UNIVERSIDADE FEDERAL DO PARÁ
INSTITUTO DE CIÊNCIAS EXATAS E NATURAIS
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

Amanda Monteiro Sizo Lino

APLICAÇÃO DE ARQUITETURA DE *SOFTWARE* NO DESENVOLVIMENTO DE SISTEMAS DISTRIBUÍDOS EM JAVA

Belém
2011

Amanda Monteiro Sizo Lino

APLICAÇÃO DE ARQUITETURA DE *SOFTWARE* NO DESENVOLVIMENTO DE SISTEMAS DISTRIBUÍDOS EM JAVA

Dissertação de Mestrado apresentada para obtenção do grau de Mestre em Ciência da Computação. Programa de Pós-Graduação em Ciência da Computação. Instituto de Ciências Exatas e Naturais. Universidade Federal do Pará.
Área de concentração: Engenharia de Software.
Orientador Prof. Dr. Eloi Luiz Favero.

Belém
2011

Lino, Amanda Monteiro Sizo
Arquitetura de software no desenvolvimento de sistemas distribuídos em Java / (Amanda Monteiro Sizo Lino); orientador, Eloi Luiz Favero. – 2011.
115 p. il. 28 cm
Dissertação (Mestrado) – Universidade Federal do Pará. Instituto de Ciências Exatas e Naturais. Programa de Pós-Graduação em Ciência da Computação. Belém, 2011.

1. Arquitetura de software. 2. Padrões de software. 3. Engenharia de Software. I. Favero, Eloi Luiz, orient. II. Universidade Federal do Pará, Instituto de Ciências Exatas e Naturais, Programa de Pós-Graduação em Ciência da Computação. III. Título.

CDD 22.ed. 005.1

Amanda Monteiro Sizo Lino

APLICAÇÃO DE ARQUITETURA DE *SOFTWARE* NO DESENVOLVIMENTO DE SISTEMAS DISTRIBUÍDOS EM JAVA

Dissertação de Mestrado apresentada para obtenção do grau de Mestre em Ciência da Computação. Programa de Pós-Graduação em Ciência da Computação, Instituto de Ciências Exatas e Naturais. Universidade Federal do Pará.

Data da aprovação: Belém-Pa. 30-10-2011

Banca Examinadora

_____

Prof. Dr. Eloi Luiz Favero
(ORIENTADOR – UFPA)

_____

Prof. Dr. Sandro Ronaldo Bezerra Oliveira
(MEMBRO-UFPA)

_____

Prof. Dr. Mauro Margalho Coutinho
(MEMBRO-UNAMA)

VISTO:

_____

Prof. Dr. Sandro Ronaldo Bezerra Oliveira
(COORDENADOR DO PPGCC/ICEN/UFPA)

À minha filha Alice, pra quem desde sua chegada, dedico toda minha vida.

AGRADECIMENTOS

Agradeço a Deus por iluminar sempre meu caminho.

Ao orientador, Prof Dr. Eloi Favero, pela sua disposição, compreensão e por sua segura orientação na condução deste trabalho.

Ao Prof.Dr. Adagenor Ribeiro, que me presenteou com bons conselhos para escrita de trabalhos científicos e orientou no inicio da minha vida dentro do programa de mestrado.

Aos meus pais, Ruy e Dóris Sizo, por me proporcionar a educação e me apoiar em todos os momentos desta jornada.

Ao meu marido Adriano Del Pino Lino, pelo auxilio nas revisões deste trabalho.

A Empresa de processamento de dados do Pará, por me liberar para cumprir as disciplinas do mestrado.

Ao diretor do CTIC - UFOPA, Hélio Correa, por compreender minha ausência no trabalho para concluir o mestrado.

Aos meus familiares que sempre acreditaram na minha capacidade, em especial a minha avó Brizalinda Sizo, madrinha Linda Pontes e tia Sandra Monteiro por toda forca e apoio. E também ao meu tio Euclides, tia Cristina, primo Danilo e Kidinho, graças a eles consegui superar as leis de Murphy na hora de imprimir este trabalho.

Ao meu primo Adriano Sizo Pontes por ser um exemplo vivo de que precisamos lutar todos os dias para conquistar a vitória.

Aos meus amigos Ádamo Santana, Regiane Brito, Lorenna Góes, Anderson Costa, Steven Alexander e demais amigos que contribuíram para elaboração deste trabalho.

A minha amiga Adriana Teles meu agradecimento especial, por estar comigo desde o inicio da elaboração desta pesquisa.

E finalmente, aos professores Mauro Margalho e Sandro Bezerra por aceitarem participar desta banca de mestrado.

RESUMO

Arquitetura de *software* consiste na definição dos componentes de softwares, suas propriedades externas e seus relacionamentos com outros softwares, é um artefato crítico para a obtenção da qualidade esperada de um sistema, mais especificamente no caso de sistemas distribuídos onde os atributos de qualidade são muito importantes para o seu funcionamento adequado. Nas empresas de grande porte, existe uma crescente demanda para desenvolvimento de sistemas de forma ágil e eficiente nas mais variadas plataformas e isso incentiva a criação de uma arquitetura padrão para o desenvolvimento de sistemas distribuídos.

A ausência de uma arquitetura de desenvolvimento de *software* resulta em alguns problemas, como: falta de padronização no desenvolvimento; dificuldade para incorporar mudanças inesperadas de requisitos; limitações na especificação de requisitos funcionais; e dificuldade de manutenção.

Inserido no cenário de uma empresa pública de tecnologia da informação de grande porte, este trabalho busca contribuir com um artefato importante para o aprimoramento do desenvolvimento de sistemas em sua fábrica de *software*. Para tanto é proposto uma arquitetura de *software* fundamentada no padrão arquitetural MVC (*Model View Controller*) e baseada na plataforma JEE (*Java Enterprise Edition*), que é projetada para fornecer suporte ao desenvolvimento de aplicações distribuídas para *Web*.

Esta arquitetura de *software* tem como principal objetivo desenvolver sistemas e integrá-los de maneira distribuída a outros ambientes independe de plataforma ou linguagem. Esta proposta é implementada a partir da construção de um sistema de controle de acesso e avaliada por meio de entrevistas com os envolvidos no projeto.

Com base nos resultados alcançados com a aplicação da arquitetura no desenvolvimento de um sistema de controle de acesso, verificou-se: adição de performance no tratamento de objetos, interoperabilidade, reusabilidade, aumento de produtividade por meio de reuso de componentes e padronização do desenvolvimento.

PALAVRAS-CHAVES: Arquitetura de *software*, Padrões de projeto, Engenharia de *software*, Sistemas distribuídos.

ABSTRACT

Software Architecture assist on the definition of software components, external dependencies from other software, is a critical artifact to achieve the quality expectations of an automated solution, specifically in the case of distributed system where the variables are extremely important in order to get whole solution working well. In big companies, there is a big demand of software development to be done in an agile and efficient fashion which increases the need for a standardization of distributed-software development.

The absence of such architecture standards might result in some problems, like: Lack of development standards, difficulties to include ad-hoc modifications, limitations in the specification and complicated maintenance.

In the context of a big public company, with focus on Information Technology, this work tries to enhance the software development by introducing an important artifact within Software Houses. In order to achieve this it is proposed a standard architecture based on the design pattern MVC (Model View Controller) and the Java platform, which is designed to provide support for enterprise web-applications.

This software architecture has as a main goal developing systems and integrate them independently of the platform or language. This use-case targets a control management system and it was evaluated with interviews done the people involved in the project.

Based on the results, applied to an access-control system, the following observations were noted: Performance improvements when treating objects, interoperability, reusability, increase of productivity, all of these due to the reuse of components and standardization of the development.

KEYWORDS: Software architecture, Project patterns, Software engineering, Distributed systems.

LISTA DE ILUSTRAÇÕES

LISTA DE QUADROS

LISTA DE SIGLAS

ADD *Attribute Driven Design*
ADL *Architecture Description Language*
ANSA *Advanced Network Systems Archtecture*
API *Application Programming Interface*
ARID *Active Review for Intermediate Designs*
ATAM *Architecture Tradeoff Analysis Method*
CRUD *Create, Read, Update and Delete*
DAO *Data Access Object*
DTO *Data transfer object*
EJB *Enterprise Java Beans*
FAQ *Frequently Asked Questions*
GUI *Graphical User Interface*
HTML *Hypertext Mark-up Language*
IEC *International Electrotechnical Commission*
ISO *International Organization for Standardization*
JEE *Java Enterprise Edition*
JSF *Java Server Faces*
LabSQL Laboratório de Ensino e Aprendizagem de SQL
LDAP *Lightweight Directory Access Protocol*
MC Mapa Conceitual
MPS.BR Melhoria do Processo de Software Brasileiro
MVC *Model View Controller*
OMG *Object Management Group*
PHP *Hipertext PreProcessor*
POJO *Plain Old Java Objects*
RF Requisitos Funcionais
RMI *Remote Method Invocation*
RNF Requisitos Não-funcionais
RPC *Remote Procedure Call*
SAAM *Software Architecture Analysis Method*
SEI *Software Engineering Institute*
SGBD Sistema de Gerenciamento de Banco de Dados
SQL *Structured Query Language*
UFOPA Universidade Federal do Oeste do Pará
UFPA Universidade Federal do Pará
UML *Unified Modeling Language*
WEB Abreviação *de WWW (World Wide WEB)*
XML *Extensible Markup Language*

SUMÁRIO

CAPÍTULO 1
INTRODUÇÃO

O capítulo 1 apresenta uma visão geral da dissertação e os objetivos da pesquisa.

1.1 INTRODUCAO

Diversas empresas têm adotado o conceito de fábrica de software no intuito de desenvolver sistemas da forma mais produtiva e econômica, esse conceito refere-se ao processo estruturado, controlado e melhorado continuamente, considerando as abordagens da engenharia industrial orientadas para o atendimento a múltiplas demandas de natureza e escopo distintas, visando à geração de produtos de software (Fernandes, 2004).

Um dos grandes desafios das fábricas de software está em aprimorar o processo de desenvolvimento com o intuito de conceber seus produtos de maneira mais ágil e precisa. A forma seriada como estas fábricas têm construído *software*, reutilizando a mesma arquitetura e os componentes associados a esta, trazem vantagens substanciais como redução de custos de produção e do tempo para a conclusão do *software*. Segundo Bass (1997), esta é a idéia principal da abordagem denominada linha de produto de *software*: uma coleção de sistemas que compartilham um conjunto de características, construídas de um conjunto de artefatos[1] de *software* previamente desenvolvidos, estes artefatos incluem a arquitetura base e um conjunto comum de componentes para integrá-la.

Segundo a definição clássica de arquitetura apresentada por Shaw & Garlan (1996), uma arquitetura de *software* define o que é sistema em termos de componentes computacionais e os relacionamentos entre estes componentes. Adicionalmente, descreve a estrutura técnica, limitações e características dos componentes, bem como as *interfaces* entre eles. Para (Pressman, 2006) uma arquitetura de *software* pode ser definida como uma estrutura que abrange os componentes de *software*, propriedades externamente visíveis desses componentes e as relações entre estes componentes.

Conforme Varoto (2002), a ausência de uma arquitetura de desenvolvimento de *software* consistente resulta em alguns problemas, como:

- Falta de padronização no desenvolvimento, utilizando diferentes tecnologias como solução para o desenvolvimento, sem analisar o contexto, o domínio e a adequação ao problema: levando, assim, à ausência de um facilitador na construção de sistemas que derivam de uma mesma linha de produção;

[1] É o produto de uma ou mais atividades dentro do contexto de desenvolvimento de um software ou sistema.

- Inflexibilidade para incorporar mudanças inesperadas de requisitos e futuras evoluções, relegando práticas de reutilização e resultando na demora na construção do *software*;
- Limitações na especificação de requisitos funcionais apontados pelos usuários, desconsiderando aspectos implícitos de qualidade, tais como: portabilidade, extensibilidade, desempenho e segurança;
- Dificuldade de manutenção, à medida que avança o desenvolvimento do sistema, novas partes vão sendo implementadas e, conseqüentemente, a manutenção vai se tornando cada vez mais complexa. Aproveitar-se de estruturas ou componentes já prontos garante mais robustez para o sistema além de aumentar o grau de abstração, escondendo detalhes do problema que podem ser substituídos por uma solução pronta.

Uma arquitetura se comporta como facilitador na construção de sistemas que derivam de uma mesma linha de produção (Queiroz & Braga, 2008). Existem muitos sistemas que derivam de uma mesma classe de problemas e possuem características propagáveis, podendo fazer uso dos mesmos processos e artefatos, o que promove a reutilização dos conceitos e funcionalidades em comum.

Diversas arquiteturas de *software* são propostas para o desenvolvimento de sistemas, para as mais diversas plataformas. Apesar de uma grande quantidade permanecer privada e pertencer a grandes organizações, diversas arquiteturas estão publicamente disponíveis (Carvalho, 2011). Como é o caso da plataforma de desenvolvimento Pinhão (Celepar, 2011) e da plataforma de desenvolvimento Demoiselle (Serpro, 2011), que são arquiteturas aplicadas a sistemas para o governo. No contexto de arquiteturas para linha de produtos, pode-se citar a arquitetura para sistemas de *workflow* (Lasilha, 2002), para sistemas de controle de satélite (Thomé, 2005), para arranjos produtivos locais (Almeida, 2005), entre outros.

Como se pode observar existem diversas pesquisas na obtenção de arquiteturas para diferentes domínios, porém a crescente demanda para desenvolvimento de novos sistemas de forma ágil e eficiente nas mais variadas plataformas, estimulou a necessidade da criação de uma arquitetura para o desenvolvimento de sistemas distribuídos.

A arquitetura proposta foi aplicada em um cenário de fábrica de software em uma empresa pública de tecnologia da informação de grande porte. Culturalmente, esta empresa apresenta fortes aspectos políticos e, por conseguinte alta rotatividade de tecnologia, a falta de

padronização nas ferramentas reflete em sistemas desenvolvidos nas mais variadas plataformas. A fim de minimizar os problemas oriundos pela falta de padronização no desenvolvimento, definiu-se pela construção de sistemas ou serviços *web*, na plataforma Java que pudessem ser acessados de qualquer outra plataforma computacional.

Dentro deste contexto a criação da arquitetura, veio suprir a necessidade de prover um meio de reutilização de componentes e arcabouços no ciclo de construção de sistemas na fábrica de *software* da empresa, viabilizando e uniformizando os procedimentos, diminuindo prazos e aumentando a produtividade, proporcionando a obtenção de melhores resultados com menores custos.

Um sistema distribuído é definido como sendo aquele no qual os componentes de *hardware* ou *software*, localizados em computadores interligados em rede, se comunicam e coordenam suas ações apenas enviando mensagens entre si (Coulouris, 2005). O domínio de sistemas distribuídos deriva de uma mesma classe de problema, pois apresenta requisitos comuns de qualidade, como: concorrência, *interface* pública, compartilhamento de recursos, escalabilidade e transparência, características estas presentes na maioria dos sistemas modernos.

Os desafios advindos da construção de sistemas distribuídos são a heterogeneidade de seus componentes, ser um sistema aberto – o que permite que componentes sejam adicionados ou substituídos –a segurança, a escalabilidade – a capacidade de funcionar bem quando o número de usuários aumenta – o tratamento à falhas, a concorrência de componentes e a transparência (Coulouris, 2005).

Neste contexto, este trabalho propõe uma arquitetura para implementação de sistemas dentro do domínio de sistemas distribuídos, que segue o padrão arquitetural em camadas, *Model-View-Controller* (MVC) que, por definição, desacopla as camadas provendo maior flexibilidade. A arquitetura é fundamentada em alguns princípios de GoF (Gamma *et al.*, 2003), como *factory* e *abstract service*, além de padrões como *business object* e DAO, estabelecidos em (Alur, 2003). Também foram integrados nessa arquitetura os *framework*s *Jboss-Seam*, EJB3 e *Hibernate*, que adicionalmente auxilia o trabalho do desenvolvedor e viabiliza o processo proposto neste trabalho.

Esta proposta é implementada a partir da construção de um sistema de controle de acesso e avaliada por meio de entrevistas com os envolvidos no projeto. A avaliação forneceu

resultados quantitativos e qualitativos quanto ao cumprimento dos requisitos não-funcionais especificados pelo cliente, como eficiência e portabilidade.

A partir da análise dos resultados obtidos, foram aplicados ajustes na arquitetura de modo que oferecessem melhoria em relação ao atributo de eficiência, pois este estava diretamente relacionado ao requisito não funcional exigido pelo cliente. Assim, a arquitetura resultante foi testada na construção de um serviço de geração e avaliação de mapas conceituais que é integrada ao ambiente de aprendizagem LabSQL[2] (Laboratório para ensino e aprendizagem de SQL), desenvolvido por (Lino *et. al.*, 2007).

A principal contribuição desse trabalho é a definição de uma arquitetura de linha de produto para sistemas distribuídos. Esta arquitetura foi utilizada para facilitar o processo de produção de diferentes sistemas distribuídos que possuem características comuns, mas que também possuem aspectos diferentes de acordo com a necessidade da organização e foi aplicada no desenvolvimento de mais de 30 sistemas.

1.2 RELEVÂNCIA DO TRABALHO

A disseminação do uso das redes de computadores e a ampla difusão da Internet têm incitado os estudos referentes ao desenvolvimento de sistemas distribuídos. Sistemas modernos possuem características típicas de sistemas distribuídos como: recursos compartilhados, desempenho, tolerância a falhas, transparência, concorrência (Coulouris, 2005). Por isso é relevante pesquisas na concepção de artefatos que auxiliem o desenvolvimento dessas aplicações.

Arquitetura de *software* é um artefato crítico para a obtenção da qualidade esperada de um sistema e mais especificamente no caso de sistemas distribuídos, os atributos de qualidade são muito importantes para o funcionamento adequado. Mesmo sabendo que a arquitetura de *software* por si só não é capaz de garantir qualidade a um sistema, ela fornece a base necessária para que os fatores de qualidade esperados sejam satisfeitos.

Uma vez implementada, a arquitetura proposta oferece um resultado de melhoria dentro do contexto de sistemas distribuídos e abre oportunidades de pesquisa para outras famílias de sistemas.

[2] O LabSQL é um ambiente virtual de ensino e aprendizagem da linguagem de manipulação de banco de dados SQL, utilizado nas disciplinas de banco de dados nos cursos de graduação e pós-graduação. Essa ferramenta foi desenvolvido por Adriano Del Pino Lino, disponível em www.ufpa.br/labsql

1.3 OBJETIVOS DA DISSERTAÇÃO

O principal objetivo desta dissertação é projetar uma arquitetura de *software* para desenvolvimento de sistemas distribuídos, na plataforma *java*, a fim de que possam ser acessados de qualquer lugar independente de plataforma ou linguagem. Para atingi-lo teremos os seguintes objetivos secundários:

- Apresentar a importância e os conceitos relevantes na área da arquitetura de *software*;
- Apresentar um resumo a cerca das arquiteturas de *software* para linhas de produtos encontradas na literatura;
- Projetar uma arquitetura de *software* para sistemas distribuídos em Java;
- Aplicar a arquitetura proposta no desenvolvimento de um sistema de controle de acesso distribuído;
- Avaliar a arquitetura junto aos envolvidos no projeto;
- Implementar um segundo protótipo de acordo com a arquitetura resultante da avaliação para gerar uma evidencia de implementação e servir de referência para trabalhos futuros.

1.4 TRABALHOS CORRELATOS

Diversas arquiteturas de *software* são propostas para o desenvolvimento de sistemas, para as mais diversas plataformas. Apesar de uma grande quantidade permanecer privada e pertencer a grandes organizações, diversas arquiteturas estão publicamente disponíveis (Carvalho, 2011).

Tiboni (2009) apresenta a plataforma *Demoiselle*, desenvolvido pelo SERPRO (Serviço Federal de Processamento de Dados) que é uma arquitetura totalmente livre que visa garantir a interoperabilidade e facilidade de manutenção dos sistemas das diferentes instituições do governo federal. A idéia é que, a partir de um *framework* e de uma arquitetura de referência, um conjunto de requisitos gere uma aplicação que possa ser mantida por qualquer um que conheça os dois primeiros.

No site da plataforma Pinhão (Pinhão, 2011) é descrita a arquitetura de *software* desenvolvida na CELEPAR (Companhia de Informática do Paraná) que é composta por uma metodologia de desenvolvimento baseada nos padrões de mercado e por várias aplicações que tratam determinadas classes de problemas. Estas aplicações, chamadas de Proto-Agentes,

podem ser conectadas a quaisquer outros sistemas que utilizem sua forma de concepção genérica.

Dentre as arquiteturas especificadas para domínios específicos, têm-se os listados a seguir:

Xavier (2001) se baseia nos atributos de qualidade do domínio para a indicação de um padrão arquitetural possivelmente adequado para a construção de uma arquitetura de domínio. Em seu trabalho, padrões arquiteturais são selecionados para um domínio comparando-se os atributos de qualidade desejados no domínio com a probabilidade em se obter estes atributos por meio da utilização de um determinado padrão arquitetural.

Em O'Brien e Stoermer (2001), apresenta-se uma análise de sistemas legados em um domínio para apoiar a definição de uma arquitetura de referência. Eles propõem um método de mineração de arquiteturas, o qual se inicia com a avaliação do domínio, a seleção dos sistemas candidatos e a recuperação das arquiteturas dos sistemas legados.

Almeida (2005) propõe uma arquitetura de *software* para arranjos produtivos locais (APLs) que serve de base para a construção de sistemas orientados a esta organização de empresas. Para isso levantou requisitos e informações sobre o domínio de negócio por meio da consulta a especialistas, empresários e entidades ligadas ao desenvolvimento de APLs.

Hira (2008) projeta uma arquitetura denominada Arquimedia, no estilo arquitetural em camadas, para terminal de acesso para TV digital interativa, visando contemplar, flexibilidade, inclusão digital e escalabilidade.

Thomé (2005) propõe uma arquitetura de *software* distribuída, configurável e adaptável aplicada às várias missões de controle de satélites, chamada SICSDA. O objetivo desta arquitetura é controlar mais de um satélite a partir de um mesmo conjunto de computadores, possibilitando a escolha de qual satélite deseja-se monitorar em um determinado instante.

Lazilha (2002) propõe uma arquitetura de linha de produto para sistemas de gerenciamento de *workflow*. Esta arquitetura pode ser usada para facilitar o processo de produção de diferentes sistemas de gerenciamento de *workflow* que possuem características comuns.

A maioria dos trabalhos encontrados evidencia as possibilidades decorrentes da variedade de arquiteturas para linha de produto a partir da diversidade de aplicações. Apesar da variedade

de pesquisas na área de arquitetura de *software*, raros trabalhos detalham as decisões arquiteturais relacionando as tecnologias aos requisitos pretendidos. É enriquecedor para os arquitetos de software e pesquisadores terem acesso a maior possibilidade de soluções técnicas diante dos desafios na elaboração de uma arquitetura, nesse sentido, a arquitetura proposta vem suprir a necessidade de uma arquitetura para linha de sistemas distribuídos e apresenta neste trabalho, detalhes de sua concepção.

1.5 ORGANIZAÇÃO DO TEXTO

Este trabalho constitui-se, além desta introdução, de mais cinco capítulos distribuídos como segue.

O Capítulo 2 apresenta os conceitos fundamentais para o entendimento desta pesquisa, como arquitetura de *software*, estilos arquiteturais, processo de desenvolvimento entre outros;

O Capítulo 3 descreve os padrões de projeto utilizados na concepção da arquitetura proposta;

O Capítulo 4 descreve a aplicação da arquitetura a partir da construção de um sistema de controle de acesso;

O Capítulo 5 apresenta a análise de resultados da avaliação da arquitetura. Nesta avaliação procuramos identificar os pontos fortes e fracos da arquitetura na visão dos envolvidos a fim de aperfeiçoar a mesma;

E, finalmente, o Capítulo 6 apresenta a conclusão e as contribuições deste trabalho, a indicação de trabalhos futuros e as considerações finais desta dissertação.

CAPÍTULO 2
CONCEITOS DE ARQUITETURA DE *SOFTWARE*

O capítulo 2 tem como objetivo apresentar alguns conceitos que serão úteis para a compreensão da solução arquitetural proposta.

2.1 ARQUITETURA DE *SOFTWARE*

Segundo a definição clássica de arquitetura apresentada por Shaw & Garlan (1996), uma arquitetura de *software* define o que é sistema em termos de componentes computacionais e os relacionamentos entre estes componentes. Adicionalmente descreve a estrutura técnica, limitações e características dos componentes, bem como as *interface*s entre eles. A arquitetura é o esqueleto do sistema e, por isso, torna-se o plano de mais alto nível da construção de cada novo sistema (Krafzig *et al.*, 2004).

