Unidade II

3 TECNOLOGIA DE APOIO AO PROJETO ORIENTADO A OBJETOS

3.1 A UML

Antes de nos aprofundarmos sobre o tema, listaremos a seguir, algumas das principais confusões a respeito da UML.

A UML **não** é:

- Uma linguagem de programação.
- Uma plataforma de desenvolvimento.
- Uma ferramenta de modelagem.
- Um *software*.

A UML (Unified Modeling Language), na definição de seus criadores, Booch, Jacobson e Rumbaugh (2006, p.13) "é uma linguagem-padrão para elaboração da estrutura de projetos de *software* [...] adequada para a modelagem de sistemas".

Os autores ainda pontuam:

- A UML é apenas uma linguagem.
- É independente do modelo de processo adotado.
- É destinada a visualização, especificação e documentação de artefatos.

Vamos então tentar desmitificar um a um os mal-entendidos acerca da UML.

- Uma linguagem de programação

A UML não é uma linguagem de programação, muito embora seja possível a geração de código a partir de alguns diagramas, o diagrama de classes principalmente, assim como o inverso, ou seja, a geração de diagramas a partir de código-fonte.

Todavia é importante que se tenha em mente que a geração de código não é algo propriamente da UML, mas sim de ferramentas de modelagem que tenham a UML como padrão e que tenham recursos para geração de código ou de engenharia reversa.

- Uma plataforma de desenvolvimento.
- Uma ferramenta de modelagem.
- Um *software*.

Existem inúmeras ferramentas no mercado que se utilizam da UML para a geração de diagramas, mas a UML em si não pode ser considerada uma plataforma ou uma ferramenta de modelagem, tampouco um *software*.

A confusão, nesse ponto, dá-se pela falsa sensação de que não conseguimos trabalhar com UML e desenvolver diagramas sem que tenhamos uma ferramenta de modelagem.

Nesse caso, as ferramentas de modelagem são importantes facilitadores na utilização da UML e nada mais.

Assim como na fase de análise, ou na disciplina de análise de sistemas OO, na fase de projetos a UML é uma das principais ferramentas de apoio para a modelagem da solução.

Mas antes de saber como utilizar um diagrama, precisamos saber quais diagramas devemos utilizar e em quais circunstâncias, dentro da atividade de projetos.

Segundo a abordagem de Kruchten (1995), um sistema de *software* pode ser organizado em cinco visões, e cada visão possui um conjunto de diagramas UML, que representam aspectos particulares desse sistema.

Saiba mais

A mesma divisão pode ser vista, em menor grau de detalhe, em:

BOOCH, G.; JACOBSON, I.; RUMBAUGH, J. *UML*: guia do usuário. 2. ed. Rio de Janeiro: Campus, 2006.

A figura a seguir demonstra as cinco visões propostas por Kruchten (1995) e por Booch, Jacobson e Rumbaugh (2006):

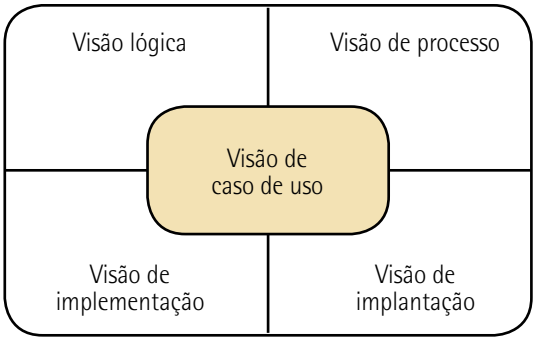

Figura 6 – Visões da UML

Visão de caso de uso

Tem como objetivo capturar as funcionalidades, os requisitos, e seu comportamento sob a ótica do usuário final, ou dos atores.

A visão de caso de uso é centralizada, uma vez que o que é produzido nessa visão é a base para as outras visões do sistema.

Visão lógica

Também chamada de visão de projeto ou visão de classe, tem como objetivo representar como as funcionalidades serão implementadas sob o aspecto da solução de projeto.

Representa a estrutura estática de um sistema, seus componentes e o relacionamento entre eles e como esses interagem para resolver um determinado problema. Essa interação é capturada pela estrutura dinâmica do sistema.

Visão de processo

Também chamada de visão de concorrência, a visão de processo tem como objetivo capturar aspectos de paralelismo de execução de atividades sob o ponto de vista não funcional de um sistema.

A visão de processo representa uma visão mais técnica do sistema tratada como unidades de processos e processamentos, que podem ocorrer de forma síncrona ou paralela. Essas unidades também podem ser interpretadas como subsistemas.

Visão de implementação

Também chamada de visão de componentes, a visão de implementação tem como objetivo representar aspectos físicos necessários para a construção do sistema e como eles interagem e fazem interface com o sistema.

São exemplos desses aspectos físicos: sistemas de *software*, programas e rotinas internas, bancos de dados e bibliotecas.

Visão de implantação

Também chamada de visão de organização, tem como objetivo representar a organização física de *hardware* do sistema, como computadores, servidores e periféricos, e como eles se relacionam com o sistema.

Essa visão é utilizada principalmente no processo de implantação, também chamado de instalação ou distribuição do sistema.

O quadro a seguir mostra as visões e os respectivos diagramas da UML que compõem cada visão.

Quadro 5 – Visões da UML x Diagramas UML

Visão	Diagramas
Visão de caso de uso	Diagrama de caso de uso Diagrama de processo Diagrama de atividade
Visão lógica	- Estrutura estática: Diagrama de classe Diagrama de objeto - Estrutura dinâmica: Diagrama de estado Diagrama de sequência Diagrama de colaboração Diagrama de interação Diagrama de atividade
Visão de processo	São utilizados os mesmos diagramas utilizados na visão lógica, mas com ênfase na linha de execução do sistema.
Visão de implementação	Diagrama de componentes
Visão de implantação	Diagrama de implantação

Na fase de análise trabalhamos com mais ênfase na visão de caso de uso, desenvolvendo os diagramas de caso de uso, de processo e de atividade, além de trabalhar com os diagramas de classe e sequência sob o aspecto de conhecimento do domínio.

Já na fase de projetos daremos ênfase à visão lógica, bem como à visão de implementação e à de implantação, que são as visões e, consequentemente, os diagramas que dão suporte à modelagem do aspecto solução dos requisitos elicitados nas demais visões.

É importante que tenhamos em mente exatamente quais diagramas nos estão disponíveis para uso e exatamente o que estamos querendo representar quando do uso de um ou de outro, e para isso é importante gravar o Quadro 5.

Outro aspecto importante que devemos levar em consideração na adoção de um diagrama para representar algo é a natureza desse diagrama: se ele representa uma visão estática ou dinâmica.

O modelo estrutural estático representa a coleção desses objetos, seus atributos, métodos e seus relacionamentos, no entanto as formas pelas quais eles interagem para resolver um problema não pode ser visualizada em um diagrama de classes, tampouco em um diagrama de objetos.

A visão estrutural dinâmica, também chamada de visão comportamental, tem como objetivo representar a interação dos objetos para atingir um determinado objetivo; essa interação se dá ao longo de uma linha de tempo.

Observação

O modelo estrutural dinâmico é originado do modelo estático. Não existe comportamento dinâmico que não tenha sido representado nos diagramas de classe ou de objeto. Qualquer tipo de alteração no modelo dinâmico acarreta mudança no estático e vice-versa.

A figura a seguir mostra a estrutura dos diagramas da UML, com base em Booch, Jacobson e Rumbaugh (2006, p. 96-9).

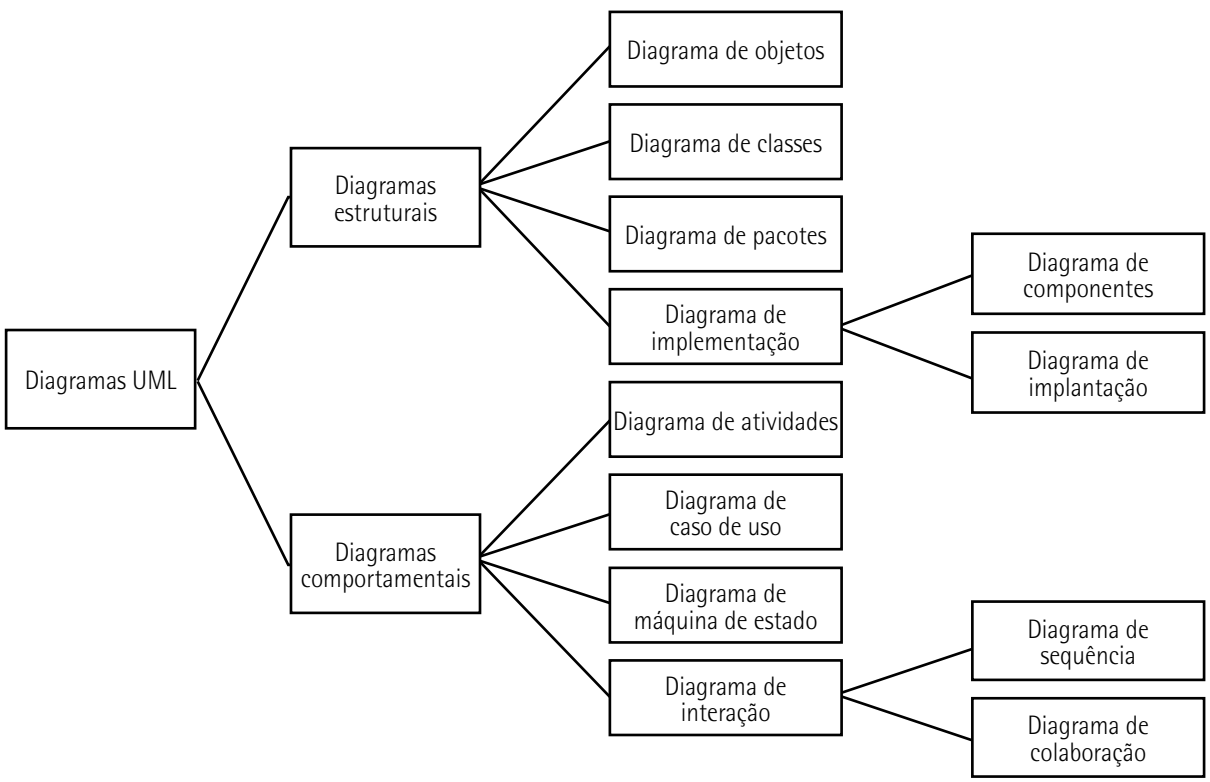

Figura 7 – Diagramas estruturais e comportamentais da UML

Neste livro-texto iremos dar ênfase aos seguintes diagramas estruturais: diagrama de classes, diagrama de pacotes, diagrama de componentes e diagrama de implantação. Além dos diagramas comportamentais: diagrama de máquina de estado, diagrama de sequência e diagrama de colaboração.

