; entretanto, as etiquetas só podem ser aplicadas a um elemento estendido por um estereótipo antes de poder ser estendido por uma etiqueta valorada. Como um exemplo, é possível usar uma etiqueta valorada para definir o autor e a data da criação de um dado diagrama, com pode ser visto na figura 3.

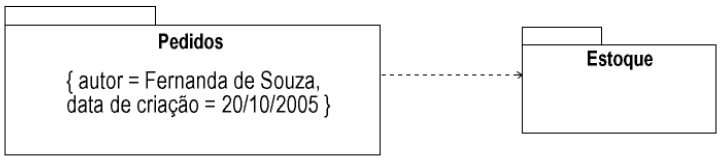


Figura 3: Uso de uma etiqueta valorada.

As possíveis sintaxes para definir as propriedades e seu valor são:

{ tag }
{ tag = valor }
{ tag1 = valor1, tag2 = valor2, ... , tagn = valorn }

1.4. Restrições

Todo elemento da UML tem um significado muito bem definido. Sempre que existe o desejo de estender ou alterar esse significado, acrescentando restrições sobre valores de um ou mais elementos do modelo, deve ser usado este mecanismo geral.

As restrições podem ser especificadas em linguagem informal ou em OCL; as restrições sempre aparecem dentro de uma nota explicativa e **sempre** devem vir delimitadas por chaves: { restrição }.

1.5. Pacotes

O mecanismo para "agrupar" da UML são os pacotes, podendo ser usado para agrupar elementos semanticamente relacionados. A representação é a de uma pasta com uma aba. Um pacote pode conter outros pacotes, ser especialização (ou generalização) de outros pacotes, depender de outros pacotes etc. Na figura 3 temos dois pacotes, sendo que o pacote Pedidos depende do pacote Estoque.

A forma de representar o conteúdo de um pacote é simplesmente representar os elementos dentro do quadro maior do pacote. Por exemplo, um pacote com duas classes seria representado como na figura 4.

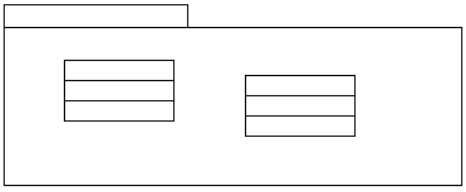


Figura 4: Um pacote contendo duas classes.

1.6. A OCL

A UML define uma linguagem formal para especificação de restrições (OCL, Linguagem de Restrição de Objetos, em português). Ela pode ser utilizada para expressar a navegação entre objetos, expressões lógicas, pré-condições etc.

O uso da OCL está além do escopo deste curso, mas para os interessados, uma descrição completa pode ser encontrada em *UML - Guia do Usuário* (Booch et al., 2006).

2. Ferramentas CASE

CASE significa "Computer-Aided Software Engineering" (ou, em bom português, "Engenharia de Software Auxiliada por Computador"). Assim, ferramentas CASE são ferramentas que facilitam a vida do projetista e desenvolvedor de software.

A relação da ferramenta CASE com o projetista de software é bastante similar à das ferramentas CAD (Computer-Aided Design) com o projetista de edificações e peças mecânicas, ou das ferramentas CAM (Computer-Aided Math) para todos os profissionais que precisam trabalhar com matemática e estatística.

Em geral as ferramentas CASE permitem que um software seja editado através de seus modelos (através de uma linguagem gráfica como o UML, por exemplo) e são capazes de gerar toda a estrutura do código para o programador, de forma que este precisa apenas "preencher as lacunas".

Uma das ferramentas CASE mais famosas e completas é o Rational Rose, da IBM, sendo também uma das mais completas e caras. Outras ferramentas proprietárias são a System Architect, Enterprise Architect e Microsoft Visio. Outras ferramentas CASE famosas são o Umbrello e o ArgoUML, sendo estas "free" e a última escrita totalmente em Java.

2.1 A relação entre as IDEs e as Ferramentas CASE

IDE é um "Integrated Development Environment" ("Ambiente de Desenvolvimento Integrado). Em outras palavras, é um software que integra todas as atividades de programação, como edição, compilação e debugging (resolução de defeitos), de forma simples e prática.

As primeiras IDEs surgiram ainda na década de 70, com linguagens como o BASIC, e seu objetivo era permitir que pessoas leigas fossem capazes de digitar e executar programas sem tomar conhecimento de todo o mecanismo reponsável por transformar um código em linguagem de alto nível em um código em linguagem de máquina. Pouco depois surgiram as

primeiras versões de IDEs famosas até os dias de hoje, como as da Borland (Borland Pascal, Turbo Pascal, Borland C, Turbo C, etc).

As IDEs evoluiram muito e hoje incluem muitas características de documentação (consulta e criação), bem como intrumentos de controle de versão e outros. Grandes exemplos desta evolução são produtos como o Eclipse, NetBeans e mesmo os proprietátios Microsoft Visual Studio e outros.

As IDEs, entretanto, sempre foram softwares mais voltados ao ambiente de programação em si: quem gosta de usar IDE é programador. As IDEs sempre foram complexas e freqüentemente não possuem ferramentas administrativas adequadas para lidar com as etapas de análise e projeto. É exatamenet neste nicho que as ferramentas CASE entram.

As ferramentas CASE são usadas para a modelagem do problema, constituindo a forma que o software terá. Uma vez que o modelo esteja pronto, a ferramenta CASE pode ser usada para gerar o código da estrutura do software, que será então importado em uma IDE, onde os programadores completarão a atividade de implementação. Algumas boas ferramentas CASE permitem também que sejam gerados os diagramas do modelo a partir de um código, mas para que isso funcione corretamente é preciso que os programadores sejam bastante disciplinados em seguir as convenções da ferramenta CASE em questão, para que tal ferramenta possa compreender tudo que os programadores fizeram.

2.2. As Ferramentas Mistas

Com o passar do tempo, muitos programadores passaram a se utilizar também de ferramentas CASE na primeira etapa de seus projetos, antes de partir para a programação propriamente dita. Como ficou comprovado, esta é uma atitude muito sábia, mas alguns destes programadores se ressentiram com dois pequenos problemas:

1) Usar dois programas separados (um CASE e um IDE) não era muito agradável, pois depois que o código gerado pelo CASE era editado na IDE, nem sempre era possível voltar atrás e alterá-lo novamente com a ferramenta CASE.

2) As ferramentas CASE puras possuem, normalmente, muitas opções de descrição de projeto relativas ao comportamento do software, que são importantes para o programador saber o que fazer, mas na grande maioria dos casos não é transformada em código automaticamente.

Como uma solução para isso - e também para facilitar o aprendizado de programação ligadas ao projeto de sistemas - foram desenvolvidas ferramentas mistas, que incluem algumas características de modelagem estrutural das ferramentas CASE em uma IDE comum.

Uma destas ferramentas é o BlueJ, que vamos usar neste curso, que permite que a estrutura do software (as classes e suas inter-relações) sejam modeladas num diagrama muito

similar ao diagrama de classes UML simplificado, ao mesmo tempo que ela inclui grande parte das caracterísitcas de uma IDE completa (como debugging, documentação, etc).

2.3. Nomenclatura das Ferramentas CASE

Em essência, de acordo com o significado de "CASE", as IDEs não deixam de ser, também, ferramentas CASE. Por esta razão, existe uma nomenclatura bastante comum para definir todas estas ferramentas: IDEs, CASEs e mistas:

- Lower CASE: são as ferramentas usadas em programação, em especial, as IDEs.
- Upper CASE: são as ferramentas usadas em análise e projeto.
- Integrated CASE: são as ferramentas que anteriormente chamei de "mistas".

Bibliografia

BEZERRA, E. **Princípios de Análise e Projeto de Sistemas com UML**. 2ed. Rio de Janeiro: Editora Campus Elsevier, 2007.

DEITEL, H.M; DEITEL, P.J. **Java**: como programar - 6ed. São Paulo: Pearson-Prentice Hall, 2005.

Etapa 3: Desenvolvimento da Etapa de Análise
Tópico 7: Modelagem de Casos de Uso
Prof. Daniel Caetano

Objetivo: Apresentar os elementos da modelagem de casos de uso e apresentar algumas dicas na elaboração de Diagramas de Casos de Uso.

Bibliografia Básica:

- BEZERRA, E. **Princípios de Análise e Projeto de Sistemas com UML**. 2ed. Rio de Janeiro: Editora Campus Elsevier, 2007.
- DEITEL, H.M; DEITEL, P.J. **Java**: como programar - 6ed. São Paulo: Pearson-Prentice Hall, 2005.

Introdução

Após as primeiras fases da análise de requisitos, quando já foi tomado conhecimento sobre o problema a ser solucionado, usuários, requisitos funcionais e não funcionais, restrições etc, é interessante a elaboração de documentos estruturados e gráficos que representem as funcionalidades externamente observáveis do sistema.

O objetivo destes documentos será não só o apoio ao desenvolvimento mas também uma melhor compreensão, por parte do cliente, do sistema a ser desenvolvido. Os documentos estruturados são as descrições dos casos de uso e os diagramas são chamados de "Diagramas de Casos de Uso" ou simplesmente DCU.

É importante lembrar que os casos de uso (e por conseqüência os DCUs) não representam funcionalidades internas do sistema, não sendo adequada a construção dos mesmos com base em regras de decomposição funcional ou de representação de fluxo de dados. Os casos de uso representam uma visão externa das funcionalidades do sistema, representando também os elementos externos que interagem com ele.

1. Descrição de Casos de Uso

Cada caso de uso é definido por meio de uma descrição narrativa (textual) das interações que ocorrem entre os elementos externos (atores) e o sistema. Não há uma estrutura textual padrão e ela pode variar em termos de formato, abstração e detalhamento.

Quanto ao **formato**, ele pode ser *contínuo*, *numerado* ou *tabular*. O formato **contínuo** é uma descrição simples, em forma narrativa, das interações existentes entre o sistema.

"Este caso inicia quando o cliente chega ao caixa eletrônico e insere seu cartão. O sistema solicita a senha e, após o cliente digitar a senha, o sistema validará a mesma,

exibindo as opções disponíveis em seguida. O cliente indica que deseja imprimir um extrato e o sistema imprime o extrato. O cliente retira o extrato e o caso de uso termina."

O formato **numerado** é separado em frases numeradas seqüenciais, cada uma representando uma interação.

1. *Cliente insere seu cartão no caixa eletrônico.*
2. *Sistema solicita a senha ao cliente.*
3. *Cliente digita a senha.*
4. *Sistema valida a senha e exibindo as opções disponíveis.*
5. *Cliente indica que deseja imprimir um extrato*
6. *Sistema imprime o extrato.*
7. *Cliente retira o extrato e o caso de uso termina.*

O formato **tabular** indica uma coluna para cada elemento que interage com o sistema (atores) e uma coluna para o sistema.

Cliente	***Sistema***
Insere cartão no caixa eletrônico	*Solicita senha*
Digita senha	*Valida senha e exibe opções*
Solicita impressão de extrato	*Imprime extrato*
Retira extrato	

Quanto ao **detalhamento**, a descrição pode ser desde sucinta, apresentando apenas as interações entre atores e sistemas sem muitos detalhes, até uma descrição extremamente detalhada de cada interação.

Quanto ao **grau de abstração**, a descrição pode ser *essencial* ou *real*. Uma descrição essencial é aquela que não cita detalhes de tecnologia ("O cliente fornece sua identificação") enquanto que uma descrição real é aquela que cita detalhes tecnológicos ("O cliente insere o cartão no caixa eletrônico"). Para saber se uma descrição está na forma essencial ou real, basta imaginar se a descrição faria sentido 100 anos atrás ou 100 anos no futuro. Se fizer, é uma descrição essencial, caso contrário, é uma descrição real.

1.1. Cenários

É comum que uma funcionalidade descrita por um caso de uso possa ser usada de diversas maneiras diferentes. Para representar estas diversas maneiras, são utilizados os chamados "cenários", que nada mais são que casos de uso de situações específicas: por exemplo: o que acontece se o cartão do cliente não for aceito quando ele for imprimir seu extrato?

Uma coleção de cenários para cada caso de uso pode ser usada posteriormente, na fase de testes, para avaliar se o sistema realmente está respondendo de acordo com o esperado.

1.2. Atores

Qualquer elemento externo ao sistema é denominado "ator". Os atores são chamados de "elementos externos" porque eles não fazem parte do sistema. Atores interagem com o sistema (trocam informações), representando a forma com que o sistema "percebe" o mundo.

Existem diversos tipos de atores. Alguns exemplos são *cargos* (empregado, cliente, gerente, vendedor etc.), *organizações* (ou suas divisões: empresa, fornecedora, administradora de cartões, rh, diretoria etc.), *outros sistemas de software* (sistema de cobrança, sistema de estoque etc.), *equipamentos* (sensor, leitor de código de barras etc.), dentre outros.

É importante lembrar que o termo "ator" é usado porque os atores representam "papéis" e não entidades propriamente dita. Por exemplo, podemos ter num caso de uso um ator chamado Cliente (quem faz compras) e um ator chamado Agendador (quem agenda a data de entrega das compras) e uma mesma pessoa pode em um dado momento representar o papel de *cliente* e, no momento seguinte, representar o papel de *agendador*. Por esta razão, nomes próprios como "UniRadial", "Joaquim" etc não são bons nomes para atores de um caso de uso.

Adicionalmente, é comum que um ator participe de vários casos de uso, bem como a participação de vários atores em um mesmo caso de uso. O ator que iniciar a seqüência de interações de um caso de uso é chamado de "ator primário", sendo os outros os "atores secundários".

1.3. Relacionamentos

Não faz sentido falar em casos de uso e atores de forma isolada. É preciso representar também os *relacionamentos* entre eles. Um ator deve estar relacionado a um ou mais casos de uso. Um ator pode estar relacionado a outro ator e um caso de uso pode estar também relacionado a outro caso de uso.

Os tipos de relacionamento são: *comunicação*, *inclusão*, *extensão* e *generalização*.

A **comunicação** é o relacionamento mais comum, e ocorre entre atores e casos de uso. Ela indica uma interação com o sistema, ligando um ator ao caso de uso com o qual ele está associado.

A **inclusão** existe somente entre casos de uso. É usada para indicar que um caso de uso *inclui* outro, funcionando como uma espécie de "subrotina", para evitar a necessidade de reproduzir uma mesma seqüência de passos em todos os casos de uso. Por exemplo, para qualquer operação em um caixa eletrônico, será necessária uma *identificação e autenticação*.

Ao invés de descrever estes passos em todos os casos de uso, é possível criar um caso de uso chamado *identificação e autenticação* e **incluí-lo** em todos os outros casos de uso.

A **extensão** existe somente entre dois casos de uso, e serve para que um estenda o outro. Assim, se um caso A estende o caso B, isso significa que, em algum cenário específico de B, A será usado. Em outras palavras, o fato de B estar "estendido" não significa que o caso de uso A ocorrerá sempre que o caso de uso B ocorrer. Um exemplo prático é o caso de uso "corrigir ortografia" que estende o caso de uso "editar texto". Nem sempre que se edita um texto se corrige a ortografia, mas é uma atividade que pode ser executada, e que é uma estensão se "editar texto".

A **generalização** ocorre tanto entre dois casos de uso quanto entre dois atores, permitindo que o caso de uso ou ator mais especializado *herde* as características do caso de uso ou ator mais genérico. Se um caso de uso A é uma generalização do caso de uso B, isso significa que B compreende todas as interações de A, implementando também alguma característica distinta de A. Se um ator C é uma generalização de um ator D, isso significa que o ator D pode participar de todas as interações que C participa, além de algumas que C não pode participar.

Um exemplo de generalização de casos de uso é a especialização do caso de uso "Realizar Pagamento" no caso de uso "Realizar Pagamento com Cartão de Crédito". Um exemplo de especialização de um ator é um "usuário de biblioteca" com o ator "professor", que além de poder retirar livros como um usuário qualquer, também pode solicitar novas obras à biblioteca.

1.3.1. Quando Usar Relacionamentos

A regra de ouro é utilizá-los com parcimônia. O uso exagerado pode tornar seus casos de uso de difícil compreensão e manutenção. De qualquer forma, algumas regras podem auxiliá-lo:

Inclusão: use quando um comportamento se repetir em mais de um caso de uso, estando incluindo em *todos* os cenários dos casos de uso inclusores, que *não são completos* sem essa inclusão.

Extensão: use quando um comportamento eventual de um caso de uso tiver de ser descrito.

Generalização entre Casos de Uso: use quando identificar que há dois ou mais casos de uso com comportamento semelhante. Crie então um caso de uso genérico abstrato e relacione por generalização os casos de uso semelhantes.

Generalização entre Atores: use quando precisar definir um ator que desempenhe o papel de um ator já existente, mas que possui comportamento particular adicional.

2. Diagramas de Casos de Uso (DCUs)

O Diagrama de Casos de Uso é um tipo de diagrama definido pela UML que representa a visão externa de alto nível do sistema, representando graficamente os atores, casos de uso e os relacionamentos entre eles. Trata-se de uma forma de "diagrama de contexto", que apresenta elementos externos a um sistema e a maneira segundo as quais eles o utilizam.

