SERGIO MARTINS FERNANDES

ESPECIALIZAÇÃO DA DISCIPLINA ANÁLISE E PROJETO DO RATIONAL UNIFIED PROCESS (RUP) PARA A ARQUITETURA JAVA 2 PLATFORM, ENTERPRISE EDITION (J2EE)

Dissertação apresentada à Escola Politécnica da Universidade de São Paulo para obtenção do Título de Mestre em Engenharia

São Paulo
2002

SERGIO MARTINS FERNANDES

ESPECIALIZAÇÃO DA DISCIPLINA ANÁLISE E PROJETO DO RATIONAL UNIFIED PROCESS (RUP) PARA A ARQUITETURA JAVA 2 PLATFORM, ENTERPRISE EDITION (J2EE)

Dissertação apresentada à Escola Politécnica da Universidade de São Paulo para obtenção do Título de Mestre em Engenharia

Área de Concentração:
Sistemas Digitais

Orientador:
Profa. Livre Docente
Selma Shin Shimizu Melnikoff

São Paulo
2002

UNIVERSIDADE DE SÃO PAULO
ESCOLA POLITÉCNICA
TERMO DE JULGAMENTO
DE
DEFESA DE DISSERTAÇÃO DE MESTRADO

Aos 13 dias do mês de setembro de 2002, às 16:00 horas, no Departamento de Engenharia de Computação e Sistemas Digitais da Escola Politécnica da Universidade de São Paulo, presente a Comissão Julgadora, integrada pelos Senhores Professores Doutores Selma Shin Shimizu Melnikoff, Orientador(a) do(a) candidato(a), Jorge Luis Risco Becerra e Mauro de Mesquita Spínola, iniciou-se a Defesa de Dissertação de Mestrado do(a) Sr(a). **SÉRGIO MARTINS FERNANDES**

Título da Dissertação: **"ESPECIALIZAÇÃO DA DISCIPLINA ANÁLISE E PROJETO DO RATIONAL UNIFIED PROCESS (RUP) PARA A ARQUITETURA JAVA 2 PLATFORM, ENTERPRISE EDITION (J2EE)".**

Concluída a argüição, procedeu-se ao julgamento na forma regulamentar, tendo a Comissão Julgadora considerado o(a) candidato(a):

Profª.Drª. Selma Shin Shimizu Melnikoff....(Aprovada)
Prof.Dr. Jorge Luis Risco Becerra.....(Aprovado)
Prof.Dr. Mauro de Mesquita Spínola.....(Aprovado)

Para constar, é lavrado o presente termo, que vai assinado pela Comissão Julgadora e pela Secretária de Pós-Graduação.

São Paulo, 13 de setembro de 2002.
A COMISSÃO JULGADORA

Secretária: Elisabete Apª F.S.Ramos

Obs: Resolução 4476, de 17.09.1997/Altera dispositivos do Regimento Geral da USP Art.109 - Imediatamente após o encerramento da argüição da dissertação ou da tese cada examinador expressará seu julgamento em sessão secreta, considerando o candidato **aprovado** ou **reprovado**.

Homologado pela C.P.G. em reunião realizada 23/09/02.

Para minha mãe, Formosa, por todo o estímulo.

Para meu pai, Irisnaldo, pelas oportunidades.

Para minha irmã, Sandra, pela proteção.

AGRADECIMENTO

Agradeço à minha querida orientadora, Selma Melnikoff, por todo o apoio e amizade ao longo desses anos.

ABSTRACT

The purpose of this work is to provide some specific analysis and design guidelines for J2EE application developers, by specializing some elements of the Rational Unified Process's Analysis and Design discipline to the J2EE environment. It aims to help reducing the semantic gap between the abstractions and services that J2EE provides and the final design that the team must build, and, generally, to promote analysis and design software engineering best practices.

This work is based on and integrates knowledge that is part of:

- Rational Unified Process (RUP), the very complete and detailed software engineering process developed by Rational Software. Particularly, this work uses RUP's Analysis and Design discipline and software architecture concepts;
- Architectural and design patterns, since very basic patterns like the GRASP patterns proposed by Larman and the classic GoF patterns, to Buschmann's architecture patterns and, specifically, the J2EE design patterns from Sun;
- The J2EE specification.

RUP's Analysis and Design discipline is detailed investigated, looking for activities which can be specialized for the J2EE environment. Based on these activities definitions, some specific guidelines and architectural mechanisms (a RUP concept) are developed, focusing the J2EE architecture.

The work comprises a case study in which some key RUP artifacts from Business Modeling and Requirements Management as the background for the development of a partial design of a J2EE application, using the guidelines previously developed.

RESUMO

O objetivo deste trabalho é prover diretrizes de análise e projeto para equipes que desenvolvam aplicações J2EE, com base na especialização para J2EE de alguns elementos da disciplina Análise e Projeto do Rational Unified Process (RUP). Desta forma, visa-se contribuir para reduzir o gap semântico entre as abstrações e serviços que a plataforma J2EE provê e o projeto final de uma aplicação J2EE. Mais genericamente, o objetivo é promover a utilização de boas práticas de engenharia de software voltadas para análise e projeto.

O trabalho se baseia e integra conhecimento dos seguintes elementos:

- O RUP, um processo de software muito completo e efetivo produzido pela Rational Software. Particularmente, a disciplina Análise e Projeto do RUP e os conceitos de Arquitetura de Software definidos no RUP são focados;
- Patterns de arquitetura e projeto, desde os muito básicos como patterns GRASP os propostos por Larman e os famosos patterns GoF, passando por patterns de arquitetura propostos por Buschmann, até os patterns de projeto J2EE criados por equipes da Sun Microsystems;
- A especificação da arquitetura J2EE.

A disciplina Análise e Projeto do RUP é investigada, com o intuito de identificar as atividades nas quais a especialização para J2EE agregaria valor. Essa especialização consiste na definição de diretrizes específicas para o ambiente J2EE, com base em patterns, e na definição de mecanismos de arquitetura (um conceito do RUP) específicos para J2EE.

É desenvolvido um estudo de caso, o projeto parcial de uma aplicação fictícia, como forma de ilustrar as propostas apresentadas.

ERRATA

Página 18: no primeiro parágrafo do item 1.6.1, substituir "envolveuma" por "envolve uma".

Página 18: no penúltimo parágrafo, substituir '"todas as" por "a maior parte das".

Página 18: no penúltimo parágrafo, substituir "[referência]" por "[RATIONAL 2000]".

Página 27: no último parágrafo, substituir "especialiação" por "especialização".

Página 44: no primeiro parágrafo, substituir "Javabenas" por "Javabeans".

Página 44: no segundo parágrafo, substituir "simplemente" por "simplesmente".

Página 51: no último parágrafo, substituir "performqnce" por "performance".

Página 61: na primeira linha, substituir "O RUP considera necessário" por "É uma premissa da definição do processo".

Página 65: no último item da lista que antecede o título "Visão – Defina uma Visão Clara", substituir "Proceso" por "Processo".

Página 73: na última linha, substituir "paapéis" por "papéis".

Página 75: no penúltimo parágrafo, substituir "concisae" por "concisa e".

Página 87: no primeiro parágrafo, substituir "***" por "3.6".

Página 94: no segundo parágrafo, substituir "àengenharia" por "à engenharia".

Página 104: na quarta linha, substituir "encapsuladoem" por "encapsulado em".

Página 115: na quinta linha, substituir "apresntada" por "apresentada".

Página 119: na sexta linha, substituir "Serlvets" por "Servlets".

Página 108: na sexta linha, substituir "responsbilidade por por" por "responsabilidade por".

Página 124: na primeira linha do segundo parágrafo, substituir "O RUP não organiza a disciplina Análise e Projeto" por "A disciplina Análise e Projeto do RUP não está organizada".

Página 140: no quarto parágrafo, substituir "persistentese" por "persistentes e".

Página 145: na primeira linha, substituir "Boundary" por "Fronteira".

Página 145: no subtítulo após o tópico 4.9.3, substituir "Boundary" por "Fronteira".

Página 145: no terceiro parágrafo, substituir "Boundary" por "Fronteira", tanto na segunda quanto na última linhas.

Página 145: no penúltimo parágrafo, substituir "anterormente" por "anteriormente".

Página 146: na primeira linha, substituir "boundary" por "fronteira".

Página 146: na terceira linha, substituir "poisele" por "pois ele".

Página 146: no quarto parágrafo, substituir "boundary" por "fronteira".

Página 146, no penúltimo parágrafo, substituir "boundary" por "fronteira".

Página 148: na primeira linha, substituir "Serlvet" por "Servlet".

Página 149: na última linha, substituir "granuralidade" por "granularidade".

Página 169: no quarto parágrafo, substituir "idenficados" por "identificados".

Página 179: na primeira linha, última célula da tabela, substituir "cliclo" por "ciclo".

Página 181: no primeiro parágrafo, substituir "infra-estutura" por "infra-estrutura".

Página 184: no último parágrafo, substituir "Unigen" por "Genetec".

Página 185: no segundo parágrafo, substituir "Unigen" por "Genetec".

Página 193: na primeira linha, substituir "apresentaremo" por "apresentaremos".

Página 198: no penúltimo parágrafo, substituir "SolcicitacoesFacade" por "SolicitacoesFacade".

Página 214: no penúltimo parágrafo, substituir "enfazitado" por "enfatizado".

Página 218: incluir a referência: [RATIONAL 2000] Reaching CMM Levels 2 and 3 with the Rational Unified Process. http://www.rational.com/products/whitepapers/100416.jsp

SUMÁRIO

1. Introdução

1.1 *Breve Perspectiva histórica*

No início dos anos 90, provavelmente não pela primeira vez, falou-se muito que a função de análise de sistemas tenderia a desaparecer com o tempo, ou pelo menos a reduzir o escopo de atividades a determinado tipo de aplicações. O advento de ambientes de desenvolvimento denominados de "quarta geração" permitiria que o próprio usuário final caracterizasse, em linguagem natural, as suas necessidades de informações, sem necessitar de intermediários para definir abstrações, modelar e implementar software que atendesse a essas necessidades.

Em retrospectiva, é óbvio que a suposta tendência não se confirmou. Entre os motivos do não cumprimento dessa previsão, interessa enfatizar que o ambiente de desenvolvimento e de implantação de software se tornou muito mais poderoso, permitindo automatizar e mesmo conceber processos de negócio de forma inimaginável há dez anos atrás. Mas isso ocorreu ao custo de uma complexidade muito maior, o que se tornou particularmente verdade no momento em que as aplicações da web deixaram de apenas disponibilizar informações, e passaram a interagir com o usuário e a mapear processos inteiros de negócio, por exemplo, em sistemas de comércio eletrônico (*e-commerce*).

Se antes era fundamental conhecer bem uma linguagem de programação e recursos de acesso a gerenciadores de bancos de dados, para implementar aplicativos, agora esse é o menor dos desafios. Surgiram novos problemas e novas soluções para lidar com questões que se tornaram críticas, como segurança, escalabilidade, acesso às bases de dados e integração de aplicações com o legado.

Muitas das aplicações têm não milhares, mas milhões de usuários potenciais. O desenvolvimento de aplicações hoje é um tema multidisciplinar, que envolve necessidade de conhecimento de diversos ambientes e tecnologias e formas adequadas para fazê-los interagir.

Tudo isso exigiu esforços tanto dos projetistas de linguagens de programação e ambientes de desenvolvimento, como dos próprios projetistas de aplicativos de

software, estes tendo que projetar aplicações que lidem adequadamente com as necessidades demandadas pelo ambiente.

Nesse cenário, ocorreu o contrário do previsto no início dos anos 90. O número de analistas de sistemas e programadores cresceu acentuadamente, e as exigências referentes àqueles que desempenham essas funções cresceram talvez em maior proporção.

1.2 *Contexto*

O *Standish Group* – um instituto americano especializado em consultoria na área gerencial, com foco em gerência de projetos de software, realiza a cada dois anos um amplo estudo em centenas de empresas americanas de médio e grande porte (descrito num relatório curiosamente denominado CHAOS *Report*) visando caracterizar o grau de sucesso de projetos de desenvolvimento/manutenção de software. O critério de sucesso é a entrega de produtos aceitos pelos clientes no prazo e respeitando o orçamento previsto. Essa pesquisa começou a ser efetuada em 1994, e tem demonstrado taxas de sucesso crescentes em projetos de software. Ainda assim, a norma tem sido projetos que não cumprem prazo e/ou orçamento ou que simplesmente fracassam (o CHAOS Report de 1999 [STANDISH 1999] apontou um índice de sucesso pleno de apenas 26% dos projetos) e o custo dos projetos que fracassam tem aumentado.

Não é necessário, porém, o acesso a um relatório detalhado para identificar os problemas que continuam a afligir a indústria de software, em escala global. Basta interagir com as áreas de desenvolvimento de software na maioria das empresas, mesmo nos Estados Unidos. Ainda que a adesão das organizações a processos de engenharia de software tenha se intensificado nos últimos anos, essas organizações ainda constituem uma minoria. Os problemas continuam a existir, como provam os CHAOS *Reports*, e têm sido agravados pelo aumento da complexidade do desenvolvimento de software, citado anteriormente.

É claro que essas questões não se colocam em relação a qualquer tipo de software. A indústria de software abrange hoje quase todas as áreas de atuação humana. Este texto não pretende referenciar todas essas áreas. O foco desta dissertação é o desenvolvimento de aplicações empresarias que automatizem total ou parcialmente

processos de negócio. Nesse contexto, o ambiente principal de desenvolvimento hoje é a web (intranets, Internet, extranets).

1.3 *Plataforma de Desenvolvimento*

Do ponto de vista de tecnologia de implementação e execução de aplicações, a resposta ao desafio da complexidade do software empresarial web são as plataformas de desenvolvimento definidas a partir do final da década de 90: a plataforma .Net (diz-se dot Net), da Microsoft Corporation; e a plataforma Java 2 Platform, Enterprise Edition (J2EE), desenvolvida por um consórcio liderado pela Sun Microsystems. Ambas focam o desenvolvimento de aplicações componentizadas em ambiente distribuído, e demandam um esforço razoável de aprendizado, conceitual e prático.

Uma plataforma de desenvolvimento contempla uma infra-estrutura de serviços disponíveis às aplicações nela hospedadas e um ambiente de desenvolvimento e de execução. Esse ambiente abrange linguagens de programação e também uma arquitetura para estruturação, implementação e execução das aplicações em produção.

Dentre as duas plataformas disponíveis, esta dissertação foca a plataforma J2EE, primariamente porque os elementos que a constituem conduzem a uma arquitetura de sofware para as aplicações nela hospedadas que se adequa aos objetivos deste texto. As razões desta escolha são abordadas em maior detalhe neste capítulo.

A plataforma J2EE é uma arquitetura criada pela Sun Microsystems para desenvolver, implantar e executar aplicações baseadas em componentes, em ambiente distribuído. O termo *enterprise*, que consta no nome da plataforma, deseja caracterizar a natureza e, implicitamente, o grau de complexidade das aplicações desenvolvidas sob essa plataforma. Aplicações dessa natureza demandam serviços, como gerência de transações, segurança, acesso a bases de dados, entre outros. Uma das características da plataforma J2EE é prover, ela própria, esses serviços, de modo que as equipes de desenvolvimento possam se focar na lógica de negócio das aplicações e não nos serviços de infra-estrutura.

1.4 *Processo de Software*

Conhecer a tecnologia de implementação e execução das aplicações, entretanto, não é suficiente para garantir o sucesso de projetos de software. A efetiva utilização de um processo de engenharia de software bem definido, se sempre foi uma necessidade importante (mas, paradoxalmente, pouco enfatizada no mundo empresarial), ganha relevo diante desse ambiente complexo. Essa também é uma questão ampla, e dela destacamos dois aspectos, respectivamente, gerenciais e técnicos:

- Diz-se que todos os projetos simples já foram executados. Sob o aspecto gerencial, há a demanda por ciclos de desenvolvimento muito mais curtos, contrapondo-se à necessidade de projetar adequadamente aplicações que tenham um ciclo de vida mais longo, e que possam evoluir de forma não traumática. Seguir um **processo** que aborde adequadamente os aspectos gerenciais relacionados a projetos de software ganha relevância nesse contexto;
- Embora os paradigmas de orientação a objetos, componentização e aplicações distribuídas sejam evoluções bem vindas ao universo do software, para que possam efetivamente prover os benefícios a que se propõem, demandam grande habilidade de projeto dos aplicativos. Dispor de um **processo** que oriente e facilite o projeto das aplicações, considerando esses paradigmas, é fundamental, até porque as linguagens e ambientes de programação mais recentes praticamente os impõem.

Dentre os processos de engenharia de software que ganharam relevância nos anos 90, destaca-se o Rational Unified Process (RUP), desenvolvido e comercializado pela Rational Software, e que tem uma versão simplificada e não comercial, denominada Unified Process [JACOBSON 1999]. O RUP evoluiu principalmente a partir do processo Objectory (comercializado pela Objective Systems, empresa sueca adquirida pela Rational), e de sua versão não comercial, denominada Object Oriented Software Engineering, de Ivar Jacobson [JACOBSON 1992]; do Object Modeling Technique (OMT), de James Rumbaugh [RUMBAUGH 1991]; e do método de Grady Booch [BOOCH 1994], tendo incorporado ainda influências de diversos outros métodos.

Kruchten define o Rational Unified Process (RUP) como uma abordagem metódica para atribuir e gerenciar tarefas e responsabilidades em uma organização de desenvolvimento. O objetivo do processo é produzir, em cronograma e orçamento previsíveis, software de alta qualidade que atenda às necessidades dos usuários. [KRUCHTEN 2000].

O RUP é um *framework* de processo, e deve ser especializado para cada Organização e/ou para cada projeto no qual será utilizado. Uma das disciplinas do RUP, denominada Ambiente, contempla as diretrizes e as estratégias para essa especialização, que implica essencialmente em selecionar / alterar / adicionar os elementos do processo (atividades, papéis, artefatos, diretrizes) que se deseja utilizar numa certa organização ou projeto.

O RUP 2002 incorporou um recurso denominado *plugin*. Cada *plugin* se constitui numa especialização do processo a um certo ambiente ou objetivo. Há *plugins* fornecidos pela Rational Software, mas os usuários do RUP podem desenvolver os seus próprios *plugins*. Há *plugins* para a plataforma .Net da Microsoft, para a plataforma J2EE, para ambientes de determinados servidores de aplicações, como BEA Weblogic e IBM WebSphere, dentre outros. Estes últimos vêm associados ao *plugin* mais genérico da plataforma J2EE, da qual esses servidores de aplicação são implementações. Esses *plugins* se constituem numa especialização do RUP. Em maior ou menor grau, adaptam determinados aspectos do RUP a um ambiente específico.

1.5 *Objetivo do Trabalho*

Os dois tópicos anteriores foram apresentados com o objetivo de caracterizar dois pré-requisitos fundamentais para o sucesso de projetos de software empresarial:

- O balizamento do projeto por um processo de engenharia de software adequado;
- A adoção de uma plataforma de desenvolvimento adequada, em relação à qual haja amplo conhecimento.

Existe ainda um terceiro fator, derivado destes citados anteriormente: há muitos pontos de contato entre o processo e a plataforma e, para reduzir a complexidade do projeto de aplicações nesse ambiente, é útil especializar o processo de software, de

modo que este possa melhor contribuir para o desenvolvimento na plataforma selecionada.

Esta questão é relevante, porque é possível conhecer os recursos tecnológicos de uma plataforma de desenvolvimento e, ainda assim, projetar software que não atende aos requisitos. Neste contexto, incluem-se também os requisitos não funcionais, tais como desempenho, escalabilidade, robustez. Para evitar esse problema, é necessário uma aplicação adequada da engenharia de software, considerando a arquitetura e os recursos da plataforma selecionada.

No entanto, um processo, pela sua própria característica é genérico, não sendo voltado a uma plataforma específica, pois as plataformas tecnológicas estão muito mais sujeitas às mudanças de tecnologia e do mercado do que um processo de software.

A solução desta questão é a especialização que um processo genérico deve sofrer, para atender a uma situação específica. O RUP, como foi citado, foi concebido com a premissa de ser especializado para cada Organização em que seja implantado e em cada projeto em que seja utilizado.

Então, o processo de software pode (e deve) contemplar as diretrizes e os artefatos especializados para o desenvolvimento de software em uma determinada plataforma de desenvolvimento.

O objetivo desta dissertação foca este aspecto e consiste da especialização de um processo de software (RUP) a uma particular plataforma de desenvolvimento (J2EE).

O ponto de partida desta especialização é o RUP 2002 com o *plugin* J2EE já instalado, ou seja, considera um primeiro nível de especialização do RUP para a plataforma J2EE, efetuada pela própria Rational. A maior contribuição do *plugin* é a definição de um tópico denominado *roadmap*, que descreve como conduzir as fases e atividade do processo para projetos J2EE. No entanto, o *plugin* não aborda a especialização de elementos chave da disciplina Análise e Projeto para a plataforma J2EE, que vai ser desenvolvido neste trabalho.

Este trabalho define diretrizes para efetuar análise e projeto de aplicações J2EE, focando os seguintes aspectos, sob o ponto de vista do RUP:

- Arquitetura de software;
- Utilização de *patterns* de arquitetura e projeto;
- Mecanismos de análise e projeto;
- Diretrizes para projeto a partir da análise.

O uso do processo e as questões relevantes identificadas durante a especialização são ilustradas, de forma prática, através de um estudo de caso que apresenta o projeto de uma aplicação J2EE utilizando as diretrizes definidas.

1.6 *Justificativa*

1.6.1 Por que selecionar o RUP

O RUP é fruto de um esforço coletivo para produção de um processo de software efetivo. Os mesmos metodologistas – Ivar Jacobson, Grady Booch e James Rumbaugh – que estiveram à frente do projeto UML, também inspiraram o projeto do processo unificado. Entretanto, o RUP envolveuma equipe muito maior, coordenada por Philippe Kruchten, além destas três pessoas.

O RUP está a um passo adiante dos processos de software que o antecederam, pela abrangência e profundidade da proposta, e pela sua natureza de processo que evolui gerando seguidas versões, como um produto comercial.

O RUP não é um método para desenvolvimento de software, mas um processo de engenharia de software, para ser implantado em escala industrial. Foca desde questões técnicas mais básicas, como a definição de classes e a interação entre objetos, até questões mais estratégicas, como a definição de uma arquitetura de software componentizada e em camadas.

Com a mesma ênfase, aborda questões gerenciais, como gerência de projetos e gerência de mudança e configuração. No aspecto gerencial, o RUP contempla todas as Áreas Chave de Processo previstas pelo CMM-SW (*Capability Maturity Model for Software*) nos níveis de maturidade 2 e 3 [referenciar].

Em uma primeira abordagem, o RUP pode parecer complexo e até mesmo excessivo, já que define um sem número de artefatos associados a cada disciplina que o

compõe. De fato, o aprendizado do RUP demanda um certo esforço, mas o processo não é excessivo, já que foi concebido para ser especializado para cada Organização e para cada projeto, conforme prevê o nível de maturidade 3 do CMM-SW. Desta forma, permite que seja gerado um processo sob medida, com o grau de formalidade adequado para cada situação.

Em resumo, o RUP tem o mérito de ter abrangência e profundidade, mantendo ainda – tanto quanto possível – a qualidade da simplicidade.

Há ainda um ponto a ser justificado: por que utilizar o RUP (Rational Unified Process) e não o UP (Unified Process). O RUP é um produto comercial, e o UP é um processo público, da mesma forma que o OMT (*Object Modeling Technique*) de Rumbaugh, e tantos outros métodos. O UP, descrito no livro *The Unified Software Development Process*, de Jacobson e outros [JACOBSON1998], é uma versão simplificada do RUP, que foca os aspectos técnicos. Essencialmente, ambos são processos de engenharia de software, e contemplam as mesmas questões. O RUP tem o cuidado de não misturar a descrição do processo com recomendações de uso de ferramentas da Rational Software que dão suporte ao processo. A vantagem do RUP é seu maior grau de detalhamento e o fato de evoluir com maior rapidez. Para os objetivos deste trabalho, o RUP é mais adequado, porque já contempla o *plugin* J2EE, uma especialização para a plataforma J2EE.

1.6.2 Por que selecionar a plataforma J2EE

A **plataforma J2EE** direciona as aplicações construídas sob sua especificação para uma arquitetura de software em camadas, pois os componentes de aplicação são especializados (esse termo visa caracterizar que cada tipo de componente tem uma vocação específica – interface, controle, lógica de negócio, abstração de dados), facilitando o desenvolvimento em camadas.

Pode-se desenvolver um software para a web, funcionalmente equivalente a software construído sobre a plataforma J2EE, de forma muito mais simples. No entanto, esse software não necessariamente é **empresarial**, no sentido de atender a requisitos de escalabilidade, segurança, robustez e facilidade de evolução. Quando se pensa em todos esses aspectos, a plataforma J2EE vem **simplificar** o desenvolvimento de

software, já que o ambiente oferece diversos serviços de infra-estrutura às aplicações, e os projetistas ficam livres para focar a lógica do negócio.

A plataforma .Net da Microsoft é, na maioria dos aspectos, semelhantes à plataforma J2EE. Um aspecto importante é que a arquitetura e o projeto das aplicações tenham a devida qualidade, já que software não é (ou não deveria ser) um investimento de curto prazo. Nesse aspecto, a plataforma J2EE tem algumas vantagens sutis sobre a plataforma .Net e, por isso, foi adotada no escopo deste trabalho.

1.6.3 Por que especializar o RUP para J2EE

Os recursos disponibilizados pelo ambiente J2EE às aplicações nele hospedadas têm um preço, pois, para que o ambiente possa prestar esses serviços, é necessário que o projeto das aplicações seja efetuado com muito cuidado, considerando os requisitos que devem ser atendidos para que a infra-estrutura de serviços possa funcionar adequadamente. Por exemplo: o ciclo de vida dos componentes é gerenciado pelo container J2EE, e as aplicações não lidam com esses aspectos. Mas os projetistas de aplicação têm que estar alertas a essa questão quando projetam cada componente e a sua interação com os demais, pois a forma que o container atua sobre cada componente tem que ser considerada, para que a aplicação se comporte de forma adequada. Isso aumenta a complexidade do projeto de componentes na plataforma J2EE. Trata-se, portanto, de um "ganha-perde" em termos de complexidade, mas espera-se que o resultado final seja vantajoso para os projetistas de software.

Por conta de questões como a apresentada anteriormente, o projeto de aplicações J2EE tem que ser efetuado de forma muito cuidadosa. Essa é uma realidade conhecida pelo mercado. Quando uma empresa efetua um processo de seleção para um profissional que vai desenvolver aplicações J2EE, geralmente demanda bons conhecimentos de engenharia de software, além de conhecimento sobre a tecnologia de implementação.

Projetos de desenvolvimento de software na plataforma J2EE demandam um processo de desenvolvimento de software com o conjunto de características que o RUP possui:

- A definição de uma **arquitetura de software** tem um papel central no processo. Ainda nos estágios iniciais do desenvolvimento de um aplicativo, há diretrizes detalhadas para que se modele uma arquitetura de software que foca em requisitos (vista de casos de uso), projeto (vista lógica), implementação, processos (*threads* de execução, processos de execução) e organização da implantação do software.
- A disciplina de Análise e Projeto do RUP prevê que sejam identificados o que se denomina **mecanismos comuns**, de análise, projeto e implementação. Esses mecanismos definem soluções comuns para problemas comuns em um certo contexto (por exemplo, o contexto de uma certa plataforma de desenvolvimento).
- Na análise, o RUP define **classes especializadas**. O processo prevê que as classes de análise sejam classificadas como classes Entidade, classes de Controle ou Classes de Fronteira. Essa definição facilita a definição de diretrizes para derivar classes de projeto voltadas para determinada plataforma, a partir de cada tipo de classe de análise.

Uma forma efetiva de auxiliar os projetistas de software de aplicações empresariais, especialmente no ambiente web, é especializar o RUP para a plataforma J2EE. Isso implica em adaptar ou especializar atividades, diretrizes, artefatos e papéis previstos pelo RUP para o desenvolvimento na plataforma J2EE. Há alguns anos essa especialização está contemplada no RUP, e novamente através plugin J2EE do RUP 2002. No entanto, esta especialização pode ser estendida e aprofundada, e diretrizes mais detalhadas podem ser geradas, justificando o objetivo deste trabalho.

No prefácio de *J2EE Design Patterns*, Booch afirma que J2EE certamente é uma plataforma importante, que permite que equipes desenvolvam sistemas muito poderosos. Entretanto, a realidade é que ainda existe um grande *gap* semântico entre as abstrações e serviços que a plataforma J2EE provê e a aplicação final que a equipe deve construir [DEEPAK 2001].

A plataforma J2EE foi concebida com o intuito de permitir que a equipe de desenvolvimento possa se focar na lógica de negócio das aplicações, já que a plataforma já provê serviços de infra-estrutura de forma padronizada – muitos deles serviços declarativos, e que, portanto, não demandam codificação – inclusive para

questões relativamente complexas, como a persistência dos objetos em ambiente relacional. A seguir, são apresentados alguns exemplos, para caracterizar melhor a abordagem J2EE referente a serviços de infra-estrutura:

- Há um tipo de componente de negócio na plataforma J2EE, denominado *entity bean*, que é responsável pela persistência das informações, geralmente em bancos de dados relacionais. Nesses componentes, a execução dos métodos responsáveis por persistir e recuperar informações das bases de dados é responsabilidade do ambiente, não da aplicação. Mais ainda, em determinados casos, o programador não tem nem mesmo que escrever o código dos métodos – o próprio ambiente supre a implementação dos métodos, simplificando uma tarefa potencialmente complexa, que é o mapeamento objeto-relacional;
- O controle transacional pode ser efetuado de forma declarativa (não programática), associando o início/fim de uma transação à execução de um certo método, por exemplo;
- O ciclo de vida dos objetos é controlado pelo ambiente, que desta forma dispõe de recursos para ajustar a oferta de objetos à demanda, provendo escalabilidade à aplicação de forma transparente ao projetista/programador.

Como esses, há diversos outros exemplos, que caracterizam o grau de sofisticação do ambiente, mas infelizmente, isso não significa que projetar e implementar software para a plataforma J2EE tenha se tornado uma tarefa simples. Pelo contrário, o *gap* semântico, citado por Booch tem muitas facetas. Tarefas como acessar um banco de dados relacional utilizando JDBC, ou acessar um objeto remoto, demandam uma codificação – e um esforço para compreender seu funcionamento que não é trivial. Um problema maior é a existência de uma grande distância entre o que pode ser feito (e que freqüentemente é feito) e o que deveria ser feito, em termos de projeto/implementação na plataforma. É comum que aplicações desenvolvidas para a plataforma J2EE, na prática, não alcancem os benefícios propostos pela plataforma, por conta de projetos altamente inadequados.

Um exemplo primário de problemas desse tipo: por conta da relativa complexidade dos conceitos associados a um tipo de elemento que a plataforma J2EE caracteriza como sendo o componente de negócio, muitas vezes desenvolvem-se aplicações nas

quais toda a lógica de negócio é implementada na camada web da aplicação, sem utilizar os componentes de negócio. Só que o elemento central da plataforma J2EE são esses componentes, é em função deles que a arquitetura foi montada, e é o seu uso adequado que garante que os benefícios propostos pela plataforma possam ser atingidos.

Outro exemplo: *entity beans* comumente implementam métodos que devolvem coleções de instâncias do *entity bean*, a partir de um argumento de pesquisa. Seja um aplicativo de venda de livros on-line, onde seja possível pesquisar livros por autor, editora e nome. O projetista poderia achar natural utilizar esse método de pesquisa desse tipo, do *entity bean* Livro, para retornar uma coleção de livros que atendem a um certo critério de pesquisa. Desta forma, quando o usuário da aplicação solicitar que seja exibida uma lista de livros de determinada editora que possua mil títulos em catálogo, seriam instanciados mil *entity beans*, apenas para que seja exibida a lista de livros daquela editora ao usuário. Caso o usuário deseje adquirir um livro da editora, teriam sido instanciados mil *entity beans* para que o usuário efetivamente utilizasse os serviços de um deles. Há, evidentemente, soluções muito mais eficientes para este problema, mas a solução mais óbvia, que parece respeitar plenamente a filosofia da plataforma J2EE, é a apontada anteriormente – extremamente ineficiente.

Utilizar boas práticas de engenharia de software é fundamental em qualquer projeto, independente de plataforma, mas neste caso a questão talvez mereça ainda mais destaque. Mais ainda, o desenvolvimento para a plataforma expande o conjunto de boas práticas que devem ser consideradas, que devem agora englobar também práticas de projeto específicas para o ambiente J2EE. Isso não é uma desvantagem da plataforma, mas o preço a pagar para dispor da infra-estrutura que ela provê.

A demanda por boas práticas de engenharia de software aproxima a plataforma J2EE ao RUP por três razões:

a) O RUP aborda de forma concreta um conjunto fundamental de boas práticas de engenharia de software;

b) O RUP tem se expandido nos últimos anos para incorporar especializações específicas para certas áreas (como o desenvolvimento de aplicações *e-business*)

ou plataformas (como .Net e J2EE). Como já foi citado, o ponto de partida deste texto é o RUP com o *plugin* J2EE;

c) Mais importante: alguns elementos do processo contribuem para que boas práticas de projeto específicas para o ambiente J2EE sejam desenvolvidas. São eles:

- A ênfase que o RUP coloca na definição de uma arquitetura de software e na definição de mecanismos de arquitetura;
- A definição de alguns princípios para derivação do modelo de projeto a partir do modelo de análise, voltados especificamente para a plataforma J2EE.

1.7 *Detalhamento dos Objetivos*

Os elementos citados anteriormente serão aqui denominados estruturadores. Estes não são efetivamente elementos de projeto que serão posteriormente implementados, mas definições mais gerais, que estão em um plano que poderia ser denominado de **meta-projeto**: serão utilizados para estruturar, simplificar, agilizar e dar coerência ao projeto. Essa é uma idéia muito poderosa do RUP.

Este trabalho pretende definir os elementos estruturadores específicos para a plataforma J2EE, contribuindo para reduzir a distância entre conhecer os recursos da plataforma e poder projetar aplicações eficientes nessa plataforma.

A especialização no sentido mais típico, definir quais atividades e artefatos serão utilizados num projeto, já é efetuada pelo *plugin* J2EE (sucintamente, mas com informação suficiente), principalmente no *Roadmap* Desenvolvendo Soluções Baseadas em Componentes usando J2EE.

Os temas centrais da especialização são comentados a seguir:

- Arquitetura de software;
- Mecanismos de arquitetura;
- Derivação do modelo de projeto a partir do modelo de análise;
- Uso de patterns.

1.7.1 Arquitetura de Software

A arquitetura é um aspecto do projeto do software (ou uma parte do projeto), que define as questões mais amplas referentes ao projeto e diz respeito à estrutura: como o software vai ser organizado em subsistemas ou componentes, que por sua vez serão decompostos em outros subsistemas ou componentes, e através de quais interfaces esses elementos irão se comunicar e prover determinados comportamentos. O conceito de arquitetura de software abrange outros aspectos, não relevantes neste ponto do trabalho.

Segundo o RUP, o objetivo de definir uma Arquitetura de Software é mitigar riscos relativos a desempenho, capacidade, confiabilidade, dentre outros tipos de requisitos não funcionais, de modo que a funcionalidade do sistema possa ser adicionada na fase de construção sobre uma fundação sólida, sem risco de falhas.

1.7.2 Mecanismos de Arquitetura

Um mecanismo caracteriza uma solução concreta para problemas encontrados freqüentemente, e pode definir um padrão de estrutura, comportamento, ou ambos.

A disciplina Análise e Projeto do RUP enfatiza a definição de mecanismos de arquitetura, de modo a tornar o projeto mais coerente e simples.

Um exemplo clássico de problema que demanda a definição de um mecanismo de arquitetura é a persistência de objetos. Como os sistemas gerenciadores de bancos de dados que dominam o mercado adotam uma abordagem relacional, não orientada a objetos, a questão não tem uma solução trivial. Outro exemplo é a comunicação entre objetos distribuídos, que envolve um grau de complexidade. Em ambos os casos, há um problema freqüente, para o qual é possível definir um padrão de solução – esse é o mecanismo.

A definição de mecanismos de arquitetura específicos para a plataforma J2EE é uma das contribuições que esta dissertação procurar prover, já que há diversos desses problemas comuns no ambiente J2EE, não triviais, para os quais a definição de soluções consistentes é certamente uma contribuição para o sucesso dos projetos.

1.7.3 Derivação do modelo de projeto a partir do modelo de análise

O projeto de aplicações para a plataforma J2EE é muito particular, pois é fortemente influenciado pelas características e recursos específicos da plataforma. O objetivo é definir soluções padronizadas para a derivação do modelo de projeto para uma aplicação J2EE, a partir de um modelo de análise, de modo a simplificar a tarefa de projetar e a propiciar que o projeto contemple soluções adequadas tanto do ponto de vista de orientação a objetos, em geral, quanto da plataforma J2EE, em particular.

1.7.4 Uso de patterns

O uso consistente de patterns é uma excelente forma de garantir um projeto de alta qualidade, pois implica em utilizar a experiência acumulada por experts em engenharia de software. Por isso esse trabalho baseia-se fortemente na utilização de patterns.

Para embasar a definição dos elementos estruturadores, serão contemplados patterns de arquitetura de software, patterns de projeto genéricos, e patterns de projeto específicos para a plataforma J2EE.

1.8 *Estrutura do Trabalho*

1.8.1 Capítulo 1 – Introdução

Este é o presente capítulo. Este trabalho é contextualizado, os objetivos da dissertação e os elementos centrais envolvidos e apresentados e as escolhas efetuadas são justificadas.

1.8.2 Capítulo 2 – Plataforma J2EE

O capítulo 2 apresenta a linguagem Java e a plataforma J2EE. São descritos os aspectos relevantes que impactam no desenvolvimento de software, e apresentados o direcionamento, os benefícios e os recursos que uma infra-estrutura deste tipo pode oferecer para o desenvolvimento de software.

1.8.3 Capítulo 3 – Rational Unified Process

O capítulo 3 descreve o RUP, apresentando o seu objetivo, estrutura, as boas práticas de engenharia de software que podem ser utilizadas no desenvolvimento e as estratégias sugeridas para a sua utilização adequada.

A disciplina de Análise e Projeto, mais intimamente relacionada aos objetivos deste texto, será detalhada.

1.8.4 Capítulo 4 – Especialização do RUP para a plataforma J2EE

Este capítulo seleciona elementos da disciplina Análise e Projeto do RUP e, com base em *patterns* de arquitetura de software, *patterns* de projeto genéricos e *patterns* de projeto específicos para a plataforma J2EE, especializa os elementos selecionados para a plataforma J2EE. Os benefícios dessa abordagem são apresentados.

Os elementos selecionados para a especialização são relacionados a: arquitetura de software, mecanismos de arquitetura e derivação de modelo de projeto a partir do modelo de análise.

1.8.5 Capítulo 5 – Estudo de Caso

Para ilustrar os resultados obtidos no capítulo 4, e contemplando artefatos que compõem o RUP, o projeto parcial de uma aplicação é efetuado, envolvendo modelagem de negócio, gerência de requisitos e análise e projeto.

Os produtos da análise e projeto focados são baseados nos elementos especializados para J2EE definidos no capítulo 4.

1.8.6 Capítulo 6 – Considerações Finais

O capítulo 6 apresenta as conclusões e as contribuições da dissertação e os trabalhos futuros que podem ser realizados.

Uma análise dos resultados da especialiação é efetuada. As principais conclusões serão destacadas bem como linhas de trabalho futuras, desdobramentos desta dissertação.

1.9 *Síntese*

Desenvolver aplicações empresariais no ambiente web implica em uma complexidade maior, e a definição de um processo de software adequado e o uso da tecnologia apropriada se tornam ainda mais relevantes. Seguir um processo de desenvolvimento de software como o RUP aumenta as probabilidades de sucesso de projetos dessa natureza.

A plataforma J2EE, por outro lado, embora possa prover uma série de serviços de infra-estrutura de forma padronizada, demanda um projeto muito bem elaborado, principalmente por conta de questões de desempenho e manutenção evolutiva. A sofisticação da plataforma e a complexidade do ambiente web, em aplicações corporativas, cobram um projeto bem elaborado.

A própria plataforma J2EE estimula, em alguns aspectos, que boas práticas de engenharia de software sejam utilizadas. Mas isso não é suficiente, pois nada garante que um projeto adequado será efetuado. Seguir um processo de desenvolvimento que simplifique e efetivamente contribua para a solução de questões chave específicas do ambiente J2EE é fator chave para garantir o sucesso do desenvolvimento de software dessa natureza.

2. Plataforma J2EE

2.1 *Considerações Iniciais*

A linguagem Java surgiu como fruto de um projeto iniciado em 1991 por um grupo de engenheiros da Sun Microsystems, liderados por Patrick Naughton e James Gosling . Esse grupo, considerado visionário, recebeu da Sun a missão de antecipar e planejar a próxima onda da computação, e a sua conclusão inicial foi que uma das tendências significativas seria a convergência entre equipamentos de consumo controlados digitalmente e computadores [BYOUS 2002]. Esses aparelhos funcionariam em rede, e poderiam ser controlados remotamente.

O grupo planejou o desenvolvimento de uma linguagem de programação que poderia ser utilizada em equipamentos eletrodomésticos ou eletrônicos de consumo, como equipamentos de controle de TV a cabo. Esse objetivo determinou uma das características essenciais da linguagem Java, pois como diferentes fabricantes de equipamentos poderiam selecionar processadores diferentes, a linguagem não deveria ser focada em uma arquitetura específica. Foi desenvolvida então uma linguagem portável, que gerava código intermediário para uma máquina hipotética - ou máquina virtual. Desta forma, a máquina virtual é dependente do dispositivo, mas o código escrito em Java não.

Uma boa idéia dos projetistas da linguagem foi utilizar em Java a mesma sintaxe da linguagem C++, então extremamente popular, e que se constitui numa extensão da linguagem de programação C, para dotá-las de recursos para programação orientada a objetos. Isso facilitou a transição para Java de programadores C++.

Enquanto C++ é uma linguagem híbrida (incorpora recursos para programação orientada a objetos, mas é um superconjunto de C, uma linguagem estruturada), Java é uma linguagem puramente orientada a objetos. Em vez de funções, como as disponibilizadas pela linguagem C, Java contempla uma ampla hierarquia de classes prestadoras de serviços de diversas naturezas. O estudo dessa hierarquia de classes e de como utilizá-las já é educativo sobre os mecanismos de orientação a objetos.

Definida a linguagem, entretanto, durante alguns anos o projeto não encontrou nenhuma empresa que tivesse interesse em utilizá-la em equipamentos eletrônicos.

Com a popularização da *World Wide Web*, os engenheiros que haviam participado do projeto de Java vislumbraram, nesta tecnologia, uma oportunidade para utilização do conceito e da linguagem que desenvolveram – e, para ilustrar essa convergência de idéias, foi construído um *browser* denominado HotJava. A divulgação do *browser* via Internet foi o primeiro passo para a popularização da linguagem Java e dos conceitos que a embasavam. O grande marco para a popularização de Java, porém, foi a decisão da Netscape, em 1995, de incluir suporte a Java na versão 2.0 do então extremamente popular *browser* Netscape Navigator.

A Sun apresentou a primeira versão pública de Java em 1996. Ao longo desses anos, a linguagem tem evoluído de forma extremamente dinâmica e, nas versões seguintes à inicial, importantes características foram incorporadas à linguagem, tornando-a de fato um ambiente profissional para desenvolvimento.

Um ponto fundamental é que a especificação de Java foi sendo enriquecida com elementos que a habilitavam a ser utilizada para desenvolver os componentes do servidor, em aplicações cuja arquitetura compreende várias camada. Essa evolução levou ao desenvolvimento da especificação J2EE.

2.2 *Origem da Plataforma J2EE*

A plataforma Java 2 Platform, Enterprise Edition (J2EE) é um ambiente baseado na linguagem Java, para desenvolver, implantar e executar as aplicações baseadas em componentes distribuídos.

O termo *enterprise*, que consta no nome da plataforma, caracteriza a natureza e, implicitamente, o grau de complexidade das aplicações que podem ser desenvolvidas sobre essa plataforma. Aplicações com enfoque empresarial demandam, geralmente, serviços como gerência de transações, segurança, acesso a bases de dados, entre outros. Uma das características da plataforma J2EE é prover, ela própria, esses serviços, de modo a que as equipes de desenvolvimento possam se focar na lógica de negócio das aplicações, sem a necessidade de se envolver nos serviços de infra-estrutura.

A lógica de negócio das aplicações desenvolvidas sobre a plataforma J2EE é encapsulada em componentes que são executados em servidores J2EE, e que podem ser acessados por programas clientes. Esses servidores atuam como *middleware* entre os programas clientes e as camadas de acesso a dados ou aplicações legadas.

A plataforma J2EE é o último estágio da evolução da plataforma Java.

Em um primeiro momento, a Sun Microsystems focou a construção do ambiente para desenvolvimento e execução de aplicações Java, denominado JDK - Java Development Kit. O foco principal era a linguagem de programação, e não se considerava a infra-estrutura para os componentes executados em servidores.

Em um segundo momento, com a mudança de foco das aplicações corporativas do ambiente cliente-servidor tradicional para o ambiente *web* e arquitetura multicamadas, a Sun desenvolveu várias APIs (*Application Programming Interfaces*) que disponibilizaram os serviços de infra-estrutura, entre os quais, serviços transacionais, serviços de nomes e diretórios e, mais importante, a primeira versão da API EJB (*Enterprise Javabeans*, descrita neste capítulo).

Com o advento dos primeiros servidores de aplicações - dos quais o BEA *Web*logic Server foi o primeiro - que implementavam tanto a especificação EJB quanto as diversas APIs de serviços desenvolvidas pela Sun, foi evidenciado o potencial do ambiente, mas também diversas falhas, dentre as quais se destacam:

- Lacunas na especificação das APIs, especialmente da API EJB, impediam a portabilidade de componentes, fato inaceitável dada a proposta de Java;
- Embora as APIs fossem inter-relacionadas, foram desenvolvidas independentemente. Conseqüentemente, a sincronia entre elas era baixa. Mais ainda, cada uma das APIs evoluía separadamente. Isso tornava mais complexo o desenvolvimento de EJBs, já que estes dependiam das demais APIs, e era outro fator a dificultar a portabilidade de aplicações.
- Sendo as APIs apenas especificações, sem implementação de referência, os envolvidos com o desenvolvimento de aplicações não tinham como testar se seu produto era compatível com a especificação da Sun. Da mesma forma, era

impossível testar se um determinado servidor de aplicações era compatível com a especificação EJB 1.0.

Quando os problemas foram evidenciados, a Sun concebeu uma solução que consistia na definição de **três plataformas Java distintas**, cada uma delas englobando a anterior:

- Java 2 Platform, Micro Edition (J2ME), voltada para dispositivos específicos, como pagers ou PDAs (*personal digital assistants*);
- Java 2 Platform, Standard Edition (J2SE), versão básica da linguagem Java, que contempla, dentre outros, recursos para entrada/saída, ou recursos gráficos para uso em applets e aplicações;
- Java 2 Platform, Enterprise Edition (J2EE), que integra as APIs Java Enterprise numa plataforma integrada e que evolui de forma coerente, o que resolve os problemas identificados anteriormente.

2.3 *Caracterização da Plataforma J2EE*

Segundo Allamaraju, a plataforma J2EE é essencialmente um ambiente Java com um servidor de aplicações distribuídas que provê [ALAMARAJU 2001]:

- Um ambiente de execução que hospeda e gerencia aplicações. Este ambiente é um servidor no qual as aplicações residem.
- Um conjunto de APIs extensão de Java para construir aplicações. Essas APIs definem um modelo de programação para as aplicações J2EE.

Estes recursos são descritos nas sub-seções seguintes.

2.3.1 Ambiente de execução J2EE

A plataforma J2EE contempla um ambiente de execução em um servidor que hospeda e gerencia os componentes das aplicações J2EE nele instalados.

A especificação J2EE não define como o ambiente de execução deve ser construído. Em vez disso, define os papéis e as interfaces para os componentes de aplicações que executarão nesse ambiente, e uma demarcação clara entre as aplicações e os serviços

de infra-estrutura que essas aplicações demandam, tais como acesso a bases de dados, serviços de nomes e diretórios, controle transacional e serviços de mensagens.

Esse ambiente de execução é denominado *container*, e se constitui conceitualmente em uma camada que envolve todas as aplicações que são executadas no servidor J2EE. Cada aplicação acessa os serviços de infra-estrutura de que necessita através do *container*, que provê acesso a cada implementação de serviço através de uma API J2EE.

Qualquer fornecedor de um servidor J2EE pode implementar, da forma que achar mais conveniente, cada um dos serviços prestados pelo ambiente de execução, mas deverá necessariamente disponibilizá-los com interfaces padronizadas.

2.3.2 APIs J2EE

Antes do advento da plataforma J2EE, boa parte do esforço de construção de aplicações distribuídas era voltada para prover mecanismos de acesso a esses serviços de infra-estrutura, geralmente utilizando APIs proprietárias para cada tipo de serviço (com a notória exceção do acesso a bases de dados, feito com base em soluções padronizadas).

Além da necessidade de lidar com diferentes APIs proprietárias, outra questão importante do ambiente pré J2EE era a necessidade de desenvolver soluções artesanais para lidar com a inevitável escassez de recursos nos servidores. Como exemplo, pode-se citar a construção de *pools* de conexões para acesso a bases de dados.

Para resolver essas questões, a plataforma J2EE define um conjunto de APIs que serão utilizadas pelas aplicações para prover a estas serviços de infra-estrutura.

AS APIs J2EE são especificações, independentes de implementação. A premissa é que deve ser possível acessar os serviços providos por essas APIs de forma padronizada, independentemente de como são implementadas.

A especificação J2EE não requer que o *container* implemente, ele próprio, os serviços de infra-estrutura que o ambiente provê. O *container* pode agir como intermediário ou implementar ele próprio o serviço. O requisito é que a API padrão correspondente deva ser disponibilizada pelo *container*.

A especificação J2EE tem evoluído de forma coerente. A sua versão mais atual é a 1.3, e inclui o seguinte conjunto de extensões padrões de Java, que cada servidor de aplicações J2EE deve implementar:

- **Enterprise JavaBeans (EJB) versão 2.0**

  Esta especificação define uma arquitetura que contempla diferentes tipos de componentes que compõem as aplicações distribuídas multicamadas. Tanto a constituição desses componentes quanto a infra-estrutura de execução, que os hospeda e gerencia, é definida por esta especificação.

  BODOFF define um *enterprise bean* como um corpo de código com campos e métodos que implementam módulos de lógica de negócio. Pode-se pensar em um *enterprise bean* como um bloco de construção que pode ser utilizado isoladamente ou com outros *enterprise beans*, para executar a lógica de negócio em um servidor J2EE [BODOFF 2002].

- **Java *Servlets* versão 2.3**

  Esta especificação contempla as classes Java especializadas na construção de aplicações *web* dinâmicas.

- **JavaServer Pages (JSP) versão 1.2**

  Esta especificação contempla os recursos para a geração de páginas *web* dinâmicas a partir de modelos de páginas HTML.

- **Java Database Connectivity (JDBC) 2.0**

  A versão padrão desta API define um conjunto de classes extensão de Java para acesso de forma padronizada a bancos de dados. Uma extensão desta API, denominada JDBC 2.0 *Optional Package* aprimora a especificação JDBC 2.0, adicionando meios mais eficientes de obter conexões, *pools* de conexões, transações distribuídas, etc.

- **Java Message Service (JMS) 1.0**

  JMS é um padrão que permite que componentes de aplicações J2EE criem, enviem, recebam e leiam mensagem. Permite comunicação distribuída com baixo acoplamento, confiabilidade e de forma assíncrona [BODOFF 2002].

Esta API Java disponibiliza recursos referentes a filas de mensagens, e serviços de *publish and subscribe*.

- **Java Transaction API (JTA) 1.0**

  Esta API disponibiliza os recursos para o desenvolvimento de aplicações transacionais distribuídas.

  A plataforma J2EE permite construir aplicações em que o controle transacional é definido de forma declarativa. Isso significa que o *deployment descriptor* de um componente EJB pode conter parâmetros que definem o comportamento de cada método do componente, sob o aspecto transacional. Assim, a execução de um método de um componente EJB pode, por exemplo, disparar uma nova transação, que será encerrada quando a execução do método for concluída.

- **JavaMail 1.2**

  A API JavaMail disponibiliza recursos para que aplicações Java possam enviar e-mails.

- **JavaBeans Activation Framework (JAF) 1.0**

  Esta API é requerida pela API JavaMail, para determinar o conteúdo de arquivos MIME (Multipurpose Internet Mail Extension) e determinar quais operações podem ser efetuadas sobre diferentes partes de um e-mail.

- **Java API for XML Parsing (JAXP) 1.1**

  Esta API provê recursos para manipulação de documentos XML por aplicações Java.

- **The Java Connector Architecture (JCA) 1.0**

  Esta API permite integrar componentes de aplicações J2EE com sistemas legados.

- **Java Authentication and Authorization Service (JAAS) 1.0**

  Esta API provê mecanismos de segurança, especificamente a autenticação e a autorização.

As APIs citadas acima são específicas da especificação J2EE, e se somam às seguintes APIs da plataforma J2SE:

- **RMI-IIOP API**

  O ambiente Java contempla um protocolo específico para viabilizar a comunicação entre objetos distribuídos, denominado Java Remote Method Invocation (RMI). O ambiente CORBA possui um protocolo com mesma finalidade, denominado Internet Inter Orb Protocol (IIOP). Esta API torna possível a comunicação distribuída entre esses dois ambientes.