3.2 Ferramentas de modelagem UML

Como já vimos, a UML é uma coisa e as ferramentas de modelagem que têm como base a UML são outra coisa.

Neste caso, as ferramentas agem como grandes facilitadoras para o uso da UML.

Um dos grandes desafios, dada a imensa variedade de ferramentas de modelagem disponíveis atualmente no mercado, é escolher qual ferramenta deve ser usada para modelar o projeto do sistema.

Durante a fase de projeto chega a hora de adotar uma ferramenta de modelagem, e alguns aspectos importantes devem ser considerados:

- A ferramenta está efetivamente de acordo com a UML?
- A ferramenta está atualizada com a UML 2.0?
- A ferramenta é de uso livre, ou seja, sem custos? Sendo livre, existe alguma limitação de uso (como limite de quantidade de diagramas ou limite de quantidade de classes)?
- A ferramenta possui recursos de engenharia reversa? Esses recursos são limitados a alguma linguagem? As linguagens oferecidas estão de acordo com as suas necessidades?

Observação

Engenharia reversa é o nome que se dá à capacidade da ferramenta de modelagem de gerar código-fonte a partir de diagramas e vice-versa.

Todos esses pontos são extremamente importantes quando da seleção de uma ferramenta de modelagem, principalmente, porque existem inúmeras e incontáveis ferramentas disponíveis.

O bônus da grande variedade de ferramentas está no amplo leque de escolhas que o projetista terá, e quanto maior a concorrência entre as ferramentas, maior é a qualidade do produto final.

O ponto negativo está relacionado à própria UML. Por se tratar de um padrão de linguagem aberto, qualquer um pode seguir a especificação, desenvolver sua própria ferramenta e disponibilizá-la para a comunidade.

No entanto, existem muitas ferramentas de boa qualidade e um grande número de ferramentas de qualidade duvidosa.

Portanto é essencial a realização de uma grande pesquisa antes da adoção de uma ferramenta de modelagem; é importante realizar um mapeamento e verificar em que nível a ferramenta se encaixa nas características anteriormente citadas.

Segue uma relação de algumas ferramentas que existem no mercado:

- Enterprise Architect.
- Rational Rose.
- Visual Paradigm.
- Astah.
- StarUML.

Saiba mais

Pesquise e veja uma lista de ferramentas de modelagem em:

<http://uml-directory.omg.org/>.

O *StarUML* é uma ferramenta gratuita que pode ser baixada no *site*:

<http://staruml.io/>.

3.3 As ferramentas CASE

O objetivo da engenharia de *software* é apoiar o desenvolvimento de *software* a partir de técnicas para especificar, projetar, manter e gerir um sistema de *software* (SOMMERVILLE, 2010). Ela é baseada em três pilares: métodos, ferramentas e processos.

Ferramentas, de qualquer natureza, são utilizadas na engenharia para dar suporte ou auxiliar, na execução de uma ou mais atividades.

CASE é o acrônimo em inglês para Computer-Aided Software Engineering, ou em português: Engenharia de *Software* Auxiliada por Computador.

Sommerville (2010, p. 512), define CASE como "o processo de desenvolvimento de *software* com uso de suporte automatizado".

Pressman (2006) segue a mesma linha e define CASE como um sistema de *software* que dá suporte a profissionais da engenharia de *software* em todas as atividades do processo de *software*.

Huang (1998) sai um pouco da linha de Pressman (2006) definindo CASE como um produto de *software* que dá suporte a uma ou mais atividades do processo de *software*, enquanto Pressman (2006) defende que uma ferramenta CASE deva dar suporte a todas as atividades do processo.

Qualquer que seja a definição, todos convergem para a mesma linha de pensamento, que é justamente a oposta do que muitos adotam quando falamos em ferramenta CASE, pois uma ferramenta destas não serve apenas para desenho de diagramas.

Ferramentas CASE são classificadas de acordo com a sua funcionalidade dentro do processo; seguem alguns exemplos.

- Ferramentas para planejamento de projetos.
- Ferramentas para gerenciamento de projetos.
- Ferramentas para análise e projeto.
- Ferramenta para construção (desenvolvimento).
- Ferramentas para integração de testes.

Algumas de suas principais características:

- Criação de diagramas e manutenção da consistência entre esses diagramas.
- Engenharia reversa: geração de código a partir de diagramas e vice-versa.
- Gerenciamento de versão.
- Gerenciamento de mudança.
- Testes automáticos, verificação e relatórios.

Precisamos ter algo muito claro em mente: essas ferramentas têm como único objetivo apoiar o processo de *software*, e não substituí-lo. Elas devem ser usadas em conjunto com o processo de engenharia de *software*.

Sendo assim, todas as ferramentas CASE devem ser integradas ao processo e consequentemente entre si.

Por exemplo, um diagrama de classes gerado na fase de análise, ou seja, com o auxílio de uma ferramenta de modelagem UML (conforme vimos anteriormente), é utilizado como base para o diagrama de classes que será gerado na fase de projeto.

Neste caso, um diagrama não substitui, mas complementa o outro. Logo, eles passam a ter uma relação, na qual uma alteração ou mudança em um necessariamente se refletirá no outro; assim, temos algumas necessidades implícitas a esse cenário, como: gerenciamento de dependência e gerenciamento de versão.

A mesma linha de pensamento podemos usar na relação de diagrama e descrição de caso de uso, com os artefatos produzidos na fase de projetos; consequentemente, esses artefatos devem ter relação com o código-fonte produzido na construção, que, por sua vez, possui relação com os planos de testes, e assim por diante.

A forma de integração das ferramentas CASE, e não somente sua simples adoção, é fator fundamental para o sucesso do projeto, mas não somente isso.

Pfleeger (2004, p. 27) mostra que apenas a integração do processo à ferramenta não garante o sucesso do projeto. A autora afirma que durante um longo período, fabricantes dessas ferramentas se esforçaram para vender a ideia de que apenas "ferramentas padronizadas e integradas em ambientes de desenvolvimento, melhorariam o desenvolvimento de *software*".

A palavra-chave para a melhoria no desenvolvimento de *software* a partir do uso de ferramentas CASE é "maturidade". Apenas a maturidade do ambiente CASE associada à maturidade da equipe aumenta o fator de produtividade (PLEEGER, 2004. p. 89).

Nós, enquanto analistas e projetistas, devemos estar atentos quando do momento da adoção de uma ferramenta CASE em um de nossos projetos, uma vez que, como vimos, é importante, na fase de projetos, que tenhamos suporte e rastreabilidade dos artefatos produzidos na análise, dos que serão produzidos no projeto e ainda do que será efetivamente construído.

O trabalho de Tsuda *et al.* (1992) mostra um pouco dessa realidade, discutindo alguns fatores que levam ao aumento da produtividade com a adoção de ferramentas a partir de um estudo de caso de um cenário real.

Uma frase do artigo de Brooks (1987, p. 1), de título reflexivo *No Silver Bullet: Essence and Accidents of Software Engineering*, em português, algo como "não existe bala de prata: essência e acidentes da engenharia de *software*", é ainda muito atual, pois "não existe um único desenvolvimento, em qualquer tecnologia ou gestão técnica, que por si só prometa uma ordem de grandeza de melhoria de uma década em termos de produtividade, confiabilidade ou simplicidade".

Saiba mais

Para aprofundar seus conhecimentos sobre avaliação de *benchmarking* envolvendo algumas das principais ferramentas CASE, leia:

MAHDY. A. A. E-R.; AHMED, M. M. A. E-S.; ZAHRAN, S. M. Flexible CASE tools for requirements engineering: a benchmarking survey. *In*: INTERNATIONAL CONFERENCE OF INFORMATICS AND SYSTEMS, 2008. *Proceedings*... Cairo, 2008. Disponível em: <http://infos2008.fci.cu.edu.eg/infos/se_03_p020-026.pdf>. Acesso em: 17 abr. 2015.

Ferramentas CASE ou ferramentas de apoio à engenharia de *software* podem ser classificadas em dois grupos: *front-end* e *back-end*.

3.4 Tecnologia *back-end*

Tecnologias de apoio ao projeto classificadas como *back-end* estão relacionadas ao gerenciamento e armazenamento das informações. São os Sistemas Gerenciadores de Banco de Dados (SGBD).

Segundo Silberschatz, Korth e Sudarshan (2006, p. 1), "um sistema de gerenciamento de banco de dados (DBMS) é uma coleção de dados inter-relacionados e um conjunto de programas para acessar esses dados que tem como objetivo fornecer uma maneira de recuperar informações de um banco de dados de forma conveniente e eficiente".

Observação

O objetivo desta disciplina não é se aprofundar no tema Banco de Dados. Abordaremos apenas aspectos que tangenciem a fase de análise e que tenham correlação com o tema.

A figura a seguir mostra a evolução do conceito de banco de dados, à medida que cresceu também a complexidade dos sistemas de informação.

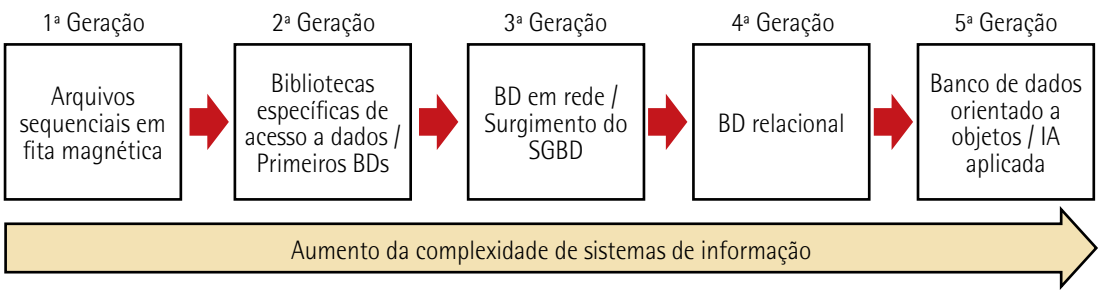

Figura 8 – Evolução dos bancos de dados

A primeira geração dos repositórios de informações é marcada pela gravação sequencial das informações em fita magnética, sendo de responsabilidade do programador desenvolver seu próprio algoritmo para leitura e escrita dessas informações.

