“Uma Avaliação da influência da Arquitetura no Desempenho de Sistemas de Software “

Gisele Consoline

Trabalho Final de Mestrado Profissional em Computação

Instituto de Computação
Universidade Estadual de Campinas

Uma Avaliação da influência da Arquitetura no Desempenho de Sistemas de Software

Gisele Consoline

Fevereiro de 2006

Banca Examinadora:

- **Prof. Dr. Mário Lúcio Côrtes (Orientador)**
  Instituto de Computação – UNICAMP
- **Profa. Thelma Cecília dos Santos Chiossi (Co-orientadora)**
  Instituto de Computação - UNICAMP
- **Profa. Dra. Ana Cervigni Guerra**
  Centro de Pesquisas Renato Archer – CenPRA/MCT
- **Profa. Dra. Cecília Mary Fischer Rubira**
  Instituto de Computação - UNICAMP
- **Profa. Dra. Eliane Martins (Suplente)**
  Instituto de Computação – UNICAMP

FICHA CATALOGRÁFICA ELABORADA PELA BIBLIOTECA DO IMECC DA UNICAMP
Bibliotecária: Miriam Cristina Alves – CRB8a / 5094

Consoline, Gisele
C765u Uma avaliação da influência da arquitetura no desempenho de sistemas de software / -- Campinas, [S.P. :s.n.], 2006.

Orientadores: Mario Lúcio Côrtes; Thelma Cecília dos Santos Chiossi
Trabalho final (mestrado profissional) - Universidade Estadual de Campinas, Instituto de Computação.

1. Software - Arquitetura. 2. Software (Sistemas). 3. Software – Testes. I. Côrtes, Mario Lúcio. II. Chiossi, Thelma Cecília dos Santos. III. Universidade Estadual de Campinas. Instituto de Computação. IV. Título.

Título em inglês: An evaluation of the Architecture's influence on Software systems performance.

Palavras-chave em inglês (Keywords): 1. Software - Architecture. 2. Software(Systems). 3. Software – Test.

Área de concentração: Engenharia de Computação, Engenharia de Software.

Titulação: Mestre em Engenharia de Computação

Banca examinadora: Prof. Dr. Mario Lúcio Côrtes (IC-UNICAMP)
Profa. Dra. Thelma Cecília dos Santos Chiossi (IC-UNICAMP)
Profa. Dra. Ana Cervigni Guerra (CenPRA/MCT)
Profa. Dra. Cecília Mary Fischer Rubira (IC-UNICAMP)
Profa. Dra. Eliane Martins (IC-UNICAMP)

Data da defesa: 06/03/2006

Programa de Pós-Graduação: Mestrado Profissional em Engenharia de Computação

“Uma avaliação da influência da Arquitetura no desempenho de Sistemas de software”

Este exemplar corresponde à redação final do Trabalho Final devidamente corrigida e defendida por Gisele Consoline e aprovada pela Banca Examinadora.

Campinas, 06 de março de 2006.

Prof. Dr. Mário Lúcio Côrtes
(Orientador)

Profa. Thelma Cecília dos Santos Chiossi
(co-orientadora)

Trabalho Final apresentado ao Instituto de Computação, UNICAMP, como requisito parcial para a obtenção do título de Mestre em Computação na área de Engenharia de Computação

TERMO DE APROVAÇÃO

Trabalho Final Escrito defendido e aprovado em 24 de fevereiro de 2006, pela Banca Examinadora composta pelos Professores Doutores:

Profa. Dra. Cecília Mary Fischer Rubira
IC - UNICAMP

Profa. Dra. Ana Cervigni Guerra
CenPRA

Prof. Dr. Mário Lúcio Côrtes
IC- UNICAMP

Dedicatória

Agradeço ao James Rumbaugh pela compreensão desse meu trabalho e por ter me enviado a seguinte mensagem: **"I am happy that many of my books have been translated into Brazilian Portuguese. Brazil is a large and important country and I hope that it will become a leader in software engineering."**

Dedico este trabalho à Deus, à minha família, aos meus amigos Marcos, Paulo, Janaine, Fernando Galdino e à pessoa que mais me apoiou nessa luta, meu amado marido Michel do Couto.

Resumo

Buscar soluções que atendam todas as necessidades do cliente pode não ser uma tarefa simples, pois pode haver falhas no entendimento de requisitos do sistema. Essa falha pode ocasionar problemas para o sistema ocasionando retrabalho e comprometendo a entrega do sistema dentro do prazo previamente estimado. Por meio do presente estudo, que analisa um sistema de *Workflow* já existente através de um processo para avaliar a arquitetura, pretende-se contribuir apresentando um exemplo de aplicação de um processo de avaliação de arquitetura com ênfase na abordagem quantitativa. O objetivo é superar problemas usando medidas que facilitem a análise da arquitetura e principalmente auxilie o arquiteto na busca da solução arquitetural mais adequada para o projeto.

Abstract

To search for solutions that encompass all the client's requirements might not be a simple task, for misunderstandings of the implicit and explicit requirements could happen. Such misunderstandings could generate problems related. Rework could be necessary, implicating the delivery date previously estimated. The current study which analyses an existing Workflow system through a process of architecture evaluation, it is intended to contribute by presenting an example of the use of an architecture evaluation process with emphasis in a quantitative approach. The goal is to solve problems by using metrics which facilitate the architecture analysis and mainly assists the architect in the search of the more adjusted architectural solution for the project.

Lista de Tabelas

Lista de Figuras

1 Introdução

Considerando a evolução dos sistemas de *software*, dentro do contexto histórico, tem-se que aproximadamente há quatro décadas atrás eles constituíam uma pequena porção dos sistemas computacionais e, portanto, seus custos de desenvolvimento e manutenção eram insignificantes quando comparado ao *hardware*. Hoje em dia, os sistemas de *software* constituem uma grande parcela dos sistemas computacionais e seus custos cresceram significativamente. Como resultado desta mudança, muitos métodos têm sido propostos objetivando melhorar o processo de desenvolvimento bem como minimizar os custos de manutenção [Lun97]. Este cenário agrava-se quando sistemas complexos e de grande porte são considerados.

Para lidar com a complexidade e o tamanho de sistemas, arquitetos de *software* têm feito uso de princípios da Engenharia de *Software* como, por exemplo, ocultação de informações e abstração. Contudo, à medida que os sistemas tornam-se cada vez maiores, o uso de uma disciplina deve ser enfatizado de modo a obter resultados de baixo custo e maior qualidade. Dentro deste contexto, a arquitetura de *software* tem entrado em cena de modo a lidar com sistemas grandes e complexos com o objetivo de suprir os requisitos funcionais e não-funcionais (que serão vistos na seção 1.5 deste capítulo) e eliminar os riscos do sistema [Kaz94].

Deve-se notar que à medida que o tamanho e a complexidade dos sistemas de *software* aumentam, as estruturas de dados e algoritmos de computação podem sofrer alterações. Dessa forma, projetar a estrutura geral do sistema emerge como uma solução. Questões estruturais envolvem organização e estrutura geral de controle; protocolos de comunicação, sincronização; atribuição de funcionalidade a componentes de projeto; escalabilidade e desempenho [Kaz97]; seleção de alternativas de projeto. Estas questões compreendem o projeto de *software* no nível arquitetural. Para se desenvolver um projeto arquitetural de forma adequada se faz necessário atender requisitos não-funcionais bem definidos, como, por exemplo, desempenho, segurança, disponibilidade, manutenibilidade e funcionalidade [Bas98]. Se o desempenho for um requisito importante isso sugere que a arquitetura deve ser projetada para restringir as operações mais importantes dentro de um pequeno número de subsistemas com a menor comunicação possível entre esses subsistemas. Isso pode significar a utilização de componentes com granularidade

relativamente maior, em vez de componentes com menor granularidade, a fim de reduzir a comunicação entre eles [Som04]. Os sistemas de *software* normalmente representam o funcionamento das corporações, integrando seus clientes, fornecedores, parceiros e funcionários. O bom funcionamento dos sistemas de *software* é fundamental para a realização dos negócios da corporação e, conseqüentemente, para a sua existência, competitividade e evolução [Bar98].

Este capítulo apresenta os problemas existentes no processo de desenvolvimento de *software*, motivações, solução proposta, objetivos dessa dissertação bem como trabalhos relacionados ao assunto, conceitos de requisitos funcionais e não-funcionais e finalmente a organização dessa dissertação.

1.1 Motivação

É muito comum existir problema na escolha de arquiteturas, o que traz impactos sobre o resultado do desenvolvimento causando a insatisfação do cliente e comprometendo o sucesso do projeto. Estes impactos refletem diretamente na equipe de desenvolvimento, que enfrenta problemas de diversas naturezas e nem sempre tem um método objetivo para diagnosticá-los. A equipe muitas vezes não consegue identificar se o impacto foi causado pela escolha da arquitetura e muito menos identificar os gargalos e onde eles ocorrem para possibilitar correções no projeto. Dessa forma, uma correta análise desses impactos é a grande motivação para esta dissertação.

1.2 Solução proposta

A solução proposta constitui-se em exercitar a utilização de um processo formal já descrito na literatura para análise da arquitetura sobre um caso real de desenvolvimento (estudo de caso) visando levantar informações que possibilitem a identificação de impactos na arquitetura e a sua melhoria. A solução compreende os seguintes passos:

- Seleção de um processo formal para a avaliação da arquitetura
- Seleção de um sistema real para aplicação do processo de avaliação
- Aplicação de processo de medição em requisitos não-funcionais do sistema real
- Análise dos resultados das medidas e diagnóstico de problemas
- Proposta de redesenho da arquitetura

- Implementação da proposta
- Reaplicação do processo de medição para confirmar as melhorias nos resultados obtidos
- Elaboração das conclusões

1.3 Objetivo

O objetivo dessa dissertação é analisar a arquitetura de um sistema de *software* já existente, utilizando um processo formal de análise e, com base nos resultados dessa análise, apresentar sugestões de melhorias na arquitetura. Além disso, pretende-se mostrar que a alteração foi eficaz comparando as medidas de desempenho da arquitetura sugerida com as anteriores e apresentando os resultados dessa comparação.

1.4 Trabalhos Relacionados

Esta seção descreve estudos relativos à arquitetura de *software* envolvendo definições, as vantagens de se projetar e documentar a arquitetura de *software* avaliando diferentes tipos de arquiteturas, bem como a melhor forma de sua utilização em um determinado projeto e, finalmente, descrevendo possíveis riscos que levam ao desvio da arquitetura escolhida (ou seja, que a arquitetura não atende aos requisitos do sistema) e analisando e propondo uma solução para os desvios relatados.

Em um estudo feito por Bass [Bas98], tem-se a indicação de algumas vantagens de se projetar e documentar explicitamente uma arquitetura de *software*. As três vantagens indicadas por Bass, são:

- **Comunicação com os usuários chave do sistema:** a arquitetura é uma apresentação de alto nível do sistema, que pode ser utilizada como ponto de discussão com uma gama de diferentes usuários.
- **Análise de Sistema:** tornar explícita uma arquitetura de sistemas em um estágio inicial do desenvolvimento, significa que alguma análise pode ser realizada. As decisões de projeto de arquitetura têm um profundo efeito sobre se o sistema pode ou não cumprir requisitos importantes como desempenho, confiabilidade e facilidade de manutenção.

- **Reutilização em larga escala:** a arquitetura de sistemas é uma descrição compacta e administrável de como um sistema é organizado e de como os componentes operam entre si. A arquitetura pode ser transferida para sistemas com requisitos similares e, dessa maneira, pode fornecer apoio ao reuso de *software* em grande escala.

A escolha de uma arquitetura que atenda a todos os requisitos funcionais e não-funcionais que será visto na seção 1.5 desta dissertação, é uma etapa criticamente importante no projeto de um sistema [Kip98]. Torna-se cada vez mais evidente que processos de engenharia de *software* requerem um projeto arquitetural por vários motivos: É importante ser capaz de reconhecer estruturas comuns de modo que projetistas de *software* possam compreender as relações existentes entre sistemas e desenvolver novos sistemas baseados em variações de sistemas antigos [Fea02]; O entendimento de arquiteturas de *software* permite aos arquitetos tomarem uma decisão sobre alternativas de projeto [Olo99]; Uma descrição arquitetural do sistema é essencial para analisar e descrever propriedades de um sistema complexo [Zhu04]; O conhecimento de notações para descrever arquiteturas possibilita aos arquitetos apresentarem novos projetos de sistemas a outros membros de uma equipe de desenvolvimento; Prover a melhor solução de arquitetura que atenda as necessidades do projeto.

Uma vez definidas as vantagens da utilização de um projeto da arquitetura de *software*, o próximo passo será avaliar as diferentes arquiteturas. Segundo o estudo feito por James Kiper [Kip98], a enumeração dos riscos de um sistema pode ajudar a avaliar a arquitetura, pois a partir de uma tabela contendo os requisitos não-funcionais relacionados com os riscos enumerados pode-se chegar à conclusão de quais arquiteturas são mais indicadas para atender ao sistema. Esta matriz de risco mencionada será utilizada nesta dissertação para análise da arquitetura do caso de estudo.

Depois que a arquitetura é definida, o próximo passo é se avaliar os riscos que resultam em um desvio na arquitetura e aplicar um processo para avaliar esta arquitetura com o objetivo de eliminar estes riscos. Baseando-se em estudos tais como [Bab98] já publicados a respeito de processos para avaliar a arquitetura, verificou-se que um grande número de processos não menciona explicitamente medidas como parte integrante do processo. Em outro estudo, Kazman [Kaz00] propôs um processo para avaliar a arquitetura chamado ATAM (Métodos de Análise de relacionamento de arquitetura). Os passos propostos para a análise foram: 1) apresentar a ATAM.

2) apresentar perspectiva de negócio. 3)apresentar a arquitetura. 4) identificar o foco arquitetural. 5) gerar a árvore de requisitos de qualidade. 6) analisar o foco arquitetural. 7) investigar e priorizar os cenários. 8) analisar o foco arquitetural. 9) apresentar os resultados.

Tesoriero [Tes02] propôs um processo para avaliar a arquitetura através de medidas, documentar e comunicar esta arquitetura aos novos desenvolvedores, checando se a mesma foi entendida por eles. Além disso, a autora do estudo sugere a implementação de um processo para avaliar a implementação sem a necessidade de se rever todo o código. Os passos propostos para a análise são: 1) selecionar a perspectiva (foco) da avaliação; 2) definir regras e estabelecer medidas para a avaliação; 3) analisar a arquitetura planejada, para definir os objetivos dessa arquitetura; 4) analisar o código fonte para verificar como está a arquitetura atual; 5) comparar o estado atual da arquitetura ao estado planejado, para identificar desvios; 6) formular recomendações de mudanças para alinhar o estado atual ao planejado; 7) verificar que as violações de objetivos foram corrigidos, repetindo os passos 4 a 6.

Considerando-se que o objetivo dessa dissertação é analisar a arquitetura através de medidas baseando-se em um processo de avaliação, optou-se pelo processo de Tesoriero [Tes02], pois a utilização de medidas é parte integrante desse processo de avaliação de arquitetura.

Na próxima seção serão apresentados os conceitos básicos de requisitos funcionais e não-funcionais bem como o relacionamento dos requisitos com arquitetura de sistemas.

1.5 Requisitos de Sistemas de Software

Nessa seção serão introduzidos alguns conceitos para a compreensão do trabalho como os requisitos funcionais e não-funcionais, a relação entre a arquitetura de sistemas com esses requisitos e o conceito de visões da arquitetura de sistemas importantes no uso dos requisitos.

Os problemas que os arquitetos de *software* têm para solucionar são, muitas vezes, imensamente complexos. Compreender a natureza dos problemas pode ser uma tarefa muito difícil, especialmente se o sistema estiver na fase inicial. Conseqüentemente, é difícil estabelecer com exatidão o que o sistema deve fazer. O desenvolvimento de qualquer sistema de *software* começa com o levantamento das necessidades do usuário. O detalhamento dos requisitos

implícitos e explícitos do usuário são chamados de requisitos do sistema e o processo de descobrir, analisar, documentar e verificar esses requisitos é chamado de engenharia de requisitos [Som04].

A engenharia de requisitos é um processo de transformação das idéias que estão na mente dos usuários (a entrada) em um documento formal (saída); essa transformação só é possível através da determinação dos objetivos do produto e das restrições para sua operacionalidade, através de uma análise do problema, documentação dos resultados e verificação do entendimento do problema. A saída descreve *o que* o produto a ser desenvolvido deverá fazer, sem, entretanto, descrever *como* deve ser feito. Idealmente, esse documento deve ser completo e consistente; entretanto, a entrada para esse processo não tem nenhuma dessas propriedades, isto é, não é completa nem consistente. Como conseqüência, o processo de extração de requisitos não pode ser totalmente formal e, portanto, não pode ser totalmente automatizado.

Durante a extração de requisitos, o foco é o entendimento do produto a ser desenvolvido e de seus requisitos. Quanto mais complexo for o produto, mais difícil se torna o processo. Os requisitos podem ser tanto funcionais, descrevendo um serviço ou função do sistema, como não-funcionais, descrevendo entre outras informações, as restrições ao processo de desenvolvimento ou ao tempo de resposta do sistema. Os requisitos funcionais e não-funcionais serão descritos detalhadamente nas subseções deste capítulo.

1.5.1 Requisitos Funcionais

Os requisitos funcionais para um sistema descrevem a funcionalidade ou os serviços que se espera que o sistema forneça [Mac99]. Freqüentemente, o usuário descreve o problema em termos vagos, como, por exemplo, “Eu desejo automatizar os serviços de escritório”, ou então “Eu desejo um sistema que calcule o imposto a pagar”. Nesses dois casos, o usuário não deu nenhuma pista sobre o escopo do problema, suas restrições e outras informações indispensáveis para o início do desenvolvimento do sistema. Os requisitos do usuário (ou seja, os requisitos funcionais) devem ser analisados de maneira cuidadosa para que sejam determinadas as funções do sistema (o que é para o sistema fazer) e os requisitos do sistema (também chamados de requisitos não-funcionais que determinam as propriedades que restringem o desenvolvimento do sistema), de forma que o plano de desenvolvimento tenha uma base sólida.

Os usuários descrevem suas necessidades, e o desenvolvedor deve considerar essa descrição como ponto de partida para definição dos objetos e restrições e, baseando-se nessas informações, traçar o plano de projeto. Sob o ponto de vista do planejamento, podem-se considerar os requisitos do usuário avaliando-se, entre outras coisas, a complexidade do sistema e o seu custo. O usuário deve compreender e aprovar os objetivos e restrições definidos para o projeto, de forma que um contrato entre ele e o desenvolvedor possa ser estabelecido. Os requisitos do usuário podem ser expressos de diversas maneiras com diferentes níveis de detalhamento. Considerando por exemplo um sistema de controle bibliotecário, pode-se ter os seguintes requisitos:

1. O usuário deverá ser capaz de buscar a lista de livros cadastrados no sistema.
2. O sistema fornecerá telas apropriadas para o usuário ler os resultados da busca.
3. O usuário poderá ser capaz de listar todos os livros cadastrados em seu nome e marcar um determinado livro como emprestado.

Estes requisitos do usuário definem recursos específicos que devem ser fornecidos pelo sistema. Após a definição dos requisitos funcionais, deve-se observar quais os requisitos não-funcionais do projeto (o tempo de resposta dos resultados da busca, no caso) e avaliar se esses requisitos são viáveis ao projeto.

Embora possa parecer que o processo de extração, análise, especificação e validação de requisitos seja uma seqüência linear, essas atividades não podem ser totalmente separadas e executadas seqüencialmente; todas são intercaladas e executadas repetidamente [Car01]. Pode acontecer que algumas partes do sistema sejam analisadas e especificadas enquanto outras ainda estão sendo analisadas. Além disso, a fase de validação pode revelar problemas com a especificação, o que pode acarretar um retorno à fase de análise, e conseqüentemente, a fase de especificação; problemas com o entendimento do domínio exigem um retorno a essa atividade.

Com freqüência, o próprio documento de especificação de requisitos e o processo de extração dão novas idéias aos usuários sobre as suas necessidades sobre as funções do sistema. Portanto, mudanças nos requisitos acontecem na maioria dos sistemas complexos. Embora muitas delas sejam devido a mudanças nos requisitos dos usuários, outras advêm da interpretação incorreta dos requisitos do sistema a ser desenvolvido. Requisitos incompletos, incorretos, ou mal

entendidos, são as causas mais freqüentes da baixa qualidade, ultrapassagem dos custos previstos e atraso na entrega do *software*.

Em princípio, a especificação de requisitos do usuário deve ser completa e consistente. A completeza significa que todas as funções requeridas pelo usuário devem estar definidas. A consistência significa que os requisitos não devem ter definições contraditórias. Algumas das razões são por causa da complexidade inerente ao sistema. Essas inconsistências podem não ser óbvias quando é feita a primeira versão da especificação. Os problemas somente emergem depois de uma análise mais profunda. À medida que os problemas são descobertos durante as revisões, eles devem ser corrigidos no documento de requisitos.

1.5.2 Requisitos Não-Funcionais

Os requisitos não-funcionais, como o nome sugere, são aqueles que não dizem respeito diretamente às funções específicas do sistema. Em geral, eles estão relacionados aos requisitos de qualidade do sistema como confiabilidade, tempo de resposta e espaço em disco. Como conseqüência, eles podem definir restrições para o sistema, como a capacidade dos dispositivos de E/S (entrada/saída) e as representações de dados utilizadas nas interfaces de sistema. Muitos requisitos não-funcionais dizem respeito ao sistema como um todo, e não a características individuais do sistema. Enquanto a falha em cumprir com um requisito funcional individual pode degradar o sistema, a falha em cumprir um requisito não-funcional pode tornar todo os sistema inútil. Por exemplo, se um sistema de aviação não atender a seus requisitos de confiabilidade, ele não será atestado como seguro para operação; se um sistema de controle em tempo real falhar em cumprir com seus requisitos de desempenho, as funções de controle não operarão corretamente, comprometendo o sistema como um todo.

Os requisitos não-funcionais surgem conforme a necessidade dos usuários, em razão de restrições de orçamento, de políticas organizacionais, pela necessidade de interoperabilidade com outros sistemas de *software* ou *hardware* ou devido a fatores externos, como por exemplo, regulamentos de segurança e legislação sobre privacidade [Mac99]. Os diferentes tipos de requisitos não-funcionais podem ser classificados de acordo com a sua procedência [Som04]:

- Requisitos de produtos

São os requisitos que especificam o comportamento do produto. Entre os exemplos estão os requisitos de desempenho, que indicam sobre com que rapidez o sistema deve operar e quanta memória ele requer, os requisitos de confiabilidade, que estabelecem a taxa aceitável de falhas, os requisitos de portabilidade que permitem que o sistema funcione em qualquer sistema operacional e os requisitos de facilidade de uso que fornecem mecanismos para facilitar o uso do sistema.

- Requisitos organizacionais

São procedentes de políticas e procedimentos nas organizações do cliente e do desenvolvedor. Entre os exemplos estão os padrões de processo que devem ser utilizados, os requisitos de implementação, como a linguagem de programação ou o método de projeto utilizado, e os requisitos de fornecimento, que especificam quando o produto e seus documentos devem ser entregues.

- Requisitos externos

Abrange todos os requisitos procedentes de fatores externos ao sistema e a seu processo de desenvolvimento. Dentre eles destacam-se os requisitos de interoperabilidade, que definem como o sistema interage com sistemas em outras organizações, os requisitos legais, que devem ser seguidos para assegurar que o sistema opera de acordo com a lei, e os requisitos éticos. Os requisitos éticos são definidos em um sistema para garantir que este será aceitável para seus usuários e o público em geral.