Os elementos gráficos que a UML usa para representar as características vistas anteriormente estão representados na figura 1:

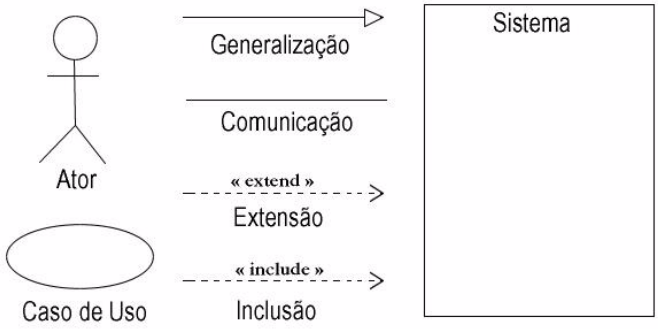


Figura 1: Elementos gráficos usados pela UML em DCUs

Um exemplo simples de um único ator, relacionado a um único caso de uso, pode ser visto na figura 2 (BEZERRA, 2007):

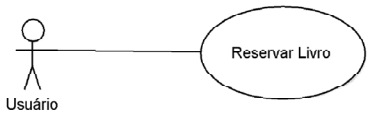


Figura 2: Exemplo de caso de uso simples

Um exemplo mais complexo, com vários casos de uso e vários atores pode ser visto na figura 3 (BEZERRA, 2007):

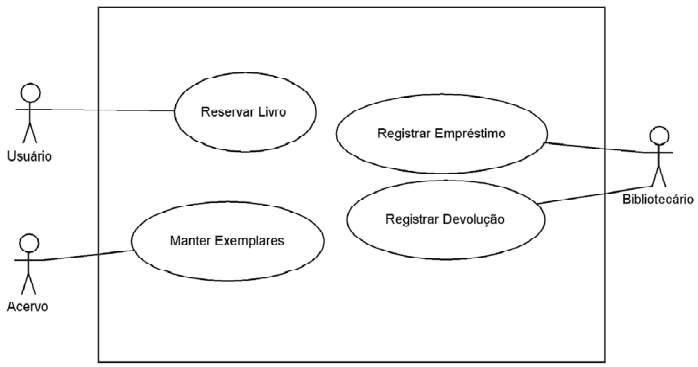


Figura 3: Diagrama com vários casos de uso e vários atores

Um exemplo de inclusão é representado na figura 4 (BEZERRA, 2007):

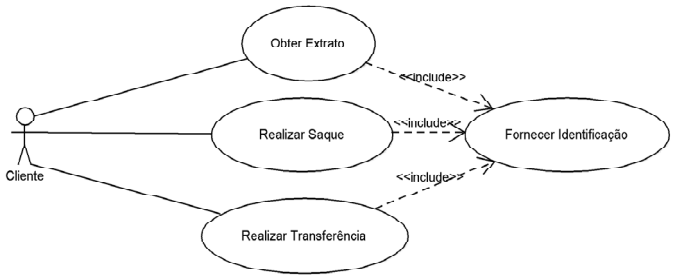


Figura 4: Diagrama com casos de uso que incluem outro

Um exemplo de extensão é representado na figura 5 (BEZERRA, 2007):

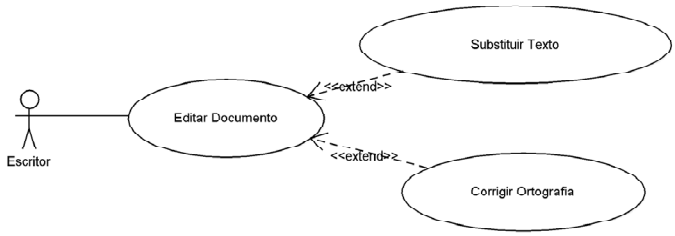


Figura 5: Diagrama com um caso de uso estendido por outros

Um exemplo de generalização de ator é representado na figura 6 (BEZERRA, 2007):

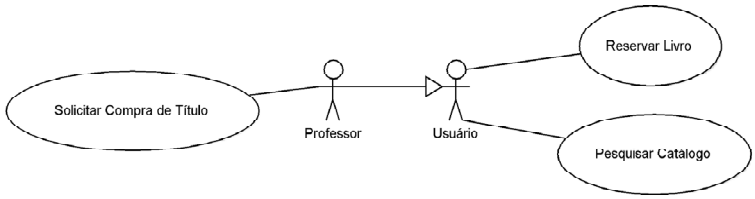


Figura 6: Diagrama com generalização de atores

Um exemplo de generalização de caso de uso é representado na figura 7 (BEZERRA, 2007):

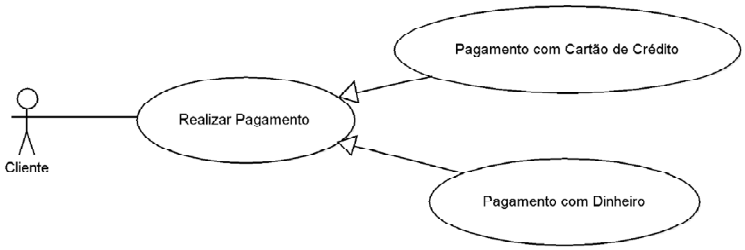


Figura 6: Diagrama com generalização de caso de uso

3. Identificação dos Elementos de um Modelo de Caso de Uso

Como foi falado anteriormente, após as primeiras etapas da análise, quando já foram identificados grande parte dos usuários do sistema, requisitos funcionais, restrições tecnológicas, etc., o passo seguinte é iniciar a identificação dos atores de casos de uso.

Nesta etapa devemos pensar em quais são as fontes de informações a serem processadas e quais são os destinos das informações processadas. Estes elementos, que irão itneragir com o sistema, serão os atores. Algumas perguntas podem ajudar a identificá-los:

1. Quais órgãos, empresas e pessoas utilizarão o sistema?
2. Quais sistemas e equipamentos interagirão com o sistema?
3. Quais pessoas que devem ser informadas de ocorrências especiais?
4. Quais pessoas estão interessadas em um certo requisito funcional do sistema?

Após a identificação dos atores, deve-se partir para a identificação dos casos de uso; entretanto, sempre vale lembrar que novos atores podem vir a ser identificados durante a identificação dos casos de uso. Os casos de uso podem ser separados em casos de uso primários e casos de uso secundários.

Casos de Uso Primários

Estes casos representam os objetivos dos atores; representam os processos que estão sendo automatizados pelo sistema. Algumas perguntas ajudam a identificá-los:

1. Quais as necessidades e objetivos de cada ator, com relação ao sistema?
2. Quais informações o sistema deve produzir?
3. O sistema deve realizar alguma ação regular (no tempo)? (caso de uso temporal) Uso de atores fictícios (Tempo).
4. Existe um ou mais caso de uso para atender a cada requisito funcional?
5. Para cada caso de uso já criado, a operação realizada pode ser desfeita? (caso de uso oposto) Ex.: um pedido já feito pode ser cancelado.
6. É preciso ocorrer alguma coisa antes de um caso de uso poder ocorrer? (caso de uso que precede outro) Ex.: é preciso existir cadastro antes da realização de compras.
7. É preciso ocorrer alguma coisa após um caso de uso ocorrer? (caso de uso que sucede outro) Ex.: é preciso agendar uma entrega após a realização de uma compra.
8. O sistema deve realizar alguma ação que depende de eventos internos? (caso de uso relacionado a alguma condição interna) Ex.: O sistema deve avisar ao usuário que chegou uma nova mensagem de e-mail.

Casos de Uso Secundários

Estes casos representam casos de uso que não trazem benefícios diretos para os atores, mas são necessários para manutenção do sistema. Algumas das categorias mais comuns destes casos de uso são:

1. Manutenção de Cadastros: operações de criação, consulta, atualização e exclusão de dados cadastrais. Normalmente são registradas como um único caso de uso, se todas as operações forem realizadas pelo mesmo ator.

2. Manutenção de Usuários e Seus Perfis: adição de novos usuários, permissões de uso, níveis de acesso etc.

3. Manutenção de Informações de Outros Sistemas: sincronização de informações entre dois sistemas que troquem dados.

4. Construção dos Modelos de Caso de Uso

Após a identificação de atores e casos de uso, é preciso construir o modelo de casos de uso, em sua forma gráfica e textual.

Documentação Gráfica

A parte gráfica é construída de forma mais ou menos tranquila para sistemas menores. Para sistemas maiores, entretanto, é preciso tomar algumas medidas de particionamento para que a visualização não fique complexa demais. Algumas destas abordagens são:

1) Diagrama que exibe um caso de uso e todos os seus relacionamentos
2) Diagrama que exibe todos os casos de uso para um ator
3) Diagrama que exibe todos os casos de uso a serem implementados em uma iteração de desenvolvimento
4) Diagrama que exibe todos os casos de uso de uma divisão específica da empresa.

Documentação Textual

Esta documentação contem basicamente duas partes: uma que descreve os atores, bem simples, com uma ou duas frases descrevendo este ator; a outra parte é a uma descrição detalhada dos casos de uso. Uma sugestão de estruturação, já que a UML não a define, segue:

1) Nome do caso de uso: o mesmo usado no DCU e deve ser único.
2) Identificador: um código que pode ser usado para referenciar o caso de uso (ex: CSU01, CSU02...)
3) Importância: identificando a criticidade de implementação (prioridade alta, média ou baixa).
4) Sumário: declaração curta do objetivo do ator ao usar esse caso de uso.
5) Ator primário: quem inicia o caso de uso.
6) Atores secundários: outros atores que participam do caso de uso.
7) Precondições: o que precisa ocorrer para que este caso de uso possa acontecer.
8) Fluxo Principal: descrição passo a passo das interações do caso de uso.
9) Fluxos Alternativos: descrição de possíveis cenários para o caso de uso.

10) Fluxos de Excessão: similar ao fluxo alternativo, mas descreve cenários em que algo inesperado ocorreu.

11) Pós-condições: indicação do estado alcançado após o caso de uso (por exemplo: a indicação de que alguma informação foi modificada no sistema).

12) Regras de Negócio: referencia cruzada do caso de uso às regras de negócio. Por exemplo: O valor de um pedido é igual à soma do valor dos itens mais 10% da taxa de entrega. Normalmente aqui vem apenas as siglas das regras de negócio, que devem ser definidas em uma parte suplementar do MCU.

13) Histórico: descrição de autor, data de criação e modificações do caso de uso.

14) Notas de implementação: rascunho de idéias para a implementação.

As partes 13 e 14 não devem ser usadas na validação.

Usualmente, também é feita uma documentação suplementar ao MCU, detalhando as regras de negócio (com seus respectivos identificadores), requisitos de interface (cor, estilos, etc... apenas restrições à interface) e requisitos de desempenho (número de utilizações de um caso de uso em um intervalo de tempo, tempo de resposta máximo, etc).

5. Exemplo de um Caso de Uso de Um Problema Real

Considere o seguinte sistema a ser modelado:

Pretende-se abrir um novo negócio, chamado ClickPizza. A idéia é vender pizza "delivery", da mesma forma com que é feito por telefone, mas através de um site. Resumidamente, os **principais** (mas não todos) requisitos do sistema são:

a) Cadastro de clientes
b) Histórico de Pedidos
c) Permitir venda de Pizza online para os clientes
d) Emitir e aceitar cupons de desconto de clientes que compraram:
2 Pizzas: 10% em uma próxima compra;
3 Pizzas: 15% em uma próxima compra;
4 Pizzas ou mais: 20% em uma próxima compra.
e) Os cupons devem ser representados apenas por letras e números, e só deve ser possível usá-los uma única vez depois que foram emitidos.
f) O sistema deve vender refrigerantes também, mas eles não contam para a emissão de comprovantes.
g) Permitir que gerente crie novos tipos de Pizza para o cardápio, além de adicionar novos refrigerantes.
h) Permitir que o gerente adicione novos ingredientes de Pizza para poder criar novas Pizzas.
i) Permitir que o gerente indique preços aos ingredientes para que eles componham o preço da Pizza. Alternativamente, o gerente pode especificar o preço da Pizza diretamente.
j) O gerente pode desligar um tipo de Pizza específico, sem ter de apagá-lo do cardápio.

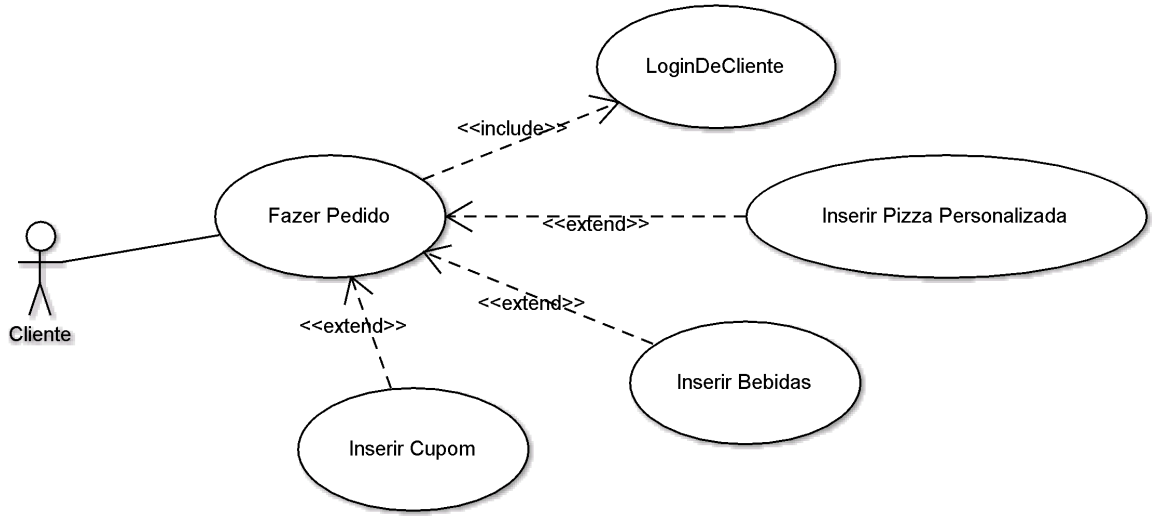
Seria também interessante que o cliente pudesse montar sua "Pizza Pessoal", selecionando os ingredientes que desejasse na Pizza, e o sistema comporia o preço da mesma. Este, não é, um requisito obrigatório.

Baseando-se neste projeto, será apresentado um pedaço do Modelo de Casos de Uso do sistema a ser projetado. Será apresentada tanto a parte gráfica (DCU) quanto a parte descritiva do caso de uso.

Note que, eventualmente, esta descrição não está completa. É apenas um exemplo de onde se começa - e como se começa - a elaboração.

O caso de uso a ser modelado é o *pedido do cliente*. O pedido do cliente certamente deve incluir o caso de uso de *login do cliente*, além de poder ser extendido pela *inserção de pizzas personalizadas*, *inserção de bebidas* e *inserção de cupons de desconto*.

Assim, o diagrama de casos de uso - para este caso de uso específico - deve ter a aparência apresentada na figura abaixo:



Entretanto, esta descrição não é suficiente para a fase posterior, que é a elaboração das classes de análise. É necessário fazer a descrição detalhada destes casos de uso. Esta descrição segue a partir da próxima página.

Descrição de Casos de Uso

Nome: Fazer Pedido
Código: DCU-0001
Prioridade: Alta
Sumário: O cliente (ator) realiza este caso de uso como forma de realizar um pedido, que pode incluir pizzas e bebidas.
Atores
Primário: Cliente
Secundário: -
Pré-Condição:
- Usuário precisa ter cadastro no sistema.

Fluxos
Principal: 1) Inclusão: Cliente realiza login (DCUA-0001)
2) Cliente inicia pedido
3) Sistema cria pedido e solicita qual o pedido.
4) Cliente solicita lista de pizzas.
5) Sistema responde lista de pizzas, incluindo "Personalizada".
6) Cliente escolhe pizza.
7) Sistema responde que pizza foi adicionada e (passo 3) solicita qual o pedido.
8) Extensão: Cliente realiza pedido de pizza personalizada (DCUA-0002)
9) Extensão: Cliente realiza pedido de bebida (DCUA-0003)
A) Extensão: Cliente realiza inserção de cupom (DCUA-0004)
B) Cliente solicita fechamento do pedido.
C) Sistema fornece número do cupom de desconto e responde número do pedido.
Alternativo: Nenhum especial.
Exceção: 1) Inclusão: Cliente realiza login (DCUA-0001)
2) Cliente inicia pedido
3) Sistema cria pedido e solicita qual o pedido.
4) Cliente solicita fechamento do pedido.
5) Sistema responde que pelo menos uma pizza deve ser solicitada.
Pós-Condição:
- O sistema deve armazenar o pedido realizado em um banco de dados de pedidos realizados.

Regras: - Pelo menos uma pizza deve ser pedida (RN-0001)
- Não fornecer cupom de desconto quando o cliente usou cupom de desconto (RN-0006)
- O cálculo do preço no fechamento do pedido deve respeitar à RN-0008.

Histórico: - 01/10/2007 - Daniel Caetano criou o caso de uso.
Implementação:
- As pizzas podem ser incluídas na forma de um loop infinito, até que o usuário selecione fechar o pedido.

Nome: Login de Cliente
Código: DCUA-0001
Prioridade: Alta
Sumário: O cliente (ator) realiza este caso de uso como forma de se identificar no sistema, para poder realizar pedidos.
Atores
Primário: Cliente
Secundário: -
Pré-Condição:
- Usuário precisa ter cadastro no sistema.

Fluxos
Principal:
1) Sistema solicita nome de usuário e senha.
2) Cliente informa nome e senha.
3) Sistema responde que login ocorreu com sucesso.

Alternativo:
1) Sistema solicita nome de usuário e senha.
2) Cliente informa que perdeu senha.
3) Sistema solicita e-mail de cadastro.
4) Cliente informa e-mail de cadastro.
5) Sistema responde que mensagem foi enviada com a senha.

Exceção:
1) Sistema solicita nome de usuário e senha.
2) Cliente informa nome e senha.
3) Sistema responde que login não foi realizado, seja por falta de cadastro ou por nome/senha incorretos.

Pós-Condição:
- O usuário estará autenticado após este passo.

Regras: - O sistema não deve informar especificamante a razão pela qual o login não foi realizado (RN-0002)
Histórico: - 01/10/2007 - Daniel Caetano criou o caso de uso.
Implementação:
- Guardar a senha como um hash usando SHA-1.
- Lembrar de adicionar salting no hash.

Nome: Inserir Pizza Personalizada
Código: DCUA-0002
Prioridade: Média
Sumário: Este caso de uso possibilita a criação de uma pizza personalizada de forma que ela possa ser adicionada ao pedido de um cliente.
Atores
Primário: Cliente
Secundário: -
Pré-Condição:
- Ingredientes precisam estar cadastrados no sistema.
- Usuário precisa estar autenticado no sistema.
- Pedido precisa estar criado.

Fluxos
Principal:
1) Ciente solicita pizza personalizada.
2) Sistema cria uma pizza "vazia", mostra lista de ingredientes disponíveis e solicita quais são os ingredientes da pizza.
3) Cliente informa ingredientes.
4) Sistema calcula preço e pergunta se cliente concorda.
5) Cliente concorda.
6) Sistema informa que pizza foi adicionada.

Alternativo: Nenhum especial.
Exceção: Nenhum especial.
Pós-Condição:
- O sistema deve armazenar a pizza personalizada, associada ao cliente, para futuras referências.

Regras:
- Pelo menos um ingrediente deve ser adicionado (RN-0003)
- O cálculo do preço obedece à RN-0004.

Histórico:
- 01/10/2007 - Daniel Caetano criou o caso de uso.

Implementação:
- Os ingredientes podem aparecer como uma lista de checkboxes.

Nome: Inserir Bebidas
Código: DCUA-0003
Prioridade: Média
Sumário: Este caso de uso possibilita a adição de uma bebida ao pedido de um cliente.
Atores
Primário: Cliente
Secundário: -
Pré-Condição:
- Bebidas precisam estar cadastradas no sistema.
- Usuário precisa estar autenticado no sistema.
- Pedido precisa estar criado.

Fluxos
Principal:
1) Ciente solicita adição de bebida.
2) Sistema cria mostra lista de bebidas disponíveis.
3) Cliente informa as bebidas que deseja (e quantidades).
4) Sistema informa que bebidas foram adicionadas ao pedido.

Alternativo: Nenhum especial.
Exceção: Nenhum especial.
Pós-Condição:
Regras: - As bebidas não entram no cálculo do cupom (RN-0005)
Histórico: - 01/10/2007 - Daniel Caetano criou o caso de uso.
Implementação:
- As bebidas podem aparecer como uma lista de checkboxes.

Nome: Inserção de Cupom
Código: DCUA-0004
Prioridade: Média
Sumário: Este caso de uso possibilita o uso de um cupom de desconto por parte do cliente.
Atores
Primário: Cliente
Secundário: -
Pré-Condição:
- Cupons precisam estar cadastrados no sistema.
- Usuário precisa estar autenticado no sistema.
- Pedido precisa estar criado.