Na segunda geração começam a surgir as primeiras bibliotecas de acesso aos dados, facilitando um pouco o dia a dia do desenvolvedor. É nessa época também que surgem os primeiros esboços de bancos de dados, que viriam em substituição às fitas magnéticas.

Em seguida, na terceira geração, surgem as padronizações no acesso a dados, sistemas que faziam desde o gerenciamento da atualização até a leitura das informações. A esses sistemas foi dado o nome de Sistemas Gerenciadores de Banco de Dados (SGBD).

A quarta geração é aquela na qual se baseia o que temos atualmente: consolidação dos SGBDs, dos bancos de dados relacionais e da linguagem SQL.

A quinta geração ainda é algo incipiente e que ainda carece de maior desenvolvimento, que é a inclusão de banco de dados orientado a objetos e a incorporação de técnicas de inteligência artificial no conceito de banco de dados (MEDEIROS, 2013).

Na fase de projetos, e isso se aplica também ao paradigma da orientação a objetos, atualmente se utilizam dois modelos de banco de dados: o modelo entidade-relacionamento e o modelo orientado a objetos, sendo este último utilizado em menor escala.

O modelo entidade-relacionamento ou E-R, tem como base a percepção do mundo real como um conjunto de entidades e do relacionamento entre elas. As entidades são caracterizadas no banco de dados pelos seus atributos (SILBERSCHATZ; KORTH; SUDARSHAN, 2006).

Similar ao E-R, o modelo orientado a objetos tem por base um conjunto de objetos. Cada objeto possui um conjunto de características (atributos) que, ao possuírem um determinado conjunto de valores, passam a constituir uma identidade para o objeto.

A diferença conceitual do modelo orientado a objetos para o E-R está em alguns pontos:

- No modelo orientado a objetos, cada objeto possui um conjunto de códigos que operam sobre este objeto, chamado de método.
- Objetos que possuem o mesmo conjunto de atributos e métodos são denominados classe.
- Cada objeto possui uma identidade única independente dos valores contidos em seus atributos, ou seja, mesmo que um objeto possua os mesmos valores dos atributos de outro objeto no mesmo banco de dados, estes possuem identidades diferentes.
- Adoção de mecanismos de relacionamento: composição, agregação e herança.

Lembrete

Os conceitos fundamentais do modelo orientado a objetos para banco de dados são os mesmos conceitos fundamentais do paradigma da orientação a objetos: objeto, classe, métodos, atributos, herança e identidade.

O banco de dados orientado a objetos ainda esbarra em alguns pontos, frutos da própria incipiência da tecnologia.

Um desses pontos é a mistificação do desempenho. A pesquisa de Saxena e Pratap (2013) é um exemplo de trabalho que faz um comparativo entre as duas tecnologias E-R e OO em um cenário específico.

O resultado obtido nessa pesquisa, e que pode ser visto na figura a seguir, é muito próximo ao cenário obtido por muitas pesquisas que seguem essa mesma linha: o desempenho do BD OO aumenta à medida que a complexidade do negócio aumenta.

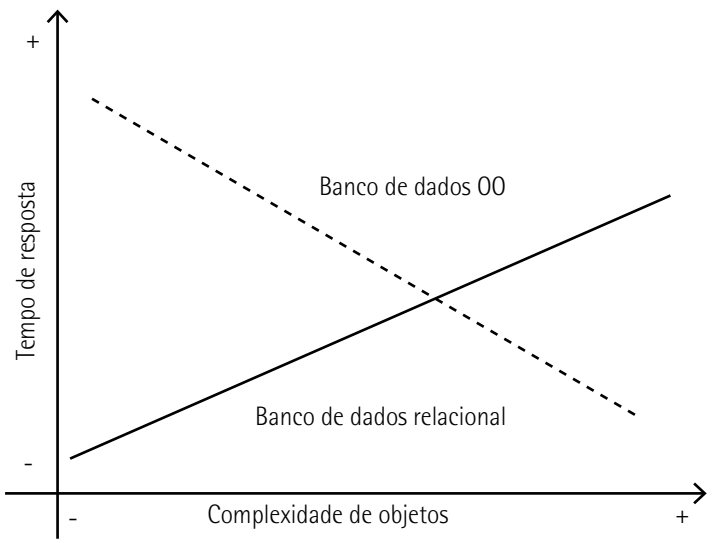

Figura 9 – Comparativo de desempenho: banco de dados OO *versus* relacional

A pesquisa de Saxena e Pratap (2013) chega à conclusão de que o tempo de resposta de um banco de dados relacional é maior em todos os cenários – atualização, leitura e escrita – quando o número de objetos e a complexidade deles é baixa; do contrário, se o número e a complexidade de objetos for maior, o desempenho do banco de dados orientado a objetos mostrar-se-á mais satisfatório.

Atualmente, em quantidade de uso, bancos de dados orientados a objetos possuem números menores se comparados a bancos de dados relacionais, e muito disso se deve à falta de maturidade e à falta de uso de uma padronização.

São dois os principais padrões para banco de dados orientado a objetos: o padrão Common Object Request Broker Architecture (CORBA) e o padrão Object Request Broker (ORB), ambos definidos pela OMG.

O padrão CORBA define uma arquitetura-padrão para comunicação: troca de informações entre objetos independentemente do produto, *hardware* ou sistema operacional, garantindo a portabilidade e a interoperabilidade (OMG, 2014).

Saiba mais

Para aprofundar seus conhecimentos sobre a especificação CORBA, acesse:

<http://www.omg.org/spec/CORBA/>.

O Object Management Architecture (OMA) define um padrão arquitetural para gerenciamento de objetos em um modelo de dados em cujo centro está o padrão ORB, que, segundo a OMG (2014), "fornece uma infraestrutura que permita que objetos se comuniquem independente das plataformas e de técnicas específicas utilizadas para implementar os objetos abordados. A ORB garante portabilidade e interoperabilidade dos objetos através de uma rede de sistema heterogêneo".

Saiba mais

Sobre a especificação OMA, acesse:

OBJECT MANAGEMENT ARCHITECTURE. *Object Management Group*, 2014. Disponível em: <http://www.omg.org/oma/>. Acesso em: 17 abr. 2015.

Paralelamente aos padrões definidos pela OMG, surge também o padrão definido pela Object Data Management Group (ODMG): o Object-oriented Database Management Systems (OODBMS), que também tem como objetivo definir um padrão de comunicação entre aplicações orientadas a objetos e um sistema gerenciador de banco de dados orientado a objetos (CATTELL *et al.*, 2000).

Saiba mais

Saiba mais sobre o padrão da ODMG em:

CATTELL, R. G. *et al. The object data management standard*: ODMG 3.0. California: Morgan Kaufmann, 2000.

Ou acesse:

<http://www.odbms.org/>.

Essa falta de maturidade na padronização ainda não colocou os principais participantes do mercado da indústria de bancos de dados (Microsoft, Oracle etc.) com a força que se espera no mercado de banco de dados orientado a objetos.

A seguir a lista de alguns dos principais bancos de dados orientados a objetos disponíveis atualmente:

- Gemstone.
- ObjectStore.
- Versant.
- Objectivity/DB.
- Easy/DB.
- Ontos/DB.
- Jasmine.

3.5 Tecnologia *front-end*

Tecnologias de apoio *front-end* são subdivididas em duas categorias: ferramentas de modelagem e linguagens de programação OO.

As ferramentas de modelagem, voltadas para o paradigma da orientação a objetos, em sua maioria, restringem-se às ferramentas que possuem a UML como base.

Lembrete

Como vimos, o assunto Ferramentas de Modelagem é extenso e costumeiramente gera dúvidas quanto a sua relação com a UML, conforme debatemos nas seções iniciais desta unidade.

As linguagens de programação orientada a objetos são mecanismos de implementação do modelo de projeto que desenhamos na fase de projeto.

Analogamente ao exemplo da construção de uma casa, é como se o modelo produzido na fase de projetos fosse o nosso conjunto de plantas e os mecanismos de construção desse modelo – tijolos, areia, cimento, canos etc. – fossem a nossa plataforma de desenvolvimento orientado a objetos.

A primeira linguagem orientada a objetos pura utilizada para a construção dentro do paradigma OO foi a linguagem Smalltalk, que teve seu uso difundido na década de 1980 e tem importância para as gerações futuras, pois sua terminologia é referencial para outras linguagens OO (THE HISTORY..., 2010).

Saiba mais

Saiba mais sobre a linguagem Smalltalk em:

<http://www.smalltalk.org/>.

Não é qualquer linguagem de programação que pode ser considerada orientada a objetos; para tal, é necessário que se cumpram determinados prerrequisitos (LEE; TAPFENHART, 2001):

- Encapsulamento de dados – garantir a confiabilidade e a capacidade de modificação do sistema de *software*, expondo apenas o necessário ao restante do sistema, sendo que o estado de um objeto deve estar representado em atributos privados, visíveis apenas no escopo do próprio objeto.

- Abstração de dados – a linguagem deve estar apta a implementar um tipo abstrato de dados, ou seja, um conjunto de métodos utilizados para manipular essas informações.

- Coesão dinâmica – a linguagem deve implementar o mecanismo de troca de mensagens no qual existe o objeto chamador, que envia a mensagem e o objeto receptor que recebe a mensagem e executa um método.

- Herança – permitir a codificação de classes que sejam especializações de outras classes.

Lee e Tapfenhart (2001, p. 31) observam que as linguagens ditas puras OO, ou seja, que seguem à risca todos os conceitos do paradigma OO, tais como: Object C, Smalltalk e Eiffel, não se mostraram produtivas quanto ao desenvolvimento em larga escala de sistemas de *software* para fins comerciais.