- **Java Interface Definition Language (IDL) API**

  Esta API disponibiliza recursos para que componentes J2EE possam acessar componentes CORBA através do protocolo Internet Inter Orb Protocol (IIOP). Esta API se relaciona à API RMI-IIOP, descrita mais adiante.

- **Java Naming and Directory Interface (JNDI) API**

  Esta API padroniza o acesso a diversos serviços de nomes e diretórios hoje utilizados. Uma árvore JNDI é uma estrutura de nós. Cada nó possui um nome e pode conter objetos serializados (serialização é um recurso da linguagem Java para armazenar objetos em meios magnéticos). Um cliente que deseja ter acesso a um objeto pesquisa o objeto pelo nome na árvore JNDI, uma vez que este seja encontrado em determinado nó da árvore, a instância serializada do objeto é devolvida ao cliente.

- **JDBC Core API**

  Esta API provê recursos básicos de acesso a bancos de dados.

2.4 *Arquitetura da Plataforma J2EE*

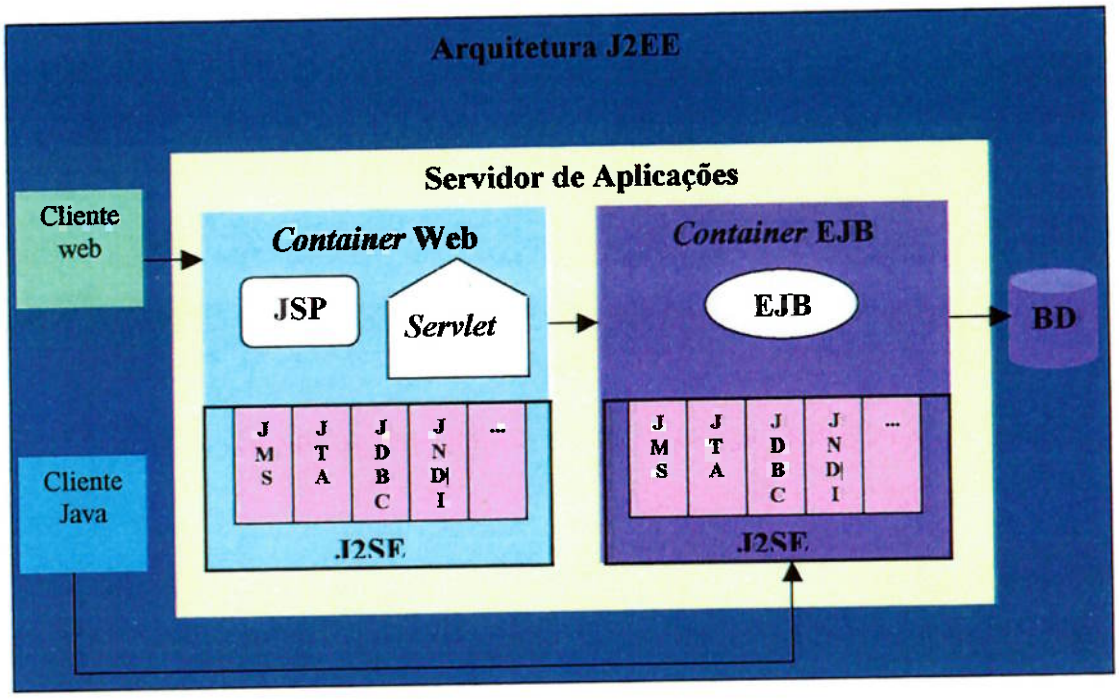


Figura 2-1 Arquitetura J2EE

A figura 2.1 ilustra a arquitetura da plataforma J2EE.

O elemento central da arquitetura de um servidor de aplicações J2EE é denominado *container*.

Allamaraju define um *container* como um ambiente de execução para gerenciar componentes de aplicações desenvolvidos de acordo com a especificação J2EE, e para prover acesso às APIs J2EE [ALLAMARAJU 2001]..

Os componentes hospedados nos *Containers*, denominados componentes de aplicação, são muitas vezes também referidos como objetos gerenciados, pois seu ciclo de vida é gerenciado pelo *container* no qual estão hospedados.

Containers são parte dos denominados Servidores de Aplicações J2EE.

A especificação J2EE define o conjunto de serviços que devem ser prestados pelo *container*, mas o *container* pode prestar outros serviços, além dos demandados. Diferentes fornecedores de servidores de aplicações J2EE podem, por conta disso, competir entre si, oferecendo serviços que os diferenciariam dos concorrentes, ainda

que, necessariamente, respeitando a padronização que permite portabilidade das aplicações entre diferentes servidores J2EE.

A plataforma J2EE contempla dois *Containers* que devem ser executados no servidor de aplicações:

- um *web container*, para hospedar *servlets* e páginas JSP;
- um EJB *container*, para hospedar componentes EJBs.

Cada *container* provê acesso dos componentes nele gerenciados às APIs que prestam serviços de infra-estrutura. Além disso, cada *container* define sua própria API. O *web container* define a API Java *Servlet*, e o EJB *container* define a API EJB.

Tanto os componentes que são executados no *container web* quanto os que são executados no *container* EJB são prestadores de serviços, e como tal têm clientes.

Os clientes dos componentes do *container web* normalmente são executados em *web* browsers. Neste caso, a interface visual é gerada no servidor como HTML ou XML e exibida pelo browser cliente. O protocolo utilizado pelos clientes para se comunicarem com o *container web* é o HTTP. O *container web* recebe requisições do cliente *web* e gera resposta a partir dos componentes de aplicação (JSPs e *Servlets* nele hospedados).

Os clientes dos componentes do *container* EJB podem ser de três tipos:

- aplicações cliente, que acessam os componentes EJB via protocolo RMI-IIOP;
- componentes de um *web container*, *Servlets* e JSPs, que utilizam o mesmo protocolo das aplicações *standalone*;
- outros EJBs rodando no *container* EJB, que se comunicam utilizando chamadas de métodos Java padrão.

Em todas essas situações, os componentes no *container* são acessados via próprio *container*.

2.4.1 Arquitetura do *Container*

O *container* hospeda componentes de aplicação, que podem ser *Servlets*, JSPs (no caso do *container web*), ou EJBs (no caso do *container* EJB). Esses componentes podem ser empacotados em arquivos denominados *archive*.

Cada componente é acompanhado por um arquivo XML denominado *deployment descriptor* (descritor de implantação), que descreve o respectivo componente e também contempla informações necessárias para que o *container* possa gerenciar os componentes de aplicação.

A arquitetura do *container* pode ser dividida em quatro partes: contrato do *container*, APIs de serviço do *container*, serviços declarativos e outros serviços. Cada um desses elementos será descrito a seguir.

2.4.2 Contrato do *container*

A especificação do *container* contempla um conjunto de APIs. Os componentes de aplicação devem obrigatoriamente estender (herdar) ou implementar essas APIs.

Assim como uma *applet* Java, que é executada num *browser web*, roda na máquina virtual Java desse *browser*, os componentes de aplicação J2EE são instanciados e iniciados na máquina virtual Java do *container* que os hospeda.

Como foi citado anteriormente, esses componentes de aplicação são ditos gerenciados pelo *container*. Como se trata de um ambiente distribuído, os clientes dos componentes que rodam no *container* não têm como ser acionados diretamente pelos respectivos clientes (nem isso seria desejável, neste caso, pois a gerência que o *container* pode exercer sobre os componentes seria comprometida). Em vez disso, os clientes acionam sempre o *container*, que por sua vez aciona os métodos dos componentes de aplicação. Para que o *container* possa acionar os componentes de aplicação, estes devem ter uma interface conhecida pelo *container*. A solução, bastante engenhosa, para resolver esse problema, é apresentada a seguir, e demanda a explicação de um conceito de orientação a objetos denominado Interface, que pode ser modelado em UML e é implementado pela linguagem Java.

Uma interface é uma classe desprovida de qualquer implementação de método. Uma classe Java convencional pode definir uma relação denominada "implementa" com uma interface Java. Isso vai obrigar a classe Java a implementar métodos que tenham a mesma assinatura dos métodos declarados na interface. Tanto esse recurso como a herança convencional entre classes são utilizados para definir o contrato que os componentes de aplicação devem implementar.

Para que o *container* possa se comunicar com os componentes de aplicação, ele deve conhecer a interface (ou pelo menos determinados métodos da interface) desses componentes. A especificação J2EE exige que os componentes de aplicação implementem determinadas interfaces pré-definidas ou estendam certas classes. Componentes em execução no *container web* (*Servlets* e JSPs) devem seguir a interface Java *Servlet*, que requer que os componentes estendam (definam uma relação de herança com) a classe `javax.servlet.http.HttpServlet`, e implementem certos métodos desta classe (como `doGet()` ou `doPost()`). Da mesma forma, quando compiladas, as classes geradas a partir das páginas JSP estendem a classe `javax.servlet.jsp.HttpJspPage`.

No caso do *container* EJB, tanto *session beans* quanto *entity beans* devem implementar as interfaces `javax.ejb.EJBHome` e `javax.ejb.EJBObject`, além de, respectivamente, as interfaces `javax.ejb.SessionBean`, ou `javax.ejb.EntityBean`. Já os message-driven beans devem implementar as interfaces `javax.ejb.MessageDrivenBean` e `javax.jmx.MessageListener`.

Essas interfaces são conhecidas do *container*, e há regras bem definidas sobre como e quando cada método das interfaces estendidas deve ser implementado pelos componentes de aplicação. É através desses métodos que o *container* pode gerenciar os componentes de aplicação. Essa gerência inclui localização, instanciação, gerenciamento de *pools* desses componentes, inicialização, invocação de serviços, e remoção de componentes. Todos esses aspectos do ciclo de vida dos componentes de aplicação são da responsabilidade do *container*, e os componentes de aplicação não têm nenhum controle direto sobre seu próprio ciclo de vida.

2.4.3 APIs de serviço do *container*

São serviços adicionais disponibilizados pelo *container*, que normalmente são requeridos pelas aplicações nele hospedadas.

O item 1.3.2 deste documento apresenta as APIs que a plataforma J2EE contempla, diversas das quais implementam os serviços de infra-estrutura. Por ser o ambiente J2EE distribuído, as aplicações hospedadas no *container* não dispõem de acesso direto à implementação dos serviços, que são acessados a partir do *container*.

O serviço Java Naming and Directory Interface (JNDI) é chave para acesso aos demais serviços de infra-estrutura pelos componentes de aplicação. Para cada API de serviço que se deseja acessar, um objeto apropriado deve ter sido criado e armazenado numa árvore de diretórios JNDI. Os componentes de aplicação dispõem de um mecanismo padronizado para acessar esses objetos (usando o serviço de nomes JNDI) e, através deles, ter acesso à implementação do serviço.

Os serviços sempre são acessados através do *container*, que implementa ou dispõe de recursos para acessar a implementação dos serviços. Há, conseqüentemente, um baixo acoplamento entre o cliente do serviço (o componente de aplicação) e a implementação do serviço. Desde que a interface exposta pelo objeto armazenado na árvore JNDI não mude, a implementação do serviço pode mudar sem afetar o código cliente.

Um benefício chave dessa abordagem é que o padrão (as APIs J2EE) pode se comunicar com uma variedade de sistemas de gerenciamento de bancos de dados, sistemas de processamento de transações, entre outros, mantendo, por um lado, a possibilidade de diversidade de implementações e, de outro, uma interface padronizada para utilização dos serviços.

2.4.4 Serviços declarativos

Serviços declarativos são serviços que o *container* presta às aplicações nele hospedadas. Dentre esses, destacam-se serviços de segurança e controle transacional. O termo declarativo se refere ao fato de que esses serviços não são solicitados programaticamente pelos componentes J2EE, mas definidos em *deployment descriptors* que acompanham cada componente.

O *deployment descriptor* é um arquivo XML que descreve cada componente de aplicação J2EE e define um contrato entre o componente e seu *container*. A responsabilidade por escrever o *deployment descriptor* é daquele que desenvolve o componente. Para cada tipo de componente J2EE há uma estrutura específica XML de *deployment descriptor*. Felizmente, as ferramentas de desenvolvimento J2EE automatizam a geração dos *deployment descriptors*.

Uma série de benefícios resulta dessa abordagem. O projeto das aplicações e a implementação dos componentes são simplificados, pois estes deixam de ter de explicitamente invocar os serviços declarativos. Se for necessário alterar a forma como o serviço declarativo deve ser executado para determinado componente, é suficiente alterar o *deployment descriptor* do componente (um arquivo texto convencional) – o código do componente não terá que ser alterado.

Caso a plataforma J2EE não implementasse esses serviços de forma declarativa, eles teriam que ser explicitamente invocados pelos componentes. O controle transacional, por exemplo, teria que ser demarcado no código dos componentes através da chamada de métodos para iniciar ou finalizar uma transação (*commit* ou *rollback*). Havendo os serviços declarativos, não há nenhuma referência no código dos componentes à demarcação transacional. Em vez disso, o *deployment descriptor* categoriza cada método de negócio de cada componente no que diz respeito ao controle transacional. Assim, uma transação pode ser iniciada quando da invocação de um certo método, e encerrada quando a execução desse método termina. Nesse exemplo, caso esse método chamasse outros métodos, esses estariam dentro do escopo da transação iniciada.

Na arquitetura J2EE, como já foi dito, o *container* recebe invocação de serviços dos clientes (pois os clientes não se comunicam diretamente com os componentes de negócio gerenciados pelo *container*) e os delega ao componente de aplicação adequado. Isso dá uma oportunidade ao *container* de interpor um serviço antes de transferir a solicitação para o serviço adequado. É desta forma que o *container* pode iniciar uma transação definida de forma declarativa antes de chamar um método de um componente de aplicação, e encerrar a transação imediatamente após o final da execução do método.

2.4.5 Outros serviços

Além dos serviços identificados nos itens anteriores, a seguir são listados alguns outros serviços que podem ser prestados pelos *containers*. Nem todos eles devem obrigatoriamente ser prestados, pois não previstos na especificação J2EE podem ser providos por servidores de aplicação. Os fornecedores de servidores de aplicação buscam se diferenciar das soluções concorrentes acrescentando serviços àqueles obrigatórios pela especificação J2EE.

- Gerência de Ciclo de Vida dos Componentes de Aplicação.

  Uma das questões centrais no caso de aplicações distribuídas, particularmente no caso daquelas voltadas para a Internet, é a escalabilidade. O desempenho das aplicações deve se manter satisfatório diante de uma demanda de clientes que pode ter grandes variações e que, muitas vezes, não pode ser prevista com muita precisão. *containers* J2EE têm a responsabilidade de adequar a quantidade de instâncias de cada tipo de componente à demanda, criando novas instâncias de componentes de aplicação, armazenando-as em *pools* de componentes ou destruindo-as quando não mais são necessárias.

- *Pools* de recursos.

  Containers podem, opcionalmente, implementar *pools* de recursos, tais como *pools* de objetos (citados anteriormente) ou *pools* de conexões a bancos de dados, ou ainda *pools* de *threads* de execução (caso, por exemplo, do servidor de aplicações BEA *Web*logic).

- *Clustering*.

  Outro recurso para prover escalabilidade e também para aumentar a disponibilidade de aplicações é denominado *clustering*, que consiste em integrar mais de um *container*, na mesma máquina ou em máquinas distintas. Os mesmos componentes de aplicação, nesse caso, são implantados em distintos *Containers* integrados. Valendo-se de determinada estratégia de balanceamento de carga e em função do tipo de componente de aplicação, as requisições de serviço são enviadas para um ou outro dos *Containers* integrados.

2.5 *Elementos Estruturais*

Os elementos descritos, a seguir, são denominados como estruturais porque a estrutura das aplicações é definida a partir deles. Esses elementos já foram citados no item APIs J2EE e são abordados neste tópico em maior detalhe. São eles: *Servlet*s, Java Server Pages (JSP), Enterprise Javabenas (EJBs).

2.5.1 Servlet

*Servlet*s são classes Java executadas em um *container web* e que contêm lógica de aplicação inserida em um processo denominado request-response do protocolo HTTP. Esse protocolo significa simplemente que determinado método de um *Servlet* a partir de uma requisição de serviço (request) de um cliente *web*, e produzirá uma resposta (response) a essa requisição. Essa resposta pode se constituir na geração de uma nova página HTML a ser exibida como resultado da requisição cliente, ou pode se constituir na invocação de um outro elemento responsável pela geração da vista ao cliente.

*Servlet*s estendem a funcionalidade de servidores *web*, pois geram conteúdo dinâmico HTML ou XML como resposta à requisição do cliente. Mas *Servlet*s são classes Java de pleno direito que podem se valer todos os recursos da linguagem. Desta forma, podem ser muito mais que apenas geradores de conteúdo HTML dinâmico.

Originalmente a função dos *Servlet*s efetivamente envolvia receber requisições e gerar dinamicamente páginas HTML, mas, com o advento da API JSP, mais adequada a esse propósito (veja tópico a seguir), os *Servlet*s passaram a ter mais uma função de controle do atendimento das solicitações de clientes, delegando a páginas JSP a responsabilidade pela geração dinâmica de páginas *web*.

2.5.2 JavaServer Pages (JSP)

JavaServer Pages (JSP) são de uma classe de API voltada para a geração de conteúdo *web* dinâmico. Uma página JSP pode conter código HTML, código Java e acessar componentes Javabeans.

Obs.: Um *javabean*, apesar do nome similar, é um elemento muito diferente de um Enterprise JavaBean (EJB). Um *javabean* é apenas uma classe Java convencional, que possui métodos de acesso (para leitura e escrita) a cada atributo que o componha. Javabeans são comumente denominados *helpers*, já que sua função geralmente é empacotar informação transmitida entre dois outros elementos (componentes de negócio e/ou da camada *web*).

A API JSP surgiu como uma resposta a um elemento semelhante do ambiente Microsoft, denominado Active Server Pages (ASP) e são, na verdade, uma extensão do modelo de programação de *servlet*s. Páginas JSP são compiláveis e, quando isso ocorre, são convertidas em classes do tipo *Servlet*s.

A idéia por trás das páginas JSP é aproveitar as diferentes especializações dos profissionais envolvidos com desenvolvimento de aplicações *web*. Um processo típico de trabalho que envolve JSP seria um *web* designer definir o aspecto visual da página (escrevendo em HTML) e, posteriormente, um programador Java inserir código Java em pontos específicos da página, de modo a permitir que, quando executada, o conteúdo dinâmico seja gerado.

Por ser esse modelo muito mais vantajoso para geração de conteúdo dinâmico que o modelo dos *Servlet*s, as páginas JSP passaram a ser o meio preferido para geração de conteúdo dinâmico.

2.5.3 EJB

A especificação Enterprise JavaBeans (EJB), atualmente na versão 2.0, como um modelo de componentes distribuídos para desenvolver componentes seguros, escaláveis, transacionais, de múltiplos usos. Idealmente, EJBs são unidades reusáveis de software contendo lógica de negócio. Assim como páginas JSP permitem a separação entre lógica de apresentação e lógica de aplicação, EJBs permitem a separação entre a lógica de aplicação e os serviços em nível de sistema, permitindo portanto que o projetista de software se concentre no domínio de negócio e não em programação do sistema [ALLAMARAJU 2001].

Um EJB não é uma classe Java, mas um conjunto delas, encapsuladas como um único componente. Um implementador de EJBs deve escrever o código de duas interfaces e de uma classe:

- Na *Home Interface* são declarados (mas não implementados, já que se trata de uma interface) os métodos de controle do ciclo de vida do EJB;
- Na *Remote Interface* são declarados os métodos que contemplam a lógica de negócio do componente EJB;
- O *enterprise bean* é a classe na qual são implementados métodos de negócio, métodos de ciclo de vida e métodos de contrato (métodos que o *container* usa para prover serviços e controlar o ciclo de vida do EJB).

Além dessa *enterprise bean*, outras classes Java podem ser incorporadas ao EJB, pois um EJB não corresponde à abstração de uma classe, mas sim à de um componente propriamente dito. Mas uma delas será a o *enterprise bean*, a partir da qual as outras serão acionadas.

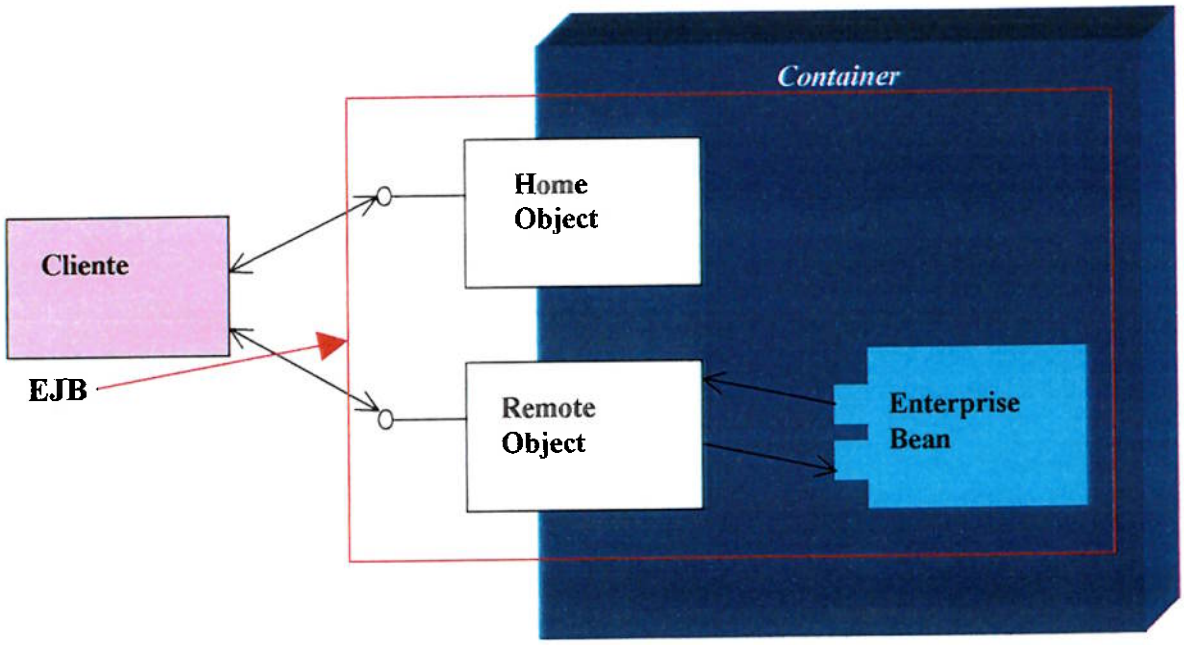


Figura 2-2 Elementos que compõem um EJB

Quando um EJB é compilado, o compilador EJB gera uma classe referenciada como Home Object a partir da Home Interface e classe referenciada como Remote Object a partir da Remote Interface. O implementador do código não toma conhecimento desse processo. O compilador EJB também compila o *enterprise bean*. Veja figura 2.2.

O *home object* será acionado por clientes para instanciar EJBs. Quando um EJB é instanciado, o cliente passa a ter acesso *ao remote object*. Esse é um objeto distribuído que pode ser acessado pelo cliente, e é através dele que o cliente acessa, indiretamente, os métodos de negócio implementados no enterprise bean.

O *enterprise bean* não é um objeto distribuído. Ele vive no *container*, e só pode ser acessado pelo *container*. Cada invocação de um método de negócio do *remote object* provoca o acionamento, via *container*, do método correspondente no *enterprise bean*.

Há três tipos básicos de EJBs: *session beans*, *entity beans* e *message-driven beans*. São descritos a seguir.

Session Beans

Um *session bean* representa trabalho executado para código cliente que o chamou. *Session beans* implementam a lógica de negócio, as regras de negócio, e os fluxos de trabalho [ROMAN 1999]. O termo *session* está associado ao conceito de sessão de usuário, importante no contexto de aplicações *web*. O tempo de vida de um *session bean* (conceitualmente, já que o ciclo de vida de um componente no servidor é gerenciado pelo *container*, e depende de outros fatores, como a existência de um pool de componentes) não excede o tempo de uma sessão de uso da aplicação por um cliente (normalmente, um browser *web*).

Supõe-se que um usuário, acessando um *browser web*, acesse uma loja virtual, pesquise produtos e efetue uma compra. A partir do momento que o *browser* acessa o servidor, uma sessão de cliente é definida, e perdura até que o usuário explicitamente se desconecte (efetuando *logout*, por exemplo), ou expire o prazo da sessão (ativa, em alguns casos, ou, na maioria das vezes, por estar inativa há um certo tempo). Esse seria o limite da existência dos *session beans* criados para atender a a pesquisa e a compra dos produtos. Na prática, em vez do *session bean* ser destruído ao final do

uso pelo cliente, ele poderia retornar a um pool de componentes, e ser reutilizado posteriormente.

Session beans são associados a um único cliente em um determinado momento, ao contrário de *entity beans*, que podem ser compartilhados entre diversos clientes.

Há dois tipos de *session beans*: *stateful session beans* e *stateless session beans*.

Um *stateful session bean* conserva seu estado entre diversas chamadas do mesmo cliente. O *container* aloca o *stateful session bean* a um cliente quando há a primeira chamada ao *bean*, e as chamadas subseqüentes irão acessar o mesmo *bean*, que preservará seu estado entre as chamadas, até que a sessão seja encerrada. Um exemplo típico de um *stateful session bean* é o carrinho de compras das aplicações de *e-commerce*. O carrinho deve manter a informação sobre os itens selecionados, até que a compra seja efetivada ou abandonada.

Um ***stateless session bean*** não retém estado entre diversas chamadas de um mesmo cliente. Um exemplo típico de um *stateless session bean* é o de um componente voltado para cálculo de dígito verificador de CPF. O bean é acionado, efetua o cálculo e devolve o resultado, e não há por que reter estado. O tempo em que um *stateless session bean* fica associado ao cliente é apenas a duração de uma execução de método seu acionado. Como não preserva estado entre chamadas, esse tipo de *bean* normalmente não é destruído após ser acessado (seria um desperdício de recursos). Em vez disso, normalmente volta a um pool de componentes, e fica disponível para atender qualquer cliente.

Evidentemente, o sistema consome uma quantidade muito maior de recursos para lidar com um *stateful session bean* do que com um *stateless* bean. Portanto, é muito importante caracterizar a necessidade real de uso de *stateful* beans.

Entity Beans

Como o nome já sugere, um *entity bean* é um objeto que modela uma entidade persistente num sistema de armazenamento de dados (geralmente, uma base de dados relacional). Por exemplo, os dados referentes a uma conta corrente ou a um pedido de compra podem ser modelados usando um *entity bean* ContaCorrente ou PedidoCompra, respectivamente.

As entidades modeladas na base de dados terão, portanto, nesse ambiente, um elemento correspondente na aplicação, os *entity beans*, que serão responsáveis pelo armazenamento e recuperação dessas informações na base de dados. Isso não implica que os *entity beans* não possam conter lógica de negócio. Sendo objetos de pleno direito, a eles cabe disponibilizar os serviços que, em uma abstração orientada a objetos, cabe às entidades que representam. Assim, por exemplo, o *entity bean* PedidoCompra deveria ter inteligência para realizar operações sobre pedido de compra, como informar os itens que compõem o pedido ou calcular o preço total do pedido. Nada impede que essa lógica seja complexa, se a abstração que o objeto (componente) representa assim o demandar.

Enquanto um *session bean* serve a um cliente de cada vez (até ser liberado por aquele cliente), um *entity bean* pode ser compartilhado por diversos clientes, podendo haver diversas instâncias de *entity beans* que representam a mesma instância de entidade. Por exemplo, um *entity bean* denominado ListaPreços de uma loja virtual pode ter diversas instâncias simultaneamente e representam a mesma lista de preços, devido ao requisito de desempenho. Como todos contêm exatamente a mesma informação, podem ser intercambiados entre os clientes sem nenhum prejuízo. Se um *session bean* denominado AtualizadorDePreços alterar o preço de uma instância, vai implicar em uma atualização na base de dados e, conseqüentemente, uma atualização de todas as demais instâncias, para que permaneçam com as mesmas informações. O *container* é responsável por esse processo.

Os mecanismos de persistência dos *entity beans* são, portanto, sempre gerenciados pelo *container*. Há, entretanto, duas formas de codificar os *entity beans*: uma denominada *container* managed persistence (CMP) e a outra denominada bean managed persistence (BMP). São descritas a seguir.

Ao contrário do que o nome sugere, um *entity bean* BMP terá a gerência de sua persistência efetuada pelo *container*, Mas o programador o responsável pela implementação do bean terá que escrever código em alguns métodos do contrato (acionados pelo *container*) segundo um estilo definido para persistir e recuperar dados da base.

Caso se opte pelo mecanismo de CMP, embora os métodos de contrato estejam sempre disponíveis, não é necessário escrever código referente à persistência e recuperação de dados. Essas operações serão realizadas próprio *container*, com base em informações contidas no *deployment descriptor* que mapeiam os atributos do objeto em campos da base de dados.

Message-Driven Beans

Esse tipo de *bean* foi incorporado às versões mais recentes da especificação EJB. Ao contrário dos outros tipos de *beans*, eles não são invocados diretamente pelos clientes. Seu objetivo é processar as mensagens recebidas através de Java Message Server (JMS). Eles propiciam um meio para processar as mensagens recebidas via JMS dentro do *container*. O *container* é responsável por invocar o *message-driven bean* apropriado para processar a mensagem, quando um cliente envia uma mensagem via JMS.

2.6 *Benefícios da plataforma J2EE*

De acordo com a Sun Microsystems, os principais benefícios trazidos pela plataforma J2EE são os seguintes [SUN 2002]:

- Padronização da arquitetura e do desenvolvimento;
- Escalabilidade para diversidade demanda;
- Integração com sistemas de informações existentes;
- Possibilidade de escolha de servidores, ferramentas e componentes.;

A seguir, cada um desses benefícios é comentado.

2.6.1 Padronização da Arquitetura e do Desenvolvimento

Um dos benefícios da plataforma J2EE vem do fato de ela se apoiar no paradigma da linguagem Java denominado *write once, run anywhere* (escreva o código uma só vez e execute-o em qualquer plataforma). Isso permite que o código compilado uma vez possa ser migrado para outras plataformas. O autor deste trabalho já fez a experiência, bem sucedida, de migrar o código compilado dos componentes de aplicação, do ambiente Windows para o ambiente Linux.

Os componentes de aplicação podem utilizar os serviços de infra-estrutura prestados pelo ambiente, e podem ser dinamicamente conectados a outros componentes. Aplicações podem ser configuradas de maneira flexível, valendo-se dos recursos declarativos definidos pela plataforma.

2.6.2 Escalabilidade para diversidade de demanda

O requisito de escalabilidade das aplicações na plataforma J2EE torna-se crítico, devido à natureza de aplicações empresariais para o ambiente *web* que, em muitas situações, não há como prever a quantidade de acessos simultâneos de clientes às aplicações ou nem se deseja isso.

É responsabilidade dos servidores de aplicações J2EE – não das aplicações – prover diversos serviços de infra-estrutura cujo objetivo é prover escalabilidade e conseqüentemente performance à aplicação. Alguns exemplos:

- Os servidores de aplicação podem gerenciar, por exemplo, pools de conexões a bancos de dados que sejam dinâmicos, ampliando ou reduzindo a quantidade de conexões abertas em função da demanda. Uma conexão a um banco de dados é um recurso "caro", que tem que ser utilizado com parcimônia. Por outro lado, número insuficiente de conexões implica em gargalos para as aplicações clientes dos bancos de dados. O servidor de aplicações define o número de conexões que compõem o pool dinamicamente, em função da demanda.
- O servidor de aplicações pode gerenciar pools de instâncias de componentes de negócio, adequando o número de instâncias ativas à demanda. Desta forma, na prática, o ciclo de vida de uma instância não é gerenciado pela aplicação, mas pelo servidor de aplicações. Quando a aplicação instancia um componente, ela pode receber uma componente que já estava instanciado no pool. Quando a aplicação destrói um componente, é possível que, a critério do servidor, o componente continue ativo, mas no pool de instâncias ou atendendo a outro cliente.
- É possível fazer diversos servidores de aplicação trabalharem em conjunto, para melhor performqnce – e escalabilidade, pois novos servidores podem ser incluídos no pool se houver demanda. Nesse caso, múltiplos *containers* EJBs que

hospedam as mesmas aplicações, acessados a partir de containers *web* implementados para realizar balanceamento de carga entre esses *containers* EJB, considerando a flutuação da demanda por uma determinada aplicação.

2.6.3 Integração com sistemas de informação existentes

A plataforma provê diversas APIs que permitem acessar sistemas de informação previamente existentes. As seguintes APIs provêem esse tipo de recurso:

- A API JDBC permite acessar gerenciadores de bancos de dados relacionais, utilizando a linguagem Java;
- A Java Transaction API (JTA) viabiliza controle transacional em ambiente distribuído heterogêneo;
- A API Java Naming and Directory Interface (JNDI) permite acesso a serviços de nomes e diretórios em diversos ambientes;
- A API Java Message Service (JMS) permite receber e enviar mensagens utilizando sistemas empresariais de mensagens como o IBM MQ Series e o TIBCO Rendezvous;
- A API JavaMail permite enviar e receber e-mails;
- A API Java IDL viabiliza a chamada de serviços CORBA.

É possível também viabilizar a comunicação de componentes J2EE com os componentes do ambiente Microsoft, utilizando soluções denominadas *bridges* disponíveis no mercado (entre outras soluções). Além disso, acesso especializado a sistemas de ERP e *mainframe*, como IBM CICS e IMS serão providos em futuras versões da plataforma J2EE, utilizando um recurso denominado arquitetura Connector. A evolução da plataforma J2EE permitirá que EJBs possam combinar o uso de objetos de acesso *connector* e APIs de serviço para desempenhar suas funções [ALLAMARAJU 2001].

2.6.4 Possibilidade de escolha de servidores, ferramentas e componentes

Um dos objetivos da plataforma J2EE é se posicionar, no mínimo, como uma alternativa válida para a plataforma de desenvolvimento da Microsoft. Um fator diferenciador dessa plataforma é justamente o fato de que empresas que aderem a ela não se tornam dependentes de um único fornecedor.

O padrão e o prestígio da J2EE são fatores para incentivar as empresas a desenvolver servidores de aplicações aderentes ao padrão, de modo a atingir o objetivo de diversidade de oferta de produtos. Mais ainda, como a especificação deixa espaço para prestação de serviços adicionais aos que compõem o padrão, os provedores de servidores de aplicação J2EE competem entre si para fornecer soluções reconhecidamente aderentes ao padrão e prestar serviços que os diferenciem dos concorrentes.

A especificação da plataforma foi feita de forma a que os componentes EJB e JSP sejam manipulados por ferramentas gráficas de desenvolvimento. Mais importante, diversas das tarefas do desenvolvimento de componentes podem ser automatizadas. Por exemplo, quando se usa EJBs BMP, a geração do código de diversos métodos de contrato pode ser facilmente automatizado, se a ferramenta de desenvolvimento dispuser de recursos para acessar as definições das tabelas da base de dados associada aos EJBs. Outro recurso relativamente simples e muito útil é a possibilidade de automatizar a geração dos *deployment descriptors* em XML, a partir de parâmetros fornecidos pelo projetista de software. Sendo os componentes especializados, é perfeitamente possível que ferramentas distintas ofereçam recursos especializados para determinados tipos de componentes, e a escolha das ferramentas mais adequadas se daria em função dos objetivos de um determinado projetista de software ou uma equipe. Novamente, a intenção é que seja gerado um mercado diversificado que ofereça ambientes de desenvolvimento.

Por conta da arquitetura componentizada, outro objetivo é criar um amplo mercado de componentes, que possam prover soluções específicas ou mesmo soluções verticais (soluções que abranjam um determinado domínio de interesse) inteiras. Esse mercado, ainda que não amplamente maduro, já existe. Há diversos produtos de

customer relationship management (CRM), por exemplo, que apregoam as virtudes de terem sido desenvolvidos na plataforma J2EE e, portanto, serem facilmente adaptáveis às necessidades específicas dos clientes.

A questão chave é a compatibilidade, e o objetivo é a liberdade de escolha, de modo a aumentar a confiança no fato de que os investimentos feitos estarão protegidos.

3. Rational Unified Process (RUP)

3.1 *Considerações Iniciais*

O Rational Unified Process (RUP) é um produto da Rational Software Corporation. É um processo de Engenharia de Software, com o objetivo de assegurar produção de software de alta qualidade, que atenda às expectativas dos usuários finais, com prazo e orçamento previsíveis.

O processo, no contexto considerado, tem o sentido proposto por Jacobson, que conceituou este termo em relação ao método [JACOBSON 1992]:

> "Um método define explicitamente os procedimentos a serem seguidos passo a passo".
>
> "Um processo amplia a escala do método, de forma que possa ser aplicado a projetos com muitas atividades e participantes que interagem".

O RUP contempla a definição de atividades técnicas de desenvolvimento de software, que produzem os artefatos em *Unified Modeling Language* (UML). Contempla também atividades gerenciais e de suporte, como gerência de projeto, garantia da qualidade, análise de riscos, gerência de configuração, preparação e manutenção de infra-estrutura de projeto, etc. Além das fases e dos produtos de um projeto, que são os elementos geralmente considerados nos modelos de desenvolvimento tradicionais, incorpora a atribuição de papel e responsabilidade aos participantes do projeto.

O RUP é apresentado sob a forma de um hipertexto *web*, organizando em duas dimensões: uma dimensão temporal – as fases do processo; uma dimensão de conteúdo – as disciplinas do processo. Cada disciplina é descrita, no nível mais alto, por um diagrama de atividades (UML) no qual cada elemento representa uma macro atividade, denominada Detalhe de Workflow. Cada detalhe de workflow, por usa vez, é descrito em termos de atividades, responsabilidades e artefatos. Um artefato é qualquer produto de trabalho gerado ao longo de um projeto que segue o processo.

As atividades são descritas de forma razoavelmente sucintas, mas cada disciplina também contempla um conjunto de Diretrizes e Conceitos, que funcionam como tutoriais, abordando temas de interesse para a disciplina de forma mais detalhada.

O RUP é o resultado da consolidação da experiência de inúmeros profissionais das áreas de desenvolvimento de software e de gerência de projetos. Tendo o RUP como referência, foi definida uma versão simplificada de processo de software, o Unified Process (Processo Unificado, conhecido como UP). A descrição do UP pode ser encontrada em [JACOBSON 1999].

3.2 *Histórico*

A Objectory AB – empresa sueca de engenharia de software fundada por Ivar Jacobson –, comercializava, desde 1987, um processo de engenharia de software denominado Objectory. Em 1992, Jacobson publicou uma versão simplificada do Objectory, com o nome de "Object Oriented Software Engineering" (OOSE) [JACOBSON 1992]. Esse método se diferenciava dos demais, entre alguns outros aspectos, por definir uma forma nova para capturar os requisitos funcionais de software, o Modelo de Casos de Uso.

Ao longo dos anos 90, a Rational se expandiu pela aquisição ou fusão com outras companhias de software. Um ganho importante para a Rational nessas fusões e aquisições é que, a cada evento desses, determinada inteligência referente ao processo (bem como ferramentas de software de suporte ao processo) era incorporada à Rational.

Em 1995, a Rational adquiriu a Objectory AB, e incorporou o processo Objectory, então na versão 3.8, ao conjunto de produtos que comercializava.

Após a aquisição da Objectory pela Rational, foi lançado o Objectory 4.0, que agregava à versão anterior do processo pontos chaves da abordagem Rational, como o ciclo de vida iterativo e a ênfase na definição de uma Arquitetura de Software. Kruchten cita que informa que essa versão do processo também incorporava material sobre gerência de requisitos da Requisite, Inc., e um processo detalhado de testes da empresa SQA, Inc., companhias também fundidas à Rational Software. Além disso,

esta versão do processo foi a primeira a utilizar a então recentemente criada Unified Modeling Language (UML 0.8) [KRUCHTEN 2000].

Em 1997 o Objectory teve sua denominação alterada para Rational Unified Process, e sua descrição enriquecida. Desde então, tem havido evoluções periódicas do processo. A versão atual é denominada RUP 2002, e alterou substancialmente a disciplina de testes, por exemplo, em relação à versão 2001.

3.3 *Características Principais do RUP*

O RUP é muito extenso e, para que sua manipulação seja eficiente, é necessário que se invista em dois tipos de abordagem. A primeira está relacionada com a navegação no hipertexto, para buscar a informação de interesse em cada situação específica, no nível de detalhe necessário. A segunda está relacionada com o entendimento do processo propriamente dito. Neste contexto, esta seção apresenta as principais características do RUP.

A formatação do RUP como hipertexto é muito adequada, pois permite que as informações sejam organizadas em camadas. As camadas mais altas são mais abstratos, genéricos e abrangentes; as camadas inferiores descrevem, em detalhe, cada elemento específico do processo. Profissionais experientes podem se beneficiar das camadas mais altas da descrição do RUP, que descrevem questões mais estratégicas do processo, sem necessariamente acessar os detalhes, enquanto profissionais podem precisar utilizar as camadas inferiores como tutoriais para executar atividades específicas.

Freqüentemente o RUP é criticado por uma suposta complexidade excessiva. É fato que o hipertexto web que descreve o RUP é muito extenso. Se impresso, seriam geradas milhares de páginas. Alguém que desconheça o processo, e se proponha a compreendê-lo em detalhes a partir de uma leitura seqüencial do texto que o descreve, poderá facilmente se perder.

Os métodos de desenvolvimento de software orientados a objetos que antecederam o RUP focavam apenas a análise e o projeto de software orientado a objetos. O RUP é muito mais amplo, contemplando aspectos como gerência de projetos de software,

diretrizes e atividades para a implantação do software, e modelagem de negócio, entre outros.

O RUP foi concebido para atender aos níveis de maturidade 2 e 3 do modelo denominado Capability Maturity Model–Software (Modelo de Maturidade da Capacidade de Processos de Software – CMM-SW). Esse modelo foi desenvolvido pelo Software Engineering Institute (SEI), da Universidade Carnegie Mellon, para que os contratantes de software possam avaliar o grau de maturidade do processo de software de seus fornecedores potenciais [SEI 1995].

Um dos pontos chave do nível 3 do CMM define que cada Organização deve dispor de um *framework* de processo de engenharia de software, genérico, que será especializado para as características de cada projeto específico de desenvolvimento. O RUP se propõe a ser esse *framework*, ou a servir de base a concepção do *framework* de uma determinada organização. O processo foi construído para atender às necessidades de um amplo espectro de Organizações que desenvolvem software: desde as de grande porte, com projetos extremamente complexos que envolvem equipes geograficamente dispersas, até organizações de pequeno porte, nas quais os projetos envolvem, por exemplo, três profissionais ao longo de um mês.

Observa-se, no entanto, que nenhum profissional envolvido com o uso do RUP, independentemente do porte da Organização de que este faça parte, precisa conhecer o RUP na totalidade, para que este possa ser utilizado. Em uma organização de maior porte, o grau de formalismo requerido pelo processo é maior, mais detalhes terão que ser considerados, mas haverá um número maior de profissionais, com perfis especializados. Em organizações de menor porte, os profissionais são comumente mais generalistas. Em ambos os casos, o *framework* de processo será especializado para contemplar as características da organização e dos seus projetos.

Para utilizar o RUP de forma adequada é importante que este seja considerado como um vetor para uma mudança de atitude, e não um simples roteiro de atividades. Para isso, não é necessário aprender quais são todas as atividades previstas, todos os produtos de trabalho contemplados, todos os papéis, e assim por diante. Inicialmente deve-se considerar que:

- O RUP é fruto de experiência acumulada. O RUP contempla atividades e listas de verificação descritas detalhadamente. A consulta a esses elementos do processo pode ressaltar atividades importantes que, de outra forma, poderiam ser esquecidas;
- O RUP atua como um tutor, ensinando detalhadamente a realizar determinadas atividades, tanto técnicas (Como projetar um sistema web?), como gerenciais (Como planejar o projeto?).

Cabe, no entanto, ressaltar que a compreensão dos conceitos chave é fundamental para que os detalhes do RUP possam ser aproveitados. Caso esses conceitos não sejam compreendidos, não será útil fazer útil dos detalhes existentes no processo.

É muito comum que os artigos sobre o RUP abordem o que se denomina essência do processo ou conceitos chaves. Cada autor geralmente adota um ponto de vista semelhante aos demais, com alguns aspectos mais específicos. Nesta seção será apresentada a abordagem seguida pelo próprio RUP, que contempla dois tópicos referentes a essa visão essencial do processo: um denominado Boas Práticas de Engenharia de Software e outro denominado explicitamente Essência do Processo.

3.3.1 Boas Práticas de Engenharia de Software

A Rational aborda as Boas Práticas de Engenharia de Software como sendo conhecimento empírico acumulado ao longo de muitos anos de experiência de inúmeros profissionais envolvidos com o desenvolvimento de software. Na visão da Rational, o desenvolvimento de software está sujeito a inúmeros problemas, normalmente inter-relacionados; como exemplos típicos, cita-se a falta de integração entre componentes ou a descoberta tardia de erros de implementação. As Boas Práticas são fortemente inter-relacionadas, e visam, coletivamente, evitar que esses problemas se materializem; conseqüentemente contribuem para o sucesso dos projetos de software.

O RUP foi construído de modo a implementar as Boas Práticas. Com isso queremos dizer que a organização do processo, as atividades nele previstas e as diretrizes definidas para executar essas atividades, foram definidas de modo que as Boas

Práticas sejam naturalmente seguidas quando efetivamente se implementa o processo.

As Boas Práticas consideradas no RUP são: Desenvolvimento Iterativo, Gerência de Requisitos, Utilização de Arquiteturas Baseadas em Componentes, Modelagem Visual, Verificação Contínua de Qualidade, Gerência de Mudanças.

Desenvolvimento Iterativo

Este tópico é considerado o pilar do RUP, pois este é organizado em função do ciclo de vida iterativo, que molda todos os aspectos do processo. A mudança de um ciclo cascata de desenvolvimento de software (ainda adotado na maioria das organizações) para o ciclo iterativo é considerada fundamental para que o processo possa ser efetivamente implantado.

No RUP, o software é desenvolvido através de alguns ciclos, denominados iterações. Cada iteração é um mini ciclo cascata, em que todas as disciplinas do processo (captura de requisitos, análise e projeto, implementação, testes, etc.) podem ser efetuadas. Em cada uma dessas iterações haverá ênfase diferente nas disciplinas do processo, em função do estágio em que se encontre o desenvolvimento do software. Assim, uma iteração no início do projeto focará mais a captura de requisitos, análise e projeto (e em determinadas atividades da captura de requisitos e análise e projeto), enquanto iterações mais ao final do projeto focarão implementação, testes e implantação. Um ponto a ser destacado é que cada iteração gera um software executável, não necessariamente para ser implantado no ambiente do Cliente, e é gerenciada como um *timebox* (com duração pré-fixada, rígida), sendo seu escopo variável para se adequar ao *timebox*. O planejamento em mais alto nível de um projeto consiste em definir o número, duração e objetivo das iterações.

O ciclo iterativo está fortemente relacionado com um ponto considerado essencial no RUP: mitigar riscos o mais cedo possível. O processo é definido como sendo dirigido por riscos. Isso quer dizer que os riscos são identificados e (re) avaliados antes de cada iteração, ao longo de todo o projeto, e as funções priorizadas para a próxima iteração são aquelas que contribuirão para mitigar os riscos mais altos.

O RUP considera necessário que o código seja implementado e testado, para que se possa efetivamente mitigar riscos o quanto antes. Desta forma, que a proposta é fazer isso o mais cedo possível, ainda quando apenas parte dos requisitos tenha sido detalhada. Isso torna o RUP mais realista, pois em geral é praticamente impossível que todos os requisitos sejam detalhados antecipadamente, quando nenhum código foi produzido, testado, e o resultado avaliado. Da mesma forma, muitas vezes não é viável terminar o projeto do sistema antes de iniciar a implementação.

Uma das grandes vantagens associadas ao ciclo iterativo é a maior facilidade de lidar com mudanças dos requisitos e permitir mudanças táticas ao longo do projeto. Existe ainda maior possibilidade de propiciar a obtenção de componentes reutilizáveis.

Gerência de Requisitos

Essa Boa Prática define que os requisitos devem ser criteriosamente identificados, detalhadamente documentados e, mais que isso, gerenciados, ao longo do ciclo de vida. A premissa do RUP é que os requisitos vão mudar ao longo do projeto – por diversos motivos, vários dos quais estão totalmente fora do controle da equipe de desenvolvimento. Por isso os requisitos devem ser identificados, detalhados, seus inter-relacionamentos mapeados (por exemplo, requisitos genéricos com requisitos detalhados, quando estes são derivados daqueles), acordados com usuários e clientes, e, por fim, as eventuais mudanças nos requisitos e suas conseqüências adequadamente tratadas.

O RUP contempla uma disciplina denominada Gerência de Requisitos, que define todos os aspectos relativos à implementação dessa Boa Prática.

Utilização de Arquiteturas Baseadas em Componentes

Outro ponto essencial do processo é a definição de uma Arquitetura de Software baseada em componentes.

Dois conceitos embasam essa Boa Prática: o de Arquitetura de Software e o de Componente.

O conceito de Arquitetura de Software envolve diversos aspectos:

- Organização geral do software;

- Organização do software em subsistemas e componentes;
- Protocolos de comunicação, sincronização e acesso a dados;
- Distribuição física;
- Escalabilidade e desempenho;
- Estética;
- Estilo.

A Arquitetura do Software é descrita no artefato denominado Documento de Arquitetura de Software, e representada por vistas dos modelos gerados. Estas vistas são subconjuntos dos modelos que contêm os elementos arquiteturais importantes.

Diversas atividades que compõem a disciplina Análise e Projeto do RUP são voltadas para a definição e validação da Arquitetura de Software. O RUP preconiza que a definição da Arquitetura de Software seja validada pela construção de um protótipo, que poderia evoluir para se tornar o produto final.

Segundo o RUP, o objetivo de definir uma Arquitetura de Software é mitigar riscos relativos a desempenho, capacidade, confiabilidade, dentre outros tipos de requisitos não funcionais, de modo que a funcionalidade do sistema possa ser adicionada na fase de construção sobre uma fundação sólida, sem risco de falhas [RUP 2002].

O outro conceito referido nesta Boa Prática é o de componente: um componente é um trecho de código que possui funcionalidade e interfaces externas claramente definidas. As linguagens e plataformas de desenvolvimento mais atuais contemplam claramente o componente como a unidade para estruturação do código executável. O conceito de componente é fortemente associado à idéia de reuso de software. O RUP enfatiza o uso de componentes, e propõe que a Arquitetura de Software seja estruturada a partir de componentes. Conceitos, tais como pacotes, subsistemas, e camadas, são utilizados durante a Análise e Projeto para organizar componentes e especificar interfaces.

Modelagem Visual

O RUP define modelagem visual como o uso de notações de projeto semanticamente ricas, gráficas e textuais, para capturar projetos de software [RUP 2002]. A notação

selecionada é a UML, que apresenta nível de abstração elevado, ao mesmo tempo mantendo uma sintaxe e semântica rigorosas. Os diversos benefícios da modelagem visual são resumidos a seguir:

- Ajuda a compreender sistemas complexos: quanto mais complexo o sistema, mais importante se torna modelá-lo graficamente.
- Explora e compara alternativas de projeto a baixo custo: o custo de projetar determinado um aspecto do sistema é substancialmente inferior ao de implementá-lo. Isso torna viável averiguar, na etapa de projeto, alternativas de soluções.
- Define a fundação para a implementação: projetar o sistema torna mais viável atingir os objetivos propostos por tecnologias como orientação a objetos e componentização – reuso, estabilidade, flexibilidade.
- Captura requisitos de forma precisa: a modelagem visual contempla também um modelo preciso para captura de requisitos, fator crítico para sucesso de projetos de software.
- Comunica decisões sem ambigüidade. Booch apresenta os seguintes objetivos para a modelagem visual [BOOCH 1995]:
    - Permite comunicar decisões não óbvias e que não podem ser inferidas a partir do código;
    - Provê uma semântica rica o suficiente para capturar decisões importantes estratégicas e táticas;
    - Oferece uma forma concreta o suficiente para o raciocínio humano e para manipulação por máquinas.

Verificação Contínua de Qualidade

A motivação da verificação contínua de qualidade em um projeto de software consiste no fato de que o custo da correção de um problema é tanto maior quanto mais avançado esteja o projeto.

Para o RUP, gerenciar qualidade significa:

- Identificar os indicadores (métricas) apropriados de qualidade;

- Identificar as medidas apropriadas a ser utilizadas para avaliar qualidade;
- Identificar e endereçar adequadamente as questões que afetam qualidade tão cedo e efetivamente quanto possível.

O conceito de qualidade é visto como multidimensional. A dimensão principal de qualidade é funcional: o sistema tem que adequadamente implementar os requisitos funcionais acordados com clientes e usuários. As outras dimensões de qualidade importantes podem depender do tipo de aplicação.

Todas as disciplinas do RUP contemplam aspectos ligados à verificação contínua de qualidade. A disciplina de Testes, em particular, é especificamente focada em gerência de qualidade.

Gerência de Mudanças

Uma das premissas mais básicas que motivam esta Boa Prática é a de que haverá mudanças ao longo do projeto, em particular – mas não exclusivamente – mudanças nos requisitos de software. É impossível, e nem seria desejável, impedir que haja mudanças, uma vez que o objetivo é entregar o sistema adequado ao uso do cliente. É necessário, porém, controlar como e quando mudanças são introduzidas nos artefatos do projeto, e quem introduz essas mudanças. Deve-se também manter uma sincronia do efeito dessas mudanças entre os diversos envolvidos com o projeto.