Fluxos
Principal:
1) Ciente solicita adição de cupom.
2) Sistema solicita o número do cupom.
3) Cliente informa o número do cupom.
4) Sistema informa que o cupom foi aceito e o desconto que o cupom dá.

Alternativo: Nenhum especial.
Exceção:
1) Ciente solicita adição de cupom.
2) Sistema solicita o número do cupom.
3) Cliente informa o número do cupom.
4) Sistema informa que o cupom não existe ou já foi usado.

Pós-Condição:
- O sistema deve registar que o cupom já foi usado.
- O pedido fica em um estado em que o deconto do cupom deverá ser atendido.

Regras: - Apenas um cupom por pedido (RN-0007)
Histórico: - 01/10/2007 - Daniel Caetano criou o caso de uso.
Implementação:

Descrição das Regras de Negócio

Nome: Mínimo de Pizzas
Código: RN-0001
Descrição: Não deve ser possível realizar um pedido sem nenhuma pizza.

Nome: Segurança de Login
Código: RN-0002
Descrição: O sistema não deve informar se o login falhou por erro na senha, nem se é por usuário inexistente.

Nome: Mínimo de Ingredientes
Código: RN-0003
Descrição: Não deve ser possível criar uma pizza personalizada sem ingredientes.

Nome: Cálculo de Preço de Pizza Personalizada
Código: RN-0004
Descrição: O preço das pizzas personalizadas deve ser calculado da seguinte maneira.

$$P = SPI * 1.1 + L$$

Onde: P: Preço da Pizza Personalizada
SPI: Soma dos preços dos ingredientes
L: Lucro por unidade de pizza (R$ 5,00)

Nome: Bebidas não têm Desconto de Cupom
Código: RN-0005
Descrição: O valor das bebidas não deve sofrer o desconto do cupom.

Nome: Compras com Cupom não dão Direito a Cupom.
Código: RN-0006
Descrição: Quando um pedido for fechado com um cupom de desconto, o sistema não deve fornecer cupons nesta venda.

Nome: Só um Cupom por Pedido.
Código: RN-0007
Descrição: Apenas um único cupom pode ser usado em um pedido.

Nome: Cálculo do Desconto pelo cupom.
Código: RN-0008
Descrição: O preço do pedido deve ser calculado, com o cupom, da seguinte forma:

$$PF = SPP * (100-D)/100 + SPB + TE$$

Onde: PF: Preço Final do Pedido
SPP: Soma dos Preços das Pizzas do Pedido
D: valor de 0 a 100, sendo o desconto fornecido pelo cupom.
SPB: Soma dos Preços das Bebdidas do Pedido
TE: Taxa de Entrega (R$ 5,00)

Bibliografia

BEZERRA, E. **Princípios de Análise e Projeto de Sistemas com UML**. 2ed. Rio de Janeiro: Editora Campus Elsevier, 2007.

DEITEL, H.M; DEITEL, P.J. **Java**: como programar - 6ed. São Paulo: Pearson-Prentice Hall, 2005.

Etapa 3: Desenvolvimento da Etapa de Análise
Tópico 8: Modelagem das Classes de Análise
Prof. Daniel Caetano

Objetivo: Apresentar os conceitos de Classes de Análise, detalhes dos diagramas de classes da UML, técnica de identificação de classes e apresentação da metodologia CRC.

Bibliografia Básica:

- BEZERRA, E. **Princípios de Análise e Projeto de Sistemas com UML**. 2ed. Rio de Janeiro: Editora Campus Elsevier, 2007.
- DEITEL, H.M; DEITEL, P.J. **Java**: como programar - 6ed. São Paulo: Pearson-Prentice Hall, 2005.

Introdução

Após a correta identificação dos casos de uso, a última etapa da análise constitui em identificar e documentar as chamadas *classes de análise*, criando o *modelo de classes de análise*. Esta é uma das etapas mais complexas do desenvolvimento de um projeto, sendo freqüente o retorno a esta fase com o objetivo de revisar os modelos gerados, no chamado processo incremental (ou iterativo) de desenvolvimento.

Nesta aula veremos o conceito de classes de análise (e no que elas são diferenciadas das classes de projeto) e serão apresentados detalhes de modelagem usando o diagrama de classes da UML. Adicionalmente, serão apresentadas algumas técnicas de identificação de classes (baseadas no modelo de casos de uso) e a metodologia CRC (Classes, Responsabilidades e Colaborações.

1. Estágios do Modelo de Classes

Como foi dito, nesta etapa do desenvolvimento estaremos preocupados com o a identificação e especificação das *classes de análise*... mas o que são estas classes?

Podemos dizer que existem dois níveis básicos de modelagem de classes: modelo de classes de análise e o modelo de classes de projeto. Vejamos qual o enfoque de cada um deles.

Modelo de Classes de Análise: é o modelo que é criado focando a atenção em "o quê" o sistema deve fazer e não deve envolver aspectos sobre a tecnologia de implementação. Juntamente com o Modelo de Casos de Uso, constitui a documentação a ser produzida na fase de análise.

Modelo de Classes de Projeto: é o modelo que é criado focando a atenção em "como" o sistema deve executar determinadas tarefas, levando em conta a tecnologia de implementação. Constitui a documentação fundamental - mas não única - a ser produzida na fase de projeto.

Uma analogia que ajuda a compreender é a da construção civil: no modelo de classes de análise a preocupação é com quais cômodos existirão em uma casa e qual a disposição entre eles. Já no modelo de classes de projeto, a preocupação é com quais os locais em que existirão encanamentos, pontos de luz e água etc.

Nestas aulas o foco é o modelo de classes de análise, que servirá de base para a elaboração futura das classes de projeto.

2. UML na Especificação do Modelo de Classes de Análise

Seguindo a especificação do padrão UML, nestas aulas será apresentada a especificação de classes e de suas ligações estruturais, no chamado "Diagrama de Classes".

Na UML, uma classe é representada por uma caixa retangular que pode ter de 1 a 3 compartimentos: um deles (o mais de cima) sempre será usado para especificar o nome da classe (que deve sempre ser no singular, sem nenhum espaços, sem acentos e com as primeiras letras de cada palavra em letras maiúsculas). Caso exista apenas dois compartimentos, o segundo pode servir tanto para indicar atributos quanto para indicar métodos das classes; Se existirem 3 compartimentos, o segundo será usado para os atributos e o terceiro para operações.

Usualmente há 1 ou 3 compartimentos, sendo menos incomum o uso de 2 compartimentos. Exemplos de classes de análise em diferentes níveis de detalhamento são apresentados na figura 1.

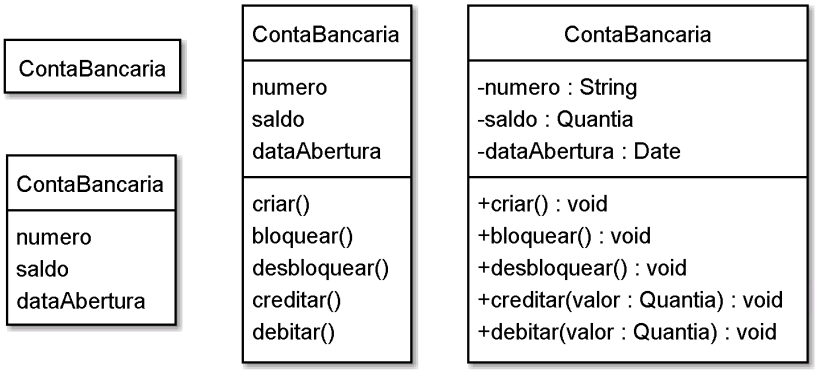


Figura 1: Representação de Classes na UML, em diferentes graus de abstração

Entretanto, as classes isoladas não compõem um modelo de classes (nem mesmo o de análise). É necessário representar também algumas relações entre elas e entre os objetos que delas forem criados. Para isso serão usados elementos de um outro tipo: as associações.

1.1. Associações

Como se sabe, o objetivo das classes é servir de "intruções" para que objetos sejam criados, já que são os objetos que irão interagir para produzir resultado desejado. As associações de um diagrama de classes existem exatamente para identificar e especificar quais tipos de objetos se relacionam (se associam) com quais tipos de objetos. Por exemplo: um *pedido de compra* tem vários *itens*; um *hotel* possui vários *quartos*... e assim por diante. A figura 2 mostra a aparência de uma associação "genérica" em um diagrama de classes.

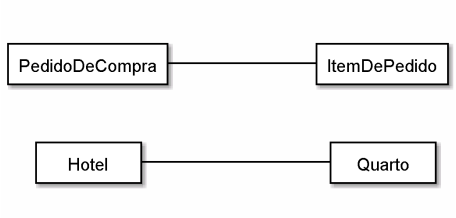


Figura 2: Representação UML de associações simples entre classes.

Esta é uma característica importante porque, apesar de estar representado em um diagrama de classes, as associações indicam ligações entre objetos, o que faz com que algumas outras características possam (e devam) ser definidas nas associações como *multiplicidade*, *nome*, *direção*, *papel* e *conectividade*. Vejamos cada uma a seguir.

1.1.1. Multiplicidades

Multiplicidade é um conceito similar à cardinalidade nos diagramas de entidade e relacionamento (DER), como definido por Peter Chen. Ele indica, em uma associação entre dois objetos, quantos objetos de um tipo podem/devem estar ligados a um objeto de outro tipo. Veja um exemplo na figura 3, sabendo que quando o valor da multiplicidade não é indicado, ele é considerado como sendo 1:

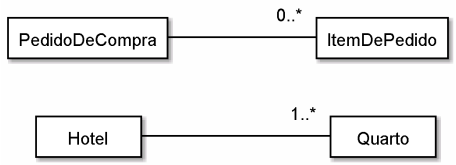


Figura 3: Representação de multiplicidades nas associações

Observe que um *pedido de compra* pode ter nenhum, 1 ou muitos *itens de pedido*. Por outro lado, cada *item de pedido* só pode estar associado a um único *pedido de compra*. O mesmo raciocínio vale para o outro caso: um *hotel* deve ter pelo menos 1 *quarto* (pode ter mais), mas um *quarto* só pode estar associado a um único *hotel*.

Da mesma forma que a cardinalidade, podemos usar as expressões: "um para um", "um para muitos" ou "muitos para muitos" com relação à multiplicidade. Note, adicionalmente, que a especificação de multiplicidade define também a obrigatoriedade/opcionalidade de uma relação. Quando um dos extremos da relação possuir a alternativa de "0" objetos associados, dizemos que a associação é *opcional*. Por outro lado, quando pelo menos um objeto de um tipo precisar estar associado ao outro, dizemos que a associação é obrigatória. Na figura 3 podemos observar que um *pedido* pode ter um *item de pedido*, ou seja, a associação com um *item de pedido* é opcional (um pedido vazio não possui item algum); por outro lado, um *item de pedido* só pode existir associado a um *pedido de compra*, já que um item de pedido não faz sentido sem um pedido, ou seja, a associação com um *pedido de compra* é obrigatória.

1.1.2. Nomes de Associações, Direção de Leitura e Papéis

Alguns elementos podem ser adicionados a uma associação para facilitar seu entendimento. Estes elementos são os nomes, a direção de leitura e os papeis. O nome deve indicar a relação representada pela associação entre um objeto e outro. Uma vez que o nome da relação depende do ponto de vista de qual dos objetos é o ativo, indicamos a direção de leitura.

Algumas vezes, entretanto, o nome da relação apenas não é suficiente para facilitar a compreensão da relação. Nestes casos também é possível identificar os papéis dos objetos, perante aquela associação. Veja um exemplo na figura 4.

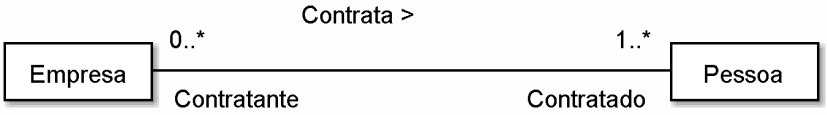


Figura 4: Representação de associação completa

No caso, "Contratante" e "Contratado" são indicações dos papeis dos objetos do tipo "Empresa" e "Pessoa", respectivamente. "Contrata" é o nome da associação e a setinha ">" indica a direção de leitura.

1.1.3. Associações Reflexivas

Algumas vezes precisamos representar que objetos de uma classe se relacionam com objetos desta mesma classe. Por exemplo, que um empregado específico (como um gerente) tem relação com outros empregados quaisquer (como os funcionários deste gerente). A UML tem uma forma especial, reflexiva, de associação, como pode ser visto na figura 5:

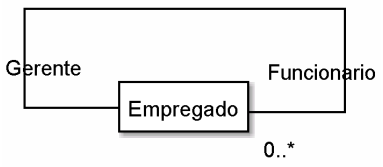


Figura 5: Associação Reflexiva

Obviamente não se deve confundir conceitos: uma associação reflexiva indica que um objeto de uma classe se associa a outros objetos daquela mesma classe; esse tipo de associação não significa que um objeto se associa a ele mesmo.

1.2. Classes Associativas

É importante lembrar que, quando existirem relações do tipo "muitos para muitos" e for necessário guardar informações ou realizar operações referentes à estas relações, será obrigatória a criação de "classes associativas", que serão classes intermediárias na relação. Observe a figura 6:

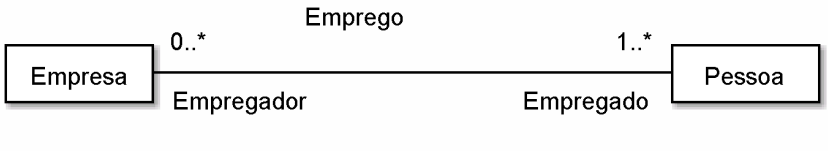


Figura 6: Relação muitos para muitos

Ora, neste caso, a informação "salário" não deve estar presente nem em "Empresa" e nem em "Pessoa"; Salário é uma propriedade, um atributo, da relação "Emprego". Neste caso, criamos uma "classe associativa", que representa as informações da associação, que passará a não mais possuir um nome: o nome estará na classe associativa. Veja na figura 7.

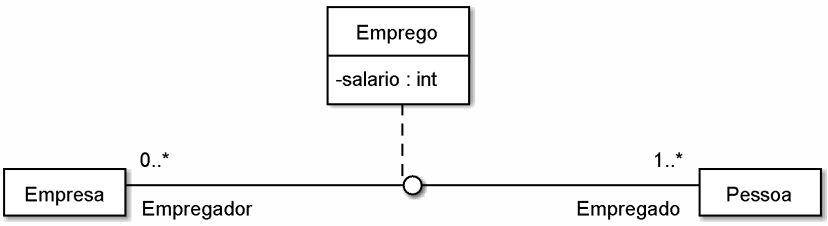


Figura 7: Classe Associativa

1.3. Agregações e Composições

Em muitos casos, as associações entre duas classes significam que os objetos destas possuem uma relação parte/todo. Um exemplo clássico é a associação entre a classe Carro e as classes Motor, Roda etc. Neste caso, os objetos da classe Carro representam o *todo* e os

objetos das outras classes, como Motor e Roda representam as *partes* que estão incluídas no todo.

Nestes casos, em geral os objetos-parte são criados pelo objeto-todo, sendo que se um objeto A é parte de um todo B, o objeto B não pode ser, ao mesmo tempo, parte do A. Além disso, se um objeto-todo é destruído, normalmente os objetos-parte também o são, ao mesmo tempo.

Quando uma associação tem essas características, denominamos as associações como *agregações* ou *composições*, dependendo da situação; chamamos de *agregação* quando o objeto-parte faz sentido mesmo sem a existência do objeto-todo e chamamos de *composição* quando o objeto-parte não faz sentido sem o objeto-todo.

Por exemplo: numa associação entre objetos do tipo "Departamento" e "Funcionário" podemos dizer que se trata de uma agregação, já que mesmo que um funcionário não faça mais parte de um departamento, ele pode continuar sendo um funcionário da empresa. Observe na figura 8 a representação da UML para agregação (um losango branco próximo ao objeto-todo):

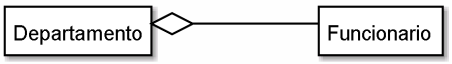


Figura 8: Representação UML para agregação

Outro exemplo: numa associação entre objetos do tipo "Pedido" e "ItemDePedido" podemos dizer que se trata de uma composição, já que a existência de um item de pedido sem a existência de um pedido associado não faz qualquer sentido. Observe na figura 9 a representação da UML para composição (um losango negro próximo ao objeto-todo):

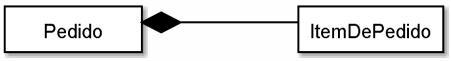


Figura 9: Representação UML para composição

Há situações em que a distinção não é clara e vai depender da interpretação da modelagem. Um exemplo clássico é a associação entre objetos do tipo "Automóvel" e "Motor". Se na aplicação fizer sentido a existência do objeto motor sozinho, ou seja, ele puder ser tirado de um automóvel e colocado em outro, por exemplo (ou mesmo ser deixado em algum outro tipo de repositório), então seria uma agregação. Por outro lado, se na aplicação o motor só fizer sentido enquanto o automóvel no qual ele está existir, então será uma composição.

1.4. Generalizações e Especializações

Até agora falamos das associações que, como foi dito, indicam relações entre objetos das classes associadas. Existe um tipo especial de relacionamento que é, de fato, entre

classes: o relacionamento de generalização/especialização, também conhecido como relacionamento de herança.

Este relacionamento serve para dizer que uma determinada classe é a especialização da outra. Em outras palavras, se a classe B é uma especialização de A, então todas as características da classe A estão também presentes na classe B; mais que isso, todos os objetos de classes que se relacionam com objetos da classe A (mais genérica) também serão capazes de se relacionar com objetos da classe B (mais especializada). Podemos dizer, então, que todo objeto que é da classe B *é também da classe A*. O inverso, entretanto, não é verdadeiro.

A classe mais genérica é comumente chamada de "superclasse" e a classe especializada é chamada de "subclasse". Com estes termos fica mais fácil expressar a relação entre tais classes:

- A subclasse é uma *especialização* da superclasse.
- A superclasse é uma *generalização* da subclasse.

Exemplos sempre são bem vindos para tornar claro alguns destes conceitos. O primeiro deles, o mais clássico, refere-se à relação entre as classes Veículo e Carro. Carro certamente é uma definição mais específica que Veículo; Entretanto, é de se esperar que Carro possua todas as características que um Veículo possui, uma vez que faz total sentido dizer que objeto do tipo Carro é também um objeto do tipo Veículo, certo? Na UML, representamos esta relação conforme apresentado na figura 10.

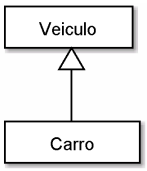


Figura 10: Representação UML para generalização/especialização

1.4.1. Classes Abstratas

Em alguns casos, percebemos que algumas classes possuem muitas coisas em comum e percebemos que é interessante agrupar todas as características comuns em uma mesma classe, especializando esta classe mais genérica (que contém as semelhanças entre as classes em questão). Entretanto, observamos que não faria sentido criar objetos desta nova classe. Nestes casos, chamamos esta classe de *abstrata*.