Para tal, foi criada uma série de extensões das linguagens ditas puras que partem do princípio da extensão, e não da modificação, na qual, o programador pode utilizar técnicas não orientada a objetos quando apropriado.

Linguagens como: C++, Java e C# passam a ocupar fatia importante do mercado de desenvolvimento.

Saiba mais

Saiba mais sobre aplicação de C++ em projeto orientado a objetos em:

LEE, R. C; TAPFENHART, W. M. *UML e C++*: guia prático de desenvolvimento orientado a objeto. São Paulo: Makron Books, 2001.

Esta disciplina não tem como objetivo dar ênfase a uma linguagem específica, mas faz parte das habilidades necessárias do arquiteto, conforme vimos, o conhecimento aprofundado nas plataformas tecnológicas, uma vez que soluções técnicas poderão ser adotadas ou não, em função da plataforma escolhida.

De qualquer forma, é importante que nós, neste momento, nos livremos de alguns mitos e inverdades que possam limitar erroneamente nosso campo de visão.

3.5.1 Linguagens OO, um breve comparativo

Atualmente, quando falamos em plataforma de desenvolvimento, duas das mais utilizadas são: Java e Microsoft .NET.

Essas duas plataformas respondem por parcela significativa no mercado de desenvolvimento. Segundo a pesquisa do Instituto Gartner (DRIVER, 2007), aproximadamente 80% das grandes empresas dependem de ambas.

Essa pesquisa traz resultados interessantes, pois coloca em números questões sobre esse debate que muitas vezes envereda pelo lado passional ou por preferências pessoais às tecnologias A ou B.

Observação

O Instituto Gartner é um importante e confiável instituto de pesquisa norte-americano especializado na análise de tendências do mercado de tecnologia.

Um dos muitos indicativos interessantes que a pesquisa traz, e que começa a derrubar um pouco um dos principais mitos, está relacionado ao domínio de mercado por uma tecnologia. Seguem os números desses indicadores (DRIVER, 2007):

- Empresas consideradas pequenas: 70% utilizam Microsoft e menos de 5% utilizam Java.
- Empresas consideradas médias: 60% utilizam Microsoft e 25% utilizam Java.
- Empresas consideradas grandes: 25% utilizam Microsoft e 50% utilizam Java.

Em todos os intervalos de números a diferença para a totalidade é destinada a outras tecnologias, como 4GL, Legacy, dentre outros.

Outra conclusão importante, e que permeia muitas discussões, está relacionada à produtividade. Segundo a pesquisa, a produtividade do desenvolvedor na plataforma Java é menor que a produtividade do desenvolvedor Microsoft (DRIVER, 2007).

Um fator interessante deste indicador. Se compararmos o indicador flexibilidade com o fator produtividade, a flexibilidade técnica da plataforma Java será maior que a flexibilidade Microsoft, com um detalhe: a produtividade Java *versus* a produtividade Microsoft e a flexibilidade Java *versus* a flexibilidade Microsoft apresentam indicadores inversamente proporcionais.

Pode-se concluir que ambas as plataformas possuem qualidades e defeitos bem pontuados e atenuados, de tal forma que uma discussão que questione qual plataforma é melhor, sem uma análise minuciosa do cenário no qual será aplicada, certamente será direcionada para o lado passional da discussão, o que não nos parece algo relacionado à engenharia.

Saiba mais

Veja mais indicadores em:

DRIVER, M. Java and .NET: you can't pick a favorite child. *In*: DEVELOPER SUMMIT, 2007. Palm Springs. *Proceedings...* Palm Springs, 2007. Disponível em: <http://proceedings.esri.com/library/userconf/devsummit07/papers/keynote_presentation-mark_driver_gartner.pdf>. Acesso em: 22 abr. 2015.

4 PASSANDO DA ANÁLISE AO PROJETO

Como vimos, em um projeto de desenvolvimento de *software* temos a necessidade de representar as diversas visões desse *software* e para tal nos utilizamos das chamadas visões da UML, ou visões de modelagem.

A visão central deste modelo é a de caso de uso, que é a base de tudo e que fornece uma perspectiva do *software* sob o ponto de vista do externo a este *software* (negócio/ator).

Ainda na fase de análise, produzimos o modelo de classes conceitual que representa a estrutura estática da interação de objetos para resolver um determinado problema. Nesse modelo são representados os atributos, os métodos e como as classes se relacionam (herança, ligação, composição, agregação).

O diagrama de objetos também pode ser utilizado como complemento ao modelo de classes de domínio, uma vez que ele também representa uma visão estrutural que pode ser considerada como uma instância do diagrama de classes (BEZERRA, 2006).

Na fase de análise o modelo de classe é utilizado para representar objetos do domínio do problema. Logo, podemos dizer que o modelo de casos de uso somado ao modelo de classes de domínio tem por objetivo esclarecer o problema a ser resolvido.

Observação

Na fase de projetos utilizamos como insumo os artefatos produzidos durante a fase de análise que, dentre outras atividades, compreende elicitação e análise dos requisitos do negócio no qual o sistema de *software* se dispõe a atender uma necessidade ou resolver um determinado problema.

O modelo de classes evolui durante o ciclo de vida do *software* e na fase de projeto (*design*) é utilizado para representar objetos da solução.

Se o modelo de classes é o mesmo, quando termina a fase de análise e começa a fase de projetos?

Podemos dizer que a fase de projetos começa a partir do momento em que fechamos a primeira versão do modelo de classe de domínio; nesse momento, temos a transição análise-projeto.

4.1 Desenvolver o modelo de classes de projeto refinando o modelo conceitual

A perspectiva dada pelo modelo de classe de domínio não é suficiente para avançarmos para a fase de implementação (construção). Precisamos definir aspectos inerentes à solução do problema.

Há três perspectivas do modelo de classes:

- Modelo conceitual.
- Modelo de projeto.
- Modelo de implementação.

A seguir, cada modelo será detalhado para melhor compreensão.

4.1.1 Modelo conceitual

Desenvolvido na fase de análise, este modelo representa as classes no domínio do negócio em questão e não leva em consideração restrições inerentes à tecnologia a ser utilizada na solução de um problema.

O modelo conceitual representa uma entidade do mundo real que possui:

- Identidade: valor de uma característica que o identifica para reconhecimento.
- Atributos: qualidades e características.
- Comportamento: habilidades de processamento.

4.1.2 Modelo de projeto

Também chamado de modelo de especificação, o modelo de projeto é desenvolvido na fase de projeto (*design*) e é uma evolução do modelo conceitual, no qual são adicionados detalhes da solução de *software*.

Representa um objeto do mundo real, dentro de um universo de *software* que contém:

- Identidade: identificador em termos de implementação (em memória).
- Atributos: os atributos que definem o objeto passam a ter tipos de dados. Nota-se aqui, uma correlação maior com a linguagem ou plataforma de desenvolvimento a ser utilizada, uma vez que os tipos de dados podem variar de uma linguagem para outra, por exemplo, Int32, Int64, Integer, String, Double, Decimal etc.

Saiba mais

Para aprofundar seus conhecimentos sobre os tipos de dados presentes no Microsoft .Net Framework, consulte a biblioteca:

<http://msdn.microsoft.com/en-us/vstudio/aa496123.aspx>.

Se preferir a plataforma Java, consulte-a em:

<http://www.oracle.com/technetwork/java/index.html>.

- Comportamento: funções ou procedimentos que determinam o comportamento do objeto passam a possuir assinaturas. e é obrigatório que se defina o nível de visibilidade dos métodos.

Observação

A assinatura de um método, assim como a assinatura de uma função em algoritmos, é composta de: tipo de retorno + nome do método + parâmetros de entrada (tipo de dado, nome do parâmetro).

Vamos relembrar alguns pontos importantes sobre visibilidade de atributos e métodos, uma vez que são pontos obrigatórios na fase de projetos.

Visibilidade indica quando e em que nível um atributo ou um método de um objeto pode ser acessível aos objetos que se relacionam com ele. Em orientação a objetos, temos três níveis de visibilidade de atributos e métodos:

- Público: o atributo ou o método pode ser acessado por qualquer classe. Na UML, indicamos que um atributo ou um método é público utilizando o sinal + (positivo), conforme exemplo a seguir.

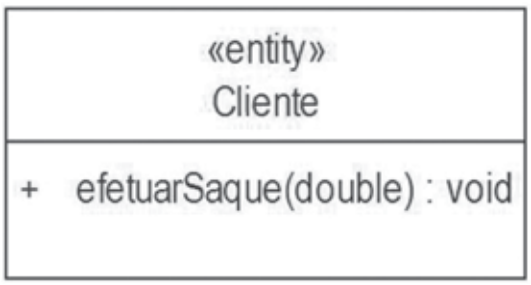

Figura 10 – Exemplo de representação de um método público

Ainda de acordo com a figura anterior, podemos dizer que o método efetuarSaque pode ser chamado por qualquer outro objeto.

- Privado: um atributo ou método privado pode ser acessado somente na própria classe em que está declarado. Na UML, indicamos que um atributo ou um método é privado utilizando o sinal - (negativo), conforme exemplo a seguir.

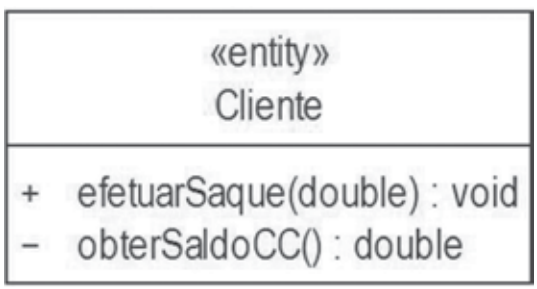

Figura 11 – Exemplo de representação de um método privado

Ainda a partir da figura apresentada, podemos dizer que o método obterSaldoCC pode ser chamado apenas dentro da própria classe Cliente. Por exemplo: poderíamos ter uma chamada a esse método dentro do método efetuarSaque.

Observação

Atributos e métodos marcados como privado são a chave para implementação de encapsulamento.