Um problema comum com os requisitos não-funcionais é que eles são, às vezes, de difícil verificação; eles podem ser escritos, por exemplo, para refletir as necessidades do cliente, como a facilidade de uso, a habilidade do sistema se recuperar de uma falha ou a rapidez de resposta ao usuário. A não definição quantitativa desses requisitos causa problemas para os desenvolvedores de sistema, à medida que eles deixam o enfoque aberto à interpretação e à conseqüente discussão quando o sistema é finalizado. De maneira ideal, os requisitos não-funcionais devem ser expressos quantitativamente, utilizando-se medidas que possam ser objetivamente testadas.

A Tabela 1 apresenta uma adaptação da tabela apresentada por Sommerville [Som04] que mostra uma série de possíveis medidas, que podem ser utilizadas para especificar requisitos não-funcionais de sistemas. Durante a extração de requisitos deve-se quantificar esses requisitos e posteriormente as medições podem ser feitas durante o teste de sistema, para determinar se o sistema cumpre com esses requisitos ou não.

Tabela 1 - Medidas para especificar requisitos não-funcionais

Requisitos Não-Funcionais	**Medidas**
Desempenho	Transações processadas por minuto
	Tempo de resposta ao usuário por evento
	Tempo de *refresh* da tela
Tamanho	K bytes
	Número de chips de RAM
Facilidade de uso	Número de telas
	Número de *frames* de ajuda
Confiabilidade	Tempo médio para falhar
	Probabilidade de indisponibilidade
	Taxa de ocorrência de falhas
	Disponibilidade
Robustez	Tempo de reinício depois de uma falha
	Porcentagem de eventos que causam falhas
Portabilidade	Porcentagem de declarações dependentes de sistema-alvo
	Número de sistemas-alvo

Na prática, em geral, a especificação quantitativa de requisitos é difícil. Os usuários podem não ser capazes de traduzir suas metas em requisitos quantitativos; para alguns requisitos, como os requisitos de facilidade de manutenção o custo pode ser muito alto. Portanto, os documentos de requisitos, muitas vezes, incluem declarações de metas misturadas com requisitos. Essas metas podem ser úteis para os desenvolvedores porque fornecem algumas pistas sobre as prioridades do usuário. Contudo, os usuários devem saber que essas metas estão sujeitas a más interpretações e não podem ser objetivamente verificadas.

Em princípio, os requisitos funcionais e não-funcionais devem ser diferenciados em um documento de requisitos; na prática, isso é difícil. É preciso encontrar um equilíbrio adequado e isso depende do tipo de sistema que está sendo especificado. Contudo, requisitos claramente relacionados às restrições do sistema devem ser explicitamente destacados. Isso pode ser feito colocando-os em uma seção separada do documento de requisitos funcionais. Essa separação facilita o entendimento dos requisitos, possibilitando aos arquitetos o foco na solução de uma arquitetura que atenda os requisitos não-funcionais.

1.5.3 Arquitetura X Requisitos Não-Funcionais

A escolha da arquitetura é muito importante para um sistema, pois ela depende diretamente dos requisitos não-funcionais que foram especificados pela equipe de requisitos. Um exemplo simples de erro na escolha da arquitetura seria se um sistema precisasse de portabilidade, ou seja, que o sistema rodasse em qualquer sistema operacional e o arquiteto decidisse por utilizar a tecnologia. O problema de se usar.Net é que o sistema apenas roda no sistema operacional da Microsoft, não atendendo os requisitos não-funcionais do sistema. Devido a esse problema, se faz necessário que o arquiteto estude detalhadamente os requisitos não-funcionais do sistema. Uma forma de facilitar o entendimento dos requisitos não-funcionais e aplicá-las na arquitetura do sistema é através do conceito de visões que consiste em visualizar sistemas sob várias perspectivas. Diferentes participantes como usuários finais, analistas, desenvolvedores, gerentes de projetos e arquitetos trazem contribuições próprias ao projeto e observa o sistema de maneira distinta em momentos diferentes ao longo do desenvolvimento do projeto. A arquitetura do sistema talvez seja o artefato mais importante a ser utilizado com o objetivo de gerenciar esses diferentes pontos de vista e, assim, tornar possível o controle do desenvolvimento de um sistema durante seu ciclo de vida. As visões a serem utilizadas variam de acordo com as características de cada projeto. Para descrever a arquitetura do Estudo de Caso *(*que será visto no capítulo 4), serão utilizadas as seguintes visões:

- Visão de casos de uso – visão que descreve o sistema do ponto de vista de requisitos funcionais. Nessa visão são identificados os casos de uso do sistema mais relevantes e os atores que acionam esses casos de uso.
- Visão lógica – descreve o sistema em termos de subsistemas e, para cada subsistema, os pacotes mais relevantes do ponto de vista arquitetural.
- Visão de implementação - descreve a organização física do *software*, por exemplo, executáveis, bibliotecas e código fonte.

A especificação dos requisitos implícitos e explícitos [Pre87] é muito importante para o sucesso do desenvolvimento do *software*, pois possibilita a visibilidade do problema. Em especial, para o projeto da arquitetura, a especificação dos requisitos direciona a busca por soluções que permite aplicá-las no desenvolvimento de *software* com o intuito de satisfazer da melhor forma todos os requisitos solicitados pelo cliente.

1.6 Organização da dissertação

Esse trabalho está organizado da forma que o capítulo 1 apresenta uma introdução sobre o assunto e os trabalhos relacionados com esse estudo. Já no capítulo 2 serão apresentados conceitos básicos necessários para o entendimento das arquiteturas. No capítulo 3 serão mostradas algumas tecnologias para implementação de arquiteturas.

No capítulo 4, será apresentado um caso de estudo contendo a descrição do sistema, sua arquitetura, problemas de desempenho encontrados e a solução analisada para que o sistema atenda os requisitos solicitados pelo usuário e, finalmente, o capítulo 5 apresentará a conclusão da análise e a sugestão de trabalhos futuros.

2 Conceitos de Arquitetura de *Software*

Nesse capítulo serão apresentadas abordagens de Sommerville[Som04] e de outros autores contendo conceitos importantes para a compreensão da arquitetura de *software* como decomposição, estrutura, modelos e padrões arquiteturais e de projeto. Além desses conceitos, serão apresentados processos para avaliar a arquitetura de *software*. Será mostrada inicialmente, uma abordagem mais genérica desses conceitos segundo Sommerville (seções 2.1 a 2.4), posteriormente as abordagens de outros autores relativos aos conceitos de padrões arquiteturais e de projeto (seção 2.5) e finalmente serão apresentados os processos para avaliar a arquitetura (seção 2.6) segundo Tesoriero [Tes02] e Kazman [Kaz00].

2.1 Arquitetura de Software

De acordo com Booch [Boo99] e Kazman [Bas98], uma arquitetura é um conjunto de decisões significativas sobre a organização de um sistema de *software*, a seleção de requisitos não-funcionais (citados no capítulo 1 seção 1.5) e suas interfaces, assim como o relacionamento entre estes requisitos. Para Sommerville [Som04], a arquitetura de *software* é o primeiro artefato que pode ser analisado para determinar como os requisitos de qualidade como concorrência, portabilidade, usabilidade e segurança são aplicados da forma mais adequada ao sistema.

A palavra arquitetura está associada à arte e ciência de projetar e construir o *software*, levando em consideração métodos, ferramentas e padronizações utilizadas no seu projeto e na sua construção. A Arquitetura de *Software* integra a disciplina Engenharia de *Software* e tem sido o centro das atenções de vários estudos e pesquisas, devido à complexidade e importância que os *softwares* alcançaram.

A arquitetura se preocupa em estabelecer uma estrutura básica para um sistema identificando os seus componentes arquiteturais principais e os seus conectores responsáveis pela comunicação entre esses componentes. Componentes têm interfaces bem definidas que permitem

a sua iteração com outros componentes. Na Figura 1, têm-se dois componentes A e B. Componentes que executam tarefas específicas facilitam uma partição natural do *software* em unidades coesivas, além de garantir maior flexibilidade às aplicações que os utilizam.

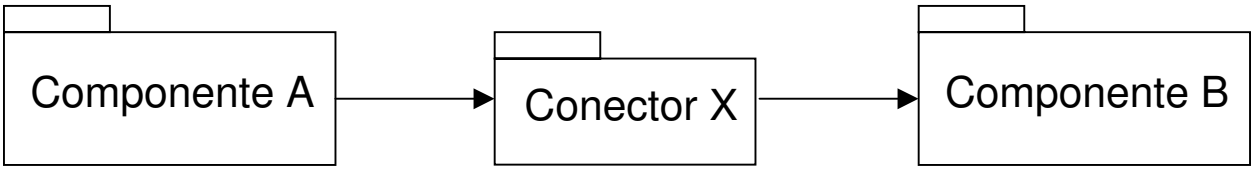

Figura 1 – Mostra a conexão dos subsistemas A e B através de um conector X.

Os componentes interagem em maneiras distintas e correspondem aproximadamente às unidades de compilação de linguagens de programação convencionais e de outros objetos no nível de usuário tais como arquivos. Os conectores mediam interações entre componentes, isto é, estabelecem as regras que governam a interação dos componentes e especificam todo mecanismo auxiliar de execução requerido. Os conectores no geral, não correspondem individualmente às unidades de compilação; eles manifestam-se como dados de entradas, estruturas de dados dinâmicas, chamadas de sistema, parâmetros de iniciação e servidores que suportam múltiplas conexões independentes.

Para a arquitetura não importa se o componente de *software* é um processo, um objeto, uma biblioteca, um banco de dados ou um produto comercial. O importante é atender aos requisitos não-funcionais do sistema de forma eficaz [Gar99]. Como ideal, os requisitos não-funcionais discutidos na seção 1.5 devem ser expressos quantitativamente utilizando-se de métricas, conforme foi apresentado na Tabela 1, que possam ser objetivamente testadas. Nesta seção serão mostrados os conceitos básicos para se entender as arquiteturas.

2.2 Decomposição

A idéia da decomposição é a de dividir um sistema em partes menores para facilitar o gerenciamento de sua complexidade. A decomposição ajuda a definir e tornar mais fácil o entendimento, as interfaces entre diferentes partes do sistema. É muito útil também em situações

onde se deva integrar aplicações desenvolvidas por terceiros. Outra vantagem da decomposição é a de facilitar o desenvolvimento do *software* por uma equipe grande [Som04].

Considera-se dois modelos que podem ser utilizados na decomposição de um subsistema em módulos:

- **Modelo orientado a objetos:** o sistema é decomposto em um conjunto de objetos que se comunicam.
- **Modelo de fluxo de dados:** o sistema é decomposto em módulos funcionais que aceitam a entrada de dados e os transforma, de alguma maneira, em dados de saída. Isso é também chamado de abordagem de *pipeline*.

No modelo orientado a objetos, os módulos são objetos com estado privativo e operações definidas nesse estado. No modelo de fluxo de dados, os módulos são transformações funcionais. Em ambos os casos, os módulos podem ser implementados como componentes seqüenciais, pois são mais fáceis de projetar, implementar, verificar e testar do que os sistemas paralelos. É recomendado decompor sistemas em módulos e, então, decidir durante a implementação se eles devem ser executados em seqüência ou em paralelo. A Figura 2 é um simples exemplo de decomposição em vários subsistemas, pois conforme mostra a figura, o sistema Banco Online é composto pelos subsistemas de Faturamento, Relatório e Administração.

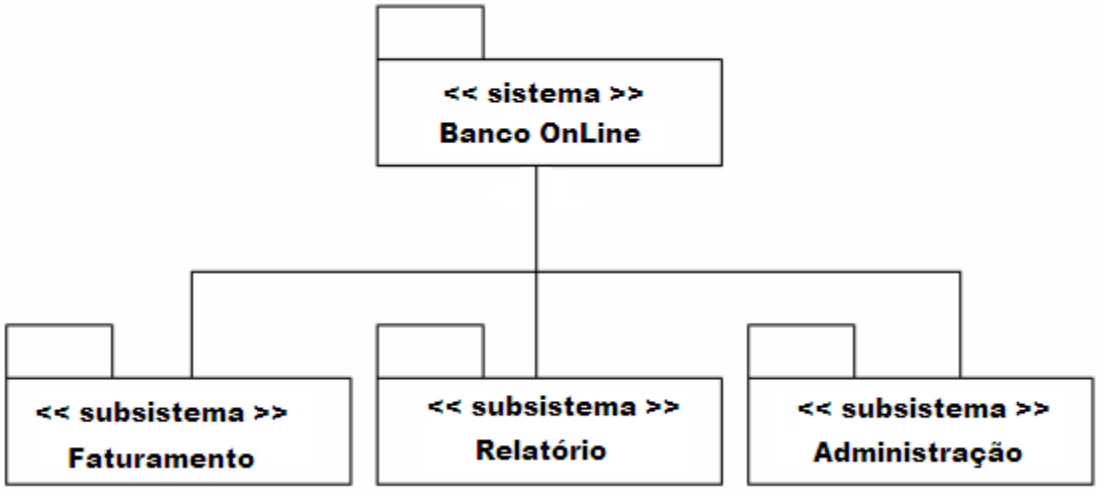

Figura 2 – Decomposição em vários subsistemas

2.3 Estruturação do Sistema e Modelos Associados

Segundo Sommerville [Som04] projeto de arquitetura pode ser representado como um diagrama de blocos, em que cada bloco do diagrama representa um subsistema. Um diagrama de blocos de arquitetura apresenta uma visão geral da estrutura do sistema. De modo geral, ele é compreensível a vários arquitetos e interessados que estejam envolvidos no processo de desenvolvimento de sistema. A Figura 3 é um exemplo de modelo estrutural de arquitetura na visão lógica onde os conectores existentes entre os componentes são representados por setas de acordo com o formato de representação proposto por Sommerville [Som04]. Blocos dentro de blocos indicam que o subsistema foi, por sua vez, decomposto em subsistemas conforme mostra a Figura 3 onde o Sistema X foi decomposto nos subsistemas D e E. As setas significam que dados e/ou controles são passados de subsistema para subsistema, na direção das setas.

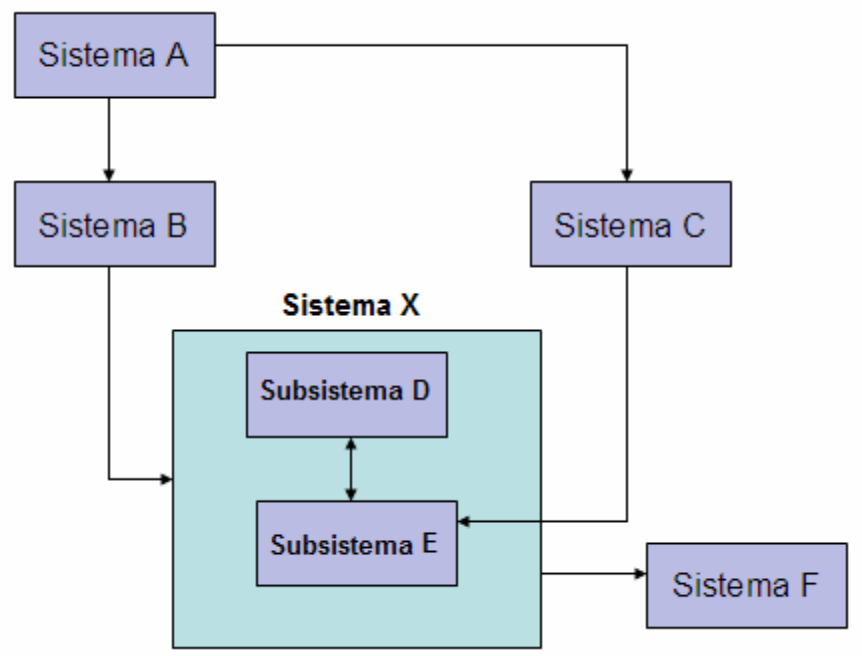

Figura 3 – Modelo estrutural de arquitetura na visão lógica

Os diagramas de blocos e linhas certamente não devem ser as únicas representações de arquitetura a serem utilizadas; contudo, pode-se dizer que elas fazem parte de uma série de modelos úteis de arquitetura.

Segundo Sommerville[Som04], modelos mais específicos da estrutura podem ser desenvolvidos, os quais mostram como os subsistemas compartilham dados, como estão distribuídos e como atuam como interface entre si. São discutidos, nesta seção, os modelos padrão de arquitetura de sistemas: modelo de repositório, modelo em camadas que engloba as arquiteturas de cliente-servidor e de objetos distribuídos.

2.3.1 Modelo de Repositório

Os subsistemas que constituem um sistema precisam trocar informações, para que possam trabalhar em conjunto de modo eficaz [Som04]. Existem duas maneiras fundamentais para fazê-lo:

- Todos os dados compartilhados são mantidos em um banco de dados central, que pode ser acessado por todos os subsistemas. Um modelo de sistema com base em um banco de dados compartilhado é, algumas vezes, chamado de modelo de repositório.
- Cada subsistema mantém seu próprio banco de dados. Os dados são intercambiados com outros subsistemas, transmitindo mensagens para eles.

A maioria dos sistemas que utilizam grandes quantidades de dados são organizadas em um banco de dados compartilhado ou de um repositório de dados. Esse modelo é, portanto, adequado a aplicações em que os dados são gerados por um subsistema e utilizados por outro. São exemplos desse tipo de sistema de arquitetura, os sistemas de comando e controle, os sistemas de informações gerenciais, os sistemas de CAD (*computer aided design* - projeto auxiliado por computador) e os conjuntos de ferramentas CASE.

2.3.2 Modelo em Camadas

A motivação para uso de Arquitetura em Camadas consiste em uma evolução do modelo cliente servidor introduzindo múltiplos servidores de aplicação e bancos de dados. Podem-se distribuir componentes em servidores e acessar dados em diversos bancos de dados. Um *pool* de conexões de banco de dados pode ser adquirido na inicialização do servidor e então ser compartilhado entre clientes conforme necessário. A arquitetura contém uma estação de trabalho de cliente, um componente ou servidor de aplicação e um ou mais servidores de banco de dados. A interface com o usuário está no lado do cliente, enquanto a lógica de negócios e o

gerenciamento de dados residem nos níveis dedicados. A lógica de negócios reside em um servidor de aplicação, onde pode ser facilmente gerenciado. Um ambiente de várias camadas requer um planejamento cuidadoso das interfaces de componente para permitir a reutilização e o compartilhamento. A descrição de cada camada é descrita a seguir:

- **Camada cliente** para a interação com o usuário. A camada cliente reside na máquina do usuário e existe um navegador responsável pela exibição dessas telas.
- **Camada de apresentação** reside em um servidor, sendo responsável por processar as requisições da camada cliente e repassar essas requisições para a camada de negócios
- **Camada de negócio** *(middleware)* implementa a lógica de negócio da aplicação.
- **Camada de dados** é composta por Sistema de Gerenciamento de Banco de Dados e aplicações legadas que são responsáveis por armazenar e fornecer as informações utilizadas pelo sistema.

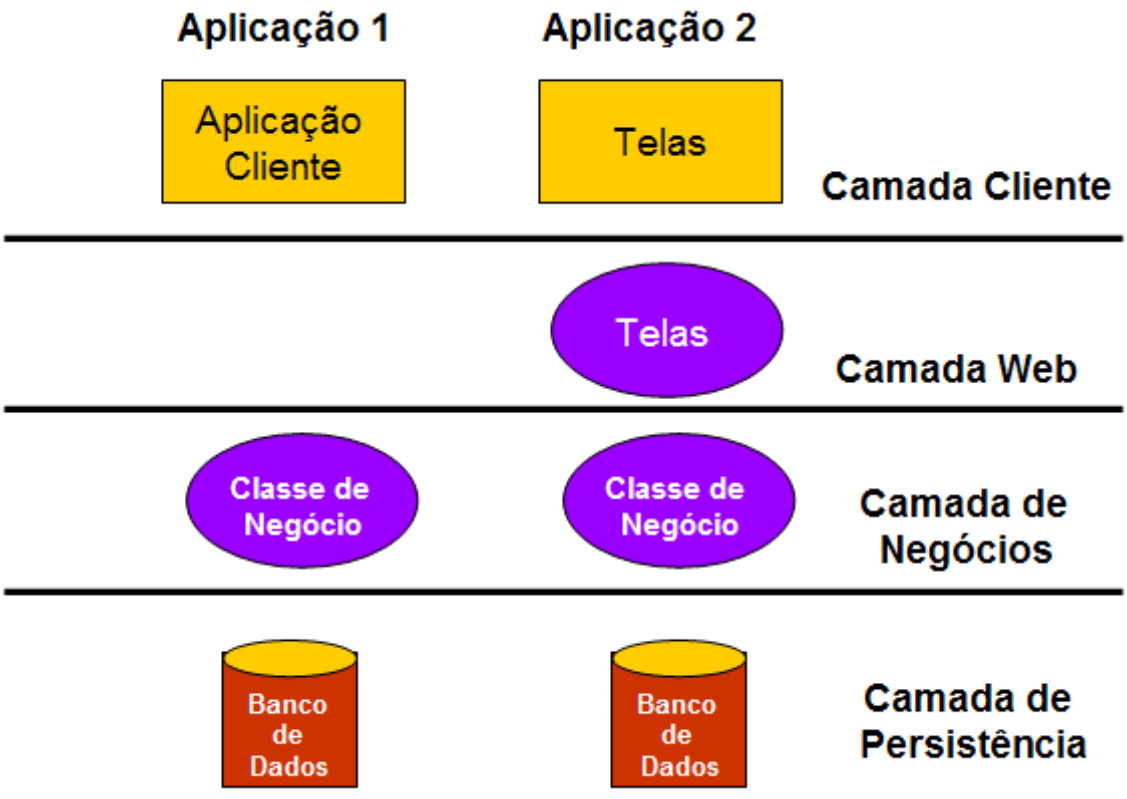

Figura 4 – Modelo em Camadas Genérica

Pode-se considerar também 2 tipos de camadas para a arquitetura de *software*: camada lógica e camada física [Som04].

Camada Lógica é utilizado para decomposição lógica do sistema em subsistemas e módulos. As camadas agrupam e separam subsistemas, de modo a restringir quem pode usar os subsistemas, componentes e módulos. Elas criam separações de interesses no *software*, abstraindo tipos específicos de funcionalidade em camadas funcionais e fornecendo uma fronteira conceitual entre os conjuntos de serviços.

Camada física foi concebida visando distribuição de *software* em processadores múltiplos, separados. Processos podem ser fisicamente distribuídos em processadores múltiplos. Em uma visão física, a máquina cliente pode conter a camada cliente, o servidor de aplicação pode conter as camadas web e de negócio e o servidor de banco de dados pode conter a camada de persistência. Essas camadas são desejáveis do ponto de vista a melhorar o fluxo, disponibilidade ou funcionalidade do sistema, melhorando o poder de processamento.

O desafio para os arquitetos de sistemas está em projetar *software* e *hardware* para que eles forneçam características desejáveis dos sistemas e, ao mesmo tempo, minimizem os problemas inerentes a esses sistemas. Para fazer isso, é preciso que eles compreendam as diferentes arquiteturas de sistemas em camadas. Neste texto, são analisados dois tipos genéricos de arquitetura de sistemas em camadas: arquitetura cliente-servidor e a arquitetura de objetos distribuídos.

2.3.2.1 Arquitetura Cliente-Servidor

O modelo de arquitetura de cliente-servidor foi concebido para ser um modelo de sistema distribuído, que mostra como os dados [Som04] e o processamento são distribuídos em uma série de processadores. Os componentes principais desse modelo são:

- Um conjunto de servidores em *stand-alone*, que oferece serviços a outros sistemas. Podemos citar como exemplo os servidores de impressoras, que oferecem serviços de impressão; os servidores de arquivo, que oferecem serviços de gerenciamento de

arquivos, e os servidores de compilação, que oferecem serviços de compilação de linguagem de programação.

- Um conjunto de clientes que solicita os serviços oferecidos pelos servidores. Estes são, normalmente, subsistemas em si.
- Uma rede que permite aos clientes acessar esses serviços. Em princípio, isso não é realmente necessário, uma vez que tanto clientes quanto servidores poderiam ser executados em uma única máquina. Normalmente, os clientes precisam saber quais servidores estão disponíveis e quais serviços que eles fornecem. Contudo, os servidores não precisam saber a identidade de clientes ou quantos clientes existem. Os clientes têm acesso aos serviços fornecidos por um servidor por meio de chamadas remotas.