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 (Garlan & Perry, 1995). 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.

Um dos fatores que servem de motivação para o estudo da arquitetura de um produto de *software* é o crescente aumento do tamanho e da complexidade desses produtos nos últimos anos. Neste contexto, os projetistas de *software* começaram a dar mais importância à estrutura dos produtos e à reutilização de seus componentes. Adotar uma estrutura correta para esses produtos pode trazer benefícios como a redução do seu tempo de manutenção.

O uso da arquitetura proporciona maior agilidade no desenvolvimento das aplicações, visto que permite a reusabilidade das estruturas pré-existentes, diminui o esforço dispensado para resolver detalhes de acesso a dados e padroniza o desenvolvimento de sistemas (Jazayeri, 2000). Tal padronização é fundamental para diminuir a curva de aprendizagem e facilitar a comunicação entre os membros da fábrica de *software*, pois estabelece um vocabulário comum e um fluxo padrão de tarefas. 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 *interface*s bem definidas que permitem a sua iteração com outros componentes.

Desta forma é possível definir quais componentes serão empregados, que tipo de integração ocorrerá entre eles e quais *frameworks* auxiliares devem ser utilizados para complementar a solução. Assim, o desenvolvedor é direcionado para resolver problemas referentes às regras de negócios que surgem no domínio da aplicação e não com a construção e adaptação de componentes de infra-estrutura.

A origem da arquitetura de *software* está na verificação de que determinadas estruturas de *software* são adequadas para determinados tipos de problema de projeto. A iniciativa de construção de arquiteturas enfoca a importância da escolha do estilo (Shaw & Garlan, 1996) ou padrão arquitetural (Buschamnn *et al.*, 1996) que irá guiar a especificação arquitetural.

2.2 ESTILO ARQUITETURAL

Estilos ou padrões arquiteturais 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 (Alexander, 1977). Esses estilos são úteis porque os usando pode-se reduzir o risco total de falha do sistema causados por tipos específicos de erros, podem auxiliar na resolução de problemas encontrados em situações similares e ainda facilitar e melhorar a comunicação dentro da equipe (Alur, 2003).

Um estilo arquitetural é fundamentado 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 (Shaw, 1995). Tem-se que estilos são alguns dos principais tipos de abstrações arquiteturais e tem como finalidade: a) impor uma estrutura completa para um sistema ou subsistema de *software* que seja apropriado ao problema que o sistema ou o subsistema está resolvendo; b) esclarecer as intenções do arquiteto sobre a organização do sistema ou do subsistema; c) fornecer um paradigma que ajude a estabelecer e manter uma consistência interna e d) permitir uma verificação e análise apropriada preservando informações sobre a estrutura para referência durante uma manutenção posterior.

Cada estilo de arquitetura lida com diferentes tipos de atributos de qualidade. Para obter a definição de uma arquitetura a partir dos estilos existentes, basta saber quais os atributos mais relevantes para a solução e confrontá-los com os atributos que o estilo atende. Além disso, os estilos podem ser combinados entre si para suportar os requisitos necessários e apoiar a definição de uma arquitetura mais adequada para o problema.

Tais técnicas podem ser usadas para definir a arquitetura para a construção de um sistema simples, bem como para a construção de uma família de sistemas. Em arquiteturas de família de sistemas, cada sistema da família possui características particulares que diferenciam suas arquiteturas específicas, mas as características genéricas são compartilhadas pela arquitetura definida para a família.

Usando o mesmo esquema descritivo para os estilos, podem-se identificar mais facilmente diferenças significativas entre eles. Uma vez que o estilo informal está claro, os detalhes podem ser formalizados (Allen, 1994). A descrição de cada estilo inclui as seguintes informações:

- **Problema:** O problema ao qual o estilo 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 estilo?
- **Solução:** O modelo de sistema capturado pelo estilo, junto com os componentes, conectores e a estrutura de controle que completam o padrão.

É necessário observar que a arquitetura não é definida apenas pela adoção de um ou vários estilos. Eles são apenas um primeiro passo para especificar a estrutura fundamental de um sistema. Além dos estilos existem também os padrões de projeto que constituem na pratica um mesmo conceito, diferenciando-se pela granularidade da sua estrutura. Enquanto os estilos arquiteturais são de largo escopo, abrangendo subsistemas completos, os padrões de projeto apresentam soluções em nível arquitetural para problemas pontuais dos sistemas.

As subseções abaixo apresentam os estilos arquiteturais institucionalizados na Empresa, de acordo com o Catálogo de Padrões Arquiteturais (Prodepa, 2010a), que é um ativo organizacional, resultado do convênio UFPA/PRODEPA, que define os padrões arquiteturais a serem utilizados no desenvolvimento e aquisição de *software* pelo Governo do Estado do Pará.

2.2.1 Estilo Arquitetural em Camadas

Problema de acoplamento e manutenção: O estilo arquitetural Camadas foi proposto para resolver problemas relacionados com a necessidade de desenvolver e evoluir sistemas de maneira independente, em função do tamanho do sistema e pressão por prazos. Além disso, o estilo é capaz de lidar com problemas de interação e acoplamento entre partes internas de um sistema.

Contexto: Este estilo arquitetural é recomendado para definir a estrutura e comportamento de um sistema utilizando um conjunto de regras e restrições. Entretanto é possível aplicá-lo em contextos mais específicos. Porém, há situações em que não existe uma clara separação das preocupações na arquitetura do sistema, as interações entre as partes não

são apoiadas apropriadamente e nem é possível evoluí-las independentemente. Assim, nestes casos é difícil identificar a separação do sistema para a aplicação do estilo arquitetural em camadas.

É comum encontrar a aplicação do estilo em camadas para lidar com alguns requisitos não funcionais de sistemas, principalmente quando todo o fluxo interno de mensagens é influenciado por uma característica de qualidade. Assim, a criação de uma camada para lidar com aspectos de criptografia (para lidar com o requisito de segurança) ou para abstrair os detalhes internos da implementação de um mecanismo de persistência (para lidar com exigências de portabilidade) são exemplos da adoção deste estilo.

Para a adoção deste estilo é necessário que a interação entre os componentes do sistema ocorra aos pares, considerando que uma camada só deve se comunicar com a camada imediatamente abaixo na hierarquia das camadas para prover serviços à camada superior.

Solução: No estilo em Camadas, estas são interligadas hierarquicamente duas a duas e cada camada só interage com a camada inferior e superior imediatamente vizinha. Cada camada é independente da camada imediatamente superior e podem ser executadas em *hosts* separados. A Figura 2-1 ilustra a estruturação de três camadas no qual cada camada possui sua *interface* provida para camadas superiores e a exigida das camadas inferiores.

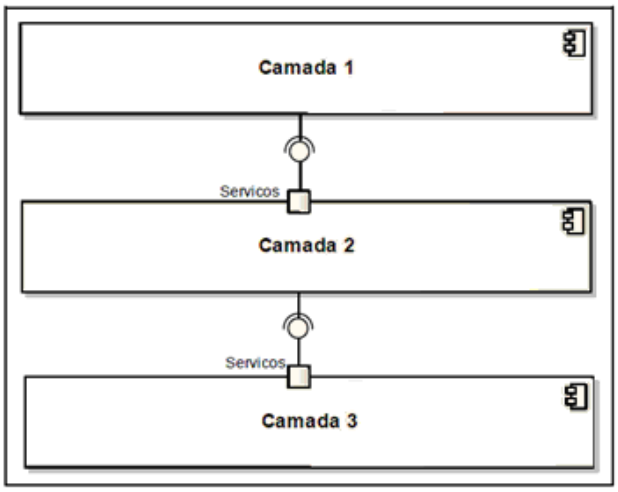

Figura 2-1 Estrutura do estilo em camadas.

FONTE: Catálogo de padrão arquitetural (Prodepa, 2010a).

Com a utilização do estilo em camadas as principais vantagens e desvantagens estão associadas ao número e granularidade das camadas do sistema. O número de camadas de um sistema deve ser razoável para que não ocorra um *overhead* em função de um número

excessivo de camadas e nem ocorra problemas de alto acoplamento, que dificulta a manutenção em função de um número muito pequeno de camadas. O ideal é uma quantidade que seja balanceada, no qual cada camada tenha uma responsabilidade específica e a coesão entre seus serviços esteja adequada, isto facilita a manutenção do sistema sem prejudicar o desempenho. Alguns exemplos de sistemas em que o estilo pode ser utilizado são: portais web, protocolos de rede, *middleware*, *frameworks* de acesso a dados (banco de dados, XML, arquivos em formatos proprietários), entre outros.

2.2.2 Estilo Arquitetural Modelo-Visão-Controlador (MVC)

Problema de manutenibilidade e acoplamento: *Interface*s gráficas com usuário (GUI – *Graphical User Interface*) são componentes que possuem grande tendência a ser modificada, devido a necessidade de estar adaptado aos diferentes perfis de usuário que tenham acesso ao sistema. Essas modificações freqüentes podem dificultar o desenvolvimento e a manutenção de um sistema em casos que os componentes da GUI são fortemente acoplados aos dados e às regras de negócio do sistema. O estilo arquitetural MVC é aplicado nesse contexto, ou seja, quando um sistema precisa atender requisitos de portabilidade ou manutenibilidade de *interface*s com usuário.

Contexto: A principal motivação para utilização do padrão MVC está relacionada a sistemas em que é necessária uma separação entre a lógica de negócios, o modelo do sistema e a apresentação. A utilização do MVC é recomendada quando há benefício na característica de manutenibilidade do sistema (Fowler, 2002). Neste caso a manutenibilidade é atendida por meio do desacoplamento das camadas, isto possibilita que não haja necessidade de alterar os componentes internos do sistema a cada modificação na *interface* com o usuário. Caso não haja uma separação da lógica das camadas de negócios e de apresentação, isto pode dificultar a manutenibilidade do sistema, além de não atender a sistemas que pretendem atender múltiplos tipos de clientes, no qual será necessário o desenvolvimento de uma aplicação para cada tipo de cliente (clientes *web*, telefones celulares, terminais em modo texto, entre outros). Em aplicações *web* baseada em documentos HTML é recomendado separar o controlador, pois a navegação no sistema necessita da interação com um servidor de páginas *web*.

Solução: O padrão MVC se aplica a sistemas em que a interação com o usuário é realizada por meio da interação entre diferentes componentes gráficos. Dependendo da volatilidade dos requisitos do sistema, pode ser necessário modificar a seqüência da interação entre componentes gráficos existentes ou ainda a adição ou remoção de componentes na *interface* gráfica. Nestes casos, o padrão MVC pode ser aplicado para tratar dos problemas de

manutenibilidade e portabilidade das aplicações, tornando-as mais flexíveis quanto às suas *interface*s com usuário.

Em casos em que o sistema possui poucas interações e poucos componentes gráficos, o uso o padrão MVC pode causar uma sobrecarga de processamento desnecessária. Isto pode acontecer pelo fato do padrão MVC adicionar complexidade de componentes extras ao sistema, que gerenciam a interação entre a camada de apresentação e o modelo do sistema. Caso este mesmo sistema evolua posteriormente com a adição de interações mais complexas na *interface* gráfica o padrão MVC pode ser aplicado após uma refatoração.

O modelo MVC é composto por três componentes principais: Modelo, Visão e Controlador, os quais interagem conforme o diagrama de componentes da Figura 2-2.

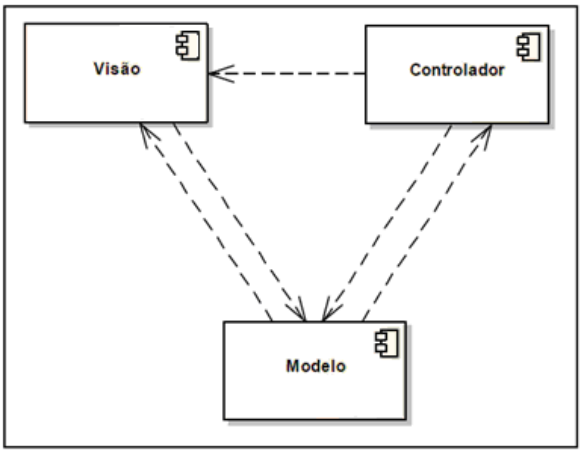

Figura 2-2 Estilo Arquitetural MVC.

FONTE: Catálogo de padrão arquitetural (Prodepa, 2010a).

Os componentes Controlador e Visão dependem do Modelo para obter informações do sistema e apresentar os dados neste contido. O Modelo depende dos outros componentes a fim de notificá-los em relação às mudanças ocorridas. E, finalmente, o Controlador depende da Visão para alterá-lo quando outra Visão deve ser apresentada ao usuário.

Dentre os benefícios associados à aplicação do estilo MVC em um sistema, pode-se citar:

- Portabilidade provida pela separação entre a *interface* gráfica com o usuário e o modelo (dados e regras de negócio) do sistema: A portabilidade é derivada da possibilidade de implementar diferentes clientes para uma mesma aplicação. Assim, a aplicação pode ser acessada de diferentes plataformas;

- Manutenibilidade resultante do desacoplamento das *interface*s do usuário e das classes de negócio que implementam a lógica do sistema. Assim, em futuras modificações da *interface* do sistema, a lógica de negócio não precisa ser alterada. Em aplicações que não utilizam esta separação, alterações que a principio se aplicariam somente a *interface* do usuário, acabam causando alterações em outras partes do sistema.

Como desvantagens relacionadas à implementação do padrão MVC, pode-se citar:

- Para permitir a separação de conceitos e desacoplamento entre as camadas de *software* é necessária a definição de componentes específicos. Tais componentes adicionam uma complexidade ao sistema e estão relacionados apenas com requisitos do padrão e não diretamente com a lógica de negócio do sistema. Entretanto, existem diversos *framework*s que facilitam esse trabalho para as mais diversas linguagens de programação;
- *Overhead* adicional de processamento causado pela comunicação entre os componentes reelacionado com a necessidade de implementar mecanismos de passagem de parâmetros entre as diferentes camadas de *software*.

2.2.3 Estilo Arquitetural de Tubos e Filtros

Problema de combinação de aplicações em cadeia: O estilo arquitetural Tubos e Filtros (do inglês, *Pipes and Filters*) é indicado quando existem grandes tarefas a serem realizadas e estas podem ser subdivididas em tarefas menores. Essas tarefas menores não devem requerer interações com usuário. Nesse contexto, componentes independentes, combinados em cadeia, realizariam as tarefas menores. O resultado do processamento em cadeia resultante é a solução da tarefa maior

Contexto: No estilo Tubos e Filtros cada componente tem um conjunto de entradas e um conjunto de saídas. Um componente lê um fluxo de dados da sua entrada e produz um fluxo de dados em sua saída, entregando uma instância completa do resultado em uma ordem pré-estabelecida. Este processamento é comumente alcançado pela aplicação de transformações locais dos fluxos de entrada e, de maneira incremental, computando esse fluxo de tal forma que a saída inicie antes da entrada ser totalmente consumida. Tais componentes são denominados de “Filtros“. Os conectores desse estilo servem como condutores para os fluxos, transmitindo saídas de um filtro para entrada de outro. Tais conectores são denominados “Tubos” (Microsoft, 2011).

Solução: Este estilo arquitetural é aplicado quando existe o requisito de decompor grandes tarefas em pequenas partes independentes. Como exemplo, este estilo tem sido utilizado na definição de diferentes aplicações utilitárias de sistemas operacionais (como *Windows* ou UNIX), no processamento de requisições *web* por *Servlets* na tecnologia Java, processamento de mensagens de *web-services*, entre outras.

O diagrama de componentes da

Figura 2-3 apresenta a estrutura geral do estilo Tubos e Filtros. Os componentes Origem e Destino são diferentes aplicações que produzem e recebem um fluxo de dados, respectivamente. Os Filtros funcionam de maneira independente e podem ser combinados dependendo da necessidade da aplicação. Os conectores entre os filtros são os tubos.

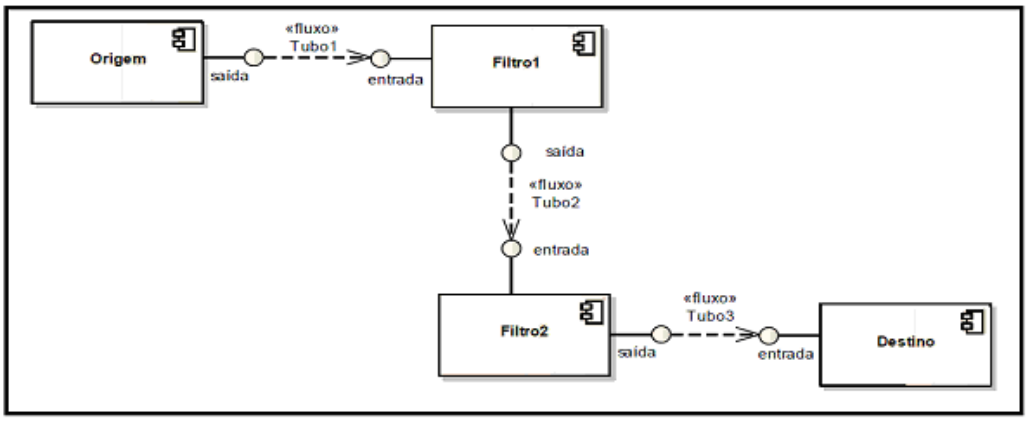

Figura 2-3 Estilo Arquitetural tubos e filtros.

FONTE: Catálogo de padrão arquitetural (Prodepa, 2010a).

Os componentes participantes do estilo Tubos e Filtros são: Tubos que representam os conectores para os Filtros, pelo qual passam os fluxos de dados de entrada e saída dos Filtros e os Filtros que representam os componentes que realizam um processamento específico sobre um fluxo de dados de entrada, resultando em um fluxo de dados de saída.

2.2.4 Estilo Arquitetural Publicação–Subscrição (*Publisher – Subscriber*)

Problema de Interoperabilidade e Notificação de Eventos: O estilo arquitetural Publicação-Subscrição é adequado tanto para contextos co-localizados quanto distribuídos que necessitem de um mecanismo de notificação de eventos para sincronização do estado entre os componentes de um sistema, ou ainda, para eventos que afetam ou coordenam dados necessários entre sistemas, ou os módulos deles (Buschmann, 2007).

Contexto: O estilo arquitetural Publicação-Subscrição provê um mecanismo no qual classes de publicadores (*publishers*) possam enviar informações que são de interesse de classes assinantes (*subscribers*). Este mecanismo funciona de forma que a informação é publicada a todos os assinantes registrados no *Publisher* e interessados na mesma.

Em alguns sistemas podem existir casos em que componentes não acoplados, sejam estes internos ou pertencentes a outro sistema, precisem propagar informações entre eles. O mecanismo de propagação deve funcionar de tal forma que possa enviar mensagens a todos os destinos interessados nesta mensagem. Além disso, não é necessário que a solução tenha um alto acoplamento, que evita problemas como a dificuldade de manutenção, por exemplo.

Solução: Este estilo é recomendado em situações em que existe a necessidade de enviar notificações assíncronas entre componentes com um baixo acoplamento. O publicador não precisa de uma resposta do assinante que recebeu a notificação, portanto este estilo é diferente de um mecanismo de requisição-resposta.

A Figura 2-4 mostra a estrutura básica do estilo Publicação-Subscrição, por meio de um diagrama de componentes. Este demonstra a estrutura geral na qual publicadores enviam mensagens aos assinantes registrados.

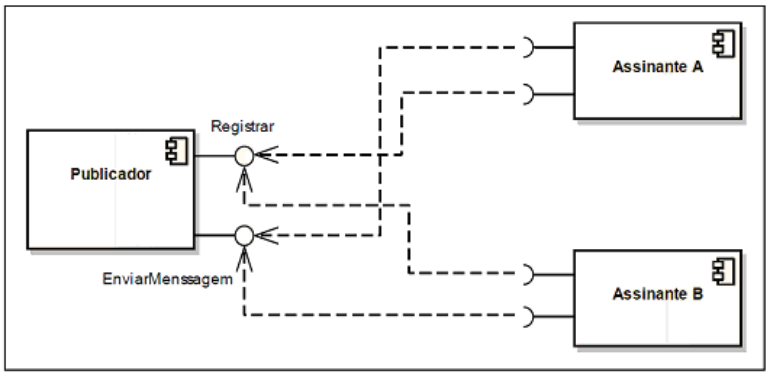

Figura 2-4 Estilo Arquitetural publicação–subscrição.

FONTE: Catálogo de padrão arquitetural (Prodepa, 2010a).

Os participantes da Figura 2-4 que representam a estrutura do estilo Publicação-Subscrição são os seguintes:

Publicador: serviço que publica (disponibiliza) mensagens aos assinantes pelo canal de comunicação, por intermédio de uma *interface* EnviarMensagem. O envio de mensagens ocorre somente aos assinantes previamente registrados por meio da *interface* Registrar.

Assinante A, B: serviços que assinam (recebem) mensagens do canal de comunicação, após o registro nos publicadores.

2.2.5 Estilo Arquitetural Repositório Compartilhado (*Shared Repository*)

Problema de interoperabilidade: Este estilo é recomendado para aplicações em que os componentes possuem uma dependência mútua de dados para desempenhar suas funções. A escolha de uma abordagem de comunicação direta entre eles pode causar problemas de confiabilidade, de desempenho e de difícil manutenção.

Contexto: Sistemas que lidam com grandes volumes de dados, como por exemplo, as finanças do governo, realizam operações como a monitoração de gastos e geração de relatórios. O componente que desempenha este papel necessita de informações produzidas por outros componentes responsáveis pela geração de relatórios de cada departamento tais como estoque de material e pessoal. A comunicação entre eles ocorre diretamente proporcionando uma abordagem com forte acoplamento.

Solução: O estilo Repositório Compartilhado integra a funcionalidade de componentes, por meio de um repositório compartilhado de dados. Este repositório serve como o meio de comunicação, armazenando todas as informações comuns a dois ou mais componentes. Desta forma, existe a garantia de confiabilidade na transmissão de dados aos clientes, ao contrário de outros sistemas, nos quais mecanismos de tolerância a falhas podem proporcionar a perda de informações (Hophe, 2003).

A Figura 2-5 representa a estrutura do estilo Repositório Compartilhado.

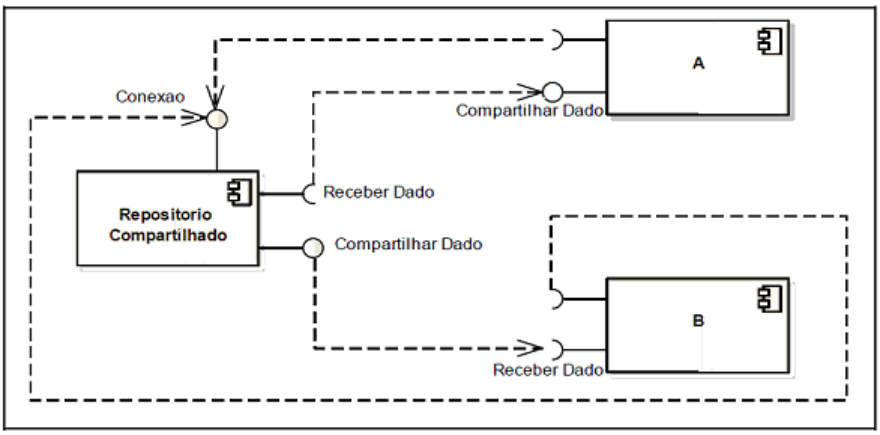

Figura 2-5 Arquitetura repositório compartilhado.

FONTE: Catálogo de padrão arquitetural (Prodepa, 2010a).

De acordo com a Figura 2-5, o estilo repositório compartilhado é composto essencialmente pelo repositório de dados e os componentes interessados em obtê-los ou compartilhá-los (componentes A e B).

De acordo com trabalho de Lalanda (1998), existem vários aspectos que podem ser ressaltados na adoção do estilo Repositório Compartilhado. Quanto às vantagens, pode-se citar:

- Redução do tráfego de comunicação entre componentes: alterações em dados geram atualizações somente para o repositório, reduzindo o envolvimento dos componentes que fazem a requisição dos mesmos;
- Segurança na informação: em caso de falhas na transferência de dados, basta somente realização de uma nova chamada ao repositório;
- Manutenção torna-se mais fácil: quando há necessidade de realizar alguma mudança nos componentes que registram e requisitam dados, as únicas *interface*s utilizadas na troca de informações são aquelas referentes à comunicação com o repositório.