- Protegido: um atributo ou um método protegido pode ser acessado apenas na classe em que está declarado e em suas classes-filhas. Na UML, indicamos que um atributo ou um método é protegido utilizando o sinal # (sustenido), conforme exemplo a seguir.

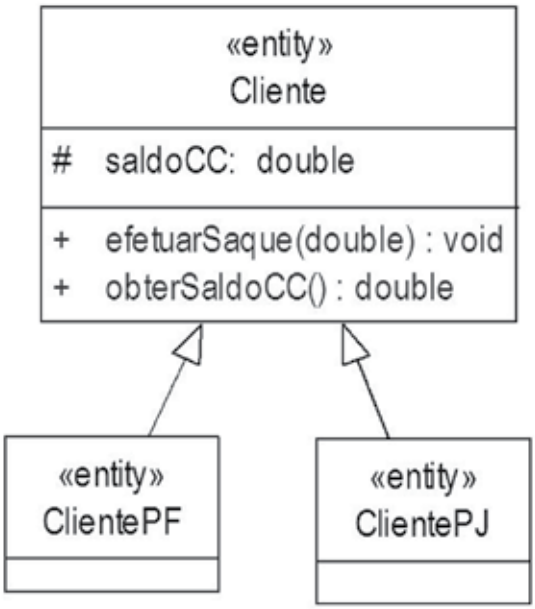

Figura 12 – Exemplo de representação de um atributo protegido

Podemos dizer, de acordo com a figura anterior, que o atributo saldoCC é acessível apenas na classe Cliente e nas suas classes-filhas: ClientePF e ClientePJ.

4.1.3 Modelo de implementação

O modelo de classes de implementação corresponde à implementação das classes em alguma linguagem de programação, por exemplo, C# (Microsoft .Net Framework) ou Java, como mostram as figuras a seguir.

Nestes exemplos, estamos codificando uma classe chamada Cliente que possui um atributo (saldo) e um método (efetuarSaque).

```
public class Cliente
{
    private double saldo;

    public void efetuarSaque(double valor)
    {
        saldo = saldo - valor;
    }
}
```

Figura 13 – Exemplo de implementação de classe em C#

```
public class Cliente
{
    private BigDecimal saldo;

    public void efetuarSaque(BigDecimal valor)
    {
        saldo = saldo - valor;
    }
}
```

Figura 14 – Exemplo de implementação de classe em Java

4.2 Atividades clássicas para passagem da análise para o projeto

Segundo Bezerra (2006), o objetivo desse momento do ciclo de vida do projeto é obter um nível de detalhamento dos modelos tal que possamos passá-los para os desenvolvedores implementarem o sistema (modelo de implementação).

Observação

Um pouco de vivência prática sobre a visão do arquiteto e a codificação: o arquiteto adentra a fase de construção codificando a estrutura ou adotando algum *framework* utilizando, dentre outros elementos, padrão de projeto, além de codificar os pontos da aplicação considerados fundamentais.

São três as atividades clássicas enumeradas por Bezerra (2006): detalhamento dos aspectos dinâmicos do sistema, refinamento dos aspectos estáticos e estruturais do sistema e definição de outros aspectos da solução.

4.2.1 Detalhamento dos aspectos dinâmicos do sistema

O objetivo dessa fase é a representação da interação dos objetos de forma dinâmica. O dinamismo está relacionado à representação do estado e do comportamento do objeto em uma linha de tempo.

Lembrete

O diagrama de classes representa uma visão estática, pois não nos dá o aspecto temporal de um objeto.

Os modelos que representam os aspectos dinâmicos de um sistema são: modelo de interações, modelo de estados e modelo de atividades; este último é o menos utilizado.

O modelo de interação dos objetos representa a forma pela qual estes interagem para resolver um determinado problema, e, conforme vimos pelo paradigma da orientação a objetos, a interação dos objetos se dá pela troca de mensagens, e estas são o centro da representação desse modelo.

Antes de entrar na modelagem das mensagens vamos relembrar os possíveis significados de uma mensagem.

Uma mensagem pode ter três significados (STADZISZ, 2002):

- Chamada de função ou procedimento: é o tipo mais utilizado de mensagem. Significa que um objeto está solicitando a execução de um método de outro objeto.

Neste caso, a função ou procedimento corresponde a um método público em um objeto que pode ser acessível a um objeto "chamador", conforme mostra a figura a seguir.

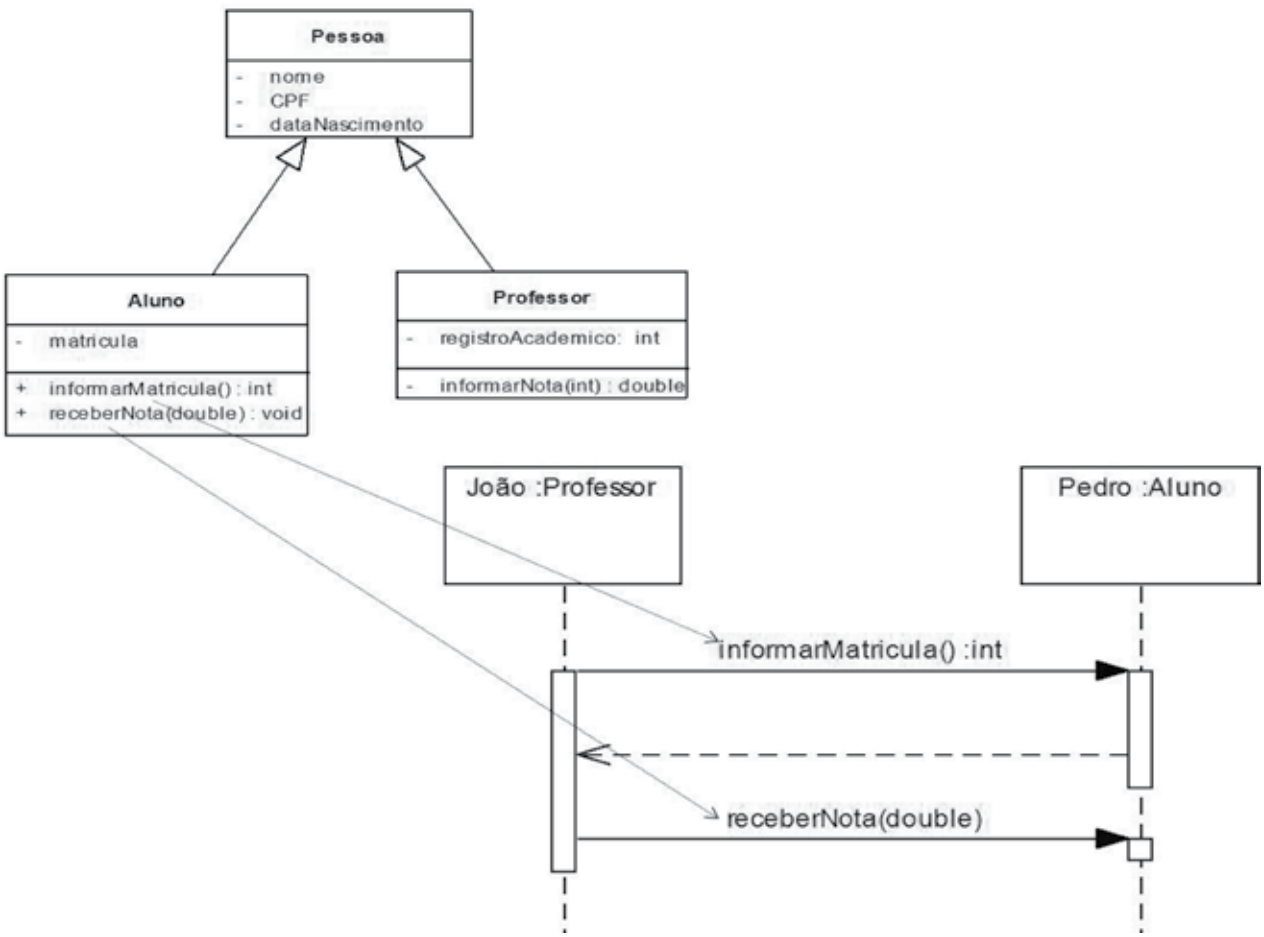

Figura 15 – Exemplo de troca de mensagens: chamada de função

É possível notar, nessa figura, que os métodos e alguns atributos das classes já possuem características do modelo de projeto, ou seja, tipos de dados e nível de visibilidade. Além do mais,

as mensagens do diagrama de sequência devem obrigatoriamente refletir os métodos e a assinatura descritos no diagrama de classes.

- Envio explícito de mensagem: utilizando um tipo de serviço bem específico e especializado para envio e recebimento de mensagens, por exemplo, Message Queue (fila de mensagens).

- Evento: quando uma mensagem é enviada para um objeto do sistema originária de um evento externo ao sistema. É comum que esse tipo de mensagem seja próprio da comunicação entre atores e objetos. Por exemplo: um clique de *mouse* pode ser considerado um evento externo ao sistema e que deve ser interpretado por um objeto interno do mesmo sistema.

Além do significado de uma mensagem, ao detalharmos os aspectos dinâmicos do sistema, devemos entender que tipo de mensagem estamos querendo representar, pois isso definirá como o comportamento do sistema será construído. Aqui começamos a tratar de questões importantes de um sistema, por exemplo, paralelismo.

Segundo o padrão de comunicação de interação de objetos, que pode ser observado em Stadzisz (2002), existem dois tipos de mensagem entre objetos:

- Mensagens síncronas: são mensagens que implicam um sincronismo rígido entre os estados do objeto que envia a mensagem e os do objeto de destino da mensagem.

  – Em um nível de projeto, podemos interpretar que este tipo de mensagem é utilizado quando o método do objeto chamado possui algum tipo de retorno no qual o objeto chamador espera.

- Mensagens assíncronas: não há dependência de estado do objeto chamador e do processamento do objeto chamado. O objeto chamador envia a mensagem e continua o seu processamento.

  – Em um nível de projeto, utilizamos esse tipo de mensagem quando queremos iniciar algum tipo de processamento paralelo entre dois objetos, ou seja, o objeto chamador pode continuar o seu processamento independentemente do retorno do objeto chamado ou dos processamentos iniciados pelo objeto chamado após o envio da mensagem.