A arquitetura cliente-servidor mais simples é chamada de arquitetura cliente-servidor de duas camadas, em que uma aplicação é organizada como um servidor (ou vários servidores idênticos) e um conjunto de clientes. O problema essencial com a abordagem cliente-servidor de duas camadas é que podem ocorrer problemas com o desempenho ou problemas de gerenciamento de sistema dependendo do modelo escolhido. Para evitar esses problemas, uma abordagem alternativa é utilizar uma arquitetura cliente-servidor de três camadas. Nessa arquitetura, a apresentação, o processamento de aplicações e o gerenciamento de dados são processos logicamente separados.

Uma arquitetura de *software* cliente-servidor de três camadas não significa necessariamente que haja três sistemas de computador conectados à rede. Um único servidor pode executar tanto o processamento de aplicações quanto o gerenciamento de dados de aplicações, como servidores lógicos separados. Contudo, se a necessidade aumentar, será relativamente simples separar o processamento de aplicações e o gerenciamento de dados e executá-los em processadores separados.

2.3.2.2 Arquitetura de objetos distribuídos

Uma abordagem mais geral para o projeto de sistemas distribuídos é eliminar a distinção entre cliente e servidor e projetar a arquitetura de sistema como uma arquitetura de objetos

distribuídos. Em uma arquitetura de objetos distribuídos, os componentes fundamentais do sistema são objetos que oferecem uma interface para um conjunto de serviços que eles fornecem. Outros objetos solicitam esses serviços sem fazer nenhuma distinção lógica entre um cliente (quem recebe o serviço) e um servidor (o provedor do serviço). As vantagens desse modelo de arquitetura de sistema distribuído são:

- Ele permite ao arquiteto de sistema adiar decisões sobre onde ou como os serviços devem ser fornecidos. Os objetos provedores de serviços podem ser executados em qualquer nó da rede. Portanto, a distinção entre os modelos cliente-gordo e cliente-magro [Som04] torna-se irrelevante, uma vez que não existe nenhuma necessidade de decidir antecipadamente onde os objetos de lógica de aplicações estão localizados.
- O sistema é flexível e extensível. Diferentes instâncias do sistema, com o mesmo serviço fornecido por diferentes objetos ou por objetos duplicados, podem ser criadas para lidar com diferentes cargas do sistema. Novos objetos podem ser acrescentados, à medida que a carga do sistema aumenta, sem interferir em outros objetos do sistema.
- É possível reconfigurar o sistema dinamicamente com objetos que migrem pela rede, conforme necessário. Essa opção pode ser importante, quando existem padrões variáveis de demanda por serviços. Um objeto provedor de serviços pode migrar para o mesmo processador que os objetos solicitantes de serviços, melhorando, assim, o desempenho do sistema.

2.4 Modelos de Controle

Os modelos para estruturar um sistema se ocupam de como um sistema é decomposto em subsistemas. Para trabalhar como um sistema, os subsistemas devem ser controlados de modo que seus serviços sejam fornecidos no local certo e no tempo certo [Som04]. Os modelos estruturais não incluem, e nem devem, informações de controle. Em vez disso, o arquiteto deve organizar os subsistemas de acordo com algum modelo de controle que complemente o modelo estrutural utilizado. Os modelos de controle em nível de arquitetura dizem respeito ao controle de fluxo entre subsistemas.

Duas abordagens gerais de controle podem ser identificadas:

- **Controle centralizado:** um subsistema tem a responsabilidade geral pelo controle e inicia e interrompe os outros subsistemas. Ele pode também devolver o controle para outro subsistema, mas aguardará que essa responsabilidade pelo controle seja devolvida a ele.
- **Controle baseado em eventos:** Em vez de ter as informações de controle embutidas em um subsistema, cada subsistema pode responder a eventos gerados externamente. Esses eventos podem provir de outros subsistemas ou do ambiente do sistema.

Os modelos de controle complementam os modelos estruturais. Todos os modelos estruturais podem ser realizados com o uso do controle centralizado ou do controle baseado em eventos. Existem alguns mecanismos que podem auxiliar os arquitetos no mapeamento do melhor modelo arquitetural a ser utilizado. Estes mecanismos serão descritos na próxima seção.

2.5 Qualidade na Arquitetura através de padrões

Padrões são soluções de eficiência já comprovadas e amplamente utilizadas para a resolução de problemas comuns em projetos de *software*. Estas soluções são desenvolvidas e conhecidas por arquitetos e tornam-se padrões por serem reutilizadas várias vezes em vários projetos e por terem eficácia comprovada [Ale78]. Esses padrões são úteis porque os usando pode-se reduzir o risco total de falha do sistema devido a tipos específicos de erros, podem ajudar a resolver problemas difíceis encontrados em situações similares e ainda facilitam e melhoram a comunicação dentro da equipe [Alu02].

Garlan e Shaw [Gar93] descrevem diversos padrões de *design* comuns para arquiteturas que oferecem oportunidades ricas tanto para a elaboração quanto para a estruturação do sistema. Estes padrões idiomáticos diferem em quatro principais aspectos: a idéia por trás do padrão ou o modelo de sistema; os tipos de componentes que são usados no desenvolvimento de um sistema de acordo com o padrão; os conectores, ou tipos de interações entre os componentes; e a estrutura de controle ou disciplina de execução. Usando o mesmo esquema descritivo para os padrões, pode-se identificar mais facilmente diferenças significativas entre eles. Uma vez que o padrão informal está claro, os detalhes podem ser formalizados [All94]. A descrição de cada padrão inclui as seguintes informações:

• **Problema:** O problema ao qual o padrão se dirige. Isto é, que características de requisitos da aplicação conduzem o arquiteto a selecionar este padrão?

• **Contexto:** Que aspectos do ambiente computacional levam o arquiteto ao uso deste padrão?

• **Solução:** O modelo de sistema capturado pelo padrão, junto com os componentes, conectores, e a estrutura de controle que completam o padrão.

Nessa seção são apresentados alguns padrões arquiteturais e padrões de design relacionados à arquitetura.

2.5.1 Padrões Arquiteturais

Um padrão arquitetural é baseado em tipos selecionados de componentes e de conectores, juntos com uma estrutura do controle que governa a execução. Um modelo de sistema completo trata como eles são integrados [Sha95]. Tem-se que padrões são alguns dos principais tipos de abstrações arquiteturais. A finalidade de cada um destes padrões é: impor uma estrutura completa para um sistema ou subsistema de *software* que seja apropriado ao problema que o sistema ou o subsistema está resolvendo; esclarecer as intenções do arquiteto sobre a organização do sistema ou do subsistema; fornecer um paradigma que ajude a estabelecer e manter uma consistência interna; permitir uma verificação e análise apropriada preservando informações sobre a estrutura para referência durante uma manutenção posterior.

Na prática, um arquiteto adota um ou mais destes padrões para dar forma ao projeto. Os padrões podem ser usados em combinação, de forma a fornecer visões complementares durante o projeto inicial como um repositório e um interpretador [Gar93] ou elaborando um componente de um padrão usando abstrações de dados. Esta elaboração progressiva pode ser continuada repetidamente até que o problema arquitetural seja resolvido, ponto no qual técnicas de programação convencionais são necessárias [Sha96].

2.5.1.1 Padrão Arquitetural de Abstração de dados

Problema

Este padrão é apropriado para as aplicações nas quais o problema central é identificar e proteger informações, especialmente informações representativas [Boo86]. Quando os dados são decompostos obtendo a sua estrutura natural e a sua essência, os componentes podem então encapsular os dados, as operações essenciais nos dados, e as integridades dos dados e das operações conforme mostra a Figura 5.

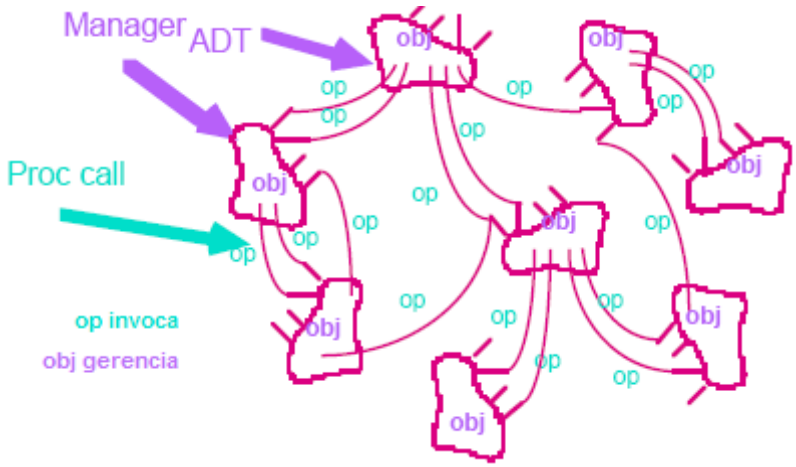

Figura 5 – Padrão Abstração de Dados

Contexto

Vários métodos de projeto fornecem estratégias para identificar objetos naturais. Linguagens de programação atuais suportam várias variações deste padrão, então se a escolha da linguagem ou da metodologia forem fixas, isto vai influenciar fortemente como a decomposição será feita.

Solução

O modelo de sistema significa a manutenção de estado, os componentes são gerenciadores (por exemplo, servidores, objetos, tipos abstratos de dados), os conectores são chamadas de procedimentos e a estrutura de controle é descentralizada. Exemplos do uso desta arquitetura podem ser vistos nas referências [GoF95], [Lin85] e [Par72].

2.5.1.2 Padrão Arquitetural de Comunicação de Processos

Problema

Este é apropriado para aplicações que envolvem uma coleção de processos distintos e altamente independentes cuja execução deva prosseguir independentemente [And91]. Os processos envolvem coordenação de dados ou controle em um dado período de tempo. Como conseqüência, a exatidão do sistema requer atenção à distribuição e à sincronização das mensagens ilustrado na Figura 6. Embora muitos outros padrões possam ser implementados com transferência de mensagens, o padrão de transferência de mensagem pode ser utilizado quando a abstração envolve comunicação.

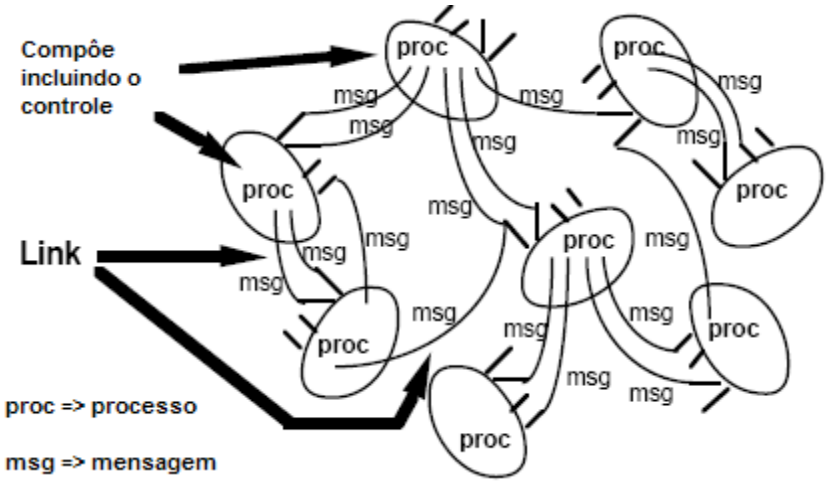

Figura 6 – Padrão Arquitetural de Comunicação de Processos

Contexto

A seleção de uma estratégia de comunicação é ditada freqüentemente pelo suporte de comunicação fornecido pelo sistema operacional.

Solução

O modelo de sistema é representado pelos processos independentes de comunicação. Os componentes são representados pelos processos que enviam e recebem mensagens de/para receptores explicitamente selecionados. Os conectores são as mensagens discretas (sem informação compartilhada) com parceiros de comunicação conhecidos e a estrutura de controle é

representada por cada processo que tem sua própria *thread* de controle, que pode tanto suspender ou continuar pontos de comunicação. Os exemplos dessa solução podem ser vistas nas referências [And91] e [Pau85].

2.5.1.3 Padrão Arquitetural de Repositório

Problema

Este padrão é apropriado para aplicações em que o problema principal é estabelecer, escalar, e manter uma estrutura central de informação complexa [Amb90] conforme mostra a Figura 7. Tipicamente a informação deve ser manipulada em uma grande variedade de maneiras. Freqüentemente, uma persistência em longo prazo também pode ser requerida. Diferentes variações suportam estratégias radicalmente diferentes de controle.

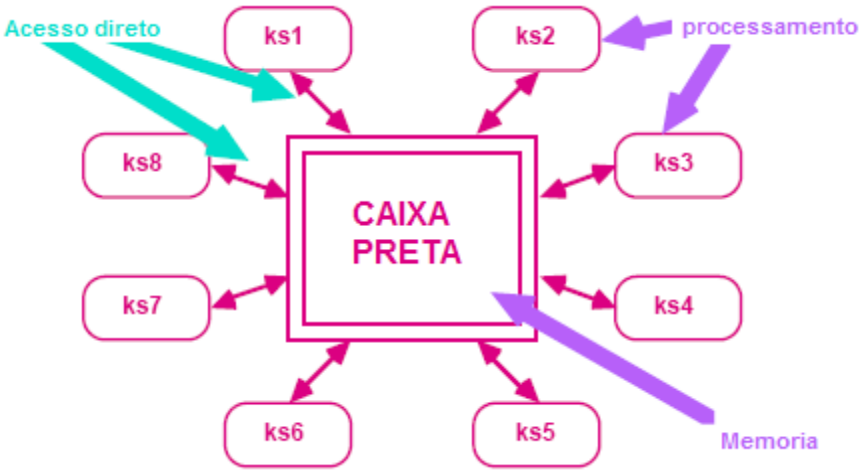

Figura 7 – Padrão Arquitetural de Repositório

Contexto

Os repositórios freqüentemente requerem um suporte considerável, seja por um grande sistema em tempo de execução (tal como uma base de dados) ou um framework ou gerador para processar as definições de dados.

Solução

O modelo de sistema é representado através de dados centralizados, usualmente fortemente estruturados. O componente é representado por uma memória contendo vários processos puramente computacionais e os conectores são unidades computacionais interagem com a memória por acesso direto ou chamada de procedimento. Exemplos dessa solução podem ser vistas nas referências [Amb90] [Nii86] [Bar84].

Os padrões que serão usados no caso de estudo dessa dissertação estão relacionados nas próximas páginas.

2.5.1.4 Padrão Arquitetural de Camadas

Problema

Este padrão é apropriado para aplicações que envolvem classes de serviços distintas que podem ser agrupadas hierarquicamente. Freqüentemente há camadas para serviços básicos de sistema, para utilidades apropriadas a muitas aplicações, e para tarefas específicas da aplicação.

Contexto

Freqüentemente, cada classe de serviço é atribuída a uma camada e diversos padrões diferentes são usados para refinar as várias camadas [Bat91]. As camadas são apresentadas na Figura 8.

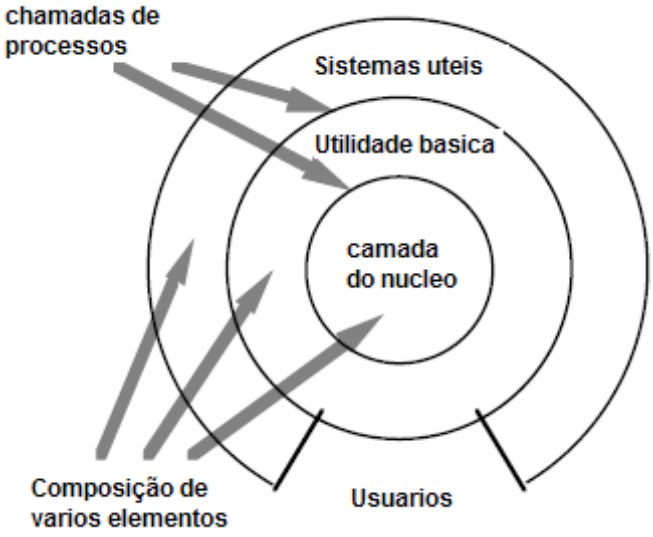

Figura 8 – Padrão Arquitetural de Camadas

Solução

O modelo de sistema é representado pela hierarquia de camadas. Os componentes são geralmente compostos, sendo que as composições são geralmente coleções de procedimentos. Os conectores dependem da estrutura dos componentes e são freqüentemente chamadas de procedimento sob visibilidade restrita, pode ser também cliente-servidor e a estrutura de controle representada por um processo único. Exemplos desta solução podem ser vistos nas referências [Bat91], [Fri85], [Lau79] e [Pau85].

2.5.1.5 Padrão Arquitetural Modelo-Visão-Controle

Problema

Este padrão é apropriado para minimizar o acoplamento entre objetos num sistema, alinhando-os com um conjunto específico de responsabilidades na área de persistência de dados e regras associadas (*Modelo*), apresentação (*Visão*), e a lógica da aplicação (*Controle*) também conhecido como arquitetura MVC (Modelo, Visão e Controle).

Contexto

Permite uma melhor estruturação do sistema, com uma clara separação das funcionalidades em entrada, processamento e saída. A arquitetura MVC ajuda a entender como tecnologias trabalham juntas conforme mostra a Figura 9.

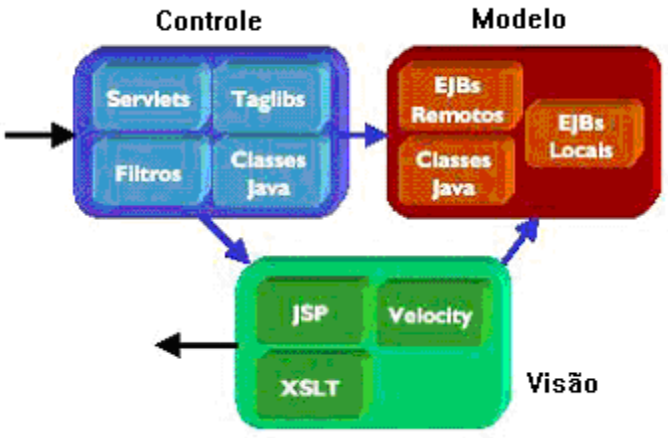

Figura 9 – Arquitetura MVC

O *Modelo* é responsável por manter o estado da aplicação e dos dados. Pode receber e responder a solicitações da camada de apresentação (*Visão*) e pode fornecer notificações para a camada de apresentação quando algo for mudado.

O *Controle* atualiza o *Modelo* baseado na execução da lógica da aplicação em resposta às ações executadas pelo usuário. Também é responsável por informar à camada de apresentação sobre o quê apresentar em resposta às requisições do usuário.

Solução

O modelo de sistema é representado pela composição do modelo, visão e controle. Cada componente é composto por um conjunto de classes. Os conectores dependem da estrutura dos componentes e são freqüentemente chamadas de procedimento.

2.5.2 Padrões de Projeto

Padrões de projeto (*design patterns*) são conhecidos na literatura como soluções para problemas recorrentes em um dado contexto, tendo como sua essência os elementos problema, solução e contexto [Gof95]. O objetivo dos padrões de projeto é identificar e documentar a solução central de um determinado problema. Essas documentações são úteis na solução de problemas comuns que surgem durante a engenharia de *software*. Abaixo estão relacionados alguns padrões de projeto [Alu02] úteis na solução de problemas de *software*:

2.5.2.1 ***Value Object***

Problema

O *Value Object* é referido como um objeto de dados e permite a comunicação das informações como um único conjunto de dados, melhorando a eficiência do sistema ao permitir a transmissão desses dados de uma única vez ao invés de realizar várias chamadas diferentes sendo muito usado pelos arquitetos que precisam de eficiência no sistema.

Contexto

A Figura 10 mostra um diagrama de classe que representa o padrão *Value Object* em sua forma mais simples. Como é mostrado nesse diagrama de classe, o objeto de dados é criado sob demanda pelo *enterprise bean,* ou seja, componente de negócio (que será detalhado no Capítulo 3) e retornado ao cliente remoto.

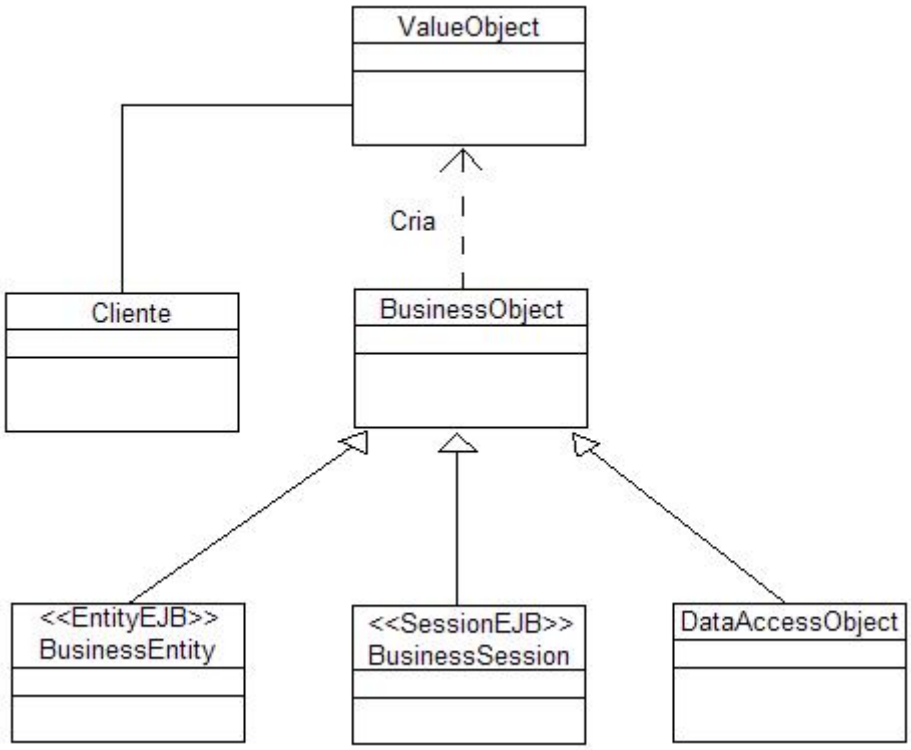

Figura 10 – Diagrama de classe do padrão *Value Object*

Solução

O modelo de sistema é representado pela hierarquia das classes. O componente é composto por um conjunto de classes conforme mostra a Figura 10. Os conectores são classes que se comunicam entre si através de chamadas de métodos.

2.5.2.2 *Business Delegate*

Problema

Permite separar, em um único ponto do sistema, o conhecimento que a camada de apresentação precisa ter a respeito da estrutura da camada de negócios, de forma que qualquer alteração em uma dessas camadas não afete a outra (basta fazer as alterações correspondentes nesse único ponto de interface entre as camadas).

Contexto

Dependendo da estratégia de implementação, o *Business Delegate* pode proteger clientes da possível volatilidade na implementação da classe dos serviços de negócios reduzindo potencialmente o numero de alterações que precisam ser feitas no sistema.

O *Business Delegate* permite de forma transparente executar qualquer operação necessária de tentativa ou recuperação caso haja um falha no serviço, sem expor o cliente ao problema ate que seja determinado que o problema não é solucionável. Outro beneficio é que o delegate pode colocar em memória resultados e referências para serviços de negócios remotos. Esse tipo de armazenamento em memória pode melhorar significativamente o desempenho, porque ele limita transmissões desnecessárias e potencialmente dispendiosas através da rede. A Figura 11 mostra o diagrama de classes representando o padrão *Business Delegate*. O cliente solicita que o *Business Delegate* proporcione acesso a serviços de negócios subjacentes. O *Business Delegate* utiliza um *Lookup Service* para localizar o componente do *BusinessService* solicitado.