Vamos considerar um exemplo. Imagine uma classe que represente uma ContaBancaria. Esta classe pode estar relacionada a uma classe PessoaFisica ou a uma classe PessoaJuridica, mas nunca às duas ao mesmo tempo.

Ora, tanto a PessoaFisica quanto a PessoaJuridica possuem características distintas, mas algumas delas (como nome, data de cadastro, endereço etc) existem em ambas. Estas

características poderiam, então, ser agrupadas em uma outra classe chamada *Cliente*, por exemplo, sendo esta, então, especializada pelas classes ClientePessoaFisica e ClientePessoaJuridica.

Entretanto, apesar de existir a classe *Cliente*, não faz sentido a existência de um objeto desta classe pura, que não seja nem uma pessoa física e nem uma pessoa jurídica. Por esta razão, determinamos que esta classe *Cliente* é uma classe abstrata e, na UML identificamos isto escrevendo o nome da classe em texto *itálico*. Veja a situação ilustrada nos últimos parágrafos na figura 11:

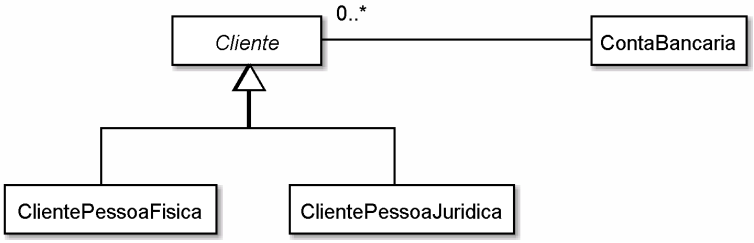


Figura 11: Uso de especialização com classe abstrata

Observe que não faz sentido indicar multiplicidade na relação de herança (generalização/especialização), pois é uma relação entre classes (e não entre objetos destas classes).

Observe que muitas vezes é preciso tomar cuidado com o uso de especializações. Muitas vezes somos tentados a modelar um *papel* de um objeto como uma classe própria (por exemplo, numa relação entre a classe Pessoa e a classe Edifício, inserir uma classe "Morador" no meio, sendo que Morador é apenas o *papel* do objeto da classe Pessoa naquela relação), mas isso não é correto.

É possível que alguém argumente que foi exatamente isso que foi feito no caso da relação entre Cliente e ContaBancaria, representada na figura 11: Cliente é o papel que a Pessoa Física ou a Pessoa Jurítica tem perante a Conta Bancária. Entretanto, naquele caso temos dois tipos diferentes de clientes, fazendo sentido representar o papel como uma classe abstrata; Se houvesse apenas um único tipo de cliente, não faria sentido inserir um intermediário.

2. Técnicas de Identificação de Classes

É claro que saber representar as classes e suas relações e associações é algo necessário para gerar um modelo de classes de análise. Entretanto, isso não é suficiente: para criar o modelo será necessário identificar as classes que irão compor o sistema a ser modelado, bem como a relação entre elas.

Para realizar esta identificação, serão usados os documentos já existentes: o documento de especificação de requisitos **mais** o modelo de casos de uso. O primeiro passo é

dar uma boa lida no documento de especificação de requisitos, prestando atenção aos substantivos que podem se tornar classes e ações que podem se tornar métodos.

Entretanto, esta é só uma fase inicial para identificar alguns dos possíveis elementos, a fim de embasar as próximas etapas, que são a identificação pelos diagramas de casos de uso e a técnica CRC.

2.1. Análise de Casos de Uso

A Análise de Casos de Uso é a técnica fundamental para identificação de classes no processo RUP (Rational Unified Process).

Os passos para a execução desta análise são:

a) Complementar as descrições dos casos de uso, visando facilitar a identificação de todas as classes envolvidas.
b) Em cada caso de uso, identifique as possíveis classes que compõem a funcionalidade, distribuindo (posteriormente) o comportamento do caso de uso pelas classes identificadas.
c) Para cada classe resultante do fim da etapa b, descreva suas responsabilidades e, posteriormente, seus atributos e associações.
d) Unifique as classes em um diagrama de classes.

A etapa de complementação de descrições é, na realidade, o último passo da geração do modelo de casos de uso.

Na segunda etapa (identificação de classes nos casos de uso), o modelador deverá selecionar um caso de uso específico, analizar seus textos (fluxos principal e alternativos) e tentar identificar quais seriam as classes candidatas a participar daquele caso de uso, ou seja, classes que podem fornecer o comportamento do mesmo, quando combinadas. Quando as classes do primeiro caso de uso estiverem determinadas, parte-se para o próximo, até que todas (ou a grande maioria) das classes tenham sido determinadas.

A explicação da RUP para este procedimento é que "a existência de uma classe só pode ser justificada se ela participar de alguma forma do comportamento externamente visível do sistema" (BEZERRA, 2007, pág 139).

Esta não é uma etapa simples, e por isso acrescentamos aqui alguns conceitos introduzidos por Ivar Jacobson em 1992, chamada de Análise de Robustez (que não faz parte do RUP, mas não há problema algum em ser utilizada.

2.1.1. Análise de Robustez e a Categorização BCE

Na Análise de Robustez tentamos dividir a identificação de objetos em 3 tipos de objetos: objetos de fronteira (Boundary), objetos de controle (Control) e objetos de entidade

(Entity), o que é chamado de "Categorização BCE". A aplicação desta técnica de identificação em cada um dos casos de uso pode facilitar muito a vida do modelador, como veremos a seguir.

Antes de mais nada, vale lembrar que a UML fornece estereótipos padrão para identificar classes destes três tipos, sendo eles: « boundary », « control » e « entity ».

2.1.1.1 Objetos de Fronteira

Objetos de fronteira são aqueles que propicial a comunicação dos atores com o sistema. Em outras palavras, eles provém a interface de interação entre os atores (humanos ou não) com o sistema a ser desenvolvido.

Objetos de fronteira possuem, normalmente, as seguintes responsabilidades:

a) Notificar aos outros objetos do sistema sobre eventos externos ao sistema (vindos dos atores).
b) Notificar aos atores os resultados de interações de objetos do sistema.

Tipicamente, existe uma classe de fronteira para cada ator que se comunica com cada caso de uso. Os nomes de classes de fonteira não devem conter verbos, devendo ser substantivos que lembrem qual é o canal de comunicação que ele representa, como por exemplo: FormularioCadastro ou LeitoraCodigoBarras. Adicionalmente, na fase de identificação das classes, ninguém deve se preocupar com o conteúdo destas classes. Como a interação será feita, se é por interface gráfica ou não e outras decisões do tipo deverão ficar para momentos futuros.

Os objetos de fronteira normalmente são *transientes*, isto é, existem apenas brevemente na memória do sistema, no momento em que são usados. Logo em seguida são destruídos e recriados novamente quando necessário.

2.1.1.2 Objetos de Controle

Objetos de Controle são uma espécie de "despachante" que cuida da comunicação entre os objetos de fronteira e os objetos de entidade. Ele tem a responsabilidade de realizar as comunicações na ordem correta de forma que o objetivo pretendido pelo usuário ao interagir com o sistema seja cumprida. Em outras palavras, eles coordenam a execução de uma funcionalidade específica do sistema.

Os objetos de controle possuem, normalmente, as seguintes responsabilidades:

a) Monitorar os objetos de fronteira, a fim se responder a eventos externos.
b) Coordenar a execução de um caso de uso, trocando mensagens com objetos de fronteira e de entidade.
c) Criar associações entre alguns objetos de entidade.

d) Manter registro de alguns valores especiais relativos ao caso de uso.
e) Manter o estado atual do caso de uso.

Tipicamente, existe uma classe de controle para cada caso de uso. Os nomes de classes de controle devem lembrar o nome do caso de uso pelo qual ela é responsável, como por exemplo: GerenciadorCadastros ou MarcadorTempo.

Os objetos de controle normalmente são, também, *transientes*. Eles existirão apenas durante a execução de um caso de uso.

2.1.1.3 Objetos de Entidade

Objetos de Entidade são representações de conceitos envolvidos no domínio do problema a ser resolvido. Podem servir como repositórios de informações que são manipuladas e freqüentemente envolvem as regras/lógica de negócio.

Os objetos de entidade possuem, normalmente, as seguintes responsabilidades:

a) Informar o valor de seus atributos a outros objetos.
b) Realizar cálculos (e impor restrições) relativas às regras de negócio (por exemplo, um objeto Pessoa não deve ter um CPF inválido)
c) Criar e destruir seus objetos-parte.

Os objetos de entidade normalmente são usados em diversos casos de uso e, portanto, são manipulados por diversos objetos de controle. Os nomes de classes de entidade variam enormemente, já que representam conceitos do domínio do problema, mas normalmente são substantivos deste domínio, como por exemplo: Nota, Prato e Veículo.

Estes objetos costumam ser *permanentes*, existindo por tempos quase tão longos quanto o próprio sistema. É comum a co-existência de diversos objetos de cada classe de entidade. Estes objetos são, normalmente, armazenados em um banco de dados e são, muitas vezes, denominados *objetos persistentes*.

2.2. Identificação de Responsabilidades

No curso passado vimos uma técnica chamada CRC: Classes, Responsabilidades, Colaboradores. Este método consistia em identificar possíveis classes e responsabilidades e, posteriormente, colaboradores através de uma espécie de brain storm. Esta metodologia pode ser também usada aqui, individualmente para cada caso de uso, lembrando que as classes já identificadas em um caso de uso anterior podem voltar a ser usadas em um caso de uso seguinte.

Os passos deste método são:

a) Para cada caso de uso, realizar uma sessão CRC.

b) No início da sessão, distribuir cartões (modelo na figura 12) para os membros do grupo (até 6 membros), procurando não passar de muito mais de um cartão por membro (2 ou mais são admitidos, mas deve ser evitado).

c) O grupo deve tentar identificar as classes envolvidas no caso de uso, podendo usar as técnicas previamente citadas. Cada membro irá adotar uma das classes para si e escrever o nome da classe no cartão.

d) A partir da análise do caso de uso, cada membro irá escrever no cartão quais as responsabilidades que acredita ser de sua classe.

e) Todos os membros debatem se as responsabilidades de cada classe estão adequadas, se não é necessário mover nenhuma delas para outra classe ou mesmo criar novas classes.

f) Cada membro analisa novamente o caso de uso e as responsabilidades já definidas e preenche no cartão os colaboradores para a realização daquele caso de uso (outras classes com as quais será necessário interagir para que as responsabilidades daquela classe possam ser cumpridas).

g) Todos os membros debatem se os colaboradores de cada classe estão adequados, se não é necessário adicionar nenhum ou remover nenhum.

Finalmente, com as classes, responsabilidades e colaboradores preenchidos, um dos membros simula o ator primário e cada membro atua como a sua classe, avaliando se as responsabilidades e colaboradores preenchidos são realmente suficientes para realizar a funcionalidade daquele caso de uso.

Nome da Classe	
Responsabilidade 1	Colaborador 1
Responsabilidade 2	Colaborador 2
Responsabilidade 3	Colaborador 3
Responsabilidade 4	Colaborador 4
Responsabilidade 5	Colaborador 5
Responsabilidade 6	Colaborador 6
Responsabilidade 7	Colaborador 7

Figura 12: Modelo de cartão CRC

2.2.1. Dicas para Atribuição de Classes e Responsabilidades

As sessões CRC são bastante dinâmicas e muitas coisas podem mudar de uma sessão para outra, ainda que o caso de uso em debate seja o mesmo. Algumas dicas para evitar "andar em círculos" são apresentadas agora.

a) Associe responsabilidades com base nas especialidades da classe. Uma pizza deve saber quais são seus ingredientes, mas não faz sentido a pizza informar o valor total da fatura, por exemplo.

b) Distribua a inteligência do sistema. É inadequado que algumas classes tenham dezenas de responsabilidades e outras nenhuma. Quando ocorrer de algumas classes possuírem muitas responsabilidades, tente quebrá-las. Por exemplo: um Pedido sabe seu valor total, um ItemDePedido sabe seu subtotal e um Produto sabe seu valor. O subtotal do ItemDePedido é calculado solicitando os valores dos Produtos que dele fazem parte e o total do Pedido é calculado solicitando os subtotais dos ItensDePedido que dele fazem parte.

c) Agrupe responsabilidades conceitualmente relacionadas. Responsabilidades que são conceitualmente relacionadas devem ser mantidas em uma mesma classe. Assim, as informações que a classe Cliente deve conhecer devem estar na classe Cliente. Por outro lado, às vezes criamos classes diferentes com o mesmo possível conteúdo, como classes Pedido e Fatura, já que elas representam *conceitos distintos*.

d) Evite responsabilidades redundantes. Se mais de uma classe possui a mesma responsabilidade, avalie se não faz mais sentido eliminar a responsabilidade de uma delas e, quando esta precisar da informação, deverá solicitar à outra. Em alguns casos até mesmo valerá a pena criar uma nova classe para esta responsabilidade.

3. Construção do Modelo de Classes

Depois de finalizada a identificação de classes e das responsabilidades, os modeladores devem ser capazes de justificar a existências destas classes e de suas responsabilidades. Caso contrário, é melhor "voltar para a prancheta".

Quando todos estiverem seguros com as classes definidas, inicia-se o processo de mapeamento de classes e responsabilidades para os diagramas de classes.

3.1. Definição de Propriedades

Propriedades é um termo genérico que se refere tanto aos atributos quanto às operações de uma classe. A idéia aqui é transformar as responsabilidades em atributos e operações.

Inicialmente não estaremos muito preocupados com as ações executadas por uma classe, visto que isso estará definido mais claramente através de diagramas futuros. Assim, iremos nos preocupar, neste instante, na identificação dos atributos.

Definição de Atributos

Normalmente, toda responsabilidade de **conhecer** acaba por ser mapeada como um atributo ou uma associação. Após mapear as responsabilidades para atributos, deve-se verificar:

a) Este atributo guarda um valor atômico? (quantidade, quantia, nome etc)
b) Este atributo não contém estrutura interna? (algum dado complexo que não possa ser representado por um número ou uma string, por exemplo)
c) Este atributo faz sentido para *todos* os objetos da classe? (salário não deve estar em Pessoa, pois nem toda Pessoa tem salário!)

Se a) e b) forem respondidas com "não", então talvez seja o caso de criar um novo objeto para representar este atributo, transformando este atributo em uma associação. Se c) foi respondida com "não", talvez este atributo devesse estar em uma outra classe, com a qual esta classe atual teria que colaborar (estar associada).

Definição de Associações

Além de responsabilidades do tipo conhecer, que podem se tornar associações como visto anteriormente, as colaborações via de regra se tornam também associações, já que um objeto precisa conhecer o outro para poder lhe solicitar alguma informação ou operação.

Classes de Associação podem surgir de responsabilidades de "saber" e, algumas vezes, de "fazer" que existem mas não foi possível associar a qualquer classe específica.

3.2. Descrição Textual de Classes

Uma parte importante do Modelo de Classes de Análise é a parte textual do diagrama, que descreve de forma breve cada classe, com uma pequena frase descrevendo o que a classe é e uma breve descrição de cada atributo e operação.

É importante lembrar que os nomes de classe devem estar sempre no singular, devem ser iniciados com letra maiúscula e os nomes devem ser provenientes do domínio da aplicação.

Assim como no modelo de casos de uso, é útil haver uma seção para detalhes de implementação, como um arquivo de notas a ser utilizado futuramente, na implementação.

Após a realização desta documentação, deve-se voltar e analisar sua coerência com o modelo de casos de uso e, se necessário, corrigir qualquer discrepância.

4. Exemplo de Modelo de Classes de Análise

Baseando-se na idéia do projeto apresentado nas aulas anteriores, para o qual foi feita uma parte do MCU, nesta aula será apresentado um pedaço do Modelo de Classes de Análise relativa àquele MCU. Será apresentada principalmente a parte de identificação de classes e responsabilidades e alguma coisa sobre o diagrama de classes de análise.

Note que certamente esta descrição não está completa. É apenas um exemplo de onde se começa - e como se começa - a elaboração.

O diagrama de classes de análise do sistema será modelado com base somente no caso de uso *pedido do cliente*, conforme visto na aula anterior.

O primeiro passo é identificar algumas possíveis classes observando a descrição do caso de uso principal *pedido do cliente*.

Neste caso de uso existe uma pré-condição:

- Usuário precisa ter cadastro no sistema.

Para que esta condição seja verificada, é preciso ter um objeto de uma classe **Cliente**, que será responsável pela autenticação dos clientes e conterá demais informações relevantes sobre o cliente (como endereço de entrega). Será uma classe de objetos de entidade << entity >>.

Continuando com o caso de uso, temos:

1) Inclusão: Cliente realiza login (DCUA-0001)

Isto significa que, se chamamos outro caso de uso, ele deve ter um controlador, que aqui vamos especificar como uma classe **IdentificadorDeCliente** (<< control >>), que deverá proceder o caso de uso de login de cliente e devolver o objeto de cliente correto.

Continuando...

2) Cliente inicia pedido

Isso significa que precisamos de um objeto de uma classe **Pedido** << entity >>, que será responsável por armazenar o conteúdo do pedido, além de contabilizar o custo total e preço para o cliente, considerar descontos etc.
4) Cliente solicita lista de pizzas.
5) Sistema responde lista de pizzas, incluindo "Personalizada".

Este passo significa duas coisas: primeiro que é necessário um objeto que contenha a coleção das pizzas, algo como um objeto **Cardapio** << entity >>, que deve ser

capaz de devolver uma coleção de objetos **Pizza** << entity >> de acordo com a necessidade (aqui, apenas as que estão disponíveis mais uma pizza "vazia" ou de uma classe diferente **PizzaPersonalizada** << entity >>).

8) Extensão: Cliente realiza pedido de pizza personalizada (DCUA-0002)

Este passo significa que, se chamamos outro caso de uso, ele deve ter um controlador, que aqui vamos especificar como uma classe **CriadorDePizzaPersonalizada** (<< control >>), que deverá proceder o caso de uso *inserir pizza* e devolver a **PizzaPersonalizada** com os **Ingredientes** << entity >> corretos inseridos.

9) Extensão: Cliente realiza pedido de bebida (DCUA-0003)

Este passo significa que, se chamamos outro caso de uso, ele deve ter um controlador, que aqui vamos especificar como uma classe **SeletorDeBebidas** (<< control >>), que deverá proceder o caso de uso *inserir bebida* (ou talvez um nome melhor fosse *selecionar bebida*, neste contexto) e devolver uma coleção de objetos do tipo **Bebida** << entity >>.

A) Extensão: Cliente realiza inserção de cupom (DCUA-0004)

Este passo significa que, se chamamos outro caso de uso, ele deve ter um controlador, que aqui vamos especificar como uma classe **IdentificadorDeCupom** (<< control >>), que deverá proceder o caso de uso *inserir cupom* e, caso o cupom seja verdadeiro (e válido), devolver o objeto do tipo **Cupom** << entity >>, que contém, dentre outras coisas, o valor do desconto.