O RUP contempla uma abordagem denominada *Unified Change Management* (UCM – Gerência de Mudanças Unificada) para gerenciar as mudanças que ocorram ao longo do projeto de software, desde mudanças nos requisitos a erros descobertos durante a implantação que impliquem em alteração no sistema. Essa abordagem tem várias dimensões, entre as quais:

- Gerência de Requisições de Mudança, que define um processo para o registro, avaliação e priorização de solicitações de mudanças efetuadas por quaisquer interessados no sistema (usuários, testadores, programadores);
- Gerência de Configuração, que descreve a estrutura do produto e identifica os itens de configuração que o constituem, e as versões do produto geradas a partir de versões específicas dos itens de configuração.

3.3.2 Essência do RUP

A essência do RUP são as características que necessariamente têm que ser preservadas quando se especializa ao processo a uma Organização ou projeto. O RUP define dez tópicos como sendo essenciais [RUP 2002]:

- Visão – Defina uma visão clara;
- Planejamento – Gerencie a partir de um plano;
- Riscos – Mitigue riscos;
- Caso de Negócio - examine o Caso de Negócio;
- Arquitetura – Projete uma Arquitetura Componentizada;
- Protótipo – construa e teste o produto incrementalmente;
- Avaliação – avalie os resultados regularmente;
- Solicitações de Mudança – gerencie e controle mudanças;
- Suporte ao usuário – implante um produto utilizável;
- Proceso – adote o processo adequado ao seu projeto.

Visão – Defina uma Visão Clara

O RUP contempla um artefato denominado Visão, que deve ser criado no início do projeto e,se necessário, refinado e atualizado posteriormente.

A Visão, como o nome indica, contempla uma visão de alto nível do produto que se deseja construir ou fazer evoluir, incluindo, entre outras informações: uma definição do problema que o software vem resolver, as causas raízes desse problema e a proposta geral da solução; a definição do escopo do projeto; um resumo das características do software. Tem como alvo tanto os técnicos envolvidos no projeto quanto à gerência de alto nível da organização.

A Visão, juntamente com o Caso de Negócio (*business case*), é um artefato importante para a decisão de continuidade do projeto, ao final da fase de Concepção.

Planejamento - gerencie a partir de um plano

O processo destaca a importância da atividade de planejamento. O plano é importante, mas mais ainda é o esforço de planejamento e discernimento que se obtêm desse esforço, ou seja: conceber um novo projeto, avaliar escopo e risco, monitorar e controlar o projeto, planejar e avaliar cada iteração e fase.

A disciplina Gerência de Projeto contempla diretrizes, atividades e uma gama de artefatos referentes ao planejamento e acompanhamento do projeto.

Riscos – Mitigue Riscos

O RUP é dirigido por riscos. A proposta é identificar e atuar sobre os riscos maiores ainda no início do projeto; planejar cada iteração de modo a que os riscos maiores naquele momento sejam mitigados ou eliminados; reavaliar e re-priorizar os riscos ao final de cada iteração.

Essa abordagem de atuar sempre sobre os riscos maiores em cada iteração não parece intuitiva, já que as equipes de projeto geralmente priorizam efetuar as atividades sobre as quais têm maior domínio – e que, portanto, representam menos risco. Entretanto, a abordagem do RUP certamente é a mais adequada para o sucesso dos projetos de software, na medida que não posterga as questões críticas para o final do projeto, quando a exigência de cumprimento de prazos tende a ser mais intensa.

Caso de Negócio (business case) – examine o Caso de Negócio

O artefato denominado Caso de Negócio é produzido a partir da Visão, e tem como objetivo caracterizar o projeto do ponto de vista econômico. A informação essencial contida no Caso de Negócio é uma estimativa de retorno de investimento (ROI – *return of investiment*) referente ao produto a ser desenvolvido. Essa informação é fundamental para os responsáveis pela aprovação ou não do projeto. Sendo o ciclo de vida adotado o iterativo, o Caso de Negócio deve ser periodicamente reavaliado, ao longo do projeto, para verificar se as premissas iniciais continuam válidas.

Arquitetura – Projete uma Arquitetura Componentizada

Uma arquitetura executável é aquela validada pela construção de um protótipo operacional. Produzi-la é o objetivo principal da fase de Elaboração do RUP, e essencial à proposta do processo. O RUP é denominado "dirigido pela Arquitetura", para enfatizar o quão essencial é este aspecto para a proposta do processo. O tópico

Boas Práticas de Engenharia de Software também contempla um item referente à definição de uma arquitetura baseada em componentes, na qual o conceito de Arquitetura de Software é detalhado.

Protótipo – construa e teste o produto incrementalmente

Esta é a essência do ciclo de vida iterativo proposto pelo RUP. Cada iteração (exceto as iterações preliminares) produz mais um incremento da funcionalidade do sistema, que é integrada às funcionalidades previamente desenvolvidas, de modo a resolver problemas e mitigar riscos o mais cedo possível.

Avaliação – avalie os resultados regularmente

O RUP considera que uma comunicação ampla e contínua considerando dados derivados diretamente das atividades em andamento e das configurações do produto em evolução é importante em qualquer projeto. Avaliações periódicas de status provêem mecanismos para endereçar, comunicar e resolver questões gerenciais, técnicas e riscos do projeto.

A disciplina Gerência de Projeto do RUP contempla duas atividades, Avaliar Status e Avaliar Iteração, que materializam essa proposta essencial, juntamente com as atividades para definição e coleta de métricas ao longo das iterações.

Solicitações de Mudança – gerencie e controle mudanças

Este ponto já foi abordado na Boa Prática Gerenciar Mudanças. O RUP contempla uma disciplina, denominada Gerência de Mudanças e Configuração, que aborda os assuntos relativos a mudanças (de requisitos ou derivadas de erros descobertos em atividades de teste, por exemplo).

Em relação à Solicitações de Mudanças, o RUP define que todas as solicitações, tanto de melhoria quanto para correção de erros, devem ser registradas segundo um único canal de comunicação e avaliadas e priorizadas segundo um processo decisório bem definido.

O objetivo é manter sob controle a gerência do projeto e avaliar adequadamente o impacto de solicitações de mudança sobre o produto e sobre os envolvidos com o desenvolvimento.

Suporte ao Usuário – implante um produto utilizável

O produto de um projeto de software não é apenas software executável, mas também o material explicativo que o acompanha, como um Manual de Usuário, suporte interativo e, eventualmente, material de treinamento. A disciplina Implantação do RUP contempla as atividades e artefatos relativas à produção desse material.

Processo – Adote um processo adequado ao seu projeto

A adoção do processo não deve implicar em renúncia ao espírito crítico pela equipe de desenvolvimento. Com isso deseja-se enfatizar que o RUP é um framework de processo, e deve ser adaptado para se adequar às necessidades da organização, da equipe e do projeto. A disciplina Ambiente do RUP contempla diretrizes, atividades e artefatos para adaptação do processo a cada situação específica.

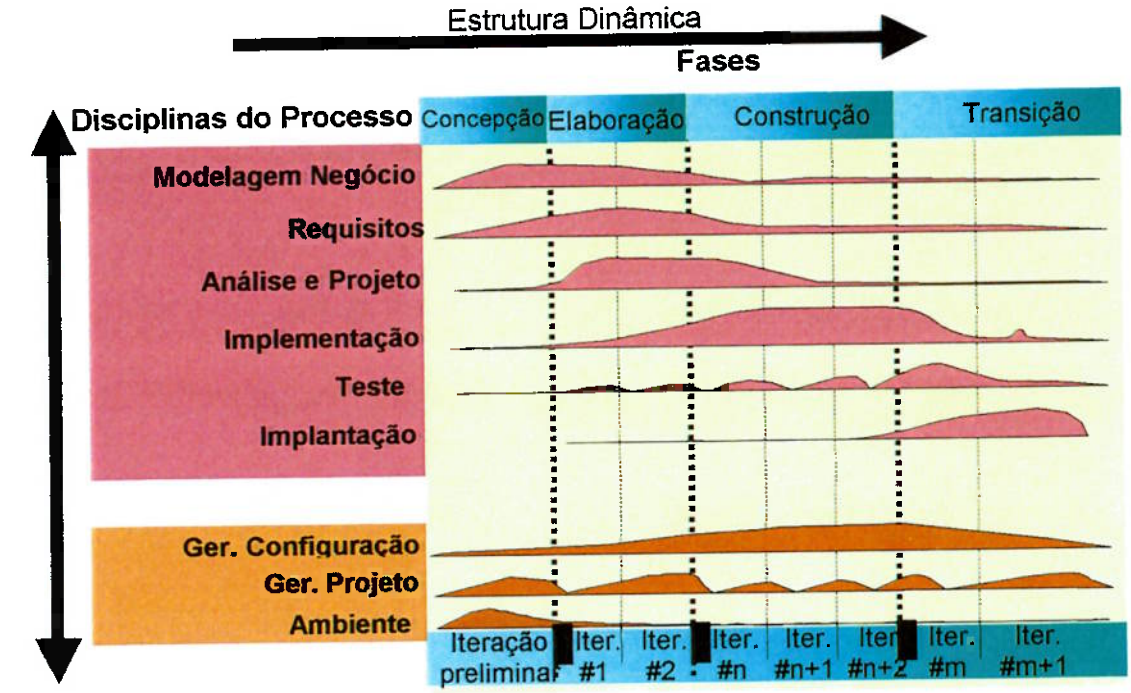
3.4 *Estrutura do Processo*

Em um processo de desenvolvimento de software que contemple um ciclo de vida em cascata, a estrutura do processo é definida pelas disciplinas que o compõem. O termo disciplina é definido como um conjunto de atividades relacionadas, concernentes a uma determinada área de interesse. Disciplinas típicas em processos de software incluem Análise, Projeto, ou Teste, entre outras.

Por adotar o ciclo de vida iterativo, o RUP é organizado em duas estruturas ortogonais:

- Uma estrutura ao longo do tempo, que define o ciclo de vida do processo, composta por fases e iterações – estrutura dinâmica;
- Uma estrutura baseada em conteúdo, composta pelas disciplinas do processo, papéis, atividades e artefatos – estrutura estática.

A figura 3.1 contempla a representação visual da estrutura do RUP. As estruturas dinâmica e estática são descritas a seguir.



3.4.1 Estrutura Dinâmica do Processo

A figura 3.1 apresenta, no eixo horizontal, a organização do RUP ao longo do tempo, quando aplicado a um projeto. Essa organização inclui fases, iterações e marcos de controle.

O RUP é composto por fases, cada uma delas delimitada por um marco de controle principal, que corresponde a um ponto de decisão sobre o projeto, do ponto de vista de negócio. Uma fase é definida como o intervalo entre dois marcos principais. São quatro fases: Concepção, Elaboração, Construção e Transição. Cada fase tem macro objetivos, e a conclusão de cada fase implica numa avaliação do projeto. Caso os objetivos da fase tenham sido alcançados, o projeto evolui para a próxima fase. Caso contrário, novas iterações terão que ser planejadas na fase atual do projeto ou, numa situação mais drástica, o projeto poderia ser cancelado.

A seguir, são descritos os objetivos e principais atividades de cada fase, e também a relação entre fases e iterações.

Concepção

A concepção é uma fase relativamente curta que, num projeto típico, demoraria de alguns dias a algumas semanas. Como o nome indica, durante a Concepção o projeto é abordado em nível macro:

- Os objetivos de negócio são definidos;
- Uma primeira avaliação da viabilidade técnica e econômica do projeto é elaborada;
- As características do projeto (requisitos expressos ainda em alto nível) são definidas em conjunto com os interessados;
- A infra-estrutura de suporte ao projeto e a especialização do processo para o projeto têm uma primeira versão definida;
- Uma primeira abordagem, ainda não muito precisa, de esforço e cronograma é elaborada;
- Uma primeira abordagem dos riscos mais altos aos quais o projeto está sujeito é desenvolvida.

Ao final da Concepção, há um marco principal do processo denominado Objetivos do Ciclo de Vida, no qual a viabilidade do projeto é avaliada e é efetuada uma decisão de continuar ou abortar o projeto.

Elaboração

A fase de Elaboração tem dois objetivos principais:

- Definir e validar a Arquitetura de Software do projeto;
- Detalhar a maior parte dos requisitos, geralmente sob a forma de descrições de Casos de Uso.

A Elaboração é também uma fase relativamente curta, em relação à fase seguinte (Construção), embora já mais ambiciosa que a Concepção, em termos de objetivos e de duração. A equipe de projeto não está ainda totalmente envolvida, já que o essencial do projeto, implementação e teste do sistema ainda não está sendo efetuado. A proposta da Elaboração é reduzir, ainda numa fase inicial, os riscos

potenciais sobre o projeto, pelo detalhamento dos requisitos funcionais, e, principalmente, pela elaboração de uma Arquitetura de Software, que será validada pela construção de um protótipo evolutivo.

Durante a parte final da Elaboração, é efetuado o planejamento detalhado da fase de Construção, a que vai consumir mais recursos do projeto.

Ao final da Elaboração há um marco denominado Marco da Arquitetura, no qual os resultados da Elaboração são avaliados – com foco principal, na Arquitetura de Software desenvolvida e implementada através do protótipo.

Construção

Durante a construção, é finalizado o detalhamento dos requisitos, e a análise, projeto, implementação e teste do aplicativo são elaborados ao longo de algumas iterações (normalmente, uma a três iterações). Todo esse trabalho é realizado tendo como base a Arquitetura de Software elaborada na fase anterior. Nesta fase aumenta a ênfase nas atividades de gerência de projetos, para garantir que a produtividade da equipe envolvida e a qualidade do produto gerado são adequadas. Normalmente, cada iteração desenvolve um subconjunto da funcionalidade do aplicativo e, ao final da iteração, a funcionalidade desenvolvida é integrada à desenvolvida anteriormente, se for o caso.

A Elaboração é concluída quando o produto é concluído e, em princípio, poderá ser implantado no ambiente de produção.

Transição

Durante esta fase, o produto é preparado para implantação no ambiente do usuário. Dependendo na natureza do produto, que pode ter sido desenvolvido para ser implantado num ambiente de produção específico de um cliente ou, no outro extremo, para venda ao mercado de varejo, as atividades da Transição vão variar bastante. No primeiro caso, o software pode ser implantado em um ambiente de teste ao qual o usuário tem acesso, para homologação. No segundo caso, versões beta podem ser produzidas para avaliação por potenciais clientes. Em qualquer caso, a documentação de usuário e diretrizes para implantação / operação serão completadas.

Caso os testes nesta fase ou a realimentação dos clientes e usuários tornem necessários, pode ser necessário alterar o código ou mesmo requisitos e o modelo de projeto.

Fases e Iterações

Cada fase é composta por um certo número de iterações. Uma iteração é uma espécie de mini ciclo cascata, na qual, em princípio, todas as disciplinas do processo (modelagem de negócio, gerência de requisitos, análise e projeto, implantação, teste, implantação) são realizadas. O foco de cada iteração, entretanto, muda ao longo do projeto, de modo que cada iteração tem seu próprio plano, que define quais atividades de quais disciplinas serão executadas.

Um ponto chave no RUP é que cada iteração produz uma versão executável do software. Essa versão pode ser apenas um protótipo (na Concepção ou Elaboração), um subconjunto do produto final apenas para uso interno pela equipe de projeto, ou uma versão para uso externo. Dessa forma, o trabalho de codificação começa cedo no projeto, e não quando o modelo de projeto da aplicação já está completo.

A concepção pode não ter nenhuma iteração, no sentido estrito, caso não seja desenvolvido um protótipo nessa fase. A elaboração costuma ter de uma a três iterações, mais comumente uma a duas – depende da complexidade da arquitetura de software e dos riscos a serem abordados. A construção costuma ter de uma a três iterações, e a Transição uma ou duas.

A proposta do RUP é que cada iteração seja desenvolvida como um com uma duração pré-definida, com duração típica de duas a seis semanas. É também fundamental que cada iteração seja avaliada contra objetivos definidos no Plano da Iteração. Desvios em relação ao plano devem ser contemplados na próxima iteração.

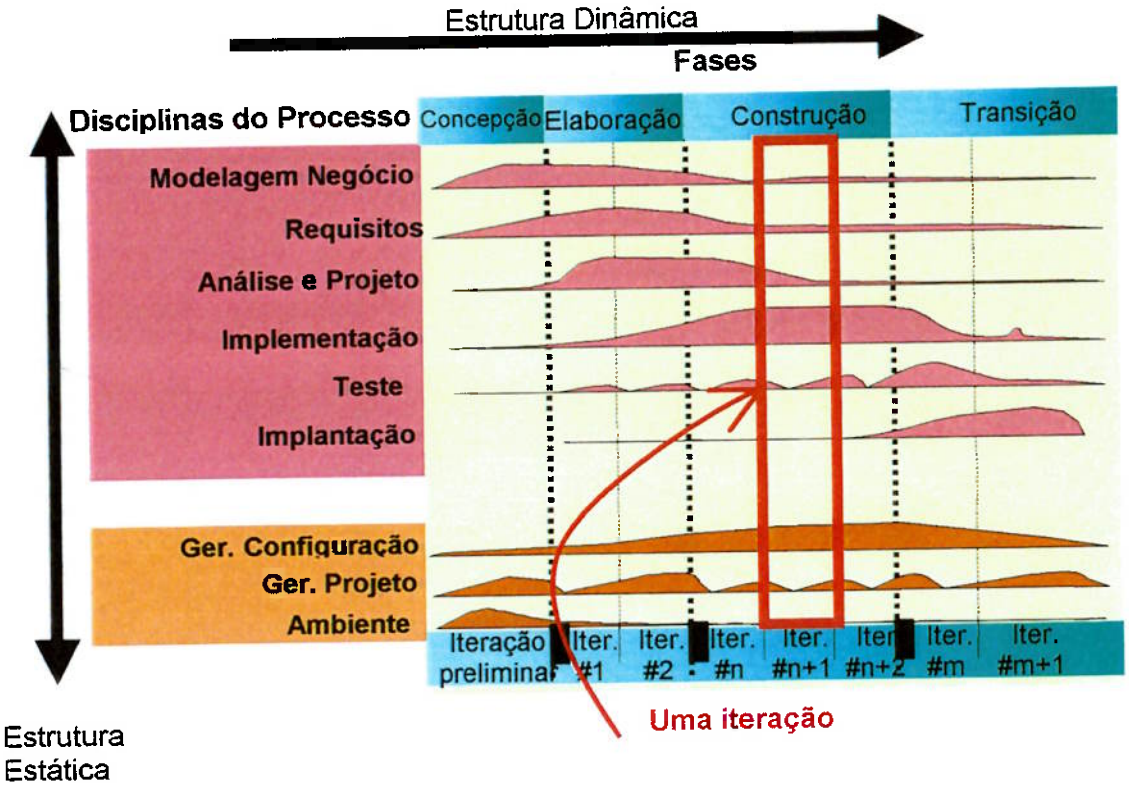


Figura 3-2 Uma iteração, no contexto do ciclo de vida

A figura 3.2 caracteriza uma iteração ao longo do ciclo de vida do projeto. As figuras em rosa pretendem demonstrar que o foco em cada disciplina muda ao longo do projeto, e uma iteração se focará mais em determinadas disciplinas e determinadas atividades em função de sua posição do ciclo de vida e da natureza do projeto.

3.4.2 Estrutura Estática do Processo

A figura 3.2 apresenta, no eixo vertical, a organização do processo no que concerne a conteúdo. Essa dimensão do processo é composta por Disciplinas, e estas são descritas em termos de *Workflows*, Papéis, Atividades e Artefatos.

Um Papel define o comportamento e a responsabilidades de um indivíduo, ou de um conjunto de indivíduos que trabalha como uma equipe [RUP 2002]. O comportamento caracteriza as atividades que são desempenhadas por quem desempenha o Papel e as responsabilidades são correspondentes àquelas definidas em relação aos artefatos produzidos por quem desempenha o Papel. Analista de Sistema, Arquiteto de Software, Gerente de Projeto são exemplos de paapéis.

Uma Atividade é aquilo realizado por alguém que desempenha um papel e que produz um resultado significativo no contexto do projeto [RUP 2002]. Uma atividade típica leva algumas horas a alguns dias para ser executada, e pode ser efetuada várias vezes (em contexto diferentes) ao longo de várias iterações. Identificar Atores e Casos de Uso, Revisar o Projeto, Executar Teste são exemplos de atividades.

Cada disciplina possui um *workflow*, que descreve a seqüência de atividades que deve ser executada para que os objetivos da disciplina sejam cumpridos. As estruturas de conteúdo (estática) e temporal (dinâmica) do RUP são ortogonais, pelo fato de o RUP adotar um ciclo de vida iterativo. Isso implica que cada disciplina é executada mais de uma vez, ao longo de um projeto e que, por outro lado , o *workflow* de uma disciplina geralmente não é executado completamente em cada iteração. Em vez disso, determinadas atividades ou passos de atividades do *workflow* são executados numa iteração específica, em função do contexto daquela iteração no ciclo de vida do projeto.

Um *workflow* define um fluxo de atividades; no contexto do RUP, é composto por elementos denominados de Detalhes de Workflow que, por sua vez, são descritos em termos de Atividades, Papéis e Artefatos. . Uma mesma atividade pode constar em mais de um detalhe de *workflow* de uma certa disciplina mas, geralmente, possui ênfases diferentes, conforme o Detalhe de Workflow ao qual pertence.

Um Artefato é um conjunto de informações que é produzido, alterado ou serve como fonte para uma atividade. Cada artefato é responsabilidade de apenas um papel; mas se um artefato for composto outros artefatos, cada artefato componente pode ser responsabilidade de vários papéis.

A figura 3.3 esquematiza as relações entre detalhes de *workflow*, atividades, papéis, artefatos.

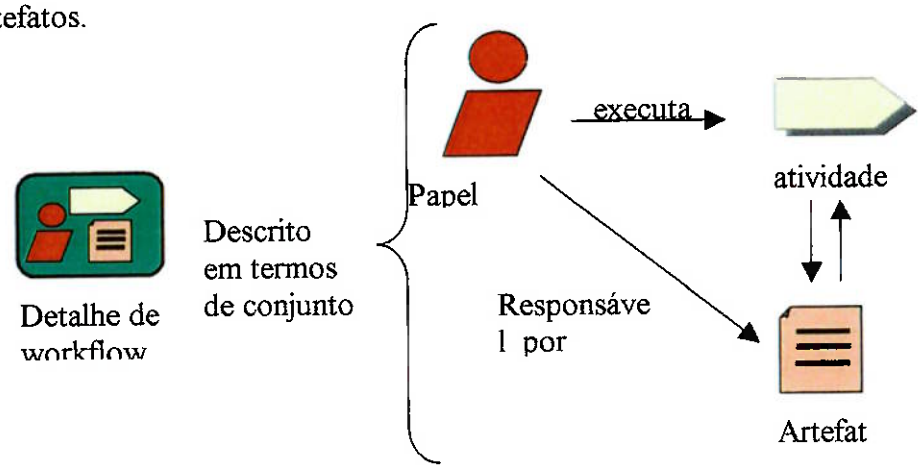


Figura 3-1 Elementos que descrevem uma disciplina do RUP

3.4.3 Elementos Adicionais do Processo

A estrutura de conteúdo do RUP contempla mais alguns elementos para complementar o conteúdo das disciplinas ou de determinados elementos das disciplinas. Esses elementos são conceitos, diretrizes e modelos.

Os **Conceitos** específicos são associados a cada disciplina e apresentam tópicos relevantes àquela disciplina. Podem conter informações relevantes para executar tarefas. Como exemplo, apresentamos conceitos explicados pela disciplina Gerência de Requisitos: Gerência de Requisitos (o conceito referente ao próprio tema da disciplina é explicado), Tipos de Requisitos, Rastreabilidade.

As **Diretrizes** são regras, recomendações, heurísticas, que dão suporte à execução de atividades e de passos de atividades. As Atividades e Passos são descritos de forma muito objetiva e concisae, se necessário, podem-se conseguir mais detalhes sobre a sua execução, acessando os conceitos correspondentes. Há diretrizes que descrevem artefatos, outras relacionadas com técnicas específicas, outras ainda dedicadas a auxiliar a avaliar a qualidade dos artefatos produzidos.

Modelos de Artefatos: O RUP contempla uma série de modelos de artefatos, principalmente de artefatos do tipo documento. Cada modelo contempla uma

estrutura de tópicos para a descrição do artefato, bem como diretrizes sobre o tipo de informação que deve ser contemplada em cada tópico.

Roadmaps, segundo o RUP, são um meio de descrever como utilizar o processo genérico descrito no RUP para resolver tipos específicos de problemas. São formas de descrever variantes – ou especializações – do processo, definindo como executar as fases especializadas ao tipo de problema em questão [RUP 2002]. O RUP contempla alguns *roadmaps*: há um *roadmap* que descreve como utilizar o RUP em projetos pequenos, um outro para soluções de *e-business*, um voltado para soluções baseadas em componentes, dentre outros.

3.5 *Disciplinas do Processo*

As disciplinas, que compõem a dimensão estática do processo, estão detalhadas nesta seção, pela sua importância no contexto do RUP. As disciplinas Análise e Projeto e Ambiente estão descritas em maior detalhe, por estarem mais relacionadas com o foco deste trabalho.

Cada disciplina do processo está associada com um ou mais modelos, que são, por sua vez, compostos por artefatos. Assim, a disciplina Modelagem de Negócio produz o Modelo de Casos de Uso de Negócio e o Modelo de Objetos de Negócio; a disciplina de Requisitos produz o Modelo de Casos de Uso; a disciplina Análise e Projeto produz o Modelo de Projeto e o Modelo de Implantação; a disciplina Implementação produz o Modelo de Implementação; a disciplina de Teste produz o Modelo de Teste.

3.5.1 Modelagem de Negócio

A disciplina Modelagem de Negócio tem como objetivo produzir um modelo da estrutura e dinâmica da Organização alvo, para:

- uma adequada compreensão da Organização;
- eventualmente, para aprimorar os processos de negócio da organização;
- em última análise, para a derivação de requisitos do sistema que automatizará (normalmente, em parte) processo(s) de negócio da Organização.

O processo aborda a modelagem do negócio de forma muito similar à adotada para a modelagem do sistema. Assim como os requisitos funcionais de software são descritos como Casos de Uso, os fluxos de trabalho dos processos de negócio da Organização são descritos através dos Casos de Uso de Negócio, usando a mesma linguagem, a mesma estrutura, mas com objetivos que transcendem o universo do software.

Na disciplina Análise e Projeto, os Casos de Uso são realizados (sua dinâmica interna descrita) através dos diagramas de classes e, principalmente, dos diagramas de seqüência da UML (também, muitas vezes, em Diagramas de Estado ou Diagramas de Atividade). Na Modelagem de Negócio, os Casos de Uso de Negócio são realizados nesses mesmos tipos de diagramas. O termo realizar significa definir a dinâmica interna (do sistema ou do negócio) em termos de objetos que se comunicam; no caso de Análise e Projeto, são os objetos de software; no caso de Modelagem de Negócio são os objetos conceituais do negócio.

O RUP contempla também a possibilidade de se fazer a modelagem do negócio de forma simplificada. Em vez de se elaborar o Modelo de Casos de Uso de Negócio e o Modelo de Objetos de Negócio , a outra possibilidade é desenvolver apenas um Modelo de Domínio, que é um conjunto de diagramas de classes conceituais, contemplando as classes, associações entre elas e atributos principais. O Modelo de Domínio funciona como um Glossário sob a forma gráfica, mas também pode ser utilizado posteriormente como fonte de informação para a modelagem do software.

3.5.2 Gerência de Requisitos

A disciplina Gerência de Requisitos tem como objetivo identificar, registrar, organizar e gerenciar requisitos ao longo do ciclo de vida do software. O termo Gerência visa destacar um aspecto fundamental: não é suficiente identificar e registrar os requisitos – os requisitos devem ser gerenciados.

A disciplina contempla a definição de tipos diferentes de requisitos, que devem ser adequadamente relacionados.Devem também ser definidos os relacionamentos entre os requisitos e os testes derivados desses requisitos, e entre requisitos e os elementos de projeto deles derivados, de modo a viabilizar análise de impacto decorrente de mudanças nos requisitos.

Esta disciplina contempla a definição de diversas técnicas para interação com clientes e usuários, para levantar os requisitos, tais como *workshops* de requisitos, entrevistas, prototipação, modelagem de negócio, entre outras.

Uma visão mais geral do sistema é capturada no artefato sugestivamente denominado Visão. Neste artefato, são definidos os principais interessados nos resultados do sistema, as fronteiras do sistema, as causas raízes do problema que o sistema deve resolver e uma descrição em alto nível da solução prevista. O ponto principal deste artefato é a definição das características do sistema, ou seja, os requisitos expressos em alto nível.

Em um segundo momento, dessas características serão derivados requisitos detalhados:

- Requisitos funcionais preferencialmente expressos como Casos de Uso;
- Requisitos não funcionais, tais como facilidade de uso, confiabilidade, performance, infra-estrutura e organização do software, descritos em um artefato denominado Especificação Suplementar.

3.5.3 Análise e Projeto

Esta disciplina é uma das centrais aos objetivos deste texto, já que as principais adaptações que o RUP deve sofrer para ser especializado para a plataforma J2EE estão relacionadas à análise e projeto e que a maioria das demais disciplinas (exceto a disciplina Ambiente) não aborda as questões diretamente ligadas à plataforma de desenvolvimento utilizada.

O RUP define três objetivos para a disciplina Análise e Projeto:

- Transformar os requisitos no projeto do sistema em desenvolvimento;
- Desenvolver uma arquitetura robusta para o sistema;
- Adaptar o projeto para que este se adapte ao ambiente de implementação, atendendo aos requisitos de desempenho.

As duas tarefas essenciais na Análise e Projeto são, portanto, a definição e a validação de uma Arquitetura de Software e a derivação de um modelo de projeto a partir dos requisitos – tanto requisitos funcionais expressos como casos de uso

quanto outros tipos de requisitos, como desempenho e confiabilidade, que influenciarão as decisões na elaboração do modelo de projeto.

Serão abordados, a seguir, os seguintes temas:

- Arquitetura de software;
- Mecanismos de arquitetura;
- Derivação de projeto a partir dos casos de uso;
- Principais papéis;
- Principais artefatos;
- Visão geral do *workflow* da disciplina Análise e Projeto;
- Diferenças entre o RUP clássico e o RUP com *plugin* J2EE.

Arquitetura de Software

O conceito de Arquitetura de Software, embora intuitivo, não tem como ser expresso de forma trivial, pois envolve múltiplos aspectos, e também existem definições diversas. Portanto, é importante posicionar este conceito para o presente trabalho.

A arquitetura é um aspecto do projeto do software (ou uma parte do projeto), que define as questões mais amplas referentes ao projeto e diz respeito à estrutura: como o software vai ser organizado em subsistemas ou componentes, que por sua vez serão decompostos em outros subsistemas ou componentes, e através de quais interfaces esses elementos irão se comunicar e prover determinados comportamentos.

A definição de uma arquitetura de software adequada é importante por vários fatores:

- Permite, à equipe, exercer um controle intelectual maior sobre o projeto, gerenciando adequadamente complexidade e garantindo maior integridade do produto;
- Define uma base para reuso do software, numa escala maior e potencialmente geradora de muito mais benefícios que apenas o reuso de classes ou componentes individuais. Também a própria arquitetura do software, ou parte dela, pode ser reutilizada em outros projetos;

- Oferece oportunidade adequada para definir como atender a requisitos de qualidade, como desempenho, disponibilidade, portabilidade, e segurança;
- É uma base para a gerência de projetos, pois a estrutura do software será fator determinante para o planejamento das atividades, atribuições de responsabilidades em função de competências e também como organização para entrega (ou entregas parciais) do produto.

O RUP representa a arquitetura de software através de um conjunto de vistas (denominado 4+1 vistas), conforma representado na figura 3.4, a seguir.

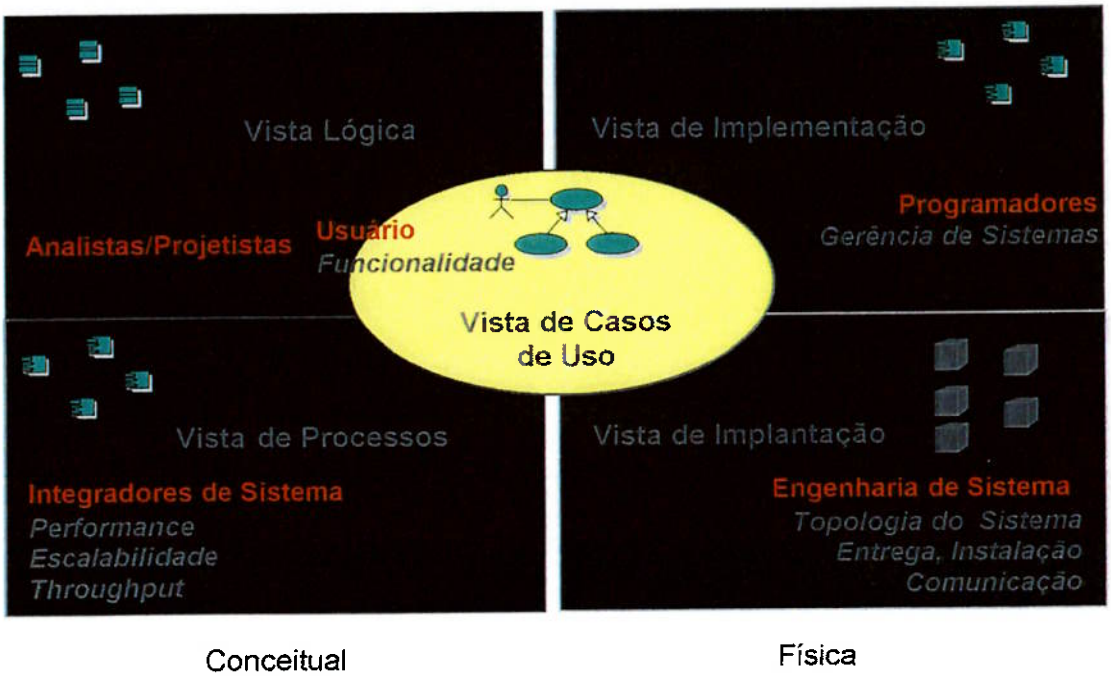


Figura 3-4 Vistas da Arquitetura de Software

A vista lógica apresenta a estrutura conceitual do sistema. É uma abstração do modelo de projeto, identificando os principais pacotes de projeto, subsistemas e classes e, eventualmente, padrões (*patterns*) de arquitetura, por exemplo, a organização do sistema em camadas.

A vista de implementação descreve a organização estática dos módulos de software no ambiente de desenvolvimento, em termos de empacotamento, camadas, e gerência de configuração.

A vista de processos aborda os aspectos concorrentes do sistema em execução: tarefas, *threads* ou processos, e suas interações.

A vista de implantação mostra como os elementos de software executável estarão distribuídos nos nós computacionais.

A vista de casos de uso contém alguns cenários ou casos de uso chave utilizados para definir e validar a arquitetura.

Mecanismos de Arquitetura

Mecanismo arquitetural é um termo genérico, para se referir a mecanismos de análise, de projeto ou de implementação. Um mecanismo caracteriza uma solução concreta para problemas encontrados freqüentemente e pode definir um padrão de estrutura, comportamento, ou ambos. Um mecanismo pode ser visto como uma definição mais informal de um *pattern*, talvez para uso em um único projeto (mas não necessariamente). Um mecanismo de análise pode ser refinado por um mecanismo de projeto, e este por um mecanismo de implementação. O primeiro será uma definição mais conceitual, os demais acrescentam detalhes progressivamente mais concretos à solução.

A disciplina Análise e Projeto enfatiza a definição de mecanismos arquiteturais, de modo a tornar coerente e simplificar o projeto do projeto. Mecanismos podem compor a vista lógica da arquitetura de software e serem expressos através de colaborações (diagramas de classe e de interação, comumente).

Um exemplo clássico de problema que demanda a definição de um mecanismo arquitetural é a persistência de objetos, pois os sistemas gerenciadores de bancos de dados que dominam o mercado adotarem uma abordagem de modelo relacional, não orientada a objetos, o que torna o mapeamento do artefato de análise ao do projeto não direto. Um outro exemplo é a comunicação entre objetos distribuídos, que envolve um grau de complexidade. Um terceiro exemplo é a definição de mecanismos para que seja preservada a identidade do cliente web entre dois acessos ao servidor, já que o protocolo de comunicação http não preserva estado.

Derivação do projeto a partir dos casos de uso

O objetivo final da disciplina Análise e Projeto é elaborar o Modelo de Projeto do sistema, a partir das descrições dos casos de uso. Os casos de uso definem o comportamento do sistema, de forma textual e os objetos implementam o

comportamento do sistema. Em tempo de execução, os objetos colaboram uns com os outros para realizar as tarefas do sistema.

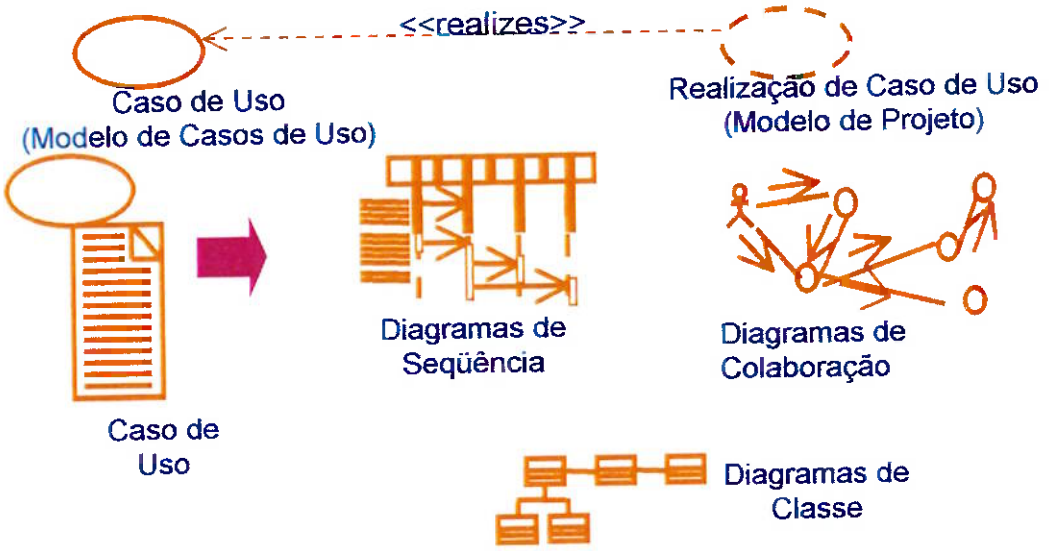


Figura 3-5 Colaborações entre classes são derivadas a partir das descrições de casos de uso

A UML contempla um elemento de modelagem denominado Colaboração que é uma sociedade de classes, interfaces, e outros elementos que trabalham juntos para prover comportamento cooperativo que é maior que a soma das partes. Graficamente, uma colaboração é representada como uma elipse com linhas tracejadas [BOOCH 1998], como pode ser visto na figura 3.5.

Normalmente, uma colaboração é utilizada para representar a dinâmica de um grupo de objetos que interagem, para prover a funcionalidade descrita por um caso de uso. Não por acaso o símbolo da colaboração é semelhante ao símbolo do caso de uso. A figura 5 caracteriza uma colaboração associada a um caso de uso através de uma relação de dependência com o estereótipo <<realizes>>. Essa colaboração é composta por diagramas de classe, colaboração e seqüência, conforme exemplificado pela figura.

Uma tarefa essencial na Análise e Projeto é definir as colaborações que vão realizar casos de uso ou cenário(s) de um caso de uso. Existe um mapeamento – de certa forma – direto entre a descrição do caso de uso e as mensagens trocadas pelos

objetos que compõem a colaboração. O RUP contempla diretrizes detalhadas para essa derivação. Essas colaborações, representadas principalmente por diagramas de seqüência, constituem a informação essencial com a qual vai trabalhar o programador para implementar o sistema.

Principais Papéis

São descritos, nesta seção, os papéis do Arquiteto e do Projetista, por serem os mais relevantes para o contexto deste trabalho.

O Arquiteto é caracterizado por Kruchten como sendo aquele que lidera e coordena as atividades técnicas e artefatos ao longo do projeto [KRUCHTEN 2000]. Ele estabelece a estrutura geral de cada vista da arquitetura: a decomposição da vista, o agrupamento de elementos e as interfaces entre os agrupamentos maiores. Em contraste com a abordagem dos outros papéis, a do arquiteto é abrangente em vez de profunda.

O Projetista (*designer*) efetua o trabalho complementar ao do Arquiteto. Ele projeta cada caso de uso (define as colaborações correspondentes aos cenários e casos de uso) e, em um grau de detalhe maior, cada classe, em termos de responsabilidade, operações, atributos e relacionamentos com outras classes.

Outros papéis que compõem a disciplina são citados a seguir. Têm responsabilidades mais específicas que os citados logo acima:

- Projetista de Banco de Dados;
- Projetista de Cápsulas (para projeto de sistemas de tempo real);
- Revisor da Arquitetura;
- Revisor do Projeto.

Principais Artefatos

Os dois artefatos chave da Análise e Projeto são o Modelo de Projeto e o Documento de Arquitetura de Software.

O Modelo de Projeto é composto por uma série de colaborações de instâncias de classes, que descrevem em detalhe o comportamento do sistema, e dos diagramas de classes correspondentes, estruturados em pacotes e subsistemas.

O Documento de Arquitetura de Software agrega as vistas que compõem a arquitetura de software e as descrições textuais que acompanham essas vistas.

Visão Geral do Workflow da Disciplina Análise e Projeto

O *workflow* definido pelo RUP para a disciplina Análise e Projeto é apresentado na figura 3.6, através de um Diagrama de Atividades em que cada elemento é um Detalhe de *Workflow*. Cada detalhe de *workflow*, por sua vez, é descrito no RUP através de papéis, atividades e artefatos. Como o objetivo é fornecer uma idéia mais geral, apresenta-se uma descrição textual dos objetivos de cada detalhe de *workflow*.

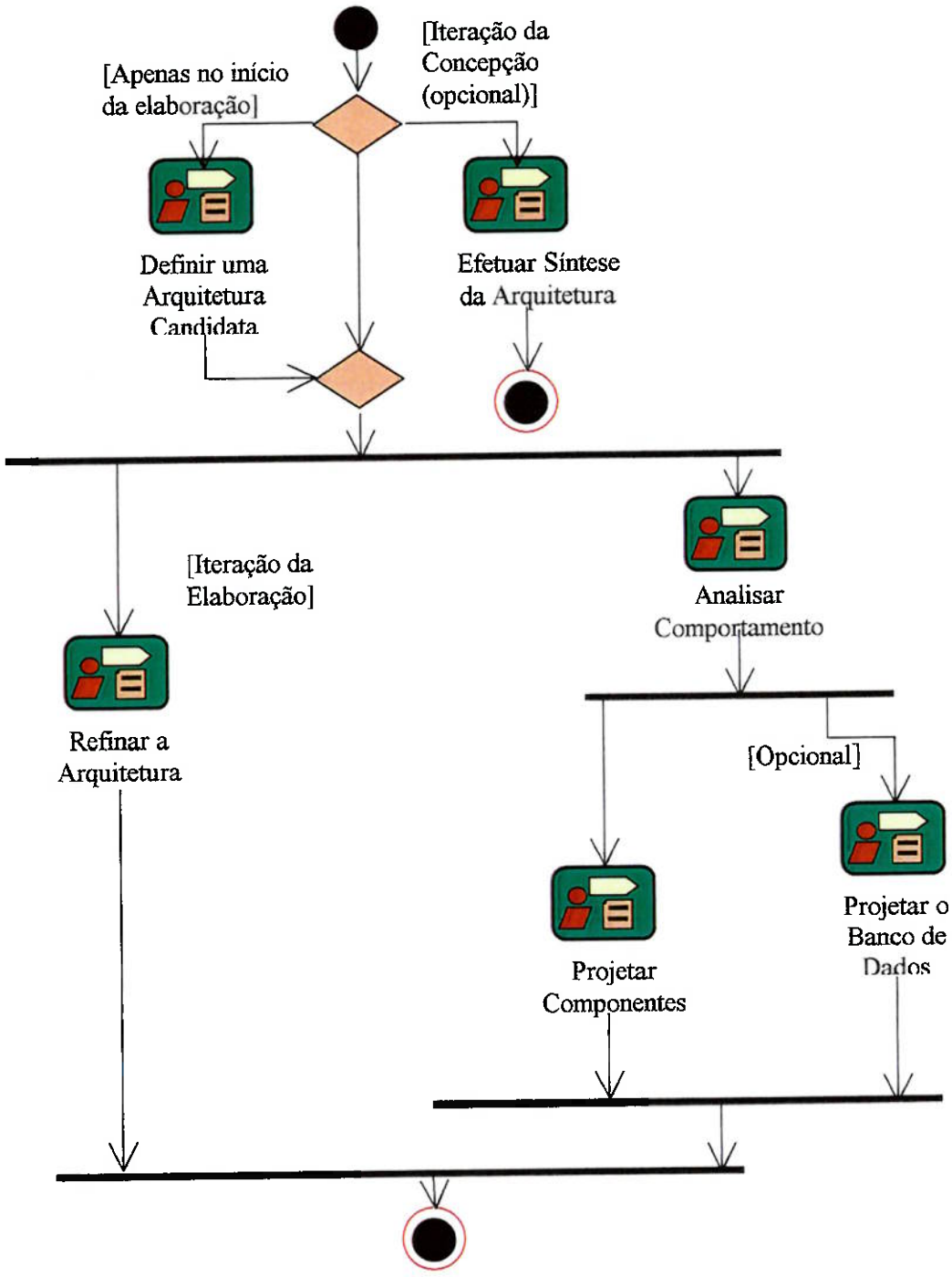


Figura 3-6 Workflow da disciplina Análise e Projeto

O detalhe de *workflow* Efetuar Síntese da Arquitetura é opcionalmente realizado durante a fase de Concepção. Visa a definição e validação de um uma prova de

conceito da arquitetura, ou seja, um protótipo que mostre a viabilidade da concepção do sistema.

O detalhe de *workflow* Definir uma Arquitetura Candidata é executado no início da fase de Elaboração e tem, como objetivo, definir um esboço da arquitetura de software, a ser refinado em iterações posteriores. Contempla a definição inicial da organização do sistema em camadas e elementos principais (subsistemas ou componentes), bem como a seleção dos primeiros casos de uso ou cenários de casos de uso a serem realizados em termos de colaborações. Além disso, uma primeira definição dos mecanismos de análise para o projeto deve ser realizada.

O detalhe de *workflow* Analisar Comportamento transforma as descrições funcionais dos casos de uso em elementos que servirão como base para o projeto do sistema. A atividade chave é Analisar Caso de Uso e o produto desta atividade é constituído pelas colaborações (diagramas de classe e de interação, principalmente), ainda no nível de análise.

O detalhe de *workflow* Refinar a Arquitetura visa efetuar a transição da análise para o projeto, definindo os elementos de projeto, a partir de elementos de análise (por exemplo, enriquecendo os diagramas de classe e interações com os elementos tipicamente de projeto). Para manter a consistência da arquitetura de software em elaboração, os elementos de projeto identificados devem ser adequadamente integrados com elementos existentes (normalmente definidos em iterações anteriores).

O detalhe de *workflow* Projetar Componentes atua claramente no nível de projeto. Visa projetar os casos de uso e as classes individuais, bem como revisar os elementos de projeto definidos.

O detalhe de *workflow* Projetar o Banco de Dados visa identificar as classes persistentes no modelo, projetar (ou identificar em um modelo) os elementos do banco de dados, que serão utilizados para armazenar as informações das classes persistentes, e definir mecanismos e estratégias para implementar a persistência.

Diferenças entre o RUP clássico e o RUP com Plugin J2EE

A maior parte das diferenças entre o RUP clássico e o RUP com o *plugin* J2EE instalado se concentram em tópicos da disciplina Análise e Projeto. O único tópico que não será abordado nesta seção, apesar de apresentar diferenças relevantes é o *roadmap* Desenvolvendo Soluções com Componentes, abordado no item *** deste texto, por não fazer parte da disciplina Análise e Projeto.

Para efeito de simplicidade, o RUP clássico será referido como RUP e a versão do RUP especializada pela incorporação do *plugin* J2EE, desenvolvido pela Rational será referida como RUP J2EE.

O RUP J2EE contempla, além das diretrizes constantes da disciplina Análise e Projeto do RUP clássico, a diretriz Visão Geral da Plataforma J2EE e a diretriz Desenvolvendo Aplicações J2EE.

A diretriz Visão Geral da Plataforma J2EE é um texto baseado na documentação J2EE produzida pela *Sun Microsystems* e, como o nome da diretriz indica, apresenta um resumo das características da plataforma J2EE e benefícios potenciais do desenvolvimento de aplicações corporativas nessa plataforma.

A diretriz Desenvolvendo Aplicações J2EE é mais interessante, do ponto de vista dos propósitos desta dissertação, pois discute alguns aspectos relativos à plataforma J2EE, tendo como referência, as definições do RUP. O RUP propõe uma organização de sistemas complexos em camadas, e a diretriz define uma organização específica em camadas para aplicações J2EE, e também a alocação dos elementos da plataforma em cada camada definida. Além disso, essa diretriz também discute alguns aspectos sobre como modelar, em UML, os EJBs da plataforma J2EE.

O documento Projetando Aplicações Java 2 Enterprise Edition, constante no item Conceitos da disciplina Análise e Projeto do RUP, é um resumo do livro *Designing Enterprise Applications for the Java 2 Platform Edition*, que descreve a plataforma J2EE e apresenta um modelo de programação com intuito de auxiliar a tomada de decisões de projeto. Não há nenhuma referência ao RUP no texto, portanto este documento não é relevante à especialização.

3.5.4 Implementação

Esta disciplina tem, como objetivo, criar o sistema executável através das seguintes atividades:

- codificar as classes organizadas em componentes de software;
- estruturar os componentes em unidades maiores, como subsistemas;
- integrar os componentes e os subsistemas;
- efetuar testes em cada classe ou componente implementado (os testes de integração e de sistema pertencem ao escopo da disciplina de testes).

  As tarefas de implementação podem começar na fase de Elaboração, quando o protótipo executável da arquitetura de software é construído, e avança ao longo das iterações da fase de Construção.

Esta disciplina contempla a definição de um Modelo de Implementação, representado através dos Diagramas de Componentes da UML. Esse modelo representa os componentes do sistema (tanto código fonte quanto executável), dependências entre componentes, sua organização em subsistemas de implementação, e a organização dos subsistemas em camadas e hierarquias. Também é representado o mapeamento entre um componente e as classes que o compõem.

3.5.5 Teste

A disciplina de Teste trata da avaliação de qualidade e presta serviços às outras disciplinas, identificando falhas.. O RUP define as seguintes práticas chave para essa avaliação [RUP 2002]:

- Identificar e documentar defeitos;
- Provar a validade das suposições feitas durante a especificação de requisitos por demonstração concreta;
- Validar as funcionalidades do software conforme projetado;
- Validar se os requisitos foram implementados da forma adequada.

Para cada tipo de requisito, definido pela disciplina Gerência de Requisitos, há uma dimensão de qualidade correspondente definida na disciplina Teste.

É virtualmente impossível testar todos os aspectos de um software; por isso, um aspecto fundamental da disciplina é a definição de uma estratégia de testes, coerente com a arquitetura e o ambiente no qual a aplicação será implantada. A versão 2002 do RUP alterou substancialmente a disciplina de Teste, e a estratégia de teste é definida com o termo Missão da Avaliação.

Definida a Missão, os testes são projetados. Um artefato denominado Caso de Teste identifica os testes a serem realizados, as condições para sua realização, os procedimentos para teste e os resultados esperados. Os Casos de Teste são elaborados a partir da especificação de requisitos. Os testes funcionais derivam diretamente dos Casos de Uso definidos na especificação de requisitos, e deve ser definido um rastreamento dos requisitos nos casos de testes correspondentes, para análise de impacto de mudanças nos requisitos.

A abordagem iterativa torna o esforço de teste, de certa forma, mais complexo pois, a cada iteração, as funcionalidades já previamente testadas devem ser novamente testadas, pois a sua integração com novas funcionalidades pode ter afetado seu funcionamento. Esses testes são denominados Testes de Regressão. Automatizar a execução de testes é extremamente importante nesse caso, para maior produtividade dos testes de regressão. Para plataformas de desenvolvimento solidamente estabelecidas, como a plataforma J2EE e .Net, o mercado fornece amplo leque de ferramentas voltadas para a gerência e automatização dos testes.

3.5.6 Implantação

O objetivo da disciplina Implantação é gerenciar as atividades que vão propiciar que o software produzido esteja disponível para os usuários. Dentre estas atividades, destacam-se:

- Implantação efetiva do software no ambiente do usuário ou, se for o caso, a disponibilização do software para instalação pelo usuário;
- Teste do software nos locais que simulem o ambiente do usuário, ou efetivamente no ambiente do usuário;
- Realização dos testes beta;

- Criação do material de suporte de usuário e, se for o caso, material de treinamento.

  Em muitas situações, durante a implantação é elaborado um procedimento de homologação do software pelo usuário / cliente.

3.5.7 Gerência de Mudanças e Configuração

Um produto de software orientado a objetos normalmente é composto por diversos componentes, onde cada componente é por sua vez um agregado de classes. Para cada classe, é necessário preservar e gerenciar tanto o código fonte quanto o executável.