As situações não recomendáveis para adoção deste estilo são as seguintes:

- O acesso ao repositório ocasiona a queda do desempenho da aplicação por ser um gargalo no acesso às informações;
- O acesso concorrente pode ocasionar falhas na transmissão e consistência dos dados;
- A manutenção no repositório pode afetar toda a aplicação, isto é, existe um esforço de replicar alterações nos códigos de todas as aplicações envolvidas.

2.3 ARQUITETURA DE LINHA DE PRODUTOS

Arquitetura de linha de produtos ou arquitetura de família de sistemas define os conceitos, estruturas, componentes e restrições necessárias para obter uma variação de características em vários sistemas enquanto fornece o máximo de compartilhamento das partes da implementação. A quantidade de diferenças ou dependências entre sistemas refletem na complexidade da arquitetura (Jazayeri, 2000).

A identificação prévia de evoluções, atualizações e diversificações em um sistema ajuda a estabelecer uma estratégia que influencia e ajuda a direcionar a arquitetura. Um sistema pode ser representado por vários produtos, estabelecendo-se uma linha de produtos com características e propriedades semelhantes. Construir uma arquitetura para uma linha de produtos significa envolver esforços para maximizar o uso da mesma arquitetura para vários sistemas semelhantes.

Segundo Jazayeri (2000) as atividades que envolvem a definição da arquitetura de uma linha de produtos são:

- Coletar de requisitos que impactam na arquitetura: listar todos os produtos candidatos a participarem da linha, identificar as propriedades genéricas que existem entre produtos candidatos, evidenciando as semelhanças, definir o escopo da linha de produtos e a estratégia de produção;
- Analisar a robustez dos produtos candidatos em conformidade com as evoluções futuras: identificar as limitações nos produtos candidatos;
- Projetar as camadas de acordo com níveis de abstração, localizando a arquitetura genérica da linha e a arquitetura de cada produto candidato;
- Implementar a arquitetura;
- Testar a conformidade da arquitetura, verificando os riscos que afetam o escopo, as propriedades genéricas e a estratégia estabelecida.

Essas atividades estão intrinsecamente relacionadas de tal forma que a alteração em uma delas implica em analisar o impacto nas demais.

2.4 LINHA DE SISTEMAS DISTRIBUÍDOS

Sistemas distribuídos consistem em uma coleção de computadores autônomos ligados por uma rede, buscando-se, desta forma, coordenar as atividades de maneira eficiente além de propiciar o compartilhamento de recursos, que sejam de *hardware*s ou *software* (Coulouris, 2005). Os sistemas distribuídos quando comparados aos sistemas convencionais, podem vir a apresentar maior eficiência, devido às suas características diferenciadas como:

- Compartilhamento de recursos: Em um sistema distribuído, diversos recursos podem ser compartilhados, como discos, impressoras, *software*s, banco de dados, arquivos, entre outros. Esse compartilhamento se dá por meio de comunicação, onde cada recurso deve oferecer uma *interface* de comunicação,

capacitando-o para ser manipulado, acessado e atualizado de forma confiável e consistente;

- Abertura: Possuem *interface* pública, o que permitem que novos serviços compartilhados sejam adicionados sem prejudicar os serviços existentes. Essa característica torna o sistema receptivo a modificações, isto é, não precisa de modificações complexas no sistema;
- Concorrência: Os processos em um sistema distribuído podem trabalhar concorrentemente e paralelamente o que permite a execução de vários programas simultaneamente, sem com isso afetar o desempenho do sistema;
- Escalabilidade: O sistema pode ser composto por apenas dois terminais e um servidor de arquivos ou até centenas deles e muitos servidores, possibilitando que o recursos seja compartilhado entre todos eles. Essa característica tenta garantir que o sistema e a aplicação não necessitem de mudanças quando aumentar a escala dos sistemas;
- Tolerância a falhas: Quando ocorrem falhas em *hardware* ou *software*, os programas podem produzir resultados incorretos ou eles podem parar antes de ter completado a atividade que vinha sendo realizada. O projeto de sistemas de computadores tolerantes a falhas é baseado com duas abordagens: redundância de *hardware* (uso de componentes redundantes ou em excesso) e restabelecimento de *software* (programas que recuperam as falhas ocorridas);
- Transparência: A transparência permite ao usuário enxergar um sistema distribuído como uma única maquina e não como uma coleção de componentes independentes. Segundo o manual da ANSA (*Advanced Network Systems Archtecture*) as duas formas de transparência que importam quando se trata de sistemas distribuídos são: transparência de acesso e transparência de localização (Coulouris, 2005).

No âmbito da empresa em questão, todas as aplicações são desenvolvidas para a ambiente *web*, além disso, a ampla difusão da internet motiva o enquadramento dos novos sistemas dentro de uma linha de sistemas distribuídos.

2.5 INFLUÊNCIA DOS REQUISITOS NÃO-FUNCIONAIS NA ARQUITETURA DE *SOFTWARE*

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 (Consoline, 2006).

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. Enquanto a falha em cumprir com um requisito funcional individual pode degradar o sistema, a falha em cumprir um requisito não-funcional pode tornar todos os sistemas inúteis.

Os requisitos de qualidade direcionam a arquitetura do *software* da mesma forma que as atividades centradas na arquitetura definem o ciclo de vida do *software*. O grau de associação entre a arquitetura e os atributos de qualidade gera uma série de conseqüências como pode ser visto abaixo (Ribeiro, 2005):

- As mudanças na estrutura da arquitetura com o intuito de melhorar um atributo de qualidade, geralmente afetam outras características de qualidade;
- A arquitetura é crítica para a realização dos atributos de qualidade;
- As qualidades do produto devem ser projetadas dentro da arquitetura do *software*;
- A arquitetura não pode garantir que qualquer atributo de qualidade existirá no produto final, ela apenas influencia.

Como 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 .*Net,* esta escolha faz com que o sistema apenas execute no sistema operacional da *Microsoft*, não atendendo os requisitos não-funcionais do sistema.

Sommerville (2006) lista os seguintes atributos de qualidade de *software*: segurança. confiabilidade, resiliencia, robustez, comprensibilidade, testabilidade, adaptabilidade, modularidade, complexibilidade, portabilidade, usabilidade, reusabilidade e eficiência.

Kandt (2006) considera que os atributos de qualidade de *software* mais valorizados são: disponibilidade, eficiência, manutenibilidade, portabilidade, confiabilidade, reusabilidade e usabilidade.

Padrão estabelecido internacionalmente a norma ISO/IEC 9126 define um modelo de qualidade para as características externas e internas para produtos de *softwares*, conforme Figura 2-6:

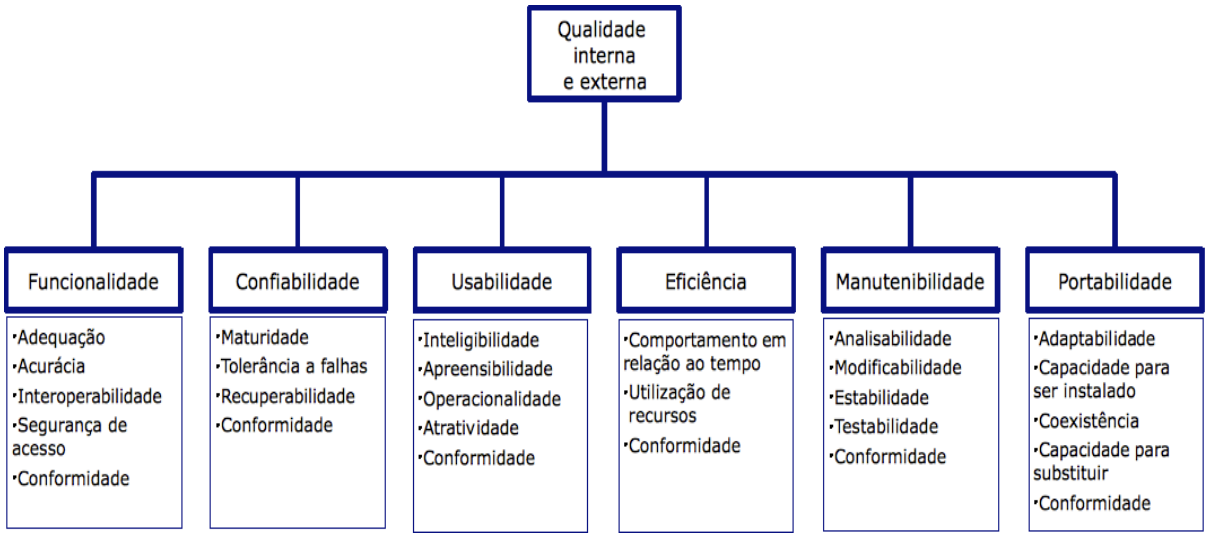

Figura 2-6 Atributos de qualidade para produto de software.

Diante dos modelos de qualidade apresentados, observa-se uma extensa quantidade de atributos de qualidade, sendo alguns comuns entre os modelos. Assim para a concepção da solução proposta, estabeleceram-se critérios para definição de atributos relevantes para implementação de uma arquitetura para linha de sistemas distribuídos. São eles:

- Modularidade: Uma arquitetura é considerada modular se os módulos discretos que a compõem puderem ser implementados separadamente e se uma mudança em um dos módulos causarem o mínimo de impacto nos demais módulos;
- Extensibilidade: Característica da arquitetura permitir extensões, isto é, adesão de novos componentes, com um mínimo de modificações, de modo a atender novas demandas de seus usuários (Jacobson *et al.*, 1997) e (Szyperski, 1998);
- Reusabilidade: facilidade de reutilizar a arquitetura em um contexto diferente, isto é, uma nova aplicação poderem ser construídas apenas com base nos componentes já disponibilizados pela arquitetura. Desta maneira não será preciso ser feita uma nova implementação;

- Manutenibilidade: Esforço requerido para localizar e corrigir uma falha em um sistema desenvolvido sob a arquitetura em seu ambiente de operação;
- Usabilidade: Facilidade em usar o sistema desenvolvimento sob a arquitetura, isto é, característica do sistema ser entendido, aprendido, usado e atrativo para o usuário, quando usado sob determinadas condições (ISO, 2001);
- Portabilidade: Característica do sistema desenvolvido sob a arquitetura se adaptar a diferentes ambientes computacionais, sem necessidade de alterações no sistema, isto é, capacidade ser transferido de um ambiente para outro. O ambiente pode ser uma organização, um *hardware*, um ambiente de *software* ou uma aplicação (ISO, 2001);
- Eficiência: Comportamento do sistema desenvolvido sob a arquitetura em relação ao tempo. Abrange aspectos relacionados à manutenção da informação com o suporte à transação e à persistência, como também questões que envolvem a manipulação de grandes quantidades de informação, como tempo de resposta e acessos a periféricos.

Foram selecionadas apenas as características de qualidade que dependem diretamente da arquitetura do *software* para a sua realização. Nossa escolha recebeu a influência de trabalhos que relacionam algumas características de qualidade ao projeto arquitetural de *software* (Bass *et al*., 1997) e (Chung et al., 1999).

2.6 DESCRIÇÃO DE ARQUITETURA *SOFTWARE*

Para representar as diferentes perspectivas das arquiteturas, tanto no tocante às suas visões estruturais, quanto às suas visões comportamentais, diagramas que fazem uso de diversos símbolos gráficos costumam ser utilizados. De uma forma geral, a notação é informal, onde comumente caixas descrevem componentes do sistema sendo representado e linhas indicam a ocorrência de alguma forma de comunicação, controle ou relacionamento entre estes componentes. Descrições informais como estas, são ambíguas e levam o leitor a interpretações pessoais e conflitantes (Perry & Wolf, 1992).

Dentre as abordagens propostas para representar arquiteturas, destacamos duas de interesse para os estudos deste trabalho: as linguagens de descrição da arquitetura (ADLs – *Architecture Description Languages*) e a utilização da UML (*Unified Modeling Language*).

Uma ADL suporta a descrição de um sistema em termos de componentes e conectores (Bass et al, 1997). Atualmente existem uma variedade de ADLs, como a *Rapide* (Rapide, 1997), SRI´s SADL (Sta, 2010), *Carnegie Mellon Unicon* (Shaw, 1995) e *Carnegie Mellon ACME* (Acme, 2011).

Embora as ADLs pareçam resolver os problemas relacionados à representação arquitetural, seu uso não vem sendo difundido de forma expressiva no projeto de *software*. Este problema se deve a uma série de fatores, dentre eles: a complexidade das ADLs, a falta da compreensão da comunidade de engenharia de *software* acerca das abstrações arquiteturais, a indefinição do que de fato representa uma ADL e a não existência de uma linguagem padrão para a descrição de arquitetura de *software* (Vasconselos, 2004).

Embora as ADLs tenham sido criadas para descrever arquiteturas, as arquiteturas aqui descritas utilizam a linguagem de modelagem UML desenvolvida por Grady Booch, James Rumbaugh e Ivar Jacobson (Booch, *19*99) que é a notação utilizada para descrever diagramas na empresa onde a arquitetura foi definida. Vários trabalhos na literatura fazem uso da UML (OMG, 2011) para este fim. A UML representa a notação padrão para sistemas orientados a objetos e seu uso em larga escala, tanto a nível acadêmico quanto industrial, é um dos fatores que impulsionaram seu uso na descrição arquitetural.

2.7 PROCESSO DE DESENVOLVIMENTO

Na fábrica de *software* onde a solução foi desenvolvida, o processo de desenvolvimento, segue o processo institucionalizado na empresa. O processo de desenvolvimento de *software* é aplicado no desenvolvimento de qualquer produto de *software*. Como o processo é longo e detalhado, este trabalho restringiu o processo em fases mais curtas e concisas.

Como o foco deste trabalho é o projeto da arquitetura de *software*, concentramos o processo de desenvolvimento na geração deste artefato. Foram criadas algumas interações junto aos envolvidos e acrescentamos, iterativa e interativamente, elementos à arquitetura, à medida que os requisitos eram levantados. Cada elemento da arquitetura tem responsabilidades que atendem a um ou mais requisitos. Em seguida, após o projeto da arquitetura, um protótipo foi desenvolvido sobre a arquitetura projetada (veja detalhes da implementação do protótipo no capítulo 4). A Figura 2-7 esboça o processo de desenvolvimento utilizado.

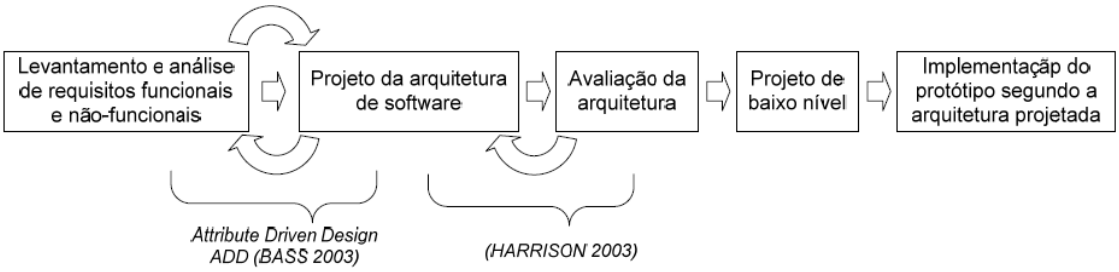

Figura 2-7 Processo de desenvolvimento usado para o projeto da arquitetura.

FONTE: (Consoline, 2006)

A atividade levantamento e análise de requisitos funcionais e não-funcionais geram como resultado, a declaração de escopo do sistema, a especificação de requisitos funcionais e não-funcionais, além dos casos de uso com os cenários elicitados.

A atividade projeto da arquitetura faz parte da fase de projeto dentro de um ciclo de desenvolvimento de *software*, sendo que durante a fase de projeto são gerados diagramas de classe para apresentação das entidades que fazem parte do domínio do sistema, diagrama de seqüência que demonstra a troca de mensagens durante uma transação e o diagrama de componentes para apresentação da arquitetura definida.

A avaliação da arquitetura gera resultados quantitativos a cerca dos atributos de qualidade definidos como relevantes na definição da arquitetura e servem para verificar o grau de atendimento esperado.

O projeto de baixo nível é a aplicação dos ajustes necessários na arquitetura após a análise de resultados obtidos com a avaliação da arquitetura, a fim de obter melhor atendimento dos requisitos-não funcionais exigidos pelo cliente.

A implementação do protótipo segundo a arquitetura projetada é a aplicação da arquitetura no desenvolvimento de um sistema, no caso desta pesquisa, um sistema de controle de acesso distribuído.

2.8 AVALIAÇÃO DA ARQUITETURA DE *SOFTWARE*

Muitas vezes sistemas são liberados com problemas de desempenho, riscos de segurança e disponibilidade, como resultado de arquiteturas inadequadas. As arquiteturas são definidas no início do ciclo de vida do projeto, mas as falhas resultantes são descobertas muito mais tarde, quando as mudanças afetarão negativamente o projeto.

Nos últimos anos, muitas organizações têm introduzido avaliação arquitetural como um componente crítico do ciclo de vida de desenvolvimento de *software*. O objetivo é identificar possíveis problemas com uma arquitetura proposta, antes da fase de construção, para determinar sua viabilidade arquitetônica e avaliar a sua capacidade de atender seus requisitos de qualidade.

O benefício mais significativo da avaliação é para tranqüilizar as partes interessadas mostrando que a arquitetura candidata é capaz de suportar os objetivos de negócio atuais e futuros e também satisfaz os seus requisitos funcionais e principalmente os não-funcionais, como desempenho, disponibilidade, extensibilidade e segurança.

Técnicas como o ATAM (*Architecture Tradeoff Analysis Method*) (Kazman, 1998), SAAM (*Software Architecture Analysis Method*) (Kazman, 1994) e ARID (*Active Review for Intermediate Designs*) (Clements, 2002), procuram identificar se a arquitetura projetada atende aos requisitos de qualidade levantados pelos envolvidos no projeto. Estas técnicas de avaliação definem um conjunto de procedimentos para estruturar a apresentação da arquitetura aos envolvidos e receber *feedback* das alterações necessárias (Bahsoon, 2003).

O método *Attribute Driven Design* ou ADD (Bass, 2004) é um processo recursivo de decomposição da arquitetura de *software*, desenvolvido pelo SEI *(Software Engineering Institute*), onde a cada estágio da decomposição são escolhidas táticas e padrões arquiteturais para satisfazer os cenários definidos para os atributos de qualidade.

Todos os padrões e técnicas levantados prevêem um contexto onde os envolvidos no projeto estão disponíveis para reuniões de avaliação de cenários e, além disso, requerem disponibilidade de tempo para o estudo da documentação da arquitetura e *feedback*.

Neste trabalho foi seguindo os passos básicos comuns aos métodos de avaliações de arquiteturas, descritos por (Harrison, 2011), foi executada uma avaliação controlada da arquitetura no contexto do projeto, conforme apresentado no seguinte: a) preparar a avaliação, definir os atributos a serem avaliados; b) definir e contatar os revisores envolvidos; c) enviar o questionário para os revisores; d) analisar o problema e coletar o *feedback* dos revisores e e) incorporar o *feedback* dos revisores à arquitetura.

CAPÍTULO 3
PADRÕES DE PROJETO

O capítulo 3 apresenta os padrões que servem de apoio aos projetistas para a construção de projetos arquiteturais dos sistemas desenvolvidos pela arquitetura proposta.

3.1 PADRÕES DE PROJETO

Defini-se padrão de projeto como uma descrição de objetos e classes que se comunicam e que são adaptados para resolver um problema de projeto geral em um contexto particular (Gamma *et al.*, 2003), ou uma solução particular, que é tão comum quanto efetiva em lidar com um ou mais problemas recorrentes (Fowler, 2002).

Os padrões de projeto documentados nesse capítulo foram adaptados do catálogo de padrões arquiteturais (Prodepa, 2010a) utilizado no desenvolvimento e aquisição de *software* pelo Governo do Estado do Pará.

3.1.1 Problema: Integração/Comunicação entre sistemas

3.1.1.1 Mensagem

Problema: O desenvolvimento focado para sistemas distribuídos traz a necessidade de estabelecer uma comunicação confiável, assíncrona e com baixo acoplamento entre os serviços que foram desenvolvidos independentemente (Buschmann, 2007).

Escopo do padrão: O padrão Mensagem estabelece uma conexão remota entre dois ou mais serviços, na qual são enviadas mensagens, cujo formato varia em relação ao contexto da solução. Tais mensagens são enviadas por meio de um canal (*Message bus*). Dependendo do cenário de aplicação, as mensagens podem ser independentes de plataforma, ou seja, podem ser enviados dados em formato XML, por exemplo.

Motivação: A interoperabilidade entre sistemas distribuídos é um requisito que pode ocasionar uma solução com baixa eficiência. Segundo Hophe (2003), existem várias alternativas que podem ser aplicadas à interoperabilidade entre sistemas como a Transferência de Arquivos (*File Transfer*), Banco de Dados Compartilhado (*Shared Database*), a Chamada de Procedimento Remota (*Remote Procedure Call* ou RPC), além do padrão Mensagem. Dentre as principais desvantagens da adoção das outras soluções se comparado ao padrão Mensagem, pode-se citar:

- Para a Transferência de Arquivos, as principais desvantagens são a latência na transferência de arquivos e o formato dos dados, isto depende da plataforma a ser adotada por cada um dos sistemas envolvidos;
- Para o Banco de Dados Compartilhado a desvantagem reside na estrutura não encapsulada dos dados. Isso pode requerer uma série de modificações no

banco, ou o não cumprimento de algumas regras do negócio da solução. Assim há um forte acoplamento dos sistemas com o banco de dados;

- Para as Chamadas de Procedimento Remotas, a desvantagem reside nas possíveis falhas de comunicação. Além disso, as chamadas têm um tempo de atraso considerável.

No estudo de caso desta pesquisa, foi requisitado um controle de acesso distribuído que é acessado por outros sistemas. Nesse contexto, existe um problema de interoperabilidade entre os sistemas *web* SISTEMA-A e o SISTEMA-B, cuja intenção é de tais sistemas serem executados em servidores distintos (distribuídos), conforme o diagrama de implantação (ver Figura 3-1). O SISTEMA-A requisita dados ao SISTEMA-B via *Interface* de comunicação RPC (*Remote Procedure Call*). A utilização deste mecanismo acarreta na necessidade de ser feita uma nova requisição a cada vez que seja necessária a obtenção dos mesmos dados, já que os dados são transientes para o SISTEMA-A, implicando na intensificação das chamadas remotas e das desvantagens anteriormente citadas.

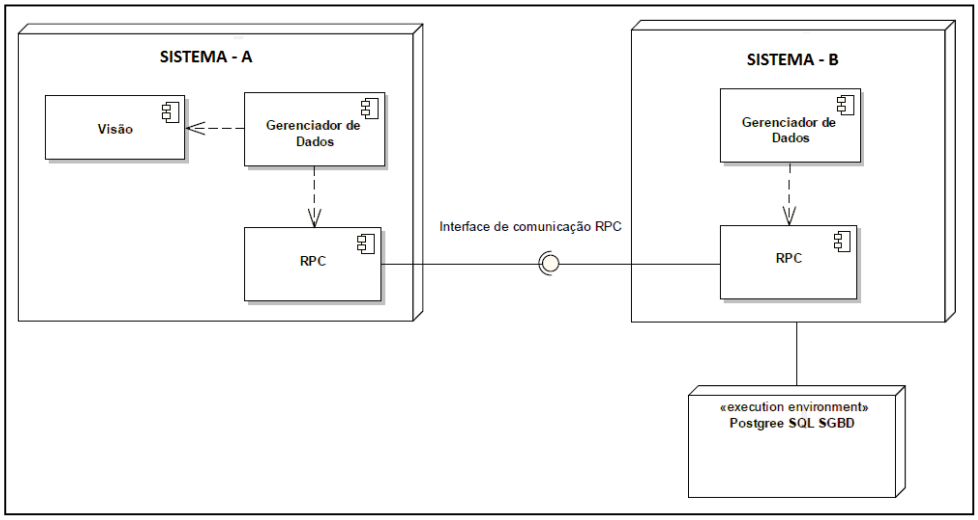

Figura 3-1 Diagrama de implantação do problema de comunicação.

FONTE: Catálogo de padrão arquitetural (Prodepa, 2010a).

Este padrão é recomendável para situações em que haja a necessidade de compartilhar dados assíncronos entre diferentes sistemas (integração). Tal integração deve ser realizada sob um mecanismo de requisição-resposta ou mesmo de notificação de eventos, de forma a alcançar baixo acoplamento e manter bom desempenho dos serviços que os sistemas do contexto disponibilizam.

A Figura 3-2 a seguir mostra a estrutura básica do padrão Mensagem, cujos participantes serão descritos a seguir.