C) Sistema fornece número do cupom de desconto e responde número do pedido.

Este passo significa que objetos do tipo **Cupom** precisam ter um número único, que precisam conhecer. Adicionalmente, objetos do tipo **Pedido** também precisam conhecer seu próprio número de identificação, único.

Mais adiante existe uma pós condição:

- O sistema deve armazenar o pedido realizado em um banco de dados de pedidos realizados.

Esta pós condição exige que exista um objeto de uma classe do tipo **ArmazenadorDePedido**s << control >>, que irá ser responsável pelo armazenamento de objetos do tipo **Pedido**.

Bem, com isso foi feita uma exploração inicial das classes dos possíveis objetos necessários. Entretanto, ainda faltam objetos. Vamos analisar pelo ponto de vista da Análise

de Robustez, ou seja, verificar os objetos faltantes das categorias de fronteira, controle e entidade.

Acredito que foram identificados boa parte dos objetos de entidade (se não todos) e provavelmente os de controle, já que indicamos um objeto de controle para cada caso de uso mencionado (e esta é a regra), menos para o controle do caso de uso atual. O controlador do caso de uso atual pode ser o **RealizadorDePedido** << control >>.

Por outro lado, ainda não definimos nenhum objeto de fronteira. A regra de objetos de fronteira é ter um tipo deles para cada ator do caso de uso. Neste caso de uso temos apenas um ator: o cliente do sistema. Assim, o objeto deve ser do tipo **FormulárioDePedido** << boundary >>, que deve ser responsável por coletar e informar tudo que for importante ao usuário durante a execução do pedido.

Agora vamos organizar todas estas classes e definir mais claramente suas responsabilidades e suas colaborações (isto deve ser feito pelo método CRC).

Classes de Fronteira

- FormularioDePedido << boundary >>

Responsabilidades:

- Conhecer o cardápio
- Apresentar lista de pizzas
- Coletar pizzas selecionadas pelo usuário.
- Permitir acionamento de adição de bebidas.
- Permitir acionamento de adição de pizza personalizada.
- Permitir acionamento de inserção de cupom.
- Devolver lista de pizzas selecionadas.

Colaborações:

- Pedido
- Pizza
- Bebida
- PizzaPersonalizada
- Cupom
- Cardapio
- ListaDePizzas(?)
- ListaDeBebidas(?)

Classes de Controle

- RealizadorDePedido << control >>

Responsabilidades:

- Criar e conhecer objeto Pedido.
- Adicionar lista de Pizzas ao pedido.
- Adicionar lista de Bebidas ao pedido.
- Adicionar Cupom ao pedido.
- Informar ao ArmazenadorDePedidos quando pedido for fechado.

Colaborações:

- FormularioDePedido
- CriadorDePizzaPersonalizada
- SeletorDeBebidas
- ArmazenadorDePedidos
- IdentificadorDeCupom
- Pedido
- Pizza
- PizzaPersonalizada
- ListaDePizzas(?)
- Bebida
- ListaDeBebidas(?)
- Cupom.
- ArmazenadorDePedidos

- IdentificadorDeCliente << control >>

Responsabilidades:

- Autenticar usuário.
- Devolver objeto Cliente autenticado apropriado.

Colaborações:

- Cliente

- CriadorDePizzaPersonalizada << control >>

Responsabilidades:

- Permitir a adição de ingredientes em uma pizza personalizada e responder a PizzaPersonalizada pronta ao fim do processo.

Colaborações:

- Ingredientes
- PizzaPersonalizada

- SeletorDeBebidas << control >>

Responsabilidades:

- Permitir a seleção de bebidas.
- Devolver uma coleção de bebidas selecionadas.

Colaborações:

- ListaDeBebidas(?)
- Bebida

- ArmazenadorDePedidos << control >>

Responsabilidades:

- Armazenar objetos de pedido completos no banco de dados.
- Conhecer o banco de dados.

Colaborações:

- Pedido
- Bebida
- Pizza
- PizzaPersonalizada
- Cupom
- PedidosDB(?)

IdentificadorDeCupom << control >>

Responsabilidades:

- Autenticar cupom.
- Devolver objeto cupom autenticado apropriado.

Colaborações:

- Cupom

Classes de Entidade

- Cliente << entity >>.

Responsabilidades:

- Conhecer o nome do cliente.
- Conhecer a identificação do cliente.
- Conhecer o telefone do cliente.
- Conhecer o endereço do cliente.

Colaborações:

- ?

- Pedido << entity >>

Responsabilidades:

- Conhecer os itens do pedido.
- Conhecer o preço total do pedido.

Colaborações:

- Pizza
- PizzaPersonalizada
- Bebida
- Cupom

- Cardapio << entity >>

Responsabilidades:

- Informar pizzas disponíveis (habilitadas).
- Informar todas as pizzas.
- Informar bebidas disponíveis (habilitadas).
- Informar todas as bebidas.
- Informar ingredientes disponíveis (habilitados).
- Informar todos os ingredientes.

Colaborações:

- Ingrediente
- Pizza

- PizzaPersonalizada
- Bebida

- Pizza << entity >>

Responsabilidades:

- Conhecer seu nome.
- Conhecer seus ingredientes.
- Conhecer sua disponibilidade.
- Conhecer seu preço (informado pelo gerente).
- Conhecer seu custo (cálculo pelos ingredientes + taxas).

Colaborações:

- Ingrediente

- PizzaPersonalizada << entity >>

Responsabilidades:

- Conhecer seu nome.
- Conhecer seus ingredientes.
- Conhecer sua disponibilidade.
- Conhecer seu preço (informado pelo gerente).
- Conhecer seu custo (cálculo pelos ingredientes + taxas).

Colaborações:

- Ingrediente

- Ingrediente << entity >>

Responsabilidades:

- Conhecer seu nome.
- Conhecer seu custo (preço).
- Conhecer sua disponibilidade.

Colaborações:

- ?

- Bebida << entity >>

Responsabilidades:

- Conhecer seu nome.
- Conhecer seu custo (preço).
- Conhecer sua disponibilidade.

Colaborações:

- ?

- Cupom << entity >>

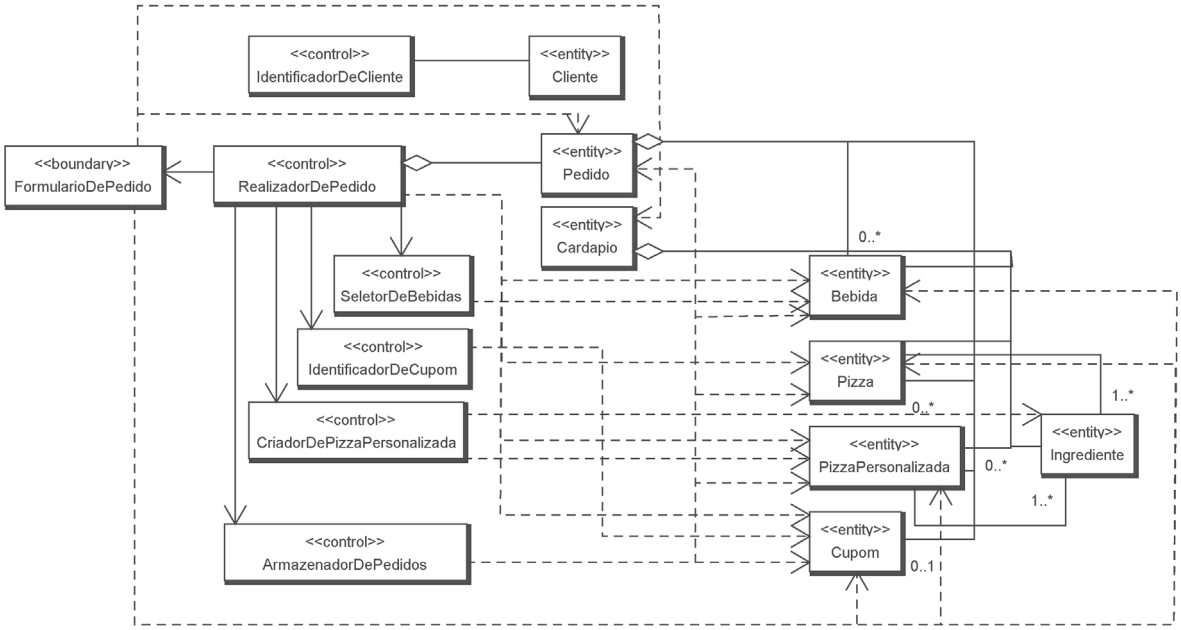
Responsabilidades:

- Conhecer seu número.
- Saber se validar.

Colaborações:

- CupomDB?

O diagrama preliminar que pode ser construído está representado a seguir.



Bibliografia

BEZERRA, E. **Princípios de Análise e Projeto de Sistemas com UML**. 2ed. Rio de Janeiro: Editora Campus Elsevier, 2007.

DEITEL, H.M; DEITEL, P.J. **Java**: como programar - 6ed. São Paulo: Pearson-Prentice Hall, 2005.

Unidade 1: Identificação de Classes

Prof. Daniel Caetano

Objetivo: Apresentar técnicas de identificação de classes e apresentação da metodologia CRC.

Bibliografia: BEZERRA; DEITEL e DEITEL.

INTRODUÇÃO

Conceitos Chave:

- Problema: Casos de Uso => Classes?
- Modelo de Classes de Análise: modelo de classes elementar
- Identificação de classes x Identificação de responsabilidades

Uma equipe de análise criou os casos de uso de um sistema que precisa ser desenvolvido e a tarefa de identificar as classes para dar prosseguimento ao projeto precisa ser realizada. O objetivo é elaborar o *modelo de classes de análise*, incluindo os diagramas UML e as descrições das classes.

Os modelos de casos de uso não fornecem as informações diretamente e é nesta fase que a parte mais complexa do trabalho tem início. É preciso identificar quais elementos são necessários para que cada um dos casos de uso possam ser executados pelo software a ser desenvolvido. Os elementos necessários serão ainda discutidos em um nível elementar, sendo suficiente explicitar sua categoria (classe) e suas principais responsabilidades.

Mas como identificar essas classes e suas responsabilidades?

1. TÉCNICAS DE IDENTIFICAÇÃO DE CLASSES

Conceitos Chave:

- Identificação de Classes => Processo mais complexo!
- Análise de Casos de Uso
- RUP x Análise de Robustez
    * Entidade => Estruturas de Dados
        = Nome => Substantivo
        = Existência => Persistente
        = Estereótipo => « entity ».
    * Controle => Gerenciar Função (Caso de Uso)
        = Nome => Ação que Desempenha
        = Existência => Não-Persistente
        = Estereótipo => « control ».
    * Fronteira => Interação com Usuário
        = Nome => Formulário que Representa
        = Existência => Não-Persistente
        = Estereótipo => « boundary ».
- Responsabilidades
    * Entidade => Conhecer / Informar
    * Controle => Coordenar / Monitorar
    * Fronteira => Interagir / Notificar

Antes que se possa partir para a representação gráfica das classes usando a UML, é necessário identificar as classes que serão representadas e suas inter-relações. Mas esta identificação deve ser feita "do nada"?

Na verdade, não. Para realizar esta identificação, serão usados os documentos já existentes: o documento de especificação de requisitos **mais** o modelo de casos de uso. O primeiro passo é dar uma boa lida no documento de especificação de requisitos, prestando atenção aos substantivos que podem se tornar classes e ações que podem se tornar métodos.

Entretanto, esta é só uma fase inicial para identificar alguns dos possíveis elementos, a fim de embasar as próximas etapas, que são a identificação pelos diagramas de casos de uso e a técnica CRC.

1.1. Análise de Casos de Uso

A Análise de Casos de Uso é a técnica fundamental para identificação de classes no processo RUP (Rational Unified Process). O processo de identificação indicado pelo RUP é complexo e requer uma boa prática; por esta razão, não será explorada aqui esta abordagem.

Os passos para a identificação na metodologia RUP são:

a) Em cada caso de uso, identifique as possíveis classes que compõem a funcionalidade, distribuindo (posteriormente) o comportamento do caso de uso pelas classes identificadas.
b) Para cada classe resultante do fim da etapa **a**, descreva suas responsabilidades e, posteriormente, seus atributos e associações.
c) Unifique as classes em um diagrama de classes.

A primeira etapa (identificação de classes nos casos de uso), o modelador deverá selecionar um caso de uso específico, analizar seus textos (fluxos principal e alternativos) e tentar identificar quais seriam as classes candidatas a participar daquele caso de uso, ou seja, classes que podem fornecer o comportamento do mesmo, quando combinadas. Quando as classes do primeiro caso de uso estiverem determinadas, parte-se para o próximo, até que todas (ou a grande maioria) das classes tenham sido determinadas.

A explicação da RUP para este procedimento é que "a existência de uma classe só pode ser justificada se ela participar de alguma forma do comportamento externamente visível do sistema" (BEZERRA, 2007, pág 139).

Neste curso serão apresentados alguns conceitos de uma outra metodologia, que não faz parte da RUP, chamada *Análise de Robustez*. Esta metodologia foi criada por um dos papas da orientação ao objetos, Ivar Jacobson, em 1992.

O fundamento desta análise que facilita a identificação de classes é a classificação das classes em três diferentes categorias, que podem ser identificadas a partir de análises específicas (e diferentes) dos modelos de casos de uso. Mas qual é essa classificação tão útil?

Na Análise de Robustez tentamos dividir a identificação de classes em 3 categorias: classes de objetos de fronteira (Boundary), classes de objetos de controle (Control) e classes de objetos de entidade (Entity). As iniciais dos nomes (em inglês) das categorias dão o nome ao processo, chamado de "Categorização BCE".

A ordem de identificação será:

1) Classes de objetos de entidade;
2) Classes de objetos de controle;
3) Classes de objetos de fronteira.

Serão apresentados os três casos, mas neste curso há interesse apenas nas duas primeiras categorias, sendo a terceira apresentada apenas para comodidade do leitor interessado.

1.1.1. Classes de Objetos de Entidade

Como Identificar: Objetos de Entidade são representações de conceitos envolvidos no domínio do problema a ser resolvido. Podem servir como repositórios de informações que são manipuladas e freqüentemente envolvem as regras/lógica de negócio.

Em palavras simples, os objetos de entidade são, em geral, aqueles objetos que representam estruturas de dados do problema a ser resolvido. Se o sistema é um controle de clientes, então haverá objetos da classe *Cliente* (que irá armazenar os dados de um cliente) e da classe *Cadastro* (que irá armazenar os objetos do tipo Cliente existentes - é uma *Collection* do java), por exemplo.

A identificação deste tipo de objeto é muito similar à identificação de relações (ou "tabelas") em um projeto de Banco de Dados, quando o objetivo é compor um *Diagrama* ***Entidade****-Relacionamento*.

Como Nomear: Os nomes de classes de entidade variam enormemente, já que representam conceitos do domínio do problema, mas normalmente são substantivos deste domínio, como por exemplo: Nota, Prato, Veículo etc.

Estereótipo UML: Na UML, o estereótipo padrão é « entity ».

Onde São Usados: Os objetos de classes de entidade são usados, normalmente, em vários casos de uso. Assim, o objeto da classe Cliente do cadastro de clientes pode ser usado pelo objeto da classe Fatura, para indicar quem é o comprador de um produto, por exemplo.

Por Quanto Tempo Existem: Por possuírem uma característica de "armazenamento de dados" tal qual num banco de dados, estes objetos costumam ser *permanentes*. Isso significa que eles existem por um tempo tão longos quanto o próprio sistema (normalmente não são apagados). Para conseguir este "efeito", estes objetos são, normalmente, armazenados em um banco de dados de fato, sendo denominados *objetos persistentes*.

1.1.2. Classes de Objetos de Controle

Como Identificar: Objetos de Controle são uma espécie de "despachante" que realizam as tarefas inportantes do sistema. Em palavras simples, estes objetos serão responsáveis por executar as principais tarefas do sistema, controlando a execução desta atividade. Como cada atividade é representada por um caso de uso, existe, tipicamente, uma classe de controle para cada caso de uso. Existindo o Modelo de Casos de Uso, a identificação delas é, portanto, imediata.

Como Nomear: Os nomes de classes de controle devem lembrar o nome do caso de uso pelo qual ela é responsável, como por exemplo: GerenciadorCadastros ou MarcadorTempo.

Estereótipo UML: Na UML, o estereótipo padrão é « control ».

Onde São Usados: Os objetos de classes de controle são, normalmente, usados por outros objetos de controle. Eles são os responsáveis por manipular todos os outros objetos do sistema.

Por Quanto Tempo Existem: Os objetos de classes de controle existem, normalmente, apenas *durante a execução do caso de um uso*, não sendo armazenados de maneira alguma. Eles são criados quando a execução do caso de uso é necessária, sendo destruídos logo que o caso de uso é finalizado. Por esta razão, estes objetos são denominados *transientes*.

1.1.3. Classes de Objetos de Fronteira

Como Identificar: Objetos de fronteira são aqueles que propiciam a comunicação dos atores com o sistema. Em outras palavras, eles compõem a interface de interação entre os atores (humanos ou não) com o sistema a ser desenvolvido. No caso de atores humanos, esta interface é composta pelas telas ou janelas do sistema. Entretanto, na fase de identificação das classes, não se deve especificar o conteúdo destas classes: se a interação será por interface gráfica ou não deverão ficar para momentos futuros.

Tipicamente, existe uma classe de fronteira para cada ator que se comunica com cada caso de uso. Assim, se existem dois casos de uso, o primeiro com 3 atores e o segundo com 2 atores, provavelmente haverá pelo menos 5 classes de objeto de fronteira.

Como Nomear: Os nomes de classes de fronteira não devem conter verbos, devendo ser substantivos que lembrem qual é o canal de comunicação que ele representa, como por exemplo: FormularioCadastro ou LeitoraCodigoBarras.

Estereótipo UML: Na UML, o estereótipo padrão é « boundary ».

Por Quanto Tempo Existem: Os objetos de fronteira são, normalmente, *transientes*. Isto significa que não são armazenados, existindo apenas brevemente na memória do sistema. São criados quando a interação com o usuário é necessária e destruídos em seguida. Assim, estes objetos também são considerados *não-persistentes*.

1.2. Identificação de Responsabilidades

Em um sistema computacional, todos os objetos possuem **responsabilidades**. Estas responsabilidades podem ser relacionadas a alguma *função* do objeto, ou seja, coisas que ele deve fazer, ou a algo que este objeto deve *saber/conhecer*, ou seja, algum atributo que ele possua ou outros objetos que estejam ligados a ele.

Cada categoria de objetos pode ter responsabilidades mais ou menos claras. É possível uma definição genérica por tipo de classe:

1.2.1. Classes de Entidade

A responsabilidade principal de um objeto de uma classe de entidade é armazenar e manter os dados pelos quais são responsáveis, de maneira que quando estes dados forem necessários, eles estarão facilmente disponíveis e sempre serão corretos.

Os objetos de entidade possuem, normalmente, as seguintes responsabilidades:

a) *Conhecer* e *Informar* o valor de seus atributos.
b) *Realizar cálculos* (e impor restrições) relativas às regras de negócio pelas quais são responsáveis (por exemplo, um objeto Pessoa não deve aceitar um CPF inválido).
c) *Criar e destruir* seus objetos-parte.