Além de mensagens síncronas e assíncronas, Larman (2007) observa mais alguns tipos de mensagens:

- Autodelegação de mensagens: um objeto pode enviar uma mensagem para ele mesmo, solicitando a execução de um método. Esse movimento é chamado de autodelegação. O método que só pode ser chamado pelo próprio objeto que o contém, conforme vimos, é um método privado, e isso é muito importante levarmos em consideração no momento do projeto, pois isso garante o correto uso do encapsulamento.

- Criação e destruição de objetos: no ciclo de vida de um objeto, existem dois métodos, construtores e destrutores, que têm como objetivo criar e remover um objeto da estrutura de memória de um sistema. Em orientação a objetos, quando um objeto cria outro para utilizá-lo ou para enviar-lhe

uma mensagem, dizemos que o primeiro possui uma referência à instância do objeto criado e, na fase de projeto, é importante que criemos as referências apenas quando necessário, pois isso define a dependência entre objetos e é o início da coesão e do acoplamento.

A figura a seguir mostra os elementos de uma estrutura dinâmica que debatemos até agora, dispostos no diagrama de sequência da UML. O exemplo dessa figura mostra um processo fictício de fechamento de notas:

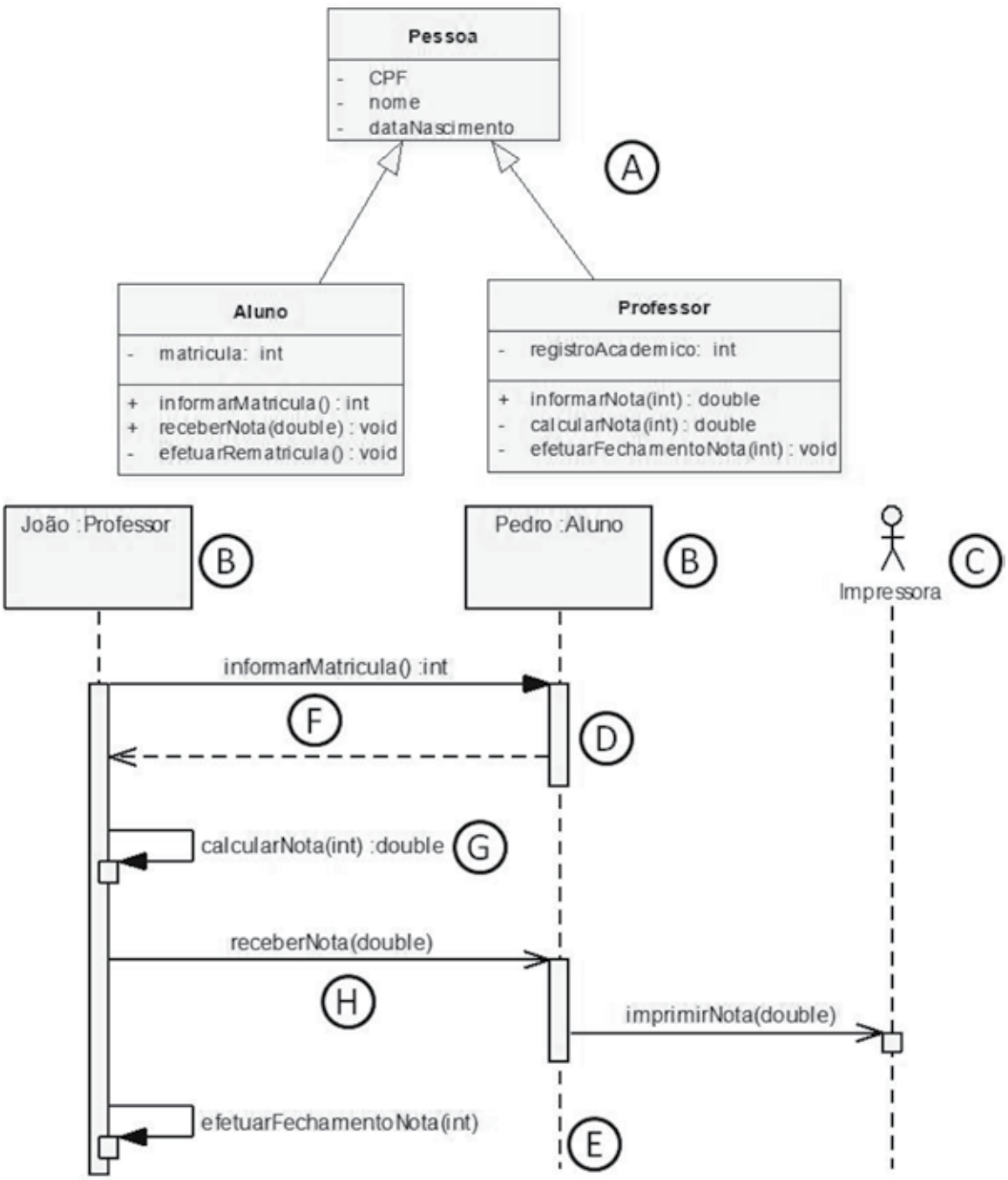

Figura 16 – Exemplo de modelagem de aspectos dinâmicos de um sistema

Quadro 6 – Exemplo de modelagem de aspectos dinâmicos de um sistema

Letra	Descrição
A	Diagrama de classes, com inclusão de tipos de dados nos atributos, assinatura completa nos métodos e visibilidade de atributos e métodos. Note que a forma pela qual os métodos estão especificados no diagrama de classes é a mesma do diagrama de sequência.
B	O retângulo representa um objeto que faz parte da execução do cenário. Note que estamos falando de objetos, e não de classes, pois quando estamos representando a troca de mensagens, no aspecto dinâmico do sistema, o objeto já está instanciado. Obrigatoriamente, os objetos do diagrama de sequência devem constar no modelo de classes.
C	Representa o ator, envolvido no contexto. Como vimos, um ator pode ser um elemento externo ao sistema, como um usuário, um dispositivo de *hardware* ou outro sistema.
D	O retângulo posicionado abaixo do objeto significa que naquele espaço de tempo iniciou-se o ciclo de vida do objeto. Como vimos, um objeto é criado, utilizado e destruído, e nesse espaço de tempo representado pelo retângulo o objeto está ativo, ou seja, está criado, e está apto a receber mensagens.
E	A linha tracejada abaixo dos atores e dos objetos representa a linha de tempo.
F	A seta com linha tracejada logo abaixo da seta com ponta cheia representa um retorno de uma mensagem, retorno este de uma mensagem síncrona representada pela seta com ponta cheia. Neste momento, o objeto do Professor está aguardando o retorno e o processamento do objeto Aluno.
G	A seta recursiva indica uma autodelegação de mensagem. É o objeto do tipo Professor enviando uma mensagem para ele mesmo a partir do método privado calcularNota.
H	A seta aberta indica o envio de uma mensagem assíncrona. Após o envio da mensagem receberNota do objeto do tipo Professor para o objeto do tipo Aluno, ambos podem executar outros processamentos, independentemente da resposta deste método. Que é o que efetivamente ocorre, o objeto do tipo Aluno envia uma mensagem (com significado de evento) para o ator Impressora, e o objeto do tipo Professor envia uma mensagem para ele mesmo: efetuarFechamento. É claro que todo processamento paralelo uma hora é sincronizado dentro do sistema, por isso a importância de modelarmos e especificarmos corretamente o que é síncrono e o que é assíncrono na fase de projeto (*design*).

Na UML, mais precisamente no diagrama de sequência, ainda podemos representar estruturas de decisão e repetição, o mesmo conceito que vimos em algoritmos básicos.

Saiba mais

Para aprofundar seus conhecimentos sobre estruturas de decisão e repetição, leia:

MARTINS, C. T. K.; RODRIGUES, M. *Algoritmos elementares C++*. São Paulo: LCTE, 2006.

A figura a seguir mostra um exemplo de estrutura de decisão. No diagrama de sequência, temos o elemento operador de controle, que pode ser observado no ponto indicado com a letra A. O indicativo "alt" no operador de controle significa fluxo alternativo, ou seja, só será executado se uma condição for satisfeita; no caso, a senha informada deve ser correta.

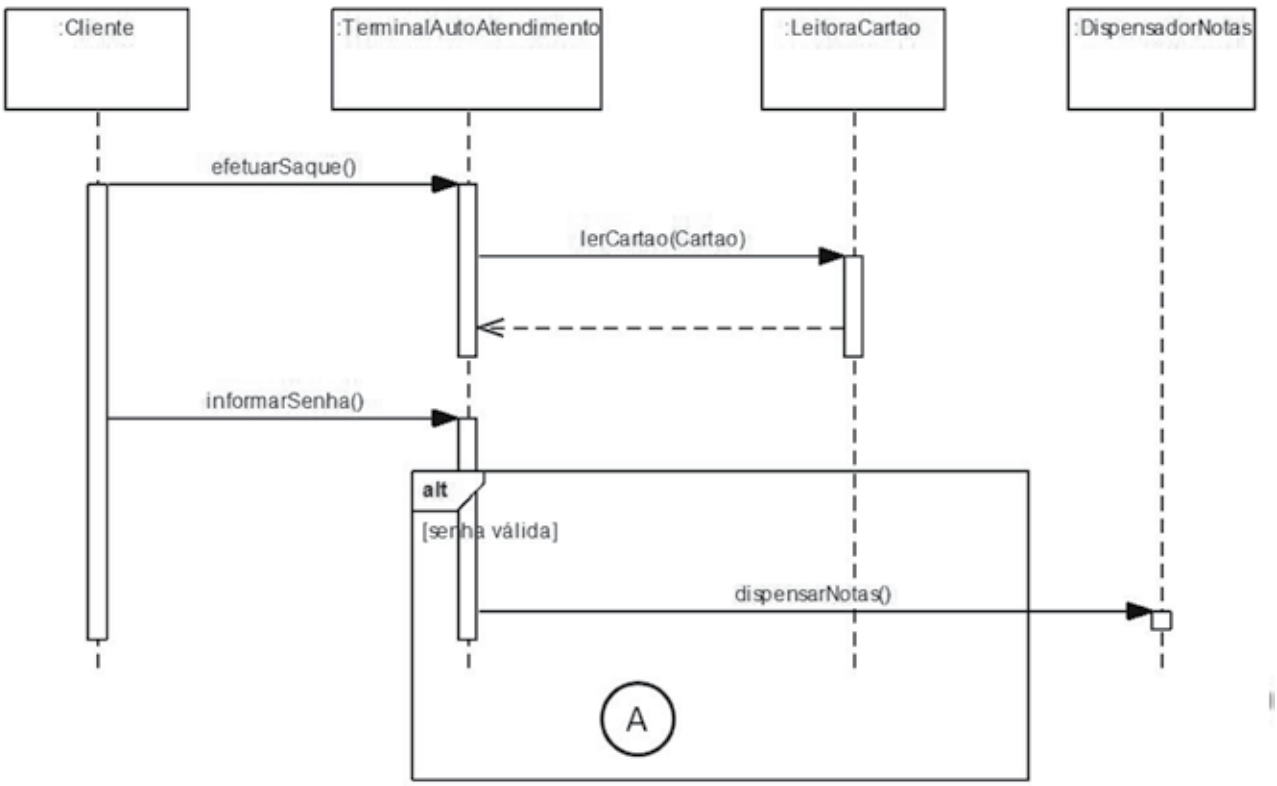

Figura 17 – Exemplo de fluxo alternativo em diagrama de sequência

A seguir, mostra-se um exemplo de laço de repetição. O indicativo *loop* no operador de controle, indicado pela letra A, sinaliza uma repetição: no caso, a mensagem informarSenha será enviada no mínimo uma vez e no máximo três vezes, conforme indicado no operador de controle.