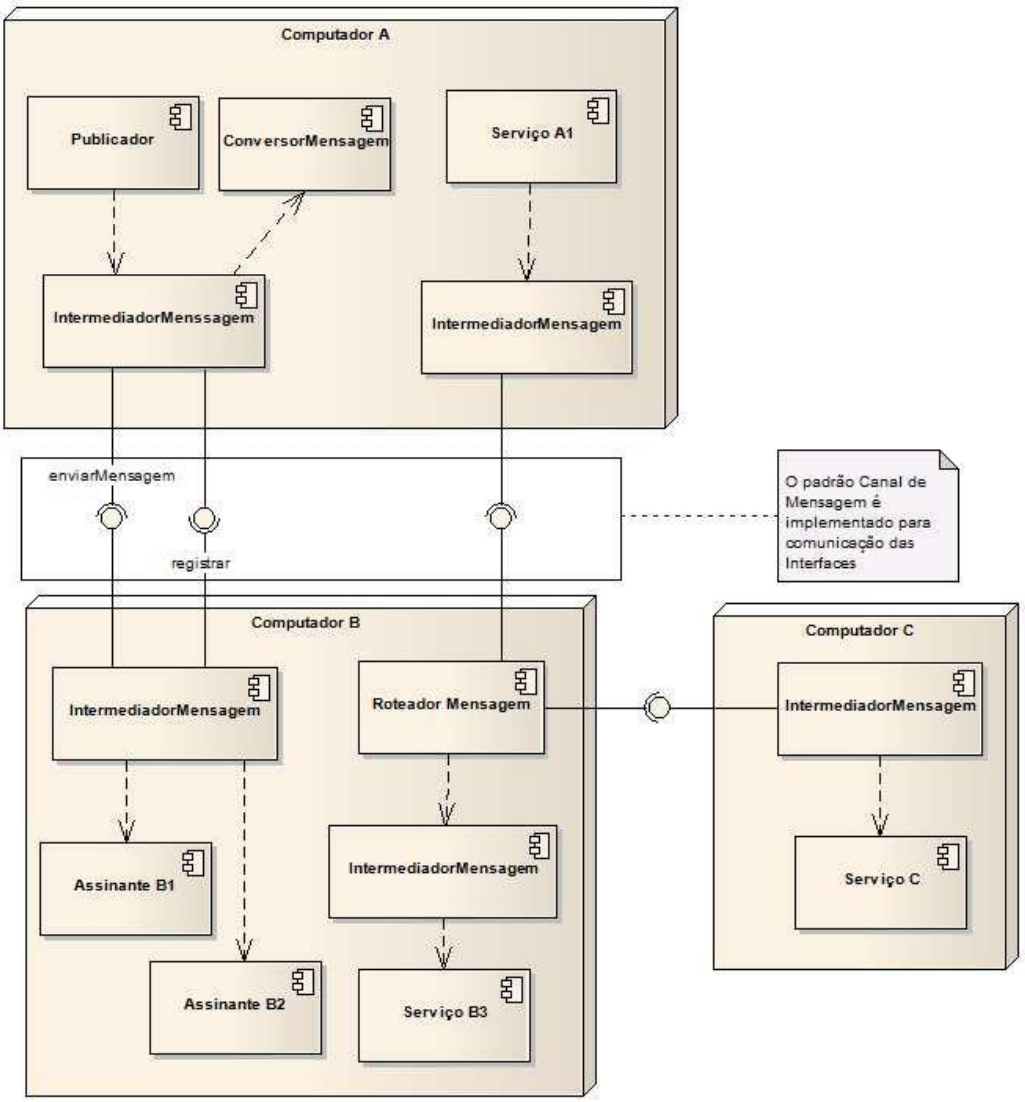

Figura 3-2 Padrão Mensagem.

FONTE: Catálogo de padrão arquitetural (Prodepa, 2010a).

A Figura acima representa várias situações de integração entre os padrões, cujos participantes são os seguintes:

- Computador A, B e C: podem representar tanto um sistema diferente, como vários computadores de um mesmo sistema;
- Serviço (A1, B3 e C1): serviços disponibilizados pelos computadores;
- Canal de Mensagem: representado pelos conectores de montagem. É o canal de comunicação que possibilita a troca de mensagens;

- Intermediador de Mensagem: trata a comunicação entre os clientes e serviços remotos, porém fazendo um tratamento na conversão dos dados em mensagens e vice-versa;
- Conversor Mensagens: trata a conversão entre tipos de mensagens;
- Roteador Mensagens: gerencia o destino das mensagens por meio do consumo destas por um canal de entrada para a inserção em outro canal de saída;
- Publicador (*Publisher*): serviço que publica (disponibiliza) mensagens no canal de comunicação. É um dos componentes do estilo Publicador-Assinante (*Publisher-Subscriber*);
- Assinante (*Subscriber*): serviço que assina (recebe) mensagens do canal de comunicação. É um dos componentes do estilo Publicador-Assinante.

Segundo Hophe (2003), como benefícios na adoção do padrão Mensagem, a solução pode ser uma transmissão de dados confiável, assíncrona e freqüente. Entretanto, ainda assim podem ocorrer problemas quando a quantidade de dados a ser trafegada é grande. Além disso, caso seja necessário sincronizar algumas mensagens, existe um esforço adicional para essa sincronização, já que essa abordagem síncrona não é nativa do padrão.

3.1.1.2 Intermediador (*Broker*)

Problema: De acordo com Buschmann (2007), o desenvolvimento de sistemas com componentes que possuem um baixo acoplamento possui benefícios como manutenibilidade, portabilidade e flexibilidade. Contudo, a comunicação entre componentes pode causar dependências e limitações no sistema. A comunicação estabelecida entre as camadas de aplicação de clientes e servidores pode se tornar limitada a uma mesma linguagem de programação, dependendo da estratégia de comunicação utilizada.

Em sistemas deste porte, serviços relacionados à adição, remoção, troca e busca de dados também são necessários. É recomendável que aplicações que façam o uso deste tipo de serviço não dependam dos detalhes de comunicação, a fim de garantir portabilidade e interoperabilidade.

Escopo do padrão: Segundo Buschmann et al. (2001), o padrão Intermediador é responsável por coordenar a comunicação, (como por exemplo, o encaminhamento de chamadas) assim como a transmissão de retornos e exceções, estes métodos, permitem o encapsulamento de detalhes em chamadas a serviços. Isto ocorre por meio de uma camada,

que serve como *interface* entre os componentes que iniciam a chamada (chamados de clientes) e os que a recebem (chamados de servidores). Além disso, a sua utilização permite a chamada de um método da mesma maneira como se estivesse sendo invocado localmente.

Motivação: Baseado no exemplo escrito por Buschmann et al. (2001), suponha que o governo do estado com intuito de estimular o turismo paraense cria uma demanda na qual se precise criar um sistema que precisa fornecer informações sobre diversos serviços e eventos, como por exemplo, bares, hotéis e restaurantes para toda a cidade de Belém (chamado de *City Information System* - CIS). Estes sistemas são acessados pela população, principalmente pelos turistas, por meio de terminais.

Sistemas como este devem ser projetados para que os serviços possuam baixo acoplamento. Desta maneira, vários tipos de manutenção podem ser realizados na aplicação, como a integração de novos serviços e a mudança de servidores e terminais. Neste contexto, o padrão Intermediador pode ser aplicado, pois a camada definida conforme o seu escopo serve como um mecanismo de coordenação de acesso aos serviços. Com a utilização deste padrão, há a disponibilização de um acesso transparente, encapsulando os detalhes relativos à comunicação entre os terminais e serviços.

De acordo com Buschmann et al. (2001), este padrão pode ser representado pelo diagrama de componentes da Figura 3-3, cujos papéis são explicados em seguida.

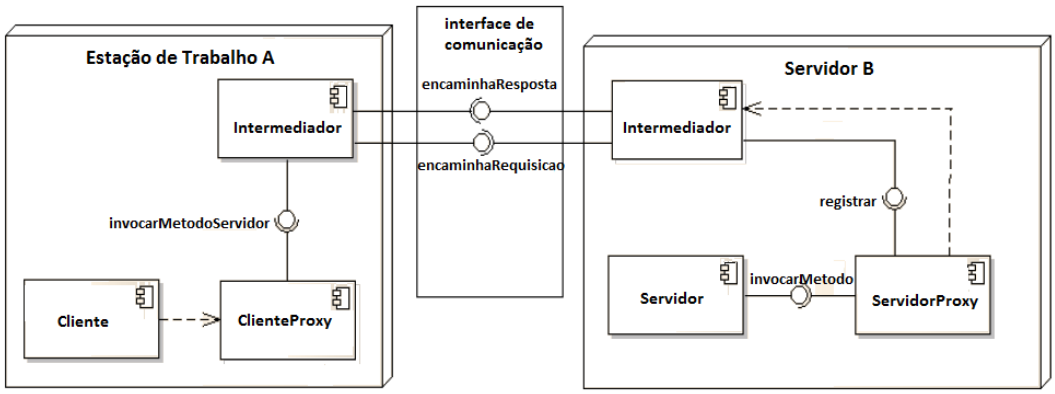

Figura 3-3 Padrão Intermediador.

FONTE: Catálogo de padrão arquitetural (Prodepa, 2010a).

A figura a cima apresenta os seguintes participantes:

- Cliente: Aplicações que acessam serviços de ao menos um único servidor (Buschmann et al, 2001). Estes encaminham as chamadas para um ClienteProxy;

- Servidor: segundo (Buschmann et al, 2001), é composto por objetos que disponibilizam suas funcionalidades por meio de *interface*s. Após estar devidamente registrado no Intermediador, o Servidor envia respostas e exceções para um Cliente, por meio de um ServidorProxy;
- Intermediador: responsável por intermediar a comunicação entre Cliente e Servidor. Oferece serviços como o registro de Servidores e invocação de métodos de um Servidor. Além destes recursos, sua função é também de localizar o Servidor apropriado quando recebe uma chamada de um Cliente;
- Cliente *Proxy* e Servidor *Proxy*: representam uma camada entre os componentes Intermediador, Cliente / Intermediador e Servidor. Sua função é servir de intermediário entre os componentes Intermediador e Cliente ou Servidor;
- *Interface* de Comunicação: componente opcional, cuja função é intermediar a comunicação entre Intermediador e a rede, encapsulando detalhes de comunicação.

Vale ressaltar que a invocação de um método por um Cliente é uma tarefa que também pode ser considerada assíncrona. Isto ocorre em virtude do fato de um Cliente poder realizar outras operações enquanto aguarda o retorno dos dados.

Segundo (Buschmann et al, 2001), o padrão Intermediador possui benefícios como:

- Transparência de localização: por serem responsáveis pela função de localização dos servidores ou clientes, estes dois componentes não precisam ter o conhecimento de onde está localizado o destinatário da mensagem a ser enviada;
- Fácil manutenção: os componentes que desempenham o papel de Proxy e *interface* de comunicação auxiliam na manutenção, visto que estes encapsulam os mecanismos de comunicação entre Intermediador, Servidores e Clientes. Desta forma, a estrutura do Intermediador não é afetada, caso seja realizada uma manutenção em um dos seus participantes que não sejam os responsáveis pela comunicação;

- Portabilidade: o padrão Intermediador encapsula os detalhes de implementação relacionados ao sistema operacional e a rede. Desta forma, a inclusão de novos clientes ou serviços feitos sob tecnologias diferentes é transparente;
- Interoperabilidade entre Intermediadores: um Intermediador pode se comunicar com outro, desde que sob um mesmo protocolo gerenciado por uma *interface* de comunicação, este último tem a função de traduzir os protocolos entre Intermediadores;
- Reusabilidade: uma nova aplicação cliente pode ser construída apenas com base nos serviços já disponibilizados na rede. Desta maneira não será preciso ser feita uma nova implementação.

Apesar de possuir tais vantagens, este padrão apresenta alguns pontos negativos:

- Eficiência limitada: aplicações utilizando o Intermediador podem ter um desempenho menor que aquelas em que a comunicação distribuída é estática e conhecida (Buschmann et al, 2001). Sistemas que dependem de um mecanismo direto de comunicação também possuem uma melhor performance, em virtude da arquitetura do Intermediador que utiliza camadas para prover portabilidade, manutenibilidade e flexibilidade;
- Baixa tolerância a falhas: comparado com sistemas não distribuídos, uma aplicação baseada no padrão Intermediador pode prejudicar a execução de todo o sistema. Caso os clientes possuam uma comunicação síncrona, estes correm o risco de não prosseguir com sua execução de uma maneira eficiente, em virtude de erros na comunicação entre Intermediadores ou no servidor;
- Teste e Depuração: a realização destas tarefas em um sistema utilizando o Intermediador pode vir a ser muito mais complexa, devido existirem muitos componentes envolvidos na aplicação.

3.1.1.3 Objeto de Transferência de Dados DTO (*Data Transfer Object*)

Problema: A granularidade fina é uma abordagem útil no projeto de funcionalidades disponibilizadas por serviços. Por meio do seu uso é possível realizar o acesso às informações em um objeto específico do negócio, o que traz uma série de vantagens: baixo acoplamento, melhor coesão e clareza no projeto. Apesar de tais vantagens, em ambientes distribuídos, a realização de chamadas a diversas informações disponibilizadas por um mesmo objeto pode

reduzir o desempenho de toda a aplicação. Isto ocorre em virtude ao tempo de resposta no qual os métodos do objeto demandam para atender a todas as requisições, além dos fatores relacionados ao tráfego de dados em uma rede, como a latência e *throughput* (Gamma *et al.*, 2003).

Outro problema que ocorre com a manipulação de dados de um sistema é a realização de múltiplas chamadas a um mesmo serviço. Esta tarefa pode causar problemas de inconsistência nos dados, mesmo que a chamada seja síncrona. Enquanto um cliente acessa um estado de um atributo, o valor pode ser alterado por outro cliente, prejudicando a consistência dos dados da aplicação.

Escopo do padrão: O padrão Objeto de Transferência de Dados provê um objeto que reúne todos os dados disponibilizados por um ou mais serviços, a fim de reduzir a quantidade de chamadas e, conseqüentemente, o tráfego de informações na rede. Desta forma, somente por meio do uso do padrão DTO que um cliente pode consultar ou alterar os dados do sistema.

Motivação: No cenário do estudo de caso desta pesquisa, o sistema requer armazenamento de informações diversas, seja de órgãos da administração direta e indireta, usuários e aplicações. Sendo que este sistema é acessado por diversos usuários simultaneamente. Isto pode resultar em um problema de desempenho, haja vista que vários usuários realizarão uma série de chamadas remotas. Além disso, a quantidade de requisições pode ocasionar uma queda no desempenho da aplicação no servidor localizado na matriz, o tempo de resposta para a obtenção de uma informação é submetido aos fatores envolvidos no processo de comunicação remota, como a tradução de informação e o tráfego na rede.

Neste contexto pode ser aplicado o padrão DTO onde os clientes recebem em apenas uma única chamada remota as informações sobre órgão, departamentos e funcionários. Desta maneira, a quantidade de chamadas pode ser reduzida. Esta abordagem auxilia o desempenho da aplicação como um todo, uma vez que os clientes podem obter tais informações por meio de um acesso local ao objeto DTO.

O padrão Objeto de Transferência de Dados é recomendado para situações nas quais os clientes necessitam realizar consultas ou alterar uma grande quantidade de dados de um mesmo serviço. Quando a quantidade de chamadas remotas influencia no desempenho de todo o sistema, a adoção deste padrão reduz o acoplamento e aperfeiçoa o desempenho da aplicação.

A estrutura do padrão é representada pela Figura 3-4 baseada em (Borysowich, 2011) e descrita na em seguida.

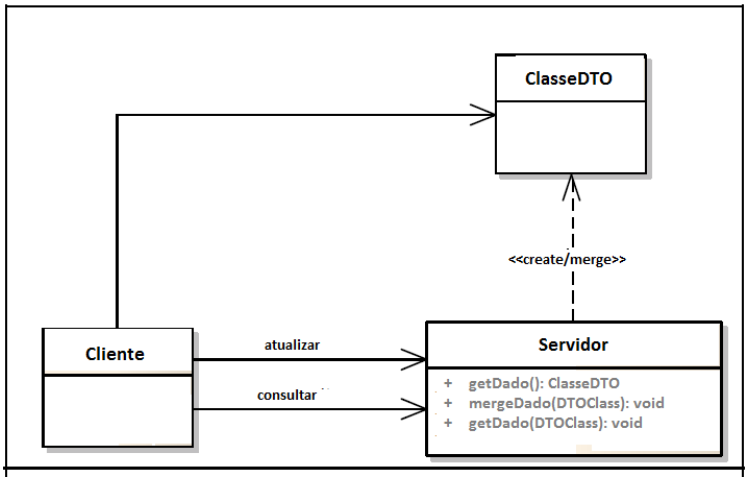

Figura 3-4 Padrão DTO.

FONTE: Catálogo de padrão arquitetural (Prodepa, 2010a).

ClasseDTO: Classe de objetos que contém uma cópia dos dados de Servidor e é acessada por objetos Cliente;

De acordo com (Microsoft, 2011), o padrão possui os seguintes benefícios como:

- Redução do número de chamadas: a quantidade é reduzida com a presença de um objeto DTO;
- Esconde a granularidade fina: reúne várias informações em uma única *interface*;
- Encapsulamento de dados: o DTO pode encapsular dados de vários objetos;
- Auxilia na representação do negócio: a reunião de métodos *getters* e *setters* possibilita uma melhor separação entre as camadas lógicas. Desta forma, propicia uma visão mais clara de alguns aspectos do negócio;
- Auxilia a fase de testes: A realização de testes de métodos provenientes de diversas classes pode ser reunida em um objeto DTO.

Em alguns casos a sua aplicação pode causar os seguintes problemas:

- Explosão de classes: Caso exista um DTO com assinaturas de métodos fortemente tipados, recomenda-se criar outro DTO voltado para cada tipo.

Porém esta abordagem pode gerar um grande número de classes, dificultando a codificação e manutenção do sistema;

- Codificação adicional: a passagem de parâmetros a uma chamada pode ser feita em menos linhas de código que com o uso do DTO. É preciso realizar chamadas a fim de obter o valor de cada parâmetro a ser utilizado, consumindo tempo na fase de codificação das chamadas remotas;
- Computação adicional: em um cenário distribuído a conversão de dados da aplicação consome um processamento extra. Dependendo da abordagem utilizada, tal situação ocasiona um *overhead* na rede, em virtude as operações de conversão.

O padrão Objeto de Transferência de Dados pode auxiliar uma série de estilos e padrões conhecidos na literatura. Dentre estes, o estilo Modelo-Visão-Controlador ou MVC (*Model-View-Controller*) que realiza requisições entre as camadas a fim consultar e atribuir valores a atributos. Neste caso, o DTO reduz a quantidade de chamadas que ocorrem entre as camadas da aplicação (Figura 3-5).

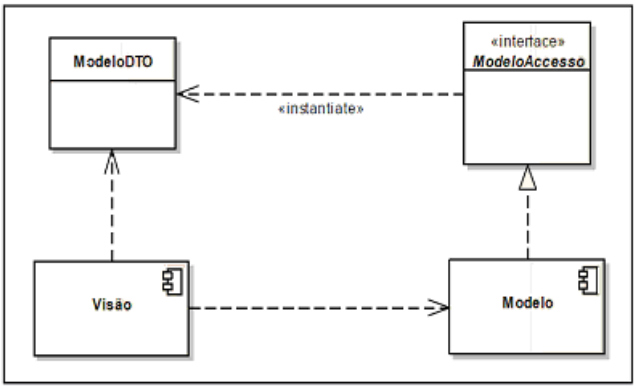

Figura 3-5 Relação entre DTO e MVC.

FONTE: Catálogo de padrão arquitetural (Prodepa,, 2010a).

3.1.1.4 Observador (*Observer*)

Problema: De acordo com Buschmann (2007), existem objetos que dependem do estado ou de dados mantidos por outros objetos. Caso o estado dos objetos que contém estes dados sejam alterados e não exista nenhum mecanismo de notificação, os estados dos objetos que dependem destes dados podem se tornar inconsistentes.

Escopo do padrão: Fornecer um mecanismo de propagação de mudanças sob uma abordagem um-para-muitos, no qual o provedor (*subject*) notifica os consumidores

devidamente registrados caso os valores dos seus atributos sejam alterados. Uma vez atualizados, estes consumidores, ou Observadores (*observers*), podem executar qualquer operação que achar necessária.

Motivação: No contexto da empresa aqui citada, o sistema requisitado possui a responsabilidade de gerar relatórios com alto desempenho, os quais, dependendo da quantidade de informações, podem consumir um custo de tempo considerável, segundo a arquitetura atual do sistema.

Para as classes que possuem o interesse em realizar alguma operação após o término da geração de um relatório, recomenda-se o envio de notificações sobre o término de tal operação. Por outro lado, não é uma boa estratégia o componente ser responsável pela geração dos relatórios e possuir o conhecimento de cada classe que deve ser notificada.

De acordo com Gamma (2003), o padrão Observador pode ser utilizado em situações em que existam duas funcionalidades dependentes uma da outra em um mesmo objeto, pode ser útil encapsular estas funcionalidades em objetos separados, permitindo o reuso dos mesmos de forma independente. Também pode ser utilizado quando uma mudança em um objeto necessita da mudança do estado de outros objetos. Além disso, não é necessário ter o conhecimento de quantos objetos precisam ser alterados.

A estrutura do padrão Observador pode ser ilustrada de acordo com a Figura 3-6, esse padrão é composto das seguintes classes e *interface*s:

- Provedor: *Interface* pela qual instâncias de Observador se inscrevem (*attach*) ou retiram a sua inscrição (*detach*) sobre a notificação de eventos de um ConcreteSubject;
- Observador: *Interface* de acesso para operação de atualização dos observadores no momento da ocorrência de um evento;
- ProvedorConcreto: Armazena o estado que é do interesse das instâncias do tipo ObservadorConcreto. Além disso, é o responsável por enviar notificações às instâncias ObservadorConcreto quando ocorre a mudança de estados;
- ObservadorConcreto: Armazena o estado que deve estar consistente com o estado de ProvedorConcreto. Também é responsável por realizar o método de atualização, a fim de estar com os valores consistentes aos observados em ProvedorConcreto.

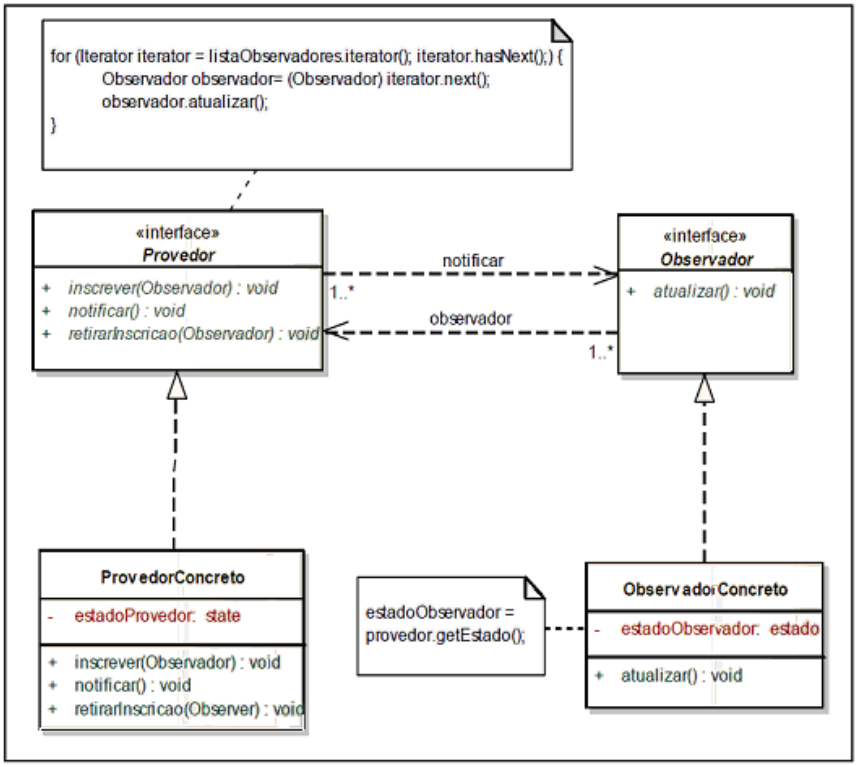

Figura 3-6 Padrão Observador.

FONTE: Catálogo de padrão arquitetural (Prodepa, 2010a).