Quase sempre há diversas pessoas envolvidas no desenvolvimento do software, trabalhando em paralelo. É possível que mais de um programador tenha necessidade de atuar sobre um mesmo componente, ou uma mesma classe. Há dependências entre os componentes e, para que um componente sendo desenvolvido por um programador funcione, é comum que necessite de componentes sendo desenvolvidos por outros. As funcionalidades vão sendo acrescidas ao longo do tempo, erros vão sendo corrigidos. Para que o componente A funcione, pode ser necessário associa-lo a uma versão específica do componente B, na qual um erro já foi corrigido ou uma determinada interface implementada.

Muitas vezes, há uma versão do software em produção, uma segunda versão em teste e uma terceira em desenvolvimento. A situação pode ser ainda mais complexa em determinados ambientes.

Como se vê, a questão é multifacetada. A disciplina Gerência de Mudanças e Configuração define um conjunto de ações integradas para lidar com as diversas facetas do problema central: controlar os inúmeros artefatos produzidos pelas diversas pessoas envolvidas com o desenvolvimento de software. Há três aspectos básicos envolvidos nesta disciplina: Gerência de Configuração, Gerência de Solicitações de Mudança e Avaliação de Status.

A Gerência de Configuração lida com questões de identificação de artefatos, versões e dependências entre artefatos, bem como identificação de configurações que são conjuntos consistentes de artefatos inter-relacionados. Também lida com a questão

de como prover áreas de trabalho a cada indivíduo e às equipes, de modo que os participantes possam desenvolver sem interferir no trabalho dos outros [KRUCHTEN 2000].

A Gerência de Solicitações de Mudança visa a definição de um processo consistente para o registro de requisições de mudanças no software (erros apontados pelos testadores ou programadores,ou solicitações de melhorias apontadas pelos usuários e clientes) e para a avaliação dessas solicitações. O processo prevê a definição de um Comitê para Controle de Mudanças, responsável por avaliar e priorizar o atendimento das solicitações aprovadas.

A Avaliação de Status lida com a extração de informações para a gerência de projeto, a partir das ferramentas que fornecem suporte à gerência de configuração e à gerência de solicitações de mudanças.

É muito difícil pensar na execução das atividades desta disciplina sem que haja ferramentas de software para automatizar os controles necessários, pois a execução desta disciplina é fortemente dirigida pelo uso dessas ferramentas e pelas suas características.

3.5.8 Ambiente

A disciplina Ambiente visa dotar a Organização ou um projeto específico com o processo e as ferramentas adequadas. O processo, evidentemente, será baseado no RUP, e as ferramentas são aquelas utilizadas durante o ciclo de desenvolvimento.

O RUP foi concebido como um *framework* de processo, para ser especializado para cada organização ou cada projeto específico desenvolvido na organização. Por isso, possui mais elementos do que a grande maioria dos projetos precisa utilizar.

A disciplina Ambiente define as responsabilidades e tarefas que devem ser desempenhadas para que a adaptação do RUP à Organização ou projeto seja efetuada adequadamente. Inicialmente o RUP deve ser especializado para uma Organização e, depois, para cada tipo de projeto a ser realizado.

O papel principal nesta disciplina é desempenhado pelo Engenheiro de Processo. Cabe a ele conduzir as atividades de especialização do processo, atuando em

conjunto com o gerente de projeto, quando for o caso de uma adaptação voltada especificamente para um projeto.

Quando o escopo da implantação do processo for para uma organização, a primeira tarefa do Engenheiro de Processo é efetuar uma avaliação da Organização alvo, para identificar até onde, do ponto de vista de adesão ao processo, a Organização quer ou pode ir. É comum que a implantação do processo em uma Organização seja efetuada em fases, para suavizar o impacto das mudanças sobre a equipe, e diminuir resistências. A abordagem mais comum na implantação consiste em desenvolver um projeto piloto, ao longo do qual o processo será simultaneamente especializado para a organização e para o projeto.

A tarefa essencial prevista na disciplina Ambiente é a elaboração de um artefato denominado Caso de Desenvolvimento. Esse artefato identifica os elementos do RUP que serão utilizados pela organização ou pelo projeto. A essência do Caso de Desenvolvimento é a definição dos artefatos que serão utilizados no projeto (ou projetos), as fases nas quais os artefatos serão utilizados e o grau de formalidade de cada artefato, em cada fase. A outra informação essencial do Caso de Desenvolvimento é a definição do *workflow* que será adotado em cada disciplina. O propósito não é reescrever o RUP, mas apontar simplificações ou divergências em relação ao processo clássico, que serve como referência.

Há exemplos de situações em que o processo foi implantado com sucesso em organizações que desenvolvem software segundo o paradigma de análise essencial, embora o RUP seja voltado exclusivamente para desenvolvimento orientado a objetos. Nesses casos, a disciplina Análise e Projeto foi devidamente adequada para desenvolvimento não orientado a objetos. Diretrizes e mesmo atividades tiveram que ser profundamente alteradas, mas a essência do processo foi mantida, uma vez que diversas disciplinas não são diretamente afetadas pelo paradigma de desenvolvimento adotado.

Outra tarefa importante, além da elaboração do Caso de Desenvolvimento, é a definição de diretrizes específicas relacionadas a cada disciplina do RUP. Sempre, a proposta é descrever apenas as divergências em relação ao processo clássico. O RUP já contempla diretrizes referentes a todas as disciplinas, mas o processo da

Organização ou do projeto pode demandar diretrizes diferentes ou mais especializadas.

Desta forma, o analista de negócio preparará as diretrizes específicas para modelagem de negócio; o arquiteto de software, as diretrizes de projeto e de implementação; o analista de sistemas, as diretrizes para modelagem dos casos de uso; o projetista de teste, diretrizes de teste, e assim por diante.

Um papel importante contemplado pela disciplina Ambiente é o de Especialista em Ferramentas: este é responsável pela definição e implantação das ferramentas que comporão os ambientes de desenvolvimento e teste.

3.5.9 Gerência de Projeto

O RUP introduz a disciplina Gerência de Projetos através da seguinte declaração: "Gerência de Projeto de Software é a arte de equilibrar objetivos conflitantes, gerenciar riscos e superar restrições para entregar com sucesso um produto que atenda às necessidades do cliente e dos usuários. O fato de que apenas poucos projetos são indiscutivelmente bem sucedidos é comentário suficiente sobre a dificuldade da tarefa".

O conteúdo e a organização da disciplina são fortemente influenciados pelo fato do processo adotar o ciclo de vida iterativo. A disciplina foca em gerência de riscos; planejamento do ciclo de vida como um todo e de cada iteração em particular; monitoramento do andamento do projeto e definição, coleta e análise de métricas.

3.6 *O RoadMap Desenvolvendo Soluções de Componentes*

O *roadmap* Desenvolvendo Soluções com Componentes. existe no RUP clássico, e é adaptado no RUP J2EE para contemplar as definições voltadas especificamente para componentes no universo J2EE. É estruturado em função das fases do RUP, e descreve, para cada detalhe de *workflow* de cada disciplina – ao longo das fases do projeto –, quais alterações ou especializações devem ser efetuadas no processo voltado para desenvolvimento J2EE.

Esse *roadmap* é bastante sucinto, mas seu conteúdo vai de encontro aos objetivos deste texto, e pode ser considerado como ponto de partida para a especialização do RUP para a plataforma J2EE.

3.7 Considerações finais sobre o RUP

Compreender o RUP exige um razoável esforço intelectual porque a quantidade de informação é muito grande,com mais de 40 megabytes de informação, sobre um amplo leque de tópicos associados àengenharia de software. Como a descrição é muito mais completa do que seria necessário para um projeto convencional, é necessário conhecer a sua estrutura e os conceitos básicos para saber consultar o seu conteúdo.

O RUP provê os meios para que o interessado entenda a essência do processo, caracterizando as seis boas práticas que implementa e os dez pontos essenciais que o embasam. Ainda assim, existe uma barreira a ser superada para que se possa praticar o processo como ferramenta para atingir o sucesso de projetos de desenvolvimento.

Para orientar o gerente de software e o engenheiro de processo, existem diretrizes muito claras, e diversos exemplos de como especializar o processo para as necessidades específicas de uma organização ou projeto, considerando apenas aqueles elementos que sejam efetivamente úteis para atingir o sucesso, e desconsiderando os demais. O grande desafio de utilizar o RUP é conseguir desenvolver a habilidade de especializar o processo no grau adequado, não adotando uma visão burocrática de que tudo o que está definido no processo deve ser executado, nem, por outro lado, desconsiderando aspectos que constituem a essência do processo.

4. Especialização do RUP para a plataforma J2EE

4.1 *Introdução*

A especialização do RUP, para determinado ambiente ou projeto, faz parte da implantação do RUP. A disciplina Ambiente, descrita no capítulo 3, contém as diretrizes, as atividades, os artefatos e as responsabilidades referentes à especialização.

Duas ações principais fazem parte da disciplina ambiente: a elaboração de um artefato denominado Caso de Desenvolvimento, que define quais artefatos e atividades serão incluídos no processo especializado, e a elaboração de um conjunto de diretrizes específicas para a execução de determinadas disciplinas.

Não é objetivo desse trabalho elaborar um Caso de Desenvolvimento para a plataforma J2EE. O RUP J2EE oferece uma solução através do *roadmap* Desenvolvendo Soluções Baseadas em Componentes para J2EE, organizado em função das fases do projeto (concepção, elaboração, construção, transição) e, em cada fase, apresenta definições quanto a atividades e artefatos. O *roadmap* é sucinto, mas suficiente.

Já a elaboração de diretrizes específicas é foco deste trabalho. A disciplina Ambiente prevê que diversas diretrizes sejam elaboradas e, entre as diversas diretrizes previstas (diretrizes para modelagem de casos de uso, diretrizes de teste, diretrizes de implementação, dentre outras), este trabalho foca as Diretrizes de Projeto.

O produto do trabalho é o artefato Documento de Arquitetura de Software parcialmente escrito (abordando, em linhas gerais, a proposta de arquitetura de software para um sistema genérico na plataforma J2EE) e um artefato Diretrizes de Projeto, contendo as diretrizes referentes a boas práticas de engenharia de software específicas para J2EE. O primeiro artefato está o escopo da própria disciplina Análise e Projeto, enquanto o segundo é elaborado pela disciplina Ambiente e utilizado pela disciplina Análise e Projeto.

4.1.1 Organização do Capítulo

Antes de abordar o tema da especialização do RUP para J2EE , será apresentada uma série de tópicos introdutórios, que contêm os fundamentos para a especialização propriamente dita.

Primeiramente, são apresentados alguns conceitos básicos, que descrevem os elementos de modelagem orientada a objetos, usando notação UML, necessários para a compreensão do capítulo.

A seguir, é apresentado um tópico sobre *Patterns*, assunto essencial desta dissertação por se constituírem nos blocos de construção para a especialização.

Por fim, é descrita a especialização do RUP para J2EE, centrada em aspectos da disciplina Análise e Projeto.

4.2 *Conceitos Básicos*

Este tópico descreve alguns elementos da linguagem UML cujo conhecimento é necessário para a compreensão adequada deste capítulo. São abordados: diagramas de interação (de seqüência e de colaboração), o conceito de colaboração e o conceito de estereótipo.

Um **diagrama de seqüência** é um elemento de modelagem da UML, definido pelo RUP como um padrão de interação entre objetos, organizado em ordem cronológica; mostra os objetos participantes da interação através de suas linhas de vida, e as mensagens que eles enviam uns aos outros. A figura 4.1 apresenta um exemplo de diagrama de seqüência.

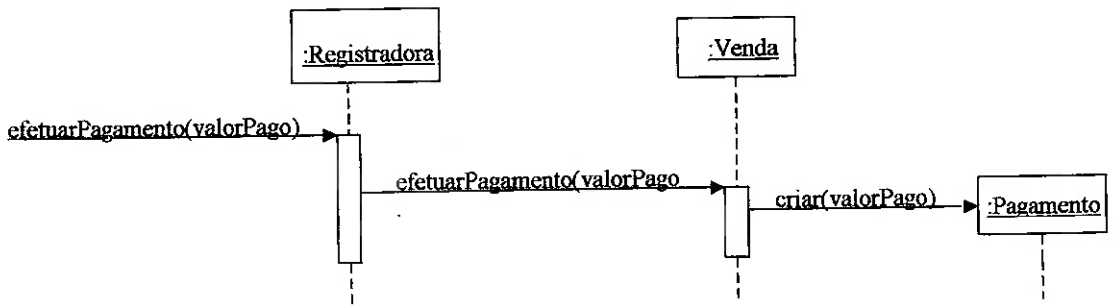


Figura 4-1 Exemplo de diagrama de seqüência

Um **diagrama de colaboração**, também um elemento de modelagem da UML, é definido como um diagrama que descreve um padrão de interação entre objetos; mostra os objetos participantes da interação, os *links* (instâncias de associações) entre os objetos e as mensagens que eles enviam uns aos outros, fluindo através desses *links*. A figura 4.2 apresenta um exemplo de diagrama de seqüência.

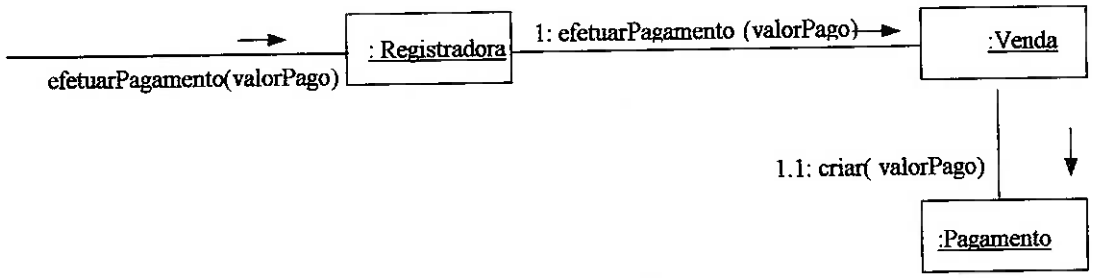


Figura 4-2 Exemplo de diagrama de colaboração

Os diagramas de seqüência e de colaboração são referidos conjuntamente pelo nome **diagramas de interação**. Eles são, na prática, intercambiáveis e uma ferramenta CASE é capaz de gerar um diagrama de seqüência dado um diagrama de colaboração, e vice versa. A UML define os dois tipos de diagramas porque eles enfatizam aspectos distintos da interação entre objetos. O diagrama de colaboração enfatiza os relacionamentos entre objetos, enquanto que o diagrama de seqüência enfatiza a seqüência de mensagens.

Uma **colaboração** é um elemento de modelagem da UML que descreve uma coleção de objetos (instâncias de classes) que interagem para implementar um certo comportamento, num determinado contexto. Uma colaboração tem uma parte estática e uma parte dinâmica. A parte estática descreve os papéis que os objetos e links desempenham na instanciação da colaboração. A parte dinâmica consiste de uma ou mais interações (descritas em diagramas de seqüência ou colaboração) na instanciação da colaboração.

A colaboração pode ser expressa, dentre outros elementos, por diagramas de colaboração.

O RUP é dirigido por casos de uso. Isso significa que o Modelo de Casos de Uso, além de ser útil para definir, em conjunto com clientes e usuários, as funcionalidades do sistema de forma detalhada, também serve como referência para projeto, implementação e definição dos testes da aplicação. Na prática, pode-se dizer que a cada caso de uso, estará associada uma colaboração, descrevendo as classes que colaboram para implementar a funcionalidade descrita no caso de uso, bem como as interações entre instâncias dessas classes que representam a dinâmica do sistema para realizar o caso de uso.

Um **estereótipo** é um mecanismo de extensão da UML. É um recurso definido pela linguagem para que se possa definir um novo elemento de modelagem a partir de um elemento de modelagem previamente definido pela UML. Esse novo elemento de modelagem particulariza a semântica do elemento de modelagem que lhe deu origem. A UML já dispõe de um conjunto de estereótipos pré-definidos, mas os usuários da linguagem podem definir novos estereótipos, para atingir objetivos específicos.

O RUP distingue **classes de análise** de **classes de projeto**. Durante a análise, são identificadas as classes em um plano conceitual, que representam os elementos no sistema que têm responsabilidades e comportamento. Classes de análise capturam uma primeira versão do modelo de objetos do sistema, e focam requisitos funcionais. Posteriormente, durante o projeto, o modelo é revisto e enriquecido com considerações sobre requisitos não funcionais e sobre o ambiente de implementação. As classes de análise tendem a evoluir para se tornar classes de projeto. Algumas classes de projeto serão derivadas diretamente de classes de análise. Outras serão definidas por contas de requisitos não funcionais. Algumas classes de análise podem se tornar subsistemas, e outras podem simplesmente desaparecer.

4.3 *Patterns*[1]

O RUP define um *pattern* como um modelo de solução para um problema recorrente, que se provou útil num certo contexto. Bons *patterns* resolvem com sucesso as forças conflitantes que afetam o problema [RUP 2002].

Buschmann enfatiza que os patterns constituem uma forma de se valer da experiência coletiva de engenheiros de software hábeis [BUSCHMANN 1996].

O *Pattern* é expresso através de um nome, da descrição de um problema e da solução para esse problema. Geralmente a definição do *pattern* também caracteriza o problema em termos de forças, que são fatores correlacionados ao problema, e que o contextualizam, e descreve os benefícios e contra-indicações do uso do *pattern*.

Freqüentemente a descrição textual do *pattern* é completada por uma visão gráfica, que representa a estrutura e/ou o comportamento dos elementos que o compõem. Quando se trata de *patterns* de projeto, a representação da estrutura é elaborada em diagramas de classe, e a representação do comportamento em diagramas de interação (colaboração ou seqüência). Usa-se, no contexto do trabalho, a terminologia UML para a representação gráfica, apesar desta tecnologia não existir, quando os primeiros trabalhos sobre patterns foram escritos. Entretanto, esses trabalhos utilizam linguagens gráficas antecessoras da UML, e semelhantes a esta, de modo que mesmo alguém que só conheça a notação UML não terá dificuldades para entender os diagramas dos patterns originais.

Patterns efetivamente são um recurso muito poderoso para contribuir para o incremento da qualidade de projetos de software. Tendo um projetista de software dominado os preceitos básicos da programação orientada a objetos, o próximo passo é elaborar projetos que efetivamente concretizem objetivos mais amplos de engenharia de software (facilidade de evolução das aplicações, componentes reutilizáveis, maior produtividade, entre outros). O uso de *patterns* potencializa a evolução do projetista nesse sentido. Pode-se mesmo de dizer que, ao projetar

---

[1] O termo pattern evidentemente poderia ser traduzido como *padrão*, mas já se tornou parte do jargão da engenharia de software o uso desse termo em inglês.

software com base em *patterns*, o trabalho do projetista passa a ser menos empírico e mais científico.

Os conceitos sobre *patterns* começaram a se popularizar a partir de um livro lançado em 1994, denominado *Design Patterns* [GAMMA 1994]. Esse livro define vinte e três *patterns* de projeto, classificados como de criação, estruturais e comportamentais. O livro se tornou um marco sobre o assunto, pelo pioneirismo e pela qualidade – os *patterns* nele definidos efetivamente são fundamentais. Os seus quatro autores passaram a ser conhecidos, curiosamente, como Gangue dos Quatro, ou GoF (de Gang of Four). Esse rótulo se estendeu para o livro, e também para os *patterns*, que hoje são amplamente conhecidos como *patterns* GoF.

Outra fonte importante para este trabalho é representada pelos *patterns* denominados GRASP (para atribuição de responsabilidades gerais – *general responsibities assignment software patterns*), definidos por Larman [LARMAN 2001]. São *patterns* que definem princípios muito básicos relacionados à atribuição de responsabilidades a classes e são mais básicos que os *patterns* GoF, também abordados em [LARMAN 2001].

Em um grau de abstração mais alto que os *patterns* de projeto, surgiram os *patterns* de arquitetura. Buschmann define *patterns* de arquitetura como *patterns* que expressam esquemas de organização estrutural fundamental para sistemas de software. Provêem um conjunto pré-definido de subsistemas, especificam suas responsabilidades, e incluem regras e diretrizes para organizar a relação entre eles [BUSCHMANN 1996].

Em um patamar mais especializado, esta dissertação também aborda *patterns* de projeto elaborados especificamente para a plataforma J2EE. A fonte é [DEEPAK 2001], livro cuja autoria é de funcionários da própria Sun Microsystems, a companhia que concebeu a plataforma J2EE. Esses *patterns* têm como objetivo preencher o *gap* semântico entre os elementos que compõem a plataforma J2EE e o projeto de aplicações J2EE.

Nos tópicos seguintes, os patterns relevantes a este trabalho são apresentados, seguindo a formato típico de descrição de um pattern.

4.3.1 Patterns de arquitetura descritos no RUP

Neste tópico e no seguinte, são apresentados todos os patterns de arquitetura propostos por Buschmann [BUSCHMANN 1996]. Este tópico descreve dois patterns Buschmann que são apresentados no RUP: o pattern Layers e o pattern Blackboard. O tópico seguinte descreve os demais patterns de arquitetura Buschmann, não descritos no RUP. A distinção em dois tópicos tem o objetivo de destacar o fato de que o RUP contempla a descrição de determinados patterns, e identificá-los.

Layers

Contexto

Um sistema grande que requer decomposição.

Problema

Um sistema que deve lidar com questões em níveis diferentes de abstração. Por exemplo: questões de controle de hardware, questões ligadas a serviços comuns e questões específicas do domínio. É indesejável escrever componentes verticais que tratam de questões em todos os níveis, pois mesma questão teria que ser tratada múltiplas vezes em componentes distintos, podendo introduzir inconsistências..

Forças

- Partes do sistema devem ser substituíveis
- Alterações nos componentes não devem se propagar
- Responsabilidades similares devem ser agrupadas
- Uniformização do tamanho dos componentes – componentes complexos podem ter que ser decompostos

Solução

Estruture o sistema em grupos de componentes que formam camadas umas sobre as outras. Faça com que as camadas superiores utilizem serviços apenas das camadas abaixo delas (nunca acima). Tente não utilizar serviços além daqueles da camada imediatamente inferior (não pule camadas, a menos que as camadas intermediárias apenas adicionem componentes de passagem).

Comentários

O número de camadas que o sistema irá ter é variável, e depende do grau de complexidade do sistema. O RUP apresenta dois exemplos, um no qual o sistema é organizado em quatro camadas, outro no qual o sistema é organizado em cinco camadas , conforme pode ser visto na figura 4.3.

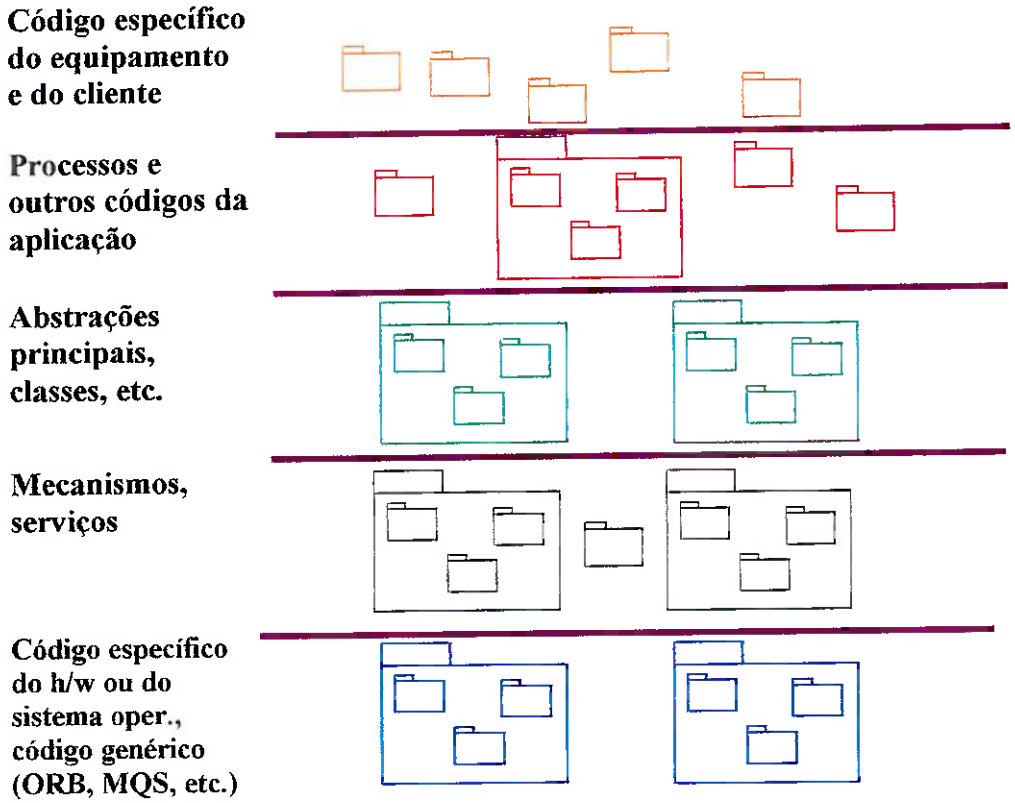


Figura 4-3 Exemplo de arquitetura em camadas oriundo do RUP

Blackboard

Contexto

Um domínio no qual nenhuma solução fechada (algorítmica) para resolver o problema é conhecida ou viável. Exemplos são sistemas de inteligência artificial, reconhecimento de voz ou sistemas de vigilância.

Problema

Múltiplos agentes devem cooperar para resolver um problema que não pode ser resolvido por agentes individuais. O resultado do trabalho dos agentes individuais deve ser acessível a outros agentes, de modo que estes possam avaliar se podem contribuir para encontrar uma solução e publicar os resultados do seu trabalho.

Forças

- A seqüência na qual os agentes podem contribuir para resolver o problema não é determinística e pode depender das estratégias para resolução do problema
- Informações oriundas de agentes diferentes (soluções parciais ou resultados) podem ter representações diferentes
- Agentes não sabem diretamente sobre a existência de outros, mas podem avaliar as contribuições publicadas pelos outros.

Solução

Diversos agentes têm acesso ao sistema de armazenamento de informações denominado *Blackboard* (quadro negro). O quadro negro provê uma interface para inspecionar e atualizar seu conteúdo. O módulo/objeto de controle ativa os agentes seguindo alguma estratégia. Quando ativado, o agente inspeciona o quadro negro para ver se pode contribuir para a solução do problema. Se o agente determinar que pode contribuir, o objeto de controle pode permitir que ele registre sua solução parcial ou final no quadro.

4.3.2 Outros patterns de arquitetura

A seguir, será apresentada uma descrição dos demais patterns de arquitetura descritos por [BUSCHMANN 1996] e não descritos pelo RUP. Alguns são descritos em maior detalhe que outros, com base na relevância do pattern para este trabalho.

São descritos os seguintes patterns:

- Pipes and Filters
- Model-*View*-Control
- Presentation-Abstraction-Control
- Microkernel

- Reflection

Pipes and Filters (tubos e filtros)

Esse *pattern* de arquitetura provê uma estrutura para sistemas que processam um fluxo contínuo de dados. Cada passo do processamento é encapsuladoem um componente filtro e o dado é passado através de tubos entre filtros adjacentes. A recombinação de filtros permite que se construa famílias de sistemas relacionados.

Exemplo.: um compilador, para uma determinada linguagem de programação, que efetua diversos níveis de análise (léxica, sintática, semântica, etc.) do código do programa.

Broker

Pode ser utilizado para estrutura sistemas distribuídos com componentes desacoplados que interagem através de invocação de serviços remotos. Um componente denominado *broker* (corretor, intermediário) é responsável por coordenar a comunicação, transmitindo requisições, resultados e mensagens de exceção.

Model-View-Controller (MVC)

Este *pattern* é citado de forma muito sucinta na versão do RUP que contém o *plugin* para J2EE, no texto Conceitos: Resumo do Esquema J2EE ("*Concepts*: J2EE *Blueprints Digest*"), como sendo um *pattern* de projeto, embora [BUSCHMANN 1996] o qualifique mais adequadamente como um *pattern* de arquitetura para sistemas interativos.

O *pattern* propõe a divisão de uma aplicação interativa em três componentes. O modelo (*model*) contém a funcionalidade central e dados;. As vistas (*view*) exibem informações aos usuários; os controladores (*controllers*) tratam as ações dos usuários. Em conjunto, *views* e *controllers* abrangem a interface com o usuário. Um mecanismo de propagação de mudanças garante a consistência entre a interface com o usuário e o modelo.

Devido à sua importância para os objetivos deste texto, esse *pattern* é mais detalhado.

Contexto

Aplicações interativas com uma interface homem/máquina flexível.

Problema

A interface com o usuário é especialmente passível de mudanças. Por conta disso, a interface deve ser flexível, mas distinta da funcionalidade do sistema que não muda tão freqüentemente.

Forças

- A mesma informação pode ser apresentada diferentemente em diferentes janelas.
- A interface visual e o comportamento da aplicação devem refletir mudanças nos dados imediatamente.
- Deve ser simples alterar a interface com o usuário – se possível, em tempo de execução.
- Estilos diferentes da interface com o usuário não devem afetar o código do núcleo da aplicação.

Solução

Estruture a aplicação em três áreas diferentes: processamento, entrada e saída. O **modelo** (*model*) encapsula funcionalidades e dados da aplicação. O modelo é independente da representação da saída e do comportamento da entrada. A **vista** (*view*) exibe informações ao usuário. A vista obtém seus dados do modelo. O **controlador** (*controller*) recebe as entradas, geralmente como eventos que são traduzidos em requisições de serviço para a vista ou para o modelo.

Presentation-Abstraction-Control

Este *pattern* define uma estrutura para sistemas interativos sob a forma de uma hierarquia de agentes cooperativos. Todo agente é responsável por um aspecto específico da funcionalidade da aplicação, e consiste de três componentes: apresentação, abstração e controle. Esta subdivisão separa os aspectos da interação homem-máquina dos agentes de seu núcleo funcional e de sua comunicação com outros agentes.

Microkernel

Este *pattern* se aplica a sistemas que devem estar aptos a se adaptar a requisitos de sistema mutáveis. Ele separa um núcleo funcional mínimo de funcionalidade estendida e partes específicas do cliente. O *microkernel* serve como um *socket* (soquete) para conectar essas extensões e coordenar sua colaboração.

Reflection

Este *pattern* provê um mecanismo para mudar dinamicamente a estrutura e o comportamento de sistemas. Suporta a modificação de aspectos fundamentais, tais como estrutura de tipos e mecanismos para chamadas de funções. Neste *pattern*, uma aplicação é dividida em duas partes. Um meta-nível provê informação sobre propriedades selecionadas do sistema e torna o software ciente sobre si mesmo. O nível base inclui a lógica da aplicação. Sua implementação é construída sobre o meta-nível e as mudanças nas informações mantidas no meta-nível afetam o comportamento subseqüente do nível base.

4.3.3 Patterns de Projeto

Os dois tópicos a seguir descrevem os patterns de projeto definidos por Larman (GRASP) e os patterns GoF.

Patterns GRASP

Esses *patterns* são denominados por Larman como GRASP, sigla em inglês para *Patterns* de Software para a Atribuição Geral de Responsabilidades (de General Responsibility Assignment Software Patterns). Como o nome indica, definem princípios gerais referentes à atribuição de responsabilidades a classes. Dois deles – Alta Coesão e Baixo Acoplamento – apresentam os conceitos fundamentais sobre orientação a objetos sob a forma de patterns [LARMAN 2001].

Estes *patterns* são apresentados por serem referências para a definição de *patterns* mais especializados e complexos, inclusive alguns de projeto específicos para a plataforma J2EE.

São apresentados os seguintes patterns GRASP:

- Information Expert;

- Creator;
- Baixa Coesão;
- Alto Acoplamento;
- Controller.

Pattern Information Expert (*expert* em informação)

Solução

Atribuir a responsabilidade ao *expert* da informação – a classe que tem a informação necessária para atender à responsabilidade.

Problema

É necessário desejável definir um princípio geral para atribuição de responsabilidades a objetos.

Comentários

Freqüentemente usado na atribuição de responsabilidades; é um princípio básico do projeto orientado a objetos. Expressa uma intuição comum de que objetos realizam ações relacionadas com as informações que eles têm.

Nem sempre é conveniente aplicá-lo, principalmente em situações em que isso implicaria em reduzir a coesão e aumentar o acoplamento no sistema – mas nas situações mais comuns, ocorre justamente o contrário.

Creator (criador)

Solução

Atribuir à classe B a responsabilidade de criar uma instância da classe A, se uma ou mais das seguintes condições for verdadeira:

- B agrega objetos de A
- B contém objetos de A
- B grava instâncias de objetos de A
- B utiliza fortemente objetos de A

- B tem os dados de iniciação que serão passados para um objeto de A quando este for criado (então B é um *Expert* no que diz respeito à criação de A)
- Se mais de uma dessas opções for verdade, é preferível que a classe B agrega ou contém a classe A.

Problema

Definir critérios para a atribuição da responsbilidade por por criar uma nova instância de determinada classe.

Comentários

O propósito básico do Creator é encontrar um criador que precisa estar conectado ao objeto criado em qualquer evento. O acoplamento não é aumentado porque o objeto criado provavelmente já seria visível pelo Creator, por conta das associações existentes. Há outros *patterns* para definição de responsabilidade de instanciação de objetos, quando Creator não se mostra adequado.

Baixo Acoplamento

Solução

Definir uma responsabilidade de forma que o acoplamento permaneça baixo.

Problema

Atribuir responsabilidades às classes de modo que haja baixa dependência, baixo impacto de mudanças e facilidade de reuso.

Definição de acoplamento

Medida de quão forte um objeto está conectado a outro, ou tem conhecimento de outro, ou precisa contar com outro. Um elemento que tem baixo acoplamento não é dependente demais de outros elementos; o que significa demais depende do contexto.

Alta Coesão

Solução

Atribuir uma responsabilidade de modo que a coesão permaneça alta.

Problema

Atribuir responsabilidades de modo a manter a complexidade gerenciável.

Comentários

Em termos de projeto de classes, a **coesão** é uma medida de quão fortemente relacionadas e focadas são as responsabilidades de um elemento. Um elemento com responsabilidades altamente relacionadas, que não executa uma quantidade de trabalho muito grande, tem alta coesão.

Controller

Solução

Atribuir a responsabilidade por receber e tratar uma mensagem de evento do sistema para uma classe que representa uma das seguintes opções:

- Representa o sistema, dispositivo ou subsistema;
- Representa um cenário de caso de uso no qual o evento do sistema ocorre.

Use a mesma classe de controle para todos os eventos do sistema nesse cenário de caso de uso.

Problema

Definir a responsabilidade por tratar uma evento do sistema.

Comentários

Um evento do sistema é um evento gerado por um ator externo. Eventos de sistema estão associados a operações do sistema – operações que o sistema executa em resposta a eventos do sistema.

Exemplo de evento do sistema: quando o caixa pressiona um botão indicando que a venda foi concluída.

Um **Controller** é uma classe (não parte da interface com o usuário) responsável por receber ou tratar um evento do sistema.

4.3.4 Patterns GoF

A seguir, são apresentados dois patterns GoF, Observer e Façade. Foram selecionados por ser relevantes a este trabalho, sendo referenciados quando da especialização do RUP para J2EE.

Observer

Embora seja um *pattern* GoF, Larman descreve Observer de forma muito clara [LARMAN 2001]:

Contexto/Problema

Diferentes tipos de objetos assinantes (*subscribers*) estão interessados em mudanças de estado ou eventos de um objeto publicador (*publisher*), e desejam reagir, cada um a seu modo, quando o publicador gera um evento. Mais ainda, o publicador deseja manter baixo acoplamento com os assinantes.

Solução

Defina uma Interface denominada Assinante (em inglês, a interface é donominada *subscriber* ou *listener*). Objetos que desempenham o papel de assinantes (ou seja, que demandam informações produzidas por outros) implementam essa Interface. Cada publicador pode registrar dinamicamente assinantes interessados em um determinado evento, e notificá-los quando o evento ocorrer.

Observação: o termo interface não deve ser confundido com interface visual. Uma Interface é um descritor de operações que não possuem implementação. É como uma classe – Java, por exemplo – composta apenas por declarações, ou assinaturas, de métodos – sem atributos e sem implementações desses métodos. Classes concretas podem realizar interfaces, o que significa que terão que implementar cada um dos métodos declarados na Interface sendo implementada.

O diagrama de classes da figura 4.4 apresenta uma visão estrutural do *pattern* Observer, com as classes que o compõem. Os objetos que representam a View fazem assinatura (*subscribe*) das informações do Model sobre as quais desejam ser informados. Quando dados do Model são alterados, os objetos da View que se

cadastraram como assinantes são informados, e podem alterar a forma como são exibidos.

Um ponto importante é que o Model não conhece diretamente nenhuma View, mas apenas a Interface Observer. Tanto View quanto Controller devem implementar os métodos definidos em Observer, e o Model vai interagir com cada View ou cada Controller, através da referência a Observer. O objetivo disso é evitar acoplamento direto entre o Model e cada View ou cada Controller. O mesmo efeito poderia ser conseguido definindo uma relação de herança entre Observer e View e Controller, embora conceitualmente a definição de Observer como Interface seja mais adequada.

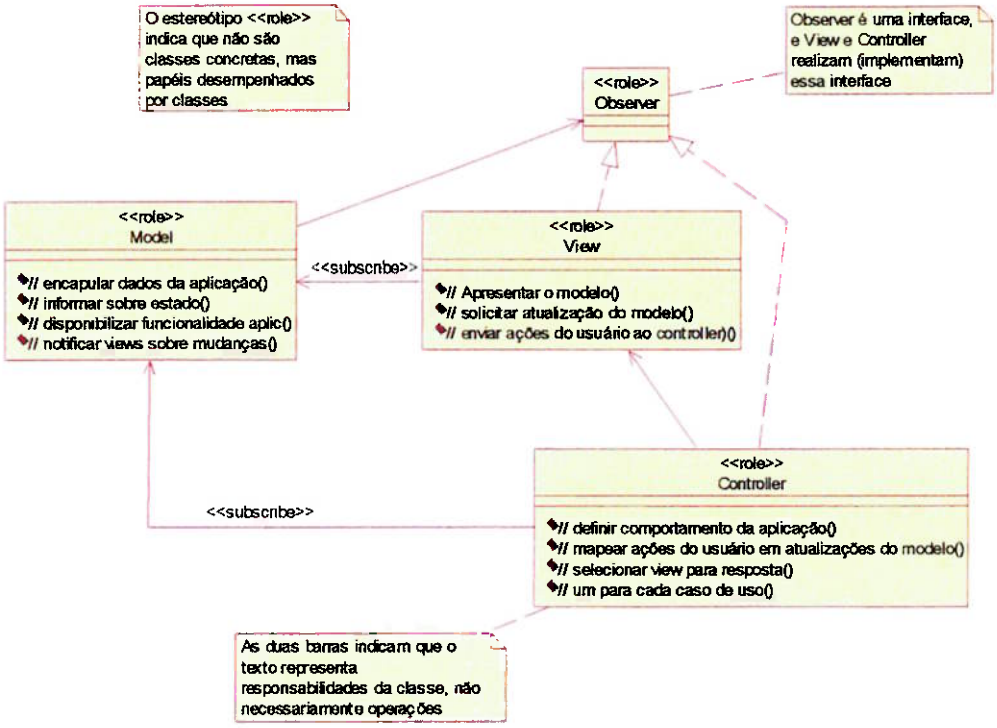


Figura 4-4 Estrutura do pattern Observer

Pattern de projeto Façade (GoF)

O objetivo desse *pattern* é definir uma interface que representa um conjunto de interfaces de um subsistema.

Motivação: Subsistemas podem prover muitas interfaces distintas para prover sua funcionalidade. A existência de muitas interfaces e operações pode confundir os projetistas/programadores que utilizam o subsistema.

Solução: oferecer uma única interface em alto nível que agrega todo o comportamento que o subsistema disponibiliza a seus clientes.

Esse *pattern* é útil para que se possa oferecer uma interface simples a um subsistema complexo, particularmente, no caso em que há muitas dependências entre os clientes e o subsistema. O Façade redefine as dependências do cliente para a interface, e permite que o subsistema seja independente e portável, ou seja, o uso de Façade reduz o acoplamento entre o cliente e o subsistema.

4.3.5 Patterns J2EE

A seguir são apresentados três dos patterns definidos pela equipe da Sun Microsystems especificamente para a plataforma J2EE. São eles: Front Controller, Session Façade e Data Access Object (DAO). Os três patterns são referenciados quando da especialização do RUP para J2EE.

Front Controller (J2EE Pattern) [DEEPAK 2001]

Contexto

O mecanismo de tratamento de requisições da camada de apresentação deve controlar e coordenar o processamento de cada usuário ao longo de diversas chamadas. Esse mecanismo de controle deve ser gerenciado de forma centralizada ou descentralizada.

Problema

O sistema requer um ponto de acesso centralizado, para tratar requisições da camada de apresentação. Esse ponto central deve prover suporte à integração de serviços do sistema, recuperação de conteúdo, gerência de *views* (*view* é a denominação de um elemento da camada de apresentação) e navegação. Se o usuário acessa a *view* diretamente, sem passar por um mecanismo centralizado, podem ocorrer dois problemas:

- Cada *view* deverá prover seus próprios serviços de sistema, resultando em duplicação de código;
- A navegação pelas *views* fica sob responsabilidade das próprias *views*, o que reduz a coesão das *views*, que deveriam tratar apenas de apresentação de conteúdo.

Forças

- Serviços de sistema comuns são acionados a cada requisição do cliente. Exemplo: serviços de segurança efetuam autenticação e autorização de usuários a cada requisição.
- Código que deveria ser definido num único ponto central fica replicado em numerosas *views*.

Solução

Utilize um controller como ponto inicial de contato para tratar requisições de clientes. O controlador gerencia o tratamento de requisições, incluindo a chamada de serviços de segurança, como autenticação e autorização, delega o processamento da lógica de negócio, gerencia a escolha da *view* apropriada para apresentar os resultados da solicitação, e gerencia a seleção das estratégias para criação de conteúdo.

Comentários

O controller provê um ponto de acesso central que controla e gerencia o tratamento de requisições web. Desta forma, os elementos que compõem as *views* são simplificados, e o objetivo de separação de áreas de interesse é atingido.

Tipicamente, o controller funciona em coordenação com um componente dispatcher (despachante). Um Dispatcher é responsável por gerência e navegação de *views*. Dispatchers podem ser encapsulados diretamente no controller ou em um componente à parte. Esta solução é preferível no caso de sistemas complexos, e a organização do sistema tende a ficar muito mais adequada: um único controller com funções muito específicas delega o tratamento de requisições de usuários a dispatchers especializados.

A visão estrutural do *pattern* Front Controller pode ser vista na figura 4.5.

Figura 4-5 Estrutura do pattern Front Controller

A visão dinâmica do *pattern* Front Controller pode ser vista na figura 4.6.

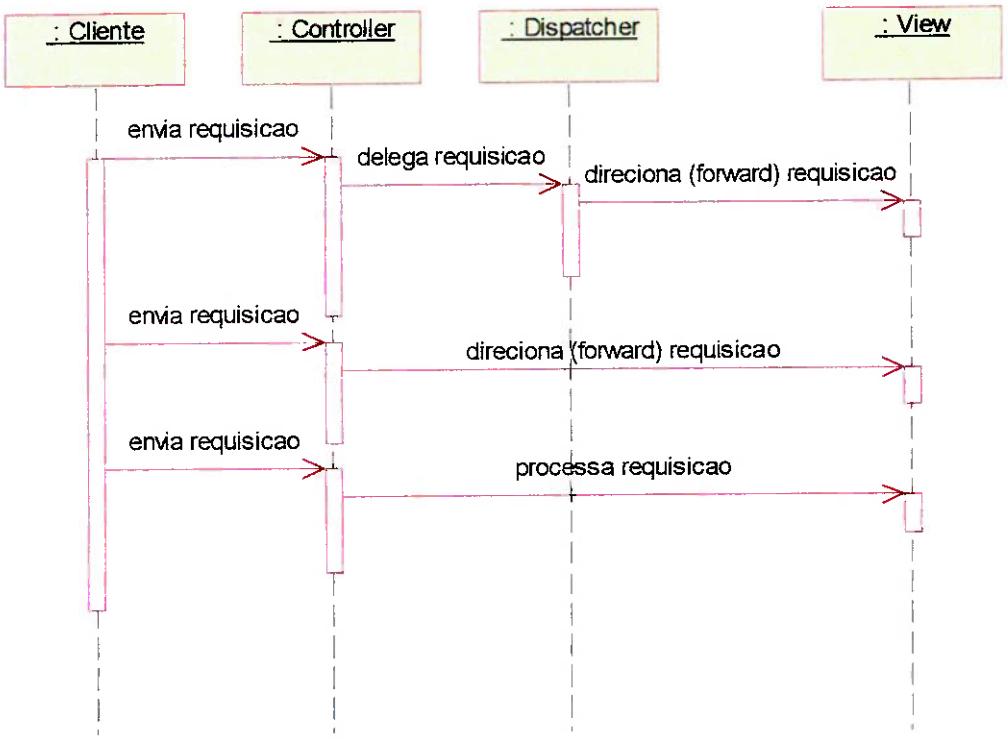


Figura 4-6 Dinâmica do pattern Front Controller

Os participantes e responsabilidades são apresentados a seguir:

Controller: é o ponto de contato inicial para tratar quaisquer requisições no sistema. O Controller pode delegar a um *Helper*, por exemplo, a autenticação e autorização de um usuário.

Dispatcher: é responsável por gerência e navegação de *view*s, gerenciando a escolha da próxima *view* a ser apresntada ao usuário.

Helper: responsável por auxiliar uma *view* ou o controller a completar sua tarefa. Têm inúmeras responsabilidades, inclusive obter dados necessários à *view* e armazenando-os temporariamente. É muito comum um tipo de *helper* denominado *value object* (objeto de valor, ou objeto de dado); armazena um conjunto de dados que uma *view* necessita para que a *view* não acesse diretamente os componentes de negócio. *Value object*s são denominados *javabeans*; um *javabean* é uma classe Java convencional que contém atributos e métodos de acesso a esses atributos.

View: é um objeto de apresentação. Obtém informações da camada de negócio (*Model*) – geralmente, através de *helper*s – e apresenta essas informações aos usuários.

Session Façade (J2EE Pattern) [DEEPAK 2001]

Contexto

Enterprise beans (EJBs) encapsulam a lógica e dados de negócio e expõem suas interfaces. Conseqüentemente, expõem a complexidade dos serviços distribuídos à camada cliente.

Problema

Numa aplicação J2EE multicamada, os seguintes problemas surgem:

- Alto acoplamento, que leva a dependência direta entre objetos clientes e de negócio;
- Excesso de chamadas de métodos de entre o cliente e o servidor, levando a problemas de desempenho de rede;
- Falta de uma estratégia uniforme para acesso pelo cliente, expondo os objetos de negócio a uso inadequado.

Uma aplicação J2EE tem inúmeros objetos denominados *enterprise beans* (EJBs), sejam *session beans* ou *entity beans*, que são referidos em conjunto como objetos de negócio, pois encapsulam a lógica e os dados de negócio.

Os clientes precisam acessar os objetos de negócio para prover suas responsabilidades e atender às requisições de usuários. Os clientes podem interagir diretamente com os objetos de negócio, pois estes expõem suas interfaces. Se isso acontecer, o cliente deve entender e tratar o fluxo de do processo de negócio, inclusive em interações complexas nas quais objetos são pesquisados, e é necessário gerenciar o relacionamento entre os objetos de negócio participantes, bem como entender as responsabilidades de demarcação transacional.

A interação direta entre o cliente e os objetos de negócio implica em alto acoplamento entre ambos, o que torna o cliente diretamente dependente da implementação dos objetos de negócio.

Há ainda os problemas potenciais resultantes do fato de que os objetos de negócio são remotos, e portanto cada requisição de serviço implica em tráfego pela rede.

Forças

- Prover uma interface mais simples aos clientes, escondendo todas as interações complexas entre objetos de negócio;
- Reduzir o número de objetos de negócio que estão expostos ao cliente;
- Esconder do cliente as interações subjacentes e interdependências entre os objetos de negócio;
- Prover uma camada de serviços uniforme, de alta granularidade, que separe a implementação dos objetos de negócio da abstração de serviços de negócio;
- Evitar expor os objetos de negócio diretamente ao cliente, para manter baixo acoplamento entre as duas camadas.

Solução

Utilizar um session bean como uma façade (fachada) para encapsular a complexidade da interação entre objetos de negócio participantes num workflow. O Session Façade

gerencia os objetos de negócio, e provê aos clientes uma camada de acesso de alta granularidade.

A figura 4.7 apresenta a estrutura do *pattern* Session Façade.

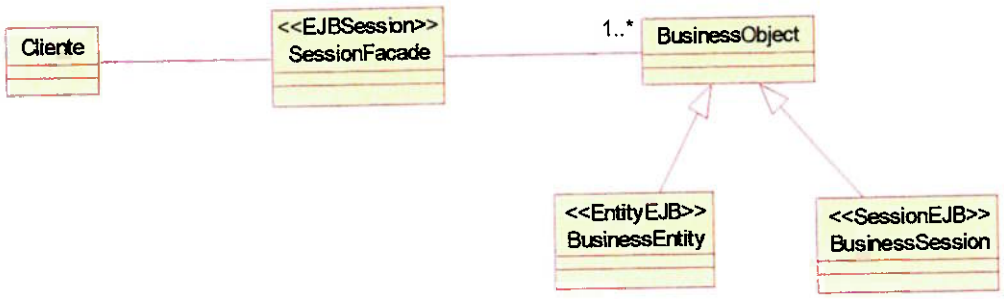


Figura 4-7 Estrutura do pattern Session Façade

A figura 4.8 apresenta a visão dinâmica de um Session Façade.

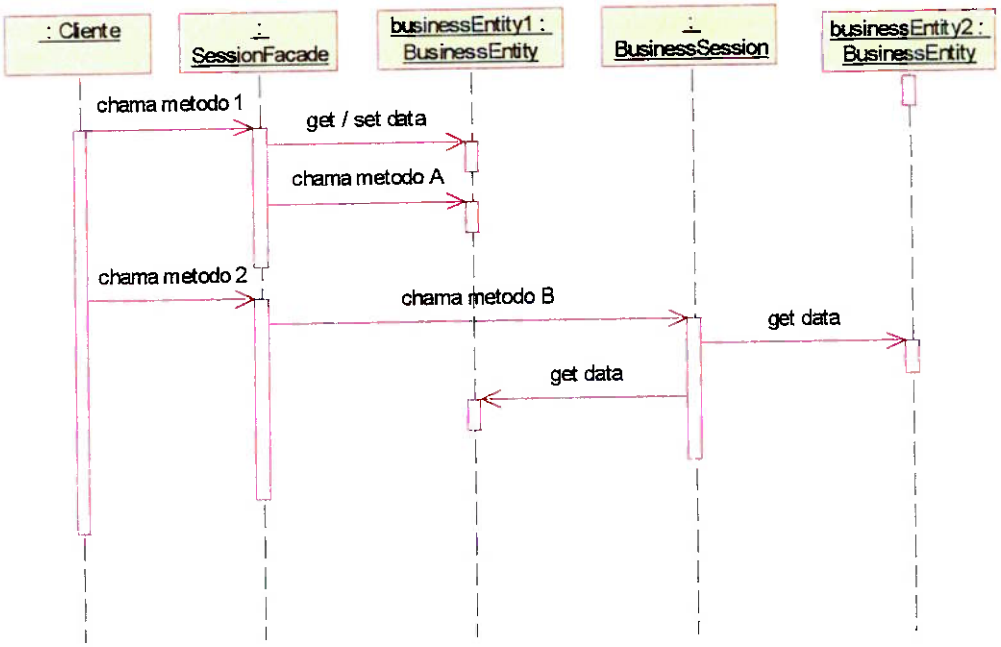


Figura 4-8 Dinâmica do pattern Session Façade

Participantes

Cliente: representa o cliente do Session Façade, que precisa acessar o serviço de negócio. Esse cliente pode ser um outro session bean na mesma camada ou um Servlet, por exemplo, do container web.

SessionFaçade: é implementado como um session bean. O Session Façade gerencia o relacionamento entre diversos objetos de negócio (business objects) e provê uma abstração de mais alto nível ao cliente.

BusinessObject: é o objeto de negócio, *session* ou *entity bean*. Provê dados ou serviços. O Session Façade interage com diversos Business Objects para prover o serviço.

Data Access Object (DAO) (J2EE Pattern) [DEEPAK 2001]

Contexto

O acesso a dados depende da fonte dos dados. O acesso a um armazenamento persistente, como um banco de dados, varia enormemente dependendo do tipo de armazenamento (banco de dados relacional, banco de dados orientado a objeto, arquivo seqüencial, e assim por diante) e da implementação do fornecedor.

Problema

Aplicações distintas utilizam forma distintas de persistir os dados, e há diferenças marcantes nas APIs utilizadas para acessar esses mecanismos de armazenamento persistente distintos.

Tipicamente, as aplicações utilizam componentes *entity beans* para representar dados persistentes. Eventualmente, *session beans* e *Servlets* podem ser utilizados para acessar o armazenamento persistente.

As aplicações geralmente utilizam a API JDBC para acessar dados residentes em um sistema gerenciador de bancos de dados relacionais (SGBDR). A forma de acesso que a API JDBC é padronizada, mas a JDBC permite que as aplicações utilizem comandos SQL para acesso ao banco de dados. Entretanto, a sintaxe de comandos SQL não é inteiramente padronizada, e pode variar entre diferentes SGBDR.

Pode haver variações ainda maiores, se se considerar que o armazenamento persistente está em um sistema legado, que deve ser acessado por uma API proprietária, ou num sistema gerenciador de banco de dados orientado a objetos.