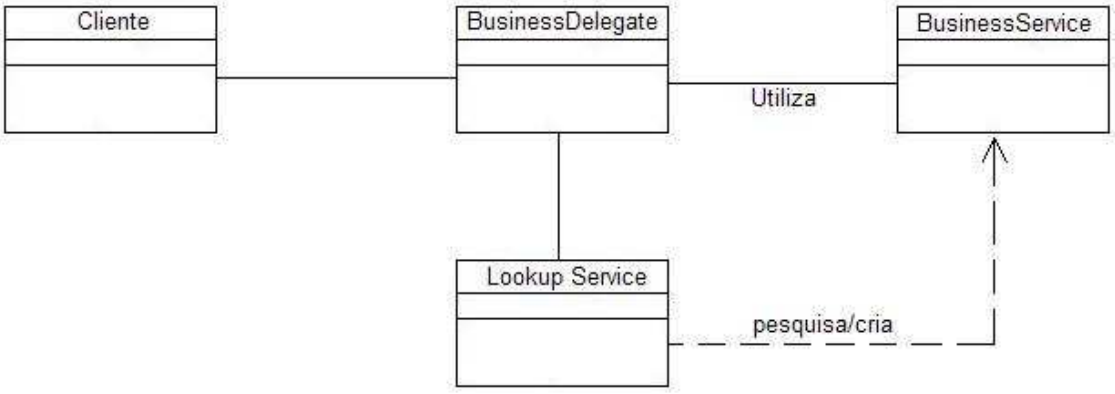

Figura 11 – Diagrama de classes do padrão *Business Delegate*

Soluçãoz

O modelo de sistema é representado pelo conjunto das classes. O componente são as próprias classes conforme mostra a Figura 11. Os conectores são classes que se comunicam entre si através de chamadas de métodos.

2.5.2.3 *Service Locator*

Problema

Encapsula a complexidade de busca e criação de serviços de negócios e localiza serviços de negócios. Vários clientes podem reutilizar o objeto *Service Locator* para reduzir a complexidade do código, fornecer um único ponto de controle e melhorar o desempenho do sistema fornecendo o recurso do armazenamento em memória.

Contexto

O *Service Locator* para componentes *enterprise bean* utiliza o objeto EJBHome. Quando o objeto EJBHome é obtido, pode ser armazenado em *cache* no *Service Locator* para utilização futura, evitando outra pesquisa de *Java Naming and Directory Interface* (JNDI) quando o cliente precisar do objeto local novamente. Dependendo da implantação, o objeto local pode ser retornado para o cliente, que pode, então, utilizá-lo para procurar, criar e remover *enterprise beans*. Por outro lado, o *Service Locator* pode manter (em *cache*) o objeto local e ganhar a responsabilidade adicional de fazer o *Proxy* de todas as chamadas de cliente para o objeto local. O diagrama de classes para a estratégia EJB *Service Locator* é mostrada da Figura 12.

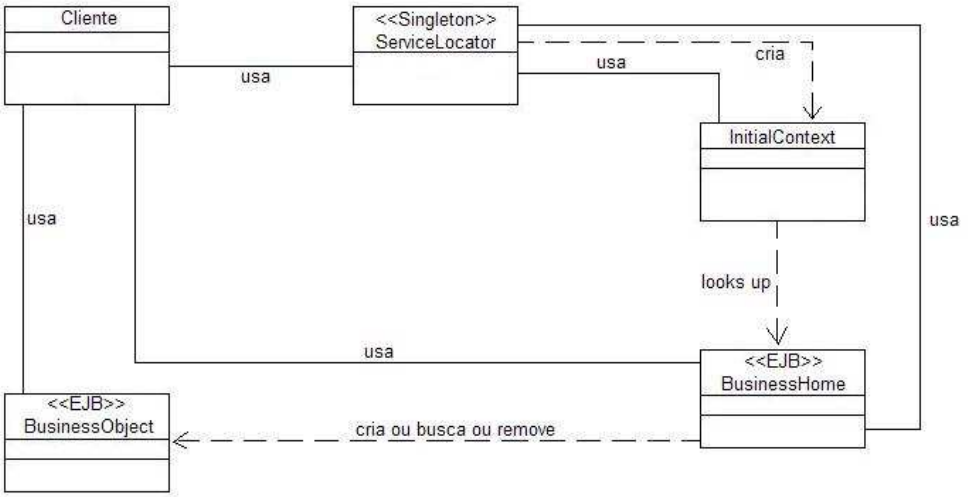

Figura 12 – Diagrama de classes da estratégia EJB *Service Locator*

Solução

O modelo de sistema é representado pela hierarquia das classes. O componente é composto pelas classes conforme mostra a Figura 12. Os conectores são classes que se comunicam entre si através de chamadas de métodos.

2.5.2.4 *Data Access Object* (DAO)

Problema

Permite a separação entre a lógica de processamento dos dados e o processo de persistência desses dados em banco de dados.

Contexto

Essencialmente, o DAO age como um adaptador entre o componente e a origem de dados. Esta origem pode ser um armazenamento persistente em um banco de dados, um repositório ou um serviço de negócios (*business service*) acessado através do CORBA *Internet Inter-ORB Protocol* (IIOP). O DAO oculta completamente os detalhes da implementação da origem de dados de seus clientes permitindo que o DAO se adapte a esquemas diferentes de armazenamento sem afetar seus clientes ou componentes de negócios. A Figura 13 mostra o diagrama de classes representando os relacionamentos para o padrão DAO.

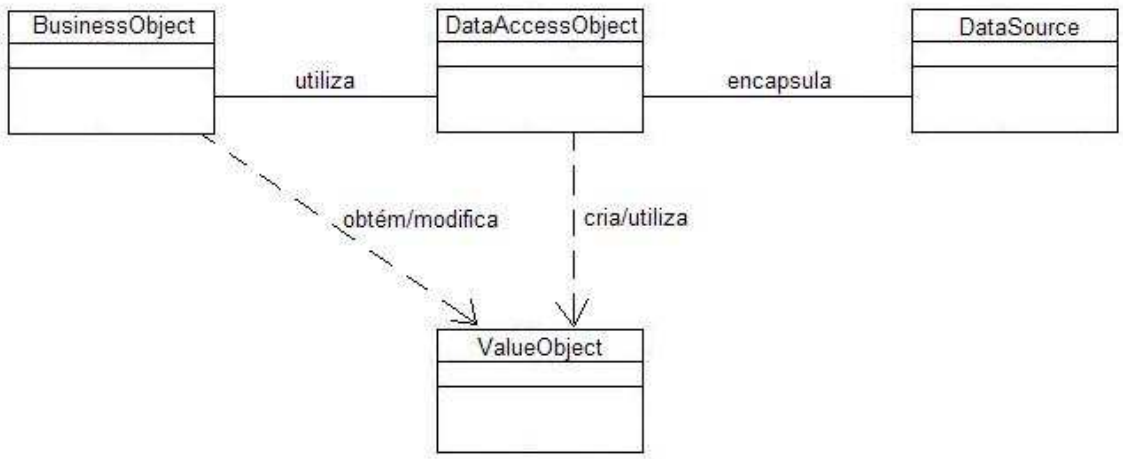

Figura 13 – *Data Access Object* (DAO)

Solução

O modelo de sistema é representado pela composição das classes. O componente é composto pelas classes conforme mostra a Figura 13. Os conectores são classes que se comunicam entre si através de chamadas de métodos.

2.6 Processos para avaliar arquitetura

O *software* é atualmente um dos maiores componentes do orçamento de muitas organizações. A maioria delas reconhece a importância de controlar os gastos com *software*, de analisar o desempenho dos resultados obtidos com o seu desenvolvimento e manutenção, a fim de permitir uma padronização. Para realizar isso, necessita-se fazer uso de medidas e de modelos apropriados. Estas medidas são necessárias para analisar qualidade e produtividade do processo de desenvolvimento e manutenção, bem como do produto de *software* construído. Medidas técnicas são necessárias para qualificar o desempenho técnico dos produtos do ponto de vista do desenvolvedor e do arquiteto. Por outro lado, medidas funcionais são necessárias para qualificar o desempenho dos produtos pela perspectiva do usuário. Estas medidas devem ser independentes das decisões do desenvolvimento técnico e da implementação. Tais medidas podem ser utilizadas para comparar a produtividade de diferentes técnicas e tecnologias.

Para avaliar a arquitetura de um sistema é necessário utilizar medidas técnicas, que são aplicadas através de um processo de avaliação. Entre os processos existentes, são mencionados nessa dissertação os processos de Tesoriero [Tes02] e Kazman [Kaz00].

2.6.1 Processo proposto por Tesoriero

O processo de avaliação será discutido descrevendo-se cada um de seus passos e as fontes de informações necessárias para executar a avaliação arquitetural. Uma equipe de análise que trabalha em conjunto com os desenvolvedores executa este processo. Esses passos do processo são listados na sua ordem básica. Entretanto, a equipe de análise e desenvolvimento pode se mover livremente entre os diferentes passos caso seja necessário.

A Figura 14 apresenta os componentes do processo de avaliação. A equipe de análise que executa o processo de avaliação deve ser tão objetiva quanto possível, devido ao fato de que a análise possa revelar falhas na arquitetura e na sua implementação. O processo é baseado em dados qualitativos e quantitativos e técnicas de análise. Muitas das decisões importantes de projeto feitas não são perceptíveis durante o ciclo de vida de um sistema, então entrevistas com a equipe de desenvolvimento são necessárias de maneira que se possa entender o sistema. O sistema é submetido a uma análise quantitativa baseada, por exemplo, em medidas técnicas.

Figura 14 – Componentes do processo de avaliação

Passo 1 – Selecionar a perspectiva da avaliação

Um sistema pode ser avaliado com diferentes objetivos em mente e de várias perspectivas diferentes. Uma avaliação pode ser baseada em requisitos funcionais especificados ou baseada na satisfação dos requisitos não-funcionais (como, por exemplo, segurança, disponibilidade e manutenibilidade). Selecionar uma perspectiva é importante para se identificar objetivos e medições apropriadas. Para este passo, Tesoriero recomenda utilizar a técnica GQM [Bas94], que ajuda na definição de um objetivo definido a partir de questões respondidas pelos interessados.

Passo 2 – Definir diretrizes e medidas

De acordo com a perspectiva selecionada, diretrizes e medidas são definidas. Diretrizes de projeto podem ser usadas para validar se a arquitetura possui as propriedades desejadas e para

definir medidas que podem ser usadas para avaliar a arquitetura. Algumas vezes, medidas podem ser derivadas de diretrizes. Por exemplo, diretrizes para avaliar manutenibilidade ditam que o acoplamento entre componentes deve ser pequeno e que a quantidade de acoplamento intermódulos deve ser reduzido sem aumentar o acoplamento interno (ou intramódulos). Em conjunto a definição de diretrizes e medidas baseadas na perspectiva escolhida, há diretrizes e medidas que são definidas baseadas no estilo da arquitetura e nos padrões de projeto escolhidos para o sistema. Uma tarefa para a equipe de análise é, então, identificar e documentar quais estilos arquiteturais e padrões de projeto são usados e com quais implicações nas regras de projeto. A equipe de análise precisa selecionar e customizar as diretrizes e medidas para o contexto específico. O conjunto selecionado de medidas deve representar as propriedades que a equipe avaliar como mais importante enquanto que, ao mesmo tempo, tenha uma boa relação custo/benefício para se coletar e analisar. Para esta dissertação serão analisados como base para análise a opinião de especialistas, bem como medidas de tempo de execução de determinadas atividades. Essas medidas serão detalhadas na seção 4.3.

Passo 3 – Definir a arquitetura planejada

A arquitetura planejada (ou ideal) é definida pelos requisitos arquiteturais, pelas diretrizes arquiteturais explicitas e implícitas e por regras de projeto, além de implicações que advem do uso de estilos arquiteturais e padrões de projeto. Na verdade, a arquitetura planejada é mais um objetivo para o qual a arquitetura deve se aproximar, ao invés de ser a arquitetura implementada de verdade. Uma razão para esta inconsistência entre a arquitetura planejada e real é que os sistemas passam por diversas mudanças em seu ciclo de vida. Os riscos de inconsistências entre as arquiteturas planejada e real e as violações de princípios de projeto são especialmente altos quando a equipe de desenvolvimento que iniciou o projeto é diferente daquela que implementa as mudanças. A equipe de análise precisa recuperar os diferentes aspectos da arquitetura planejada e criar um modelo que guiará a avaliação. Comumente, diretrizes e medidas são avaliadas e atualizadas em paralelo a este passo, na medida em que a equipe de análise aprimora seus conhecimentos sobre a arquitetura planejada.

Passo 4 – Identificar a arquitetura real

A arquitetura atual é a estrutura de alto nível do sistema implementado, seus componentes arquiteturais e relacionamentos e seus estilos de arquitetura e padrões de projeto. Estudando a implementação do sistema (código fonte do sistema) identifica-se a arquitetura atual. Deve-se notar que a avaliação da arquitetura não é equivalente à análise do código, mas identifica os componentes arquiteturais do sistema atual. Para que este passo seja tão eficaz quanto possível, a equipe de análise define seu próprio conjunto de ferramentas que os auxilia nessa tarefa. As ferramentas freqüentemente têm que ser definidas com base na linguagem de programação e outros fatores do ambiente de desenvolvimento. Para Java, por exemplo, a equipe pode usar uma ferramenta que identifique pacotes e classes e suas inter e intradependências de pacotes. Estilos arquiteturais não são sempre facilmente identificados na implementação real do sistema. Uma das tarefas da equipe de análise é identificar qual e onde estes estilos são usados. Parte da tarefa da equipe de análise é identificar ocorrências de padrões de projeto e as classes que participam dos padrões.

Passo 5 – Identificar desvios da arquitetura

Desvios na arquitetura são diferenças entre a arquitetura planejada e a real. Estes desvios podem ser violações de regras de projeto e diretrizes ou valores de medidas que excedem um certo patamar. A equipe de análise toma notas de cada desvio identificado, as circunstâncias nas quais ele foi detectado e a razão pela qual a equipe suspeita que ele é uma violação. Se necessário, a equipe conduz uma análise mais detalhada do desvio para determinar sua possível causa e grau de severidade.

Passo 6 – Formular recomendações de mudanças

Baseada nos resultados do passo anterior, a equipe de análise formula uma recomendação de mudança para remover os desvios do sistema. Algumas vezes, os desvios resultam em pedidos de mudança de código fonte, e, outras vezes, mudanças na arquitetura planejada ou diretrizes. Deve-se notar que não é tarefa da equipe de análise projetar ou implementar as mudanças. Ao

invés disso, este passo deve ser visto como uma maneira da equipe de análise contribuir para a melhoria do sistema de uma maneira construtiva, realimentando a equipe de desenvolvimento.

Passo 7 – Verificar mudanças arquiteturais

As mudanças identificadas que são implementadas requerem um passo extra para se verificar que a arquitetura atual está de acordo com a planejada. Esse passo repete o passo de identificar a arquitetura atual e qualquer desvio arquitetural. Essa verificação é feita para se ter certeza de que as mudanças foram feitas corretamente e que nenhuma violação nova foi introduzida no sistema.

Esse processo de avaliação da arquitetura foi aplicado no Caso de Estudo desta dissertação (localizado na seção 4) contendo todo o detalhamento do processo, bem como os dados extraídos para análise. No próximo capítulo serão apresentadas as diferentes tecnologias para a arquitetura de *software*.

2.6.2 Processo proposto por Kazman (ATAM)

Kazman propôs o processo de avaliar a arquitetura chamado ATAM (Architecture Tradeoff Analysis Methods – Métodos de Análise de relacionamento de arquitetura). A ATAM tem esse nome porque ela não só revela o quanto uma arquitetura satisfaz objetivos de qualidade (tal como desempenho ou manutenibilidade), mas também prover a informação de como estes objetivos de qualidade interagem entre si. O objetivo da ATAM é avaliar as conseqüências de decisões arquiteturais com a ajuda dos requisitos de qualidade. Os passos propostos para a analise são:

Passo 1 – Apresentar a ATAM

Neste passo a equipe da avaliação apresenta a ATAM para as partes interessadas detalhando como será o processo que todos seguirão. É importante que todos saibam quais informações serão coletadas, como serão examinadas, e para quem isso será reportado.

Passo 2 – Apresentar perspectiva de negócio

O sistema a ser avaliado precisa ser entendido por todos os participantes da avaliação. Neste passo o gerente de projeto apresenta uma revisão do sistema sob a perspectiva de negócio. O sistema em si deve ser apresentado, inicialmente em um nível alto de abstração, tipicamente descrevendo:

- Os requisitos funcionais mais importantes
- As limitações técnicas, gerenciais, econômicas ou políticas
- O contexto e os objetivos do negócio
- Os principais interessados

Passo 3 – Apresentação da arquitetura

A arquitetura será apresentada pelo arquiteto em um nível apropriado de detalhamento. O nível de detalhamento depende de vários fatores tais como: quantidade de informação foi documentada, quantidade de tempo que se tem disponível; o risco que o sistema está sujeito. Este é um passo importante pois a quantidade de informações disponíveis e documentadas vai afetar diretamente a análise e a sua respectiva qualidade. Freqüentemente a equipe de avaliação precisará especificar informações adicionais sobre a arquitetura, que são requeridas para serem documentas antes de uma análise mais substancial.

Passo 4 – Identificar o foco arquitetural

A ATAM tem como base a análise da arquitetura através do entendimento do foco arquitetural. Neste passo, os focos são identificados pelo arquiteto, e capturados pela equipe de análise, mas não são analisados. Os focos arquiteturais definem as estruturas importantes do sistema e descrevem as maneiras as quais o sistema pode crescer, responder a mudanças, resistir a ataques e ser integrado a outros sistemas.

Passo 5 – Gerar a árvore de requisitos de qualidade

Neste passo a equipe de avaliação trabalha com a equipe de arquitetura, gerente, e representantes do cliente para identificar, priorizar, e refinar os requisitos de qualidade e objetivos mais importantes do sistema. Este é um passo crucial pois ele guia o restante da análise. A análise, mesmo no nível da arquitetura de *software*, não é inerentemente limitada ao escopo, então é preciso ter meios de se focar a atenção de todos os interessados nos aspectos da arquitetura que são mais críticos para o sucesso do sistema. Faz-se isso através da construção de uma árvore de requisitos de qualidade.

Passo 6 – Analisar o foco arquitetural

Precisa-se estar convencido que o foco da arquitetura que está sendo analisado possui uma chance significante de cumprir os requisitos específicos para os quais ele foi planejado. A maior contribuição desta fase são uma lista dos focos ou estilos arquiteturais, as questões associadas a eles, e as respostas dos arquitetos a essas questões.

Passo 7 – Investigação e priorização de cenários

Cenários são os motores que conduzem a fase de testes da ATAM. Gerar uma série de cenários facilita nas investigações e discussões com os interessados. Cenários são exemplos de estímulos arquiteturais usados para:

- Representar necessidades dos interessados
- Entender os requisitos de qualidade

Os interessados realizam duas atividades relacionadas: a investigação de cenários de caso de uso (representando as maneiras as quais os interessados esperam que o sistema seja usado) e mudanças de cenários (representando as maneiras as quais os interessados esperam que o sistema mude no futuro). Cenários de caso de uso são um tipo cenário onde o interessado é um usuário

final, usando o sistema para executar certa função. Uma vez que os cenários foram coletados, eles devem ser priorizados através de um procedimento de votação realizado pelos interessados.

Passo 8 – Analisar o foco arquitetural

Depois que os cenários foram coletados e analisados, o arquiteto começa o processo de mapeamento dos cenários com mais alta prioridade na descrição arquitetural que foi apresentada. Idealmente esta atividade será dominada pelo mapeamento de cenários feito pelo arquiteto em um foco previamente discutido. De fato, o ponto de junção entre as duas fases é assegurar que este é o caso. Se este não for o caso, então ou o arquiteto não tem uma solução de foco ou estilo para o estímulo que o cenário representa, ou o foco existe, mas não foi revelado por nenhuma atividade até este ponto.

Passo 9 – Apresentar os resultados

Finalmente, as informações coletadas da ATAM precisam ser sumarizadas e apresentadas aos interessados. Esta apresentação tipicamente é na forma de uma apresentacao acompanhado por slides, mas pode, também, ser acompanhada de um reporte escrito mais completo entregue subseqüentemente a ATAM. Nesta apresentação recapitula-se os passos da ATAM e todas as informações coletadas nos passos do método, incluindo: o contexto do negócio, requisitos, limitações, e a arquitetura. O mais importante, no entanto, é o conjunto de contribuições da ATAM:

- Os focos/estilos da arquitetura documentados
- O conjunto de cenários e suas priorizações
- O conjunto de perguntas baseadas nos atributos
- A árvore de requisitos de qualidade
- A descoberta de riscos
- Os fatores não considerados risco, documentados
- Os pontos mais sensíveis e relacionamentos encontrados

Cada uma destas contribuições deve ser descrita e, em alguns casos, deve ser oferecida uma estratégia de mitigação. Pelo fato de que se está sistematicamente trabalhando através da arquitetura e tentando se entender os focos arquiteturais, é inevitável que, algumas vezes, se faça recomendações de como a arquitetura possa ser projetada ou analisada diferentemente. Estas estratégias de mitigação podem ser processadas em conjunto (por exemplo, um administrador de banco de dados pode ser consultado antes de completar o projeto da interface administrativa do usuário), elas podem ser gerenciáveis (por exemplo, três subgrupos dentro do desenvolvimento estão em busca de objetivos altamente similares, e estes objetivos poderiam ser agrupados), ou elas podem ser técnicas. Entretanto, oferecer estratégias de mitigações não é uma parte integral da ATAM. A ATAM é responsável por localizar riscos arquiteturais.

Existem outros processos para se avaliar uma arquitetura como por exemplo o processo SAAM (*Scenario based Architecture Analysis Method* - Método de análise de arquitetura baseado em cenários) [Kaz94], porém nessa dissertação foram estudados apenas os processos propostos por Tesoriero [Tes02] e Kazman [Kaz00].

2.7 Considerações finais

Nesse capítulo foram apresentadas abordagens de Sommerville[Som04] e de outros autores contendo conceitos importantes para a compreensão da arquitetura de *software* como decomposição, estrutura, modelos e padrões arquiteturais e de projeto. Além desses conceitos, foram apresentados processos para avaliar a arquitetura de *software* segundo Tesoriero [Tes02] e Kazman [Kaz00]. No próximo capítulo serão apresentadas algumas tecnologias utilizadas para implementação da arquitetura de *software*.

3 Tecnologias para implementação da Arquitetura de *Software*

Este capítulo apresenta as diferentes tecnologias (*CORBA* e *Java 2 Enterprise Edition - J2EE*) para a implementação da Arquitetura de *software*, bem como seus conceitos e características.

3.1 Corba

CORBA (*Common object request broker architecture*) é um conjunto de especificações usadas para implementação da arquitetura de sistema. A arquitetura CORBA define uma abordagem genérica para a computação de objetos distribuídos [Som04]. Foi desenvolvida uma série de implementações dessa arquitetura por diferentes fabricantes. As implementações CORBA são disponíveis para sistemas operacionais Unix e Microsoft [Orf98].

Existem quatro elementos principais para CORBA:

1. Um modelo de objetos para objetos de aplicação, em que um objeto CORBA é um encapsulamento de estado, com uma interface bem definida, de linguagem neutra, descrita em uma IDL (*interface definition language* – linguagem de definição de interface).

2. *Object Request Broker* (ORB) gerencia solicitações de serviços de objetos. Ele localiza o objeto que fornece o serviço, prepara-o para a requisição, envia a requisição de serviço e retorna o resultado ao solicitante.

3. Um conjunto de serviços de objetos, que são serviços gerais que provavelmente serão requisitados por muitas aplicações distribuídas. Por exemplo, serviços de diretórios, serviços de transações e serviços de persistência.

4. Um conjunto de componentes comuns construído tendo como base esses serviços básicos, que pode ser solicitado por aplicações. Esses componentes podem ser de domínio específico ou de propósito geral, que são utilizados por muitas aplicações.