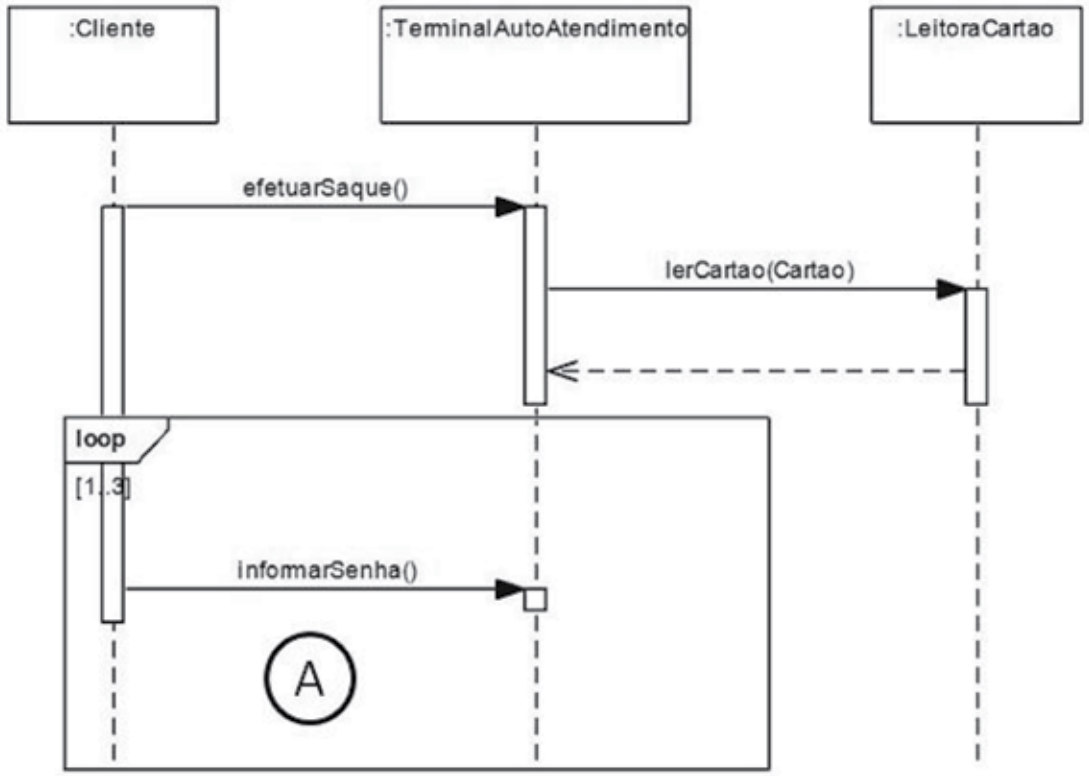

Figura 18 – Exemplo de estrutura de repetição em diagrama de sequência

Note a importância de se modelar corretamente as estruturas de decisão e de repetição; no momento em que o desenvolvedor analisar o modelo, saberá exatamente o que construir, diminuindo, e muito, problemas de entendimento ou até mesmo problemas de desempenho decorrentes de má prática de programação.

4.2.2 Refinamento dos aspectos estáticos e estruturais do sistema

O refinamento dos aspectos estáticos do sistema tem como objetivo promover a passagem do modelo de classes de domínio para o modelo de classes de projetos (BEZERRA, 2006).

O primeiro passo a ser dado é detalhar os atributos e os métodos, pois quando começamos a associar os tipos de dados e a complementar a assinatura dos métodos, alguns elementos podem ficar confusos, principalmente para o desenvolvedor.

Para evitar mal-entendido e efetivamente validar se o modelo a ser construído tem relação com o problema que pretende resolver, os atributos e métodos devem ser detalhados quanto ao seu uso e significado dentro do contexto do sistema.

O mesmo se aplica no detalhamento do relacionamento entre os objetos, que é igualmente necessário e deve ser feito na sequência, com o principal objetivo de detalhar os relacionamentos do tipo associação de objetos, uma vez que essas relações não são facilmente absorvidas em um primeiro momento.

Observação

Se um objeto envia uma mensagem para outro, eles possuem uma ligação, chamada de associação simples, ou seja, uma ligação que acontece apenas em um determinado momento.

O último passo para o refinamento é a atribuição de responsabilidades aos objetos da classe de domínio.

A atribuição de responsabilidades está relacionada aos atributos e métodos dessa classe. Devemos verificar se o objeto é suficientemente autocontido ou se, eventualmente, não possui mais responsabilidades do que deveria, o que causaria uma baixa coesão no universo sistêmico.

Uma classe do modelo de domínio pode gerar uma ou mais classes do modelo de projetos e vice-versa. Não existe uma regra para isso, tudo depende do cenário que estamos modelando.

A divisão de responsabilidades é uma das características fundamentais em uma boa modelagem de sistemas. Objetos com responsabilidades bem-definidas aumentam a sua capacidade de reúso.

Organizar e dividir os objetos por responsabilidade é a base para o conceito de padrões de projeto, que vem a ser um conjunto de soluções e organização sistêmica com um objetivo específico.

No caso, a divisão de responsabilidades pode ser encarada como um padrão de projeto com o objetivo de aumentar o reúso e diminuir o acoplamento entre objetos de um sistema. Esse conceito é a base para o padrão de projeto Model-View-Controller (MVC), que veremos com maiores detalhes.

Quando estamos definindo as responsabilidades dos objetos dentro do sistema, passamos a identificar as classes de entidade, de fronteira e de controle.

Entidade

Uma entidade, classe de entidade ou ainda objeto de entidade são objetos mais próximos do domínio do mundo real que o sistema representa, são abstrações do mundo real que normalmente conseguimos identificar nos casos de uso.

Esses objetos têm como objetivo principal manter informações referentes ao domínio de problema que queremos resolver.

Nas classes de entidade, representamos informações e comportamentos que são, de alguma forma, armazenados no sistema. Por exemplo: dentro do domínio do nosso problema de saque em terminal de autoatendimento, temos as classes de entidade **cliente**, **cartão** e **terminal de autoatendimento**.

Fronteira

Classes de fronteira ou objetos de fronteira, como o próprio nome diz, têm como responsabilidade dividir o ambiente interno do sistema e suas interações externas.

Podemos interpretar interações externas a um sistema como toda e qualquer comunicação que um sistema faz com atores do sistema ou ainda alimentar informações de outros sistemas. Por exemplo: dentro do domínio do nosso problema de saque, o sistema deve se comunicar com outro sistema, externo, responsável por efetuar a autenticação de segurança da senha de acesso do cliente.

Outros exemplos clássicos de interface externa: envio de informações para sistemas governamentais, como informações fiscais; consumo de informações de outros sistemas, como informações de crédito; interface com SGBD.

Lembrete

Na fase de análise do ciclo de vida da engenharia de *software*, na atividade de modelagem de casos de uso, atores de um sistema são agentes externos ao sistema, que executam determinada ação e que esperam algum resultado, podendo ser um usuário, um dispositivo de *hardware* ou outro sistema.

Controle

Classes de controle, objetos de controle ou ainda controladores são objetos que têm como objetivo realizar o sequenciamento da execução de um caso de uso na estrutura de objetos do sistema, bem como fazer a coordenação entre as camadas internas do sistema, representadas pelas classes de entidade, com as camadas externas ao sistema, representadas pelas classes de fronteira. Alguns autores também chamam esse movimento de orquestração.

4.2.3 Definição de outros aspectos da solução

A definição de outros aspectos da solução passa para um nível arquitetural do processo de passagem da fase de análise para a fase de projetos. Com o modelo de classes de projeto pronto, começamos a pensar em como organizar essas classes da melhor forma.

O primeiro passo a ser dado é a decomposição do sistema em subsistemas, ou também chamados de componentes; esse é o processo de componentização de um sistema de *software* (BEZERRA, 2006).

Analogamente, e também superficialmente, é como a indústria eletrônica já trabalha há muito tempo.

Cada componente eletrônico é um conjunto de CIs (circuitos integrados), e cada qual possui uma finalidade dentro do componente. Em um sistema eletrônico, apenas os componentes conversam entre si por interfaces de comunicação bem definidas, não existe comunicação entre os CIs. Desta forma, cada componente em um sistema eletrônico funciona como uma unidade autônoma nesse sistema.

A componentização na indústria eletrônica promove alguns benefícios, dentre eles: maior produtividade, pois é possível que equipes trabalhem em paralelo, uma vez que as interfaces de comunicação já estão definidas; e manutenibilidade, pois é possível que se substitua um componente do sistema (por falha ou evolução) sem que o todo seja desestruturado.

A indústria eletrônica está mais avançada que a de *software* nesse quesito, todavia o nível de maturidade das corporações e a especialização dos profissionais já mostram promissores fatores para a mudança deste cenário.