1.2.2. Classes de Controle

A responsabilidade principal de um objeto de uma classe de controle é executar um processo em uma ordem correta, realizando as comunicações com outros objetos na ordem certa, de forma que um caso de uso seja cumprido.

Os objetos de controle possuem, normalmente, as seguintes responsabilidades:

a) *Coordenar* a execução de um caso de uso, manipulando objetos de entidade e de fronteira, quando necessário.
b) *Monitorar* os objetos de fronteira, para reagir a eventos externos.
c) *Criar associações* entre alguns objetos de entidade.
d) *Manter registro* de alguns valores especiais relativos ao caso de uso.
e) *Manter o estado* atual do caso de uso.

1.2.3. Classes de Fronteira

A responsabiliade principal de um objeto de uma classe de fronteira é interagir com o usuário. Em geral, este objeto é criado por um objeto de controle, que lhe passa um objeto entidade a ser editado; após a edição do objeto de entidade pelo usuário, este objeto é devolvido para o objeto de controle, de forma que ele possa continuar seu processamento.

Objetos de fronteira possuem, normalmente, as seguintes responsabilidades:

a) *Interagir* com o usuário e notificar aos outros objetos do sistema sobre eventos externos ao sistema.
b) *Notificar* aos atores os resultados de interações de objetos do sistema.

2. METODOLOGIA CRC (Classe-Responsabilidade-Colaboração)

Conceitos Chave:

- Colaborador x Colaboração
    * Pedido x ItemDePedido => CustoTotal x Custo
- CRC
    * Cartão => Classe / Responsabilidade / Colaboração
    * Passos => Para cada Caso de Uso
        = Identificar classes
        = Identificar responsabilidades
        = Identificar colaborações
        = Simular!
    * Dicas!
        = Relação Classe/Responsabilidade
        = Inteligência distribuída
        = Agrupamento de responsabilidades conceitualmente relacionadas
        = Evitar redundância de responsabilidades

A metodologia CRC (Classes, Responsabilidades, Colaborações) consiste em identificar possíveis classes e responsabilidades e, posteriormente, colaboradores através de uma espécie de brain storm. Mas antes de apresentar esta metodologia, é necessário apresentar o conceito de *colaboração*.

2.1. Colaborador e Colaboração

O conceito de classes e responsabilidades já foi visto, mas o conceito de *colaborador* é novo. Um objeto colaborador é aquele *colabora para a obtenção de um resultado*. Por exemplo: é comum que um objeto, para executar uma de suas responsabilidades, precise de dados que serão fornecidos por outros objetos. Neste caso, estes outros objetos serão *colaboradores* do primeiro objeto, na execução daquele cálculo.

Considere que um objeto da classe Pedido tenha a responsabilidade de *conhecer* todos os objetos da classe ItemDePedido que fazem parte dele. Adicionalmente, este objeto da classe Pedido tem a responsabilidade de calcular o valor total do pedido, enquanto cada objeto da classe ItemDePedido tem a resposabilidade de saber seu valor. Resumidamente:

Classe: Pedido
Responsabilidade 1: Conhecer os objetos do tipo ItemDePedido que fazem parte dele.
Responsabilidade 2: Calcular e informar seu valor total.

Classe: ItemDePedido
Responsabilidade 1: Conhecer seu próprio valor.
Responsabilidade 2: Calcular e informar seu valor total.

Como é possível ver, para que um objeto da classe Pedido calcule seu valor total, ele precisará da *colaboração* dos objetos da classe ItemDePedido que fazem parte dele.

Compreendido este conceito, já é possível tratar da metodologia CRC.

2.2. A Metodologia CRC e Seus Passos

Na metodologia CRC clássica, várias pessoas da equipe de projeto se juntam e fazem um *brainstorm* sobre as possíveis classes do sistema, sejam de entidade, controle ou fronteira. Para cada classe identificada é criado um cartão, como o apresentado na figura 1.

Nome da Classe	
Responsabilidade 1	Colaborador 1
Responsabilidade 2	Colaborador 2
Responsabilidade 3	Colaborador 3
Responsabilidade 4	Colaborador 4
Responsabilidade 5	Colaborador 5
Responsabilidade 6	Colaborador 6
Responsabilidade 7	Colaborador 7

Figura 1: Modelo de cartão CRC

A seguir é feito um *brainstorm* para identificar as responsabilidades básicas de cada classe, sendo válidas as dicas passadas anteriormente para a realização deste processo. Identificadas as responsabilidades, deve-se identificar quais são os colaboradores necessários para a correta execução de cada uma das responsabilidades.

No contexto atual, a metodologia CRC será aplicada para cada caso de uso separadamente, já considerando as classes de entidade e controle identificadas anteriormente. É importante lembrar que, por razões óbvias, classes de entidade que já apareceram em um caso de uso podem voltar a aparecer em outro.

Os passos deste método, já adaptados, são:

I. Forme grupos de cerca de 6 pessoas.
II. Selecione um Caso de Uso para a seção CRC.
II.1. Distribuir igualmente cartões como os da figura 1 para cada um dos membros do grupo.
II.2. O grupo deve identificar as classes envolvidas no Caso de Uso (pondendo ser aquelas já identificadas anteriormente). Cada membro irá escolher sua classe e anotar seu nome no topo do cartão.
II.3. Cada membro irá analisar o caso de uso e identificar quais as responsabilidades que acredita ser de sua classe, na coluna do lado esquerdo.

II.4. Os membros devem debater se as responsabilidades de sua classe estão corretas e são suficientes. Pode ser necessário criar novas responsabilidades, mover responsabilidades de uma classe para outra ou até mesmo criar novas classes.

II.5. Cada membro analisa novamente o caso de uso e as responsabilidades já definidas e preenche os claboradores de cada responsabilidade de cada classe, para aquele caso de uso.

II.6. Os membros devem debater se as colaborações de sua classe estão corretas.

III. Com os cartões preenchidos para aquele caso de uso, um dos atores agirá como a classe de controle daquele caso de uso, e iniciará a simulação do caso de uso. Cada membro do grupo deverá agir como uma das classes, cumprindo com suas responsabilidades atribuídas. O objetivo é verificar se todas as responsabilidades e colaborações necessárias para o caso de uso foram encontradas.

IV. Volte para II até que todos os Casos de Uso tenham sido "processados".

2.3. Dicas para Atribuição de Responsabilidades e Colaborações

As sessões CRC são bastante dinâmicas e muitas coisas podem mudar de uma sessão para outra, ainda que o caso de uso em debate seja o mesmo. Além das dicas já apresentadas anteriormente, mais algumas serão apresentadas, de modo a evitar que os grupos "andem em círculos":

a) Associe responsabilidades com base nas especialidades da classe. Uma pizza deve saber quais são seus ingredientes, mas não faz sentido a pizza informar o valor total da fatura, por exemplo.

b) Distribua a inteligência do sistema. É inadequado (e incomum) que algumas classes tenham dezenas de responsabilidades e outras nenhuma. Quando ocorrer de algumas classes possuírem muitas responsabilidades, tente dividí-las. Por exemplo: um Pedido sabe seu valor total, um ItemDePedido sabe seu subtotal e um Produto sabe seu valor. O subtotal do ItemDePedido é calculado solicitando os valores dos Produtos que dele fazem parte e o total do Pedido é calculado solicitando os subtotais dos ItensDePedido que dele fazem parte.

c) Agrupe responsabilidades conceitualmente relacionadas. Responsabilidades que são conceitualmente relacionadas devem ser mantidas em uma mesma classe. Assim, as informações que a classe Cliente deve conhecer devem estar na classe Cliente. Por outro lado, às vezes criamos classes diferentes com o mesmo possível conteúdo, como classes Pedido e Fatura, já que elas representam *conceitos distintos*.

d) Evite responsabilidades redundantes. Se mais de uma classe possui a mesma responsabilidade, avalie se não faz mais sentido eliminar a responsabilidade de uma delas e, quando esta precisar da informação, deverá solicitar à outra. Em alguns casos até mesmo valerá a pena criar uma nova classe para esta responsabilidade.

2.4. Exercício Exemplo

Use a metodologia CRC para definir as classes envolvidas no caso de uso "CadastrarPizza", do gerente, para o sistema de pizzaria.

Dica: Analise os passos do Caso de Uso.

3. BIBLIOGRAFIA

BEZERRA, E. **Princípios de Análise e Projeto de Sistemas com UML**. 2ed. Rio de Janeiro: Editora Campus Elsevier, 2007.

DEITEL, H.M; DEITEL, P.J. **Java**: como programar - 6ed. São Paulo: Pearson-Prentice Hall, 2005.

Unidade 2: Transição da Análise ao Projeto

Prof. Daniel Caetano

Objetivo: Apresentar as principais etapas do projeto de um Sistema Orientado a Objetos, correlacionando-as com as etapas já desenvolvidas na análise.

Bibliografia: BEZERRA; DEITEL e DEITEL.

INTRODUÇÃO

Conceitos Chave:

- Problema: Diagrama de Classes de Análise => ... ?
- UML => Notação única do início ao fim
- Análise e Projeto OO x Análise e Projeto Estruturada
    * Quando acaba análise e quando começa projeto?

Com a análise pronta, surge o problema: como transformar isso em um projeto? Graças às premissas de sua criação, quando a UML é usada para modelar e projetar Sistemas Orientados a Objetos, a mesma representação é utilizada nas fases de análise e projeto, facilitando muito o trabalho; assim, basta usar como base o diagrama de classes desenvolvido anteriormente e acrescentar detalhes ao mesmo, além de complementá-lo com alguns outros diagramas.

Ao comparar esta característica à da modelagem e projeto estruturado, onde existe uma diversidade de diagramas e notações, fica aparente o aspecto benéfico da unificação da notação, trazendo uma maior uniformidade no desenvolvimento.

Entretanto, existe um ponto ligeiramente negativo: é difícil dizer exatamente quando a fase de análise acaba (e qual o seu produto) e quando é que começa a fase de projeto; na verdade, há um grande número de atividades que ocorrem em um período de transição, uma *zona cinza*. Neste período de transição é comum que parte da equipe esteja ainda finalizando algumas atividades de análise enquanto outras já estejam engajadas em tarefas do desenvolvimento do projeto.

1. FUNÇÃO DA FASE DE PROJETO

Conceitos Chave:

- Análise => Identificação de Classes
- Projeto => Refinamento das classes e suas interações
    * Descrição das interações
- Embasar a implementação do software

Enquanto a fase de análise está preocupada principalmente com a identificação de classes de um Sistema Orientado a Objetos, a fase de projeto está mais preocupada com o refinamento destas classes e suas interações. A principal diferença nos diagramas fica, então, por conta da quantidade de detalhes existentes.

Com o uso da técnica CRC na fase de análise foi dado início à atividade de identificação das interações entre objetos. Este modelo será refinado na fase de projeto, produzindo diagramas de interações para os casos de uso mais relevantes.

Assim, apesar dos modelos construídos na fase de análise auxiliarem no esclarecimento acerca do problema a ser resolvido, eles não são suficientes para embasar a implementação do software.

No projeto serão definidos, então, os aspectos referentes à solução a ser implementada, visando encontrar alternativas para que o sistema cumpra as funções estabelecidas pelos requisitos funcionais, ao mesmo tempo em que atendam às restrições definidas pelos requisitos não-funcionais.

2. ETAPAS DA FASE DE PROJETO

Conceitos Chave:

- Várias Fases
- Detalhamento Dinâmico
    * Verificação das colaborações
    * Modelagem Dinâmica
        = Modelos de interações
        = Modelos de estados (*)
        = Modelos de atividades
- Refinamento Estrutural
    * Refinamento de Classes => Atributos e Métodos
    * União / Divisão de Classes
    * Similaridades => Especialização / Generalização / Abstração / Interfaces

- Projeto de Arquitetura
    * Decomposição em subsistemas => Módulos ou Camadas Lógicas
        = Sistemas Distribuídos?
    * Arquitetura => Distribuição Física e Lógica do sistema + Comunicação
        = Comunicação = meio físico, interfaces, protocolos
- Persistência de Objetos
    * Objetos Transientes x Objetos Persistentes
    * Controle de Transações
        = Quando objetos persistentes serão armazenados
        = Quando objetos persistentes são recuperados
        = Quando objetos persistentes são removidos
- Projeto de Interface Gráfica
    * Determina sucesso ou fracasso
    * Objetos de Fronteira
        = Aparência + Alta Usabilidade
    * Cores, mensagens, dimensões de controles, figuras...
- Projeto de Algoritmos
    * Seleção de algoritmos (descritos em linguagem formal ou não)
    * Diagramas de atividades => ordenação, otimização, simulação etc.
    * Critérios de escolha (do mais importante ao menos importante)
        = Adequação à Função
        = Complexidade Computacional
        = Flexibilidade
        = Facilidade de Compreensão

Para que os modelos da etapa de análise sejam modificados até um nível de detalhamento suficiente para a implementação várias atividades (fases não necessariamente seqüenciais) precisam ser cumpridas (BEZERRA, 2007):

1) Detalhamento dos aspectos dinâmicos.
2) Refinamento dos aspectos estáticos e estruturais.
3) Projeto da arquitetura.
4) Persistência de objetos.
5) Projeto de interface gráfica com o usuário.
6) Projeto de algoritmos.

É importante ressaltar que o início de algumas destas atividades correspondem ainda à etapa de análise, sendo seu desenvolvimento propriamente dito parte da etapa de projeto.

2.1. Detalhamento dos Aspectos Dinâmicos

Como uma ferramenta de auxílio para a identificação das classes de análise foi apresentada a técnica CRC. Na aplicação da técnica CRC são também identificadas as colaborações entre objetos.

Na verificação das colaborações para atender às responsabilidades das classes são identificadas algumas interações entre objetos destas classes. Na fase de projeto estas interações serão detalhadas na forma dos *modelos de interações*, *modelos de estados* (não abordados por este curso) e *modelos de atividades*. Em conjunto, estes modelos definem a *modelagem dinâmica* do sistema.

2.2. Refinamento dos Aspectos Estáticos e Estruturais

No modelo de classes de análise foi definida uma estrutura em um alto nível de abstração, suficiente para a descrição do problema a ser resolvido pelo sistema. Entretanto, para que este modelo chegue a um nível descritivo suficiente para sua elaboração é necessário refinar este modelo. Neste refinamento, muitos detalhes são adicionados e, muitas vezes, uma classe do modelo da análise se torna várias classes no modelo do projeto. O inverso também pode ocorrer, embora seja mais raro: várias classes de análise se tornarem uma única classe de projeto.

Nesta etapa do projeto devem ser descritos, de forma detalhada, os atributos e as operações de cada classe, necessários para o cumprimento das responsabilidades de cada classe. As interações, definidas no modelo dinâmico, muitas vezes auxiliam na identificação de novas responsabilidades das classes, que precisam ser igualmente representadas na forma de atributos e métodos.

A identificação de similaridades é também uma atividade desta etapa, que deve identificar classes candidatas para relações de especialização e generalização, o uso de classes abstratas e interfaces etc.

2.3. Projeto de Arquitetura

Um aspecto importante a ser considerado na modelagem de um Sistema Orientado a Objetos e a sua decomposição em subsistemas, formando módulos ou camadas lógicas. Isso é de importância ainda maior quando são tratados sistemas distribuídos, onde os componentes do sistema estão distribuídos em equipamentos fisicamente distintos.

É dado o nome de *arquitetura* à forma com o sistema é distribuído, física e logicamente. Também faz parte desta etapa a definição dos meios pelos quais os objetos se comunicam através das camadas ou módulos, definindo as interfaces que permitem a reutilização destes blocos em aplicações futuras e que serão utilizados no momento da implementação.

2.4. Persistência de Objetos

Com já vimos, um sistema possui objetos *transientes* e objetos *persistentes*. Os *transientes* existem apenas durante o tempo de uma sessão, enquanto os *persistentes* existem por um tempo além de uma transação, muitas vezes existindo por todo o tempo de vida da aplicação.

Um projeto de um Sistema Orientado a Objetos deve cuidar de várias tarefas relacionadas à persistência de objetos (BEZERRA, 2007):

1) Como as transações são controladas.
2) Quando e como objetos persistentes devem ser enviados para o mecanismo de persistência.
3) Quando e como objetos persistentes devem ser lidos do mecanismo de persistência.
4) Quando e como os objetos persistentes são removidos.

2.5. Projeto de Interface Gráfica com o Usuário

Embora muitas vezes tenha um papel secundário no desenvolvimento do projeto, este é de fundamental importância no sentido que ele pode determinar o sucesso ou total fracasso de uma aplicação.

Normalmente são estes elementos que realizam a função de objetos de fronteira entre o sistema e um ator humano, nos casos de uso em que estes interagem. Neste projeto deve ser definida a aparência do sistema para seus usuários, com alta usabilidade e facilidade de operação. Devem ser definidas padronizações de cores, de mensagens de erro, de dimensões de controles, de tipos de figuras, de formatação de entradas de dados etc.

2.6. Projeto de Algoritmos

Nesta etapa devem ser definidos os algoritmos que devem ser usados em cada parte específica do sistema, podendo ser definidos em linguagem formal ou não formal. É comum o uso dos diagramas de atividades para especificar tais algoritmos com a UML. Alguns algoritmos importantes são os de ordenação, otimização, simulação etc.

Os critérios que norteiam a escolha de um algoritmo são (da mais importante para a menos importante):

a) Adequação à função, ou seja, se ele atende às necessidades do sistema.
b) Complexidade computacional, ou seja, se ele é rápido o suficiente para as necessidades do sistema.
c) Flexibilidade, ou seja, se ele pode ser usado em uma ampla gama de situações.
d) Facilidade de compreensão, ou seja, se o entendimento do mesmo por parte dos desenvolvedores não será tão complexo a ponto de prejudicar o desenvolvimento.

3. BIBLIOGRAFIA

BEZERRA, E. **Princípios de Análise e Projeto de Sistemas com UML**. 2ed. Rio de Janeiro: Editora Campus Elsevier, 2007.

DEITEL, H.M; DEITEL, P.J. **Java**: como programar - 6ed. São Paulo: Pearson-Prentice Hall, 2005.

Unidade 3: Modelagem de Interações

Prof. Daniel Caetano

Objetivo: Apresentar os principais conceitos envolvidos dos aspectos dinâmicos de um sistema orientado a objetos.

Bibliografia: BEZERRA; DEITEL e DEITEL.

INTRODUÇÃO

Conceitos Chave:

- Modelos Estruturais x Modelos Dinâmicos
- Responsabilidades x Colaborações
- Foco: Modelagem Dinâmica

Nas aulas anteriores foram apresentados os conceitos que permitiram definir uma boa parte da parte estática de um problema, ou seja: o que ele deve fazer (MCU) e com o que ele deve executar estas tarefas (Modelo de Classes de Análise).