3.2 Java 2 Enterprise Edition (J2EE)

É um conjunto das APIs (*Application Program Interface* – Interface de Programas Aplicativos) de Java corporativas e de uma plataforma de desenvolvimento para sistemas distribuídos corporativos em Java [Bam02]. O modelo de desenvolvimento J2EE tem uma clara distinção entre desenvolvimento de sistema, distribuição e execução. Devido a esse fato, desenvolvedores podem distinguir detalhes de distribuição, como o nome de uma base de dados e sua localidade, propriedades específicas da configuração do servidor, e assim por diante.

As categorias da arquitetura J2EE são descritas abaixo e mostradas através da Figura 15:

- **Camada cliente** para a interação com o usuário. A camada cliente reside na máquina do usuário. É composta por páginas HTML e por um navegador responsável pela exibição dessas páginas.

- **Camada de apresentação** reside em um servidor *web*, sendo responsável por processar as requisições da camada cliente e repassar essas requisições para a camada de negócios

- **Camada de negócio** implementa a lógica de negócio da aplicação; A camada de negócios reside em um servidor de aplicações, e é composta por *Enterprise JavaBeans* (EJB).

- **Camada de dados** é composta por SGBDs e aplicações legadas que são responsáveis por armazenar e fornecer as informações utilizadas pelo sistema.

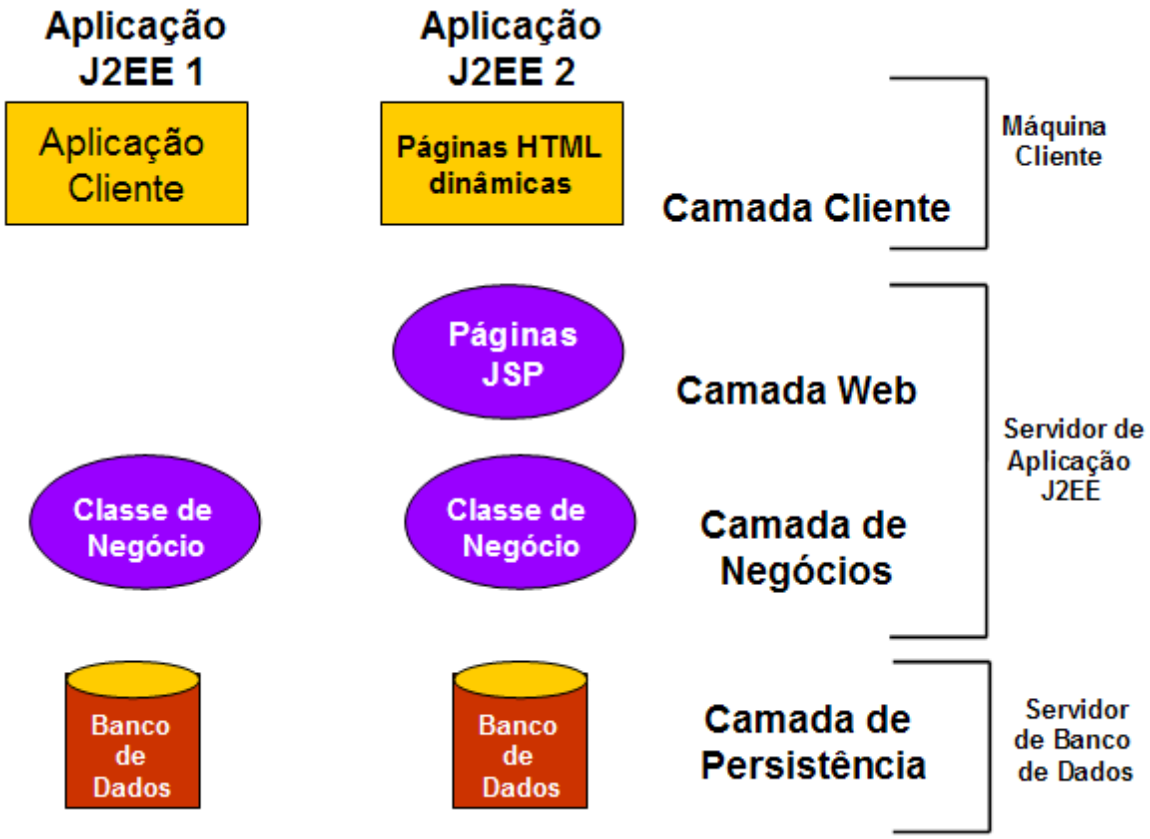

Figura 15 – Camadas da arquitetura J2EE

A missão do J2EE é fornecer independência de plataforma, portabilidade, múltiplos usuários, segurança e um padrão para grandes sistemas distribuídos, escritos na linguagem Java. Como o J2EE é uma especificação (conjunto de necessidades de várias empresas), ele não fica amarrado apenas a um tipo de fornecedor, pois vários fornecedores podem implementar diferentes versões para diferentes plataformas.

Para entender as tecnologias J2EE, deve-se entender o papel do *container*. O *Container* é uma entidade de *software* que é executada dentro de um servidor e é responsável pelo gerenciamento de tipos específicos de componentes. Ele fornece um ambiente de execução para os componentes desenvolvidos em J2EE, e gerencia o ciclo de vida dos mesmos. É através de tais *containeres* que a arquitetura J2EE pode fornecer independência entre desenvolvimento e distribuição e prover portabilidade entre diversos servidores. J2EE tem três tipos diferentes de *containeres*:

- ***Application container*:** servidores de aplicações Java *stand-alone*;
- ***Web container*:** servidores de componentes *Web*, como *Servlets* e *Java Server Pages* (JSP) que usam o padrão Modelo-Visão-Controle descrito na seção 2.5;
- ***Enterprise container*:** servidores de componentes EJB.

Serão apresentadas a seguir as tecnologias mais comumente usadas numa arquitetura J2EE.

- ***Servlets*** são componentes capazes de gerar conteúdo dinâmico. Essa tecnologia fornece um mecanismo eficiente para interação entre a camada de negócio e a camada *Web* de apresentação.
- **Java Server Pages** são muito parecidos com os *servlets*. De fato, os arquivos JSP são convertidos para *servlets* em tempo de compilação. A grande diferença entre JSP é que os arquivos não são código Java puro, pois são mais focados na aparência da página (HTML, *JavaScript*, *Flash*). Com JSP é possível inserir código Java na estrutura HTML ou XML. É recomendável inserir código Java com um JSP, se este for relativamente simples.
- **Java DataBase Connector (JDBC)**

  Interação com banco de dados é uma parte integrante de uma aplicação Java. A API JDBC foca em tornar esse aspecto fácil para o desenvolvedor Java, simplificando o acesso a banco de dados relacionais. Consiste de uma interface genérica com o banco de dados independente de fabricante. Existem algumas extensões que permitem algumas funções avançadas para os containeres J2EE, como *pool* de conexões, bem como algum suporte adicional para *JavaBeans*. Nas aplicações não é necessário usar JDBC diretamente. Por exemplo, pode-se usar *entity beans* para fazer chamadas ao banco de dados.
- **XML**

  Na década de 80, a troca de dados e documentos era limitada a formatos de documento patenteados ou fracamente definidos. O advento da HTML ofereceu às empresas um formato padrão para troca com um foco no conteúdo visual interativo. HTML está rigidamente definida e não pode suportar todos os tipos de dados corporativos; portanto,

essas desvantagens forneceram o impulso para criar a XML que significa *EXtended Markup Language*. O padrão XML permite que a empresa defina suas próprias linguagens de marcação com ênfase em tarefas específicas, como comércio eletrônico, gerenciamento de dados e publicação.

Por essas razões, o padrão XML tornou-se o instrumento estratégico para definir dados corporativos em vários domínios de aplicação facilitando a integração dos sistemas. As propriedades em XML tornam adequado representar dados, conceitos e contextos de maneira aberta, independente de plataforma, de fornecedor e de linguagem. Ela utiliza tags, identificadores que sinalizam o início e o fim de um bloco de dados relacionados, para criar uma hierarquia de componentes de dados relacionados chamados de elementos. Por sua vez, essa hierarquia de elementos fornece encapsulamento e contexto. Como resultado, há uma oportunidade maior de reutilizar esses dados fora da aplicação e origens de dados a partir das quais eles foram derivados. As aplicações escritas na linguagem de programação Java que processam XML podem ser reutilizadas em qualquer nível em um ambiente cliente-servidor multicamada, oferecendo um nível adicional de reutilização para documentos de XML. No caso de estudo que será relatado no capítulo 4, o XML será usado como protocolo de comunicação responsável pela realização da integração entre os sistemas.

- **Enterprise JavaBeans (EJBs)** são componentes de negócio de granulação grossa que são executados em um *container* dentro de um servidor de aplicação. Os EJBs são projetados para serem usados como componentes interprocessos e de forma distribuída. Existem três tipos de EJBs:

  - Session beans: são melhor usados para atividades temporárias. São não persistentes e freqüentemente encapsulam a maioria da lógica de negócio dentro de uma aplicação Java. Session beans podem ser "stateful", significando que retêm conexões entre interações sucessivas com um cliente. O outro tipo de session bean é "stateless". Neste caso, cada invocação sucessiva do session bean pelo mesmo cliente é tratada como se fosse uma nova atividade.

- **Entity Beans:** encapsulam dados persistentes em um banco de dados. Eles fornecem serviços automatizados para assegurar que a visão de orientação a objeto do dado persistente permaneça sincronizada todo o tempo com os dados residentes no banco de dados. Entity Beans também são freqüentemente usados para formatar dados que serão apresentados na camada cliente e são compostos de dois tipos Container Management Persistence (CMP) e Bean Management Persistence (BMP).

- Message-driven beans: são projetados para serem consumidores assíncronos de mensagens JMS (Java Messaging Service). Ao contrário de entity beans e de session beans, message-driven beans não tem interfaces publicadas, pois operam anonimamente atrás do cenário (contexto). São do tipo "stateless".

A Figura 16 mostra a utilização de servlets e JSP executando no Web Server como interface entre browser ou web services, e servidor de aplicações (executando EJB).

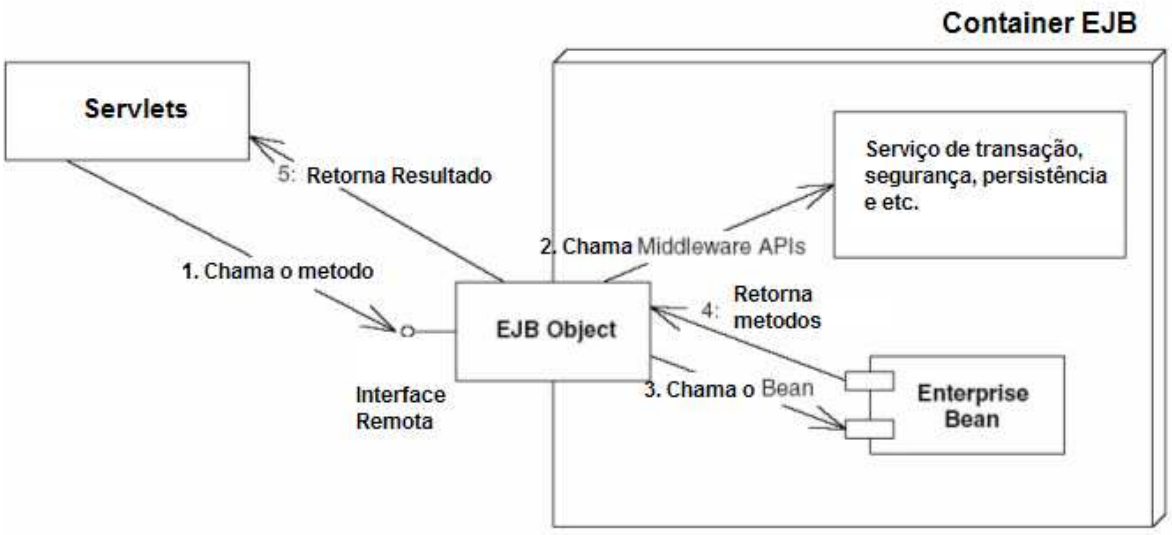

Figura 16 – Comunicação entre Servlets e EJBs

O código do cliente chama um determinado método do componente EJB. A requisição chega na interface remota que repassa o controle para o objeto EJB.

O objeto EJB antes de repassar o controle para o componente EJB realiza uma série de operações para garantir os serviços automáticos. Após a realização das operações internas o controle é repassado então para o componente EJB. O componente EJB realiza as suas operações e retorna o valor para o objeto EJB. O objeto EJB retorna a resposta para o cliente.

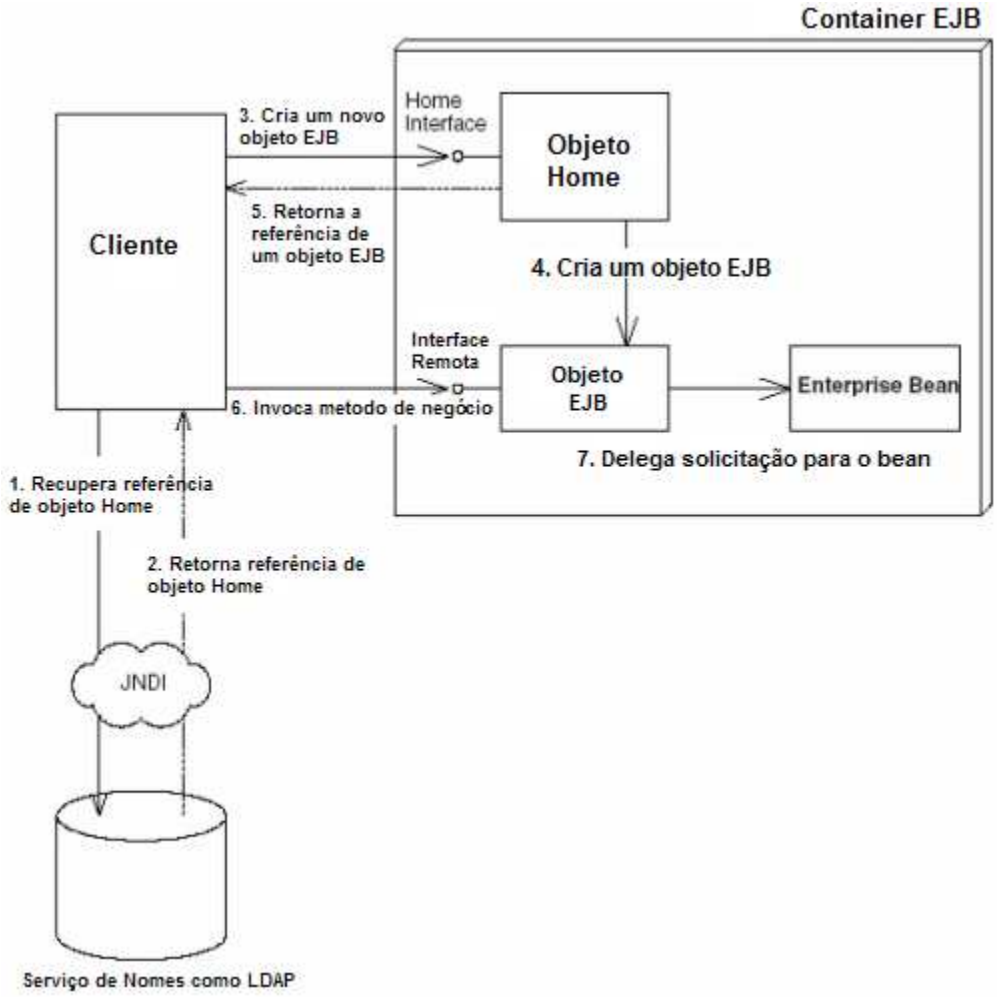

Figura 17 – Comunicação entre o Cliente e EJBs

A Figura 17 mostra como um usuário utiliza um componente EJB. O usuário utiliza um servidor de nomes para receber uma referência para o objeto

denominado home. O objeto home é único para um determinado componente EJB e serve para criar e remover as instâncias daquele tipo de componente.

O usuário utilizando a interface do objeto home chama os métodos de criação do componente EJB. A interface remota recebe a requisição e a repassa para o objeto home. Uma nova instância do objeto EJB e do componente EJB são criadas e a referência para o objeto EJB é retornada para o usuário.

Através da referência ao objeto EJB, o usuário pode chamar os métodos de negócio (utilizando a interface remota) que são interceptados pelo objeto EJB. O objeto EJB realiza as operações necessárias para garantir os serviços automáticos e repassa a chamada para o componente EJB.

Uma vez que os conceitos de arquitetura, medidas, padrões e tecnologia foram explicados detalhadamente será mais fácil para entender o Caso de Estudo que será descrito a seguir e tentar solucionar os problemas que serão discutidos na próxima seção.

3.3 Considerações finais

Nesse capítulo foram apresentadas as tecnologias CORBA e Java 2 Enterprise Edition (J2EE) para a implementação da Arquitetura de *software*, bem como seus conceitos e características. No próximo capítulo será apresentado o estudo de caso, suas características, problema e solução adotada.

4 O estudo de caso

Este capítulo descreve um sistema de *Workflow* apresentando suas características, seus requisitos funcionais e não-funcionais, sua implementação, problemas encontrados no desempenho e finalmente a solução arquitetural proposta para o sistema.

Este sistema foi implementado por uma empresa de desenvolvimento de sistemas para Telecomunicações composta por uma equipe contendo arquiteto, desenvolvedores e testadores. Este sistema já estava pronto quando foi iniciado o processo de análise dessa dissertação.

4.1 Descrição do Sistema

O *Workflow* é um sistema que integra tecnologias de *middleware* (camada de negócio) para integrar e coordenar a execução de aplicações heterogêneas no contexto de uma corporação.

O objetivo da integração é fazer com que aplicações heterogêneas e desenvolvidas isoladamente operem conjuntamente. Além da necessidade de integração, existe ainda a necessidade de descrever as diferentes interações entre essas aplicações (processos de negócio) e coordenar a execução dessas descrições.

Dentre os principais fatores que dificultam a integração de aplicações estão a diversidade técnica (sistemas operacionais, linguagens de programação, ferramentas de desenvolvimento e SGBDs heterogêneos) e a diversidade de projeto (modelo de informação, comportamento e semântica heterogêneos).

Existem várias alternativas para a integração de sistemas como, por exemplo, modelo de camadas cliente-servidor e objetos distribuídos apresentados no Capítulo 2 e 3. A mais amplamente utilizada atualmente é a integração ponto-a-ponto, conforme mostrado na Figura 18. A principal desvantagem desse tipo de integração é o alto custo de manutenção devido ao grande número de combinações possíveis.

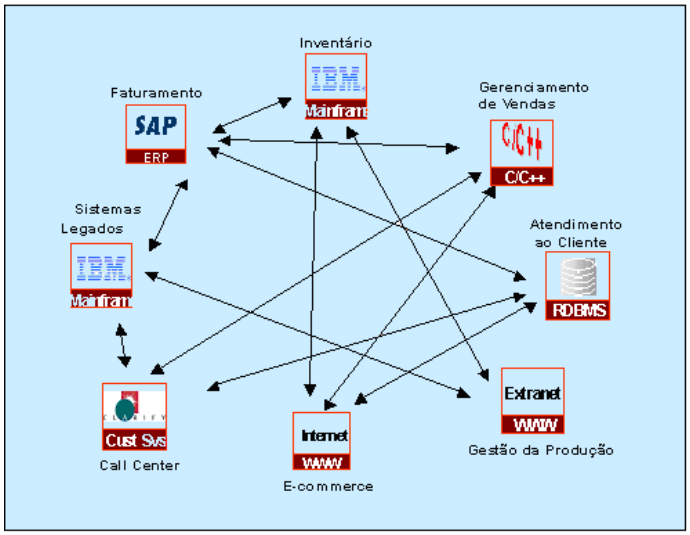

Figura 18 - Integração de aplicações ponto-a-ponto

A Figura 18 mostra o grande número de combinações entre as diferentes aplicações ponto a ponto. O objetivo do Workflow é reduzir o número dessas combinações e transferir a responsabilidade pela integração dessas aplicações para a camada de negócio (*middleware*) visto na Figura 19 e conceituados na seção 3.2.

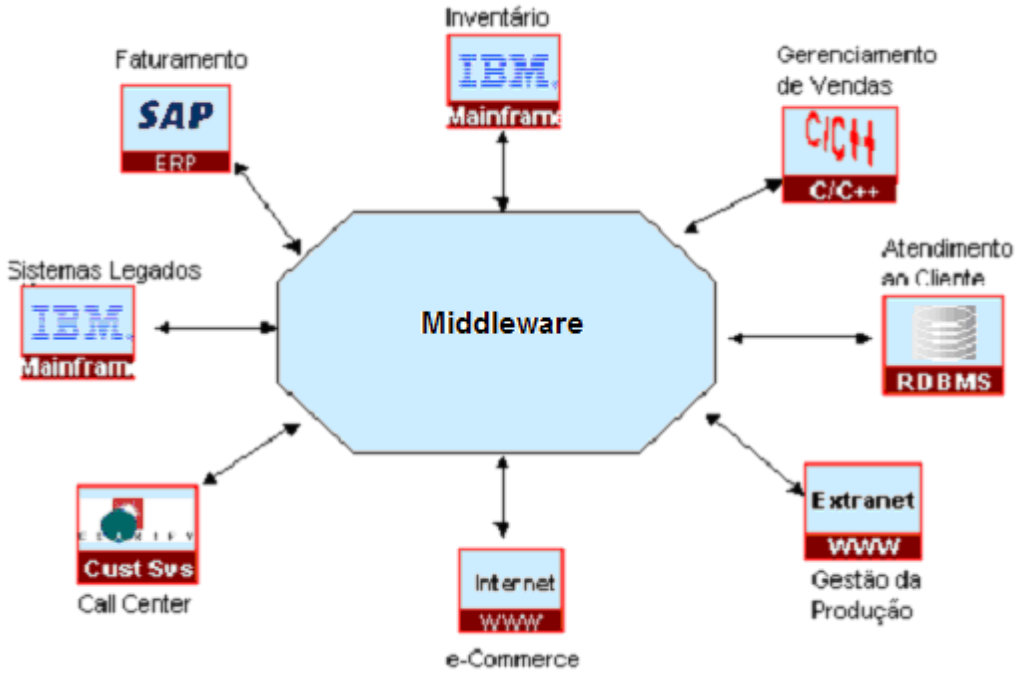

Figura 19 - Integração de aplicações via *middleware*

4.2 Requisitos do Sistema Workflow

Esta seção apresenta os requisitos funcionais e não-funcionais do sistema que foram decisivos na escolha da arquitetura do Workflow. Para facilitar o entendimento foi utilizada a sugestão de [Mac99] e [Bas98] já apresentado na seção 1.5.

Para melhor entender o que significa cada terminologia, a equipe de requisitos do sistema definiu e detalhou todas as terminologias utilizadas no Workflow.

Fila de Mensagens: é uma fila de origem/destino de troca de mensagens entre sistemas.

Definições de atividades: são criadas independentemente das definições de processos e podem ser utilizadas por vários processos. Uma definição de atividade representa uma ação atômica que pode ser acionada pelo núcleo de execução. Essa ação pode ser executada por humanos (atividades manuais) ou por sistemas externos (atividades automáticas). Uma definição de atividade pode ter uma contra-atividade associada. Essa contra-atividade representa a ação a ser executada caso seja necessário desfazer uma atividade.

As atividades podem ter os estados:

- **Aberta:** a atividade não foi ainda executada
- **Em andamento:** a atividade foi iniciada e ainda está em execução
- **Concluída:** a atividade foi finalizada contendo erro ou não

Requisição: solicitação de um usuário ou sistema externo para executar uma determinada ação no sistema Workflow.

Processo: é composto por um conjunto de atividades, uma lista de arestas (representam as dependências entre as atividades do processo) e uma lista de variáveis (utilizadas para armazenar o estado interno do processo, trocar informações entre as atividades e criar condições associadas às arestas). Uma condição associada a uma aresta especifica uma pré-condição para que a próxima atividade seja executada.