De acordo com Gamma (2003), este padrão possui vantagens na sua adoção como:

- O acoplamento abstrato entre Provedor e Observador: Todo Provedor tem uma lista de Observador, de acordo com a *interface* da classe Observador. Entretanto, o Provedor não conhece nenhuma classe concreta de Observador, caracterizando a adoção deste padrão com uma abordagem de baixo acoplamento;
- Suporte a uma comunicação em broadcast: Uma vez que não é necessário ter o conhecimento dos observadores, cada notificação de um Provedor é enviada em broadcast para todos os interessados. Esta característica permite a adição ou remoção de observadores a qualquer momento, sem afetar o desempenho do envio de notificações.

A principal desvantagem, citada por Borysowich (2011), reside no controle do momento em que uma notificação é recebida. A ocorrência desta operação a qualquer momento pode não ser do interesse dos Observadores e o controle do momento de envio da notificação está fora do escopo deste padrão. Desta maneira, o envio de notificações

indesejadas a Observadores causa problemas não só a este tipo de objeto, mas também aos seus dependentes.

3.1.2 Manutenibilidade e Modularidade

3.1.2.1 Método Combinado (*Combined Method*)

Problema: O padrão Método Combinado é recomendado para situações em que é necessário realizar uma chamada para uma seqüência de métodos. No contexto do desenvolvimento de sistemas, existem casos em que é necessária a repetição de uma sequência de métodos, a fim de realizar tarefas diversas. Dentre os problemas que envolvem tal repetição existe a manutenibilidade. Quando um dos métodos precisa ser atualizado em virtude do surgimento de um novo requisito, então, é preciso atualizar todas as classes que contém a sequência dos métodos invocados para realizar determinada tarefa.

Outro problema pode ocorrer em ambientes distribuídos, no qual o acesso concorrente a uma sequência de métodos também não é considerada uma solução eficiente para a realização de uma determinada tarefa, pois cada vez que um cliente execute esta tarefa é necessária a realização de chamadas remotas, que quando realizadas em grande quantidade, acarretam a redução no desempenho de toda a aplicação.

Escopo do padrão: O objetivo do padrão Método Combinado é reunir uma sequência de chamadas a métodos em um único método. Desta forma, oferece uma abordagem de alta granularidade à aplicação, possibilitando uma redução na quantidade de linhas de código para as classes que façam uso desta sequência de métodos e melhora sua legibilidade. Além disto, o encapsulamento do acesso à sequência de métodos possibilita a realização de uma única chamada para as classes que necessitem desta nova assinatura.

Motivação: A motivação para o uso do padrão Método Combinado provém do uso repetitivo de uma seqüência de métodos ordenados na realização de uma tarefa específica (Buschmann, 2007). O ato de repetir tal seqüência pode ocasionar erros na ordem, o que pode implicar em erros no desempenho da aplicação.

A estrutura do padrão Método Combinado pode ser demonstrada de acordo com a Figura 3-7, os participantes do padrão são os seguintes:

- *Interface*Combinado: *Interface* pela qual outros objetos acessam o método metodoCombinado;

- ClasseA / ClasseB: Classes que contém os métodos nos quais são chamados pela sequência estabelecida pelo metodoCombinado;
- ClasseCombinado: Classe que realiza a *interface* e possui o método metodoCombinado. Por meio deste, métodos de outras classes são executados de acordo com a ordem definida pelos requisitos da aplicação.

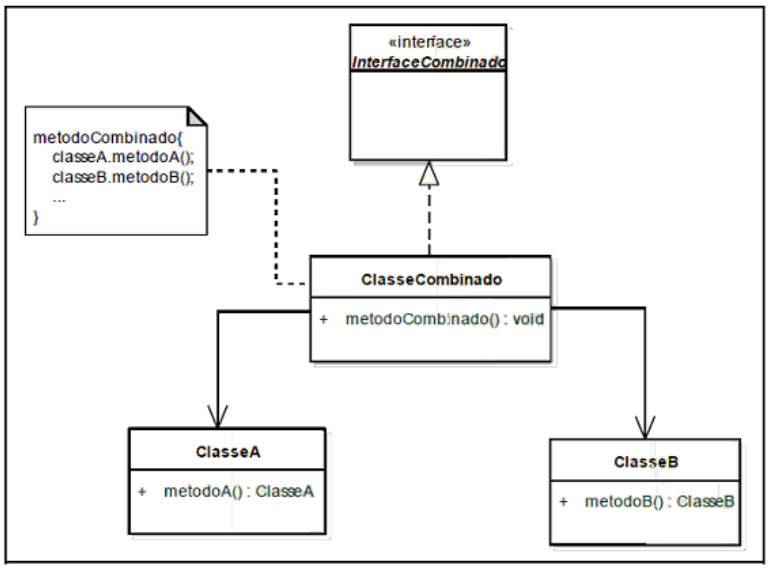

Figura 3-7 Padrão Método Combinado.

FONTE: Catálogo de padrão arquitetural (Prodepa, 2010a).

A aplicação do padrão Método Combinado afeta todas as classes envolvidas com a chamada da seqüência de métodos. Além da criação do método metodoCombinado, todas as classes que implementavam a seqüência de chamadas realizam uma chamada ao método combinado. Quanto às vantagens proporcionadas com a sua utilização, podem ser citadas as seguintes:

- Manutenibilidade: ao alterar a assinatura de algum método da seqüência modifica-se as assinaturas da própria classe e no metodoCombinado. Desta forma, tende-se a reduzir o tempo para realizar uma manutenção;
- Concorrência: quando clientes acessam uma seqüência de métodos encapsulada por um Método Combinado, a permissão de acesso se aplica a sequência como um todo e não apenas a um dos métodos dela. Caso ocorra uma exceção, outros objetos na fila de execução não são afetados, apenas o objeto que está com a permissão de acesso realiza o tratamento à exceção.

3.1.2.2 Objeto de Acesso a Dados (DAO - *Data Access Object*)

Problema: Aplicações que necessitam de acesso a dados localizados em fontes como bancos de dados, arquivos XML, arquivos *batch*, ou arquivos de texto, utilizam códigos específicos para a plataforma em que os dados estão localizados. Estes códigos devem permanecer separados da lógica da aplicação em função de mudança no esquema dos dados persistentes ou da plataforma em que estão localizados.

Escopo do padrão: A forma de acesso aos dados de uma aplicação varia consideravelmente dependendo da fonte de dados. O padrão DAO consiste em abstrair o mecanismo de persistência utilizado na aplicação. É um padrão que possibilita a separação das regras de negócio das regras de acesso ao banco de dados. Para isso, oferece uma *interface* comum de acesso aos dados, escondendo as características de uma implementação específica.

A camada de negócio acessa os dados persistidos sem ter conhecimento se os dados estão em um banco de dados relacional, um arquivo XML, ou em um arquivo texto. O padrão DAO esconde os detalhes da execução da origem dos dados, abstraindo e encapsulando o acesso a fonte de dados.

Motivação: Em muitas aplicações a persistência dos dados é implementada com mecanismos diferentes utilizando várias APIs (*Application Programming Interface*) diferentes. O processamento das estruturas de tratamento de dados em algumas aplicações é mapeado diretamente para um esquema de banco de dados relacional, manipulando diretamente coleções de dados na lógica do aplicativo de negócios via SQL, o que provoca um forte acoplamento entre o modelo da aplicação e o esquema de banco de dados utilizado (Buschmann, 2007).

Diferentes aplicações podem precisar de acesso a dados que estão em sistemas separados. Por exemplo, dados solicitados podem estar em um repositório LDAP (*Lightweight Directory Access Protocol*) ou em sistemas de *mainframe*. Em outros casos, os dados são fornecidos por sistemas externos como serviços de cartão de crédito. Nestes casos pode ocorrer da sintaxe e o formato dos comandos SQL variarem dependendo do banco de dados de cada sistema. Essas diferentes fontes de dados oferecem desafios para a aplicação e, potencialmente, podem criar uma dependência direta entre o código da aplicação e código de acesso a dados (Sun, 2011).

Para diminuição destes conflitos, o padrão DAO implementa um mecanismo de acesso para trabalhar com a fonte de dados. O componente de negócio que depende do DAO usa a *interface* mais simples exposta pelo DAO para seus clientes, que oculta completamente os dados de detalhes de implementação de origem de seus clientes. Como esta *interface* não se altera com mudanças na implementação do código, o padrão permite adaptação a diferentes tipos de armazenamentos, sem afetar seus clientes ou componentes de negócio.

A Figura 3-8 apresenta o diagrama de classes que mostra o relacionamento para o padrão DAO. O detalhe e descrição dos participantes envolvidos são descritos abaixo.

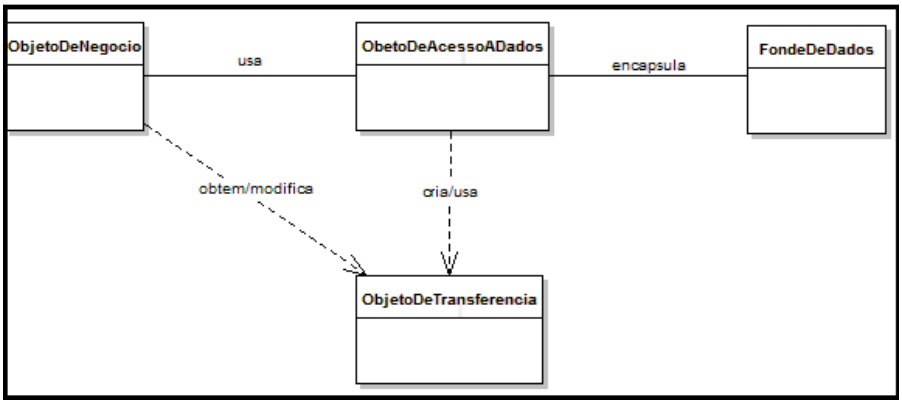

Figura 3-8 Padrão DAO.

FONTE: Catálogo de padrão arquitetural (Prodepa, 2010a).

A Figura 3-8 mostra as classes participantes da estrutura do padrão DAO, são elas:

- ObjetoDeNegocio: que representa os dados do cliente, requer acesso à fonte de dados para obter e armazenar dados. Este componente pode ser implementado como um *bean* entidade, ou algum outro objeto, além de um *servlet* ou *bean* auxiliar com acesso a fonte de dados;
- ObjetoDeAcessoADados: é a classe principal desse padrão, ele abstrai a implementação de acesso de dados para o ObjetoDeNegocio, permitindo de maneira transparente, o acesso à fonte de dados;
- FonteDeDados: representa a execução da fonte de dados que pode ser um banco de dados como um RDBMS, repositório XML, entre outros;
- ObjetoDeTransferencia: representa a transferência de um objeto utilizado como um portador de dados. Ele pode usar um objeto de transferência (DTO) para

retornar dados para o cliente, ou ainda pode receber os dados do cliente em um objeto de transferência (DTO) para atualizar os dados na fonte de dados.

Existem diferentes aspectos relacionados ao uso do padrão DAO. A seguir, são citados alguns desses aspectos de acordo com o catálogo de padrões da Sun (Sun, 2011):

- Transparência: os objetos de negócios podem usar a fonte de dados sem conhecer os detalhes específicos da implementação, para tornar o acesso transparente, visto que detalhes da implementação são encapsulados no DAO;
- Facilidade na migração: uma camada de DAOs torna mais fácil a migração entre aplicação de banco de dados diferentes. Os objetos de negócios não têm conhecimento da execução de dados subjacente. Assim, a migração envolve mudanças apenas para a camada DAO;
- Simplicidade de código: este padrão reduz a complexidade do código na camada de negócio, pois os DAOs gerenciam toda a complexidade de acesso a dados. Isto simplifica o código nos objetos de negócios e outros clientes de dados que usam o DAOs. Todas as execuções de código relacionadas (como instruções SQL) estão contidas no DAO e não no objeto de negócio. Isso melhora a legibilidade do código e facilita a manutenção;
- Facilidade no gerenciamento da aplicação: centraliza o acesso aos dados em uma camada separada, pois todas as operações de acesso a dados são transferidos para os DAOs. A camada de acesso a dados separados pode ser vista como a camada que isola o resto da aplicação a partir da implementação de acesso a dados.

Desvantagem: pode ser citado o fato de não haver mecanismos que possibilitem o mapeamento com mais de um mecanismo de armazenamento. Desta forma, ainda é preciso o desenvolvimento de códigos básicos – para lidar com a inserção, atualização e remoção de objetos. Apesar do esforço mesmo assim é vantajosa a separação das fontes de dados.

3.1.2.3 *Interface* Estendida (*Extension Interface*)

Problema: O acesso a componentes é considerado adequado quando é fornecida uma *interface* estável e com coesão (Buschmann, 2002). Entretanto, *interface*s podem sofrer mudanças. As alterações em assinaturas dos métodos de uma *interface* podem comprometer a comunicação entre os componentes e o acesso aos serviços disponibilizados pela mesma.

Um problema frequente reside na adição de novas funcionalidades em uma *interface*. Segundo Buschmann (2002), a adição de novas assinaturas pode tornar a *interface* "inchada", dificultando o acesso e a manutenção ao componente. Existem casos em que os clientes são afetados inclusive pela adição de novas funcionalidades na *interface* que não são utilizadas por eles (Buschmann, 2007). Desta forma, uma *interface* deve permanecer estável quando são adicionados novos serviços ou quando as assinaturas são atualizadas.

Escopo do padrão: De acordo com Buschmann (2002), este padrão permite que várias *interface*s sejam providas por um componente, a fim de prevenir o "inchaço" de *interface*s e o rompimento na comunicação com o código dos clientes, quando desenvolvedores adicionam ou modificam funcionalidades de um componente.

Uma *Interface* Estendida é definida para cada serviço que um componente disponibiliza. Quando acessada, a execução é redirecionada para as classes do componente que implementam o serviço.

Motivação: Considere um ambiente virtual de aprendizagem para o ensino de uma linguagem de programação. Este ambiente foi projetado inicialmente apenas com módulos voltados à disponibilização do conteúdo e do gerenciamento de usuários. Durante o projeto, foi decidido que os clientes realizariam a chamada a qualquer tipo de serviço por meio de uma única fachada de acesso. Com o tempo, foram adicionados ao sistema novos serviços relacionados à implantação e correção de avaliações, levando ao "inchaço" da fachada de acesso (Figura 3-9).

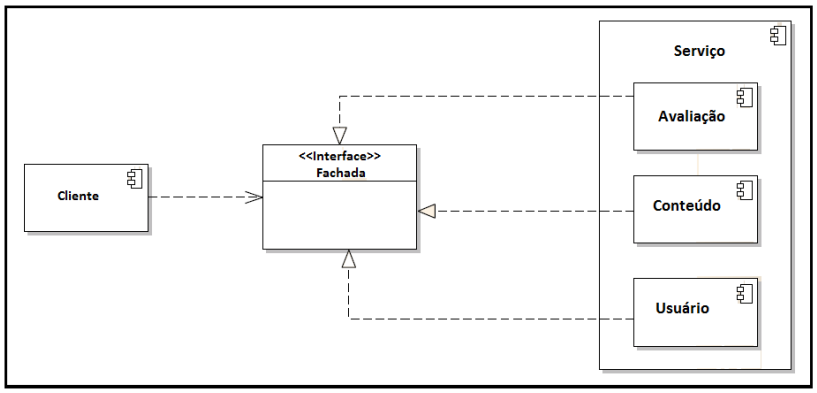

Figura 3-9 Acesso para os serviços do Ambiente Virtual de Aprendizagem.

FONTE: Catálogo de padrão arquitetural (Prodepa, 2010a).

Este é um caso no qual o padrão *Interface* Estendida pode ser aplicado, ele visa dividir as funcionalidades do sistema em *interface*s destinadas a cada tipo de serviço, o que facilita a

manutenção de assinaturas dos métodos dos componentes disponibilizados pelas *interface*s, aumentando a coesão das mesmas.

De acordo com a Figura 3-10, baseado em Schimidt (2000), o padrão *Interface* Estendida pode ser representado da seguinte forma:

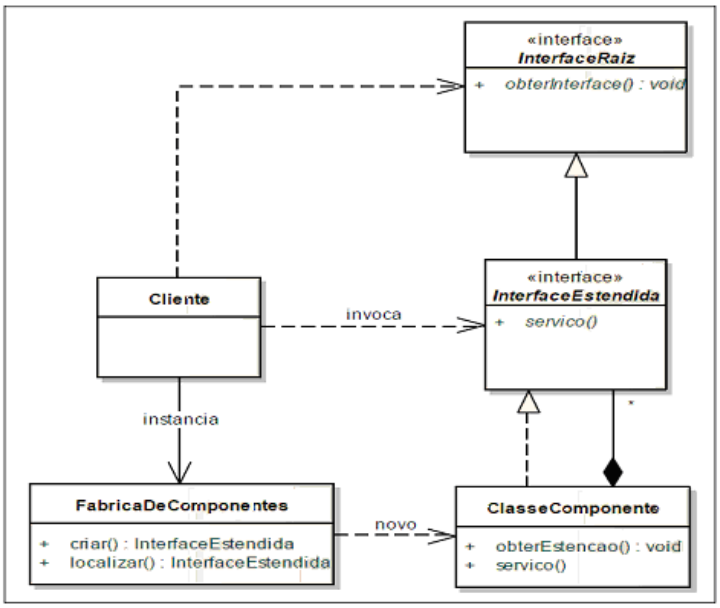

Figura 3-10 Estrutura do padrão Interface Estendida

FONTE: Catálogo de padrão arquitetural (Prodepa, 2010a).

De acordo com a Figura 3-10, os participantes são os seguintes:

- Cliente: responsável por acessar os serviços por meio da invocação à *Interface*Estendida;
- *Interface*Estendida: define uma *interface* para um papel específico desempenhado por ComponentClass;
- *Interface*Raiz: é um participante opcional neste padrão, que define papéis que devem ser desempenhados por cada *Interface*Estendida;
- ClasseComponente:classe que implementa o serviço acessado por clientes;
- FabricaDeComponentes: responsável por criar ou localizar os componentes que realizam as funcionalidades definidas por *Interface*Estendida.

O padrão *Interface* Estendida possui os seguintes benefícios (Schimidt, 2000):

- Promove extensibilidade: É papel dos desenvolvedores prever se uma *interface* pode se tornar inchada ou não. Desta forma, por meio do padrão *Interface* Estendida novas *interface*s são adicionadas quando ocorre a adição de uma nova funcionalidade em um componente;
- Separação de interesses: Um componente pode desempenhar diferentes papéis, os quais são acessados por clientes por diferentes *interface*s estendidas. Desta maneira, cada papel ou um agrupamento semelhante de papéis são organizados em diferentes *interface*s estendidas;
- Polimorfismo: Quando um cliente acessa uma *interface* estendida, não é necessário ter o conhecimento de qual componente realizará a implementação da assinatura dos métodos;
- Reduz o acoplamento entre Clientes e seus Componentes: Clientes acessam a implementação de componentes por meio de *interface* estendida, o que reduz o acoplamento entre eles;
- Suporte às *interface*s agregadas: Componentes podem agregar outros componentes e conseqüentemente as respectivas *interface*s podem estar agregadas. Desta maneira, uma *interface* agregada pode delegar requisições a qualquer um dos componentes, desde que a mesma faça o uso do método obterExtensao.

Como desvantagens na utilização deste padrão, Schimidt (2000) cita as seguintes:

- Aumenta o esforço de implementação de um componente: Dependendo da linguagem de programação utilizada, o esforço em implementar o padrão *Interface* Estendida pode ter um custo aumentado, em razão da linguagem de programação não prover mecanismos necessários para a utilização da *interface*;
- Aumenta a complexidade no código para os clientes: *Interface*s estendidas conferem aos clientes a responsabilidade em determinar quais *interface*s são as mais apropriadas e quais devem ser utilizadas em um caso específico;
- Aumento no tempo de execução: Clientes não podem acessar diretamente os componentes, o que diminui o desempenho do sistema. Além disso, a opção em utilizar um contador que faça referência ao tempo de execução de componentes inicializados pode ser complexa e ineficiente.

De acordo com Buschmann (2002), a inviabilidade na realização de manutenções em uma *interface* é uma desvantagem, portanto a adoção deste estilo arquitetural em um projeto permite que os interesses em manutenção sejam bem definidos para que seja possível avaliar o impacto de seu uso.

3.1.3 Gerenciamento de Recursos

3.1.3.1 Fábrica Abstrata (*Abstract Factory*)

Problema: Se uma aplicação possuir como característica de qualidade a portabilidade, ela precisa encapsular dependências de plataforma, estas plataformas podem incluir sistema operacional, banco de dados, sistema de gerenciamento de janelas, entre outros elementos dependentes de plataforma. Frequentemente este encapsulamento não é projetado com antecedência e as declarações de métodos e diretivas com opções para todas as plataformas suportadas atualmente começam a se replicar rapidamente pelo código.

Em muitos casos é preciso construir um conjunto de classes da mesma família de forma que a aplicação suporte todas as mudanças. Para uma melhor estruturação da arquitetura, é preciso separar detalhes relacionados à implementação de classes das *interface*s do cliente para manter o sistema mais flexível (Buschmann,2007).

Escopo do padrão: O padrão Fábrica Abstrata permite a criação de famílias de objetos, relacionados ou dependentes, por meio de uma única *interface* sem necessidade de especificação da classe concreta. Ao invés do cliente chamar um método de criação ele possui um objeto, fábrica abstrata, que usa este objeto para chamar os métodos de criação. Assim o aplicativo, ou cliente, interage por meio de classes abstratas sem ter conhecimento da implementação das classes concretas.

Motivação: A criação de cada objeto individualmente e isoladamente dificulta a configuração se os objetos criados não forem compatíveis. É importante a definição de uma *interface* de fábrica para a criação e, opcionalmente, a disposição de famílias de objetos relacionados. Esta *interface* deve ser feita com fábricas especializadas que realizarão o trabalho de criação em nome de seus clientes.

Uma fábrica abstrata dita a criação do objeto e *interface* de disposição para todos os tipos de objetos relacionados. As fábricas concretas derivam de uma fábrica abstrata controlando a criação e disposição de tipos de objetos particulares, assegurando que os objetos criados são semanticamente compatíveis. As fábricas separam as preocupações do ciclo de vida do objeto e da forma como ele será usado, de modo que os clientes da aplicação

não necessitem conhecer como os objetos que utilizam são criados ou destruídos (Buschmann, 2007).

A Figura 3-11 mostra a estrutura genérica do padrão Fábrica Abstrata, representado pelo diagrama de classes. Cada participante envolvido é descrito na seção a seguir.

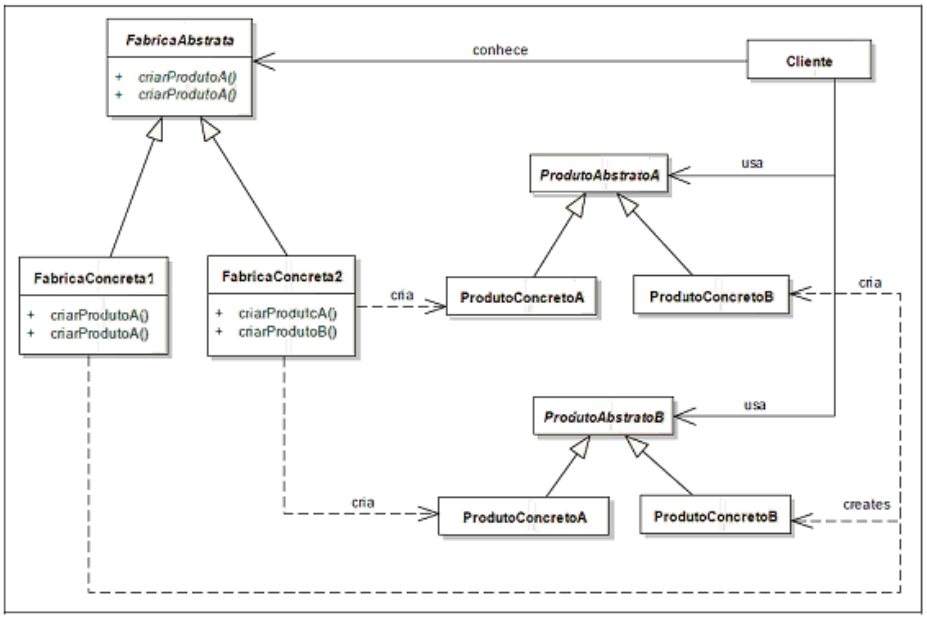

Figura 3-11 Estrutura genérica do padrão Fábrica Abstrata.

FONTE: Catálogo de padrão arquitetural (Prodepa, 2010a).