Isso pode criar uma dependência direta entre o código da aplicação e o código do acesso a dados (acoplamento), caso o código específico para acesso a dados seja incluído nos componentes *entity beans*, *session beans* ou *Servlets*. Se a fonte de dados mudar, os componentes de negócio terão que ser alterados para acessar a nova fonte de dados.

Solução

Utilize Data Access Object (DAO) para abstrair e encapsular todos os acessos a fontes de dados. O DAO gerencia a conexão com a fonte de dados, obtém e armazena dados.

O DAO implementa o mecanismo de acesso requerido pela fonte de dados. O componente de negócio que utiliza o DAO utiliza a interface mais simples exposta pelo DAO a seus clientes.

Haverá um DAO para cada objeto de negócio. A vantagem é que o código do DAO pode ser gerado automaticamente, se for desenvolvida uma ferramenta adequada com esse objetivo (há ferramentas disponíveis comercialmente com esse objetivo).

Estrutura

A figura 4.9 apresenta o diagrama de classes representando as responsabilidades do pattern DAO.

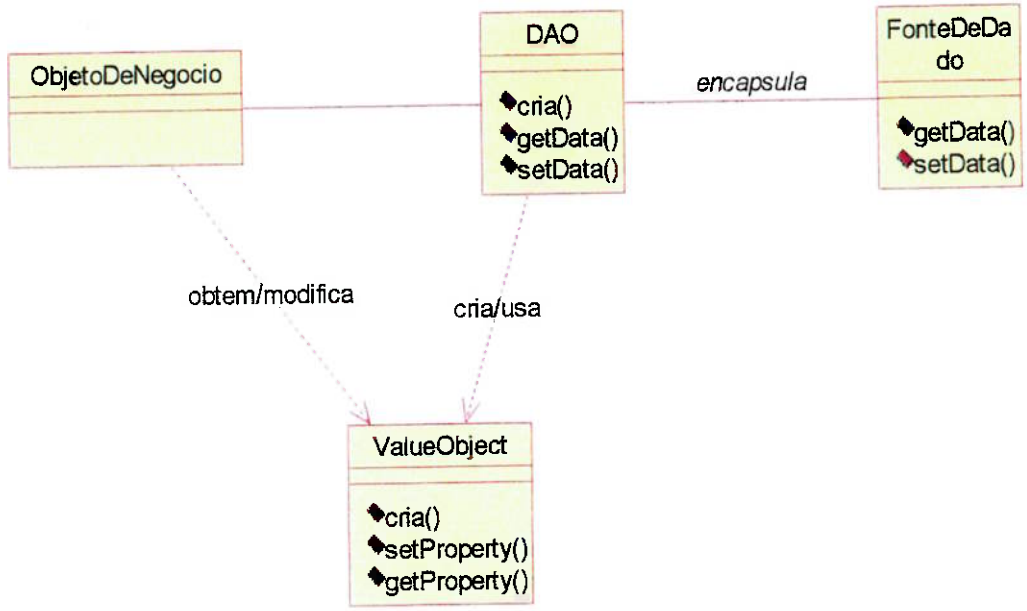


Figura 4-9 Estrutura do pattern DAO

Participantes e Responsabilidades

A figura 4.10 apresenta o diagrama de seqüência que mostra a interação entre os vários participantes do pattern.

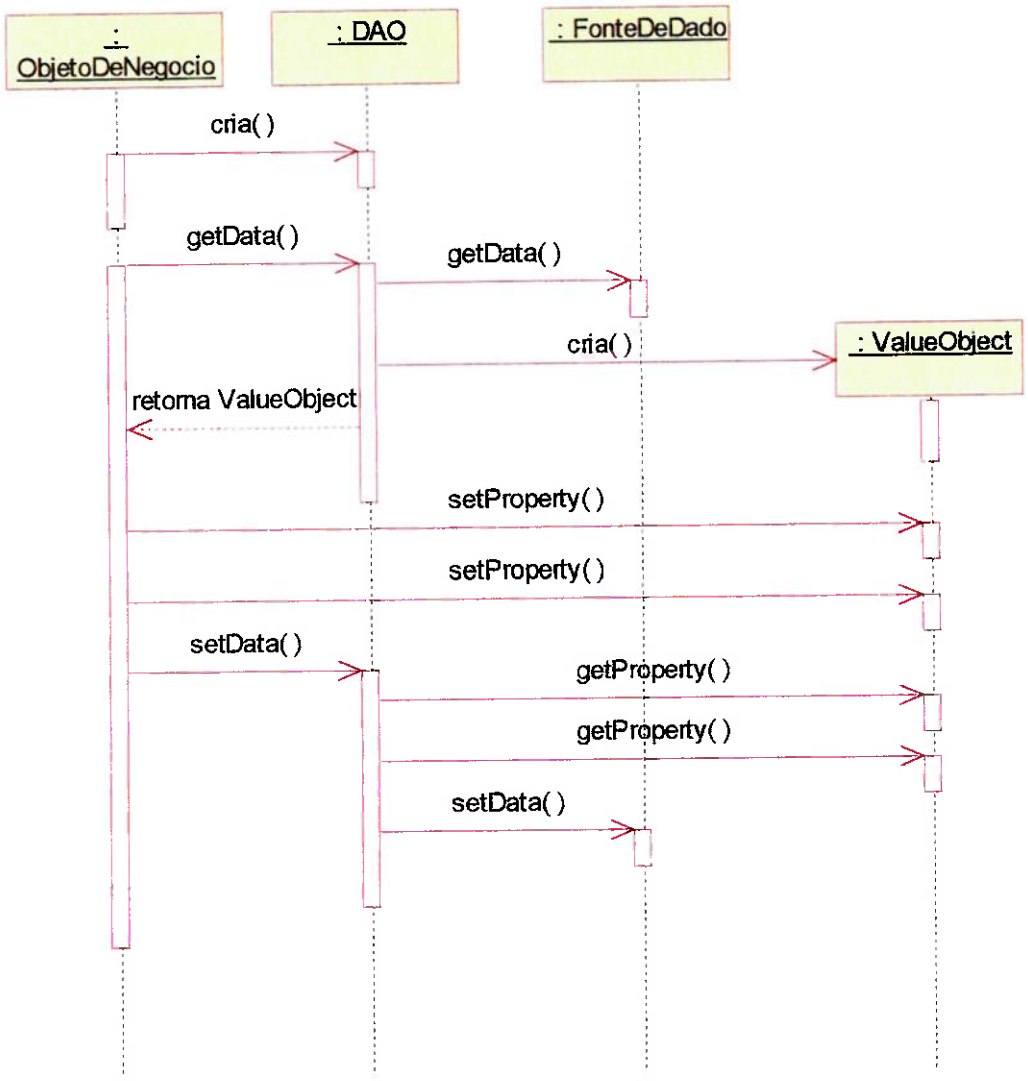


Figura 4-10 Dinâmica do pattern DAO

Objeto de Negócio

O Objeto de Negócio representa o cliente do dado. Pode ser um entity bean, um session bean, ou algum outro objeto Java.

Data Access Object

Encapsula as ações específicas para acesso à fonte de dados.

Fonte de Dados

Representa a implementação da fonte de dados. Pode ser um sistema gerenciador de banco de dados relacional, ou orientado a objetos, um repositório XML, um sistema legado.

Value object

Objeto utilizado para transportar os dados. O DAO utiliza o *value object* para retornar dado ao cliente e para receber dados e atualizar a fonte de dados.

4.4 *Especialização da Disciplina Análise e Projeto*

A organização deste tópico é definida em função da organização da disciplina Análise e Projeto, no RUP. O RUP é descrito em termos de detalhes de *workflow*, que, por sua vez, são detalhados em termos de atividades. Os detalhes de *workflow* agrupam atividades afins, mas o processo efetivamente é descrito pelas atividades e artefatos correlacionados. A figura 4.11 apresenta o *workflow* de Análise e Projeto, como um diagrama de atividades da UML, no qual cada elemento representa um detalhe de *workflow*.

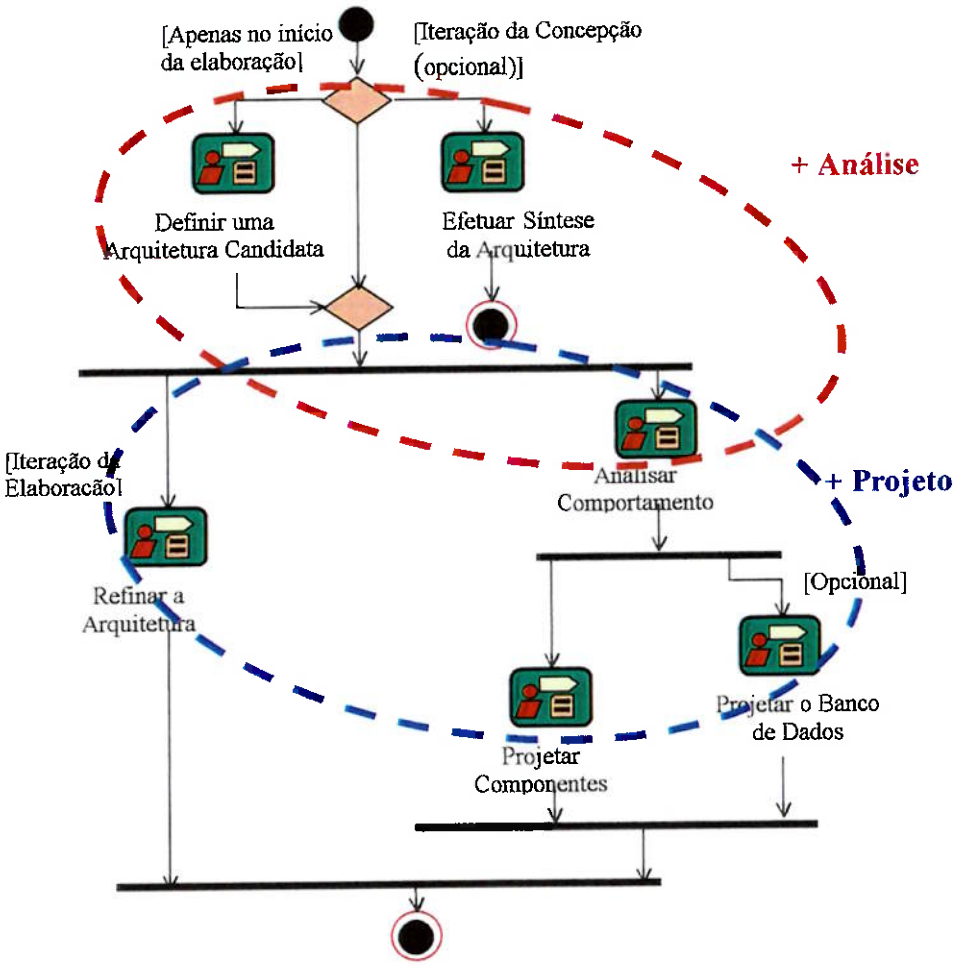


Figura 4-11 Workflow da disciplina Análise e Projeto

A especialização aborda algumas atividades da disciplina Análise e Projeto – aquelas cujo conteúdo pode ser especializado para a plataforma J2EE. Muitas atividades que são fundamentais para a execução de um projeto de software foram desconsideradas, neste contexto, por não agregarem valor aos objetivos deste texto.

O RUP não organiza a disciplina Análise e Projeto em duas partes, uma referente à análise, outra referente a projeto. No entanto, esta identificação foi feita, neste trabalho nas figuras que apresentam o *workflow* da disciplina, para, eventualmente, explicitar o foco adotado. Em princípio, poder-se-ia imaginar que atividades mais voltadas para a análise não interessariam aos objetivos deste texto, já que, se procura fazer a análise de forma independente do ambiente de implementação. Mas isso não é verdade porque a vista lógica da arquitetura de software é elaborada durante a análise e as atividades de análise definem elementos que depois serão refinados em elementos de projeto.

Cada tópico a seguir tem o nome de uma atividade da disciplina Análise e Projeto selecionada para a especialização. Cada atividade é abordada, em princípio, sob três pontos de vista:

- Será apresentada a abordagem do RUP referente ao tópico;
- Serão apresentadas considerações gerais sobre o assunto, tendo como referência a abordagem do RUP;
- Será efetuada a especialização do tópico para J2EE, quando pertinente, identificando qual a atividade ou artefato se relaciona à especialização.

Em alguns tópicos, entretanto, as considerações gerais ou a especialização para J2EE não serão abordadas, por não fazerem sentido naquela situação em particular.

4.5 *Análise da Arquitetura*

A figura 4.12 apresenta o *workflow* da disciplina Análise e Projeto, e identifica a atividade Análise da Arquitetura como parte do detalhe de *workflow* denominado Definir uma Arquitetura Candidata.

O RUP define que o objetivo desta atividade é desenvolver:

- uma arquitetura candidata (a ser a arquitetura efetiva do sistema, e que será posteriormente detalhada e validada);
- *patterns* de arquitetura, mecanismos chave e convenções de modelagem para o sistema.

As decisões tomadas pelo arquiteto de software neste ponto influenciarão toda a análise. É como se essa atividade funcionasse como uma *configuração* de uma atividade executada posteriormente, Análise dos Casos de Uso, na qual os casos de uso são realizados, em termos de objetos que interagem, para executar a funcionalidade descrita textualmente pelo caso de uso.

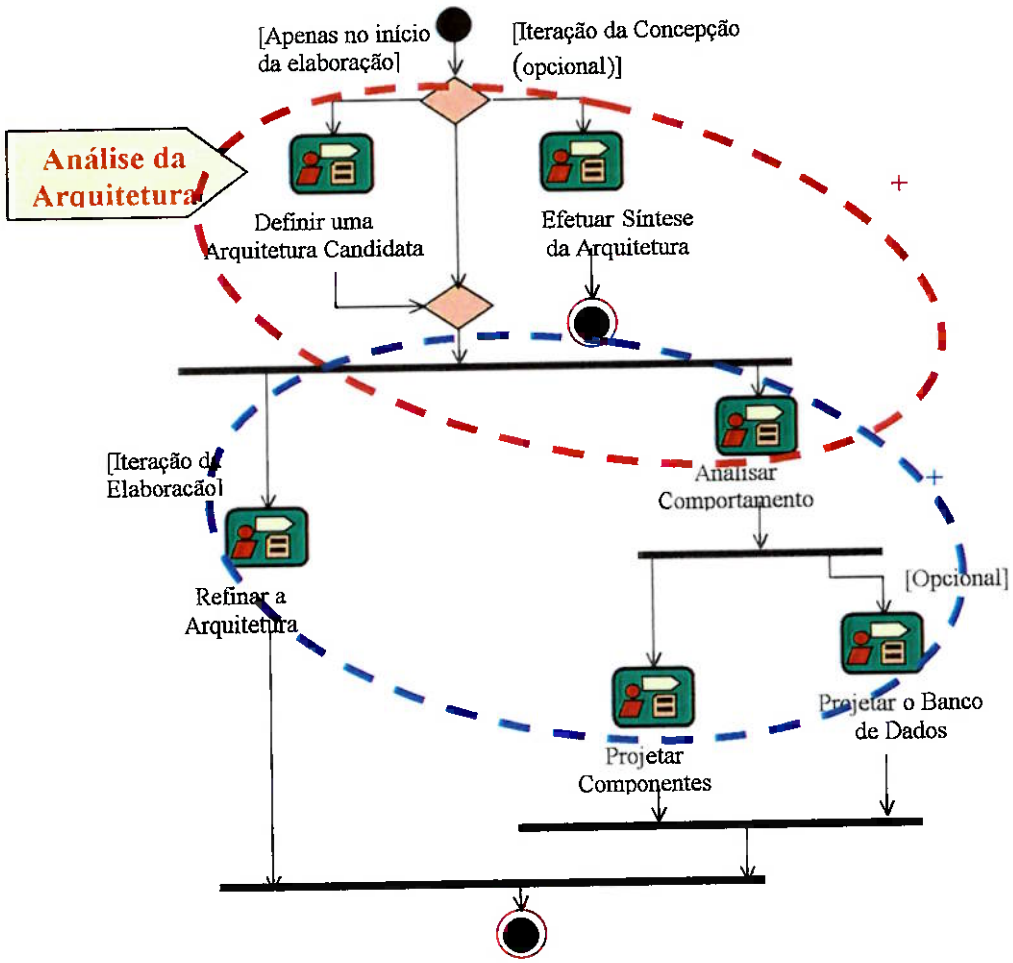


Figura 4-12 Workflow da disciplina Análise e Projeto, destacando a atividade Análise da Arquitetura

Caso de adote a abordagem de arquitetura em camadas pode-se dizer que esta atividade foca as camadas mais altas do sistema, definindo a macro organização do sistema em elementos (pacotes, subsistemas) que interagem através de interfaces definidas. Outras atividades focam as camadas mais baixas da arquitetura.

Em essência, o RUP identifica quatro passos referentes a essa atividade:

- Definir a organização dos subsistemas em alto nível (vista lógica da arquitetura de software);
- Identificar mecanismos de análise;
- Identificar abstrações chave;
- Criar as realizações de casos de uso.

Dos passos citados acima, os dois primeiros abordam temas de interesse à especialização. Por isso, cada passo será abordado a seguir segundo a estrutura definida para a especialização:

- A abordagem do RUP;
- Considerações gerais, tendo como referência a abordagem do RUP;
- Especialização para J2EE.

4.5.1 Abordagem RUP para a Definição da Organização dos Subsistemas em Alto Nível

Para direcionar o esforço de definir a organização em alto nível da aplicação em termos de subsistemas, o RUP apresenta uma diretriz denominada Layering (definição em camadas) e, no item Conceito: Arquitetura de Software, há um subitem denominado Patterns de Arquitetura. A diretriz para organizar o sistema em camadas (Layering) trata de como utilizar, de forma efetiva, um *pattern* de arquitetura (Layers) apresentado brevemente neste texto, logo a seguir.

Os conceitos de *Patterns* de Arquitetura do RUP são inteiramente extraídos de [BUSCHMANN 1996], que é a principal referência bibliográfica sobre o tema. O RUP conceitua é um *pattern* de arquitetura como "formas prontas para resolver

problemas de arquitetura recorrentes", e apresenta uma tabela 4.1, que cita os *patterns* de Buschmann, seguindo a classificação proposta pelos autores.

Categoria	Pattern
Estrutura	Layers
	Pipes and Filters
	Blackboard
Sistemas Distribuídos	Broker
Sistemas Interativos	Model-*View*-Controller
	Presentation-Abstraction-Control
Sistemas Adaptáveis	Reflection
	Microkernel

Tabela 4.1 Classificação dos patterns de arquitetura de Buschmann

Dois desses *patterns* são sucintamente descritos no RUP (e foram descritos também neste capítulo): o Layers e o Blackboard.

4.5.2 Considerações gerais sobre a organização de sistemas em alto nível

Larman aborda também alguns *patterns* de arquitetura, no contexto do Unified Process (UP, a versão simplificada e não comercial do RUP) [LARMAN 2001]. O *pattern* Layers, citado pelo RUP, também é contemplado, e conceitos fundamentais de arquitetura de software (separação de áreas de interesse, baixo acoplamento, alta coesão) contemplados pelo *pattern* são enfatizados. Larman apresenta um exemplo de organização de um sistema em camadas um pouco mais elaborado que o do RUP, e que pode ser visto na figura 4.13.

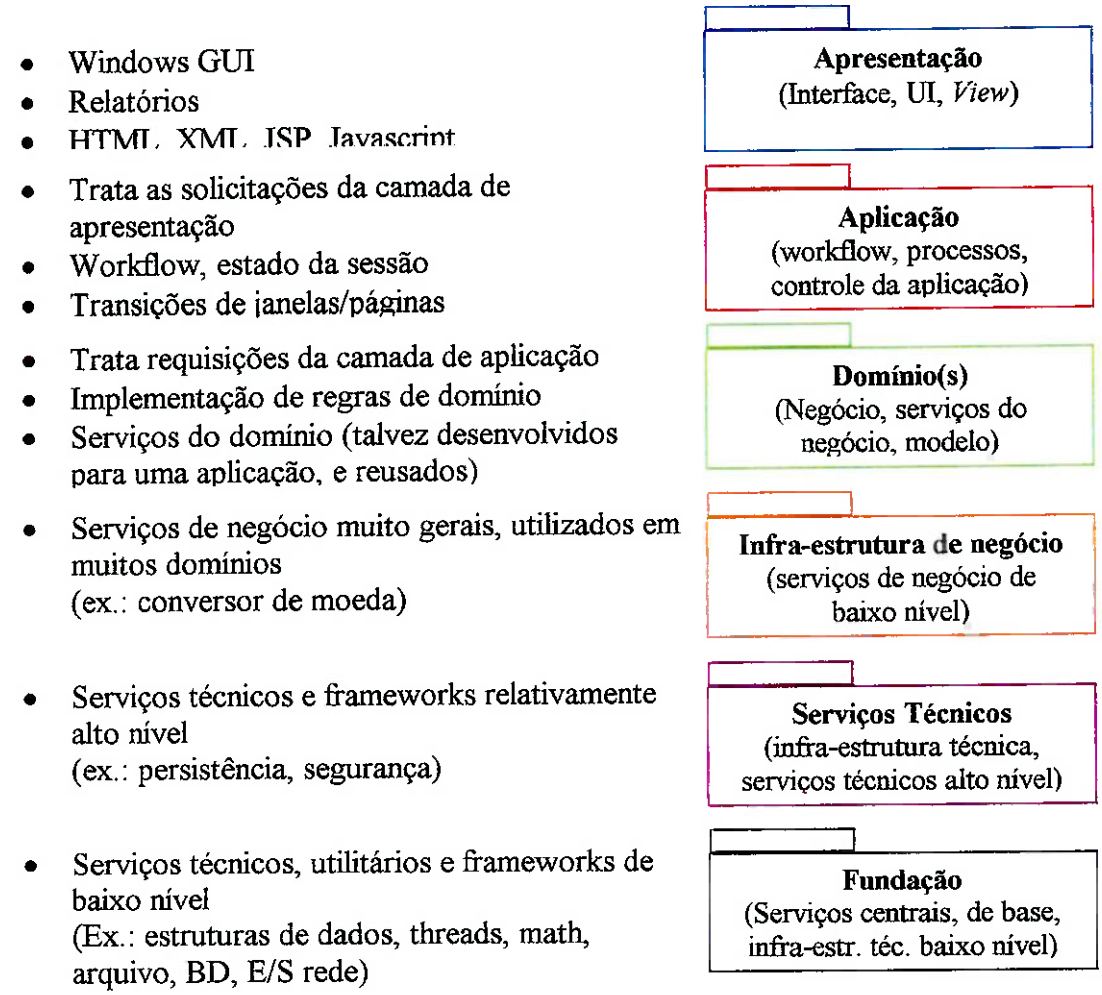


Figura 4-13 Exemplo de arquitetura em camadas apresentado por Larman

Cada uma das camadas é representada por um pacote, onde um pacote é um elemento estrutural da UML, que serve para conter outros elementos de modelagem. Como se trata de uma vista lógica da arquitetura, não se entra no mérito de componentes (elementos de implementação, não de projeto); por isso escolheu-se representar uma camada como sendo um pacote. Segundo a UML, um pacote pode conter outros pacotes e também outros elementos de modelagem (como classes). Pode-se entender, portanto, que o pacote que representa cada camada normalmente vai conter outros pacotes, que representam os elementos estruturais de maior granularidade contidos em cada camada (subsistemas, por exemplo). Estes, por sua vez, conterão classes

4.5.3 Especialização para J2EE da Organização de Sistemas em Alto Nível

Qualquer dos *patterns* de arquitetura definidos por Buschman, em princípio, pode ser utilizado em aplicações J2EE, já que não há restrições sobre qual o tipo de aplicação será desenvolvido nessa plataforma (exceto, talvez, sistemas de tempo real, incompatíveis com invocação de objetos distribuídos em uma rede pública).

Outro ponto importante é que pode ser desejável utilizar mais que um *pattern* de arquitetura no projeto de uma aplicação. Um deles será o *pattern* mais estrutural e, sobre essa estrutura base, outros *patterns* poderão ser aplicados. Esta é a questão fundamental: deseja-se identificar os *patterns* úteis para a estruturação em alto nível de aplicações empresariais web.

Isso é efetuado na tabela 4.2, a mesma que já foi apresentada anteriormente neste tópico, contemplando os *patterns* de arquitetura definidos em [BUSCHMANN 96], mas com uma terceira coluna, na qual é comentada a aplicabilidade de cada *pattern* a aplicações empresariais web.

Categoria	Pattern	Aplicabilidade a sistemas empresariais web
Estrutura	Layers	Sim. A organização da plataforma J2EE foi concebida para facilitar o desenvolvimento de aplicações em camadas. Esse é o *pattern* de arquitetura mais importante para a plataforma J2EE, e é comentado em mais detalhes neste tópico.
	Pipes and Filters	Específico para sistemas cujo foco é a manipulação de *streams* de dados, o que não é o caso da maioria dos sistemas empresariais web.
	Blackboard	Específico para sistemas não algorítmicos. Apenas em casos muito especiais seria utilizado.
Sistemas Distribuídos	Broker	Aplicações web empresariais freqüentemente são projetadas para um ambiente distribuído, no qual é necessária a atuação de um broker (intermediário) para viabilizar a comunicação com objetos remotos. Mas essa é uma questão de infra-estrutura, tornada transparente quando uma plataforma de desenvolvimento como a J2EE é utilizada. No nível lógico, do qual estamos tratando, esse *pattern* não é relevante.

Sistemas Interativos	Model-*View*-Controller	Sim. Diversos princípios importantes no desenvolvimento de aplicações empresariais web são resolvidos pelo uso desse *pattern*.
	Presentation-Abstraction-Control	Sim. Este *pattern* possui diversas semelhanças com o *pattern* MVC, e endereça questões da mesma natureza que o MVC.
Sistemas Adaptáveis	Reflection	*Pattern* para uma classe específica de sistemas. Não relevante num contexto mais genérico.
	Microkernel	*Pattern* para uma classe específica de sistemas. Não relevante num contexto mais genérico.

Tabela 4.2 aplicabilidade de *patterns* de arquitetura a aplicações empresariais web

A figura 4.14 apresenta uma visão simplificada da arquitetura da plataforma J2EE, já apresentada no capítulo 2 deste texto em maiores detalhes. O elemento central da arquitetura é um servidor de aplicações, composto por um container de componentes web e o container de componentes EJB. O container web executa JSPs e Servlets. O container EJB, como o nome indica, executa *enterprise Javabeans* (EJBs). Servlets, JSPs e EJBs são os blocos de construção de aplicações J2EE. A partir deles, acessa-se aos serviços de infra-estrutura providos pela plataforma, como acesso a bancos de dados utilizando JDBC, serviço de nomes e diretórios utilizando JNDI, acesso a objetos remotos utilizando RMI, etc.

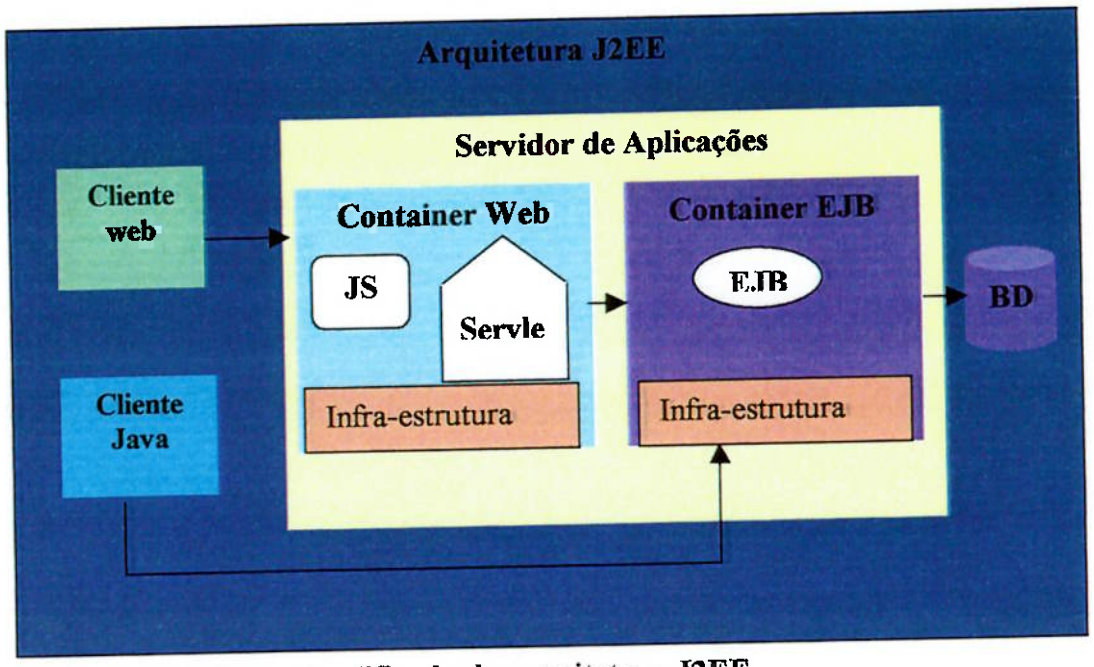


Figura 4-14 Visão simplificada da arquitetura J2EE

É possível identificar o que se o que se poderia chamar de vocação, para cada Servlets, JSPs e EJBs.

Servlets e JSPs são, em princípio, intercambiáveis (um JSP, quando compilado, é convertido num Servlet). Apesar disso, na prática, JSPs são escritos como páginas HTML na qual se embute fragmento de código Java, enquanto Servlets são classes Java nas quais é possível incluir código para a geração de páginas HTML. Ambos, Servlets e JSPs estão aptos a lidar com o mecanismo request/response, típico do protocolo http, usado na web. A vocação de um JSP é gerar páginas HTML, enquanto que a vocação de um Servlet identificar qual lógica de negócio deve ser acionada e quais páginas devem ser exibidas.

Implementar a lógica de negócio e representar as abstrações do domínio, por sua vez, é a vocação respectivamente de EJBs do tipo session bean e de EJBs do tipo entity beans.

A plataforma J2EE não obriga a que as aplicações sejam desenvolvidas com base nessas vocações, mas foi concebida para que as vocações sejam consideradas e ocorra uma especialização no uso desses elementos. Uma aplicação poderia, em princípio, ser construída apenas com JSPs, que poderiam desde gerar a interface

visual até acessar diretamente o banco de dados, a preço de não poder contar com toda a infra-estrutura de serviços prestados pela plataforma, já boa parte dos quais é prestada pelo container EJB.

Além de definir componentes especializados, a arquitetura da plataforma é também especializada: há um cliente (web ou Java) que se comunica com um servidor web, que se comunica com um servidor de componentes de negócio, que se comunica com um servidor de banco de dados ou com outros sistemas.

Esse é o cenário para a definição dos *patterns* que serão destacados para nortear a estruturação em alto nível de aplicações empresariais web desenvolvidas na plataforma J2EE. Considerando esse cenário, fica claro que plataforma J2EE foi concebida para facilitar a estruturação dos aplicativos em camadas, conforme o *pattern* Layers prevê.

O uso do *pattern* Layers é muito positivo, pois diversas boas práticas de engenharia de software são enfatizadas pela organização de um sistema em camadas, como a separação de áreas de interesse distintas (*separation of concerns*), modularização, e distinção entre interface e implementação.

Quantas e quais são as camadas nas quais o aplicativo será estruturado é uma questão para a qual não há uma resposta pré-definida e vai depender da complexidade e porte do aplicativo.

Um princípio importante ,a ser seguido em aplicações web, é o da separação entre a aplicação propriamente dita e a interface visual da aplicação. As empresas estão constantemente mudando a interface visual de suas aplicações, especialmente no ambiente web, e há grandes chances de que novos canais, com interface visual específica (celulares, PDAs) sejam incorporados às aplicações. Isso deve poder ser feito com relativa simplicidade e, principalmente, com mínimo impacto nos demais elementos da aplicação. Um dos objetivos principais do *pattern* MVC é justamente resolver essa questão: a separação entre o Modelo e a Vista do modelo.

Por conta disso, é importante citar a relação definida em [LARMAN 2001] entre o *pattern* Layers e o *pattern* **Model-*View*-Controller (MVC)**: O Modelo (model) é a camada de Domínio, a Vista (*view*) é a camada de apresentação, e o Controlador (controller) são os objetos de *workflow* na camada de apresentação. Desse ponto de

vista, Layers é um refinamento de MVC e MVC é um Layers com apenas três camadas.

A proposta é, portanto, utilizar o *pattern* Layers como referência para a organização de sistemas J2EE, definindo camadas que respeitem a separação de áreas de interesse proposta pelo *pattern* MVC. Este, em particular, poderia ser utilizado em estruturações mais simples. Sistemas mais complexos podem demandar uma estruturação mais elaborada, com mais camadas, mas sempre respeitando o princípio da separação entre modelo-vista-controle.

Para caracterizar isso de forma mais detalhada, utiliza-se o exemplo de organização de sistema em camadas elaborado por [LARMAN 2001] e adequa-se especificamente à organização da plataforma J2EE, conforme pode ser visto na figura 4.15.

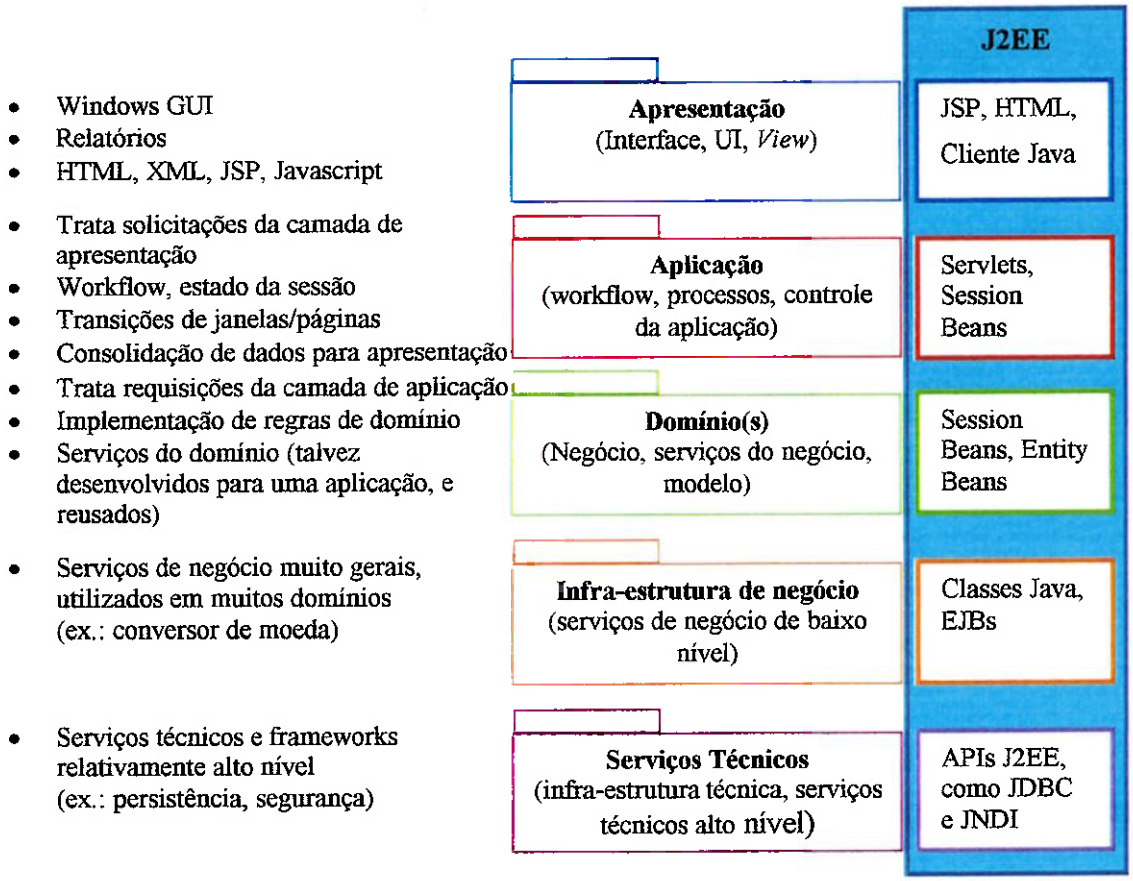


Figura 4-15 Arquitetura em camadas referenciando elementos da plataforma J2EE

A figura 4.15 acrescenta uma nova coluna à figura 4.13. Nessa coluna, os elementos da plataforma J2EE que implementam a arquitetura lógica proposta são apresentados. A camada inferior na figura original – Fundação – foi excluída, pois são serviços providos pelo ambiente de forma transparente para a equipe de desenvolvimento, e portanto não precisam ser representados no nível lógico. A organização em camadas de aplicação, domínio e infra-estrutura de negócio, embora em princípio sejam uma excelente idéia (pois contribuem para aumentar o grau de reuso de componentes) pode ser simplificada. Uma sugestão é manter essa estruturação como diretriz para uma Organização, mas o desenvolvimento de cada aplicativo específico deveria definir sua própria Estruturação, tendo como referência a Estruturação diretriz.

Além do respeito ao *pattern* de arquitetura MVC, a outra definição importante do modelo de arquitetura proposto é a distinção entre uma camada de aplicação, uma camada de domínio e uma ou mais camadas de infra-estrutura de negócio. Essa abordagem facilita a definição de um projeto no qual haja uma caracterização clara de componentes mais específicos de uma aplicação e componentes mais genéricos (de um certo domínio ou do negócio como um todo). Esses componentes das camadas mais baixas terão maior chance de reuso, e a arquitetura encoraja sua definição como abstrações mais genéricas, potencialmente reusáveis.

Esta forma não é a única de organizar uma aplicação J2EE. É uma forma básica, a partir da qual formas mais sofisticadas ou efetivamente distintas podem ser elaboradas. Exemplo: um sistema muito grande e complexo poderia ser organizado em termos de grandes componentes de negócio, cada um deles por sua vez composto por três camadas: apresentação, controle e integração.

A camada mais baixa da arquitetura proposta é a de Serviços Técnicos. Foi incluída porque, desde que se utilize os componentes EJB, alguns desses serviços serão necessariamente demandados (no mínimo, os serviços de nomes e de acesso a objetos remotos), ainda que a aplicação não envolva persistência. Essa camada contemplaria elementos construídos para acessar, em mais alto nível, os serviços já demandados pela plataforma. Portanto, esta camada não contempla os componentes das APIs Java, mas os componentes construídos pela equipe de desenvolvimento para encapsular, em um nível mais alto de abstração, as chamadas a essas APIs Java.

Observa-se ainda que o *pattern* de projeto Observer, apresentado como uma forma de concretizar a aplicação do *pattern* de arquitetura MVC – e embora útil para ilustrar na prática como utilizar o *pattern* de arquitetura MVC – teria que ser abordado de forma um tanto diferente, para se adequar à para a plataforma J2EE. Esse *pattern* foi apresentado considerando um mecanismo de atualização da *view* denominado *push-from-below* (empurrar de baixo), pelo qual, sempre que houver atualização do modelo, será disparado um mecanismo de propagação de mudanças para informar os objetos da *view* sobre a alteração, para que estes, se necessário, se atualizem. Isso não é viável no caso da plataforma J2EE, pois componentes de negócio, que representam o *model*, são passivos. Teria que ser adotado um modelo *pull-from-above* (puxar de cima), também viável, no qual a própria view consulta o modelo quando necessário, para identificar se houve alterações nas informações demandadas.

4.5.4 Abordagem RUP para Identificar Mecanismos de Análise

Este passo da Atividade Análise da Arquitetura tem como objetivo identificar e caracterizar os mecanismos de análise que devem ser considerados no projeto. Mecanismos de análise são um tipo de mecanismos de arquitetura, descritos no capítulo 3. Um ponto forte da abordagem do RUP para Análise e Projeto é a ênfase que o processo dedica à definição desses mecanismos que, essencialmente, simplificam e potencialmente agregam mais qualidade ao projeto.

O RUP prevê a seguinte abordagem referente a mecanismos de análise:

- Identificar os mecanismos de análise que são relevantes para o sistema em questão;
- Identificar as características relevantes para o mecanismo em questão (exemplo: para o mecanismo de persistência, as características poderiam ser granularidade, volume, duração, freqüência de acesso, confiabilidade);
- Identificar os mecanismos que afetam cada classe de análise;
- Modelar os mecanismos utilizando colaboração.

Essa seqüência de passos não necessariamente terá que ser seguida do início ao fim. Normalmente, apenas se identificam os mecanismos de análise e se identifica a quais

classes cada mecanismo será aplicado. A modelagem do mecanismo será efetuada no nível de projeto, não de análise.

4.5.5 Especialização do Mecanismos de Análise para J2EE

Não há propriamente como especializar mecanismos de análise para a plataforma J2EE, pois o modelo está sendo elaborado no plano mais conceitual, sem focar o ambiente de implementação. Essa especialização será efetuada mais adiante, quando abordarmos a atividade Identificar Mecanismos de Projeto.

É possível, entretanto, identificar os mecanismos de análise comuns a aplicações empresariais web em ambiente distribuído, diversos dos quais são serviços prestados pela infra-estrutura J2EE. São eles:

- Persistência;
- Segurança;
- Distribuição.

Certamente outros aspectos poderiam ser tratados utilizando a abordagem de mecanismos. Esse tema não será esgotado aqui, e é um forte candidato a ser objeto específico de futuros trabalhos de pesquisa, assim como a definição de novos *patterns* de projeto J2EE.

4.6 *Análise de Casos de Uso*

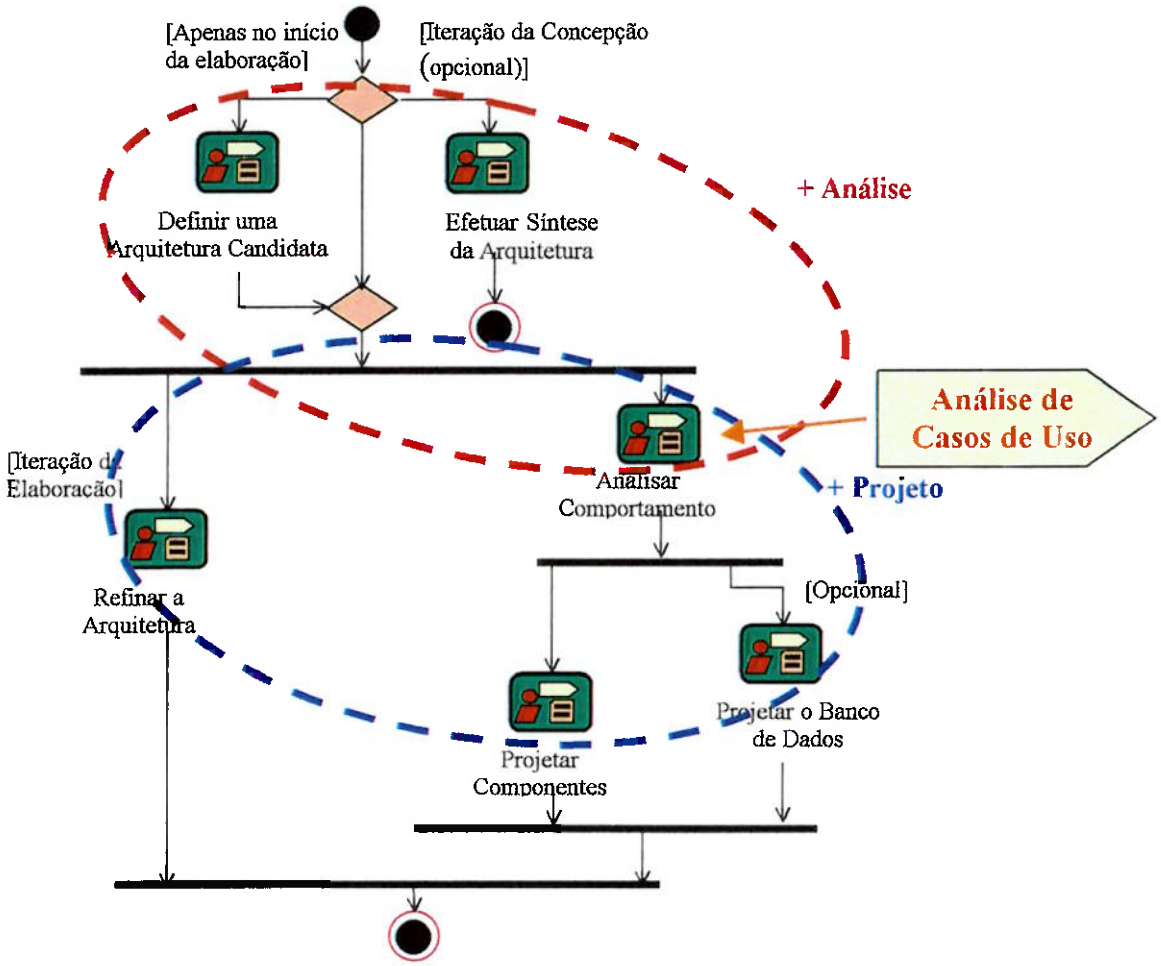


Figura 4-16 Workflow da disciplina Análise e Projeto, destacando a atividade Análise de Casos de Uso

4.6.1 Descrição da atividade Análise de Casos de Uso

Esta é, provavelmente, a principal atividade da disciplina de análise. O objetivo é gerar, para cada caso de uso, o modelo do sistema que o **realiza** em termos de diagramas de classe e diagramas de interação. Uma vez que esse modelo seja refinado pelas atividades de projeto, pode-se obter um projeto do sistema suficientemente detalhado para ser implementado. Os responsáveis pela implementação (os mesmo que o projetaram, ou outras pessoas) se valerão das

realizações de casos de uso para implementar o sistema. Essa atividade se vale das definições de arquitetura e mecanismos de análise e, evidentemente, das descrições detalhadas dos casos de uso. A atividade é repetida para cada caso de uso do sistema.

Os passos para a execução dessa atividade não serão descritos neste trabalho, já que o objetivo não é descrever o RUP em detalhes. Resumidamente, pode-se dizer que a atividade consiste em identificar as classes que realizam um caso de uso e distribuir o comportamento descrito no caso de uso entre essas classes.

4.6.2 Classes de interface, entidade e controle

Do ponto de vista dos objetivos deste texto, o aspecto mais importante a ser destacado na atividade Análise de Casos de uso é a proposição do RUP de que as **classes de análise** identificadas sejam representadas utilizando três estereótipos de classe definidos pela UML: boundary (fronteira, anteriormente denominada interface), interface e control (controle).

A definição desses estereótipos talvez pareça pouco importante, à primeira vista, mas sua utilização traz uma forte contribuição para que, ainda durante a análise, o modelo produzido já seja organizado de forma que os conceitos fundamentais de orientação a objetos (encapsulamento, modularização, baixa coesão, alto acoplamento, abstração tão fiel quanto possível ao mundo real) sejam contemplados de forma adequada, e para que a posterior elaboração do modelo de projeto tenha como base um modelo de análise bem constituído, a partir do qual a derivação em projeto seja razoavelmente natural.

Métodos como o OMT (Object Modeling Technique – Técnica de Modelagem de Objetos) [RUMBAUGH 1991], desenvolvido por James Rumbaugh, foram muito populares no início dos anos 90, e ainda hoje são uma referência forte. O OMT, como a maioria dos métodos, não procura distinguir tipos diferentes de classes. O modelo de classes gerado nesse tipo de situação tende a distribuir todas as responsabilidades para execução das funcionalidades em classes derivadas do domínio do problema. Um modelo desse tipo aparentemente pode parecer mais adequado ao padrão orientado a objetos do que o modelo gerado segundo as definições do RUP; mas na verdade tende ser menos fiel aos preceitos de orientação a objetos, pois a atribuição de responsabilidades a classes é definida de forma

potencialmente inadequada, sobrecarregando as classes de domínio – e é mais difícil derivar desse modelo um modelo de projeto da aplicação.

Uma **classe de fronteira**, segundo o RUP, é um tipo de classe utilizado para modelar a interação entre o sistema e o ambiente externo (representado sob a forma de atores – papéis desempenhados por pessoas ou outros sistemas). São responsáveis por:

- Coordenar o comportamento do ator com os elementos internos do sistema;
- Receber estímulos do ator para o sistema;
- Prover informações ao ator a partir do sistema.

Classes de interface comuns são janelas, páginas web, telas, sensores (a representação de um sensor como software).

Em UML, uma classe de fronteira é representada pelo estereótipo <<boundary>>. As três formas de representação de classes de fronteira previstas na UML podem ser vistas na figura 4.17,

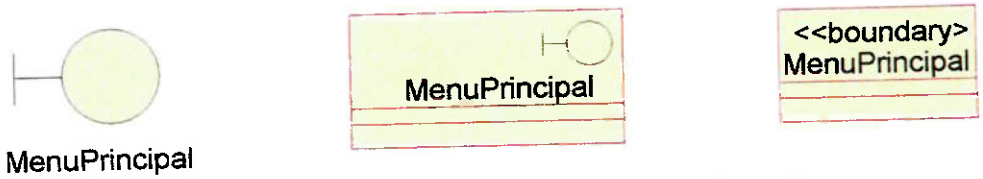


Figura 4-17 Representações UML para classes de fronteira

Uma **classe de controle** modela o comportamento de controle específico a um ou alguns casos de uso. Os objetos de controle (instâncias de classe) normalmente controlam outros objetos, ou seja, atuam como coordenadores. Essas classes são dependentes do caso de uso a partir do qual foram projetadas: se o caso de uso mudar, a classe tem que mudar.

Um objeto de controle tem um papel gerencial: ela não é responsável por executar a lógica de negócio, mas por coordenar sua execução, e por acionar os objetos que efetivamente têm a responsabilidade de executar a lógica de negócio. O princípio geral é que haverá pelo menos uma classe de controle por caso de uso, mas um caso

de uso complexo pode necessitar de um controle principal, que acionará controles específicos.

Um objetivo importante da classe de controle é desacoplar classes de fronteira de classes entidade, de modo que o sistema seja mais flexível, no sentido de que mudanças na interface do sistema não impactem a lógica interna de funcionamento.

Em UML, uma classe de controle é representada pelo estereótipo <<control>>. As três formas de representação de classes de controle previstas na UML podem ser vistas na figura 4.18.

Figura 4-18 Representações UML para classes de controle

Uma **classe entidade** é definida pelo RUP como aquela utilizada para modelar informação e comportamento associado que devem ser armazenados. Os objetos entidade (instâncias de classe entidade) são utilizados para armazenar e atualizar informação sobre fenômenos, tais como um evento, uma pessoa, ou um objeto da vida real. Geralmente são persistentese normalmente representam conceitos chave do sistema em desenvolvimento. Exemplos são ContaCorrente e Cliente, ou Nó e Link.

Classes entidade normalmente não estão associadas a casos de uso específicos. Muitas vezes inclusive não são restritas nem mesmo a apenas um sistema. Podem ter comportamento complexo, mas esse comportamento será fortemente relacionado ao fenômeno que representam.

Em UML, uma classe entidade é representada pelo estereótipo <<entity>>. As três formas de representação de classes entidade previstas na UML podem ser vistas na figura 4.19.

Figura 4-19 Representações UML para classes entidade

4.7 *Considerações Gerais*

4.7.1 Classes de Análise e o pattern MVC (model-view-controller)

Dois aspectos referentes à definição de classes de análise segundo a proposta do RUP podem ser associados ao *pattern* de arquitetura MVC.

A organização das classes de análise segundo os estereótipos propostos pelo RUP (boundary – control – entity) reforçam a organização do sistema segundo o *pattern* MVC, embora o RUP não afirme isso explicitamente. Basta verificar como as definições associadas abaixo são efetivamente muito similares :

- Entity ⇔ Model
- Boundary ⇔ *View*
- Control ⇔ Controller

A conseqüência prática a ser ressaltada é que a atribuição dos estereótipos às classes de análise auxilia o projetista a seguir o *pattern* de arquitetura MVC, ainda durante a análise.

Não se destaca neste contexto que, durante a atividade Análise de Caso de Uso, devem ser definidas associações entre as classes de análise. Esse é um procedimento básico quando da modelagem orientada a objetos, e fora do escopo deste texto. Entretanto, um tipo específico de associação deve ser comentado, pois é outro aspecto da definição de classes de análise que reforça o pattern MVC.

A UML define um estereótipo para associações entre classes denominado <<subscribe>>. Uma associação <<subscribe>> é ilustrada na figura 4.20.

Figura 4-20 Associação subscribe

Na figura citada, uma classe A tem uma associação <<subscribe>> com uma classe B, o que significa que A deve ser informada quando determinados eventos ocorrerem em B (normalmente, quando ocorrerem certas mudanças de estado em B). Pode-se definir uma condição para a associação, que identifica o evento em B cuja ocorrência deve ser notificada à classe A.

O *pattern* Observer, já citado, e que é uma forma de projeto associada ao *pattern* de arquitetura MVC, contempla associações <<subscribe>> dos elementos da *View* ou do Controller para elementos Model.

4.8 *Especialização para J2EE*

A atividade Identificar Elementos de Projeto, descrita a seguir, é o ponto adequado para abordar a derivação dos tipos de classes de análise propostos pelo RUP em classes de projeto, no contexto da plataforma J2EE.