4.2.1 Visão de Casos de Uso

Esta seção descreve o subconjunto mais significativo (do ponto de vista de arquitetura) da especificação dos requisitos funcionais através da apresentação dos casos de uso do sistema *Workflow*. As figuras apresentadas nessa seção foram escritas em inglês devido à restrição da organização onde o *software* foi desenvolvido. Estes casos de uso estão agrupados pelos subsistemas Núcleo de Execução, Integrador, Adaptador e Gerenciador de Requisições que são descritos nessa seção.

É função do administrador cadastrar e disponibilizar as atividades que o núcleo de execução seja capaz de acionar. Após o cadastro de atividades, um especialista é capaz de criar definições de processos que modelam os processo de negócio da empresa e que podem ser instanciadas e executadas pelo núcleo de execução. Além disso, o administrador é responsável por executar tarefas dos subsistemas do Workflow como, por exemplo, descrever as regras de transformação dos dados para a comunicação entre aplicações do Integrador. Os casos de uso Manter aplicações e serviços, Monitorar e auditar sistemas e Manter conversão de dados são aqueles com os quais o Administrador interage ilustrados na Figura 20.

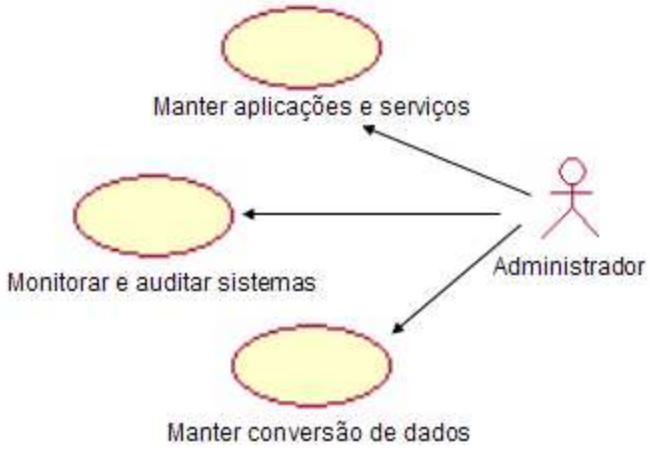

Figura 20 - Casos de Uso do Administrador

Núcleo de Execução

O diagrama de casos de uso mostrado na Figura 21 identifica os principais casos de uso do Núcleo de Execução do *Workflow*. Os principais casos de uso serão brevemente descritos a seguir.

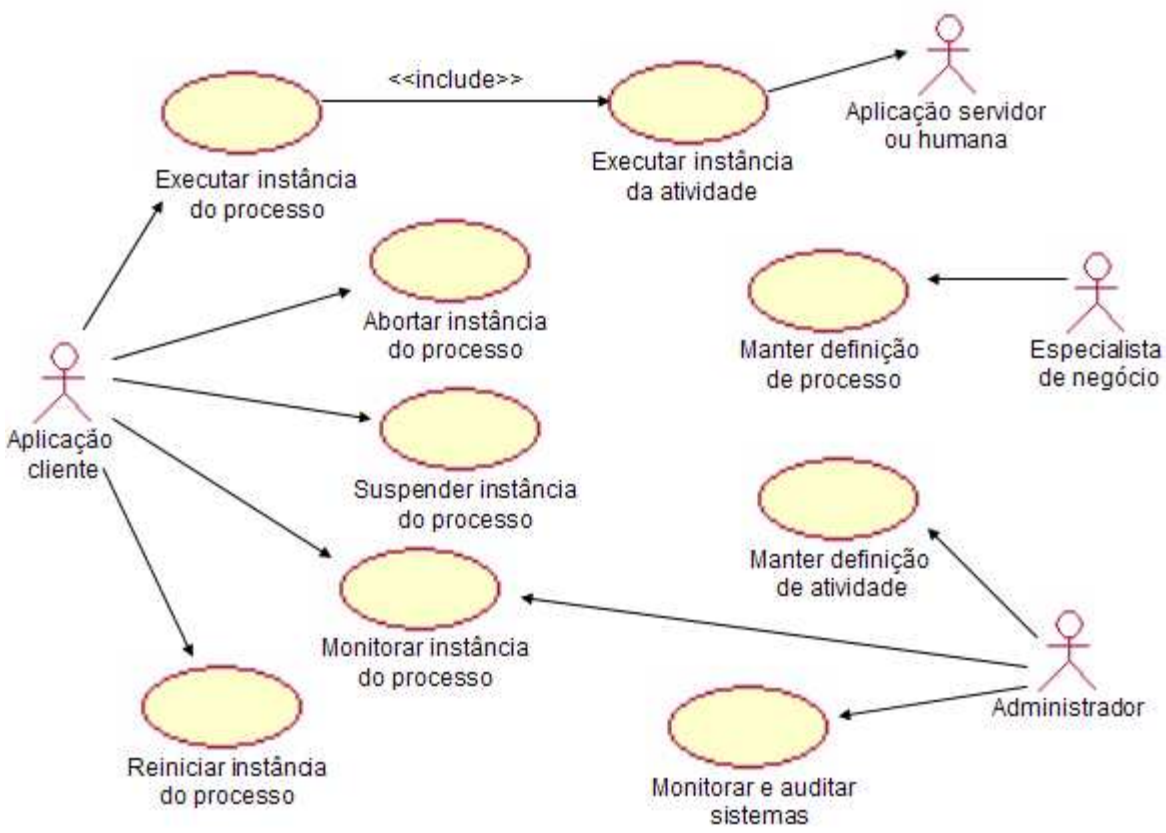

Figura 21 - Casos de Uso do Núcleo de Execução

O principal caso de uso do núcleo de execução diz respeito à execução de instâncias de processos (*Execute Process Instance*) a partir da solicitação de uma aplicação cliente conforme ilustra a Figura 21. A execução de uma instância de processo envolve a execução de instâncias de atividades por aplicações servidoras (quando a atividade é automatizada) ou por humanos (quando a atividade é manual).

A aplicação que solicitou a instanciação de um processo é capaz de monitorar a execução dessa instância e interferir em sua execução, suspendendo, reativando ou cancelando a sua

execução. O administrador do sistema também é capaz de monitorar a execução das instâncias de processo.

Integrador

O diagrama de casos de uso mostrado na Figura 22 identifica os principais casos de uso do Integrador do *Workflow*. Os principais casos de uso serão brevemente descritos a seguir.

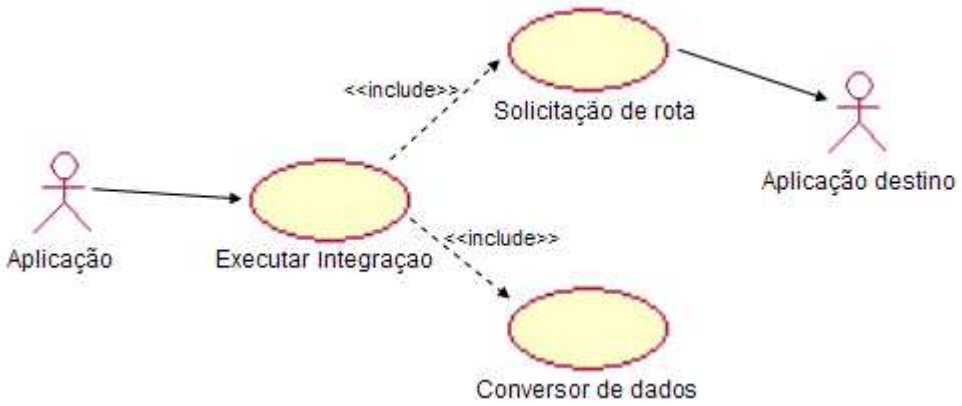

Figura 22 - Casos de uso do Integrador

O principal caso de uso do integrador é o caso de uso Executar Integração. Esse caso de uso representa toda a lógica de execução do integrador para fazer com que uma mensagem enviada por uma aplicação fonte seja entregue corretamente para uma aplicação destino, afim de que algum serviço seja executado. Essa lógica envolve tanto o roteamento da mensagem pelo uso Solicitação de rota, quanto à transformação dos dados que estão sendo transferidos através do caso de uso Conversor de Dados.

Adaptador

O diagrama de casos de uso mostrado na identifica os principais casos de uso de um Adaptador do *Workflow*. Os principais casos de uso serão brevemente descritos a seguir.

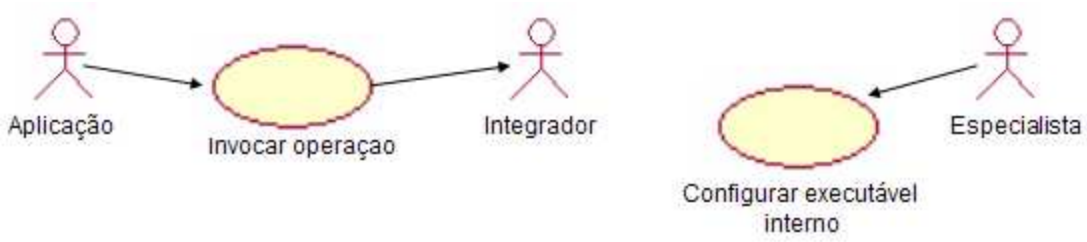

Figura 23 - Casos de uso do Adaptador

O principal caso de uso do adaptador diz respeito à invocação de operações conforme mostrado na Figura 23. A invocação de operação pode ser feita tanto pelo sistema integrado ao qual o adaptador está associado ou quando no sentido inverso. Cada sistema integrado terá um adaptador específico para o seu protocolo de comunicação XML, que eventualmente poderá ser configurado por um administrador.

Gerenciador de Requisições

O diagrama de casos de uso mostrado na Figura 24 identifica os principais casos de uso do Gerenciador de Requisições do *Workflow*. Esses casos de uso serão brevemente descritos a seguir.

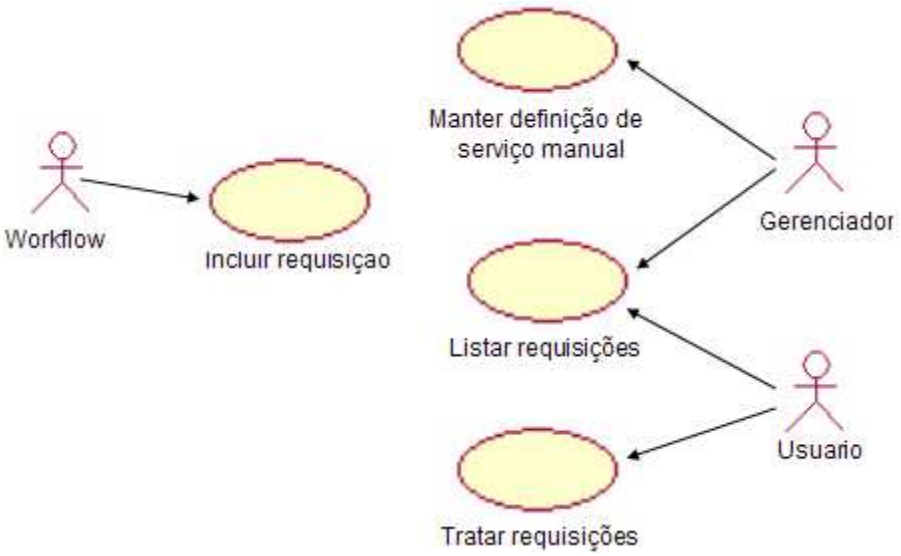

Figura 24 - Casos de uso do Gerenciador de Requisições

O principal caso de uso do gerenciador de requisições é relativo ao tratamento de requisições de serviços manuais. O núcleo de execução sinaliza para esse gerenciador toda requisição para a execução de serviços manuais, ilustrada na Figura 24. As definições dos serviços manuais que o gerenciador é capaz de tratar são cadastradas por um administrador. Quando solicitado, o gerenciador apresenta a lista de requisições para o usuário (técnico) através do caso de uso Listar requisições. O usuário então escolhe a requisição que deseja tratar e executa o serviço relativo a essa requisição. Quando o usuário termina a execução do serviço, o gerenciador sinaliza este evento de volta para o núcleo de execução, juntamente com os dados de retorno da requisição.

4.2.2 Visão Lógica

Essa seção descreve os requisitos não-funcionais bem como um panorama da arquitetura em função da decomposição do *Workflow* em subsistemas e a decomposição desses subsistemas em pacotes como pode ser visto na Figura 25. O sistema Workflow é decomposto nos subsistemas Núcleo de Execução, Integrador, Adaptador e Gerenciador de Requisições significativos do ponto de vista arquitetural conforme foi apresentado na seção 4.1. Os requisitos não-funcionais são descritos a seguir:

- O sistema deve ser compatível com os Sistemas Operacionais Microsoft Windows, Linux e Unix.
- Utilizar *frameworks* e padrões arquiteturais que facilitem o desenvolvimento e garantam a manutenibilidade do sistema conforme visto na seção 2.4;
- Interfaces Web compatíveis com Internet Explorer e Netscape;
- Log de execuções de todas as ações realizada pelos usuários;
- Protocolo de comunicação XML para troca de dados entre as aplicações conforme visto na seção 3.2;
- Permitir múltiplos usuários;

- Segurança das informações trocadas entre as aplicações, no acesso aos serviços fornecidos por cada aplicação e no acesso às interfaces do sistema;
- Suporte à comunicação assíncrona, para a manutenção da fila de mensagens;
- Permitir que o sistema seja distribuído
- Independência de Sistema de Gerenciamento de Banco de Dados (SGBD) e servidor de aplicações, pois o cliente pode ter um outro ambiente diferente do ambiente onde o sistema foi desenvolvido;

Pode-se observar que nada foi especificado sobre desempenho do sistema e por este motivo nenhuma atitude foi tomada durante a definição da arquitetura.

Depois de especificados os requisitos não-funcionais do sistema através da visão lógica, será descrita a sua decomposição conforme visto na seção 2.2.

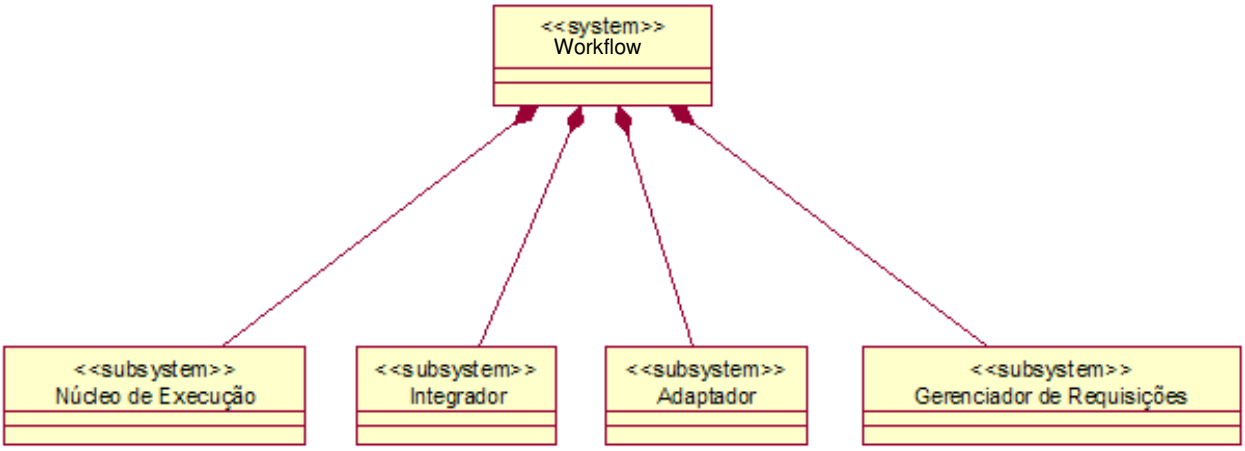

Figura 25 - Decomposição do *Workflow* em subsistemas

O *Workflow* é composto por quatro subsistemas conforme mostra a Figura 25: (1) Núcleo de execução; (2) Integrador; (3) Gerenciador de requisições e (4) Adaptador. O relacionamento entre esses subsistemas mostrado na Figura 26 utilizou os conceitos apresentados na seção 2.1.

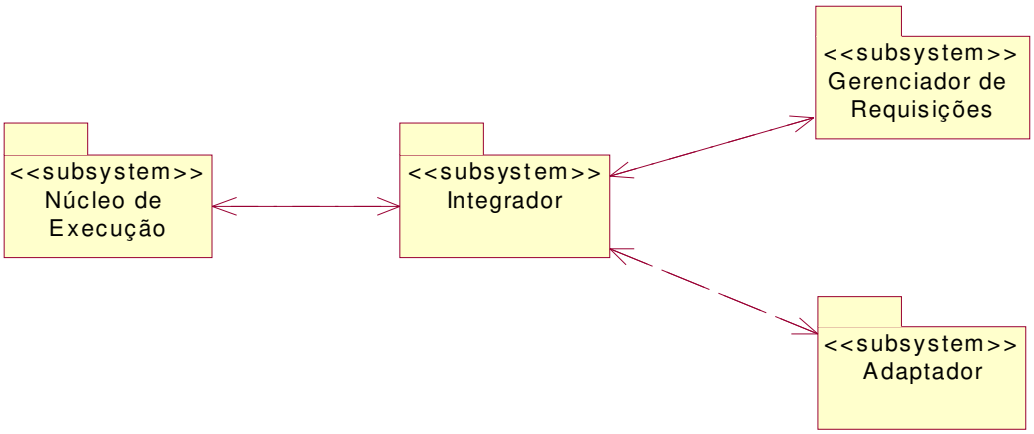

Figura 26 - Relacionamento entre os subsistemas do *Workflow*

O Núcleo de Execução de um subsistema é responsável por modelar e executar os processos de negocio da empresa. Esse subsistema oferece facilidades para definição de atividades e processos. Esses processos podem ser instanciados e é de responsabilidade do Núcleo de Execução controlar a execução dessas instâncias, garantindo que todas as restrições descritas na definição do processo sejam satisfeitas. São fornecidas interfaces para monitorar e interferir na execução dessas instâncias. O Núcleo de Execução requisita a execução de atividades por outros sistemas através do Integrador.

O Integrador é responsável por toda a lógica de integração de aplicações relativa a roteamento de mensagens e mapeamento de dados. Toda comunicação entre sistemas é feita através do integrador, conforme mostra a Figura 26. Quando o núcleo de execução deseja se comunicar com outro sistema, ele entrega a mensagem a ser enviada e a identificação do sistema com o qual deseja se comunicar (sistema destino). O integrador se responsabiliza por mapear e converter os dados da mensagem para o formato esperado pelo sistema destino, por localizar fisicamente esse sistema, e por entregar a mensagem convertida para que possa ser processada. O Integrador oferece facilidades para a descrição dos serviços fornecidos por cada sistema no formato utilizado pelo barramento de integração e para descrever graficamente o mapeamento e a conversão dos dados. Oferece também interfaces para monitorar as mensagens que são trocadas entre os sistemas.

O Adaptador é um subsistema utilizado para conectar sistemas externos ao barramento de integração. O adaptador é capaz de entender os protocolos e formatos de dados utilizados no barramento de integração, mapeando as informações recebidas para o protocolo de comunicação

e para o formato de dados específicos da aplicação externa. A parte que interage com o barramento de integração é comum a todo adaptador. No entanto, a parte do adaptador que interage com o sistema externo é específica para o protocolo de comunicação e formato de dados utilizado por cada sistema.

O Gerenciador de Requisições é um subsistema utilizado para consultar e tratar requisições de serviços manuais, ou seja, serviços não automatizados a serem executados por humanos. Esse subsistema fornece facilidades para a descrição dos serviços disponíveis e para consultar as requisições relativas a cada serviço, permitindo ao usuário consultar os dados relativos a cada requisição e fornecer os dados que devem ser retornados ao núcleo de execução assim que a execução do serviço manual for encerrada.

Para facilitar o entendimento do sistema, a Figura 27 mostra uma visão lógica da arquitetura do Workflow. Os relacionamentos entre os subsistemas Núcleo de execução, Integrador, Gerenciador de requisições e Adaptador foram omitidos para não sobrecarregar a imagem.

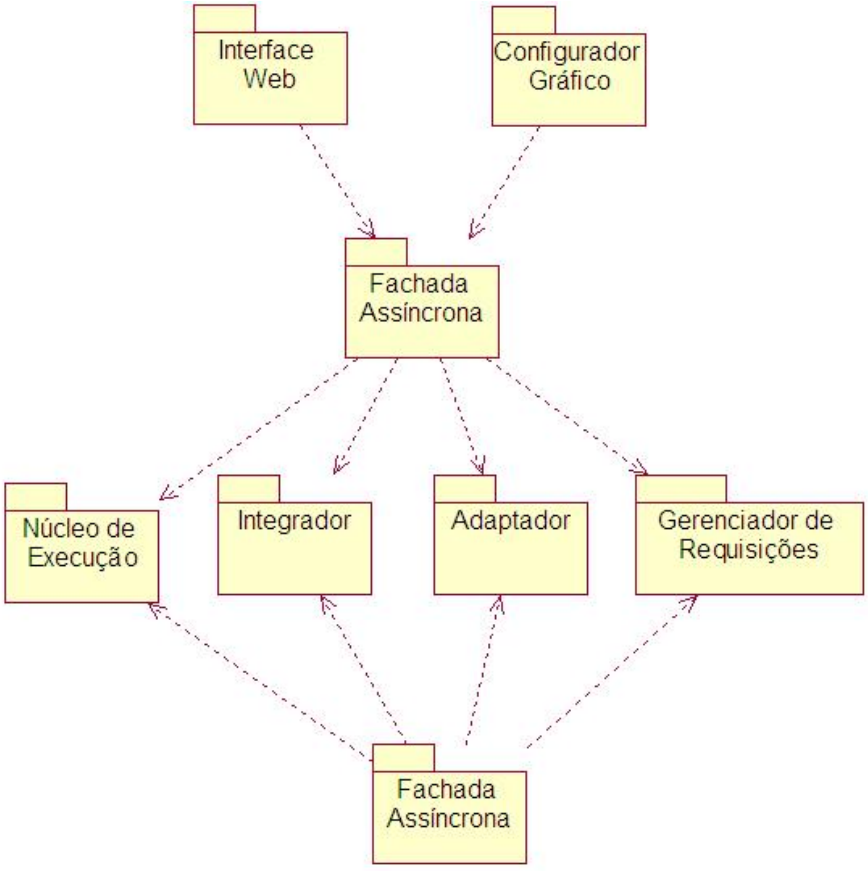

Figura 27 – Visão lógica da arquitetura do *Workflow*

Além dos subsistemas Núcleo de execução, Integrador, Gerenciador de requisições e Adaptador descritos nessa seção, o Workflow apresenta alguns componentes importantes para esse sistema mostrados na Figura 27 e descritos a seguir.

Interface Web – esse tipo de interface é utilizado principalmente para manter a descrição dos sistemas que serão integrados, sua localização física, e os serviços disponibilizados por esses sistemas. Através desse tipo de interface também é feito todo o monitoramento do integrador.

Configurador Gráfico – o configurador gráfico é uma ferramenta utilizada para descrever as regras a serem seguidas durante o mapeamento dos dados trocados entre dois sistemas. A descrição dos serviços dos sistemas envolvidos é utilizada como base para esse mapeamento. O configurador permite indicar a correspondência entre os dados, regras de transformação de tipo e formato, além de operações como concatenação e subdivisão dos dados.

Fachada síncrona – A fachada síncrona é o único ponto de entrada síncrono do Integrador. Através dessa fachada passam todas as requisições feitas pela interface Web e pelo mapeador gráfico. Essa fachada simplesmente delega a execução das requisições para o controlador apropriado, omitindo da camada de apresentação a complexidade da lógica interna do Integrador. A utilização da fachada ajuda a reduzir o acoplamento entre as camadas de negócio e de apresentação.

Fachada assíncrona – A fachada assíncrona é o único ponto de entrada assíncrono do Integrador. Através dessa fachada passam todas as requisições realizadas pelo núcleo de execução e por sistemas externos para transformação de dados e roteamento de mensagens.