Os participantes deste padrão são representados pelo diagrama de classes da Figura 3-11, descritos a seguir:

- FabricaAbstrata: declara uma *interface* para operações que criam objetos como produtos abstratos, é a classe abstrata que possui o código que é comum às FabricaConcreta;
- FabricaConcreta: implementa as operações que criam objetos para produtos concretos;
- ProdutoAbstrato: declara uma *interface* para um tipo de objeto produto, é a classe abstrata que possui o código que é comum aos ProdutoConcreto;
- ProdutoConcreto: define um objeto produto a ser criado pela Concrete*Factory* correspondente. Implementa a *interface* de ProdutoAbstrato;
- Cliente: usa as *interface*s declaras pela Abstract*Factory* e ProdutoAbstrato.

O uso deste padrão tem algumas vantagens e desvantagens que são citadas a seguir:

- Contribui para padronização de *interface*s com os usuários: quando os desenvolvedores utilizam essas classes abstratas em detrimento das classes nativas da linguagem, colaboram com a padronização da aparência e comportamento da aplicação;
- Isola classes concretas: uma fábrica encapsula a responsabilidade e o processo de criação de objetos de produtos, isolando os clientes das classes de implementação. Os clientes da aplicação manipulam as instâncias por meio de suas *interface*s abstratas;
- Facilita o câmbio de famílias de produtos: a classe de uma fábrica aparece somente uma vez na aplicação quando é instanciada, facilitando a mudança da fábrica concreta que a aplicação usa, ou seja, quando ocorre mudança na fábrica concreta, a família inteira de produtos muda;
- Promove consistência entre produtos: O padrão permite implementar a restrição do uso de apenas uma família por vez, pois os produtos de uma determinada família devem funcionar conjuntamente, eles são projetados para não trabalharem conjuntamente com produtos de outras famílias.

Desvantagens do uso: este padrão dificulta o suporte a novos tipos de produtos, isto porque a fábrica abstrata fixa o conjunto de produtos que podem ser criados.

3.1.3.2 Fábrica de Métodos (*Factory Method*)

Problema: Um problema que pode ser solucionado pelo padrão Fabrica de Métodos pode ser o caso em que *framework* precisa padronizar seu modelo de arquitetura para uma variedade de aplicações. Mesmo com esta padronização, deve-se permitir que as aplicações individuais definam e forneçam seus próprios objetos de domínio e instanciação.

Outro contexto em que este padrão pode ser aplicado onde usualmente a natureza dos objetos criados varia com as necessidades do programa. Assim, a abstração do processo de criação de objetos para uma classe de criação torna o programa mais flexível e genérico.

Escopo do padrão: Este padrão visa encapsular a criação de um objeto em um método. O padrão fornece uma *interface* para criação de uma família de objetos, relacionados ou dependentes, sem especificar suas classes concretas, o que permite repassar a instanciação para as subclasses (Gamma *et al.*, 2003). Do contrário, um cliente que precisa de um objeto

teria que realizar uma chamada ao seu construtor e assim especificar a classe concreta que ele instancia, o cliente chama um método abstrato (Fábrica de Métodos) especificado em alguma classe abstrata (ou *interface*) e a subclasse concreta decide que tipo exato de objeto criar e retornar. Caso ocorram mudanças na subclasse concreta que cria o objeto, é possível mudar a classe do objeto criado sem que o cliente tome conhecimento.

Motivação: No desenvolvimento de algumas aplicações, é preciso encapsular os detalhes de criação do objeto visando preservar o baixo acoplamento e estabilidade de uso do mesmo (Gamma *et al*., 2003). Apesar da criação de objetos ser uma questão do uso de um novo paradigma, nem todos os objetos são construídos facilmente. Podem ocorrer situações em que um tipo de objeto depende de outro tipo, ou alguns passos de inicialização precisam ser tratados fora do construtor.

A criação direta de objetos pode inadvertidamente confundir e reduzir a independência do código de chamada, caso algum dos elementos certos para criação do objeto não estejam disponíveis. No momento da chamada do objeto, algumas situações podem ser desconhecidas, tais como o conjunto de argumentos do construtor ou a execução de algumas restrições. Fazer que isto aconteça, aumenta a complexidade do código de chamada e reduz sua estabilidade, isso faz com que as mudanças nos detalhes de criação sejam mais difíceis de implementar.

Assim, é viável encapsular os detalhes concretos da criação de objetos dentro de uma fábrica de métodos, em vez de deixar os clientes da aplicação criarem o objeto. A fábrica de métodos libera os clientes da aplicação de criar quaisquer objetos que utilizem, tornando os clientes mais fáceis de entender e manter (Buschmann,2007). O padrão simplifica a instanciação de objetos cujo construtor não pode conter sua lógica de criação.

A estrutura básica do padrão Fábrica de Métodos é representada na Figura 3-12, os participantes do padrão são os seguintes:

- Produto: define a *interface* dos objetos criados pelo método fábrica;
- ProdutoConcreto: implementa a *interface* Produto;
- Criador: declara o método fábrica que retorna um objeto do tipo Produto. Algumas vezes o Criador não é apenas uma *interface*, mas pode envolver uma classe concreta que tenha uma implementação default para o método fábrica para retornar um objeto com algum tipo ProdutoConcreto *default*;

- CriadorConcreto: redefine um método fábrica para retornar a uma instância de ProdutoConcreto.

Algumas vantagens devem ser consideradas com o uso do padrão:

- Eliminam a necessidade de colocar classes específicas da aplicação no código: Assim, o código só lida com a *interface* Produto e pode, portanto funcionar com qualquer classe ProdutoConcreto. Na implementação, desenvolvedor lida apenas com as superclasses ou *interface*, para conhecer os métodos e atributos de uma forma genérica, o que permite manipular as subclasses ou implementações de forma particular;
- Maior flexibilidade para as classes: criar objetos em uma classe que possui método fábrica torna-se mais flexível que fazer de forma separada. O método fábrica funciona como uma conexão para que uma das subclasses pode prover uma versão estendida de um objeto;
- Conecta hierarquia de classes paralelas: os métodos fábricas não precisam ser chamados somente por Criadores. Os clientes podem fazer uso de métodos fábricas especialmente no caso de hierarquia de classes paralelas.

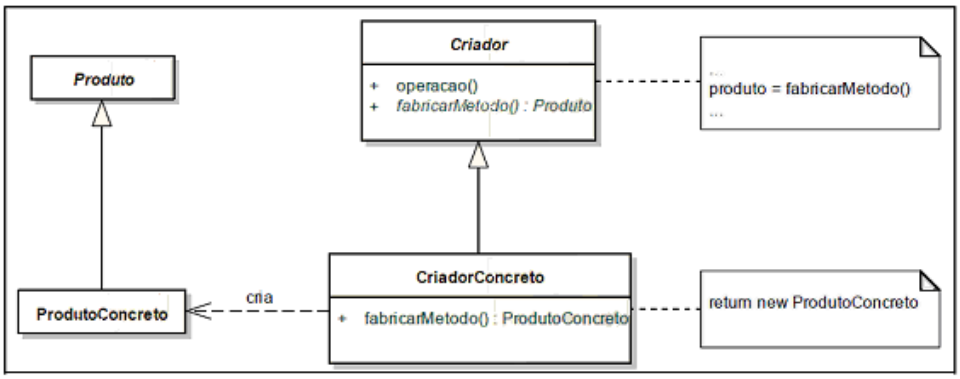

Figura 3-12 Estrutura genérica do padrão Fábrica de Métodos.

FONTE: Catálogo de padrão arquitetural (Prodepa, 2010a).

CAPÍTULO 4

ARQUITETURA PROPOSTA

O capítulo 4 apresenta a arquitetura proposta e sua aplicação no desenvolvimento de um sistema de controle de acesso distribuído.

A arquitetura proposta seguiu o processo de desenvolvimento citado no capitulo 2, sessão 2.7 e foi aplicada na construção de um sistema de controle de acesso. Esse sistema foi o primeiro entre os muitos que aplicou a solução proposta no seu desenvolvimento.

4.1 DESCRIÇÃO DA ARQUITETURA

Diante da necessidade de prover um meio de reutilização de componentes no desenvolvimento de sistemas distribuídos web, propõe-se uma arquitetura baseada na plataforma JEE (JEE, 2011), plataforma disseminada pela indústria de tecnologia, por se tratar de uma tecnologia aberta e padrão *web*. A plataforma JEE é projetada para fornecer suporte do lado servidor e do lado cliente para desenvolvimento de aplicações distribuídas para *web*. Algumas vantagens são alcançadas com a utilização da plataforma JEE, uma delas é a portabilidade oferecida pela linguagem Java que é normalmente fator de qualidade essencial.

A arquitetura proposta é aplicada nos sistemas desenvolvidos em uma empresa estadual pública de grande porte, por meio de um *framework* definido para uso interno (Muiraquitã). A arquitetura padrão e o *framework* relacionado têm a finalidade de suprir a necessidade de um meio de reutilização de componentes e arcabouços no ciclo de construção de sistemas na fábrica de *software* da empresa.

A arquitetura proposta para a construção de sistemas distribuídos utiliza o padrão arquitetural MVC (*Model-View-Controller*), que, por definição, desacopla as camadas provendo maior flexibilidade. Esta arquitetura de *software* tem como principal objetivo desenvolver sistemas e integrá-los de maneira distribuída a outros ambientes independe de plataforma ou linguagem. Em termos gerais, a arquitetura proposta busca atender alta coesão e baixo acoplamento entre os componentes, pois cada componente possui uma *interface* que define os serviços que devem ser implementados.

Alguns *frameworks* abertos foram adotados como padrões na arquitetura, como o *Jboss-Seam*, EJB3 e *Hibernate*, a reutilização de seus componentes proporcionou aumento na produtividade. A seguir, são explicadas essas tecnologias:

- EJB3: permite que os objetos chamados de *enterprise beans* sejam expostos como serviços *web*, de modo que seus métodos possam ser invocados por outro aplicativo J2EE e também por aplicativos escritos por outras linguagens de programação em diferentes plataformas (Burke & Moson-Haefel, 2007);

- Jboss-Seam: unifica o modelo de componentes do JSF e do EJB3, além de eliminar o código de integração, o que permite que o desenvolvedor dispense mais atenção às regras de negócio (Yuan, 2007);
- *Hibernate*: é responsável por mapear objetos Java em tabelas de um banco de dados relacional, estreitando o *gap* conceitual que existe entre os dois, deixando o desenvolvedor livre para se concentrar no negócio da aplicação (Bauer & King, 2006).

Alem dos *frameworks* citados acima, a arquitetura utiliza um *framework* interno, o Muiraquitã que é composto por três bibliotecas:

- pacote-*architecture*: fornece uma biblioteca de classes reutilizáveis comum a todas as aplicações desenvolvidas sob a Arquitetura, para prover recursos básicos que possibilitam que o projeto mantenha o foco no negócio em vez da tecnologia. Alguns exemplos desses recursos são: criação de relatório, criação de DAOs genéricos, criação de DTOs, entre outros;
- pacote-*utils*: fornece uma biblioteca de classes que oferecem alguns recursos adicionais já definidos pela linguagem Java, porém adaptados à Arquitetura como, por exemplo, manipulação de arquivos, datas, coleções entre outros;
- pacote-*exception*: responsável por padronizar as exceções decorrentes do tratamento de erros e validações exigidos pela arquitetura.

A descrição apresentada nas sessões 4.3.1 e 4.3.2 foi adaptada do Documento de Referência da Arquitetura (Prodepa, 2010b) que foi resultado do convênio UFPA/PRODEPA com o propósito de padronizar a metodologia de desenvolvimento de *software* na empresa.

4.1.1 Descrição Estrutural

A descrição estrutural é composta por diagramas que representam, em diferentes níveis de abstração, como o sistema está estruturado em termo de seus componentes, *interface*s e conexões entre os componentes.

A seguir, cada componente é descrito em termos de responsabilidade na Arquitetura.

Web Seam: este componente utiliza o JBoss Seam e implementa as questões de *interface* gráfica (de aplicações *Web*), onde se obtém referências para a *interface* Modulo*Facade*Local do componente *Facade*, com a finalidade de invocar as funcionalidades da aplicação.

***Facade* (Fachada)**: componente que funciona como "porta de entrada" para efetuar uma transação. A fachada é acessada pela camada cliente, ou seja, toda ação efetuada nos clientes é recebida pela fachada e delegada ao componente Serviço para ser executada, via *interface* ServicoLocal.

Serviço: componente que implementa os serviços disponibilizados pela aplicação, por meio do componente *Facade*, para as aplicações cliente. Os serviços da aplicação necessitam manter dados persistentes da aplicação e, para isso, utilizam o componente DAO (por meio da *interface* IEntityDao). Além dessas dependências, em um contexto onde se esteja utilizando DTOs para a troca de dados entre os componentes, o componente Serviço precisa utilizar o componente Converter para converter as entidades (do componente *Entity*) para os DTOs.

Business: quando necessário, o componente Serviço delega ao *Business* todas as validações necessárias para que um caso de uso seja validado de acordo com as regras de negócio da aplicação. O componente *Business* pode, assim como o Serviço, precisar dos dados persistentes durante a validação. Assim, ele também pode acessar o componente DAO por meio das duas *interface*s.

Entity: componente que representa o modelo de classes persistentes do domínio da aplicação (entidades). As classes nesse contexto são também conhecidas como classes POJO (*Plain Old Java Objects*), contendo apenas os atributos do domínio de negócio e métodos *get*, *set* e construtor. Cada entidade deve estender a classe AbstractEntity definida no *framework* Muiraquitã.

DTO: componente que implementa o padrão *Data Transfer Object*. Este componente é responsável por portar dados a serem trafegados entre diferentes camadas da aplicação, que representam objetos "peso leve" (do inglês, *light weight*).

Converter: componente que tem a função de traduzir os DTOs em entidades e vice-versa. Ao expor DTOs no componente *Facade*, isola-se o modelo de negócio do mundo externo, podendo publicar um serviço *web* sem expor o modelo de negócio da aplicação a terceiros.

DAO: componente que implementa o padrão *Data Access Object*, o qual é um padrão conhecido na literatura responsável por intermediar o acesso aos dados armazenados em um banco de dados (componente SGBD). Cada DAO deve possuir uma *interface*, que especifica os métodos de manipulação de dados. Os serviços acessam apenas as *interface*s dos DAOs, desconhecendo a implementação utilizada.

SGBD: um sistema de gerenciamento de banco de dados utilizado para armazenar as informações persistentes da aplicação, por exemplo, PostgreSQL ou Oracle.

A Figura 4-1 apresenta a visão geral em componentes da arquitetura proposta, define em alto nível de abstração, cada componente que faz parte da arquitetura, bem como suas *interface*s (providas e requeridas) e as interações com outros componentes.

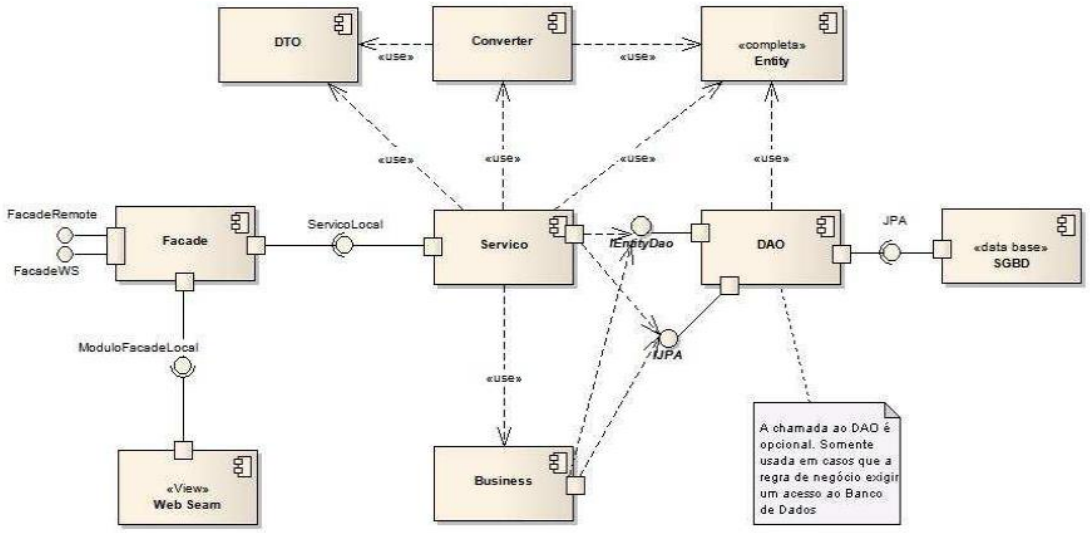

Figura 4-1 Diagrama de Componentes da Arquitetura.

FONTE: Documento de Referência (Prodepa,2010b)

A Figura 4-2 ilustra o diagrama de implantação do sistema de controle de acesso (aplicação servidora) e as *interface*s disponíveis para as aplicações clientes, dependendo da linguagem utilizada por tais aplicações. A aplicacao_cliente2, desenvolvida na linguagem de programação Java, acessa os serviços disponibilizados pela aplicação servidora por meio da *interface* remota (*Facade*Remote); a aplicação_cliente1, desenvolvida em PHP, obtém os serviços da aplicação servidora por meio da *interface web service*. A camada cliente da própria aplicação servidora disponibiliza suas funcionalidades acessando a *interface* local (*Facade*Local).

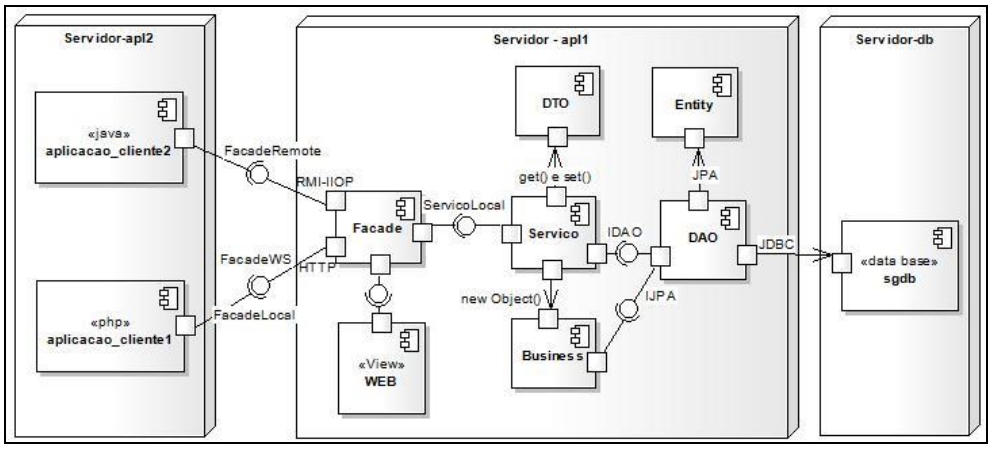

Figura 4-2 Diagrama de implantação - Integração entre as aplicações distribuídas.

Em uma visão mais detalhada da arquitetura são apresentadas as classes que realizam as *interface*s em cada componente, bem como os conectores entre componentes e *interface*s.

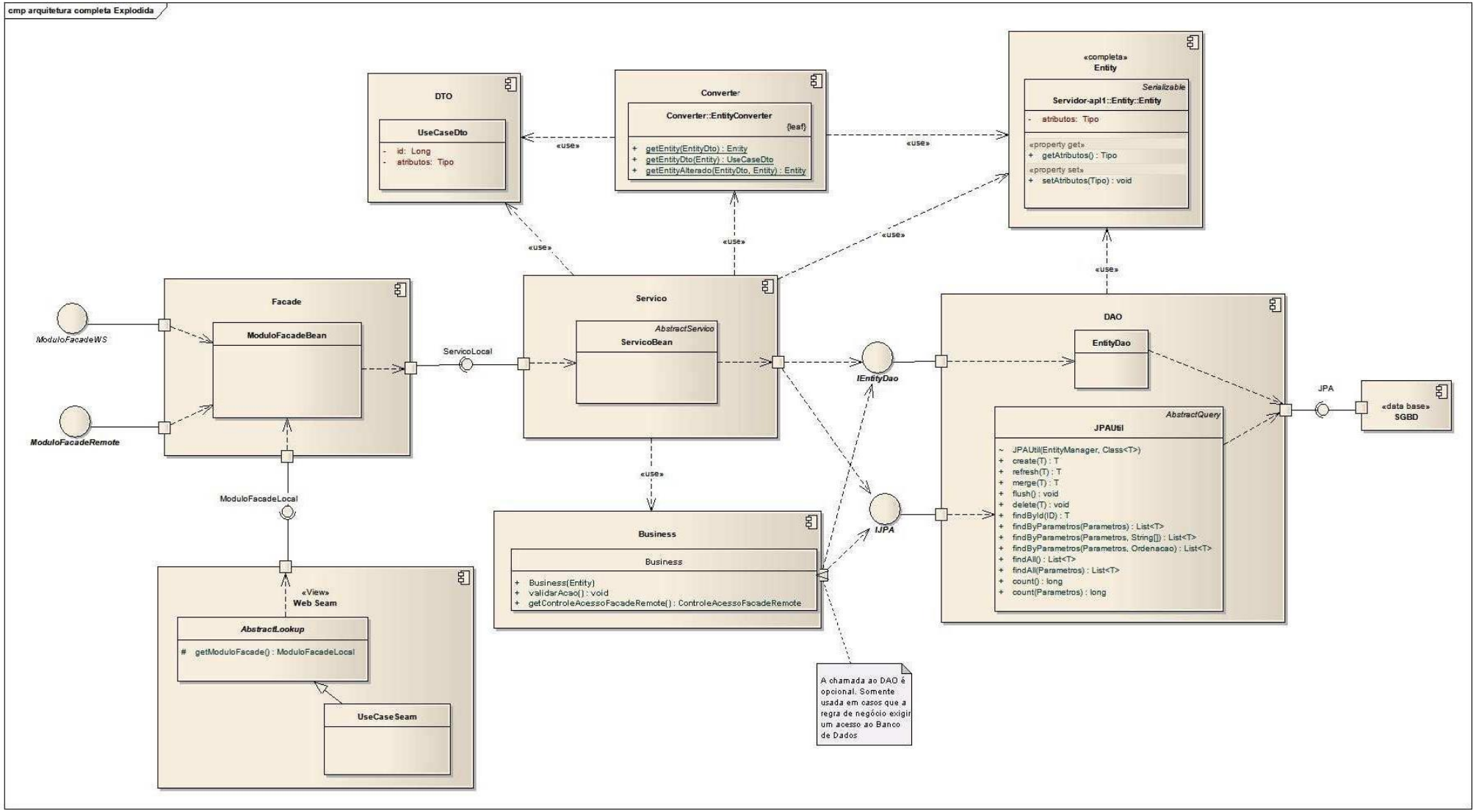

Figura 4-3 Diagrama de Componentes da Arquitetura (Visão Detalhada).

FONTE: Documento de Referência (Prodepa, 2010b)

A seguir, cada componente é detalhado em termos das classes que o compõe, bem como a descrição do papel que estas representam para o componente:

Web Seam: composto por componentes UseCaseSeam, o qual contém as informações dos formulários que devem ser repassadas à Fachada. Um UseCaseSeam herda de AbstractLookup, é essa herança que permite recuperar a referência para a fachada que expõe a funcionalidade requerida pelo cliente. O termo UseCase é substituído pelo nome do caso de uso/operação que o componente referencia;

Facade: composto pelas classes que implementam as fachadas, as classes nomeadas Modulo*Facade*Bean, sendo que o termo Modulo é substituído pelo nome do módulo a ser disponibilizado. Cada fachada disponibiliza um conjunto de serviços;

Serviço: composto pelas classes que implementam os serviços da aplicação, denominadas ServicoBean. Os serviços são providos para as fachadas por meio das *interface*s ServicoLocal, as fachadas mantém uma referência para essas *interface*s e delega a requisição da operação. As classes ServicoBean implementam as *interface*s ServicoLocal;

Business: composto pelas classes que tratam das validações sobre as regras e restrições de negócio, denominadas *Business*. Estas classes são chamadas diretamente pelas classes ServicoLocal. As classes Business podem ocasionalmente necessitar de acesso aos dados persistentes da aplicação, nesses casos, as classes *business* podem utilizar a *interface* IEntityDao. As classes b*usiness* implementam o método validarAcao. É neste método que devem ser implementadas as validações realizadas com as regras de negócio;

Entity: composto pelas classes que definem o modelo de dados (domínio), isto é, as entidades da aplicação;

DTO: composto pelas classes que implementam os DTOs para cada serviço;

Converter: composto pelas classes responsáveis por realizar a conversão dos dados das entidades para os DTOs. Utiliza-se um *Converter* específico para cada transformação;

DAO: composto pela hierarquia que representa a estrutura dos DAOs e uma classe utilitária, JPAUtil.

O componente SGBD não é detalhado, pois está fora do escopo de desenvolvimento, ou seja, são utilizados componentes externos.