4.9 *Identificar Elementos de Projeto*

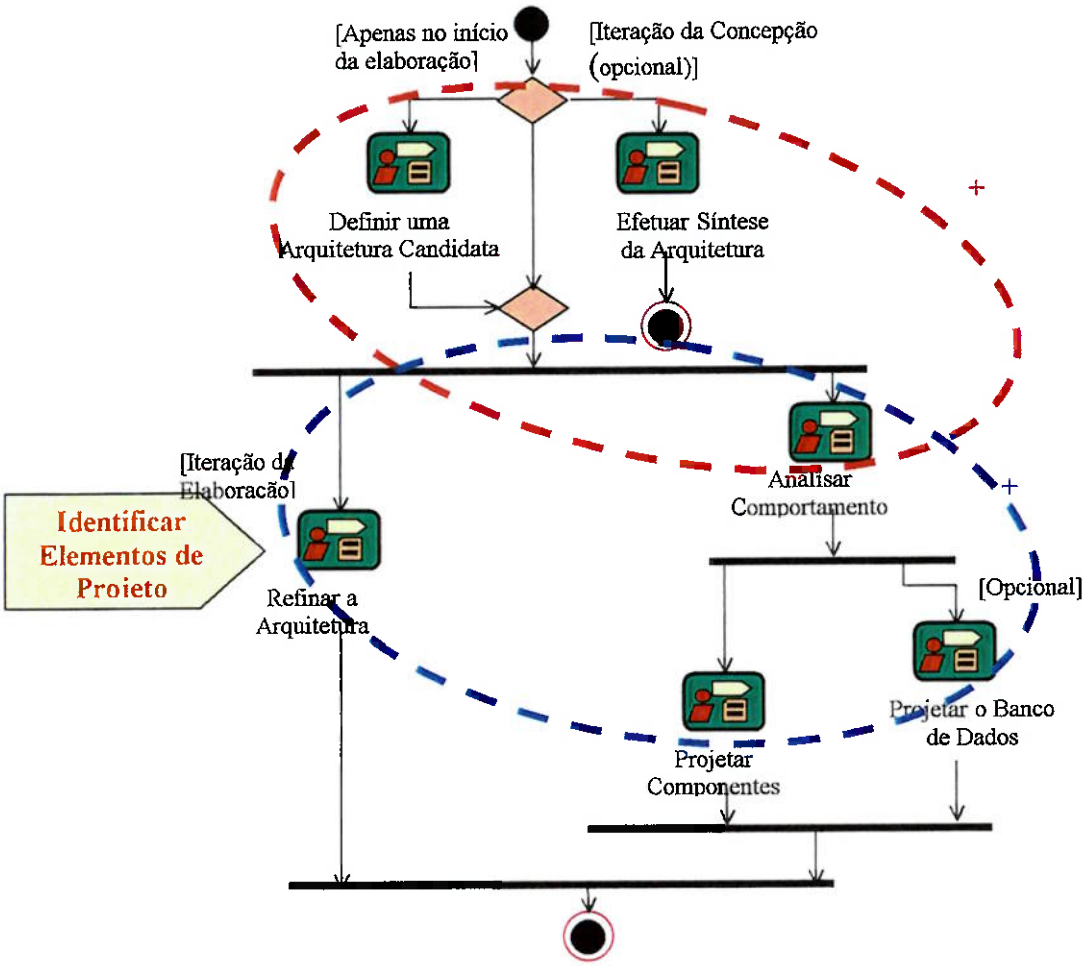


Figura 4-21 Workflow da disciplina Análise e Projeto, destacando a atividade Identificar Elementos de Projeto

4.9.1 Abordagem RUP

A figura a 4.21 apresenta o *workflow* da disciplina Análise e Projeto, e identifica a atividade Identificar Elementos de Projeto como parte do detalhe de *workflow* denominado Refinar a Arquitetura.

Esta atividade tem como fonte o modelo de análise previamente elaborado. A partir das interações entre classes de análise (colaborações que realizam casos de uso) do modelo de análise, serão identificados classes de projeto, subsistemas e interfaces.

Observa-se que as classes definidas na análise fazem parte de um modelo conceitual do sistema, e não necessariamente serão posteriormente implementadas. Essas classes, portanto, devem evoluir durante o projeto, e delas podem ser derivados os seguintes elementos de projeto:

- Classes de projeto – representam uma atribuição de responsabilidade em baixa granularidade;
- Subsistemas – representam atribuição de responsabilidade em alta granularidade;
- Classes ativas – representam *threads* de controle do sistema;
- Interfaces – representam declarações abstratas de responsabilidades providas por classes e subsistemas.

Uma classe de análise será diretamente mapeada em uma classe de projeto, se for relativamente simples e representar uma única abstração lógica. Classes de análise mais complexas podem: ser divididas em múltiplas classes; ser mapeadas em pacotes ou subsistemas; ou alguma combinação das ações anteriores.

4.9.2 Considerações Gerais

Definindo regras para a derivação a partir dos estereótipos de classe

As classes de análise definidas receberam estereótipos que as caracterizaram como sendo classes boundary, control ou entity. O RUP propõe essa classificação apenas para classes de análise, não para classes de projeto, que serão mais especializadas. Entretanto, a utilização desses estereótipos na análise cria uma oportunidade de padronização, já que é possível, para determinados ambientes ou plataformas, identificar regras gerais para a derivação de classes de projeto para cada tipo de classe de análise anteriormente definido. Este procedimento será adotado no caso da plataforma J2EE.

Classes de projeto são fortemente dependentes do ambiente de implementação

Uma mesma classe conceitual definida na análise pode ser mapeada em classes de projeto essencialmente distintas, em função do ambiente de implementação.

Classes Boundary, por exemplo: em uma aplicação convencional que utilize uma interface visual de janelas, serão derivadas em *forms* – elementos de interface que agregam botões, campos de entrada, listas, etc.; em uma aplicação web, serão derivadas em páginas HTML clientes (eventualmente com código JavaScript, que roda localmente, no browser cliente), ou em páginas dinâmicas processadas no servidor (na plataforma Java, seriam páginas JSP ou classes Servlets – para maiores detalhes, veja esses tópicos no capítulo 2), ou numa combinação desses elementos.

A especialização para J2EE associada a esta atividade foca, portanto, justamente em mapear cada tipo de classe definida nos tipos de classe de projeto específicos para a plataforma J2EE.

4.9.3 Especialização para J2EE

Classes Boundary

A função de um elemento da *view*, no modelo MVC (que corresponde às classes boundary), é formatar a interface a ser apresentada ao cliente. Os elementos estruturais, na plataforma J2EE, que têm a função de gerar a interface da aplicação, são JSPs (Java Server Page) e Servlets, descritos no capítulo 2. Portanto, uma classe boundary de análise deve ser derivada num JSP ou Servlet, no projeto.

Foi citado anterormente que um JSP, quando compilado (essa compilação é efetuada automaticamente a primeira vez que um JSP é acionado pelo container web), é convertido num Servlet. Apesar disso, JSPs e Servlets não são exatamente equivalentes, do ponto de vista da equipe de desenvolvimento. Um Servlet é uma classe Java convencional, que implementa determinadas interfaces e, portanto, deve implementar métodos de contrato dessas interfaces. Pode-se utilizar comandos e invocar objetos Java, num Servlet, para que seja gerada uma página HTML como resultado da execução do Servlet.

Um JSP, ao contrário, é uma página HTML na qual código Java foi embutido em pontos chave, para geração de conteúdo dinâmico. Diz-se que um Servlet é uma ferramenta de programador, enquanto um JSP é uma ferramenta de web designers. Por conta disso, é uma boa prática utilizar JSPs para gerar os elementos da interface visual, não Servlets, embora seja perfeitamente possível esta última alternativa. A

conclusão é que as classes boundary, no ambiente J2EE, devem ser derivadas em JSPs, durante o projeto.

Entretanto, modelar um JSP tem um complicador, poisele é um híbrido entre uma página HTML do lado cliente e lógica executada no servidor. Por conta disso, Khawar propõe que a página JSP seja representada como um par de elementos [KHAWAR 2002]:

- Uma *ClientPage* (ou seja, uma página HTML), que representa os aspectos de apresentação visíveis externamente do JSP. Essas páginas podem agregar outros elementos que rodam no cliente, como applets, código JavaScript, e forms, descritos abaixo:
    - Se a classe boundary contém informações de entrada (dados fornecidos pelo cliente ao sistema), ela seria derivada em um form web (um formulário para submissão de informações). Esse form necessariamente tem que estar agregado a uma página web convencional, que governa seu ciclo de vida. Forms são enviados para que os valores que contêm sejam processados por algum elemento no servidor, o que, em última instância, vai implicar no envio de outra página ao cliente;
    - Caso de deseje incluir comportamento dinâmico no próprio cliente, a página HTML cliente deverá conter operações, normalmente implementadas em JavaScript. A página cliente nesse caso não será estática, e terá um comportamento mais próximo ao de uma classe convencional, no sentido de conter operações e atributos;
    - Caso esse comportamento dinâmico a ser executado diretamente no cliente seja muito sofisticado, a classe boundary que representa o cliente (no caso, também uma página HTML) pode agregar uma classe applet (uma classe Java projetada para funcionar em browsers web).
- Uma *ServerPage*, que representa o comportamento do JSP no lado servidor, que consiste basicamente de lógica interna associada a processar uma requisição e gerar uma resposta. A resposta será justamente a página cliente citada anteriormente.

Classes de Controle

Os *patterns* J2EE, Front Controller e Session Façade, têm conjuntamente a solução para derivação de classes de controle em classes de projeto para o ambiente J2EE. Ambos os *patterns* foram apresentados no item referentes a patterns J2EE, neste capítulo, e a sua compreensão é necessária para entender a solução proposta neste trabalho.

A primeira parte da solução vem do *pattern* Front Controller. Esse *pattern* define uma estrutura na qual todas as solicitações oriundas de um cliente web são primeiramente tratadas por um elemento denominado Controller e, a seguir, enviadas para um Dispatcher. A idéia é que o Controller seja o ponto único de acesso de um cliente à aplicação, responsável por garantir que determinados serviços (como segurança), que terão que atuar a cada interação do cliente com a aplicação, sejam acionados a partir de um único elemento (o próprio Controller). Fica então caracterizada desta forma a responsabilidade do Controller.

Por outro lado, cada solicitação específica do cliente deve ter um tratamento também específico, uma vez que as questões comuns sejam resolvidas. Esse tratamento específico significa: acionar os componentes que encapsulam a lógica de negócio necessária para atender a solicitação; armazenar temporariamente as informações retornadas por esses componentes; acionar os elementos da camada de apresentação responsáveis por exibir ao usuário o resultado de sua solicitação disponibilizando a eles as informações retornadas. Pelo *pattern* Front Controller, este é o papel do Dispatcher.

O Dispatcher é um especialista em tratar um certo conjunto específico de solicitações, normalmente as referentes a um caso de uso específico. Há portanto, numa aplicação, um único Front Controller, e um Dispatcher para cada caso de uso. Mas a solução ainda não está completa nesse ponto e, para completá-la, será necessário o *pattern* Session Façade.

O *pattern* Front Controller define então dois elementos na camada web da aplicação: o Controller e o Dispatcher. Eles serão, geralmente, dois Servlets distintos (embora, em situações muito simples, as responsabilidades de ambos possam ser atendidas por

um único Serlvet). Falta ainda um elemento de controle, aquele que vai operar no container EJB.

O *pattern* Session Façade justifica a necessidade de que haja um Session Bean controlando cada interação entre objetos de negócio, necessária para atender às solicitações de componentes da camada web que atuam como clientes, acionando a funcionalidade no container EJB. Este é o elemento que faltava, para completar a tradução de classes de controle em elementos de projeto na plataforma J2EE. Nesta solução, o dispatcher web vai acionar um Session Façade, um session bean que coordenará a execução da lógica de negócio para atender à solicitação do cliente.

A conclusão é que cada classe de controle da análise será desdobrada, no projeto, em três elementos: um Front Controller (Servlet), um Dispatcher (Servlet), e um Session Façade (session bean).

Em princípio, haverá um único Front Controller para toda a aplicação, e um Dispatcher e um Session Façade para cada caso de uso.

Talvez a solução a princípio pareça complexa, mas ela respeita o *pattern* de arquitetura MVC e contempla uma definição clara de responsabilidades:

- O Front Controller funciona como ponto único de acesso, e executa serviços que precisam ser acionados a cada invocação da aplicação pelo usuário;
- O Dispatcher é o responsável, no ambiente web, por coordenar o atendimento da solicitação, acionando o Session Façade; recebendo e adequando as informações devolvidas pelo Session Façade; selecionando e passando informações para o elemento da camada de apresentação (um JSP) que apresentará o resultado da solicitação ao cliente.

A figura 4.22 apresenta a solução encontrada para mapear, em projeto, as classes de análise do tipo controle.

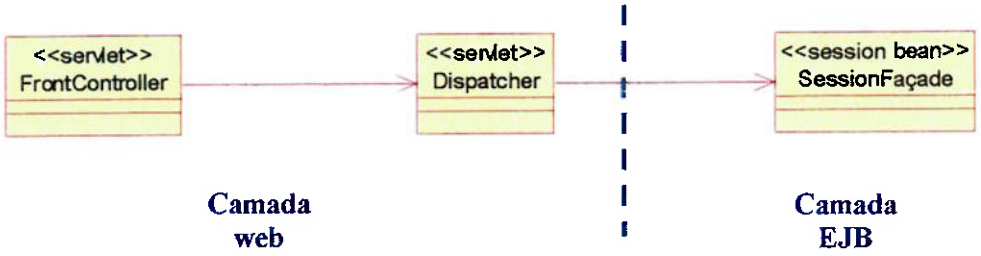


Figura 4-22 Derivação para projeto de uma classe de análise do tipo controle

Classes Entidade

A derivação de uma classe de análise do tipo Entidade em elementos de projeto para a plataforma J2EE é mais simples que no caso das classes de controle, mas também em relação a esse ponto há algumas considerações importantes.

Cada classe Entidade é candidata a ser representada como um enterprise bean. Como uma regra geral pode-se considerar que, caso a classe deva ser persistida, ela deve ser derivada em um Entity Bean; caso contrário, em um Session Bean. Desta forma, uma classe CarrinhoDeCompras seria convertida num session bean, enquanto que uma classe PedidoDeCompra seria convertida num entity bean.

A regra é bastante válida, mas há outras questões: em [DEEPAK 2001] são definidas algumas más práticas de engenharia de software, no que se refere ao projeto de entity beans. Essencialmente, deve-se converter automaticamente cada classe em um entity bean específico e; converter cada tabela do banco de dados em um entity bean específico.

Vale lembrar, no entanto, que um *enterprise bean* (dos quais um dos tipos é o *entity bean*) é um componente, não uma classe. Um *enterprise bean*, ainda que modele conceitualmente uma única classe de análise, necessariamente será, internamente, representado por diversas classes (o *home object*, o *remote object* e o *bean* propriamente dito – mais detalhes no capítulo 2). Mais ainda, um *enterprise bean* pode conter várias classes de análise.

A definição de *enterprise bean* o caracteriza como um componente de negócio de alta granularidade, não uma classe com granuralidade baixa.

O pattern GRASP Creator define responsabilidades pela instanciação de objetos: se uma classe B agrega ou contém objetos de uma classe A, B é candidato a ser criador de A. Nessa situação, recomenda-se em [DEEPAK2001] que ambas as classes componham um único *entity bean*. B será a classe principal do *entity bean*, e A será a classe dependente. O motivo desta definição de projeto é, na verdade, performance, visto que o consumo de recursos associado ao acesso a entity beans é muito maior que a uma classe convencional. Acessos a *entity bean* são remotas, e implicam em maior consumo de recursos mesmo que os dois *entity beans* estejam no mesmo container (o fabricante do EJB pode definir meios para otimizar acessos desse tipo, mas isso não faz parte da especificação J2EE).

Visualização de um modelo de solução com objetos boudary, control e entity

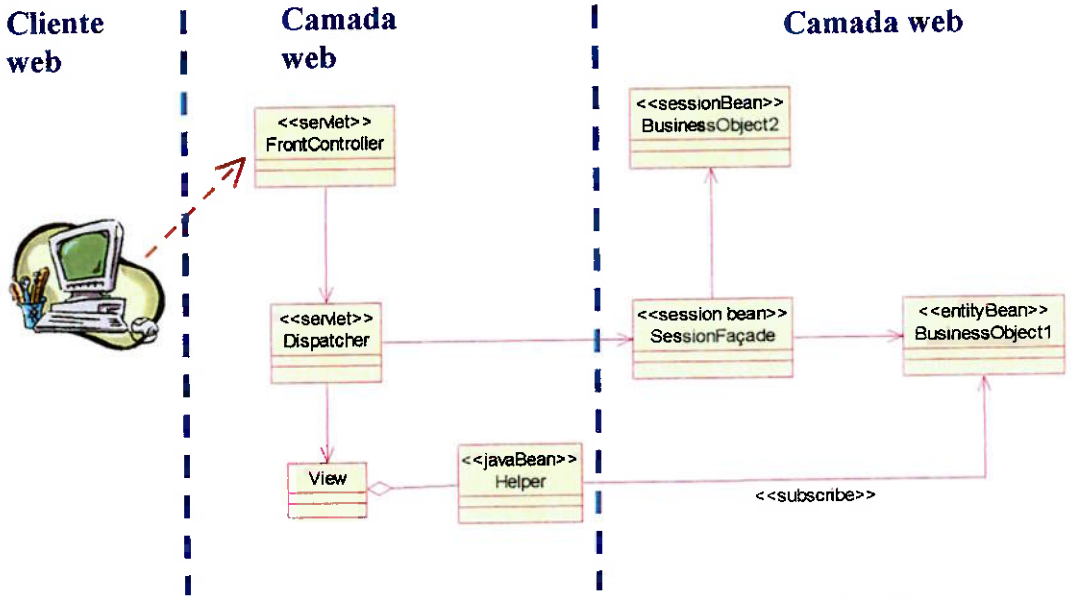


Figura 4-23 Solução para derivação de análise para projeto J2EE

A figura 4.23 apresenta um esquema de comunicação entre objetos, para o ambiente J2EE, que segue as definições elaboradas neste tópico. Os nomes das classes/componentes representam o papel que essas classes/componentes desempenham, e o estereótipo representa o tipo de componente J2EE de cada classe.

O cliente se comunica com um Servlet FrontController, ponto único para acesso do cliente à aplicação. O FrontController então aciona o Dispatcher adequado à solicitação. Este deve gerenciar, no ambiente web, o atendimento à solicitação do cliente. O Dispatcher sabe qual SessionFaçade da camada EJB deve ser ativado, e o ativa (eventualmente passando argumentos). Esse SessionFaçade aciona os objetos de negócio que têm a inteligência para atender à solicitação do cliente. Finda a interação na camada EJB, o SessionFaçade devolve, para o Dispatcher, os objetos *Helper* (veja *pattern* Front Controller, para maiores detalhes sobre objetos *Helper*) que contêm as informações que devem ser apresentadas ao cliente. De posse dessas informações, o Dispatcher aciona o objeto *View* – um JSP – especializado em gerar a página cliente que será exibida ao cliente com o resposta à sua solicitação.

4.10 *Identificar Mecanismos de Projeto*

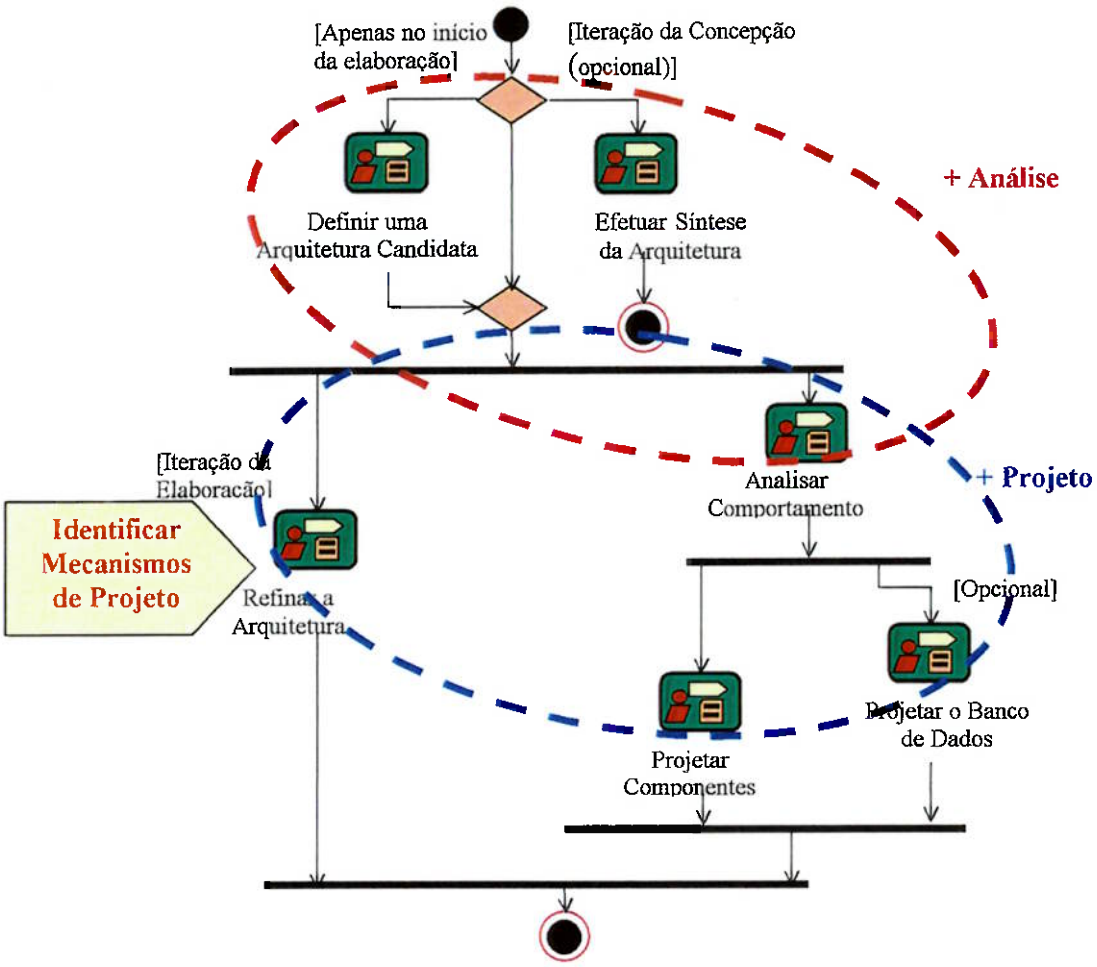


Figura 4-24 Workflow da disciplina Análise e Projeto, destacando a atividade Identificar Mecanismos de Projeto

4.10.1 Descrição da atividade segundo o RUP

Os conceitos referentes aos mecanismos de arquitetura, de análise, de projeto e de implementação foram abordados no capítulo 3.

A Atividade Identificar Mecanismos de Projeto faz parte do detalhe de *workflow* Refinar a Arquitetura, conforme pode ser visto na figura 4.24.

Nesta atividade os mecanismos de análise são refinados sob a forma de mecanismos de projeto, com base nas restrições impostas pelo ambiente de implementação. O RUP enfatiza o fato da abordagem desta disciplina ser, ao mesmo tempo. top-down

(análise → projeto) e bottom-up (implementação → projeto). Os mecanismos foram identificados/concebidos na análise, mas não podem ser projetados sem que o ambiente de implementação seja considerado.

A seqüência de passos definida pelo RUP para esta atividade é:

- Categorizar os clientes dos mecanismos de análise: para cada mecanismo de análise, identificar as classes que utilizarão os mecanismos de análise, e caracterizar de que forma essas classes utilizarão cada mecanismo, de modo a identificar perfis comuns de utilização dos mecanismos de análise.
- Pesquisar os mecanismos de implementação: pesquisar que mecanismos de implementação poderiam ser utilizados. Nesse momento está se seguindo uma abordagem bottom-up. São pesquisados mecanismos de implementação, e deles se chega a mecanismos de projeto. Pode haver também vários mecanismos de implementação disponíveis que podem ser associados a um único mecanismo de projeto. Exemplo: o sistema operacional pode ter uma solução, um produto de middleware também disponível pode dispor de outra, eventualmente há uma terceira solução, implementada localmente. Um deles deverá ser selecionado para implementar o mecanismo de projeto correspondente.
- Documentar os mecanismos: documentar do artefato Diretrizes de Projeto (textualmente e como colaborações UML) os mecanismos de projeto e, para cada mecanismo de projeto identificado, o mecanismo de implementação correspondente.

4.10.2 Considerações Gerais

O título desta atividade, Identificar Mecanismos de Projeto, pode passar a falsa impressão de que ela trata especificamente de mecanismos de projeto, e que mecanismos de implementação são abordados em outra atividade (talvez uma atividade na disciplina de implementação). No entanto, essa atividade trata do detalhamento de mecanismos de análise previamente identificados, tanto mecanismos de projeto, quanto seu refinamento como mecanismos de implementação. A seqüência de passos que descreve a atividade, apresentada na seção 4.10.1, deixa isso claro.

Uma outra questão a ser considerada é que o RUP afirma que mecanismos de análise, projeto e implementação deveriam ser descritos em termos de colaborações UML. Mas, muitas vezes, não há o que representar como colaboração, quando se aborda um mecanismo de análise, que é uma definição muito conceitual. Mesmo o mecanismo de projeto pode ainda não se caracteriza de forma que possa ser modelado visualmente. No próximo tópico, quando forem desenvolvidos alguns mecanismos para a plataforma J2EE, será visto que a colaboração é representada pelo mecanismo de implementação. Aquilo que o RUP denomina mecanismo de implementação, é, na verdade, o projeto de um mecanismo voltado a um ambiente de implementação específico.

A plataforma J2EE comporta a possibilidade de definição de diversos mecanismos de projeto, associados a serviços que as APIs J2EE provêem, e que podem ser projetados de forma padronizada. Na análise da arquitetura identificam-se alguns desses mecanismos – segurança, persistência, distribuição –, com a ressalva de que outros mecanismos poderiam ter sido identificados, inclusive alguns possivelmente menos gerais, úteis em situações mais específicas. A proposta desta dissertação não é esgotar o tema, que poderia ser objeto único de um trabalho de pesquisa. Efetua-se uma breve análise dos mecanismos de análise identificados e seleciona-se um deles para efetuar a derivação em mecanismo de projeto / implementação para a plataforma J2EE.

A definição de um mecanismo de projeto / implementação referente à distribuição é muito direta, bastando descrever o funcionamento já previsto nas APIs JNDI (para identificar objetos a partir do nome) e RMI (para acessar objetos remotos identificados utilizando recursos de JNDI). A definição deste mecanismo agregaria pouco valor aos projetistas / programadores que conheçam as duas APIs.

A definição de um mecanismo específico de projeto / implementação referente à segurança, por outro lado, é inadequada. Há muitas abordagens distintas possíveis para segurança e cada Organização provavelmente vai definir uma abordagem adequada às suas necessidades. Portanto, embora seja perfeitamente válido definir mecanismos de segurança, estes seriam mais úteis no âmbito de um projeto em particular.

Resta o mecanismo de persistência. A abordagem de projeto e implementação desse mecanismo também poderia ser considerada direta (apenas registrar o funcionamento da API JDBC), como já foi dito em relação ao mecanismo de distribuição. Entretanto, o mecanismo de persistência pode ser tornado mais interessante e útil caso se considere para o seu detalhamento o pattern DAO, apresentado previamente neste capítulo, e que define um nível de indireção para acesso às fontes de dados (no caso, acesso a um banco de dados relacional). A especialização para J2EE, portanto, vai abordar o mecanismo de persistência.

4.10.3 Especialização para J2EE

Em princípio, o mecanismo de análise pode ser derivado em mais que um mecanismo de projeto: poder-se-ia considerar a utilização de um gerenciador de banco de dados orientado a objetos como sendo o mecanismo de projeto associado a persistência, ou ainda um gerenciador de banco de dados relacional, ou ainda arquivos seqüenciais. Na realidade atual, certamente o mecanismo de projeto mais importante – derivado do mecanismo Persistência de Análise – é aquele que contemple um gerenciador de banco de dados relacional. Mas não seria como o único caso, pois em alguns casos, poderia ser interessante persistir alguns objetos em arquivos seqüenciais, e não numa base relacional (objetos que contém parametrizações gerais do aplicativo ou do ambiente, por exemplo), ou elaborar mecanismos de persistência voltados para acesso a sistemas legados.

Definido o mecanismo de persistência do ponto de vista de projeto (um banco de dados relacional), é necessário identificar o mecanismo de implementação correspondente. No caso da plataforma J2EE, isso já está pré-definido: a API JDBC tem exatamente o objetivo de permitir uma abordagem padrão para acesso a bancos de dados relacionais.

A API Java DataBase Connectivity foi citada brevemente, no capítulo 2. O seu objetivo, assim como das demais APIs de serviços J2EE, é definir uma interface padrão para determinado serviço (neste caso, comunicação com um banco de dados), de modo que soluções diferentes, fornecidas por fabricantes de software diferentes, possam ser utilizadas sem que o código da aplicação tenha que ser alterado por conta disso. Diz-se que o serviço é *vendor-neutral*. A JDBC permite:

- Efetuar uma conexão e autenticação a um servidor de banco de dados;
- Gerenciar transações;
- Mover comandos SQL para um gerenciador de banco de dados, para pré-processamento e execução;
- Executar stored procedures;
- Inspecionar e modificar os resultados de comandos SELECT.

A seguir, apresenta-se um padrão (não no sentido formal) de uso do mecanismo de persistência JDBC, que considera o pattern DAO.

A figura 4.25 apresenta o diagrama de classes que mostra uma visão estrutural desse padrão.

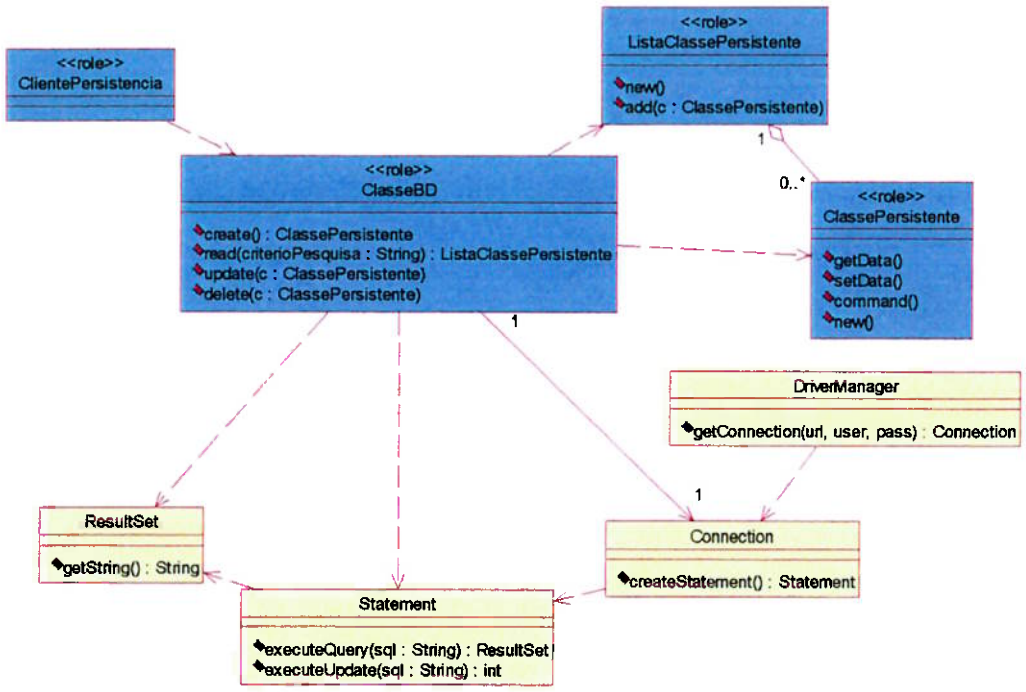


Figura 4-25 Estrutura do mecanismo de implementação com base em JDBC e pattern DAO

As classes desse diagrama que possuem o estereótipo << role>> não são classes concretas, mas papéis que serão desempenhados por classes específicas de

determinada implementação. As demais classes são classes Java, que compõem a API JDBC.

A classe denominada ClientePersistencia é, como o nome indica, a classe que solicita algum serviço de persistência. No pattern DAO, esta classe é denominada Objeto de Negócio, podendo ser session bean, entity bean ou outra classe Java que precisa persistir informações ou acessar informações persistidas.

ClasseBD é a classe que implementa as chamadas JDBC que realizam o serviço solicitado e existe uma ClasseBD para cada ClassePersistente. A ClassePersistente é a entidade e a ClasseBD tem a inteligência para efetuar serviços de persistência sobre ClassePersistente. A ClasseDB é aquela denominada DAO, no *pattern* DAO.

ClasseBD se comunica com a classe DriverManager, e obtém dela uma conexão necessária para que o banco de dados possa ser acessado. Em JDBC, a conexão é uma instância da classe Connection.

Obtida a conexão, ClasseBD monta o comando SQL, que será enviado ao gerenciador de banco de dados. Isso é feito utilizando uma classe Statement. O resultado da execução do comando SQL é retornado num objeto da classe ResultSet.

Falta comentar uma classe: ListaClassePersistente. Caso o resultado de uma pesquisa seja não uma instância da ClassePersistente, mas várias, elas serão armazenadas na classe do tipo coleção ListaClassePersistente.

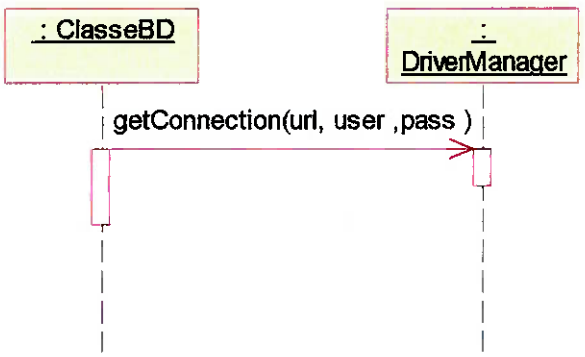


Figura 4-26 Obtendo uma conexão ao banco de dados

Para que seja possível o acesso à base de dados, é necessário que seja estabelecida uma conexão com a base de dados. O diagrama apresentado na figura 4.26 ilustra essa dinâmica. ClasseBD carrega o driver apropriado para acesso ao banco de dados

executando a operação getConnection de DriverManager, passando como parâmetro uma url, e uma identificação de usuário. A operação getConnection seleciona o driver adequado e devolve a ClasseBD um objeto de conexão com o banco de dados (Connection).

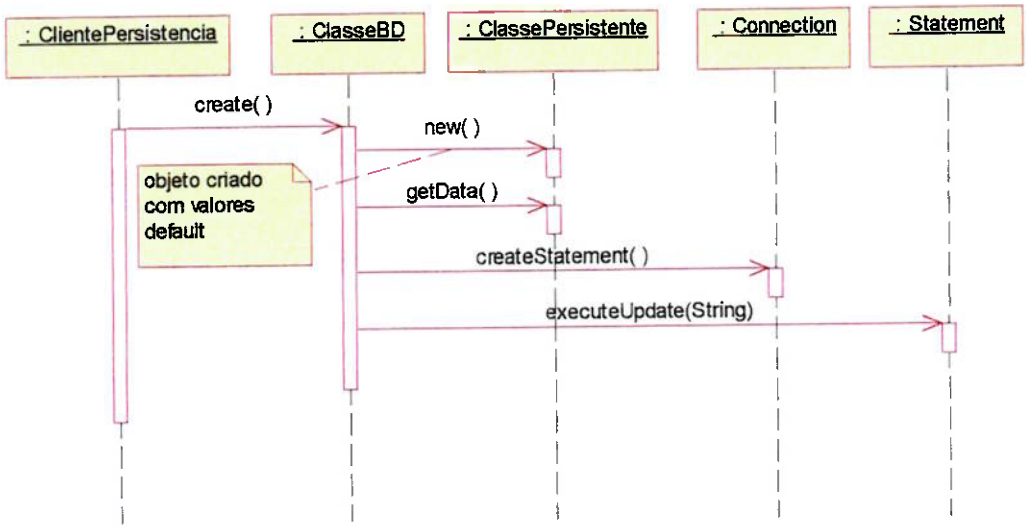


Figura 4-27 Mecanismo JDBC para criação de um objeto

A figura 4.27 apresenta a dinâmica para criar um novo objeto persistente, que será incluído na base de dados.

O cliente da criação do objeto persistente aciona a ClasseBD, que instancia o objeto com valores default. A seguir ClasseBD obtém os valores dos atributos de ClassePersistente, aciona o objeto de conexão (previamente obtido) para criar o Statement, e executa o método executeUpdate de Statement passando como parâmetro o string SQL responsável por inserir na base os dados do objeto persistente criado.

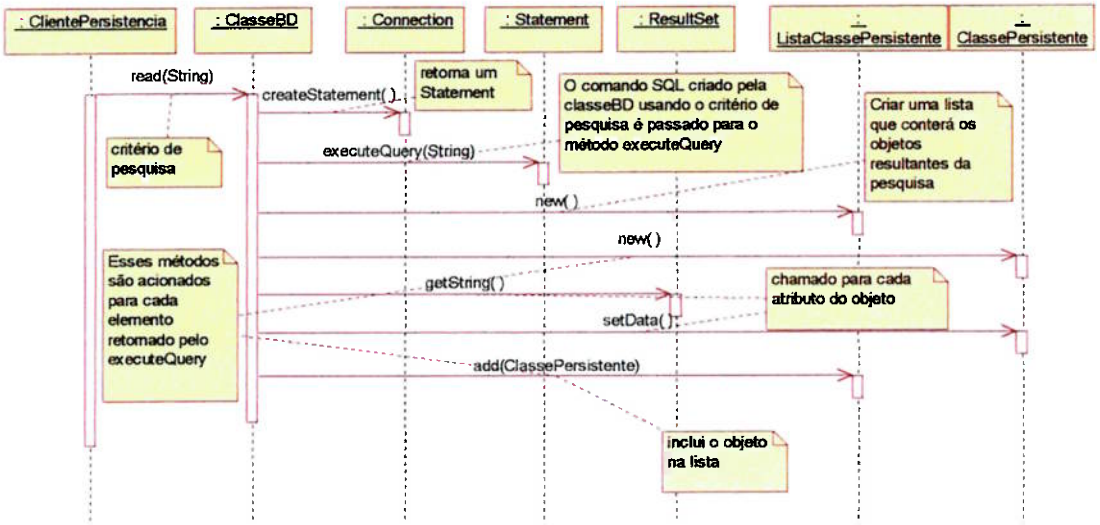


Figura 4-28 Leitura de um objeto, pelo mecanismo JDBC

A figura 4.28 apresenta a dinâmica referente à obtenção (leitura) de objetos persistentes armazenados na base. Os comentários que constam no próprio diagrama explicam o seu funcionamento.

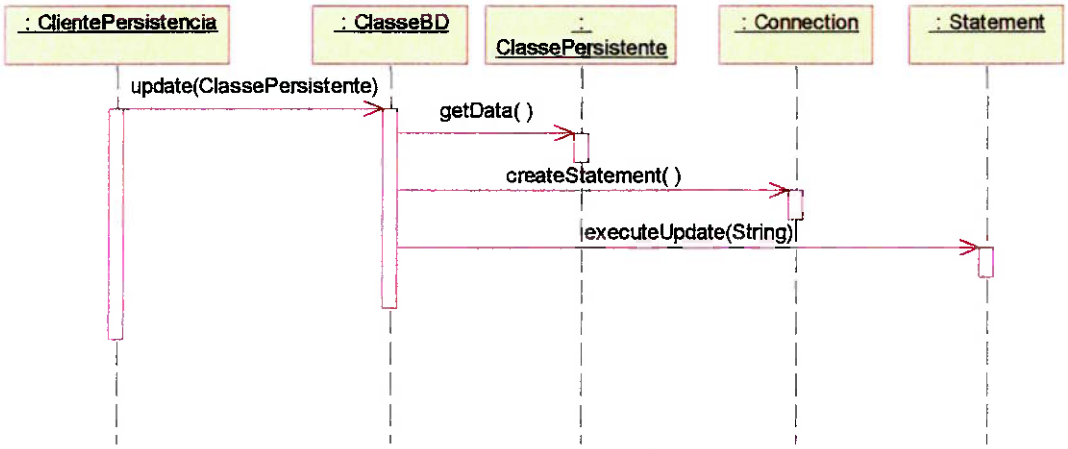


Figura 4-29 Atualização de um objeto, pelo mecanismo JDBC

A figura 4.29 apresenta o diagrama referente à atualização de um objeto. ClientePersistencia que aciona ClasseBD, passando como parâmetro a instância de ClassePersistente que deve ser atualizada na base de dados.

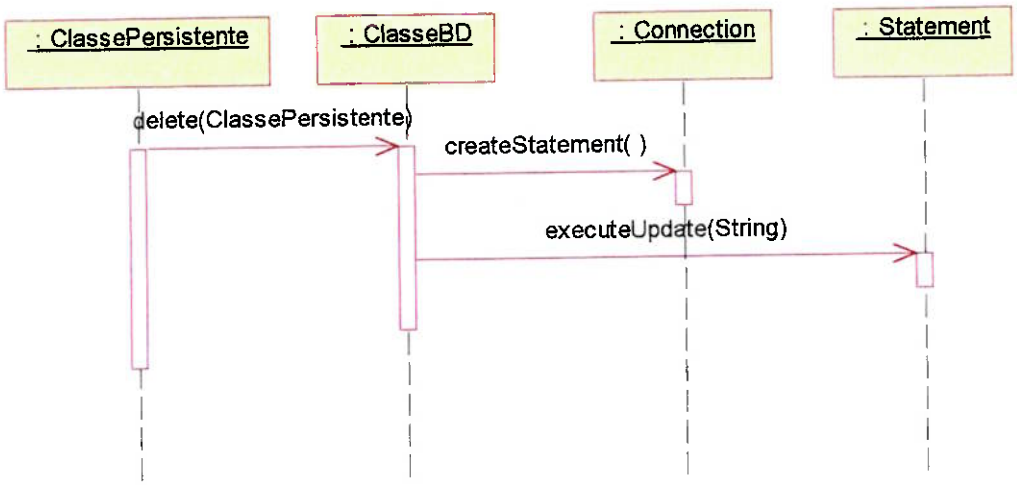


Figura 4-30 Utilizando o mecanismo JDBC par a excluir um objeto

Por fim, a figura 4.30 apresenta o diagrama referente à exclusão das informações de um objeto na base de dados. O *statement* executado evidentemente exclui os dados da base de dados.

4.11 *Considerações Finais*

Os elementos estruturadores definidos neste capítulo certamente prestam uma efetiva contribuição para o projeto de aplicações J2EE, simplificando a atividade de projeto e, simultaneamente, agregando qualidade a ele.

No capítulo 6, de considerações finais, é efetuada uma análise mais detalhada sobre os benefícios dessa abordagem.

5. Estudo de Caso: Sistema Genetec

5.1 *Introdução*

Este capítulo tem, como objetivo, apresentar uma visão prática de análise e projeto para a plataforma J2EE, que mostre a utilização dos elementos estruturadores (arquitetura de software, mecanismos, derivações de análise para projeto) definidos no capítulo 4, como fruto da especialização do RUP para esta plataforma. Para isso, foi elaborado o projeto parcial de uma aplicação fictícia, mas para a qual existe demanda real por empresas da área de biologia molecular. Uma dessas empresas – à qual será atribuída a denominação fictícia de Genetec –, especializada em exames de animais (aves, especificamente) se propôs a contribuir, disponibilizando profissionais para entrevistas que serviram como base para a especificação de requisitos elaborada. A Genetec tinha uma efetiva necessidade de um aplicativo empresarial que funcionasse no ambiente web, bem dentro dos tipos de aplicação que deveriam ser objeto do estudo de caso. Isso trouxe a oportunidade de produzir uma especificação essencialmente verossímil e objetiva.

Embora o foco deste estudo de caso seja a disciplina análise e projeto, a abordagem necessariamente tem que mais ampla. Para que a disciplina Análise e Projeto possa ser executada, é necessário que os requisitos do sistema a ser projetado tenham sido definidos em grau de detalhe suficiente e com a adequada precisão. Por isso, a seguinte abordagem foi adotada:

- Foi definido o processo de negócio que será parcialmente automatizado pela aplicação. Para isso, foi gerado um caso de uso de negócio, que é um artefato previsto na disciplina Modelagem de Negócio do RUP;
- Foi produzido o Modelo de Domínio, um outro artefato da disciplina Modelagem de Negócio, muito útil para a análise e projeto;
- Da disciplina Gerência de Requisitos, elaborou-se um artefato denominado Visão, que contém a visão em mais alto nível do aplicativo a ser desenvolvido;

- Também da disciplina de Gerência de Requisitos, elaborou-se um Modelo de Casos de Uso do sistema, e descreveram-se sucintamente os casos de uso mais importantes do sistema em desenvolvimento;
- Alguns termos essenciais do domínio são descritos em um Glossário textual;
- Um dos casos de uso de sistema foi descrito textualmente em detalhes.

Esses artefatos formam as camadas do alicerce para que a disciplina Análise e Projeto possa ser executada. Os artefatos da disciplina de Modelagem de Negócio constituem a base do alicerce. O documento de Visão fica na camada acima e, sobre esta camada, detalham-se os requisitos sob a forma de casos de uso.

Definido o alicerce, pôde-se desenvolver a análise e projeto. O objetivo deste capítulo é apresentar a aplicação da especialização do RUP para a plataforma J2EE, e não era projetar toda a aplicação; por isso, o desenvolvimento foi feito sobre um caso de uso relevante, para a aplicação dos resultados do trabalho. Esta abordagem foi adotada, pois a realização de outros casos de uso não acrescentaria elementos relevantes aos objetivos do capítulo.

Cada item do capítulo descreve sucintamente os objetivos de um tipo de artefato contemplado pelo estudo de caso, apresenta o conteúdo do artefato produzido e, quando são abordados artefatos na Análise e Projeto, tece considerações sobre o artefato e sobre os elementos estruturadores que ele contempla.

A seqüência em que as disciplinas estão definidas no RUP não será respeitada na apresentação dos artefatos, pois considera-se que apresentar inicialmente o artefato Glossário é mais elucidativo, porque termos utilizados em todo o capítulo ficam previamente esclarecidos.

Além disso, apesar de uma leitura do Caso de Uso de Negócio e do Glossário, em princípio, ser suficiente para esclarecer o significado dos elementos do modelo de domínio, a seguir é apresentada uma descrição textual informal dos elementos do modelo de domínio.

O *site web* da Genetec é um canal de comunicação para que os clientes façam solicitações de *kits* de coleta; solicitações de exames, acompanhem o andamento das solicitações e acessem resultados dos exames.

A Companhia envia gratuitamente, para qualquer solicitante, o *kit* com recipientes para coleta do material a ser examinado (sangue, penas ou unhas, normalmente). Apenas os clientes cadastrados podem efetivamente solicitar exames. O conceito fundamental é o de Solicitação de Exame, que agrega um conjunto de solicitações de exames individuais sobre determinados animais. O exame mais comum é o de sexagem, que consiste em determinar o sexo de certas aves ainda muito jovens, quando não há meios de determinar o sexo por observação visual.

Cada item de solicitação é associado a um item do *kit* de coleta, para que o resultado de um exame seja atribuído ao animal correto, que produziu a amostra utilizada.

Uma Bateria de Exames é uma definição de um conjunto de exames (eventualmente de clientes diferentes) que serão agrupados e efetuados em conjunto. Isso se deve ao fato de que é possível realizar, a cada vez, um conjunto de exames simultaneamente (exemplo: 96 exames de sexagem podem ser efetuados simultaneamente, pelo aparelho que realiza esse tipo de exame). A Bateria tem uma função administrativa, de planejamento. Serve para que seja selecionada uma certa quantidade de exames, de uma ou mais solicitações, que serão efetuados em conjunto.

Os aparelhos que efetuam exames geralmente dispõem as amostras em forma de *grid*. A identificação correta de cada elemento do *grid* é fundamental para que se possa garantir que o resultado do exame seja atribuído ao animal correto. Cada tipo de exame pode ser realizado por uma ou mais técnicas de exame. Cada técnica geralmente está associada a um aparelho no qual é realizada. O aparelho utiliza um certo formato de *grid*.

5.2 *Artefato: Glossário*

5.2.1 Introdução

O artefato Glossário é parte da disciplina Gerência de Requisitos. Termos importantes do domínio do problema são definidos de forma concisa e clara no Glossário. Isso evita ambigüidades e é extremamente útil quando a equipe de desenvolvimento não é fluente no domínio do problema. A seguir, apresenta-se o Glossário elaborado, referente ao sistema de análise laboratorial.

5.2.2 Glossário

Ciclo de Exames

Ciclo que abrange a solicitação de exames por um cliente, o envio de material de coleta, a realização dos exames e a divulgação do resultado desses exames ao cliente.

Kit de coleta

O kit de coleta é dependente do tipo de exame solicitado e pode ser composto de:

- Tubo de coleta com meio conservante com etiqueta para identificação da Ave, Espécie e ID. Essa etiqueta contém, pré-impresso, um Número Genetec, utilizado para identificar o exame;
- Fita de coleta com identificação da Ave, Espécie e ID. Essa fita contém, pré-impresso, um Número Genetec, utilizado para identificar o exame;
- Ficha de solicitação de exame. Nesta ficha informa-se, para cada exame, a identificação da Ave, Espécie e Número Genetec.

Bateria de exames

Uma bateria contempla uma certa quantidade de exames efetuados segundo uma determinada técnica. Os exames são oriundos de uma ou mais solicitações. A quantidade de exames da bateria corresponde àquela que pode ser efetuada em um *grid* para a técnica contemplada.

Ao contrário de um *grid*, uma bateria não se refere a amostras específicas. Uma bateria define apenas a quantidade de exames e a solicitação ou solicitações que os originou.

Exemplo:

Uma bateria denominada Sexagem Técnica A 27.03.2002 poderia ter como conteúdo:

- 30 exames de sexagem pela técnica A oriundos da solicitação 1234 da empresa X;
- 66 exames de sexagem pela técnica A oriundos da solicitação 1236 da empresa Y.

Essa bateria contempla, então, ao todo, 96 exames, que seria a quantidade correspondente ao *grid* para a técnica A.

Canais de comunicação com o cliente

Os canais que o cliente pode utilizar para enviar solicitações e se comunicar com a Genetec podem ser de dois tipos:

- Ligados ao sistema: Interface internet ou intranet.
- Não ligados ao sistema: Quaisquer mecanismos de comunicação com a empresa que não alimente diretamente o sistema de gerência de exames, como telefone, fax, correios.

Grid

Um *grid* é uma matriz de duas dimensões (linhas e colunas), no qual cada célula armazena informações sobre uma determinada amostra. As informações essenciais sobre cada amostra são sua identidade (número Genetec) e o resultado do exame. O *grid* como um todo também tem atributos, como o número de linhas e colunas, o exame e a técnica associados, e a bateria associada.

Durante a montagem do *grid*, o sistema alerta para erros de técnica (se o técnico, por engano, tentar incluir um item num *grid* que é referente a uma outra técnica).

Solicitação de exame

Solicitação preenchida pelo Cliente que contempla o número da solicitação (definido pela Genetec), a data em que foi efetuada, a data em que foi concluída (se já concluída), a identificação do cliente e, para cada exame, os seguintes detalhes:

- Gênero/espécie do animal
- Identificação do animal
- Número Genetec
- Exame a ser realizado
- Prazo de entrega (para tipos de exames que podem ter mais de um prazo de entrega).

Estados de uma solicitação:

- Pendente – solicitação enviada pelo Cliente, mas ainda não validada pela Genetec contra o material de coleta.
- Validada – solicitação já verificada contra o material de coleta, pronta para execução dos exames.
- Incompleta – solicitação que contém apenas itens cujo processo de validação identificou problemas. A secretária pode criar uma ou mais solicitações incompletas a partir de uma solicitação de cliente. Essas solicitações incompletas contemplarão os itens não identificados corretamente (item do material de coleta que não consta na solicitação e vice-versa).
- Em execução – já associada a uma ou mais baterias de exame, mas na qual ainda não foram registrados os resultados de todos os exames.
- Cancelada – solicitação cancelada por qualquer motivo que não demanda mais ações da Genetec.
- Concluída – solicitação na qual os resultados de todos os exames já foram registrados.
- Enviada – solicitação concluída e na qual o resultado de todos os exames já foi enviado pelo correio.

5.3 *Artefato: Caso de Uso de Negócio*

5.3.1 Introdução

Este é um artefato da disciplina Modelagem de Negócio e, conforme citado no capítulo 3, a descrição textual de um Caso de Uso de Negócio define o fluxo de trabalho de um processo de negócio. A estruturação de um caso de uso de negócio é em tudo semelhante a dos casos de uso de sistema (normalmente denominados simplesmente casos de uso), mas os objetivos são bem distintos. Os casos de uso de negócio descrevem o processo de negócio e os casos de use de sistema descrevem o funcionamento do software.

A elaboração do caso de uso de negócio pode ter, como objetivo, apenas a obtenção de conhecimento sobre o processo de negócio, que é necessário para que se possa definir os requisitos de software da aplicação. No caso específico do aplicativo para a Genetec, havia a necessidade de alterar o processo de negócio então vigente, para melhorar a sua eficiência. O caso de uso de negócio apresentado a seguir descreve o processo de negócio desejado, e não o atualmente em vigor.

A aplicação, objeto do estudo de caso, é relativa à realização de exames laboratoriais e automatiza parcialmente o processo de negócio fundamental da Genetec. O caso de uso de negócio apresentado a seguir descreve esse processo de negócio.

5.3.2 Descrição Sucinta

Este caso de uso de negócio descreve o processo de negócio referente à realização de exames: desde a solicitação pelo cliente, passando pelos procedimentos administrativos e técnicos realizados na Companhia para atender à solicitação, até a divulgação dos resultados.

5.3.3 Workflow Básico

A. Cliente solicita Kit

O processo começa quando o cliente solicita o envio de um *kit* com material de coleta. O cliente deve informar quais exames deseja realizar e a quantidade de cada tipo de exame. Para efetuar a solicitação, os clientes cadastrados se identificam; caso não queiram se cadastrar, os clientes potenciais podem apenas informar dos dados necessários para envio do *kit*.