Aplicações desenvolvidas em J2EE são normalmente decompostas em quatro camadas principais apresentados na seção 3.2: (1) camada cliente, (2) camada de apresentação, (3) camada de negócio e (4) camada de persistência. A Figura 28 representa uma arquitetura padrão que foi a adotada na implementação do *Workflow*:

Sobre as camadas do J2EE foi implementado o modelo MVC (*Modelo, Visão, Controle*) que foi apresentado na seção 2.5.1.5. No modelo MVC, o *Modelo* é responsável por manter o estado da aplicação e dos dados. O *Controle* atualiza o *Modelo* a partir das ações executadas pelo usuário através da apresentação (*Visão*). Também é responsável por selecionar o conteúdo a ser exibido para o usuário em resposta a uma solicitação.

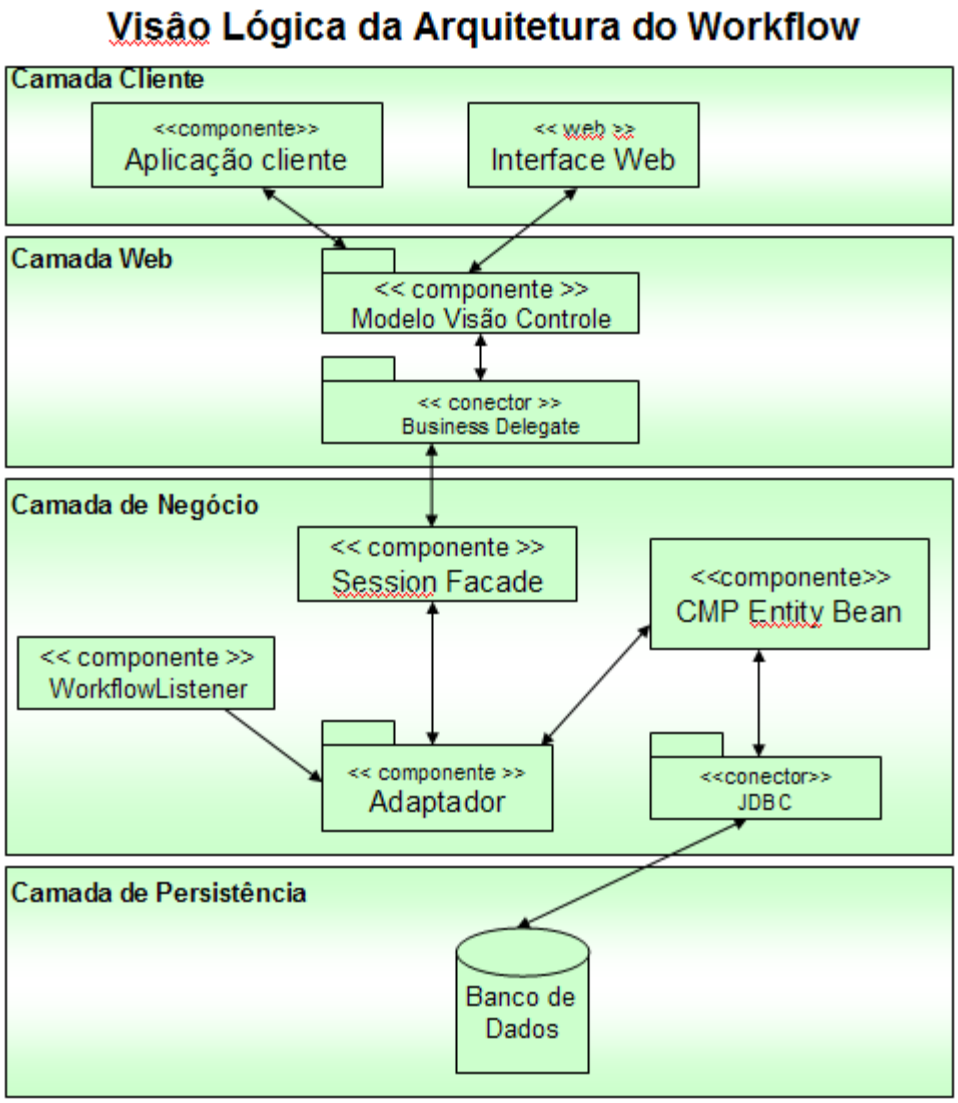

Figura 28 – Arquitetura Atual do Workflow

A camada de persistência foi implementada usando a tecnologia *Entity beans* gerenciadas pelo *Container* conforme vistos na seção 3.2. Essa arquitetura de persistência é o meio mais simples de se desenvolver porque se permite focar na lógica de negócios, delegando a responsabilidade da persistência para o c*ontainer*. Na seção 3.2 é mostrado que a tecnologia para a implementação de EJBs utiliza o container responsável por gerar a lógica necessária para armazenar o estado do *bean* automaticamente.

4.2.3 Visão de Implementação

Esta seção apresenta a implementação adotada para a solução mostrada pela visão lógica do sistema vista na seção 4.2.2 e são identificados os principais componentes, padrões de projeto e os *frameworks* utilizados e os relacionamentos entre esses elementos.

O *Workflow* foi implementado utilizando a tecnologia J2EE apresentada na seção 3.2, baseando-se na necessidade de atender aos requisitos como fornecer portabilidade, permitir múltiplos usuários, permitir segurança das informações trocadas entre as aplicações e escritos na linguagem Java que será mostrada nas subseções.

4.2.3.1 Classes de implementação do Workflow

A seguir são descritas as classes dos quatro subsistemas do *Workflow* apresentada na figura 27 sendo utilizada a notação *Unified* Model Language (UML) desenvolvida por Grady Booch, James Rumbaugh e Ivar Jacobson [Boo99].

Subsistema Núcleo de execução

O núcleo de execução do *Workflow* foi desenvolvido com base nas classes apresentadas na Figura 29. Através delas torna-se possível realizar a criação de definições de processos e atividades, e a partir dessas definições, criar e executar instâncias de processos. Inicialmente, para que uma definição de processo possa ser criada, é preciso que existam definições de atividades (*ActivityDefinition*) para compor os processos. Essas definições de atividades foram descritas na seção 4.1. Uma definição de processo é composta por um conjunto de definições de atividades (uma definição de atividade associada a um processo é chamada "*ProcessActivity*"), uma lista de arestas (*ProcessEdge*) e uma lista de variáveis (*ProcessVariable*). As arestas representam as dependências entre as atividades do processo. As variáveis são utilizadas para armazenar o estado interno do processo e são utilizadas para troca de informação entre as atividades e na criação de condições associadas às arestas. Uma condição associada a uma aresta especifica uma pré-condição para que a próxima atividade seja executada.

Quando uma definição de processo é instanciada para ser executada, são criadas a instância do processo (*ProcessInstance*), instâncias das atividades do processo (*ActivityInstance*), instâncias das arestas do processo (*EdgeInstance*) e instâncias das variáveis do processo (*VariableInstance*). Tanto instâncias de processo quanto instâncias de atividade possuem um ciclo de vida que é representado por classes de histórico (*ProcessInstanceHistory* e *ActivityInstanceHistory*, respectivamente). Essas classes são utilizadas para representar toda mudança de estado ocorrida ao longo da execução das instâncias.

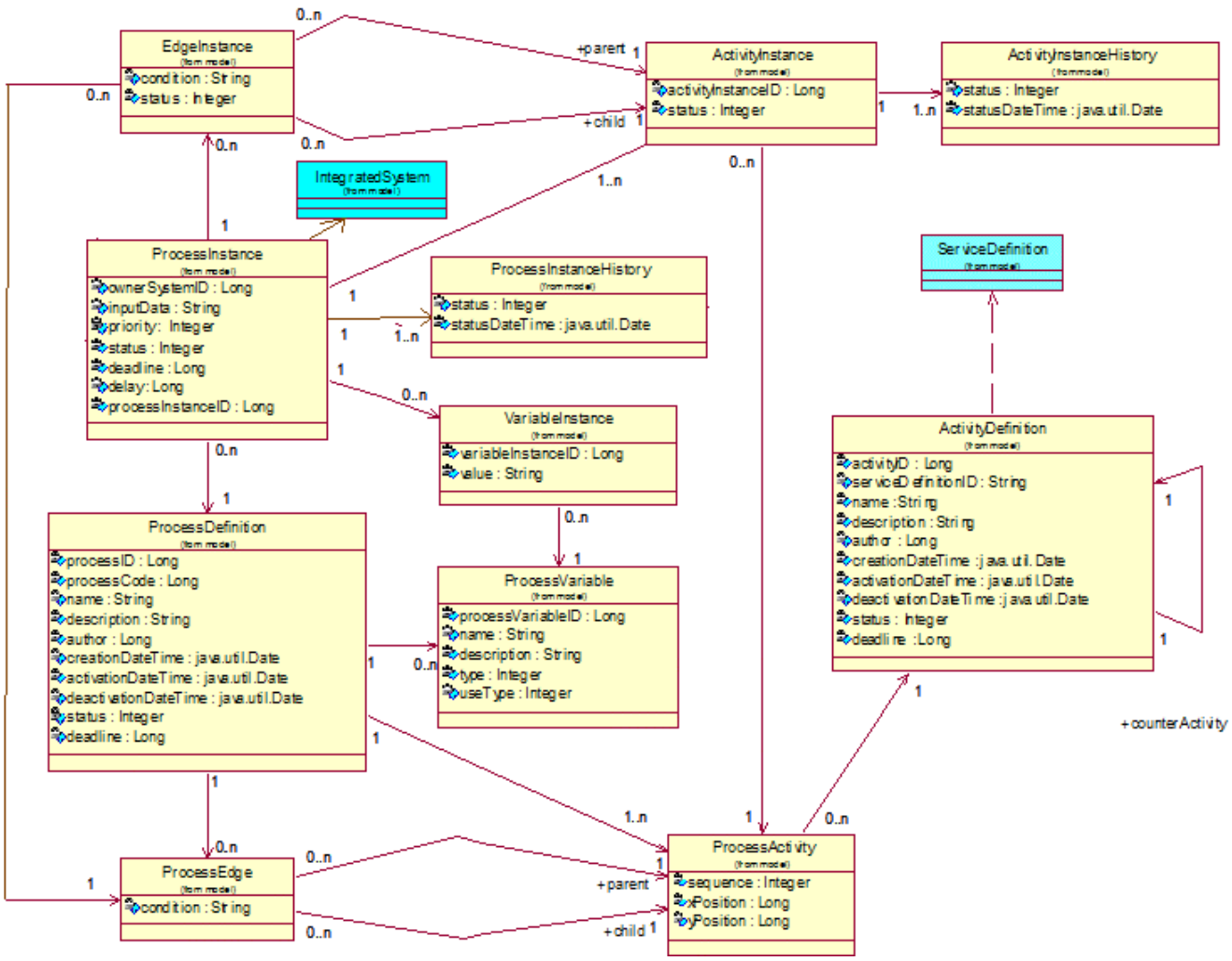

Figura 29 - Classes persistentes do Núcleo de Execução

Subsistema Integrador

A Figura 30 apresenta as classes persistentes do Integrador e seus principais atributos. Essas classes são utilizadas para descrever os sistemas a serem integrados (*SystemVO*), as filas de mensagens utilizadas para acionar esses sistemas de forma assíncrona (*QueueVO*) e a lista de serviços oferecidos por cada sistema (*ServiceVO*). Cada serviço é descrito por dois *XML Schemas*, um que descreve os dados requeridos para acionar o serviço e outro que descreve os dados retornados pelo serviço. O acionamento dos serviços é feito através de uma mensagem (*MessageVO*).

As regras para conversão dos dados são representadas pelo *DataConvertionVO*. Essa classe possui uma lista de *DataConvertionElementVO*, onde cada elemento de conversão descreve a conversão de um campo (*DataConvertionFieldVO*) do *XML Schema* de saída do

sistema que acionou o integrador para um campo do *XML Schema* de entrada do serviço acionado por esse sistema.

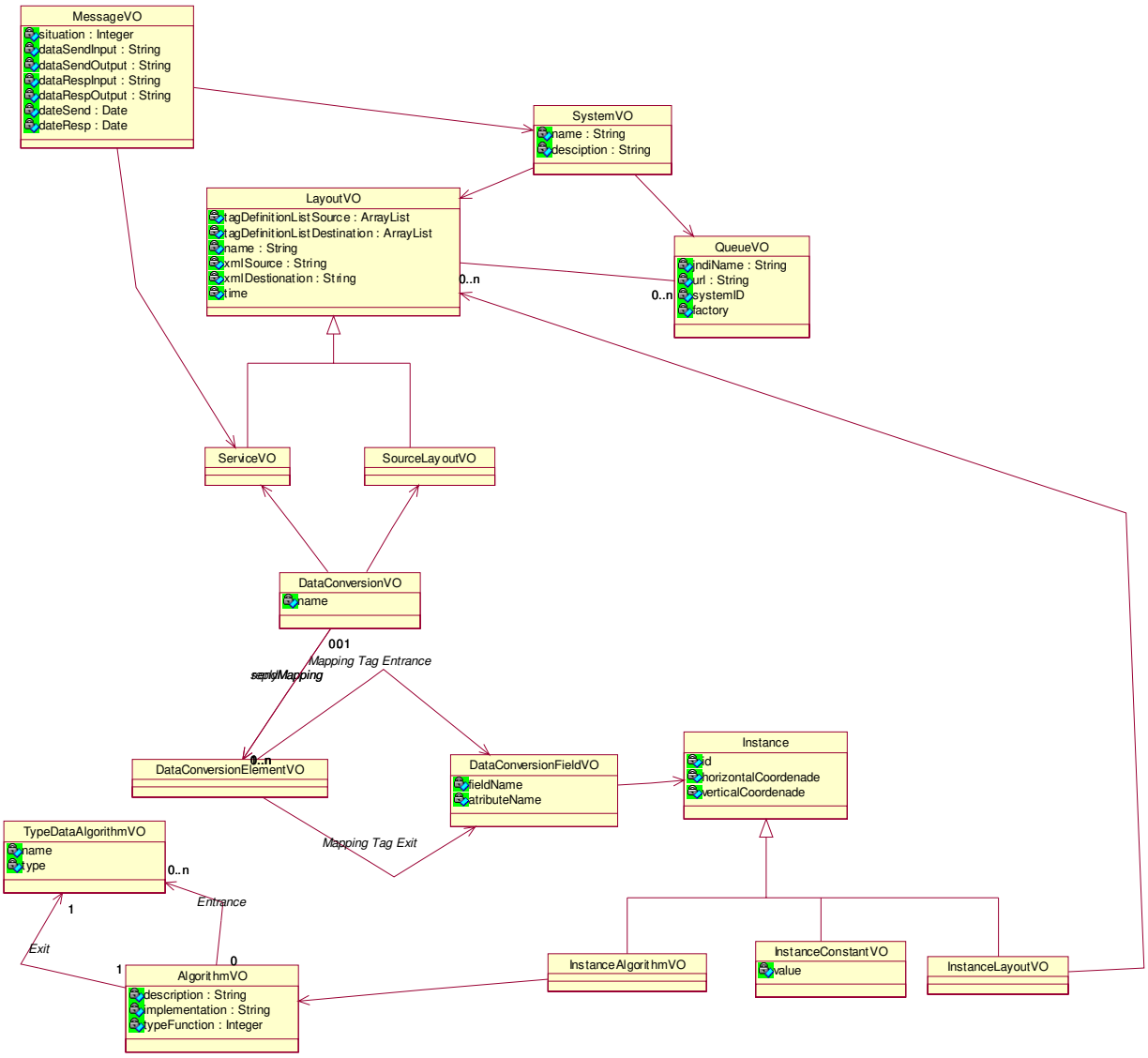

Figura 30 - Classes persistentes do integrador

Subsistema Adaptador

A Figura 31 apresenta as classes persistentes de um Adaptador projetado para comunicar-se com os sistemas adaptados via tabelas de banco de dados. Como pode ser observado, as

classes de domínio desse tipo de adaptador são muito similares à parte das classes de domínio do integrador. Essa similaridade deve-se ao fato de que o mapeamento dos dados de um XML para tabelas é muito similar ao mapeamento entre XMLs utilizado pelo integrador.

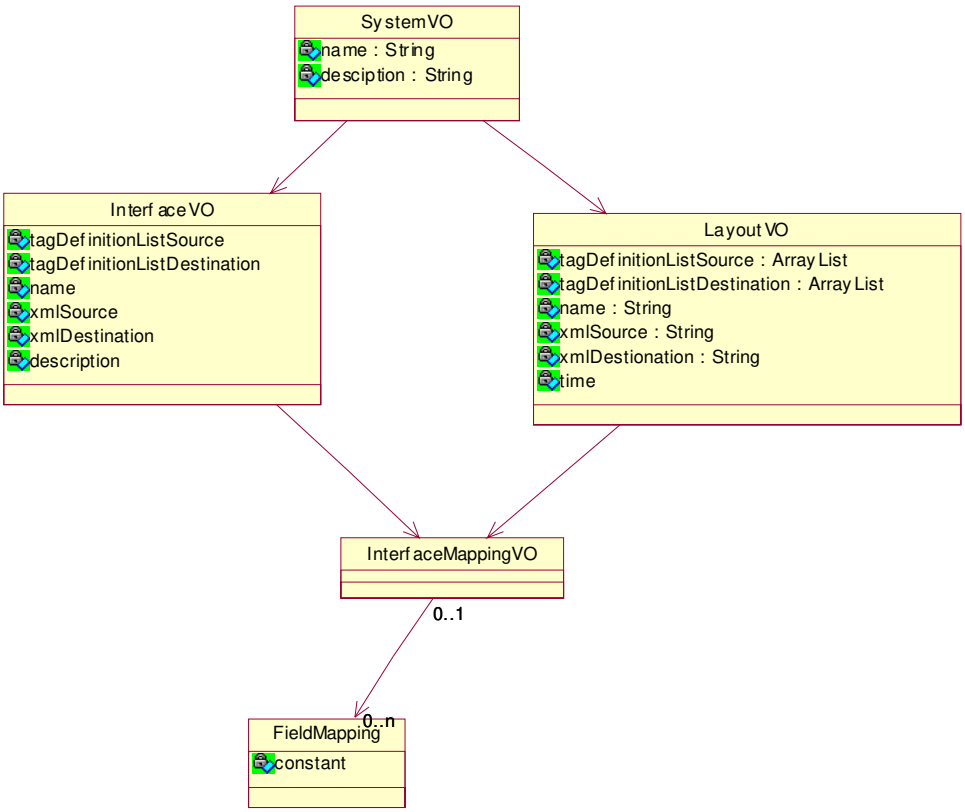

Figura 31 – Classes persistentes do adaptador

Subsistema Gerenciador de Requisições

O Gerenciador de Requisições do *Workflow* foi desenvolvido com base nas classes de domínio apresentadas do diagrama de classes mostrado na Figura 32.

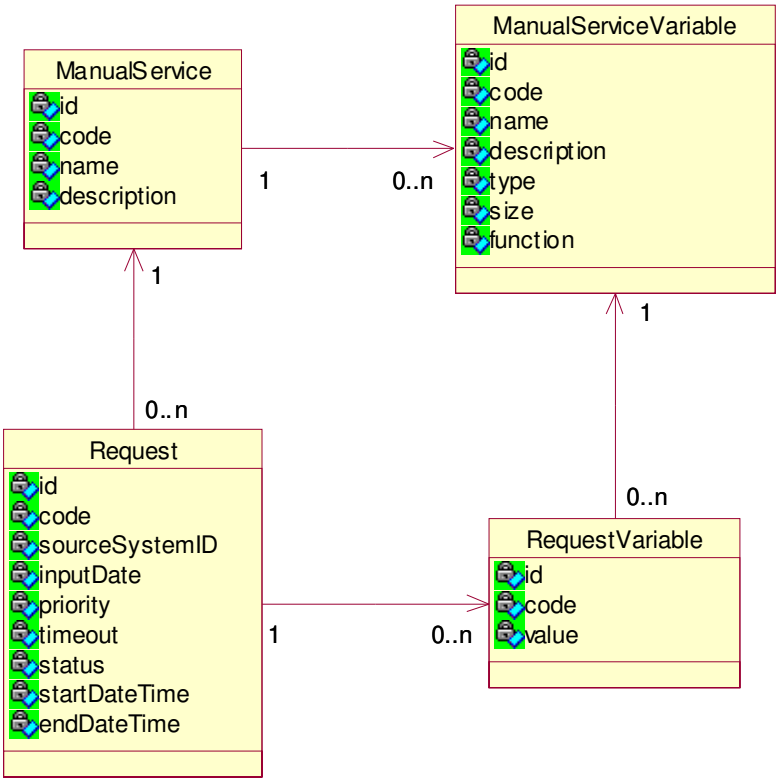

Figura 32 - Classes persistentes do Gerenciador de Requisições

Essas classes representam a descrição dos serviços manuais (*ManualService*) suportados pelo gerenciador e das requisições (*Request*) feitas pelo núcleo para execução desses serviços. A definição de um serviço manual tem associada a si uma lista de definições de variáveis (ManualServiceVariable). Essas definições determinam o nome, descrição, tipo da variável (numérico, texto, data), sua função (entrada/saída), e o tamanho máximo permitido para os valores das variáveis.

As requisições enviadas pelo núcleo têm associadas a si instâncias de variáveis (RequestVariable), de acordo com a definição dos serviços solicitados. As variáveis de entrada são utilizadas para armazenar os dados enviados pelo núcleo para a execução do serviço. As variáveis de saída são utilizadas para armazenar as informações informadas pelos técnicos quando da finalização da execução das requisições. Adicionalmente, para cada requisição é possível armazenar o seu status, designar, sua prioridade, um tempo máximo para o tratamento da requisição, e as datas/horas de início e fim do tratamento da requisição.

4.2.3.2 Atendimento dos Requisitos Não-funcionais

Da maneira que foi implementado o Workflow, são atendidos todos os requisitos funcionais e não-funcionais detalhados nesta seção e mostrados na Tabela 2.

Tabela 2 – Requisitos Não-funcionais atendidos pelo Workflow

Requisitos Não-Funcionais	Tecnologia	Arquitetura atende?
Portabilidade	J2EE	Sim
Uso de frameworks	J2EE	Sim
Desempenho	Não especificado	Não especificado
Facilidade de Uso	J2EE	Sim
Interfaces web compatíveis com Netscape e Internet Explorer	J2EE	Sim
Log de execução	J2EE	Sim
Protocolo de comunicação	J2EE	Sim
Segurança das informações trocadas entre as aplicações	J2EE	Sim
Suporte à comunicação assincrona	J2EE	Sim
Independência de Sistema de Gerenciamento de Banco de Dados	J2EE	Sim

Apesar de ter atendido todos os requisitos funcionais e não-funcionais solicitados pelo cliente, o sistema apresentou problemas de desempenho e o detalhamento deste problema será descrito na próxima seção.

4.3 Análise do problema e solução proposta para a arquitetura

Essa seção está dividida em duas subseções, a primeira subseção refere-se à análise do problema e está relacionada com os passos 1 ao 5 do processo de avaliação de arquitetura proposto por Tesoriero [Tes02]. A segunda subseção refere-se aos passos 6 e 7 do mesmo processo.