Foi iniciada também uma discussão sobre a identificação de responsabilidades e colaborações, realizando as primeiras aproximações de quais deveriam ser as interações entre os objetos de diversas classes para que um caso de uso (e seus cenários) possam ocorrer com sucesso; a modelagem de interações é, exatamente, a formalização da representação destas interações.

Segundo Jacobson (1992 apud BEZERRA, 2007). esta formalização é importante porque apenas depois dela é que se tem certeza de que todas as responsabilidades dos objetos foram identificadas. A esta formalização das interações denominamos de **modelagem dinâmica** do sistema, que também envolve a modelagem de estados (que não será vista neste curso).

1. DIAGRAMAS DE SEQÜÊNCIA E COMUNICAÇÃO

Conceitos Chave:

- Realização de Caso de Uso: O que é?
  * Diagramas de Seqüência => Ordem das interações
  * Diagramas de Comunicação => Elementos que interagem
  * Intercambiáveis

Dá-se o nome de "**realização de um caso**" às interações necessárias entre objetos para que a funcionalidade de um caso de uso se desenvolva, descrevendo o comportamento do sistema a partir de um ponto de vista interno.

Estes diagramas são conhecidos como "Diagramas de Interação", sendo compostos basicamente pelos **diagramas de seqüência** e/ou **diagramas de comunicação** (ou, na nomenclatura da UML 1.x, diagramas de colaboração). Estes dois tipos de diagramas representam as mesmas informações e são, portanto, intercambiáveis. Entretanto, a ênfase de representação de cada um deles é distinta: no **diagrama de seqüência** há uma preocupação maior com a explicitação da **ordem** com que os objetos interagem no tempo e no **diagrama de comunicação** esta preocupação é em explicitar **os relacionamentos** entre os objetos.

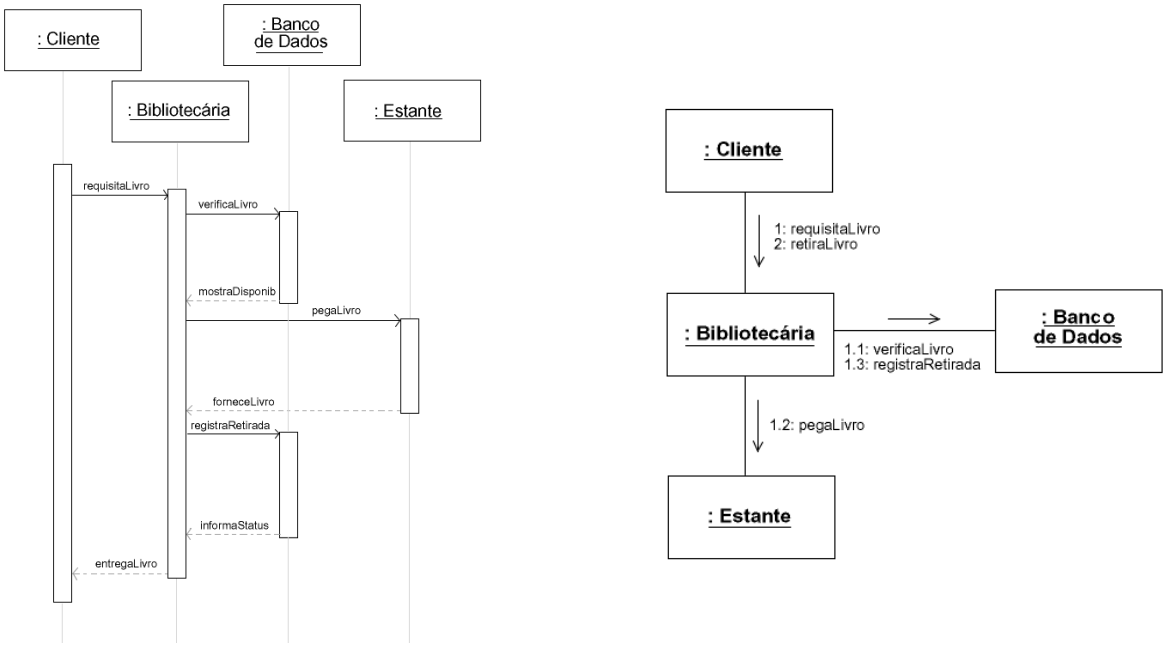


Figura 1: Exemplo de um diagrama de seqüência (esq) e comunicação (dir)

Como pode ser verificado pelos exemplos, os diagramas de seqüência e comunicação possuem muitas estruturas em comum. Entretanto, por alguma razão parece existir uma preferência pelos diagramas de seqüência (apesar de serem intercambiáveis). Por questões de tempo e praticidade, apenas os diagramas de seqüência serão apresentados; os diagramas de comunicação são extensivamente explicados na bibliografia.

2. UML PARA DIAGRAMAS DE INTERAÇÃO

Conceitos Chave:

- Elemento Fundamental: *Mensagem*
- Mensagem => requisição de um ente para outro ente
    * atores, classes, objetos, coleções de objetos
- A envia mensagem para B => A invoca método de B
    * Tipos de Mensagens:
        = Síncrona x Assíncrona
        = Mensagem de Sinal
        = Mensagem de Retorno
        = Mensagem Reflexiva
    * Formato: [[seqüência] controle:] [v :=] nome ([argumentos])
        a) Mensagem simples: **adicionarItem(item)**
        b) Mensagem com condição: **[a > b]: trocar(a, b)**
- Atores => normalmente causam o início da seqüência.
- Classes => métodos estáticos
- Objetos => mais comuns
    * Formato: [nome_do_objeto[seletor]] : nome_da_classe
- Coleções de objetos

Nos diagramas de casos de uso, os elemento fundamentais eram os próprios casos de uso. Nos diagramas de classes, os elementos básicos eram as próprias classes. Nos diagramas de interação, os elementos fundamentais são as **mensagens**.

Entretanto, as mensagens não existem sozinhas; uma mensagem é sempre trocada entre pares de entidades. Estas entidades podem ser **atores**, **classes**, **objetos** e **coleções de objetos**. Vejamos qual é a representação que a UML define para estas entidades.

2.1. Mensagens em Diagramas de Interação

As mensagens, como dito, são o elemento fundamental de um diagrama de interações. As mensagens representam a comunicação entre duas entidades e são representadas por setas direcionadas, apontando da origem para o destino da comunicação.

As mensagens representam, normalmente, requisições que uma entidade faz para outra entidade. Em termos de programação, as mensagens correspondem, normalmente, às chamadas de métodos. Em outras palavras, dizer que um objeto envia uma mensagem para outro objeto significa que um objeto invoca um método de outro objeto.

Toda mensagem terá um ente remetente e um ente receptor e, adicionalmente, a mensagem deve conter todos os elementos necessários para que o receptor possa executar a ação necessária.

Uma mensagem pode ser de diferentes naturezas, segundo a UML 2:

- **Mensagem síncrona**: é a mensagem usual, em que o remetente aguarda uma resposta do receptor antes de continuar seu "processamento".
- **Mensagem assíncrona**: é uma mensagem em que o remetente **não** aguarda a resposta para continuar seu processamento.
- **Mensagem de sinal**: é uma mensagem usada, normalmente, para enviar uma requisição entre módulos de um sistema distribuído (cliente-servidor, por exemplo).
- **Mensagem de retorno**: é uma mensagem que indica o término de uma mensagem enviada anteriormente.
- **Mensagens reflexivas**, que nada mais são do que mensagens que um ente envia para si mesmo.

A sintaxe de uma mensagem é (elementos entre colchetes são opcionais):

[[seqüência] controle:] [v :=] nome ([argumentos])

A *seqüência* é um número que indica explicitamente a ordem que as mensagens foram enviadas (1, 2, 3...) É bastante relevante nos diagramas de comunicação, onde são a única forma de indicar a seqüência de ações. Nos diagramas de seqüência quase não são usados. Podem ser usados níveis, indicando que uma mensagem ocorreu em conseqüência de uma mensagem anterior: "1.1" e "1.2" são mensagens seqüenciais que foram disparadas por uma mensagem anterior "1".

O *controle* indica uma de duas coisas: 1) alguma cláusula que deve ser verdadeira para que a mensagem seja enviada (ex.: **[a>b]**); ou 2) uma indicação de um processo de repetição (ex.: ***[i := 1..10]**).

O *v* é o nome de uma variável que poderá receber o retorno da mensagem.

O *nome*, que é a única partícula obrigatória de uma mensagem (com os respectivos parênteses), indica o nome da mensagem - que normalmente é o nome do método chamado.

Finalmente, os *argumentos* são os parâmetros que precisam ser enviados juntamente com a mensagem para que o ente receptor possa cumprir a solicitação.

Exemplos de mensagens:

a) Mensagem simples:	**adicionarItem(item)**
b) Mensagem com condição:	**[a > b]: trocar(a, b)**
c) Mensagem valor de retorno:	**x := selecionar(e)**
d) Mensagem com iterações:	***[i =: 1..10] figuras[i].desenhar()**
e) Mensagem com número de seqüência:	**1.1.2: calcular**()

2.2. Atores

Como os diagramas de interação normalmente estão ligados a um caso de uso, os atores que participam de um caso de uso também podem ser representados nos diagramas de interação. Por exemplo: um diagrama de interação pode começar com a ação inicial de seu ator primário. A notação utilizada pela UML para representar os atores nos diagramas de interação é a mesma usada no diagrama de casos de uso.

2.3. Classes

Embora sejam os objetos que normalmente interagem, em alguns casos a interação pode ser feita com uma classe. Este é um conceito que está bastante ligado à programação orientada a objetos (métodos estáticos, também conhecidos como métodos de classe). São representados da forma mais simples, como retângulos com o nome da classe dentro.

2.4. Objetos

Os objetos são os entes mais comuns nos diagramas de interação. Sua representação é simples: é usado um retângulo similar ao de uma classe, mas a inscrição do nome é ligeiramente diferente e tem a seguinte forma (simplificada):

[nome_do_objeto[seletor]] : nome_da_classe

Sendo tudo isso "sublinhado" para indicar que é, de fato, um objeto.

O *nome_do_objeto* é, certamente, o nome de um objeto específico (instância). Normalmente este é um elemento opcional, já que uma interação descrita não se limita àquele objeto específico, mas a qualquer objeto aquela classe. Quando o nome do objeto é omitido, a representação fica sendo apenas:

: nome_da_classe

Como pôde ser observado anteriormente, na figura 1.

O *seletor* é um elemento opcional para indicar se estamos falando de um elemento que está armazenado em uma lista (ou matriz) de objetos.

Finalmente, o *nome_da_classe* indica o nome da classe que define o tipo do objeto, sendo este o único elemento obrigatório da especificação de objeto, lembrando-se de adicionar um sinal de dois pontos (:) e de sublinhá-lo (_) para evidenciar que se está falando de um objeto daquele tipo (e não da classe em si).

2.5. Coleções de Objetos

As coleções de objetos não são tão incomuns em diagramas de interação. Muitas vezes temos objetos que são, na verdade, coleções de objetos de um mesmo tipo. Essas coleções são capazes de realizar operações diversas, como adicionar um item, subtrair um item, retornar próximo objeto da coleção etc. Mas, é importante ressaltar, só se fala em coleções de objetos **de um mesmo tipo**.

É claro que, em linguagens como Java, sempre podemos criar uma coleção de objetos do tipo "Object" e, com isso, passa a ser possível ter qualquer tipo de objeto lá dentro; entretanto, a linguagem tratará todos os objetos lá existentes como sendo da classe "Object" e não de qualquer outra.

A UML especifica a representação de objetos com a notação de um seletor genérico no nome do objeto (por exemplo: *[i]*) e também com dois retângulos sobrepostos, como pode ser visto na figura 2:

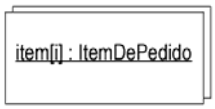


Figura 2: Representação de uma coleção de objetos

3. DIAGRAMA DE SEQÜÊNCIA DA UML

Conceitos Chave:

- Linhas de Vida
  * Ordem vertical x Ordem horizontal
- Mensagens
  * Mensagem síncrona
  * Mensagem assíncrona
  * Mensagem de retorno
  * Mensagem de criação
  * Mensagem reflexiva
- Ocorrência de Execução
- Criação x Destruição de objetos

Os elementos apresentados até agora são elementos genéricos dos diagramas de interação; servem tanto para os diagramas de seqüência quanto para os diagramas de comunicação. Nesta seção serão apresentadas algumas representações exclusivas dos diagramas de seqüência, que são: *linhas de vida*, *mensagens*, *ocorrências de execução* e *criação e destruição* de objetos.

3.1. Linhas de Vida

As linhas de vida são os elementos mais importantes da representação dos diagramas de seqüência. As linhas de vida indicam o período no tempo em que um ente existe no sistema. A representação é feita com um ente (ator, objeto etc.) no topo e uma linha vertical tracejada que se extenue para baixo, a partir da representação do ente. Um exemplo pode ser visto na figura 3 (BEZERRA, 2007).

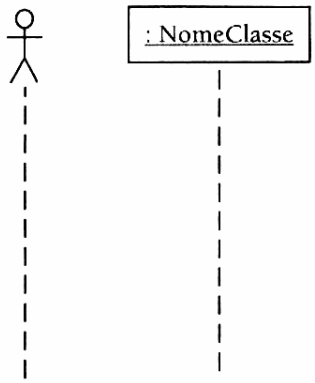


Figura 3: Representação da linha de vida

A ordem horizontal em que os entes aparecem num diagrama de seqüência não tem um significado pré-estabelecido. Entretanto, é comum a indicação na seguinte ordem: ator primário, objetos de fronteira, objetos de controle, objetos de entidade e, finalmente, atores secundários.

3.2. Mensagens

As mensagens são representadas por flechas de diferentes tipos, geralmente ligando uma linha de vida à outra, partindo do objeto que envia a mensagem e apontando para o objeto que recebe a mensagem. O rótulo da mensagem (visto na seção 2) deve ser representado acima da linha da mensagem. As notações de mensagem em diagramas de seqüência da UML podem ser vistas na figura 4 (BEZERRA, 2007):

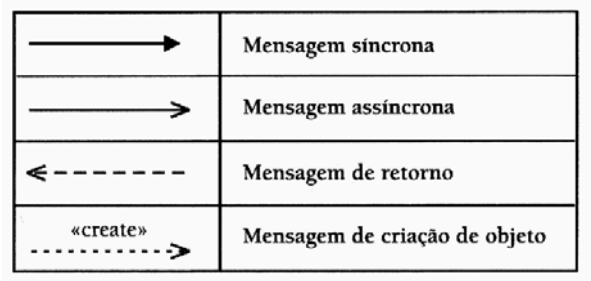


Figura 4: Notações de mensagens para diagramas de seqüência

É importante observar que, num diagrama de seqüência, a passagem do tempo é indicada na vertical, de cima para baixo. Isso significa que uma mensagem representada por uma seta numa posição mais baixa terá sido enviada *depois* de uma mensagem representada por uma seta mais no topo do diagrama (observe, na figura 5, que a mensagem "verificaLivro" só ocorre após a mensagem "requisitaLivro").

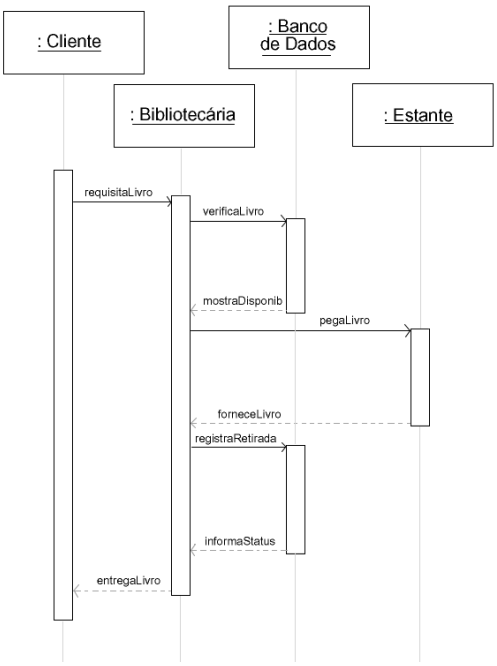


Figura 5: ordem temporal das mensagens

Adicionalmente, as mensagens reflexivas são representadas como uma seta que se inicia em uma linha de vida e acaba nela própria, como pode ser visto na figura 6 (BEZERRA, 2007).

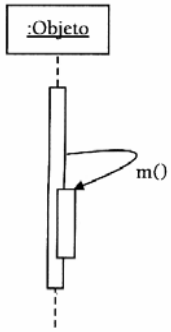


Figura 6: Mensagem reflexiva

As linhas de vida são os elementos mais importantes da representação dos diagramas

3.3. Ocorrências de Execução

A ocorrência de execução indica o período em que um objeto está realizando alguma operação, sendo representada na linha de vida como um bloco retangular.

O topo do retângulo que indica a ocorrência de execução deve ficar no mesmo nível que a seta da mensagem que causou a execução e a base do retângulo deve coincidir com o término desta ação. O uso de mensagens de retorno "desobriga" da representação das ocorrências de execução; entretanto, é bastante interessante sua representação mesmo nestes casos.

3.4. Criação e Destruição de Objetos

Criação e destruição de objetos são ações comuns em programas orientados a objetos. Assim, a UML oferece recursos para representar tais ações nos diagramas de seqüência. Isso pode implicar em uma mudança na aparência do diagrama, também.

Primeiramente, digamos que quando um objeto já existe desde o início do diagrama de seqüência, sua representação deve ser feita no topo do diagrama; por outro lado, quando um objeto é criado *durante* o diagrama de seqüência, a representação do objeto deve ser indicada no momento (posição vertical) em que este objeto é criado. A seta de criação é a de uma mensagem assíncrona que deve ligar a linha de vida do objeto criador com o retângulo que representa o objeto criado. O estereótipo << create >> deve ser usado, como pode ser visto na figura 7 (BEZERRA, 2007).

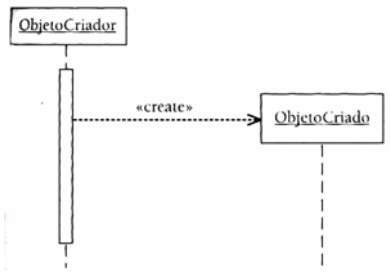


Figura 7: Representação da criação de objetos

A destruição de objetos é mais simples: é uma mensagem síncrona, sendo que a seta que a representa liga a linha de vida do objeto destruidor com a linha de vida do objeto destruído. Um pequeno X deve ser representado na linha de vida do objeto destruído abaixo do ponto em que ele recebe a mensagem de destruição (e a linha de vida dele não deve se estender além deste X). A mensagem deve ser indicada com o estereótipo << destroy >>, como pode ser visto na figura 8.