Ao receber uma solicitação, a Genetec monta o ***kit*** solicitado e o envia ao cliente.

B. Cliente efetua a Solicitação

Após receber o ***kit* de coleta** (ver glossário), o Cliente preenche uma Solicitação de Exame (ver glossário), identificando cada exame desejado e, para aqueles tipos de exames que estão que têm opção de prazo de entrega, o prazo de entrega do exame (naturalmente, o preço do exame é associado ao prazo de entrega). Somente os clientes cadastrados podem solicitar exames; assim, se um cliente ainda não se cadastrou como cliente potencial, deve se cadastrar.

Essa solicitação fica com status Pendente até que seja validada pela Genetec.

C. Cliente envia material de coleta pelo correio

O Cliente envia pelo correio o material de coleta referente aos exames solicitados.

D. Companhia valida as informações contidas na solicitação contra o material de coleta

Ao receber o *kit* com material coletado pelo Cliente, a Companhia pesquisa a Solicitação de Exame correspondente, utilizando como chave de pesquisa o Número Genetec de algum item do material de coleta. Uma vez identificada a solicitação, os itens da Solicitação de Exame são validados contra o material recebido pelo correio. Essa validação consiste em verificar se cada item da solicitação tem o material de coleta correspondente corretamente identificado, e se para cada item do material de coleta há uma identificação correta na Solicitação (gênero/espécie, exame solicitado e identificação da ave).

Os itens da solicitação que foram aprovados, durante a validação, passam a compor uma solicitação com o status Validada. Se houver itens não aprovados, estes comporão uma outra solicitação com status Incompleta.

A secretária deve separar os exames por tipo de exame e técnica, para facilitar posteriormente a montagem das grades pelo técnico.

E. Companhia organiza bateria de exame

O técnico define determinados critérios de pesquisa de exames das solicitações Validadas, considerando os dados relativos a cliente, data de chegada, tipo de exame, quantidade de exames, prazo de entrega. O sistema exibe uma lista com os exames que atendem ao critério de pesquisa. O técnico pode selecionar todos ou alguns dos exames dessa lista e definir uma bateria. A organização de uma bateria é um ponto fundamental para a execução efetiva dos exames; assim, a decisão de montar uma bateria, ainda que com poucos exames, tem implicação direta no prazo das solicitações envolvidas.

F. Companhia monta grid para exame

O técnico interage com o sistema para selecionar uma determinada bateria já montada e montar o *grid* com as amostras que comporão um certo exame (há casos em que um exame contempla 96 amostras).

A montagem do *grid* significa dispor os materiais de coletas de uma bateria em um suporte físico esquematizado pelo sistema. O técnico informa ao sistema o identificador de cada item de coleta da bateria e o sistema indica em que posição do *grid* deve ser colocado o item de coleta (isso para garantir que o registro do resultado do exame seja atribuído ao animal correto).

Durante a montagem do *grid*, o sistema alerta para erros de técnica (se o técnico, por engano, tentar incluir um item em um *grid* que é referente a outra técnica).

Observação: Se outros tipos de erros forem posteriormente idenficados, devem ser registrados neste ponto.

G. Companhia efetua exame

Uma vez montado um *grid* (associado a uma determinada bateria de exames), o técnico pode tomar as providências para retirar o material biológico de cada item do *grid*, colocar nas posições respectivas dos equipamentos adequados à execução do exame e efetuar concretamente o exame.

H. Companhia registra resultados

O técnico utiliza o sistema para registrar os resultados de uma bateria de exames. Quando o técnico termina de registrar os resultados, o sistema verifica cada solicitação envolvida. Se todos os exames de uma certa solicitação, que têm o mesmo prazo de entrega já tiverem resultado, a secretária é avisada para imprimir e enviar os resultados desses exames.

Quando o técnico registra o resultado de um exame, o Cliente passa a ter acesso ao resultado do exame via Internet, imediatamente.

I. Companhia informa resultados e emite certificação ao cliente

A cada grupo de exames de uma solicitação com prazo de entrega comum que tenha sido concluído, a secretária deve imprimir um certificado com o resultado de cada

exame e enviar o conjunto pelo correio para o cliente. Caso todos os exames de determinada solicitação já tenham sido efetuados, o sistema altera o status da solicitação para Enviada.

5.3.4 Workflows alternativos

Cliente solicita kit por canais não diretamente ligados ao sistema (telefone, carta, fax, etc)

No passo A do fluxo básico, caso o cliente solicite o *kit* por canais não ligados ao sistema, a companhia deve registrar, no sistema de gerência de exames, a(s) solicitação(ões) de *kit* do cliente, e o caso de uso continua no passo B do fluxo básico.

Cliente envia Solicitação de exames por canais não diretamente ligados ao sistema (solicitação impressa via correio, fax, etc)

No passo B do fluxo básico, caso um cliente não cadastrado envie a solicitação de exames por canais não ligados ao sistema, a companhia deve cadastrar o cliente no sistema e registrar, no sistema de gerência de exames, a solicitação de exame. O caso de uso continua no passo C do fluxo básico.

Cliente preenche Solicitação de exames na Internet sem ter o kit de coleta

No passo B do fluxo básico, caso o cliente tente enviar uma solicitação de exames via internet, sem ter em mãos o kit de coleta de material, ele não conseguirá, pois não disporá do número Genetec necessário. O caso de uso é encerrado.

Solicitação incompleta

No passo D do fluxo básico, caso existam solicitações com status incompletas por problemas com identificação e material de exames, a secretária deve tomar as providências necessárias para contatar o cliente e pedir o acerto de cada exame com problemas, e avisar ao cliente de que os exames especificados corretamente já estão em andamento.

Material de coleta não recebido

Ao início de cada expediente, a secretária deve identificar todas as solicitações em estado Pendente que foram efetuadas há mais de 20 dias (cujo material de coleta não foi recebido pelo correio). Deve providenciar um contato com o cliente para analisar a possibilidade de Cancelar a solicitação. Este fluxo é uma oportunidade de manter contato com o cliente, oferecer serviços e captar informações do mercado como exemplo pelas condições oferecidas pela concorrência.

5.4 *Artefato: Modelo de Domínio*

5.4.1 Introdução

O RUP apresenta duas abordagens em relação à Modelagem de Negócio: uma abordagem mais completa, na qual se define casos de uso de negócio e se realizam esses casos de uso através de um Modelo de Objetos de Negócio; outra mais simplificada, na qual se define apenas um ou alguns diagramas de classes conceituais, que representam elementos do domínio do problema e seus relacionamentos – este é denominado Modelo de Domínio.

No trabalho foi adotada uma abordagem híbrida, que também é possível. Definiu-se um Caso de Uso de Negócio (que não foi realizado em termos de diagramas UML) e elaborou-se um Modelo de Domínio. O caso de uso de negócio foi elaborado por ser o processo de negócio complexo e não conhecido dos projetistas do software. O modelo de domínio funciona como um Glossário gráfico, ainda que não substitua determinadas descrições textuais de um glossário típico, com a vantagem de que as classes conceituais nele definidas podem servir de base para a definição de classes de análise e de projeto. É, na verdade, extremamente conveniente que seja elaborado um modelo de domínio com esse intuito, pois é desejável que a abstração de software seja o mais fiel possível ao processo do mundo real.

O modelo de domínio elaborado pode ser visto na figura 5.1.

5.4.2 Modelo de Domínio

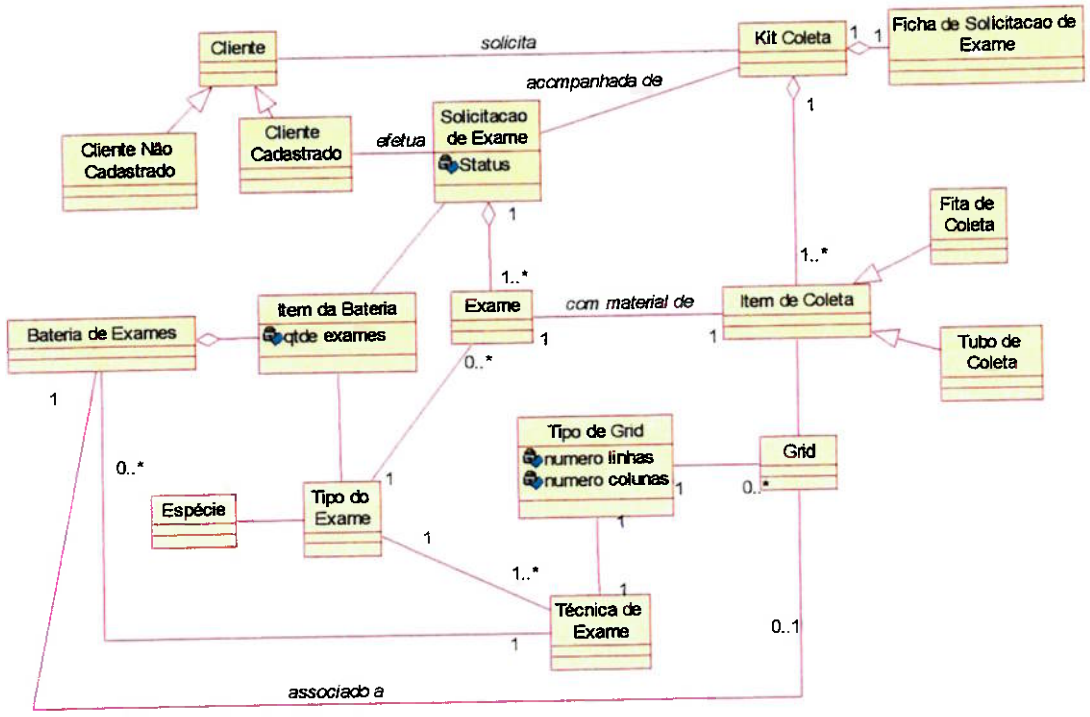


Figura 5-1 Modelo de Domínio Genetec

5.5 *Artefato: Visão*

5.5.1 Introdução

A Visão é o primeiro artefato elaborado quando se executa a disciplina Gerência de Requisitos. É um artefato fundamental, que norteia o projeto. No caso deste trabalho, a elaboração da Visão se baseou nos artefatos oriundos da Modelagem de Negócio.

A Visão é um artefato relacionado com o software a ser desenvolvido e fornece uma visão geral, em alto nível, sobre o problema e a solução proposta. É um artefato tanto para os gerentes quanto para os técnicos, embora para finalidades distintas.

Cabe ainda um esclarecimento: o RUP distingue os conceitos de *stakeholders*, clientes e usuários. Para o RUP, usuário é quem efetivamente manipula o sistema; cliente é quem tem algum objetivo de negócio que será atendido pelo sistema; *stakeholder* é um nome genérico para referência ao conjunto de interessados

(clientes, usuários, equipe de desenvolvimento, pessoal de áreas correlatas) no resultado do sistema.

A Visão para o aplicativo do estudo de caso é apresentada a seguir.

5.5.2 Objetivo

O objetivo do documento de Visão é coletar, analisar e definir necessidades globais e características do Sistema de Gerência de Exames. O documento é focado nas características necessitadas pelos *stakeholders* e usuários alvo, e nas razões pelas quais essas necessidades existem. Os detalhes de como o Sistema de Gerência de Exames atende a essas necessidades estão descritos nos casos de uso e especificações suplementares.

Escopo

O Documento Visão se refere ao Sistema de Gerência de Exames, que será desenvolvido para a Genetec, uma Companhia que atua no ramo laboratorial. A Genetec realiza e desenvolve exames diagnósticos e pesquisas na área da biologia molecular. O sistema permitirá que os clientes possam solicitar a realização de determinados exames (e solicitar o respectivo material para efetuar a coleta) e visualizar os resultados desses exames via web (sempre os clientes receberão também os resultados via correio). A funcionalidade principal do sistema é relativa à gerência do processo interno de cadastrar exames e organizar sua realização e o registro dos resultados.

5.5.3 Posicionamento

Definição do Problema

O problema de	Prazos excessivos e controles não eficientes para o ciclo que envolve coletar amostras, efetuar exames e divulgar exames laboratoriais em aves.
Afeta	Colaboradores envolvidos no processo, acionistas e clientes.
Provocando o impacto de	Carga de trabalho desnecessariamente alta para os colaboradores, faturamento e lucro abaixo do potencial, para a companhia, e potencial perda de clientes para concorrentes mais ágeis.
Uma solução adequada seria	A definição de um processo de negócio otimizado, suportado por um sistema para gerenciar todo o processo de coleta de material, e realização e divulgação dos resultados de exames de forma integrada, simplificando e otimizando os controles e permitindo a redução dos prazos de entrega de 15 dias para 3 dias.

Posicionamento do Produto

Para	A Genetec e seus potenciais clientes
Que	Precisem efetuar exames laboratoriais de sexagem de aves, DNA *Fingerprinting* (investigação de paternidade, maternidade), identificação de determinadas doenças.
O Sistema de Gerência de Exames	É um sistema para gerência do processo de negócio denominado Ciclo de Exame (ver Glossário)
Que	Gerencia eficientemente todo o processo, da solicitação e coleta, à divulgação dos resultados dos exames.
Ao contrário de	(não há conhecimento de produtos concorrentes. A referência é o processo atual)
Nosso produto	Permitirá a redução dos prazos de atendimento de 15 para 3 dias, liberando recursos humanos da companhia para atividades menos burocráticas.

5.5.4 Descrição de Usuários e *Stakeholders*

Demografia do Mercado

O produto será utilizado pelos clientes da Genetec para efetuar solicitações de exames e para visualizar os resultados dos exames elaborados, e pela própria Genetec, para gerenciar o processo de realização de exames.

Os clientes principais da Genetec são grandes empresas envolvidas com criação de aves e pequenos criadores de aves, para corte ou produção de ovos, bem como criadores de aves canoras, como curiós.

Há, portanto, uma ampla gama de clientes potenciais, com os mais variados perfis. Há clientes com amplos conhecimentos técnicos, que são capazes de identificar com precisão cada espécie de ave que possuem, e há também clientes leigos. Da mesma forma, nem todo o espectro de clientes tem acesso à Internet e, portanto, esse não pode ser o único meio de comunicação com os clientes.

O objetivo de todos os clientes, entretanto, é o mesmo: confiança nos resultados; prazos curtos de entrega de resultados e preços compatíveis com o mercado. É necessário também que a empresa possa utilizar uma linguagem acessível aos clientes leigos.

Sumário dos Stakeholders

Nome	Representa	Papel
Cliente pessoa física ou pequeno criador.	Pequenos clientes.	Solicitar exames (e material de coleta), envia material de coleta para a Genetec e recebe os resultados dos exames.
Cliente empresa de grande porte.	Clientes corporativos de maior porte.	Solicitar exames (e material de coleta), envia material de coleta para a Genetec e recebe os resultados dos exames. Demanda lotes maiores de exames.

		Em certos casos, são desenvolvidos exames específicos para as necessidades desses clientes.
Técnicos da Genetec	Funcionários técnicos, que realizam as análises laboratoriais e efetuam pesquisas.	Envolvidos no ciclo de exames. O aspecto principal desse envolvimento é que realizam os exames, registram e revisam os resultados dos exames.
Funcionário Administrativo	Funcionários que realizam tarefas administrativas, inclusive em relação ao processo de análises laboratoriais.	Responsáveis pela interação com clientes através de telefones. Envolvidos no ciclo de exames, principalmente registrando solicitações de exames, organizando as atividades do processo, enviando o material de coleta pelo correio e enviando os resultados finais por correio.
Gerente Técnico da Genetec	Gerente responsável pela atividade técnica / tecnológica da Genetec	Responsável pela criação de novos tipos de análises laboratoriais, inclusive a pedido de clientes. Define o processo do ciclo de exames. Responsável pela interação estratégica com clientes.
Gerente Administrativo da Genetec	Gerente responsável pela condução administrativa da	Define, em conjunto com o

	empresa	Gerente Técnico, o processo do ciclo de exames. Controla faturamento, cobrança, e gerência administrativa.

Sumário de Usuários

Nome	Descrição	Stakeholder
Cliente pessoa física ou pequeno criador.	Solicita o envio de material de coleta para a realização de exames via web; acessa, via web, o resultado dos exames e também recebe esses resultados pelo correio.	Representa a si próprio.
Cliente empresa de grande porte.	Solicita o envio de material de coleta para a realização de exames via web; acessa, via web, o resultado dos exames e também recebe esses resultados pelo correio.	Representa a si próprio.
Funcionário Administrativo	Cadastra clientes; registra pedido de envio de material de coleta; cadastra material de coleta recebido e registra o tipo de exames que devem ser efetuados e qual a espécie do animal; monta cronogramas de exames; obtém informações sobre resultados de exames para envio a clientes.	Representa a si próprio.
Técnico da Genetec	Obtém informações sobre os exames que devem ser realizados e qual sua prioridade; utiliza o sistema para montar as grades de exames e registrar os resultados dos exames.	Representa a si próprio.

Ambiente do Usuário

Há dois tipos básicos de ambientes de usuário:

- O Cliente acessa o sistema via Internet, para solicitar o envio de material de coleta de exames, para avaliar o estado de um exame em andamento ou para visualizar resultados. As solicitações de material de coleta também podem ser feitas por telefone ou e-mail. Os resultados dos exames também são enviados aos clientes por correio.
- Os funcionários da Genetec que utilizam o sistema para gerenciar o processo interno de exames acessam via Intranet ou via ambiente cliente-servidor convencional.

Necessidades Chave dos Stakeholders e Usuários

Necessidade	Prior.	Questões	Solução Atual	Solução Proposta
Atendimento ao cliente em até 3 dias.	Alta	Atualmente o cliente recebe os resultados em um tempo muito maior do que a concorrência é capaz de entregar.	O registro da solicitação, controle do processo interno e divulgação dos resultados é precariamente automatizado e não eficiente, gerando passos redundantes de conferência.	Sistema integrado para gerenciar o ciclo de exames, desde a solicitação, gerência da realização dos exames e validação de resultados, até a divulgação dos resultados e emissão do certificado.
Cadastro de clientes atualizado	Média	Atualmente não existe nenhum tipo de cadastro de clientes a ser utilizado no processo interno (a cada nova solicitação o cliente fornece todos os dados novamente), ou em campanhas	Não existe nenhum tipo de cadastro de clientes.	Cada contato de pedido de material de coleta servirá como entrada inicial de cadastro de cliente e será complementado e utilizado pelo sistema integrado de

		de marketing (malas diretas, etc) e fidelidade (redução de preços para grandes clientes)		cliclo de exames.
Apresentar resultados numa interface *web*.	Alta	Oferecer um diferencial em relação à concorrência.	Atualmente os clientes recebem os resultados dos exames solicitados somente através do correio.	Os resultados gerenciados pelo sistema integrado serão disponibilizados automaticament e via Internet com restrição de acesso somente ao cliente.

5.6 *Artefato: Modelo de Casos de Uso*

5.6.1 Introdução

O Modelo de Casos de Uso é um dos artefatos fundamentais definidos pelo RUP e o produto final da disciplina Gerência de Requisitos. Descreve atores (pessoas ou grupos que desempenham papéis perante o sistema) e casos de uso (descrevem os usos que os atores fazem do sistema). O Modelo de Casos de Uso é um artefato definido pela UML. Cada caso de uso identificado no modelo é posteriormente detalhado textualmente. No caso do presente trabalho:

- Elaborou-se uma versão do modelo de casos de uso, em princípio completo;
- Descreveu-se sucintamente cada caso de uso mais relevante;
- Descreveu-se, em detalhe, um caso de uso, que será posteriormente realizado em termos de diagramas de análise e projeto.

5.6.2 Diagrama de Casos de Uso

A figura 5.2 apresenta o diagrama que representa o Modelo de Casos de Uso do sistema em questão. Diversos casos de uso deste diagrama podem ser acionados por mais de um ator (geralmente o cliente, via internet; ou a secretária, internamente).

O ator Secretária é, na verdade, a pessoa que desempenha um papel administrativo no sistema. O nome Secretária foi mantido por ser o jargão vigente na própria Genetec.

Um caso de uso do diagrama está propositalmente duplicado (Gerenciar Informações do Cliente), apenas por razões de estética.

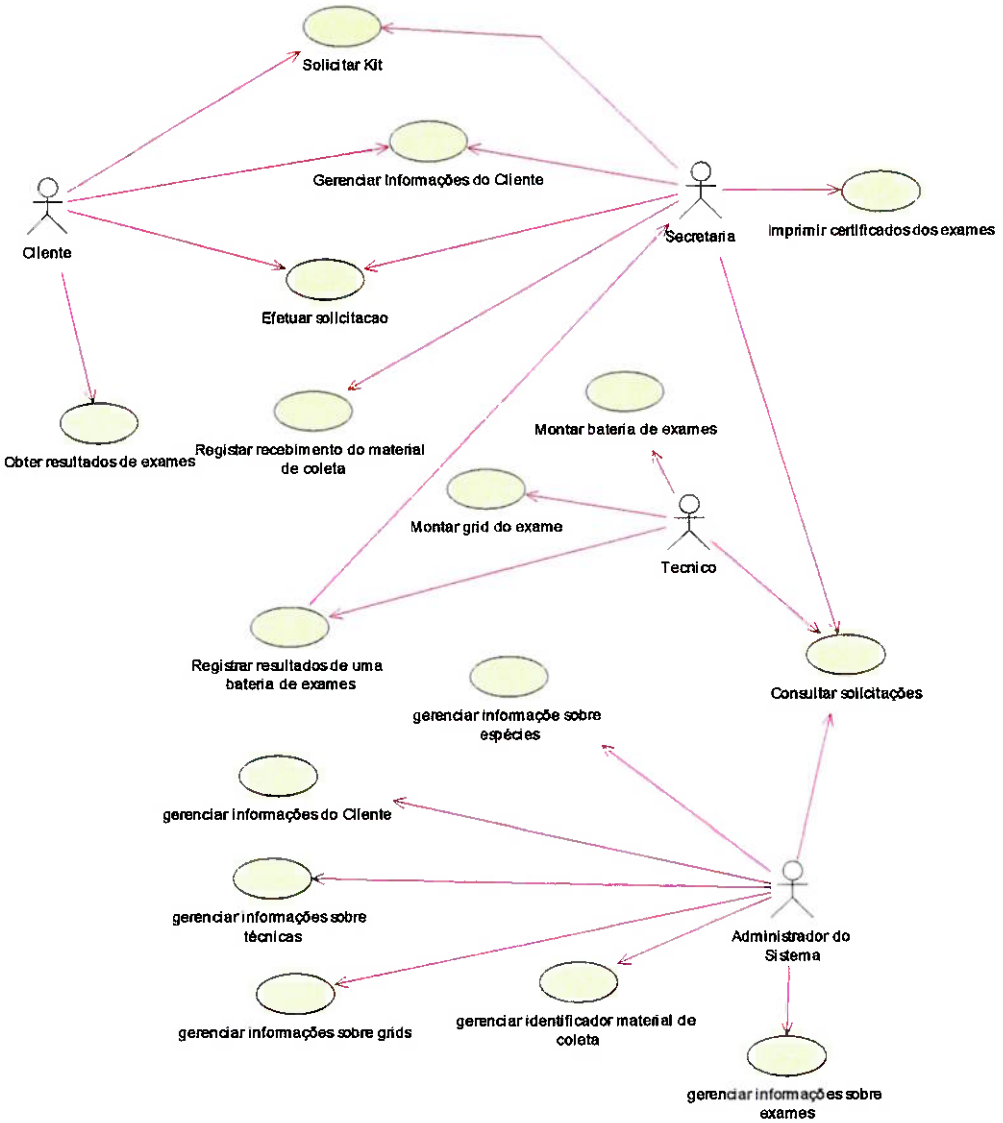


Figura 5-2 Modelo de Casos de Uso do Sistema Genetec

5.7 *Descrição Sucinta dos Casos de Uso do Sistema*

5.7.1 Introdução

A seguir, são apresentadas as descrições sucintas da maioria dos casos de uso que compõem o diagrama da figura 5.2. Apenas casos de uso cujo nome começa com "Gerenciar Informações de..." não foram descritos por se tratar de casos de uso de infra-estutura. O termo gerenciar informações foi utilizado para representar manipulações de dados cadastrais (ler, inserir, alterar, excluir informações).

5.7.2 Solicitar *Kit* de Coleta

Este caso de uso descreve a solicitação do *kit* de material de coleta por um cliente ou potencial cliente, caso o cliente acesse essa funcionalidade via *web*. Outro ator que pode disparar este caso de uso é a secretária, caso esta receba a solicitação por outro canal que não a *web*, e opte por registrar a solicitação do *kit* no sistema.

5.7.3 Efetuar solicitação

Este caso de uso descreve a solicitação do exame via web pelo cliente, após receber o *kit* com material de coleta e efetuar a coleta.

5.7.4 Registrar recebimento do material de coleta

Este caso de uso descreve o registro no sistema, pela Secretária, das informações referentes ao material de coleta de uma solicitação, recebido pelo correio. Esta funcionalidade permite que a secretária: pesquise a solicitação referente ao material de coleta, identifique os itens do material de coleta que constam na Solicitação e identifique também as divergências entre os itens do material de coleta e itens da solicitação (estes passarão a compor uma solicitação Incompleta).

5.7.5 Obter resultados de exames

Este caso de uso permite que o Cliente obtenha:

- Status de solicitações (data prevista para disponibilização dos exames de uma solicitação em andamento, ou se a solicitação já foi concluída);

- Resultados de exames (de solicitações não totalmente atendidas);
- Resultados históricos (dados de solicitações já concluídas).

5.7.6 Consultar solicitações

Este caso de uso permite pesquisar as solicitações através de diversos critérios.

5.7.7 Montar bateria de exames

Este caso de uso é responsável pela montagem de baterias de exames, em função de critérios definidos pelo técnico.

O técnico define determinados critérios de pesquisa (cliente, data de chegada, tipo de exame, quantidade de exames). O sistema exibe uma lista com os exames que atendem ao critério de pesquisa. O técnico seleciona todos ou alguns dos exames dessa lista e define que eles compõem uma bateria.

5.7.8 Montar grid do exame

Este caso de uso descreve a interação do técnico com o sistema para selecionar uma determinada bateria já definida e, a partir dela, montar o *grid* com as amostras que comporão um certo exame (há casos em que um exame contempla 96 amostras).

Durante a montagem do *grid*, o sistema alerta para erros de técnica (se o técnico, por engano, tentar incluir um item num *grid* que é referente a outra técnica).

5.7.9 Registrar resultados de uma bateria de exames

O técnico utiliza este caso de uso para registrar os resultados de uma bateria de exames.

Quando os resultados de uma bateria de exames são registrados, o(s) cliente(s) envolvido(s) é (são) informados, via e-mail, e os resultados ficam disponíveis via Internet.

Quando o técnico termina de registrar os resultados de exames de uma certa bateria, o sistema verifica cada solicitação envolvida. Se todos os exames de uma certa solicitação, que têm o mesmo prazo de entrega já tiverem resultado, a secretária é avisada para imprimir e enviar os resultados desses exames.

5.7.10 Imprimir certificados dos exames

Este caso de uso permite que a secretária selecione os exames da solicitação que já tenham resultado e que o sistema imprima os certificados correspondentes.

5.7.11 Outros Casos de Uso

Os demais casos de uso gerenciam informações cadastrais. Cada caso de uso descreve a interação com o sistema para ler/incluir/alterar/excluir/ determinado conjunto de informações cadastrais. São eles:

- Gerenciar informações sobre espécies;
- Gerenciar informações de cliente;
- Gerenciar informações sobre técnicas;
- Gerenciar informações sobre *grids*;
- Gerenciar identificador material de coleta;
- Gerenciar informações sobre exames.

5.8 *Descrição Detalhada do Caso de Uso Efetuar Solicitação*

5.8.1 Introdução

A seguir, é apresentada a descrição textual detalhada do caso de uso Efetuar Solicitação. Esse caso de uso foi selecionado para ilustrar a especialização descrita no capítulo 3 e será realizado, em termos de diagramas de classes e diagramas de seqüência, tanto do ponto de vista de análise quanto de projeto. Esta abordagem foi adotada, pois as outras realizações de casos de uso não acrescentariam novos elementos relevantes de análise e projeto. Em um projeto real, todos os casos de uso seriam necessário detalhados textualmente.

Em um caso de uso, a descrição textual é composta por dois itens principais: um fluxo básico de eventos e um conjunto de fluxos alternativos.

O fluxo básico descreve uma seqüência típica de eventos que chega a um final bem sucedido, no sentido de que a execução do caso de uso traz algum benefício (diz-se

também agrega valor) ao ator que disparou o caso de uso. Quando existirem mais de uma situação desse tipo, uma única situação será selecionada como fluxo básico.

Todas as situações diferentes do fluxo básico são representadas como variações deste e constituem os fluxos alternativos. Haverá, para um caso de uso, tantos fluxos alternativos quantos forem necessários.

Optou-se pelo caso de uso Efetuar Solicitação por representar uma funcionalidade central da aplicação: a solicitação de um conjunto de exames pelo cliente, via internet. O fato de o caso de uso referenciar diversos elementos fundamentais do domínio (a solicitação de exame, cada item da solicitação, os itens de coleta) é outro fator para justificar a escolha deste caso de uso. Por fim, o caso de uso é relativamente simples. Muito mais simples, por exemplo, do que a lógica para montar uma bateria ou *grid.*

5.8.2 Fluxo Básico de Eventos

A. Cliente seleciona opção para efetuar solicitação

O caso de uso começa quando o Cliente seleciona a opção "Efetuar Solicitação de Exames". O sistema exibe uma página para que Clientes previamente cadastrados se identifiquem.

B. Cliente se identifica

O cliente informa sua identificação e senha. O sistema valida a identificação do cliente, e exibe a página para que o cliente informe os dados da solicitação de exame.

C. Cliente informa dados da solicitação

Para cada exame específico, o Cliente informa: a espécie do animal (a espécie do animal pode ser selecionada de uma lista fornecida pelo sistema, mas também pode ser fornecido como um texto livre); o número de identificação do animal (espécie de R.G. do animal); o número Unigen do rótulo do item do *kit* no qual foi armazenado o material de coleta desse animal específico; o tipo do exame a ser realizado (selecionado de uma lista). Para exames que podem ter mais de um prazo de entrega (36h ou 4dias, por exemplo), o Cliente será solicitado a informar qual o prazo de entrega que deseja.

Após informar todos os dados a solicitação, o Cliente a submete ao sistema.

D. Concluir Solicitação

O sistema exibe uma página informando o conteúdo da solicitação, os dados do cliente, e o custo total da solicitação, considerando as políticas da Unigen referentes à definição de preço. O cliente confirma a solicitação. O caso de uso é encerrado.

5.8.3 Fluxos Alternativos

Cliente Não Cadastrado

No passo A do fluxo básico, caso não tenha se cadastrado previamente no sistema, o Cliente seleciona a opção "Efetuar Cadastro". O caso de uso **Gerenciar Informações de Cliente** é incluído neste ponto. O caso de uso continua no item C do fluxo básico.

Identificação Inválida

No passo B do fluxo básico, caso a identificação ou a senha do Cliente não seja válida, o sistema exibe uma mensagem informando que a identificação é inválida e solicita que o cliente forneça a identificação correta. O caso de uso continua no item B do fluxo básico.

Solicitação interrompida

A qualquer momento, o Cliente pode interromper o caso de uso, simplesmente mudando de página ou fechando o *browser*. O caso de uso é encerrado.

Cliente salva solicitação não concluída

No passo C do fluxo básico, o Cliente pode salvar a solicitação quando ainda não terminou de preenchê-la (para continuar o preenchimento da solicitação posteriormente). A solicitação é salva com status Não Concluída. O caso de uso é encerrado.

Obs.: só é possível salvar uma solicitação que tenha pelo menos um pedido de exame já registrado.

Cliente continua a preencher uma solicitação salva anteriormente

No passo B do fluxo básico, quando o Cliente se identificar, o sistema pesquisa se há alguma solicitação (não concluída) previamente salva desse cliente. Se houver, o sistema pergunta ao Cliente se deseja continuar a preencher uma nova previamente salva (se houver mais de uma, o cliente deve selecionar a que deseja), ou iniciar uma nova solicitação. Caso o Cliente opte por uma solicitação já existente, os seus dados são recuperados. O caso de uso continua a partir do passo C do fluxo básico.

Cliente edita dados da solicitação

No passo C do fluxo básico, o Cliente pode alterar dados de um item da solicitação ou excluir itens da solicitação, desde que a solicitação esteja com o status Pendente. O caso de uso continua no passo C do fluxo básico.

Dados do cliente incorretos

No passo D do fluxo básico, caso os dados do Cliente não sejam corretos, o Cliente seleciona a opção Alterar Dados Cadastrais. O caso de uso **Gerenciar Informações de Cliente** é incluído neste ponto. O caso de uso continua no item D do fluxo básico.

Cliente não confirma solicitação

No passo D do fluxo básico, caso o cliente não confirme a solicitação, o Sistema pede que o Cliente informe se realmente deseja cancelar a solicitação ou se deseja confirmá-la. O cliente efetua a opção desejada, e o caso de uso é encerrado, com a solicitação confirmada ou cancelada.

5.9 *Arquitetura de Software – Vista Lógica*

5.9.1 Introdução

A forma como o RUP aborda a definição de uma arquitetura de software foi descrita no capítulo 3. Quando da especialização para J2EE, no capítulo 4, destacou-se a atividade Análise da Arquitetura, que contém diretrizes para a definição da vista lógica da arquitetura de software da aplicação (definição da organização dos subsistemas de alto nível). Nesse ponto do capítulo 4, discutiram-se alguns *patterns* de arquitetura oriundos de [BUSCHMANN 1996] e definiu-se um modelo específico de vista lógica da arquitetura de software para a plataforma J2EE, com base nos patterns Layers e MVC.

5.9.2 A vista lógica da Arquitetura de Software para o estudo de caso

A figura 5.3 contempla uma proposta de vista lógica da arquitetura para o estudo de caso, respeitando as definições elaboradas no capítulo 4: definição de uma arquitetura em camadas e separação dos elementos de model-view-control.

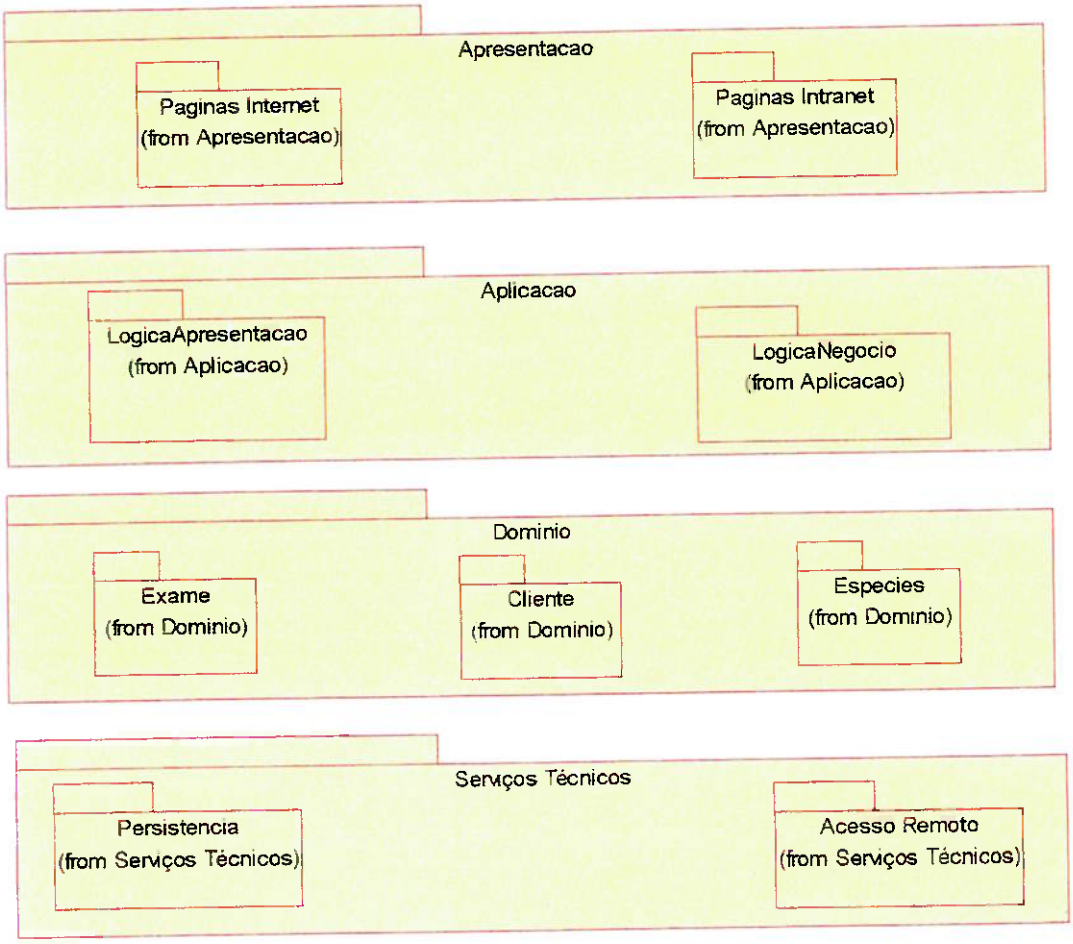


Figura 5-3 Vista lógica da arquitetura de software do

É proposta uma vista lógica da arquitetura em quatro camadas. No capítulo 4, a arquitetura proposta contempla cinco camadas, mas foi comentado que o número de camadas não é necessariamente fixo. O essencial é que a arquitetura será elaborada em camadas, e que os elementos de model, view e control serão definidos em camadas distintas. A quantidade exata de camadas é função das características de cada aplicação específica.

A arquitetura proposta no capítulo 4 prevê, em princípio, a existência de camadas distintas de aplicação, domínio, e infra-estrutura de negócio. À medida que se desce na estrutura de camadas, elas se tornam mais independentes da aplicação. Diferenciar a camada de aplicação da camada de domínio, e ambas da camada de infra-estrutura de negócio, contribui para aumentar o grau de reuso dos componentes da aplicação. Os componentes da camada de aplicação tendem a ser pouco reusáveis, pois contém a inteligência de controle específica de uma aplicação. Os componentes das camadas de domínio e, mais ainda, da camada de infra-estrutura de negócio são muito mais genéricos, e serão provavelmente utilizados em muitas aplicações.

Essas considerações na definição da vista lógica da arquitetura no estudo de caso foram respeitadas. Entretanto, a camada de Infra-Estrutura de Negócio foi desconsiderada pois, até onde o desenvolvimento do estudo de caso evoluiu, não foram identificados os elementos que se encaixassem nessa camada esta camada conteria os elementos de negócio muito gerais, independentes de qualquer domínio. Caso o desenvolvimento da aplicação tivesse continuidade e fossem identificados elementos dessa natureza, a camada poderia ser reincorporada à arquitetura. Vale lembrar a abordagem iterativa do RUP, em que a arquitetura de software começa relativamente mais simples, e vai sendo sofisticada à medida que as iterações avançam, e eventualmente se identificam pontos que incrementam a complexidade da aplicação.

Conforme citado no capítulo 4, cada camada é representada por um pacote, que é composta também por pacotes, de menor granularidade. Um pacote é um elemento estrutural da UML, que serve para conter outros elementos de modelagem (neste escopo, outros pacotes ou classes). Não se consideram ainda os subsistemas ou componentes, pois é um plano conceitual. Na abordagem da vista de implementação da arquitetura, a organização seria definida em termos de subsistemas e componentes.

A camada de apresentação, segundo o *pattern* MVC, é a View, que no sistema contempla dois pacotes: um de páginas Internet e outro de páginas Intranet. Essa organização se baseia no fato de que é possível que a aplicação contemple padrões visuais distintos entre Intranet e Internet. Além disso, também é possível que

determinadas funcionalidades sejam abordadas de forma distinta nos dois ambientes. Como não se detalhou a interface visual da aplicação, no estudo de caso, esse ponto não foi explorado e, portanto, a abordagem é tentativa, e seria reavaliada no momento em que se definisse a apresentação visual da aplicação.

A camada de aplicação, segundo o *pattern* MVC, seria o Control que também foi organizada em dois pacotes: um que contempla os elementos de controle da lógica de apresentação; outro que contempla os elementos de controle da lógica de negócio. O objetivo é distinguir claramente os dois tipos de elementos, que terão funcionamentos distintos, embora fortemente inter-relacionados (por isso fazem parte da mesma camada, já que a idéia é restringir, tanto quanto razoável, a comunicação entre camadas). Ao se fazer essa distinção, um princípio básico de engenharia de software – a separação de áreas de interesse – , é respeitado.

No caso da plataforma J2EE, o controle da lógica de apresentação é implementado por Servlets, alocados na camada web. O controle da lógica de negócio é implementado por EJBs do tipo session beans, fisicamente alocados no container EJB. Dessa forma, a separação no plano lógico é naturalmente mapeada numa separação no plano físico.

A camada de domínio, segundo o *pattern* MVC, é o Model e organiza os elementos do domínio em três pacotes: um referente ao Cliente; outro referente aos exames; um terceiro referente a espécies de animais objeto dos exames laboratoriais. Essa organização foi elaborada a partir de uma análise do modelo de domínio, procurando identificar os seus elementos fundamentais e organizando-os da forma conceitualmente mais natural e que mais estimule reuso de pacotes individuais. Considera-se que os elementos do pacote Cliente certamente têm boa possibilidade de ser reutilizados em outras aplicações, bem como elementos dos pacotes de exames.

Considerou-se a possibilidade de dividir o pacote Exame, da camada de domínio, em dois: um englobando os elementos mais associados à solicitação de exames (a própria solicitação, cada item da solicitação) e um outro englobando os elementos mais relacionados à realização do exame (*grid*, bateria de exames). Optou-se, no entanto, manter a organização inicial mais simples, conforme descrita no capítulo

anterior. À medida que se aprofunde no projeto da aplicação, essa organização poderia ser revista, caso se considere que a organização mais detalhada seja mais vantajosa (por exemplo, o pacote Exame com excesso de elementos; ou porque faz sentido, do ponto de vista de reuso, a organização mais detalhada).

A camada de serviços técnicos aborda os componentes cujo objetivo é voltado à prestação de serviços de infra-estrutura. A definição dessa camada desacopla os elementos das camadas superiores da implementação dos serviços de infra-estrutura, aumentando a robustez da aplicação.

5.10 *Realização do Caso de Uso Efetuar Solicitação – Análise*

Uma vez que os requisitos estejam definidos e a vista lógica da arquitetura de software (organização do sistema) também tenha uma primeira versão elaborada, chega o momento de detalhar a aplicação, inicialmente do ponto de vista de análise, e a seguir, do ponto de vista de projeto. Pode-se, neste ponto, questionar a necessidade da análise; com os casos de uso definidos, pode-se dar a impressão de que se poderia iniciar o projeto, para poupar o tempo. No entanto, a construção do modelo de projeto é favorecida pela existência do modelo de análise, pois as características de implementação são a ele agregadas, mantendo a coerência com o Modelo de Casos de Uso. Portanto, a proposta básica do RUP, de efetuar atividades de análise antecedendo as de projeto, foi respeitada na especialização feita no capítulo 4.

Na análise, foram construídos um diagrama de classes e um diagrama de seqüência. A ordem em que esses diagramas são construídos não é pré-determinada, pois o diagrama de classes ajuda a construir o diagrama de seqüência, e vice-versa.

Para isso, utilizou-se, como referência, o modelo de domínio previamente elaborado. Desta forma, sempre que possível, identificaram-se as classes de análise a partir das classes conceituais do modelo de domínio, aproximando o modelo de análise do domínio do problema e, desta forma, reduzindo o *gap* semântico entre o mundo real e o modelo de software.

5.10.1 Diagrama de Classes

A figura 5.4 apresenta o diagrama de classes elaborado, referente ao caso de uso Efetuar Solicitação. As classes de análise definidas receberam os estereótipos <<boundary>>, <<control>> e <<entity>>, conforme prevê o RUP. Essa definição será fundamental quando for derivado o modelo de projeto, a partir do modelo de análise. No diagrama apresentado na figura 5.4, os estereótipos são representados através de um adorno gráfico no canto superior direito da classe.

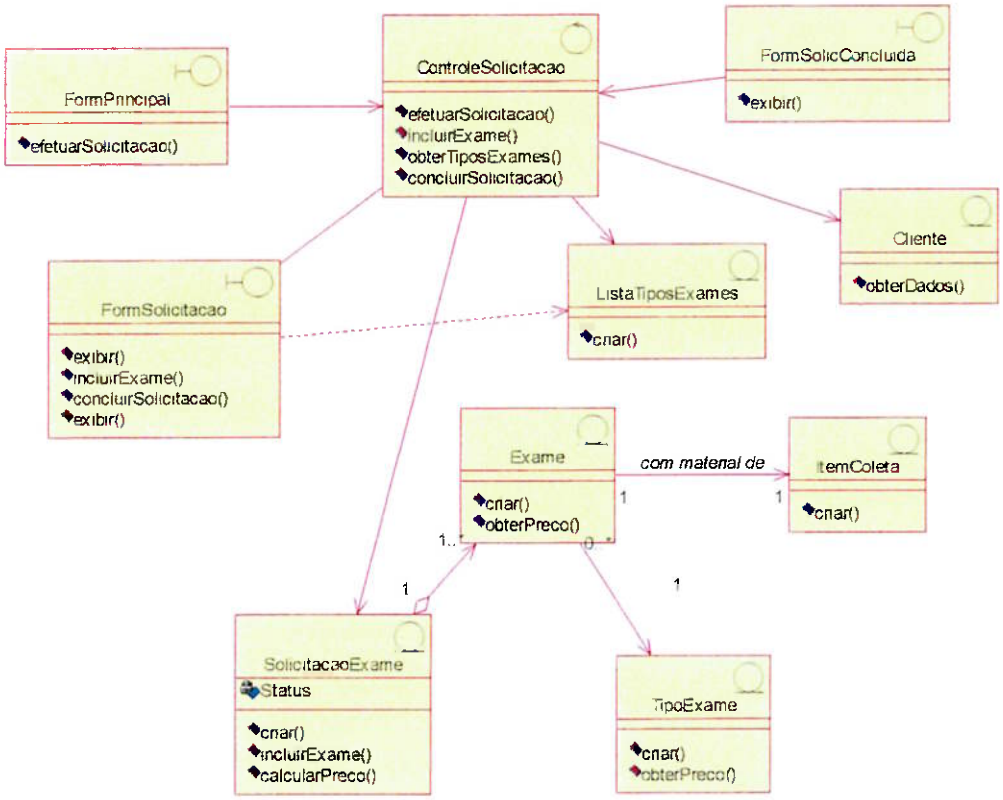


Figura 5-4 Diagrama de Classes do Sistema GENETEC

Neste diagrama, há três classes de fronteira (boundary), aquelas utilizadas pelo cliente para se comunicar com a aplicação. Essas três classes estão relacionadas à classe de controle denominada ControleSolicitacao. Esta classe foi definida considerando o pattern GRASP Controller (citado no capítulo 4), que prevê que cada caso de uso terá, em princípio, uma classe, cuja responsabilidade é coordenar a

atuação das demais classes, para prover a realização da funcionalidade descrita pelo caso de uso.

Além de se relacionar com as classes fronteira, ControleSolicitacao também se relaciona com duas classes entidade: SolicitacaoExame e ListaTiposExame.

ListaTiposExame é, como o nome indica, uma lista, na qual cada elemento representa as informações sobre um determinado tipo de exame. Essa lista é exibida ao cliente sempre que ele inclui mais um exame na sua solicitação; por isso há uma relação de dependência entre a classe FormSolicitacao e ListaTiposExame. ControleSolicitacao deve disponibilizar a FormSolicitacao a lista de tipos de exames (esta informação estará contida numa instância de ListaTiposExame).

SolicitacaoExame é a classe entidade fundamental do caso de uso. É um agregado de exames (veja diagrama). Cada instância de Exame, por sua vez, está relacionada a um item de coleta e a um tipo de exame, ambos informados pelo usuário.

5.10.2 Diagrama de Seqüência

Apresenta-se, nesta seção, o diagrama de seqüência referente ao fluxo básico do caso de uso Efetuar Solicitação, que comporta três interações do cliente com o sistema:

- O cliente informa que deseja efetuar uma solicitação;
- Para cada exame que compõe a solicitação, o cliente fornece os dados do exame e solicita que este seja incluído na solicitação;
- O cliente informa que a solicitação está concluída.

Todas as interações poderiam, em princípio, ser representadas através de um único diagrama; no entanto, devido à restrição de espaço nas páginas deste texto, cada interação do cliente com o sistema foi representada em um diagrama a parte. Os três diagramas podem ser vistos a seguir, nas figuras . Reitera-se que apenas o fluxo básico do caso de uso é descrito pelo diagrama, pois a inclusão dos diagramas ou das mensagens no diagrama básico, referentes aos fluxos alternativos, iria trazer uma complexidade desnecessária ao estudo de caso.

Vale lembrar que os diagramas de seqüência são diagramas de instâncias, não diagramas de classes. Sendo assim, os elementos representados são os objetos que,

no diagrama são caracterizados por apresentaremo nome da classe sublinhado e precedido de um dois pontos (:).

A linha pontilhada é denominada linha de vida do objeto. Objetos que são instanciados ao longo da execução da interação representada no diagrama recebem uma mensagem denominada criar diretamente para o objeto. Quando o objeto já existia antes de a interação começar, ele aparece no topo do diagrama, e as mensagens são direcionadas para a linha de vida.

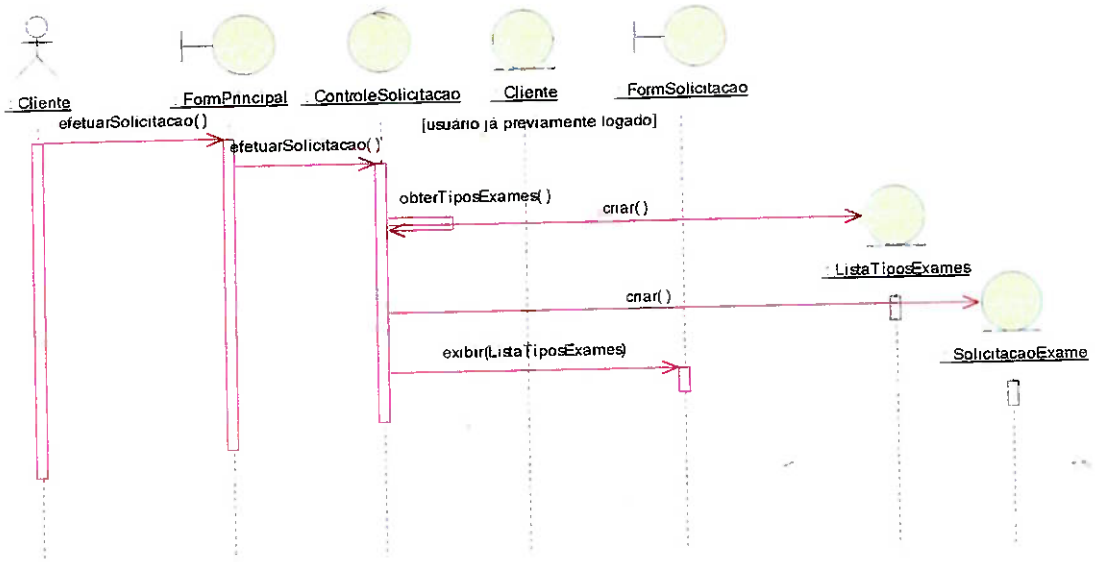


Figura 5-5 Diagrama de seqüência (análise) parte 1/3

O diagrama 1.5 descreve as ações que ocorrem no sistema quando o usuário seleciona a opção efetuar solicitação. O objeto que representa a página principal recebe o estímulo enviado pelo usuário e, como conseqüência, aciona o método efetuarSolicitacao do objeto de controle ControleSolicitacao. As ações executadas por este método são:

- Executar um método interno denominado obterTiposExames. Este método, por sua vez, instancia um objeto da classe ListaTiposExames. Fica subentendido (mas poderia ter sido explicitado em uma nota no diagrama) que, quando instanciado, o objeto ListaTiposExame automaticamente se valoriza com a lista dos tipos de exames em vigor;

- Instancia uma SolicitacaoExame, que será utilizada logo a seguir (no próximo diagrama);
- Aciona o objeto de fronteira FormSolicitacao, que deve ser exibido ao usuário para que ele informe os dados da solicitação de exame que deseja criar.O objeto FormSolicitacao recebe como parâmetro a instância de ListaTiposExames previamente instanciada.

A premissa do diagrama é que o usuário se identificou ao sistema previamente. É mais razoável descrever o procedimento de identificação do usuário em um diagrama à parte, já que esse procedimento precede cada interação do usuário com o sistema.