4.1.2 Descrição Comportamental

A descrição comportamental apresenta o comportamento da arquitetura com o passar do tempo, ou seja, a maneira e seqüência como os componentes da arquitetura trocam mensagens para realizar uma operação que realize um corte vertical na arquitetura. Como cada projeto possui operações distintas, as operações cadastrar, alterar e excluir (as quais normalmente ocorrem em todos os projetos) ocorrem de maneira muito semelhante e representam bem a troca de mensagens entre os componentes. Assim a operação alterar foi escolhido para exemplificar essa interação.

A operação alterar inicia com a requisição do usuário para alterar os dados de uma entidade relativa a operação do CRUD ao formulário *web* de *interface* com usuário. Essa requisição é encaminhada ao componente EntitySeam, o qual representa uma classe contendo os dados relativos ao formulário de entrada dos dados para o caso de uso realizado. Em seguida, o componente UseCaseSeam localiza a fachada referente ao método desejado e encaminha a requisição de alteração para o componente Modulo*Facade*Bean.

Na fachada, a requisição é encaminhada para o serviço que implementa a operação no componente ServicoBean. Na implementação do serviço, o componente ServicoBean realiza a chamada ao componente de conversão do DTO recebido como parâmetro, o componente EntityConverter com o método getEntityAlterado, para a entidade relativa ao DTO.

Em seguida, o componente ServicoBean envia a requisição para validações necessárias da operação para o componente com o método validarAcao. O componente *Business* realiza a autenticação no sistema externo de controle de acesso por meio da fachada ControleAcesso*Facade*Remote. Caso a validação tenha sucesso, o componente ServicoBean realiza uma chamada ao método alterar para o DAO relativo *Business* a classe, ou seja, o EntityDao.

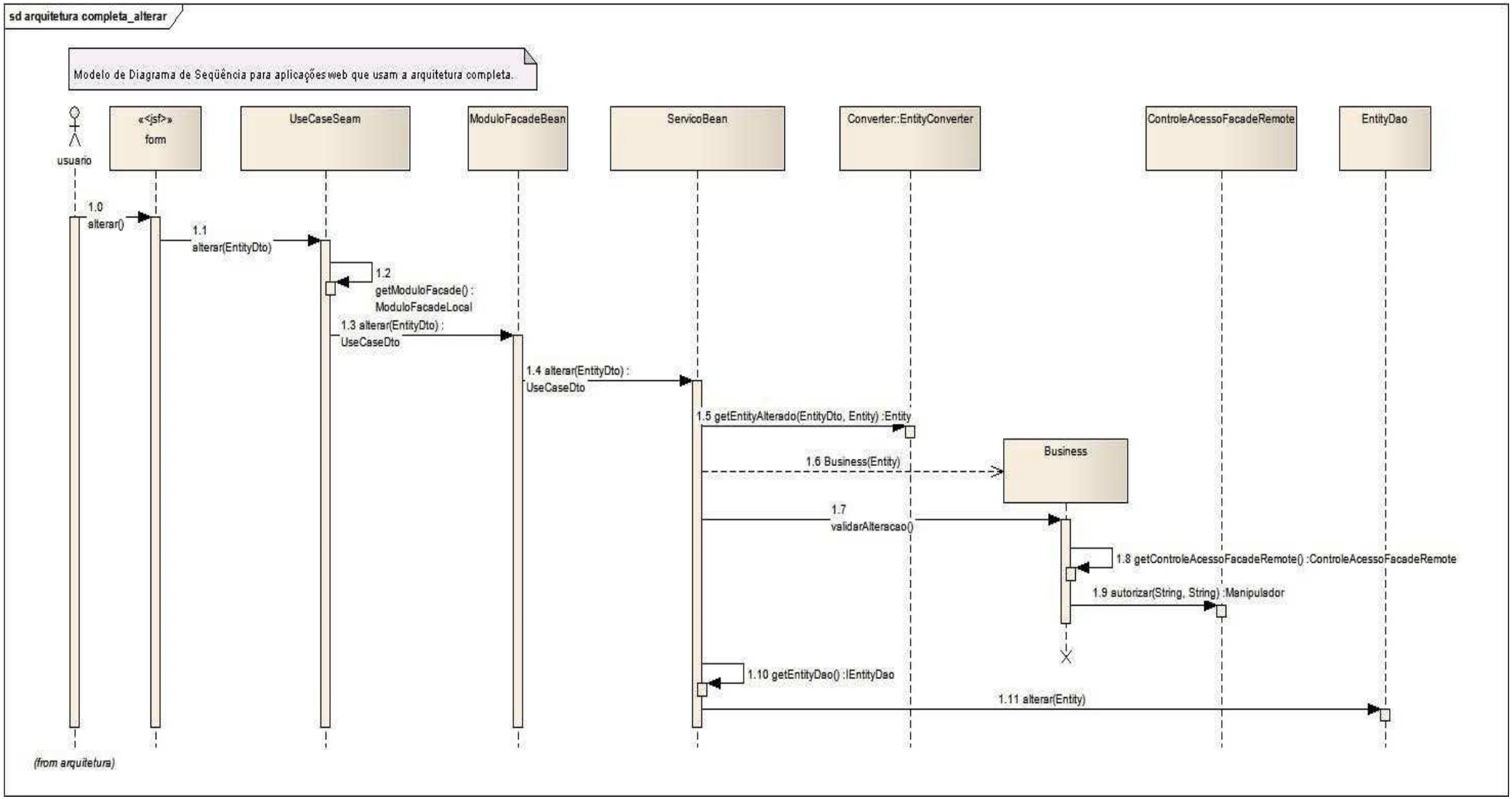

Figura 4-4 Diagrama de Sequência da operação Alterar .

FONTE: Documento de Referência (Prodepa, 2010b)

Na Figura 4-5, é ilustrado um diagrama de seqüência para representar uma visão dinâmica de como ocorre a comunicação entre as duas aplicações distribuídas e dos recursos que são utilizados.

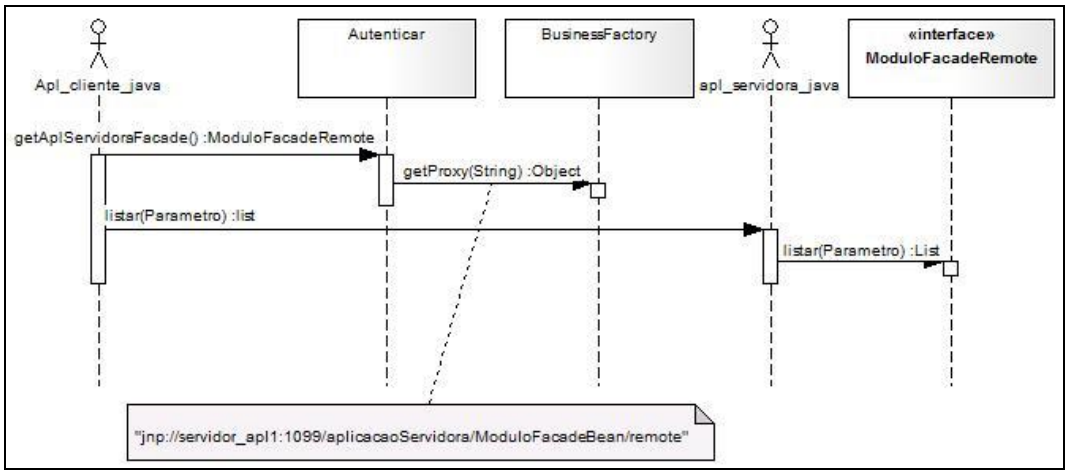

Figura 4-5 Diagrama de seqüência - Integração entre aplicações distribuídas.

Diversas aplicações utilizam serviços disponibilizados pelo sistema de controle de acesso. A maioria dessas aplicações foi desenvolvida seguindo a arquitetura, como por exemplo, o Sistema de Protocolo e o Sistema de Patrimônio do Estado do Pará.

4.2 APLICAÇÃO DA ARQUITETURA

O objetivo da implementação desse serviço é prover uma solução para unificar o mecanismo de controle de acesso dos sistemas desenvolvidos para o Estado. O cliente é a própria empresa que utilizará a aplicação para centralizar o controle de acesso às aplicações desenvolvidas pela empresa. Em linhas gerais a aplicação de acordo com o cliente deve:

- Disponibilizar uma interface que, independente das tecnologias adotadas pelos softwares que utilizarão o serviço, possibilite a autenticação dos usuários e a verificação de suas permissões para uso das funcionalidades nos diversos softwares;
- Centralizar o cadastro de todos os usuários e as respectivas permissões conforme o grupo de usuários e os softwares que eles precisam interagir;

- Estabelecer uma comunicação com os outros sistemas e validar o usuário ao informar o seu identificador e sua senha, provendo as permissões do usuário ao software;
- Fornecer interface de interação com o usuário restrita ao administrador do Controle de Acesso, que será responsável pela gerência dos dados da base da solução (usuários, sistemas, permissões e grupos de usuários).

Com o auxílio do cliente, uma síntese sobre a necessidade, impacto e benefícios com a aplicação foi elaborada, e apresentada no Quadro 4-1.

Quadro 4-1 Descrição da Necessidade.

A necessidade	Atualmente todos os *softwares* precisam implementar o seu próprio controle de acesso. Como conseqüência, além do retrabalho, há uma fragmentação da base de usuários implicando em inconsistências e dificuldade para administrar os dados de cada usuário.
Afeta	A produtividade na análise e desenvolvimento de novas soluções, além de requerer usuários administradores para cada *software*.
O seu impacto é	Com a solução não será mais necessário contemplar as funcionalidades de controle de acesso a cada novo *software* a ser construído. Além disso, a base de usuários será unificada.
Benefícios com a solução	Redução do tempo para o desenvolvimento de novas soluções quando estas fizerem uso do serviço de controle de acesso provido pela aplicação. Facilidade para manter a base de usuários de todos os sistemas.

4.3 SERVIÇO DE CONTROLE DE ACESSO

Esta seção apresenta os requisitos funcionais e não-funcionais do sistema que foram decisivos na definição da arquitetura desta pesquisa.

4.3.1 Visão de Casos de Uso

O diagrama de *use case* descreve graficamente como ocorre uma interação típica entre um ator e um sistema. Cada um dos *use case* descritos abrange uma função disponível ao usuário e define um objetivo do projeto (Larman, 2007). Esta seção descreve os cenários mais significativos da especificação dos requisitos funcionais por meio da apresentação dos casos de uso do sistema de controle de acesso.

Todos os casos de uso estão relacionados a um requisito funcional correspondente e representa um dos três principais cenários:

- **Autenticar usuários:** Um *software* fornecerá para a solução um identificador e senha de um usuário e a solução deverá verificar se o usuário pode acessar;
- **Verificar permissão de usuários:** Verificar se um usuário que foi devidamente autenticado possui permissão para realizar determinada operação no *software*;
- **Manter base de controle de acesso:** A solução deve permitir que um usuário administrador mantenha (incluir, excluir e consultar) os dados contidos na base de controle de acesso: usuários, permissões, grupos de usuários e sistemas.

Os atores envolvidos nos cenários apresentados são:

- **Administrador:** Funcionário da central de atendimento da empresa, responsável em atender os diversos órgãos (clientes) que utilizarão o controle de acesso. Este funcionário receberá uma solicitação de acesso (telefone, e-mail, ofício) a determinado sistema e ele por meio do sistema de controle de acesso irá cadastrar o usuário e lhe atribuir as permissões competentes ao seu perfil. É responsabilidade do administrador o cadastro e manutenção de todos os usuários e as respectivas permissões conforme o grupo de usuários e os *software*s que eles precisam interagir;
- **Usuário final:** Usuário de sistemas externos que por meio do fornecimento de *login* e senha obtém ou não permissão de acesso no sistema que acessa o controle de acesso;
- **Sistema:** Sistema externo que acessa o serviço de controle de acesso para autenticar seus usuários.

A Figura 4-6 ilustra os cenários e os atores da aplicação.

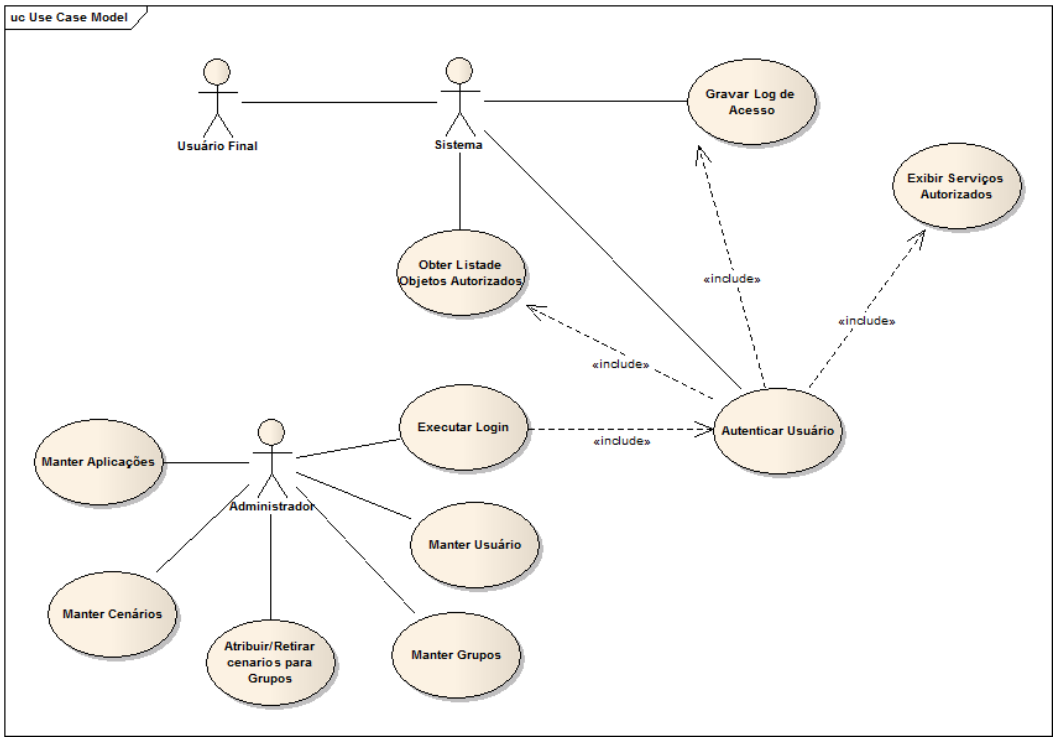

Figura 4-6 Caso de Uso do sistema de controle de acesso

Quadro 4-2 Descrição do Caso de Uso.

Use Case	Descrição
Autenticar usuário	Permite que o usuário externo (sistema) saiba os recursos que o usuário final tem acesso, ao ser informado um login (usuário e senha) válido. Os recursos são obtidos em forma de lista.
Exibir serviços autorizados	Permite que o usuário externo verifique se o usuário final validado tem acesso aos recursos que ele esta pretendendo acessar.Os serviços são exibidos em forma de lista.
Gravar log de acesso	Grava data e hora de acesso do usuário final ao sistema externo.
Obter lista de objetos autorizados	Permite que o usuário externo (sistema) saiba que objetos de apresentação o usuário final tem acesso. É obtido em forma de lista.
Manter aplicações	Inclui, altera, exclui e consulta aplicações no sistema. Cada aplicação deverá repassar seu identificador para obter acesso ao sistema de segurança.
Manter cenários	Inclui, altera, exclui e consulta cenários no sistema. Cenários é um conjunto de passos necessário para executar uma atividade.
Atribuir/retirar cenário para grupos	Possibilita dar permissão de acesso de cenários a grupos
Manter grupos	Inclui, altera, exclui e consulta grupos no sistema
Manter usuários	Inclui, altera, exclui e consulta usuários no sistema
Executar login	Permite ao usuário ter acesso ao sistema através de login e senha.

Além dos requisitos funcionais, o cliente definiu como requisitos não-funcionais:

- RNF01: Ao autenticar usuário, o sistema não deve levar mais que 100ms para retornar a lista de permissões;
- RNF02: Ao exibir serviços autorizados, o sistema não deve levar mais que 80ms para apresentar a lista de serviços autorizados pro usuário;
- RNF03: O sistema deve ser executável em qualquer sistema operacional.

4.3.2 Visão de Classes

A primeira fase para construir um projeto fundamentado na arquitetura é definir as entidades decorrentes do modelo de negócio. Essas entidades compõem o domínio da aplicação, também conhecidas como POJO (*Plain Old Java Objects*). Para o projeto de estudo de caso foi definido o modelo do domínio do negócio como ilustrado na Figura 4-7:

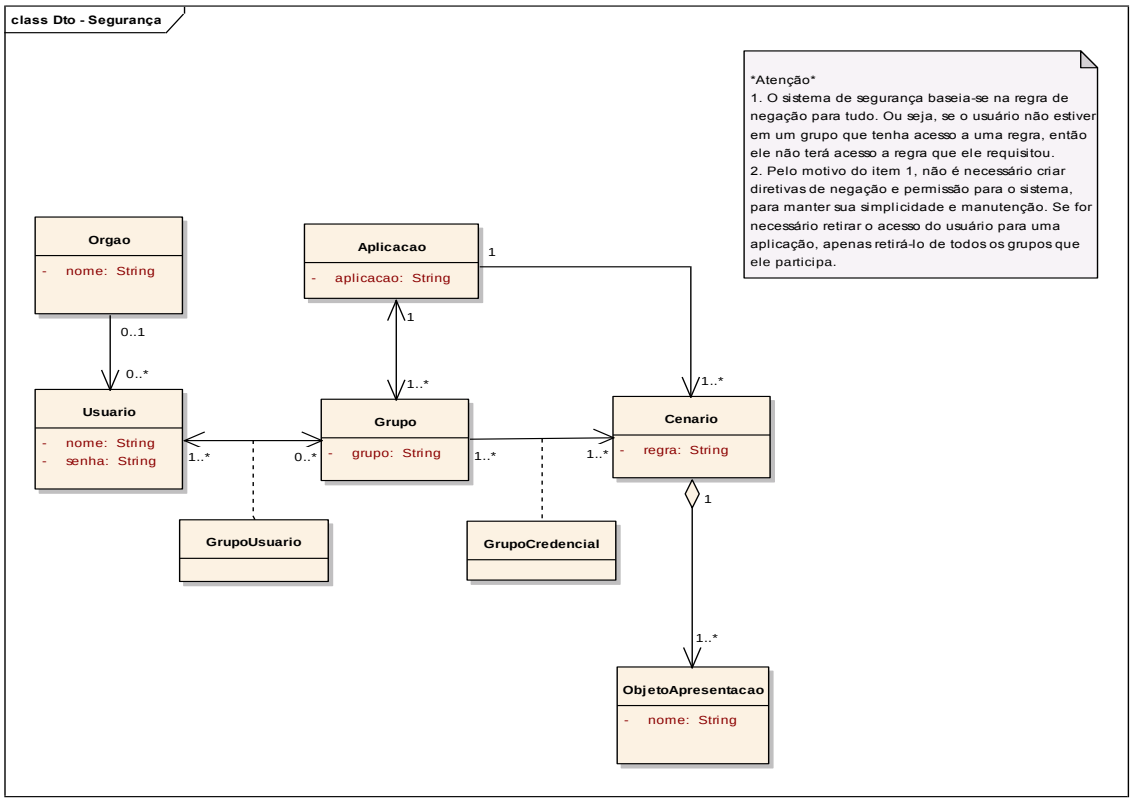

Figura 4-7 Modelo do domínio do negócio.

O diagrama acima ilustra um cenário onde um usuário de um determinado órgão, pode fazer parte de nenhum ou muitos grupos, a fim de obter acesso a um ou mais cenários de uma determinada aplicação, afim de que um ou mais objetos de apresentação sejam disponibilizados a ele.

CAPÍTULO 5
ANÁLISE DE RESULTADOS

O capítulo 5 descreve a elaboração e execução da avaliação da arquitetura. Além disso, apresenta os questionários utilizados, assim como os resultados da avaliação.

5.1 CARACTERIZAÇÃO DA AVALIAÇÃO

As decisões arquiteturais têm um profundo impacto sobre alcançar ou não os atributos de qualidade. Assim, um pré-requisito para avaliação de uma arquitetura é estabelecer claramente os atributos de qualidade, que são motivados pelas metas de negócio (Barbacci, 2002) e a especificação da arquitetura. Os três principais objetivos de uma avaliação de arquitetura são: a) Identificar e refinar os atributos de qualidade; b) Identificar e refinar as decisões arquiteturais de projeto; e c) Avaliar as decisões arquiteturais para determinar se satisfazem os atributos identificados.

O questionário aborda se a arquitetura proposta obedece às características definidas pela norma ISO/IEC 9126 (ISO, 2001), que é uma norma internacional e mais largamente aceita, que define os atributos de qualidade interna, externa e "em uso" do produto de *software*. Além de atributos específicos para avaliação de uma arquitetura de software do ponto de vista de seus componentes, como: modularidade, extensibilidade e reusabilidade.

O objetivo da análise apresentada neste capitulo é verificar, por meio da aplicação de questionário objetivo, se os atributos de qualidades relevantes satisfazem os objetivos propostos pela arquitetura conforme sessão 2.4 além de identificar pontos de melhoria. Os atributos de qualidade avaliados são ilustrados na Figura 5-1:

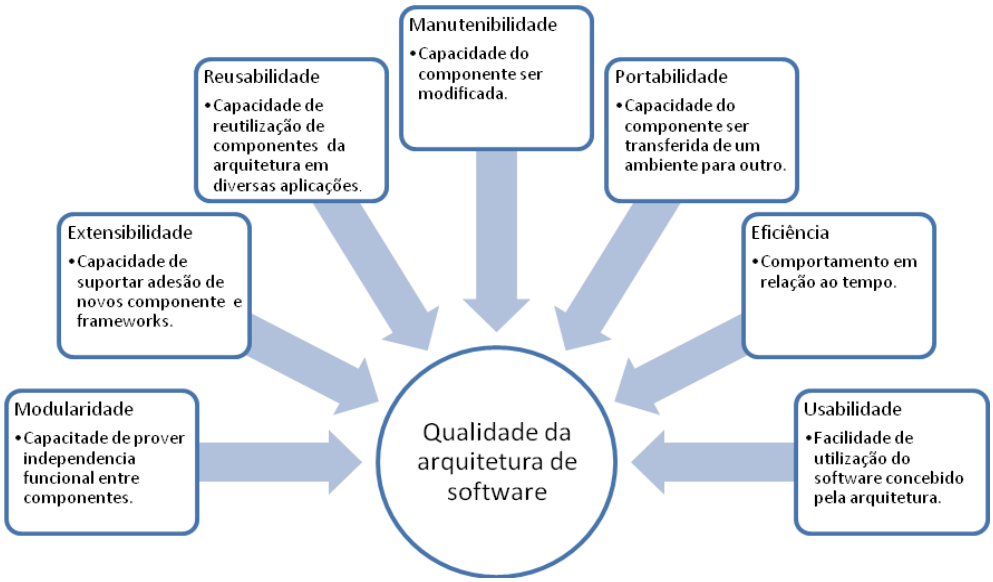

Figura 5-1 Atributos de qualidade da arquitetura de software.

A avaliação foi aplicada no ambiente da fábrica de *software* da empresa com a participação dos envolvidos no trabalho com arquitetura. Todos os entrevistados

possuem a experiência mínima de um ano com trabalho junto à arquitetura, cujo perfil pode ser:

- Analista: Identificar e documentar os requisitos do sistema (funcionais e não-funcionais) e a modelagem de casos de uso, delimitando o sistema e definindo sua funcionalidade; elaborar o diagrama de classe de negócio (conceitual) e protótipos de *interface* iniciais dos casos de uso; possuir total responsabilidade por um projeto do ponto de vista de negócio; levantar e documentar os requisitos de *interface*; definir o *layout* da aplicação; participar da análise de impacto quando das alterações de escopo (requisitos funcionais ou não-funcionais); planejar e documentar os casos de teste junto com o líder e realizar testes funcionais no sistema;

- Projetista: Atender aos requisitos de negócio do cliente; elaborar diagrama de seqüência e modelo de entidade e relacionamento; gerar código fonte e *script* do banco de dados; implementar o banco de dados no ambiente de desenvolvimento; gerar e documentar Componentes que possam ser reutilizados; receber, analisar e validar, junto ao analista e ao desenvolvedor, os casos de uso e diagrama de classe; garantir que os requisitos não-funcionais serão atendidos no projeto; disponibilizar todos os artefatos gerados pelo projeto no repositório, para a visualização da equipe e participar da análise de impacto quando das alterações de escopo (requisitos funcionais ou não-funcionais);

- Desenvolvedor: Desenvolver e testar componentes de acordo com os padrões adotados para o projeto; quando necessário, criar componentes de teste para possibilitar a realização dos testes; analisar os requisitos criticamente e aceitá-los para implementação; preparar ambiente de implementação; receber e analisar, junto ao projetista e ao analista, o modelo de projeto desenhado para o sistema; implementar funcionalidades dos casos de uso; realizar testes unitários no código gerado; comunicar ao analista a conclusão dos testes para que este inicie os testes funcionais e deixar disponível todos os artefatos gerados pelo projeto no repositório, para a visualização da equipe.