4.3.1 Análise do problema

Para que pudessem ser feitos os testes e a avaliação do desempenho do Workflow, foram analisadas medidas de *software*, orientadas a tempo de processamento utilizando-se do processo de avaliação proposto por Tesoriero [Tes02]. Após a análise dessas medidas, verificou-se que à medida que poderia resolver esta questão é a medida de tempo de resposta do sistema, ou seja, analisar quantas requisições o Workflow conseguiria processar em menor período de tempo. Como durante a fase de requisitos não havia sido estabelecido um valor de desempenho

necessário para atender as necessidades do cliente, o arquiteto projetou a arquitetura para que fosse realizada 20 atividades por minuto baseando-se na sua experiência com esse tipo de aplicação. Esse valor foi utilizado como indicador de desempenho, porém ao se iniciar os testes de carga no sistema, percebeu-se que o processamento das atividades estava abaixo do valor esperado, por isso, foi estabelecido medidas para avaliar a quantidade de atividades processadas por minuto. Essas medidas foram aplicadas desde a camada do cliente até a camada de persistência, sendo cronometradas cada fase de cada atividade. As medidas utilizadas foram: tempo de cada atividade, se a atividade esta aberta, se esta executando ou se esta concluída. Quando a atividade está aberta, o sistema abre uma requisição nova e envia essa requisição para a fila de processamento. Após isso, inicia a fase de execução. O Workflow envia a mensagem através da fila e em seguida estabelece uma comunicação com outros sistemas para que os mesmos possam processar a informação solicitada. Esses sistemas enviam as mensagens processadas para uma fila de retorno do workflow, onde o mesmo recebe as mensagens e finaliza o processamento, alterando o status para concluída. É importante considerar que o Workflow foi implantado em um sistema centralizado, ou seja, em um único servidor devido a indispobilidade de recursos por parte do cliente.

Aplicando-se o processo de avaliação proposto por Tesoriero apresentado na seção 2.6, têm-se as seguintes informações em relação a esse Caso de Estudo:

Passo 1 – Selecionar a perspectiva da avaliação

Conforme já citado anteriormente, a perspectiva da avaliação é verificar qual o tempo de resposta de cada atividade processada pelo sistema.

Passo 2 – Definir diretrizes e medidas

Para que seja analisado o tempo de cada atividade são aplicadas medidas como hora da abertura de cada atividade, quantidade de atividades abertas, quantidade de atividades em execução, quantidade de atividades já concluídas. Exemplo dos atributos a serem medidos são apresentadas na Tabela 3.

Tabela 3 - Diretrizes e medidas para avaliar a arquitetura atual

Processamento	Número de Atividades		
Tempo (em min)	Aberta	Executando	Concluída

Passo 3 – Definir a arquitetura planejada

Não existe uma arquitetura planejada, existe apenas a arquitetura atual.

Passo 4 – Identificar a arquitetura atual

Para definir esta arquitetura é preciso inicialmente avaliar o gargalo do sistema para então verificar em qual camada da arquitetura deverá ser feita a alteração da implementação. Esta arquitetura pode ser vista na Figura 28.

Passo 5 – Identificar desvios da arquitetura

Para identificar os desvios foi utilizada a técnica do uso da opinião de especialistas quando uma arquitetura atende ou não determinado requisito, foi realizada em 10 de Março de 2005 uma pesquisa envolvendo arquitetos (em média, com 5 anos de experiência em arquitetura e desenvolvimento de *software*) de um Centro de Pesquisa no Brasil. Foi apresentava uma matriz de risco citada no Capitulo 1 na seção 1.2, e apresentada aos entrevistados que deveriam colocar qual arquitetura apresentava maior risco e pontuá-la de acordo com a sua experiência de trabalho. O resultado pode ser visto na Tabela 4:

Tabela 4 – Pesquisa com especialistas em arquitetura

Riscos	Arquiteto 1		Arquiteto 2		Arquiteto 3	
	PGB	PGC	PGB	PGC	PGB	PGC
Desempenho	0.3	0.7	0.8	0.2	0.2	0.9
Manutenibilidade	0.5	0.1	0.2	0.8	0.6	0.3
Facilidade de Implementação	0.5	0.1	0.2	0.8	0.5	0.2
Uso de recursos de banco de dados	0.2	0.8	0.2	0.2	0.9	0.3
Uso de recursos do Container	0.1	1	0.1	0.9	0.1	1
Flexibilidade de outras formas de persistência como LDAP e XML	1	0.3	1	0.2	0.1	0.9
Flexibilidade de banco de dados	0.5	0.1	0.2	1	0.7	0.2
Totais	3.1	3.1	2.7	4.1	3.1	3.8

Legenda
0 => significa nenhum risco
1 => significa grande risco
PGB => Persistência Gerenciada pelo Bean
PGC => Persistência Gerenciada pelo Container

Analisando a Tabela 4, pode-se observar que as medidas escolhidas para a pesquisa puderam fornecer informações importantes para que fosse avaliado, juntamente com a experiência dos entrevistados, o modelo de arquitetura com persistência gerenciada pelo Bean que apresentasse o menor risco para um determinado projeto. Com base nessa pesquisa juntamente com os estudos realizados na seção 1.2, foi iniciado o processo de reavaliação da arquitetura do Workflow.

Foram realizados testes para todas as camadas da arquitetura do Workflow para verificar se todos os requisitos não-funcionais eram atendidos e durante os testes foi constatado pela equipe que o sistema apresentava problemas de desempenho no componente de persistência da camada de negócio. Para a realização dos testes foi utilizada a ferramenta JMeter [Jme99] e os resultados dos testes para a camada de persistência são apresentados na Tabela 5.

De acordo com a Tabela 5, no tempo 0:00 tem-se 6000 atividades abertas, 0 atividades em execução e 0 atividades concluídas. Após 5 minutos, 15 atividades estão em execução e 87 são concluídas, totalizando 5898 atividades abertas restantes. Após 20 minutos, 18 atividades estão em execução e 280 são concluídas totalizando 5702 atividades abertas restantes resultando em 280 / 20 min = 14 atividades por minuto.

Tabela 5 - Diretrizes e medidas para avaliar a arquitetura atual com valores iniciais

Processamento	**Número de Atividades**		
Tempo (em min)	Aberta	Executando	Concluída
0	6000	0	0
5	5898	15	87
10	5824	14	162
15	5783	17	200
20	5702	18	280

280 / 20 min = 14 atividades por minuto

O tempo de 14 atividades por minuto foi analisado pelo arquiteto do sistema e baseando-se no estudo de Tesoriero [Tes02] juntamente com os resultados obtidos através dos testes executados no sistema foi constatado que o problema de desempenho apresentado pelo Workflow estava na camada de negócio especificamente no componente responsável pela persistência da informação ilustrada na Figura 28 da seção 4.2.3. Porém esse problema não pode ser considerado como um desvio na arquitetura, mas deve ser considerado como uma falha na especificação dos

requisitos pela equipe responsável por não documentar requisitos implícitos como o requisito de desempenho. Esta falha fez com que a arquitetura projetada para o sistema não fosse a mais adequada.

4.3.2 Solução proposta para a arquitetura

Nessa seção serão apresentados os passos finais, do processo de Tesoriero [Tes02] para a avaliação da arquitetura.

Passo 6 – Formular recomendações de mudanças

Estas informações foram importantes para propor uma alteração na camada de persistência do *Workflow* sendo retirados os CMP *Entity beans* (persistência gerenciada pelo container), que foram substituídos por classes DAO que acessam diretamente o banco de dados via JDBC (persistência gerenciada pelo *bean*) conforme visto na seção 3.2. O componente da arquitetura que foi substituído foi indicado na Figura 33 através da seta larga.

Passo 7 – Verificar mudanças arquiteturais

Efetuando os testes novamente contemplando as alterações na camada de persistência conforme recomendado no item 6, percebeu-se os seguintes resultados:

Foram inseridas a mesma quantidade de atividades (ou seja, 6000 atividades) e as mesmas medidas de desempenho como o tempo de processamento e o *status* da atividade (aberta, executando e concluída). Com base nestas informações, pode-se ver os resultados obtidos através da Tabela 6.

Tabela 6 - Tempo de processamento de atividades no Workflow com alteração na camada de persistência da informação.

Processamento	Número de Atividades		
Tempo (em min)	Aberta	Executando	Concluída
0	6000	0	0
5	5879	18	103
10	5762	23	215
15	5634	19	347
20	5403	17	580

580 / 20 min = 29 atividades por minuto

Percebeu-se que com essa modificação eram processados 580 atividades em 20 minutos totalizando 29 atividades por minuto, o que significa um ganho de 107% (mais que o dobro) a mais de desempenho se comparado com o valor do desempenho mostrado anteriormente, pois 15 atividades a mais puderam ser processadas em um único minuto. Seguindo a recomendação do passo 6, foi feita a alteração na camada de persistência da arquitetura do *Workflow* como é mostrada na Figura 33, onde foi retirada a persistência através dos EJB Entity Beans (mostrada na Figura 28) e substituída pela persistência gerenciada pelo Bean através do padrão *Data Access Object (DAO).*

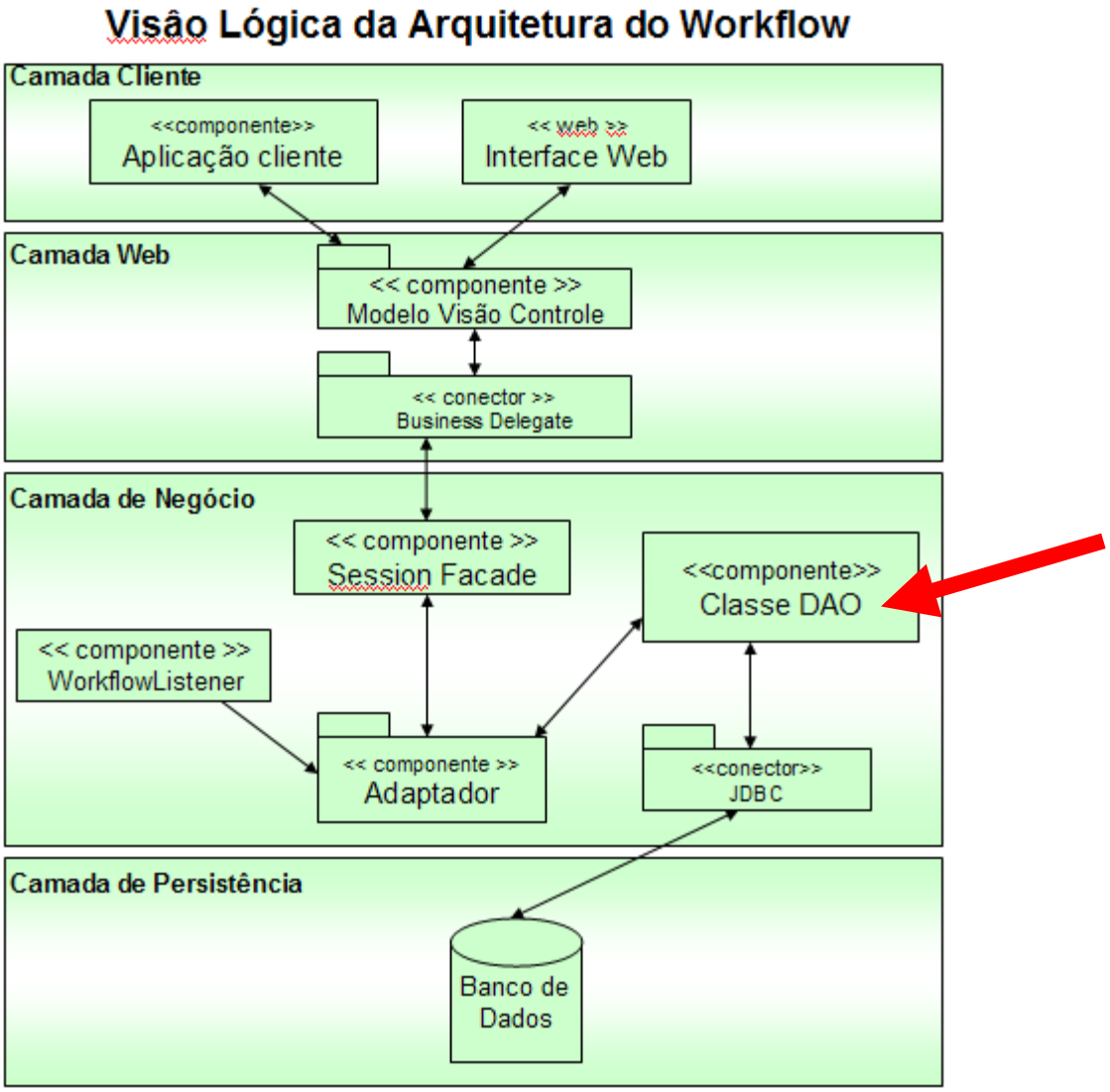

Figura 33 – Arquitetura proposta para o sistema de Workflow

Pode-se observar que devido à falha na especificação dos requisitos por não documentar um requisito implícito, o arquiteto tomou uma decisão baseada nos requisitos não-funcionais explícitos sem se preocupar com o desempenho do sistema. Pode-se perceber que também que não houve alteração na arquitetura como um todo, apenas foi necessário alterar componentes da camada de persistência desta arquitetura apresentada acima para que o problema de desempenho observado durante os testes fosse solucionado de maneira a atingir a satisfação do cliente.

4.4 Considerações finais

Nesse capítulo foi apresentado o sistema de Workflow, suas características, seus requisitos funcionais e não-funcionais, sua arquitetura, problemas encontrados no desempenho e finalmente a proposta de uma arquitetura para solucionar o problema de desempenho do sistema. No próximo capítulo serão apresentadas as conclusões dessa dissertação e sugestões de trabalhos futuros.

5 Conclusões e Trabalhos Futuros

Escolher uma arquitetura que atenda todos os requisitos funcionais e não-funcionais de um sistema não é uma tarefa fácil, pois exige uma grande experiência prévia da equipe bem como uma grande análise destes requisitos para que seja cumprido o projeto com o custo, tempo e recursos disponíveis. Uma falha no processo de levantamento de requisitos descoberta posteriormente pode prejudicar toda a arquitetura do sistema, e quanto mais tarde essa falha for descoberta, maior será o custo para adequar ou recriar os componentes relacionados à arquitetura, Nesta dissertação, procurou-se mostrar as arquiteturas existentes, analisar medidas que avaliam os requisitos não-funcionais, conceituar a tecnologia J2EE, usar um estudo de caso real [Tes02] como base para aplicar o mesmo processo no projeto Workflow e mostrar através de medidas, os resultados obtidos da análise realizada obedecendo aos requisitos do sistema.

Para poder avaliar a opinião de especialistas quando uma arquitetura atende ou não determinado requisito, foi realizada uma pesquisa envolvendo arquitetos conforme descrito no item 5 da seção 4.3. Pode-se verificar que mesmo entre profissionais experientes houve diferenças quanto a melhor escolha para a arquitetura de um sistema, no tocante ao método de persistência dos dados da camada de negócio. Pode-se perceber que as opiniões são similares entre os dois métodos. Contudo a escolha de qual método deve ser usado deve se basear nos requisitos não-funcionais de um sistema, e não em preferências pessoais por determinada tecnologia. No caso especifico do Workflow, verificou-se que essa escolha não foi feita corretamente devido ao pequeno desempenho no processamento das informações. Uma remodelagem da arquitetura, no método de persistência da camada de negócios, permitiu que o desempenho fosse significativamente melhorado e o sistema passou a processar um maior número de requisições numa mesma parcela de tempo. Isso mostra que a analise correta dos requisitos não-funcionais é essencial no processo de se propor a melhor arquitetura para um dado projeto. O erro nessa analise pode conduzir a problemas tais como os identificados no estudo de caso do sistema de Workflow, pois os requisitos não-funcionais no tocante desempenho não foram suficientemente analisados e nem especificados pela equipe do projeto.

Para mensurar o desempenho da arquitetura escolhida pelo arquiteto no caso de estudo especificado no capítulo 4, foram necessários estudos detalhados sobre os requisitos do sistema bem como análise de medidas que poderiam auxiliar na comparação sobre usar-se ou não uma determinada arquitetura. Após a conclusão dos processos executados pelo Jmeter e a geração dos relatórios de desempenho gerados pela ferramenta e sua análise, pode-se concluir que no Caso de Estudo do Sistema de *Workflow* apresentado nessa dissertação, o modelo de arquitetura de persistência que utiliza classe para gerenciar os dados apresentou um ganho de 107% no desempenho do sistema se comparado ao desempenho apresentado com o uso de Entity Beans que gerencia a persistência através do container, garantindo assim a satisfação do cliente pelas necessidades terem sido atendidas.

Baseando-se em estudos tais como de Ali Babar [Bab98] já publicados a respeito de processos para avaliar a arquitetura como ATAM, SAAM, entre outros, verificou-se que um grande número de processos não menciona explicitamente medidas como parte integrante do processo. Considerando-se que o objetivo dessa dissertação é analisar a arquitetura através de medidas baseando-se em um processo de avaliação, optou-se pelo processo de Tesoriero [Tes02], pois a utilização de medidas faz parte integrante desse processo de avaliação de arquitetura.

A contribuição desse trabalho é apresentar um exemplo de aplicação de um processo de avaliação de arquitetura com ênfase na abordagem quantitativa utilizando um sistema de *Workflow* como estudo de caso.

Recomenda-se para trabalhos futuros, que sejam realizados testes deste estudo de caso (Workflow) considerando-se um ambiente distribuído mais efetivo, ou seja, poder-se-ia executar os mesmos testes de desempenho com várias máquinas atendendo o mesmo total de requisições, pois a tecnologia escolhida (no caso J2EE) foi projetada para se obtenha melhor desempenho em sistemas distribuídos para realmente comprovar a eficácia do tipo de persistência utilizado na arquitetura. Além dessa recomendação, pode-se procurar analisar uma solução de Workflow que atenda os mesmos requisitos em um ambiente .Net e fazer um comparativo entre as tecnologias.

6 Referências Bibliográficas

[Ale78] Alexander C., "A Pattern Language", Oxford Press, Oxford, R. Unido, 1978

[All94] Allan R., Garlan D., Formalizing Architectural Connection. In "Proc 16th International Conference on Software Engineering" 1994.

[Alu02] Alur D., Crupi J., Malks D., "Core J2EE Patterns Best Practices and Design Strategies", 2002

[Amb90] Ambriola V., Ciancarini P., Montangero C.. Software Process Enactment in Oikos. Proc Fourth ACM SIGSOFT Symposium on Software Development Environments, SIGSOFT Software Enginnering Notes, December 1990.

[And91] Andrews G. R., "Paradigms for Process Interaction in Distributed Programs". ACM Computing Surveys vol 23, no 1, March 1991.

[Bab98] Babar A., Liming Z., Jeffery R., "A Framework for Classifying and Comparing Software Architecture Evaluation Methods", University of New Wales, Australia, 1998.

[Bam02] Bambara A., "Sun Certified Enterprise Architect for J2EE", McGraw-Hill Companies, 2002.

[Bas94] Basili V. R., Caldiera G., Rombach, D. H., "The Goal Question Metric Approach," Encyclopedia of Software Engineering – 2 Volume Set Wiley, pp. 528-532, 1994.

[Bas98] Bass L., Clements P., Kazman R. (1998). "Software Architecture in Practice". Boston, Addison-Wesley.

[Bar98] Kazman R., Klein M., Barbacci M., Longstaff T., Lipson H., and Carriere J., "The Architecture Tradeoff Analysis Method", Proceedings of IEEE, ICECCS, 1998.

[Bar84] Barstow D. R., Shrobe H. E., Sandewall E. (eds). “Interactive Programming Environments”. McGraw-Hill 1984.

[Bat91] Batory D., Malley S. O., “The Design and Implementation of Hierarchical Software Systems Using Reusable Components”. TR 91-22, Dept. of Computer Science, University of Texas, Austin, June 1991.

[Boo86] Grady Booch. Object-Oriented Development. IEEE Tr. On Software Enginnering, February 1986, pp. 211-221.

[Boo99] Booch G., Rumbaugh J., Jacobson I, “The Unified Modeling Language User Guide”, Addison-Wesley, 1999.

[Car01] Carvalho A., Chiossi T., “Introdução a Engenharia de Software”, Editora da Unicamp, 2001

[Fea02] Feather, M.S., Cornford & K.A. Hicks (2002) Descoping; Proceedings of the 27th IEEE/NASA Software Engineering Workshop, Greenbelt, Maryland, Dec 2002. IEEE Computer Society.

[Fri85] Marek F., Older W., “The Architecture of the XMS DistributedFile System. IEEE Software, vol 2, no 3, May 1985 (pp. 21-29).

[Gar99] Garlan D., "Software Architecture: a Road Map". School of Computer Science Carnegie Mellon University, Pittsburgh, 1999.

[Gar93] Garlan D., Shaw M., “An Introduction to Software Architecture.” In Ambriola & Tortora (eds), Advances in Software Engineering and Knowledge Engineering, vol. II, World Scientifc Pub Co., 1993.

[Gof95] Gamma E., Helm R., Johnson R., Vlissides J., “Design Patterns: Elements of Reusable Object-Oriented Software”. Addison-Wesley 1995.

[Jac94] Jackson M., “Problems, Methods and Specializations (A contribution to the Special Issue on Software Engineering in the Year 2001)”. IEE Software Engineering Journal, November 1994.

[Jme99] http://jakarta.apache.org/jmeter

[Kaz94] Kazman R., Bass L., Abowd G., Webb M., “SAAM: A Method for Analyzing the Properties of Software Architectures”, Proceedings of the 16th International Conference on Software Engineering, 1994.

[Kaz97] Abowd G., Bass L., Clements P., Kazman R., Northrop L., Zaremski L., “Recommended Best Industrial Practice for Software Architecture Evaluation”, SEI, Carnegie Mellon University CMU/SEI-96-TR-025, 1997.

[Kaz00] Kazman R., Klein M., Clements P., “ATAM: Method for Architecture Evaluation”, SEI, Carnegie Mellon University CMU/SEI-2000-TR-004, 2000.

[Kip98] Kiper J., “Requirements, Architectures and Risks.”, Dept. of Computer Science and Systems Analysis. Miami University. Oxford, OH 45056

[Lan90] Lane T., “Studying Software architecture Through Design Spaces and Rules”. Carnegie Mellon University Technical Report, September 1990.

[Lau79] Lauer H., Ed. H. Satterthwaite. “Impact of MESA on System Design”. Proc Third Int`Conf. on Software Engineering, May 1979.

[Lin85] Linton M., “Distributed Management of a Software Database”. IEEE Software, vol 2 no 4, May 1985, pp. 62-69

[Lun97] Lung C., Bot S., Kalaichelvan K., Kazman R., “An Approach to Software Architecture Analysis for Evolution and Reusability”, Proceedings of CASCON, 1997.

[Mac99] Macedo N., Leite J., “Integrando Requisitos não-funcionais aos requisitos baseados em ações concretas.”, Anais do II Workshop Iberoamericano de Engenharia de Requisitos e Ambientes de Software, Alajuela, Costa Rica, Marzo, 1999.

[Nii86] Nii P., Blackboard Systems. AI Magazine 7(3):38-53 and 7(4):82-107.

[Olo99] Olof B., Bosh J., “Architecture Level Prediction of Software Maintenance”. University of Karlskrona/Ronneby, Third European Conference on Software Maintenance and Reengineering 03, 1999, Amsterdam, Netherlands.

[Orf98] Orfali R., Harkey, D. “Client/server programming with Java and CORBA”, Nova York, John Wiley and Sons, 1998, Caps 8, 11 14.

[Par72] Parnas D., “On the Criteria to be Used in Decomposing Systems into Modules”. Comm. ACM vol 15, December 1972.

[Pau85] Paulk M., “The ARC Network: A Case Study”. IEEE Software, vol 2 no 3, May 1985, pp. 62-69

[Pre87] Pressman R., “Software Engineering”, McGraw-Hill, 1987.

[Sha95] Shaw M., DeLine R., Klein D., Ross T., Young D., Zelesnik G., “Abstractions for Software Architecture and Tools to Support Them”. IEEE Tr. On Software Engeneering, May 1995.

[Sha96] Shaw M., “Some Patterns for Software Architectures”. PLoPD2 Pages: 252-269, 1996.

[Som04] Somerville I., “Software Engineering”. 6. ed. Addison-Wesley, 2004.

[Tes02] Tesoriero R., Lindvall M., Costa P., “A Process for Software Architecture Evaluation Using Metrics”. 27th Annual NASA Goddard Software Engineering Workshop (SEW-27’02) 12 05 – 12, 2002, Greenbelt, Maryland.

[Zhu04] Babar A., Zhu L., Jeffery R., “A Framework for Classifying and Comparing Software Architecture Evaluation Methods”, Australian Software Engineering Conference, Melbourne, 2004.