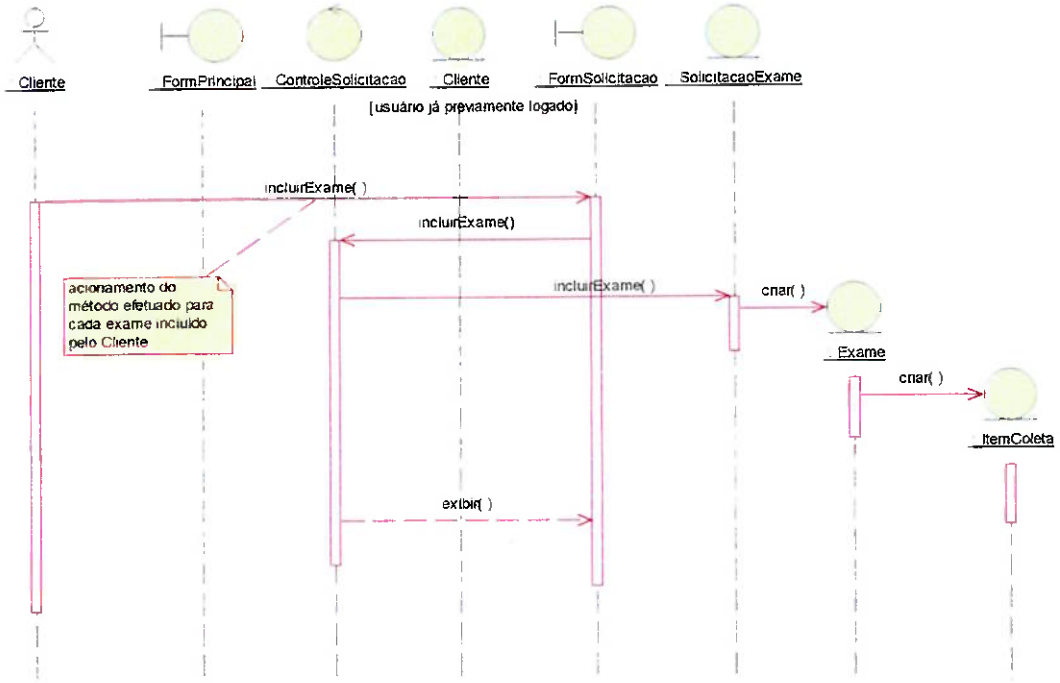


Figura 5-6 Diagrama de seqüência (análise) parte 2/3

A execução do fluxo básico do caso de uso continua no diagrama da figura 1.6 .

A funcionalidade descrita pela segunda parte do diagrama de seqüência, apresentada na figura 1.6, é executada diversas vezes, para cada exame que o cliente deseje incluir na sua solicitação.

O cliente preenche as informações referentes ao exame e seleciona a opção incluir exame. O objeto de fronteira que recebe o estímulo aciona, por sua vez, o objeto de controle (pattern Controller). O objeto de controle, então invoca o método incluirExame do objeto da classe SolicitacaoExame (instanciado na primeira parte do diagrama). Este objeto tem a inteligência para efetuar a inclusão do exame na solicitação. Embora seja um objeto entidade, ele coordena os detalhes da inclusão do exame na solicitação. Atribuir essa responsabilidade a SolicitacaoExame é perfeitamente razoável, já que classes entidade podem apresentar comportamento complexo (evidentemente, comportamento pertinente à abstração que a classe representa).

Assim, SolicitacaoExame instancia Exame; este procedimento está de acordo com o pattern GRASP Creator, onde uma classe que agrega ou inclui outra deve ser responsável pela instanciação desta.

Exame, uma vez instanciado, instancia ItemColeta. Neste ponto, poder-se-ia ter atribuído, a SolicitacaoExame, a responsabilidade de instanciar ItemColeta, mas a solução adotada respeita mais fielmente o pattern GRASP Baixo Acoplamento, já que Exame estaria, de qualquer forma, associado ao ItemColeta correspondente e, pela solução adotada, não há acoplamento entre SolicitacaoExame e ItemColeta.

O diagrama não esclarece, mas as informações fornecidas pelo usuário (inclusive o tipo do exame a ser realizado) são repassadas aos objetos que delas necessitem. Assim, Exame, por exemplo, recebe a informação de qual tipo de exame o cliente deseja, e armazena esta informação.

Após SolicitacaoExame efetuar as ações para inclusão de mais um exame na solicitação, o controle volta para ControleSolicitação, que exibe um novo FormSolicitacao, agora já apresentando o novo exame incluído.

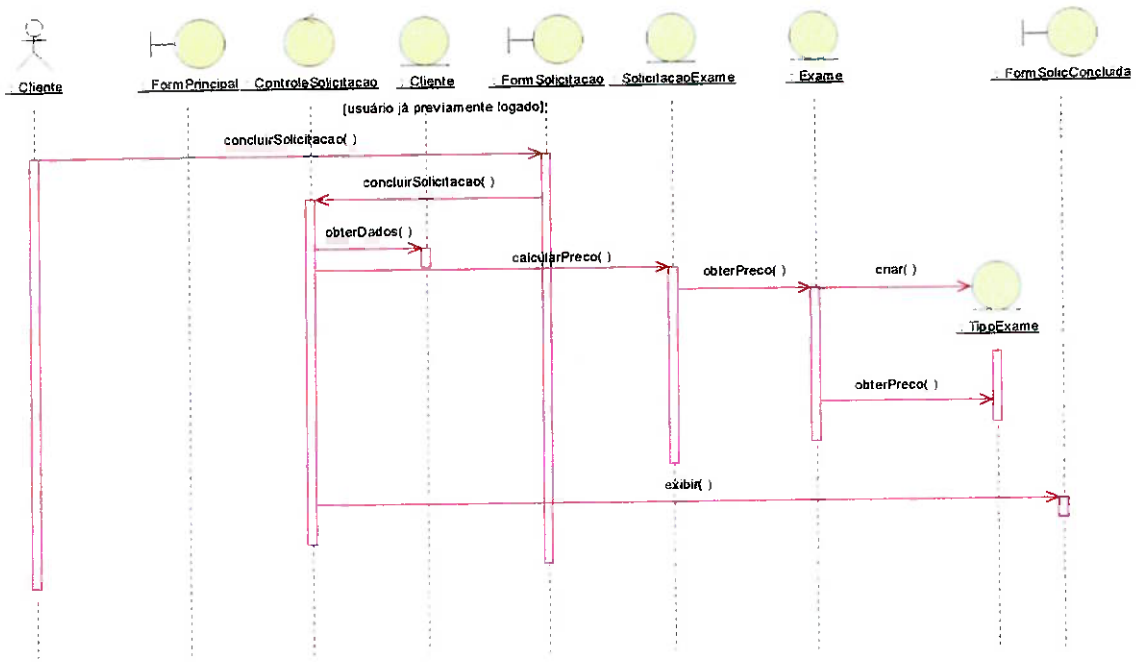


Figura 5-7 Diagrama de seqüência (análise) parte 3/3

O diagrama na figura 1.7 representa a etapa final da criação de uma nova solicitação de exame.

O procedimento que merece ser destacado, neste diagrama, é o cálculo do preço da solicitação.

Quando o método concluirSolicitacao de ControleSolicitacao é acionado, ele dispara, por sua vez, o método calcularPreco de SolicitacaoExame. Novamente, a responsabilidade por coordenar a execução da ação não cabe ao objeto de controle do caso de uso, mas ao objeto entidade. Isso está de acordo com o pattern GRASP Information Expert, que define que a classe que deve receber determinada responsabilidade é aquela que possui as informações necessárias para o cumprimento da responsabilidade. No caso, quem conhece cada exame incluído na solicitação é SolicitacaoExame, por isso cabe a ela a responsabilidade por calcular o preço do exame.

Mas essa responsabilidade será distribuída a várias classes (que funcionam como experts parciais), porque quem tem acesso ao tipo do exame é o próprio objeto Exame. Por isso, para calcular o preço total da Solicitação, SolicitacaoExame aciona

cada objeto Exame que o compõe e solicita que este informe o preço do exame correspondente (portanto, essa ação é executada múltiplas vezes). Para informar o preço do exame, o objeto Exame instancia (ou acessa, caso já tenha sido instanciado) o objeto TipoExame correspondente, e pede o preço do exame. SolicitacaoExame totaliza o preço total da solicitação e devolve essa informação ao objeto de controle, que aciona o objeto de fronteira para exibir a solicitação concluída, com o preço total. Isso encerra a realização do fluxo básico do caso de uso Efetuar Exame.

Eventualmente, certos detalhes foram omitidos nos diagramas, que não pretendiam ser exaustivos. O objetivo é apresentar, em um grau razoável de detalhe, a dinâmica dos objetos que realizam o caso de uso, respeitando os preceitos definidos pelo RUP e considerando princípios definidos pelos patterns mais básicos. A seguir, esses diagramas são utilizados como referência para a definição dos diagramas de projeto correspondentes.

5.11 *Realização do Caso de Uso Efetuar Solicitação – projeto*

5.11.1 Diagrama de Classes

O padrão adotado para identificar as classes no diagrama abaixo foi o seguinte:

- Cada classe contém um estereótipo que define o elemento da plataforma J2EE que será utilizado para implementar a classe:
    - <<input form>> ou <<form>>, identifica a classe como sendo um formulário HTML;
    - <<Server Page>> identifica a classe como sendo uma página JSP;
    - <<Http_Servlet>> identifica a classe como sendo um Servlet;
    - <<entity bean>> ou <<session bean>> identificam o tipo de EJB que a classe representa;
    - <<javabean>> é um termo comumente usando para referência a classes Java convencionais (não componentes EJB). A única restrição a um javabean é que ele implementa métodos de acesso (set e get) para cada atributo que contém. Os patterns J2EE se referem a essas classes como

helpers, porque elas auxiliam outras classes a executar suas tarefas, geralmente armazenando informações que essas outras classes acessam.

- O nome da classe identifica o pattern J2EE, ou elemento do pattern J2EE, que gerou sua definição, quando for pertinente:
    - A classe RequisicoesFrontController representa um FrontController, do pattern J2EE do mesmo nome;
    - A classe SolicitacoesDispatcher, representa um Dispatcher, do pattern J2EE FrontController;
    - A classe SolcicitacoesFacade, representa um SessionFacade, do pattern J2EE SessionFacade;
    - ListaTiposExamesHelper é uma classe Helper, do pattern FrontController.

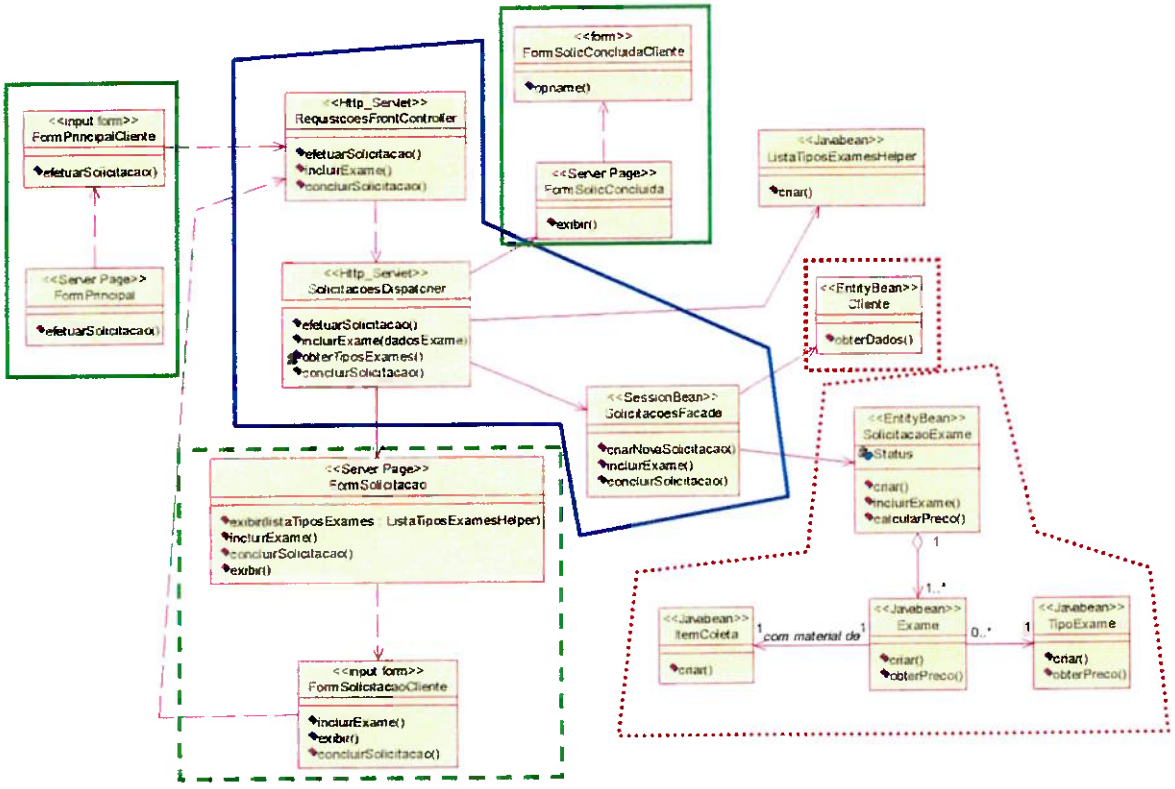


Figura 5-8 Diagrama de Classes de Projeto - Caso de Uso Efetuar Solicitação

O diagrama de classes na figura 1.8 foi elaborado a partir do diagrama de análise correspondente, utilizando as definições da especialização do RUP para J2EE elaboradas quando se abordou a atividade Identificar Elementos de Projeto (capítulo 4).

As classes envolvidas em retângulos verdes (tracejados) eram classes de fronteira, e foram convertidos no par client page-server page.

As classes envolvidas pelo polígono azul (linhas retas) constituíam, na análise, uma classe de controle (ControleSolicitacao). Usando o padrão de conversão de análise para projeto definido no capítulo 4, uma única classe de controle de análise é convertida em três classes de projeto: um FrontController (uma única classe para toda a aplicação); um par de classes SolicitacoesDispatcher e SolicitacoesFacade para cada caso de uso do aplicativo (embora seja razoável utilizar um único par Dispatcher-Facade para mais de um caso de uso, desde que as funções dos casos de uso sejam conceitualmente correlacionadas). O FrontController recebe todas as solicitações externas, trata questões ligadas a segurança, por exemplo, e repassa a solicitação para o Dispatcher adequado. Este é um objeto de controle especializado em atender um conjunto específico de solicitações (normalmente, as solicitações oriundas de um caso de uso). Ele sabe invocar o objeto de negócio adequado (o SessionFaçade, session bean da camada EJB da aplicação), receber as informações devolvidas pelo SessionFaçade, e gerenciar a apresentação da resposta ao usuário (a resposta em si é gerada por JSPs – server pages). O SessionFaçade coordena o atendimento da solicitação, acionando os objetos (componentes EJB) que executarão a lógica de negócio necessária para o atendimento à solicitação.

Cada classe ou conjunto de classes envolvidas pelo polígono vermelho (pontilhado) constitui um Entity Bean, e eram (individualmente), classes entidades na análise. Aqui há uma peculiaridade. A classe de análise Cliente foi derivada no entity bean Cliente. Já as quatro classes de análise SolicitacaoExame, ItemColeta, Exame e TipoExame vão compor um único entity bean SolicitacaoExame (isso não está representado no diagrama UML, apenas pelo retângulo vermelho que utilizamos para destacar os entity beans). Isso por conta de uma boa prática de engenharia de software, apresentada no capítulo 4, onde entity beans não devem se relacionar uns

com os outros. Em vez disso, as classes entidade que se relacionam devem ser convertidas num único componente entity bean (pois vale lembrar que um entity bean é um componente, não uma classe individual, e pode conter várias classes entidade).

Feitas essas definições fica relativamente simples definir os diagramas de seqüência de projeto, que é a última atividade de projeto a ser realizada.

5.11.2 Diagrama de Seqüência

A seguir é exibida a colaboração entre objetos de projeto que realizam o fluxo básico do caso de uso Efetuar Solicitação. Assim, como na análise, o diagrama de seqüência é expressa em três diagramas distintos, apenas por razões estéticas.

No item anterior, as classes de projeto foram definidas, levando em consideração as diretrizes elaboradas no capítulo 4, de conversão de classes de análise em classes de projeto. Feita essa definição, a conversão dos diagramas de seqüência de análise em diagramas de projeto fica razoavelmente simples, mas há agora um novo conjunto de fatores a ser levado em conta. Em primeiro lugar, a análise desconsidera determinadas questões, como persistência, e acesso a objetos remotos, que podem ter que ser consideradas no projeto.

Foi dito "podem ter que ser consideradas", e não "devem ser consideradas", pois essa é uma decisão da equipe de desenvolvimento, que deve decidir se é mesmo necessário incorporar aos diagramas a interação entre objetos, por exemplo, para efetuar acesso a objetos remotos (incluindo objetos das APIs Java responsáveis pelo serviço).

Deve-se, neste trabalho, considerar dois pontos.

Em primeiro lugar, deve-se definir o responsável por implementar o código da aplicação. A aplicação está sendo desenvolvida por diversos motivos, mas o principal é gerar a informação necessária para quem vai implementar a aplicação. Caso os programadores façam parte de uma fábrica de software externa, que conhece bem os recursos de implementação mas não necessariamente tem maturidade suficiente para resolver as questões subentendidas, um projeto detalhado é fundamental.

Em segundo lugar, deve-se avaliar se os mecanismos que devem ser considerados na implementação foram definidos claramente e projetados. Nesse caso, em vez de poluir os diagramas de seqüência com um conjunto basicamente repetitivo de ações essencialmente iguais, é suficiente referenciar os pontos em que esses mecanismos devem ser considerados, incorporando notas nos diagramas. Isso pode ser suficiente mesmo naquela situação hipotética em que uma fábrica de software externa ficará responsável pela implementação.

Essa é uma grande vantagem da abordagem RUP referente à definição de mecanismos. Em primeiro lugar, o planejamento do projeto vai prever atividades para a definição e projeto dos mecanismos (quando não se segue um processo com essas características, normalmente não se prevê tempo para a abordagem dessas questões, e elas são tratadas às pressas ou, pior, de forma não muito consciente). A seguir, durante o projeto, os mecanismos são referenciados, onde for necessário, sem poluir os diagramas. Isso reduz o esforço de modelagem, sem reduzir a clareza do modelo, e padroniza a forma de utilização dos serviços prestados pelo ambiente, com soluções oriundas de um esforço intelectual consciente.

Uma vez que os mecanismos tenham sido definidos adequadamente, podem nem mesmo precisar ser referenciados explicitamente. Todo acesso a um componente EJB é necessariamente um acesso remoto. Resolvida a forma de efetuar acesso remoto, com a definição do mecanismo correspondente, não há por que citar, a cada acesso a um EJB, que o mecanismo de acesso remoto deve ser referenciado.

Por fim, a maioria dos mecanismos não depende de características da aplicação, mas do ambiente. Os mecanismos definidos para uma aplicação J2EE certamente poderão ser utilizados na seguinte, pelo menos aqueles mais gerais e, desta forma, o projeto de aplicações será simplificado.

Por outro lado, embora a definição de mecanismos simplifique o projeto, o *gap* semântico, citado por Booch, e já comentado neste texto, aborda outras questões não resolvidas pelo uso de mecanismos.

Uma questão chave é o fato de haver dois ambientes distintos – um servidor web e um servidor EJB – que precisam se comunicar. Esses ambientes são organizados de forma muito diferente um do outro, com base em princípios diferentes. O ambiente

web é acessado via protocolo HTTP, que é *stateless* (não preserva estado conversacional entre duas solicitações de usuários). Para que o estado conversacional possa ser preservado, o ambiente web dispo do recurso de instanciar, no servidor web objetos da classe HttpSession – um para cada usuário ativo. Estes são objetos que vivem o tempo de uma sessão conversacional com o usuário e que podem armazenar outros objetos (geralmente *javabeans*, objetos Helpers). Quando um cliente inicia uma sessão conversacional com o servidor web, um objeto HttpSession é instanciado e associado àquele cliente. Mensagens posteriores enviadas por aquele cliente sempre serão automaticamente associadas à mesma instância de HttpSession, até que a sessão seja encerrada (explicitamente, a partir de uma ação do cliente, ou por tempo excedido). Cada interação do cliente com o servidor pode implicar na necessidade de armazenar ou recuperar informações na HttpSession daquele cliente. Eventualmente, algumas dessas informações podem ter que ser repassadas para os objetos de negócio da camada EJB, ou as informações enviadas pela camada de negócio podem ter que ser armazenadas na HttpSession. Essas são tarefas típicas de controle efetuadas pelos Dispatchers web.

Uma outra questão é a manipulação de objetos denominados Helpers. Esses são *javabeans* utilizados para transmitir informações entre as camadas de negócio e web, nos dois sentidos. Eventualmente, um Helper será devolvido pelo SessionFaçade para o Dispatcher, que vai armazená-lo na HttpSession, e também disponibilizá-lo a uma Server Page.

Essas questões são citadas para caracterizar a necessidade de, no projeto, descrever em detalhes os aspectos típicos do ambiente, que não foram considerados quando se modelou a aplicação do ponto de vista de análise. Os diagramas e comentários a seguir exemplificam algumas dessas questões.

Para completar o estudo de caso, apresentam-se e comentam-se brevemente os diagramas de seqüência de projeto. Os comentários não estão sendo feitos de forma integral, pois a descrição efetuada na análise é suficiente para esclarecer as intenções do diagrama. Apenas questões específicas de projeto são aqui comentadas.

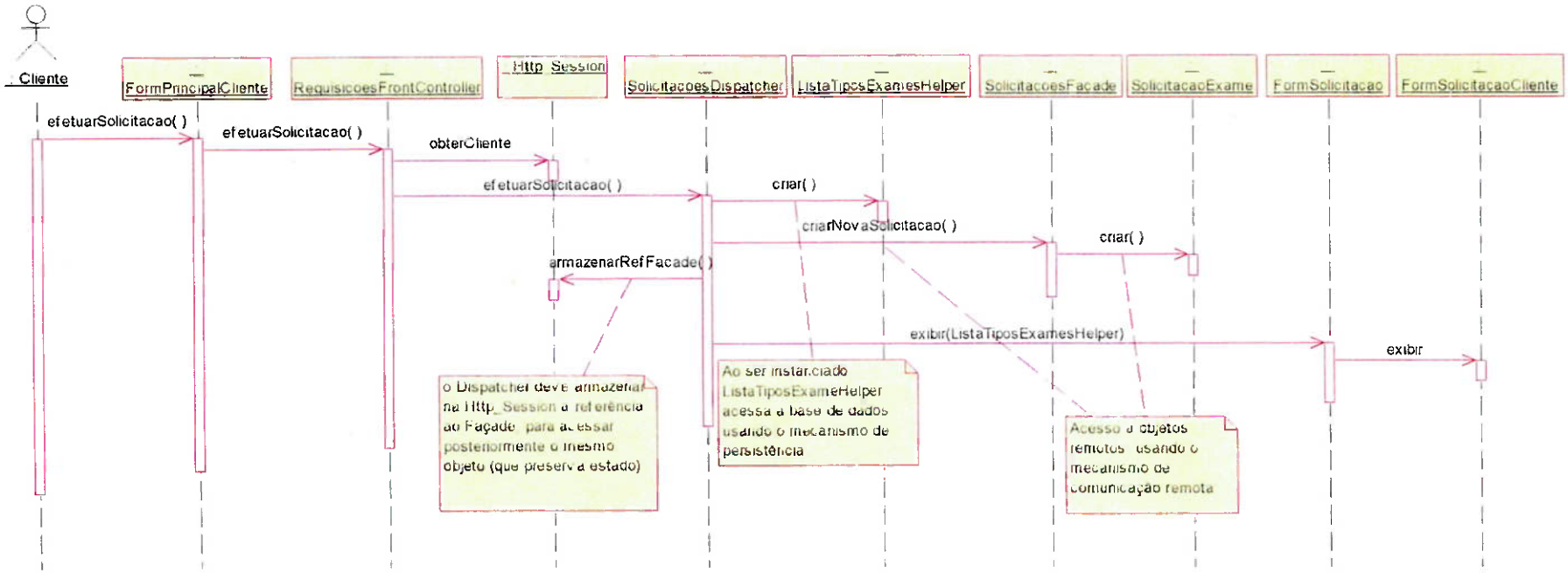


Figura 5-9 Diagrama de seqüência (projeto) parte 1/3

No diagrama da figura 1.9, incluiu-se um objeto da classe Http_Session. Cada sessão de usuário é caracterizada por uma instância desse objeto armazenada no servidor web, que contém informações sobre a conversação entre o usuário e o sistema, preservando a sessão do usuário. Na prática, o objeto Http_Session armazena outros objetos Java, e os sucessivos acessos de um determinado usuário ao sistema automaticamente vão acessar a mesma instância de Http_Session. Os stateful session beans da camada de negócio também têm o objetivo de preservar informações da sessão do usuário. Como utilizar os dois recursos é uma decisão do projetista, mas eles não são mutuamente exclusivos, e podem ser complementares.

A interação supõe que o objeto Cliente foi previamente armazenado na Http_Session, quando o cliente se identificou perante o sistema. A cada interação do usuário, o Cliente é recuperado da Http_Session e inspecionado, para garantir que ele é um cliente válido autenticado.

Uma nota no diagrama reforça que uma referência ao SessionFaçade é armazenada pelo Dispatcher na Http_Session, para acesso posterior, pois os Servlets, ao contrário dos stateful session beans, não preservam estado diretamente, e o Dispatcher é um Servlet.

Duas notas incluídas no diagrama referenciam pontos em que devem ser considerados os mecanismos de persistência e de comunicação remota. Essa é uma solução mais econômica que incluir nos diagramas a interação entre objetos que vão implementar o mecanismo.

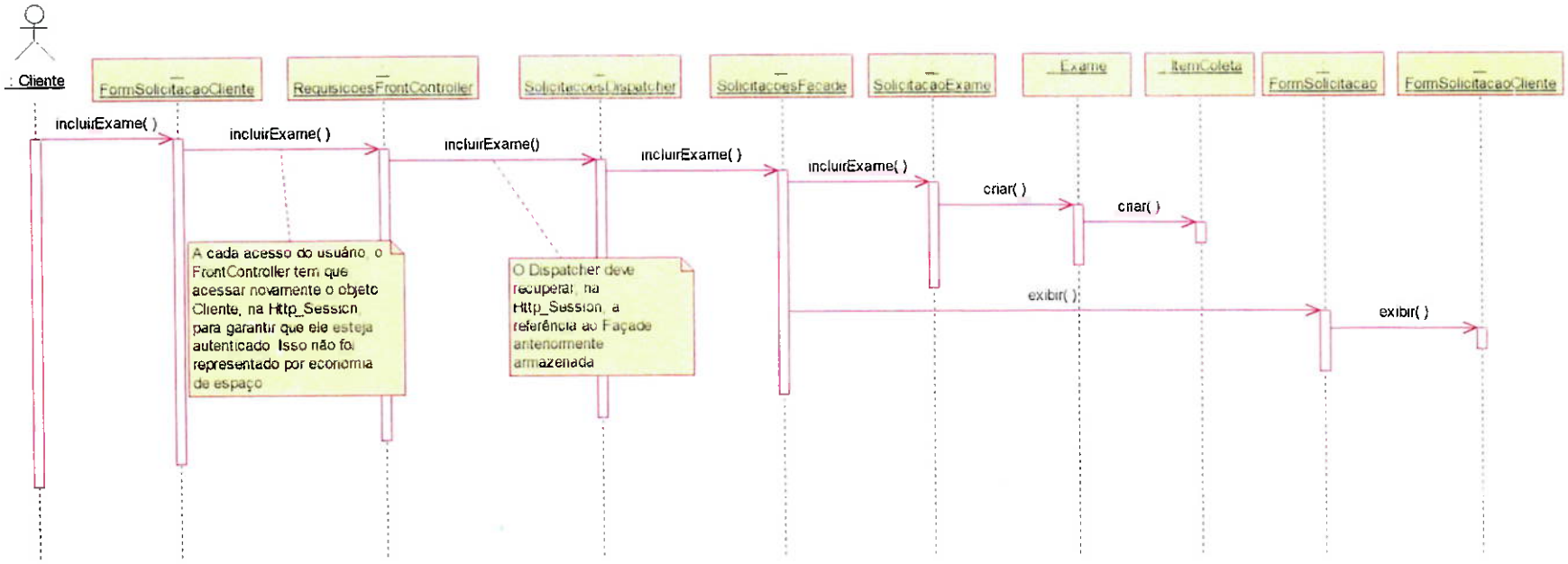


Figura 5-10 Diagrama de seqüência (projeto) parte 2/3

O diagrama da figura 1.10 apresenta a segunda parte da interação, efetuada repetidas vezes (uma vez para cada exame incluído na solicitação).

Duas notas no diagrama explicitam a comunicação entre os objetos da camada web e a Http_Session. Essa interação poderia ser explicitada no diagrama. Não o foi apenas por questões de espaço.

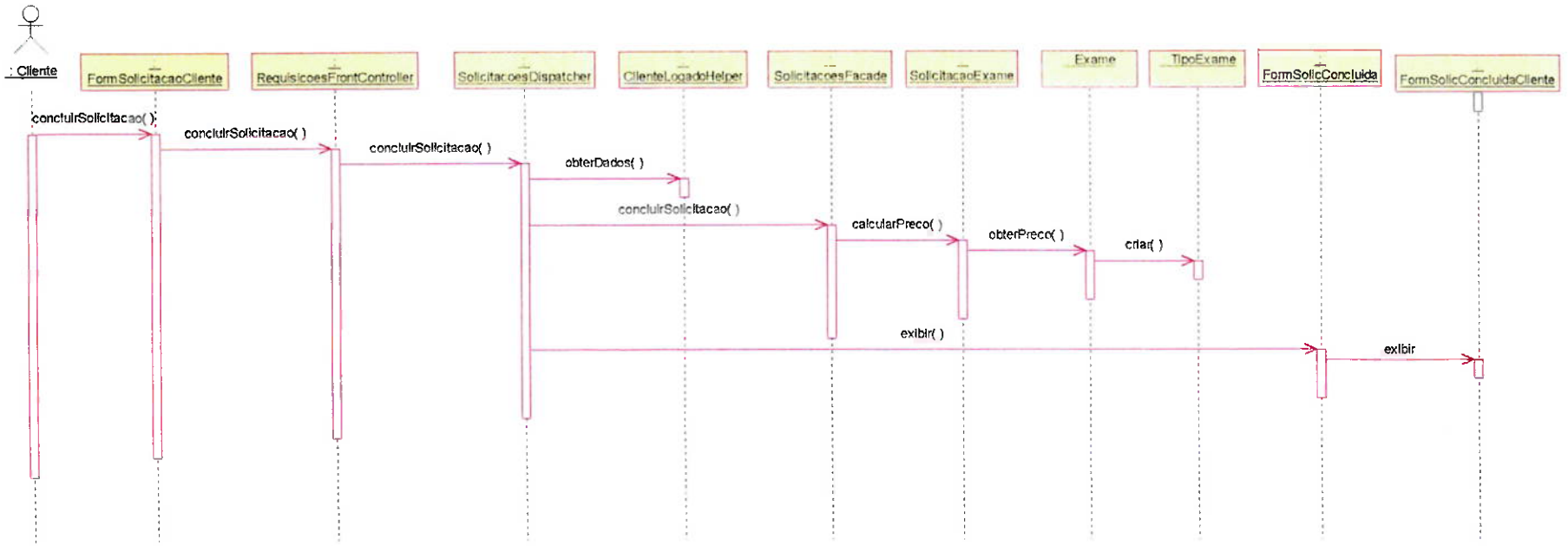


Figura 5-11 Diagrama de seqüência (projeto) parte 3/3

No diagrama da figura 1.11, a solicitação é concluída. A comunicação com a Http_Session foi omitida, por já ter sido descrita no diagrama anterior. Em uma caso real, seria melhor no mínimo referenciá-la em uma nota no diagrama.

Notar que o acesso a SolicitacaoExame é um acesso remoto, mas não o acesso a Exame e a TipoExame, pois estas são classes do EJB SolicitacaoExameEJB.

5.12 *Conclusões*

O desenvolvimento do estudo de caso foi norteado por:

- Aspectos técnicos das disciplinas técnicas do RUP;
- Pelos elementos estruturadores definidos no capítulo 4;
- Por design patterns.

A utilização do RUP foi muito informal, selecionando artefatos fundamentais de algumas disciplinas técnicas. O critério de seleção dos artefatos foi a efetiva necessidade de informação para avançar o desenvolvimento. Para um projeto real, outros aspectos teriam que ser considerados, inclusive a eventual adoção de um ciclo de vida iterativo.

O objetivo é ilustrar que, considerando os elementos citados, de forma integrada, o desenvolvimento da aplicação é efetuado de forma que, tanto as necessidades gerais de um projeto de software, quanto as específicas da plataforma J2EE, são adequadamente endereçadas, e que a complexidade do desenvolvimento para a plataforma é efetivamente reduzida.

6. Considerações Finais

6.1 *Introdução*

A atividade profissional do autor desta dissertação envolveu, no primeiro semestre de 2001, a coordenação de projetos de desenvolvimento de software na plataforma J2EE. A equipe era compostas por oito profissionais, com amplo conhecimento engenharia de software e experientes na plataforma Microsoft, mas que desconheciam a plataforma Java. Houve uma auto-capacitação em Java e J2EE ao longo dos projetos. Houve também um esforço efetivo para entendimento da arquitetura da plataforma, e para a definição do que, informalmente, poderia ser chamado de arquitetura de software dos aplicativos em desenvolvimento.

Às vésperas de iniciar a implementação do aplicativo, um consultor da empresa que fornecia o servidor de aplicações ao projeto finalmente pôde ser acionado, e passou três dias com a equipe, apresentando conceitos e discutindo o modelo já elaborado.

Para surpresa a equipe, uma série de decisões de arquitetura e projeto que pareciam bem embasadas, pelos textos que explicavam o funcionamento da plataforma J2EE, foram criticadas pelo consultor. Este apresentou novas alternativas de projeto baseadas em justificativas geralmente ligadas ao desempenho ou à escalabilidade.

Esse é um dos fatores geradores do tema desta dissertação: a percepção, com base numa experiência real, de que há um *gap* semântico entre as abstrações J2EE e a aplicação que deve ser construída.

O outro fator gerador do tema desta dissertação é a percepção de quão sofisticada é a concepção da plataforma J2EE, de como essa arquitetura é útil para produzir aplicações empresariais com alta qualidade e, mais importante, de que essa arquitetura tem muito em comum com preceitos básicos propostos pelo RUP, embora ambos estejam em planos diferentes (um processo de engenharia de software versus uma plataforma de implementação e execução de aplicações). Desde 1998 o autor desta dissertação estuda o RUP, por considerá-lo o processo de software mais efetivo e mais bem elaborado dentre os que já teve acesso.

6.2 *O abismo entre as práticas e as Boas Práticas de software*

Num artigo denominado "Objetos Ainda São Relevantes?" [BOOCH2002], Booch afirma considerar que, na década passada, houve dois desenvolvimentos muito interessantes da área de orientação a objetos: patterns e a UML. No entanto, se se ampliar o universo, pensando em engenharia de software em geral, outros desenvolvimentos poderiam também ser citados: o desenvolvimento do conceito de arquitetura de software; a evolução das técnicas de modelagem de software; o próprio RUP, um processo de engenharia de software, muito mais completo e efetivo que os que o antecederam.

A distância que separa um método como o OMT – publicado em 1991 e muito popular durante a década de 90 no Brasil –, do RUP, é imensa. Através dele, considera-se que já se atingiu um grau de maturidade suficiente para a produção de software empresarial com qualidade adequada em escala industrial.

Infelizmente, a realidade do desenvolvimento de software é muito diferente disso. Já foi dito que, em poucas áreas da atividade humana, a distância entre a prática efetiva e as boas práticas conhecidas é tão grande quanto na indústria de software.

Vigora em muitas organizações uma crença de que implantar um processo de software até aumenta a qualidade do software produzido, mas o esforço necessário impacta negativamente, e por isso não é viável, na prática. Parte das instituições de ensino superior, no Brasil, forma profissionais de informática que não conhecem princípios básicos de orientação a objetos.

A qualidade do software produzido, inclusive por grandes organizações empresariais muito bem sucedidas, é muitas vezes insatisfatória. Sob pretexto de urgência de prazos, constrói-se aplicações sem nenhum esforço de projeto efetivo. Essas aplicações posteriormente sofrerão sucessivas manutenções que as deixarão ainda mais descaracterizadas, em um círculo vicioso que amarra grandes equipes de desenvolvimento a atividades de correção de problemas em aplicações em produção.

Tudo isso era verdade nos anos 70, quando se identificou uma crise na indústria de software, e continua sendo verdade hoje, quando não se fala tanto no assunto. E,

ainda assim, software se tornou hoje muito mais presente em nossas vidas do que se poderia imaginar nos anos 70.

6.3 *Aproximação às práticas das boas práticas*

O ferramental tecnológico e metodológico disponível para a produção de software de qualidade é considerável. Quando esse ferramental é explorado em detalhes, mitos são derrubados, e vê-se que ele pode e deve efetivamente ser utilizado.

Esta dissertação propõe o uso de um processo de software, especializado a uma plataforma de desenvolvimento – e considerando a experiência concentrada em patterns de arquitetura e de projeto –, como sendo a base para a construção de software de empresarial de qualidade.

A seguir, detalham-se os pontos fortes dos elementos básicos da abordagem proposta.

6.3.1 Pontos Fortes do RUP

O RUP é útil para profissionais com variados graus de experiência e habilidade, pois sua estrutura em camadas permite que ele se dirija adequadamente a cada um desses perfis.

Afora questões gerenciais e de definição do ambiente, abordadas detalhadamente pelo RUP mas fora do escopo deste texto, o RUP define as disciplinas técnicas do processo de software de forma muito completa e aprofundada:

- A disciplina Modelagem de Negócios é adequadamente enfatizada, e utiliza recursos de modelagem propositalmente similares aos utilizados para a modelagem de software, simplificando o aprendizado e tornando mais suave a transição entre os dois domínios de conhecimento;
- A disciplina de Gerência de Requisitos é um ponto forte do processo, contemplando a definição detalhada de um Modelo de Casos de Uso, que norteará todo o ciclo de desenvolvimento;
- A disciplina de implementação é auxiliada por diretrizes de programação em diversas linguagens e ambientes já previamente elaboradas;

- A disciplina de testes é amplamente integrada à disciplina de Gerência de Requisitos, e contempla uma abordagem de automação de testes, com o uso de ferramentas de software especializadas.

Particularmente, a disciplina foco desta dissertação, Análise e projeto, contempla:

- Diretrizes básicas referentes à orientação a objetos;
- Definição de tipos distintos de classes na análise, o que ajuda a definir um modelo que pode efetivamente evoluir para a implementação do software, e não apenas um modelo conceitual;
- Diretrizes para a modelagem estática e dinâmica das aplicações;
- Atividades voltadas para a definição dos diversos aspectos de uma arquitetura de software, com os benefícios que isso traz para a simplificação e coerência do projeto;
- Abordagem para detalhamento progressivo do modelo, pela definição e posterior detalhamento de subsistemas e interfaces; pela distinção de atividades de análise e de projeto;
- Ênfase na identificação de elementos genéricos do modelo, pela definição de mecanismos de arquitetura.

Há diversos pontos chave positivos, não enfatizados por conta dos objetivos desta dissertação, como o ciclo de vida iterativo, a construção do processo a partir do nível 3 de maturidade do CMM-SW, e o uso da UML para modelagem.

6.3.2 Pontos fortes da plataforma J2EE

J2EE é uma especificação de interfaces (APIs – application programming interfaces) e de comportamento, sem definir implementações. A especificação é aberta e pode ser implementada por qualquer empresa, de modo que há concorrência para prover o melhor servidor de aplicações J2EE (IBM, BEA, Sun, Oracle, entre muitas outras, competem nesse mercado). As empresas que desenvolvem aplicações J2EE não estão necessariamente amarradas a um único fornecedor, já que, em princípio, as diversas implementações de servidores de aplicação implementam a mesma especificação, de modo que é possível mudar de servidor de aplicações sem mudar a aplicação.

Por conta de a plataforma J2EE ser baseada em Java, há também independência da aplicação em relação ao sistema operacional utilizado no servidor. Uma aplicação desenvolvida para o ambiente Windows pode ser migrada para Linux, por exemplo, sem qualquer alteração no código.

O fato de que todas as tecnologias da plataforma se baseiam na linguagem Java é também um ponto forte, pois esta é uma linguagem puramente orientada a objetos e, tendo sido concebida a partir da linguagem C++ como referência, pôde identificar as falhas desta linguagem, e corrigi-las. O uso de Java certamente estimula a adesão ao paradigma de orientação a objetos.

Um conjunto de tecnologias e de recursos de infra-estrutura é necessário para o desenvolvimento de aplicações corporativas na web. Um ponto forte da plataforma J2EE é organizar esse conjunto de tecnologias como parte de um todo coerente e integrado, em torno de uma **arquitetura** claramente definida.

O paralelo que é possível definir entre a arquitetura da plataforma e a arquitetura de software das aplicações J2EE é mais um ponto forte da plataforma, na medida em que essa arquitetura favorece o desenvolvimento de aplicações em camadas, conforme os patterns Layers e MVC, o que é ainda reforçado pela definição de tipos de componentes especializados

Um ponto forte da plataforma é a proposta de prover, de forma padronizada, o conjunto de serviços de infra-estrutura, de modo que a equipe de desenvolvimento não tenha que desenvolver artesanalmente esses serviços. A definição de *pools* de recursos (conexões a bancos de dados, *threads* de execução, instâncias de componentes de negócio) gerenciados pela plataforma, simplifica o projeto das aplicações ao mesmo tempo que aumenta o potencial desempenho e escalabilidade.

Um aspecto ainda mais importante é a forma como a plataforma trata a persistência de objetos em bases relacionais. Todo um modelo de persistência está definido, e é possível, em muitos casos (quando se utiliza entity beans do tipo CMP – container managed persistence), que todas as tarefas de persistência sejam deixadas por conta do container, sem que a equipe de desenvolvimento tenha que escrever nenhum código, para realizar uma tarefa razoavelmente complexa.

O sucesso da plataforma J2EE, principalmente nos Estados Unidos, provavelmente influenciou a Microsoft a evoluir seu próprio ambiente de desenvolvimento para o que hoje é denominado .Net. Isso mostra a força e a qualidade da plataforma. Hoje (mas não antes do .Net ser anunciado), a própria Microsoft reconhece uma superioridade tecnológica entre a plataforma J2EE e o ambiente Microsoft pré .Net.

6.3.3 Benefícios do uso de patterns

A engenharia de software persegue a reutilização de código a décadas, com sucessos relativamente pequenos. O paradigma de orientação a objetos foi disseminado muito em torno da perspectiva de reuso de classes – o que acabou não se confirmando na escala inicialmente imaginada. Hoje a proposta de reuso é mais centrada em componentes (trechos de código – no mundo da orientação a objetos, grupos de classes – que constituem unidades de implantação, têm uma interface e responsabilidades claramente definidas), por serem elementos de maior granularidade que classes de objetos.

Patterns são uma forma de reuso de conhecimento, de projeto. Os patterns GoF – os primeiros a ganhar destaque – são extremamente bem concebidos e efetivamente úteis para resolver questões importantes.

Projetar considerando patterns – e as boas práticas de engenharia de software que eles concretizam – torna o projeto de software uma atividade menos empírica, mas racional. O uso de patterns é por isso muito eficaz como ferramenta para fomentar o amadurecimento dos projetistas de software. Quando se compreende a essência de um conjunto de patterns, e os objetivos subjacentes à sua definição, naturalmente passa-se a aplicar esses princípios de forma ampla em projetos, mesmo em situações em que patterns específicos não se apliquem.

Patterns simplificam a atividade de projetar software. Em vez de uma página em branco (ou de uma janela em branco de um software CASE), agora dispõe-se de um conjunto de "moldes" de princípios e soluções de projeto, que serão utilizados para montar o projeto do aplicativo.

Os patterns podem ser também vistos como estruturados em camadas. Na base estão os patterns GRASP, definidos por Larman, que definem princípios básicos de análise

e projeto orientado a objetos. Na camada imediatamente acima, encontram-se os patterns GoF, que definem estratégias para resolver problemas de projeto já mais especializados, embora muito freqüentes. Estes, por sua vez, são a referência para a definição dos patterns J2EE, especializados a uma plataforma de desenvolvimento, e que vêm ajudar a preencher o gap semântico que é o principal ponto fraco do uso dessa plataforma.

6.3.4 O gap semântico

Adotado um processo de software que implementa as boas práticas de engenharia de software essenciais, como o RUP, dominada a complexidade tecnológica de uma plataforma, como a J2EE, ainda há obstáculos para o desenvolvimento de aplicações empresariais no ambiente web: o gap semântico entre as abstrações da plataforma e as aplicações construídas.

A utilização de patterns J2EE no projeto é um fator para reduzir esse gap semântico. A abordagem adotada nesta dissertação pretende alavancar o uso dos patterns (de arquitetura, de projeto, e J2EE), utilizando-os como ferramenta para predefinir um conjunto de elementos estruturadores contemplados pelo RUP (arquitetura de software, mecanismos de arquitetura, diretrizes para conversão de do modelo de análise em projeto).

Arquitetura de Software é um tema que ganhou destaque nos últimos anos. Sua definição é relevante para qualquer aplicação não trivial, e mais relevante se o ambiente de desenvolvimento e execução é complexo, como no caso de aplicações web empresariais. A definição adequada de uma arquitetura de software é um fator chave para a definição de um projeto de software bem concebido, e é também útil para simplificar o projeto, na medida em que padrões de estrutura e comportamento são ressaltados. A definição de um modelo arquitetura de software para a plataforma J2EE com base nos patterns de arquitetura Layers e MVC simplifica a abordagem de arquitetura para as aplicações J2EE considerando um padrão de organização já amplamente experimentado.

A definição de mecanismos de arquitetura específicos para a plataforma J2EE também é um fator para incrementar a qualidade e simplificar o projeto. Questões chave terão sido previamente identificadas e solucionadas de forma padronizada.

Equipes inexperientes no uso da plataforma J2EE certamente aumentarão o grau de compreensão dos recursos da plataforma e poderão se valer de experiência prévia acumulada.

O RUP contempla um recurso simples mas poderoso para a definição de um modelo de análise que seja mais que apenas um modelo de domínio (a utilização dos estereótipos boundary, control e entity). A definição de padrões para conversão desses tipos de classes em classes de projeto, em conjunto, ajuda a que se faça uma transição mais suave entre o modelo de análise da aplicação e um modelo de projeto que efetivamente possa ser utilizado como fonte para implementação.

O conjunto desses recursos é útil para direcionar e simplificar as atividades de análise de projeto, e contribui para que possa ser implantado um processo no qual todas os aspectos do projeto de um aplicativo são abordados no grau de detalhe adequado.

6.4 *Conclusões*

Houve um tempo em que orientação a objetos era discutida por um ponto de vista conceitual (conceitos como encapsulamento, herança, reuso), mas pouco praticada. Agora os ambientes de desenvolvimento, bastante complexos, são baseados no uso de linguagens orientadas a objetos. Orientação a objetos precisa ser praticada.

Por outro lado, agora a engenharia de software atingiu um grau de maturidade suficiente para que projetos consistentes de software orientado a objetos sejam definidos. Orientação a objetos pode ser praticada.

Qualidade e previsibilidade continuam a ser o ponto fraco da indústria de software, em um mundo cada vez mais dependente de software.

É necessário que o desenvolvimento de software passe a ser visto como atividade tecnicamente embasada, menos empírica, e que o projeto de software passe a ser adequadamente enfazitado, para o sucesso a longo prazo dos aplicativos desenvolvidos.

Os elementos para o desenvolvimento de software de qualidade já estão definidos. Um processo como o RUP e uma plataforma como a J2EE são prova do

amadurecimento da indústria de software. Agora os projetistas de software têm que se colocar à altura das ferramentas a seu dispor.

Os primeiros métodos orientados a objeto eram muito focados nas atividades de análise, mas a transição para projeto e implementação era freqüentemente negligenciada. Essa era mais lacuna básica, que dificultava enormemente a utilização de modelos orientados a objeto como base para implementação. O RUP veio abordar essa questão de forma muito bem sucedida. Esta dissertação enriqueceu esse tema com foco numa plataforma de desenvolvimento complexa e poderosa.

Buscou-se apresentar os elementos essenciais que podem contribuir para esse sucesso, e prestar uma contribuição, simplificando seu uso pela padronização de elementos estruturadores essenciais.

6.5 *PRÓXIMOS TRABALHOS*

Esta dissertação procurou apresentar, desenvolver e exemplificar o tema da especialização da disciplina análise e projeto do RUP para a plataforma J2EE, mas certamente não o esgotou. Os três temas de maior destaque – arquitetura de software, mecanismos de arquitetura, patterns de projeto – são candidatos a ser abordados individualmente em futuros trabalhos, que tenham ou não como foco a arquitetura J2EE.

Acredita-se na conveniência de continuar aprofundando esse estudo considerando especificamente a plataforma J2EE, pois o grande espaço ocupado por essa plataforma e a evolução que tem sofrido indicam que, provavelmente, ela ainda será utilizada por longo tempo. É claro que pode ocorrer uma grande guinada tecnológica que torne obsoleta a plataforma J2EE, mas nada atualmente indica que isso ocorrerá no futuro próximo.

Outra possibilidade de pesquisa seria considerar o mesmo tema voltado para a grande concorrente a plataforma J2EE, o .Net, da Microsoft, e eventualmente comparar as soluções encontradas para as duas plataformas, como forma de comparar as próprias plataformas – um assunto de grande interesse.

O assunto arquitetura de software foi apresentado de forma básica, considerando apenas a vista lógica da arquitetura (organização do sistema em subsistemas), e

apenas visando a organização do software em alto nível, para quaisquer tipos de sistemas. Um tema para estudo interessante seria definir soluções de arquitetura de software (envolvendo, talvez, a composição de patterns de arquitetura) típicas para determinados tipos de aplicações web empresariais.

O assunto Mecanismos de Arquitetura foi apresentado do ponto de vista do RUP, e os mecanismos detalhados são aqueles enfatizados pelo próprio RUP. A pesquisa de outros mecanismos ou outros tipos de mecanismos seria uma contribuição valiosa, no sentido de encontrar soluções padronizadas para problemas comuns, e desta forma simplificar e valorizar o projeto de aplicações. Assim como há um catálogo de patterns J2EE, seria muito conveniente que houvesse um catálogo de mecanismos de arquitetura para a plataforma J2EE.

Apenas alguns patterns J2EE foram apresentados, dos diversos desenvolvidos pela equipe da Sun Microsystems. Diversos outros poderiam ser estudados, com vistas à sua aplicação à especialização do RUP para a plataforma J2EE. Um tema mais ambicioso seria a pesquisa de novos patterns de projeto J2EE, para resolver problemas ainda em aberto. Certamente tipo de problema existe (por exemplo, onde preservar o estado da conversação entre o usuário e o sistema, no servidor web ou no servidor EJB) e sua solução traria contribuições efetivas para todo um ramo da indústria de software.

Lista de Referências

[ALLAMARAJU 2001] Subrahmanyam Allamaraju et al 2002. *Professional Java Server Programming J2EE 1.3 Edtion.* Wrox Press Ltd.

[BODOFF 2002], S. Bodoff et al 2002. *The J2EE Tutorial.* http://java.sun.com .

[BOOCH 1994] Grady Booch 1994. *Object-Oriented Analysis and Design with Applications.* Addison-Wesley.

[BOOCH 1995] Grady Booch 1995. *Object Solutions—Managing the Object-Oriented Project.* Addison Wesley Longman.

[BOOCH 1998] Grady Booch, James Rumbaugh, Ivar Jacobson 1998. The Unified Modeling Language User Guide. Addison-Wesley.

[BOOCH 2002] Grady Booch 2002. *Are Objects Still Relevant?* www.rational.net

[BUSCHMANN 1996] Frank Buschmann, Régine Meunier, Hans Rohnert, Peter Sommerlad, e Michael Stahl 1996. *Pattern-Oriented Software Architecture— A System of Patterns*, New York, NY: John Wiley and Sons, Inc.

[BYOUS 2002] J. Byous 2002. *Java Technology: an Early History.* http://java.sun.com .

[DEEPAK 2001] Deepak Alur, John Crupi, Dan Malks, 2001. *Core J2EE Patterns – Best Practices and Design Strategies,* Upper Saddle River, NJ: Prentice Hall Ptr.

[GAMMA 1994] Erich Gamma, Richard Helm, Ralph Johnson e John Vlissides 1994. *Design Patterns—Elements of Reusable Object-Oriented Software.* Addison Wesley Longman.

[JACOBSON 1992] Ivar Jacobson, Magnus Christerson, Patrik Konsson, Gunnar Overgaard 1992. *Object-Oriented Software Engineering. A Use Case Driven Aprproach.* Addison-Wesley.

[JACOBSON 1999] Ivar Jacobson, Grady Booch, James Rumbaugh, 1999. *The Unified Software Development Process.* Addison-Wesley.

[KHAWAR 2002] Khawar Zaman Ahmed, Cary E. Umrysh 2002. *Developing Enterprise Java Applications with J2EE and UML*, Addison Wesley Longman.

[KRUCHTEN 2000] Philippe Kruchten 2000. *The Rational Unified Process An Introduction*. Addison-Wesley.

[LARMAN 2001] Craig Larman 2001. App*lying UML and Patterns – An Introduction to Object-Oriented Analysis and Design and the Unified Process*,

[ROMAN1999] Ed Roman. *Mastering EnterpriseJavaBeans and the Java 2 Platform, Enterprise Edition* 1999. Wiley Computer Publishing.

[RUMBAUGH 1991] James Rumbaugh, Michael Blaha, William Premerlani, Frederick Eddy, William Lorensen 1991. *Object-Oriented Modeling and Design.* Prentice Hall.

[RUP 2002] RationalSoftware Corporation 2002. *Rational Unified Process* www.rational.com

[SEI 1995] Software Engineering Institute – Carnegie Mellon University 1995. *The Capability Maturity Model: Guidelines for Improving the Software Process.* Addison-Wesley.

[STANDISH 1999] Standish Group 1999. *Chaos Report*. www.standishgroup.com (acesso restrito).

[SUN 2002] *Designing Enterprise Applications with the Java 2 Platform, Enterprise Edition*. http://java.sun.com/j2ee/docs.html