Saiba mais

O assunto componentização é extenso e deve ser aprofundado na forma de especialização. O tema é muito valorizado e tem grande mercado de atuação; referências no assunto:

HEINEMAN G. T.; COUNCILL, W. T. *Component-based software engineering*: putting the pieces together. Boston: Addison-Wesley Longman Publishing, 2001.

SZYPERSKI, C. *Component software*: beyond object-oriented programming. 2. ed. Boston: Addison-Wesley Longman Publishing, 2002.

Componentes definidos, devemos especificar e modelar como distribuiremos esses componentes nos recursos de *hardware* que possuímos.

Atualmente, muitos são os recursos disponíveis para distribuição do *software*, por exemplo, sistemas distribuídos em rede, serviços *web* (ou *Web* Services), *cloud computing* etc. Trataremos com maiores detalhes sobre como representamos essa distribuição, nas próximas unidades.

Ainda sobre a definição de outros aspectos da solução, devemos definir como os objetos serão persistidos em um repositório de informações.

O ato de definir como e quais objetos serão persistidos em um repositório tem grande relação com o modelo de dados adotado para o SGBD, ou seja, qualquer que seja o modelo adotado, E-R ou orientado a objetos, deve-se refletir e planejar sobre isso neste momento. Trataremos esse assunto com maior aprofundamento nas próximas unidades.

Com relação a essas três atividades, da passagem da análise ao projeto, é importante que tenhamos em mente que elas não são executadas de forma linear em um único sentido, ou seja, todos os modelos produzidos em cada atividade poderão, e deverão, ser refinados sempre que houver necessidade.

Isso quer dizer que um modelo de classes de projeto poderá e deverá ser alterado se, no momento da modelagem dos aspectos dinâmicos, for encontrada alguma anomalia, assim como o próprio modelo de análise, se preciso for. Isso é extremamente importante nesta fase do ciclo de vida do projeto.

Resumo

Nesta unidade tratamos de dois temas centrais: a tecnologia de apoio ao projeto orientado a objetos e a transição da fase de análise para a fase de projeto.

Sobre o tema da tecnologia de apoio ao projeto orientado a objetos, vimos que boa parte desse suporte é baseado na UML.

Embora muitos ainda confundam algumas coisas, vimos que a UML não é uma tecnologia, nem um *software*, nem uma ferramenta ou uma linguagem de desenvolvimento.

A UML é uma linguagem de modelagem unificada, utilizada para representar as diversas visões de um *software* durante o seu ciclo de vida de desenvolvimento.

Um *software*, segundo Kruchten (1995) e Booch, Jacobson e Rumbaugh (2006), possui cinco visões: visão lógica, visão de processo, visão de implementação, visão de implantação e visão de caso de uso, sendo esta última a central, ou seja, a que fornece subsídios para todas as visões.

Cada visão possui uma finalidade bem-definida, logo cada uma possui um conjunto de diagramas da UML específicos para cada uma dessas finalidades. Antes de utilizarmos a UML a esmo, precisamos ter a noção de qual visão estamos querendo modelar e de quais os diagramas corretos para cada finalidade.

Para desenharmos os diagramas da UML, utilizamos como facilitadores as ferramentas de modelagem, todavia vimos que a adoção e o uso consciente de uma ferramenta de mercado são fatores fundamentais para o sucesso nessa fase do projeto.

Outra tecnologia de apoio ao projeto são as chamadas ferramentas CASE, que têm como objetivo apoiar todo o ciclo de vida da engenharia de *software*, e não somente uma fase específica.

Todavia, a simples adoção de uma ferramenta CASE também não é garantia de qualidade e sucesso no projeto. As ferramentas CASE utilizadas no projeto devem estar interligadas e, principalmente, devem ter um modelo de processo de desenvolvimento sólido e maduro como suporte.

As tecnologias de apoio OO podem ser classificadas como *back-end* e *front-end*.

Tecnologias *back-end* basicamente são as tecnologias relacionadas ao armazenamento e gerenciamento de informações; são os chamados SGDBs, ou Sistemas Gerenciadores de Bancos de Dados.

Debatemos que, em se tratando de banco de dados, dois são os principais modelos – o modelo E-R e o modelo orientado a objetos – e que, enquanto o modelo entidade-relacionamento é utilizado largamente, o orientado a objetos ainda é muito incipiente tecnologicamente, mas possui um grande campo de expansão pela frente; basta definir ainda algumas questões, principalmente as relacionadas à padronização.

Tecnologias *front-end* são as ferramentas de modelagem e as linguagens de programação orientada a objetos.

Para uma linguagem ser considerada orientada a objetos, ela deve seguir algumas predefinições, que são possibilitar: o encapsulamento de dados, a abstração de dados, a coesão dinâmica e a herança. Linguagens puramente orientadas a objetos, como vimos em Lee e Tapfenhart (2001), não tiveram grande apelo no desenvolvimento de sistemas comerciais e, para tal, foram estendidas para utilizar técnicas não orientadas a objeto apenas quando apropriado.

As linguagens OO mais utilizadas na atualidade são Java e as linguagens da plataforma Microsoft .Net, em especial, a linguagem C#. Debatemos que, na discussão sobre qual das duas é melhor, raramente se levam em consideração os números. Como mostra a pesquisa do Instituto Gartner (DRIVER, 2007), as discussões partem sempre para o lado passional e acabam se baseando em interesses particulares.

A segunda frente desta unidade tratou do tema da passagem da fase de análise para a fase de projeto, cujo objetivo central é refinar o modelo de

domínio produzido na análise para obter o modelo de projeto, um modelo que possa ser implementável.

A fase de projeto, ou *design*, começa a partir do momento em que fechamos a primeira versão do modelo de classe de domínio; nesse momento, temos a transição análise-projeto.

São três perspectivas do modelo de classes: modelo conceitual, produzido na fase de análise; modelo de projeto, produzido na fase de projeto; e modelo de implementação, que corresponde ao modelo de projeto construído em uma determinada linguagem.

A diferença entre o modelo conceitual e o de projeto está na adição, no modelo de projeto, dos tipos de dados, assinatura dos métodos, detalhamento dos atributos e métodos e inclusão de novos objetos inerentes à solução técnica com o objetivo de que este modelo possa ser implementável, ou seja, construído.

Segundo Bezerra (2006), são três as atividades para a passagem da análise para o projeto: detalhamento dos aspectos dinâmicos do sistema, refinamento dos aspectos estáticos e estruturais do sistema e definição de outros aspectos da solução.

Na fase de detalhamento dos aspectos dinâmicos do sistema nós damos ênfase à representação da troca de mensagens entre os objetos com o objetivo de resolver um determinado problema.

No refinamento dos aspectos estáticos e estruturais, a ênfase é dada a refinar o modelo conceitual de tal forma que esse novo modelo possa ser implementável a partir da adição dos tipos de dados, assinatura dos métodos, detalhamento dos atributos e métodos e inclusão de novos objetos, se necessário, e neste ponto vimos que uma classe do modelo de classes de domínio pode gerar uma ou mais classes no modelo de classes de projeto e vice-versa, e isso é muito importante ter em mente.

Por último, na definição de outros aspectos da solução, demos ênfase a: arquitetura, componentização, distribuição dos componentes nos recursos de *hardware* e mapeamento dos objetos em um SGBD.

Cada fase possui a sua sequência de atividades, no entanto é importante que tenhamos em mente que sempre podemos, e devemos, refinar e corrigir modelos produzidos nas fases anteriores quando necessário.

Exercícios

Questão 1. A UML (*Unified Modeling Language*), na definição de Booch *et al*. (2006, p. 13), "é uma linguagem-padrão para elaboração da estrutura de projetos de *software* [...] adequada para a modelagem de sistemas". Com relação a UML, analise as duas afirmativas apresentadas a seguir:

Existem inúmeras ferramentas no mercado que se utilizam da UML para a geração de diagramas. A própria UML em si pode ser considerada uma plataforma ou uma ferramenta de modelagem.

Porque

A UML não é uma linguagem de programação, muito embora seja possível a geração de código a partir de alguns diagramas, o diagrama de classes principalmente, assim como o inverso, ou seja, a geração de diagramas a partir de código fonte.

Acerca dessas afirmativas, assinale a opção correta:

A) As duas afirmativas são proposições verdadeiras e a segunda é uma justificativa correta da primeira.

B) As duas afirmativas são proposições verdadeiras e a segunda não é uma justificativa correta da primeira.

C) A primeira afirmativa é uma proposição verdadeira e a segunda é uma proposição falsa.

D) A primeira afirmativa é uma proposição falsa e a segunda é uma proposição verdadeira.

E) As duas afirmativas são proposições falsas.

Resposta correta: alternativa D.

Análise das alternativas

Justificativa geral: a primeira afirmativa é uma proposição falsa e a segunda é uma proposição verdadeira.

A versão correta da primeira afirmativa é:

Existem inúmeras ferramentas no mercado que se utilizam da UML para a geração de diagramas, mas a UML em si não pode ser considerada uma plataforma ou uma ferramenta de modelagem tampouco um *software*.

Como apresentado no livro-texto, a confusão nesse ponto se dá pela falsa sensação de que não conseguimos trabalhar com UML e desenvolver diagramas sem que tenhamos uma ferramenta de modelagem.

Questão 2. Com relação à modelagem de Sistemas Gerenciadores de Banco de Dados, alguns pontos do modelo orientado a objetos marcam sua diferença conceitual para o modelo Entidade-Relacionamento.

Com base nas diferenças do modelo orientado a objetos avalie as afirmativas a seguir:

I – Cada objeto possui um conjunto de códigos que operam sobre este objeto, chamado de método.

II – Cada objeto possui uma identidade única independente dos valores contidos em seus atributos, ou seja, mesmo que um objeto possua os mesmos valores dos atributos de outro objeto no mesmo banco de dados, estes possuem identidades diferentes.

III – A presença de atributos.

IV – Adoção de mecanismos de relacionamento: composição, agregação e herança.

É correto apenas o que apenas se afirma em:

A) I e II.

B) II e IV.

C) I, II, III e IV.

D) I, II, e IV.

E) I e III.

Resolução desta questão na plataforma.