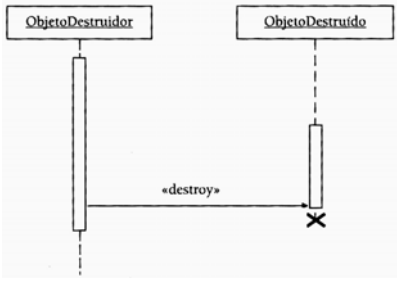


Figura 8: Representação da destruição de objetos

4. CONSTRUÇÃO DO MODELO DE INTERAÇÕES

Conceitos Chave:

- Responsabilidade x Mensagem => Métodos
- Coesão => melhorar reúso
- Acoplamento => redução de dependências
- Processo: similar ao CRC
    1. Selecionar cenário relevante do caso de uso
    2. Identificar eventos relevantes
        2.1. Indicar atores, objetos de fronteira e controle relevantes.
        2.2. Indicar as mensagens, suas condições e iterações
        2.3. Adicione objetos de entidade à medida do necessário
- Dicas
    * Antes de iniciar, separar as classes que fazem parte do caso de uso.
    * Indicar primeiro os objetos de classes que organizem o caso de uso.
    * Verificar a coerência entre todos os diagramas e cartões CRC
    * Objetos <<control>> ficam muito acoplados.
    * Objetos <<control>> só devem controlar o caso de uso.
    * Use notas explicativas sempre que necessário
    * Minimize o acoplamento. Só envie mensagens de um objeto para:
        a) outro objeto da mesma classe que ele
        b) objetos recebidos como parâmetro em métodos
        c) objetos que são atributos da classe (ou de uma coleção que o seja)
        d) objetos criados por ele

Assim como nos modelos de casos de uso e de classes, simplesmente conhecer a notação dos diagramas de seqüência não é suficiente para tornar "automática" a compreensão de como gerar tais diagramas.

Primeiramente é importante ressaltar a relação entre *resposabilidades* e *mensagens*. A idéia é que sempre que um objeto precisar de "ajuda" para cumprir uma de suas responsabilidades, ele deve enviar uma mensagem a um outro objeto, seu colaborador, sendo este o que tem responsabilidade de realizar a ação solicitada. Isso também significa que quando uma mensagem é enviada a um objeto, ele deve possuir uma operação associada, o que permite identificar os **métodos** que cada classe deve possuir. Por exemplo, se um objeto envia uma mensagem do tipo "validarSenha(id, senha)" para um objeto do tipo *Usuario*, então o objeto do tipo Usuario deve conter um método que se chame "validarSenha", receba os parâmetros id e senha corretamente e, finalmente, deve responder um valor *falso* ou *verdadeiro*, informando o resultado da validação.

Ao elaborar os diagramas de seqüência, o projetista deve se perguntar constantemente se um objeto deve ser capaz de responder àquela mensagem sozinho ou se ele precisa da ajuda de outros colaboradores para respondê-la, subdividindo o trabalho (e distribuindo a inteligência do sistema).

Um segundo conceito importante é o da *coesão e acoplamento*, que é ligado ao conceito anteriormente exposto. Pelo dito anteriormente, a definição do diagrama de seqüência pode ir de dois extremos: se tivermos um sistema com R responsabilidades, pode-se decidir que haverá uma única classe que cumpre todas as R responsabilidades até o outro extremo, que será definir R classes, cada uma com apenas uma responsabilidade. Mas... entre estas possibilidades extremas, há diversas outras. Qual é a melhor? Não há uma "forma fechada" de responder a esta questão. Como um guia, então, apresentamos os conceitos de coesão e acoplamento.

A *coesão* é uma medida de quão fortemente relacionadas e focalizadas são as responsabilidades de uma classe. Como já dito em aulas anteriores, os métodos de uma classe devem possuir um alto grau de relação entre si. Classes altamente coesas capturam melhor uma abstração e são mais facilmente reutilizadas.

O *acoplamento* é uma medida de quão fortemente uma classe está correlacionada tem conhecimento ou depende de outras classes. Uma classe pouco acoplada depende menos de outras e, por conseqüência é mais simples e reutilizável. Uma classe com forte acoplamento freqüentemente demanda mudanças quando uma das classes da qual depende sofreu mudanças.

Assim, o projetista deve buscar um modelo com alta coesão e baixo acoplamento; No diagrama de seqüência, isto significa o menor número possível de mensagens. Deve-se buscar este caminho, desde que isso não signifique "entulhar" uma classe específica com métodos que não deveriam fazer parte dela, o que significaria uma diminuição da coesão.

4.1. Dicas de Construção do Modelo

O modelo de interações é constituído, basicamente, de um diagrama de seqüência para cada caso de uso (mais um para cada cenário possível). Assim, cada passo deve ser examinado para cada um dos casos de uso e seus cenários. Alguns dos passos que serão apresentados a seguir já podem ter sido executados em etapas anteriores. Antes disso, entretanto, convém apresentar algumas dicas:

A) Identifique as classes e atores que fazem parte de um caso de uso, se ele fosse executado manualmente (como deve ter sido feito na seção CRC). Classes de fronteira também "entram no jogo", neste momento.

B) Identifique no modelo de classes quais delas que possam ajudar a organizar as tarefas a serem executadas. Em geral se tem pelo menos um diagrama de seqüência para cada classe de controle.

C) Separe os objetos de entidade que precisam ser criados e manipulados na execução deste caso de uso.

D) Verifique a consistência e coerência entre o diagrama de casos de uso, o diagrama de classes de análise e as classes selecionadas para a construção do diagrama de seqüência atual.

E) O objeto da classe de controle de um caso de uso, em geral, acaba ficando bastante acoplado aos outros objetos; para coordenar, ele precisa conhecer tais objetos. Por esta razão, é importante definir este objeto como realmente apenas um coordenador, sendo esta sua responsabilidade básica e, dentro do possível, única.

F) Durante a modelagem, tente fazer diagramas inteligíveis e use notas explicativas onde necessário.

G) Evite aumentar demais o acoplamento; procure só enviar mensagens de um objeto para os seguintes: 1) outros objetos da mesma classe que ele; 2) objetos recebidos como parâmetro em um método; 3) objetos que são atributo da classe; 4) objetos criados dentro de um de seus métodos; 5) objetos de uma coleção que é atributo da classe.

4.2. Construindo o Modelo

Agora, depois das dicas, segue um procedimento básico que, em geral, auxilia bastante na elaboração dos diagramas.

Basicamente, o processo é o mesmo da "simulação" dos casos de uso com as classes, já realizado nas seções CRC. Mas é importante ressaltar que, embora não houvesse grande preocupação com a origem de um evento naquela simulação, agora será importante detectar quem é o responsável pela ocorrência dos eventos descritos.

Assim, um diagrama de seqüência sempre começará com a ocorrência de um *evento*, que pode ser a ação de um ator (sobre um objeto de fronteira, por exemplo), a ação de um outro sistema (que também pode ser representado por um ator) etc. Agora, os passos para a elaboração do diagrama.

1. Para cada caso de uso, selecionar os cenários relevantes.
2. Para cada cenário, identifique os *eventos* relevantes e:
    - 2.1. Posicione o(s) ator(es), objeto de fronteira, objeto de controle no diagrama.
    - 2.2. Para cada passo do cenário, defina as mensagens que devem ser trocadas por estes objetos.
    - 2.3. Defina as condições ou iterações necessárias para as mensagens (se for o caso).
    - 2.4. Adicione os objetos de entidade ou as coleções de objetos à medida em que forem necessários para a realização do cenário.

5. ATIVIDADE

Crie o diagrama de seqüência para o cenário principal do caso de uso "CadastrarPizza", cujas classes foram descritas nas aulas anteriores.

6. BIBLIOGRAFIA

BEZERRA, E. **Princípios de Análise e Projeto de Sistemas com UML**. 2ed. Rio de Janeiro: Editora Campus Elsevier, 2007.

DEITEL, H.M; DEITEL, P.J. **Java**: como programar - 6ed. São Paulo: Pearson-Prentice Hall, 2005.

Unidade 4: Modelagem de Atividades

Prof. Daniel Caetano

Objetivo: Apresentar os principais conceitos envolvidos na modelagem de atividades e em que situações ela deve ser usada na modelagem orientada a objetos. Capacitar para a criação e interpretação de diagramas de atividades.

Bibliografia: BEZERRA; DEITEL e DEITEL.

INTRODUÇÃO

Conceitos Chave:

- Estados de Objetos e Atividades
    * Diagramas de Estados e Diagramas de Atividades
    * Fluxos de Estado
- Diagramas de Estados => Mudança de Estados de Objetos
    * Eventos (Login / Logout...)
- Diagramas de Atividades => Mudanças de Estados de Atividades
    * Resultado de ações (fluxo de Controle)

Uma parte importante da modelagem dinâmica de um sistema orientado a objetos é a especificação dos estados de objetos e atividades. Esta modelagem é feita para indicar os diferentes possíveis estados de um objeto ou de uma atividade a ser desempenhada por ele.

A representação em UML destes modelos é feita através dos *diagramas de estados* e dos *diagramas de atividades* e representam os *fluxos* que levam de um estado para outro. Em linhas gerais, os diagramas são muito similares, embora os diagramas de estados representem mudanças de estados de um objeto de acordo com eventos ocorridos (por exemplo: houve login, houve logout...) e os diagramas de atividades representam os estados de uma atividade de acordo com um fluxo de controle.

Neste curso será abordado apenas o diagrama de atividades e, aqueles alunos que tiverem interesse nos diagramas de estados, podem tomar maior contato com os mesmos através da bibliografia indicada.

1. DIAGRAMAS DE ATIVIDADES

Conceitos Chave:

- Diagrama de Atividades x Fluxogramas
    * Estado inicial
    * Ações intermediárias
    * Ramificações (Decisões)
    * Estado final
- Projeto Estruturado x Orientado a Objetos
- Uso na modelagem orientada a objetos
    * Tornar claro caso de uso
    * Explicar método complexo

Antes de ser exposta qualquer informação sobre a representação UML dos diagramas de atividades, é importante apresentar o que é e para que serve o digrama de atividades.

Um diagrama de atividades é muito similar aos *fluxogramas* utilizados no projeto estruturado: parte-se de um estado inicial e ações são indicadas em um fluxo, podendo haver ramificações do fluxo de acordo com os resultados das ações.

Este tipo de diagrama é muito eficiente para indicar *algoritmos*, e por isso era muito usado na programação estruturada como uma forma de traçar a espinha dorsal do sistema; entretanto, quando se trabalha com orientação a objetos, a espinha dorsal do sistema é dada pelo diagrama de classes... para que serviria um diagrama de atividades na modelagem orientada a objetos?

Por duas razões fundamentais:

a) Para tornar mais claro um caso de uso. Como pode ser observado, a descrição textual de um caso de uso é uma atividade complexa, porque a linguagem textual é unidimensional e, ao descrever um caso de uso, é preciso representar ramificações e outras condições complexas. Assim, para os casos de uso mais complexos e com muitas variações, o emprego de um diagrama de atividades para tornar explícita seu fluxo de operação é sempre bem-vindo.

b) Para explicar um método complexo. Alguns métodos como "armazenarNome" certamente não precisam de explicações mais detalhadas; entretanto, alguns métodos não são tão simples, como por exemplo: "calcularSolucaoPorLabelSetting" ou mesmo um método do tipo "verificarSeExistePizzaComNome". Nestes casos, é fundamental usar um diagrama de atividades para explicitar o funcionamento de tais métodos.

2. NOTAÇÃO UML PARA DIAGRAMAS DE ATIVIDADES

Conceitos Chave:

- Notação
  * Estado inicial
  * Estado final
  * Estado-Ação
  * Ramificações
  * Junções
  * Transições
- Processamento paralelo

Os Diagramas de Atividades possuem uma notação básica bastante simples. Os principais elementos são: Estado-Ação, Estado Inicial, Estado Final, Ramificações, Junções e as Transições. A notação destes elementos pode ser verificada na Figura 1.

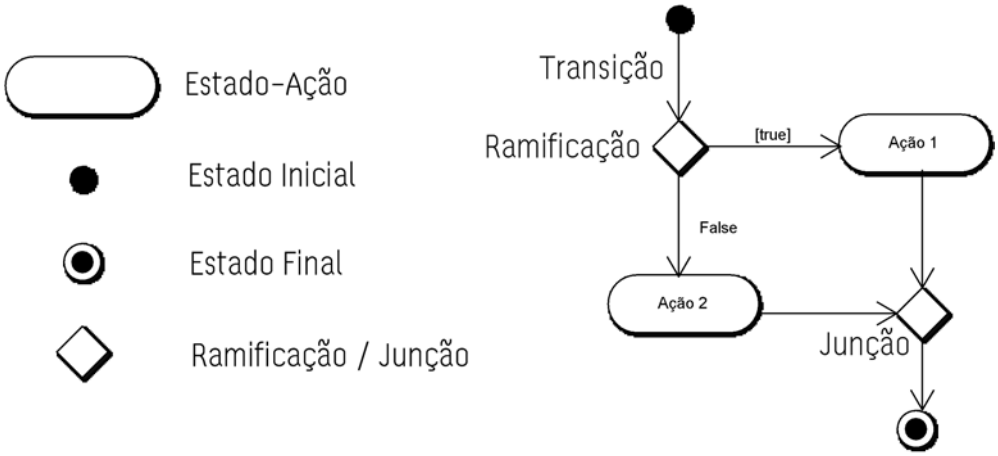


Figura 1: Notação dos elementos de um diagrama de atividades

Estado Inicial: Todo diagrama inicia com um. Nunca um diagrama deve possuir mais que um estado inicial. Este ponto indica o início do fluxo de atividades e, portanto, deve ter apenas uma saída.

Estado Final: Presente na maioria dos diagramas, podem existir vários diferentes. Em alguns casos não há nenhum estado final, o que indica que o fluxo é cíclico. Estados finais só devem possuir uma entrada.

Estado-Ação: Toda ação intermediária necessária para o cumprimento da atividade deve ser indicada como um estado-ação. Todo estado ação tem um nome, que deve indicar exatamente o que ocorre naquele estado ação. É possível indicar operações com variáveis nos estado-ação como, por exemplo: ***i:=0*** . Um estado-ação deve possuir uma entrada e uma saída.

Ramificação: Uma ramificação indica os diferentes caminhos que o fluxo pode tomar, de acordo com o resultado do último estado ação. É sempre necessário indicar a condição que leva por aquele caminho entre colchetes como, por exemplo: ***[i<0]*** ou ***[true]*** ou ***[else]***... Uma ramificação possui uma entrada e possui pelo menos duas saídas.

Junção: Sempre que dois ou mais fluxos se unirem em algum momento, deve ser usada a junção. A junção é representada de forma similar à ramificação, mas possui duas ou mais entradas e apenas uma saída. Adicionalmente, não é necessário indicar nenhum tipo de informação de condição.

Transição: Todos os elementos citados anteriormente são interligados por setas que indicam a transição entre um estado e outro. Estas setas de transição podem possuir nomes, embora isso seja incomum nos diagramas de atividades (são comuns nos de estado); entretanto, no caso das setas de transição que saem de uma ramificação, é nelas que deve ser indicada a condição da ramificação.

Há outros elementos nos diagramas de atividades, que permitem indicar processamento paralelo, sincronia de atividades etc., mas estes fogem ao escopo do curso. Recomenda-se a consulta da bibliografia para mais informações.

3. CONSTRUINDO UM DIAGRAMA DE ATIVIDADES

Conceitos Chave:

- Regra?
    * Modelo de Casos de Uso
    * Diagrama de Seqüência
- Pensar => Programação
- Exemplo: verificarSeExistePizzaComNome

Não existe, na verdade, uma regra para construir um diagrama de atividades. No caso de sua utilização para detalhar um caso de uso, o texto do caso de uso pode ser usado como ponto de partida; no caso de um diagrama que represente um algoritmo, entretanto, não há muito apoio.

Algumas vezes é possível se basear em algumas informações dos diagramas de seqüências, mas isso nem sempre é possível ou disponível. Assim, resta parar e pensar, como se estivesse programando, qual seria exatamente a seqüência de ações que deveriam ser executadas.

Por exemplo: no caso do método "verificarSePizzaExisteComNome(nome)", pode-se descrever textualmente o algoritmo como:

1) Verifica na lista interna de pizzas se há alguma pizza.
2a) Se não houver, retorna informando "null".
2b) Se houver, pegue a referência para a primeira pizza da lista.
3) Solicite que ela verifique se o nome dela é o "nome" passado como parâmetro.
4a) Se sim, retorna informando a referência para ela.
4b) Se não, verifica se há mais pizzas.
5a) Se não há mais pizzas, retorna informando "null".
5b) Se há, pega próxima pizza.
6) Retorna ao passo 3.

Esquematicamente, isso ficaria como indicado na figura 2.

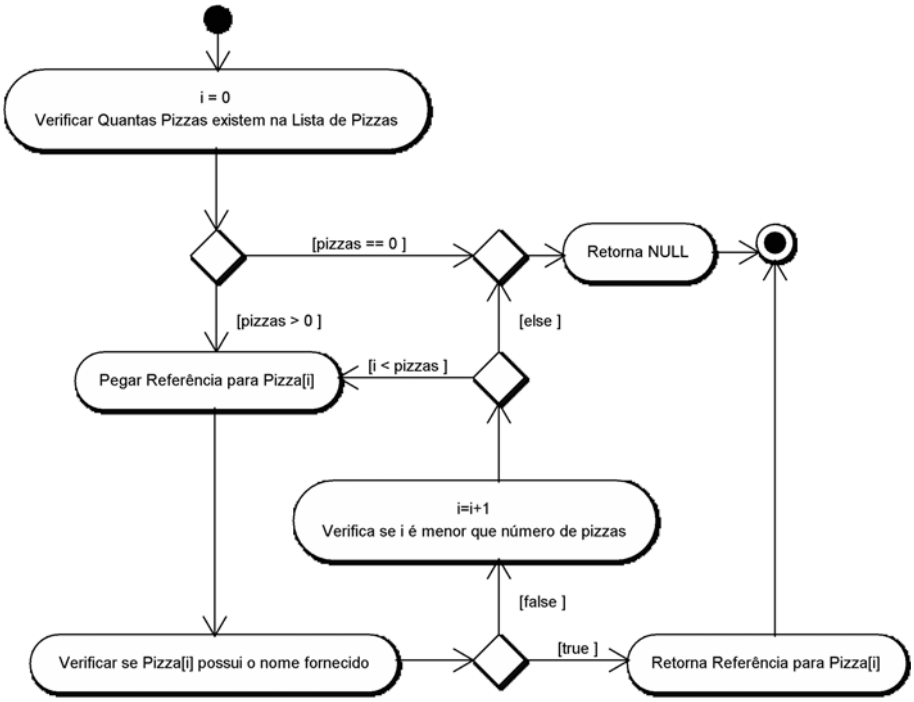


Figura 2: Diagrama de atividades do método verificaSeExistePizzaComNome(nome)

4. ATIVIDADE

Crie o diagrama de atividades para o método "armazenarIngrediente(ingrediente)" da Pizza, lembrando que ela não pode armazenar duas vezes o mesmo ingrediente.

5. BIBLIOGRAFIA

BEZERRA, E. **Princípios de Análise e Projeto de Sistemas com UML**. 2ed. Rio de Janeiro: Editora Campus Elsevier, 2007.

DEITEL, H.M; DEITEL, P.J. **Java**: como programar - 6ed. São Paulo: Pearson-Prentice Hall, 2005.