Os perfis citados são definidos pelo processo de desenvolvimento de *software* implementado em 2006 na empresa.

5.2 ANÁLISE DOS RESULTADOS

O questionário foi composto por 24 questões organizadas em seis categorias fundamentadas por Sommerville (2006) e Kandt (2006), apoiando-se nas diretrizes da ISO 9126 (2001), que definem um conjunto de características para a avaliação de qualidade no produto de *software*. O questionário contém duas questões sobre o perfil do entrevistado, duas questões para característica modularidade, duas para extensibilidade, duas para reusabilidade, três para manutenibilidade, quatro para usabilidade, quatro para portabilidade, quatro para eficiência e a ultima questão do tipo subjetiva onde o entrevistado pode descrever por meio do seu ponto de vista, que ajustes podem ser feitos na arquitetura para melhorar a qualidade do produto final. Esta avaliação foi realizada com 15 funcionários da empresa com perfis de analista, desenvolvedor e arquiteto de software.

5.2.1 Quanto ao Perfil do Entrevistado

O perfil do entrevistado é uma característica fundamental para a credibilidade da avaliação, uma vez que é importante assegurar que o mesmo tenha o perfil adequado para responder de maneira segura sobre a arquitetura que é um artefato de trabalho comum na fábrica de *software* da empresa.

Independente do perfil, todos os entrevistados possuem conhecimento mínimo sobre a arquitetura, pois de acordo com a política institucional, participam de treinamentos sobre a arquitetura ao ingressar na empresa. A análise das repostas dos projetistas e desenvolvedores são mais relevantes nesta categoria pelo fato deles utilizarem a arquitetura diariamente desenvolvendo soluções dentro do processo de construção de *software*. Assim, esses usuários possuem maior probabilidade de encontrar problemas ou melhorias na arquitetura proposta.

Conforme ilustrado na Figura 5-2, entre os entrevistados, 7% são analistas, 40% projetistas e 53% desenvolvedores. O resultado total de 93% entre projetistas e desenvolvedores asseguram o melhor perfil e maior credibilidade nas respostas.

Figura 5-2 Perfil do Entrevistado.

5.2.2 Quanto ao Conhecimento do Entrevistado

O nível de conhecimento está dividido conforme apresenta a Figura 5-3: 7% dos entrevistados consideram seu conhecimento a cerca da arquitetura insuficiente, 33% consideram ter um conhecimento razoável, 53% consideram ter um bom conhecimento e 7% um conhecimento excelente.

Como pode ser observada na Figura 5-3 a percentagem de entrevistados que consideram seu conhecimento como excelente é muito tímido, isso se deve entre outros fatores ao grau de complexidade que a arquitetura abrange.

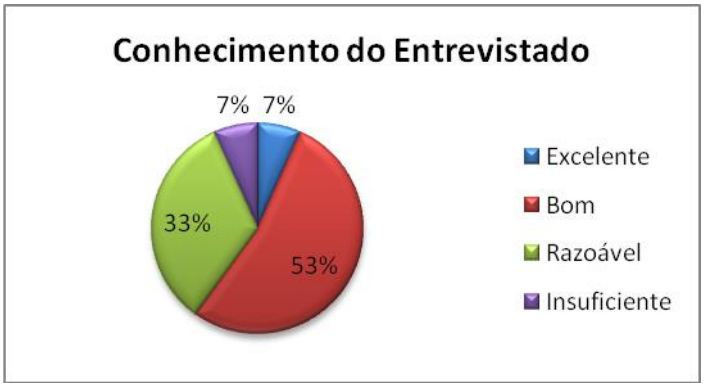

Figura 5-3 Conhecimento do Entrevistado.

Dentre os 7% que consideram seu conhecimento como insuficiente, todos são analistas, isso se justifica por meio de suas atribuições técnicas, uma vez que apesar de ter algum conhecimento sobre arquitetura, não trabalham com sua aplicação no seu dia-a-dia. Em compensação dentre os entrevistados que consideram ter um bom conhecimento, 88,89% são de desenvolvedores e projetistas conforme a Figura 5-4.

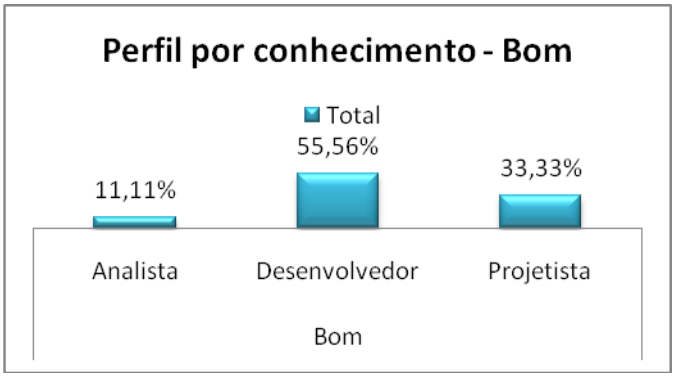

Figura 5-4 Perfil do Entrevistado que tem Bom conhecimento.

A partir da análise da figura 5-4, conclui-se que as respostas do questionário têm credibilidade devido o bom conhecimento a cerca da arquitetura avaliada pela maioria dos entrevistados, uma vez que se enquadram no melhor perfil, de desenvolvedor e projetista, que são os perfis que trabalham diretamente com a arquitetura no seu dia-a-dia.

5.2.3 Quanto ao atributo de Modularidade

De acordo com a opinião dos entrevistados a arquitetura atende a característica de modularidade satisfatoriamente conforme ilustra a Figura 5-5, sendo que 20% dos entrevistados consideram que atendem totalmente, contra 63% consideram que atendem, 17% consideram que atende pouco e 0% que não atende.

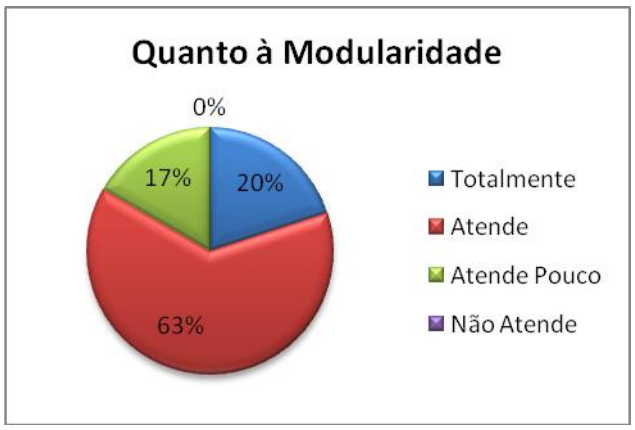

Figura 5-5 . Avaliação da característica de modularidade.

A arquitetura foi projetada pra prover alta coesão e baixo acoplamento e tem esse atributo alcançado na opinião de 83% dos entrevistados, porém alguns

apresentaram alguns pontos de melhoria para complementar este atributo, como injeção de independências e adoção de padrões específicos.

Por exemplo, para implementar uma função do tipo “listar objetos de um *combo”* as camadas *facade*, serviço e *business* são irrelevantes e podem ser desconsideradas para a implementação desta função, esta simplificação pode se tornar um padrão pronto para ser utilizado sempre no mesmo contexto, isto é, ao necessitar listar objetos de um *combo*.

Essas melhorias proveriam aumento na produtividade, entretanto precisa analisar sua viabilidade para não afetar a padronização do desenvolvimento como um todo.

5.2.4 Quanto à extensibilidade

Essa característica é muito vasta dentro do conceito de arquitetura de *software*, pois se refere à adesão de novos componentes, sendo que se entende por componentes desde uma camada até um *framework* completo, portanto está sendo avaliado a extensibilidade quanto aos *frameworks* que foram introduzidos na arquitetura, como Seam, EJB e os desenvolvidos dentro da solução como o Muiraquitã.

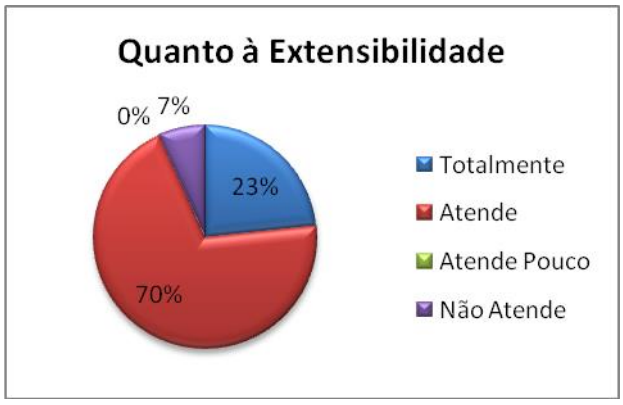

Figura 5-6 Avaliação da característica de extensibilidade.

Na avaliação, 23% acham que a arquitetura atende totalmente este atributo. 70% acham que atendem 0% que atende pouco e 7% que não atende (ver Figura 5-6). Dos 7% que dizem não atender, justificaram essa escolha relatando que a arquitetura apesar de possuir pontos de extensão por meio das *interface*s, da injeção de controle (aspectos), ainda possui outros pontos que ainda precisam ser trabalhados.

5.2.5 Quanto à reusabilidade

Este atributo tem maior peso quando se quer atingir maior produtividade no desenvolvimento, com a pesquisa observou-se que 97% dos entrevistados consideram que esta característica é atendida totalmente ou parcialmente no desenvolvimento com arquitetura de *software*.

Na Figura 5-7 se observa que 50% consideram que a reusabilidade é atendida totalmente, 47% consideram que atende bem e 3% que não atende.

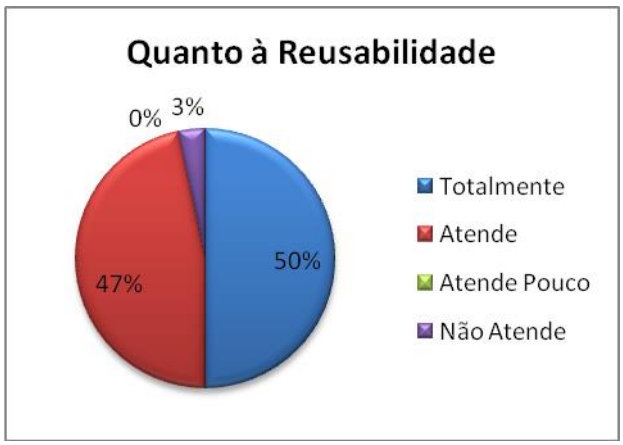

Figura 5-7 Avaliação da característica de reusabilidade.

Existem vários componentes que são reutilizáveis no *framework*, porém existem outros tantos que não estão em um repositório comum e estão indisponíveis para serem utilizados por todas as aplicações que seguem a arquitetura. Ademais, alguns entrevistados justificaram o "Não Atende" em detrimento da falta de documentação completa do *framework*, resultando desenvolvimento de componentes já existentes.

5.2.6 Quanto à manutenibilidade

A padronização no desenvolvimento é alcançada por meio da aplicação da arquitetura no desenvolvimento, isso diminui o esforço requerido para localizar e corrigir uma falha em um sistema em seu ambiente de operação. E esta afirmação é ratificada por meio do resultado apresentado na Figura 5-8, onde 19% consideram que com a aplicação da arquitetura, a facilidade na manutenção é totalmente atendida, 61% consideram que atende 20% que não atende e 0% que atende pouco.

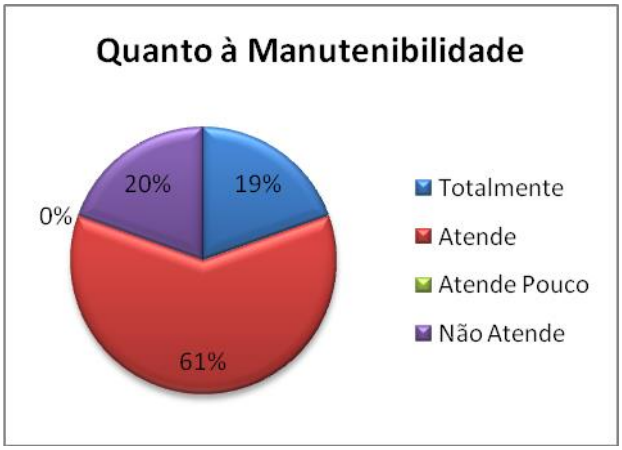

Figura 5-8 Avaliação da característica de manutenibilidade.

Conforme as respostas subjetivas dos questionários, a arquitetura facilita a manutenção, pois foca em pontos específicos. O problema é que nem todos os desenvolvedores seguem fielmente a arquitetura e a falta de obrigatoriedade resulta em codificação em desacordo com o padrão. Assim, quando é feita a manutenção por um desenvolvedor que não participou do projeto, acontece do código estar diferente do que é esperado. As camadas geralmente são implementadas, mas a forma de interligar essas camadas e até mesmo a forma de codificar torna-se complicadores para manter.

5.2.7 Quanto à usabilidade

Aqui foi avaliada a usabilidade do *software* concebido pela arquitetura, na visão de quem trabalha no processo de desenvolvimento deste *software*. Todos os perfis de entrevistados executam algum tipo de teste durante o ciclo de desenvolvimento, portanto tem conhecimento para responder as questões baseadas nesse contexto.

Ao ser questionado se para utilizar o sistema, o usuário precisa ter profundo conhecimento no domínio do *software*, 67% dos entrevistados responderam que o usuário não precisa possuir conhecimento para executar o *software* e 33% responderam que o usuário precisa ter um conhecimento parcial, conforme Figura 5-9:

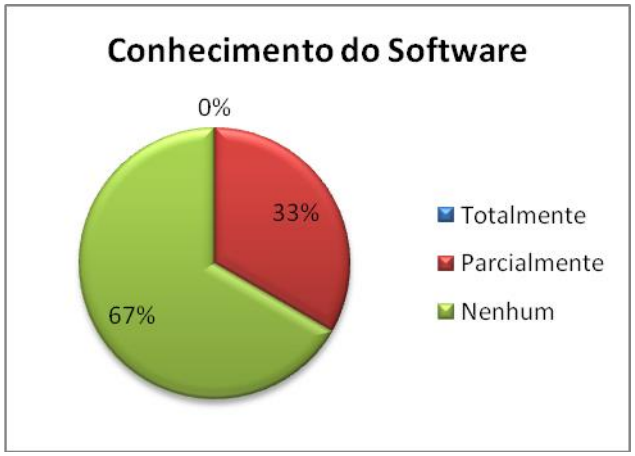

Figura 5-9 Nível de conhecimento para executar o sistema.

O alto nível de usabilidade é alcançado, pois já existe uma padronização de *interface* nos sistemas desenvolvidos no governo do Estado, o que facilita o manuseio.

As demais perguntas se referiram à capacidade de processamento, ao atendimento dos requisitos necessários para sua completa execução e à confiança transmitida ao usuário.

Dos entrevistados 58% consideram que o sistema atende totalmente o requisito de usabilidade, 40% consideram que atende, 2% que não atende e 0% que atende pouco (ver Figura 5-10).

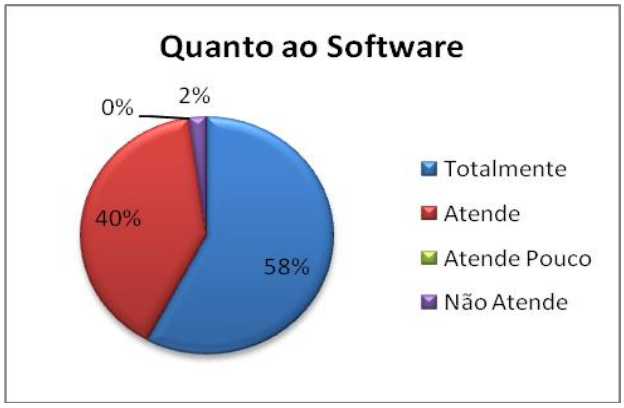

Figura 5-10 Avaliação da característica de usabilidade.

Não houveram repostas subjetivas para esse atributo a fim de justificar os 2% que indicaram o atributo como não atendido.

5.2.8 Quanto à portabilidade

A avaliação desta característica é a mais importante no contexto deste trabalho, pois alem de ser uma característica desejável em uma arquitetura de *software* é ela que vai indicar se a arquitetura é capaz de construir satisfatoriamente sistemas distribuídos.

Por meio de quatro questões que avaliaram este requisito, os entrevistados consideraram que 32% atende totalmente, 56% que atende, 12% que atende pouco e 0% que não atende, conforme Figura 5-11.

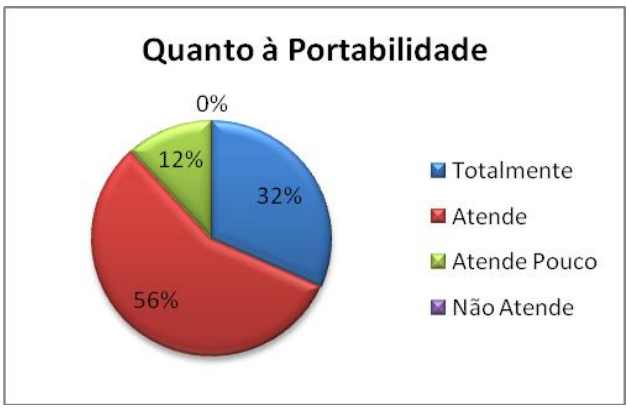

Figura 5-11 Avaliação da característica de portabilidade .

Apesar da grande maioria considerar que a arquitetura de alguma forma atende, 12% acham que a mesma atende pouco, por meio da análise da questão subjetiva, alguns entrevistados justificaram essa opinião colocando que apesar dos sistemas construídos nesta arquitetura poderem ser acessados por diferentes clientes e poderem rodar em diferentes ambientes computacionais, ainda apresentam um grau de dificuldade considerável quando o assunto é a mudança do servidor de aplicação.

A arquitetura até então trata fundamentalmente de aplicações JAVA, que é portável por natureza, que executa no servidor de aplicação Jboss 4.2.3 (Fleury, 2005), mas diante de alguns testes constatou-se certa dificuldade para mudar para uma versão mais recente. Foram feitos alguns testes isolados que apresentaram alguns problemas, em função de atualizações de *frameworks* que vem junto com essa nova versão do JBOSS. Seria completamente portável se isso não fosse uma preocupação. Quanto a sistema operacional é indiferente, pois roda em qualquer um. Apesar da dificuldade relatada, a arquitetura atende o requisito de portabilidade e constrói satisfatoriamente sistemas distribuídos, no ponto de vista do entrevistado.

5.2.9 Quanto à eficiência

Esta característica foi avaliada tanto no contexto do *software*, isto é, o desempenho alcançado em sistemas construídos com a arquitetura, como a eficiência durante o desenvolvimento com a arquitetura.

Nas questões relacionadas ao desempenho do sistema 30% responderam que o sistema atende totalmente este requisito, isto é, o tempo de resposta é adequado em relação ao volume de dados e à complexidade das funções, 60% consideraram que o sistema simplesmente atende este requisito, mas ainda pode obter melhorias, 5% consideram que o sistema atende pouco e 5% que não atende este requisito, conforme ilustra a Figura 5-12:

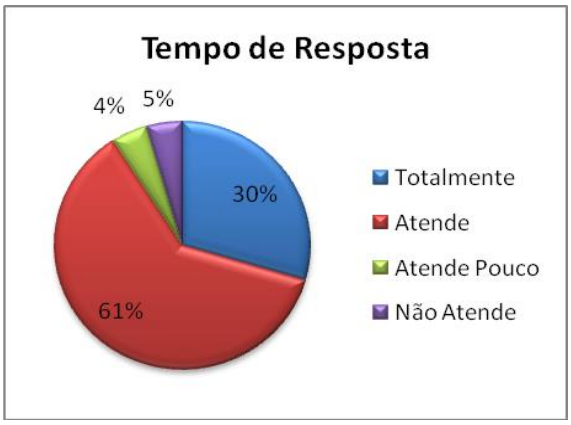

Figura 5-12 Avaliação da característica de desempenho.

Para questão relacionada ao tempo total para o desenvolvimento de sistema que aplicam a arquitetura na sua construção, 47% consideram que o tempo diminuiu, 48% que o tempo aumentou e 7% consideram que o tempo não aumentou nem diminuiu, isto é, não fez diferença, conforme Figura 5-13:

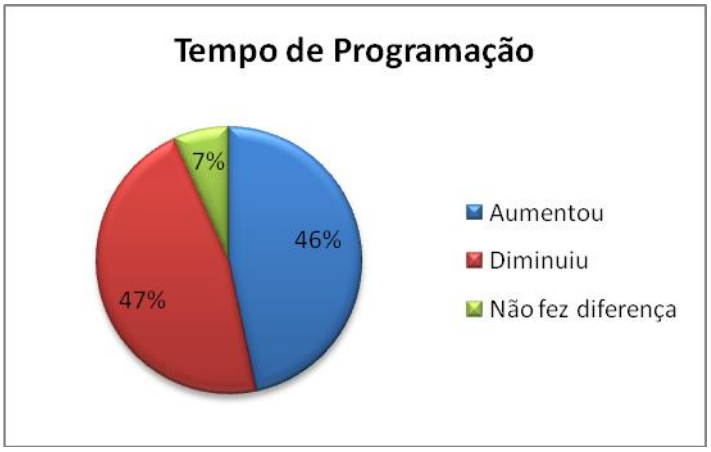

Figura 5-13 Tempo durante o desenvolvimento.

Ao tentar entender os motivos que levaram à aproximação dos resultados entre os que acharam que aumentou e diminuiu o tempo, foi aplicado um complemento do questionário onde questionava o tempo de experiência exata com a arquitetura e a justificativa para sua opção.

Como mencionado no inicio deste capitulo todos os entrevistados possuem experiência mínima de um ano no trabalho com a arquitetura, os mais experientes possuem três anos, pois a arquitetura foi institucionalizada no ano de 2009 na empresa. Esse curto espaço de tempo entre os menos e mais experientes foi fundamental para entender o porquê dos resultados terem quase se igualado.

Os menos experientes consideram que a arquitetura auxilia de forma positiva na produtividade, pois oferece componentes prontos para serem utilizados e também padroniza o desenvolvimento como um guia para o desenvolvedor que só precisa seguir as camadas sem dificuldade. Porém, quanto mais experiente o desenvolvedor, mais possibilidades ele enxerga para desenvolver o mesmo sistema de outras maneiras, por exemplo, para alguns casos não é necessário a criação das classes referente às camadas: *facade* e serviço, fazendo acesso diretamente ao DAO, neste caso o desenvolvimento fica mais rápido, pois diminui o número de classes que precisam ser criadas, esta decisão resulta na perda de modularização e ganho na produtividade.

Analisando os resultados apresentados em cada categoria concluímos que a arquitetura de uma forma geral apresentaram números satisfatórios, porém se faz necessário agregar possíveis melhorias afim de obter maior qualidade no produto final, principalmente no que diz respeito no aumento da produtividade.

Os pontos chaves desta avaliação no aspecto positivo foram as seguintes:

- Identificar gargalos na arquitetura que podem levar à construção de sistemas com baixo desempenho;
- Desativar camadas consideradas irrelevantes no contexto do negócio, por exemplo, camada serviço para criar um *combo* de objetos;
- Planejar novos padrões baseadas na avaliação subjetiva do usuário;
- Aperfeiçoar a arquitetura em termos de eficiência, tanto do ponto de vista do desenvolvedor, como no do usuário final.

5.2.10 Análise de Requisitos Não-Funcionais pela Perspectiva do Usuário Final

A definição da arquitetura é passo importante no alcance aos requisitos não-funcionais, requisitos estes que interferem diretamente na qualidade do produto e está diretamente relacionado ao usuário final.

O usuário final não executa diretamente nenhuma atividade do processo de desenvolvimento ou é responsável por qualquer artefato, entretanto o usuário representa o "cliente" do projeto de desenvolvimento, é para ele que o sistema está sendo desenvolvido. Deste ponto de vista, este passa a ser o papel de maior importância em todo o processo de desenvolvimento.

O questionário aplicado foi baseado no laudo de avaliação do produto, artefato este utilizado pela empresa para avaliação dos sistemas entregues ao usuário, esse laudo é um dos artefatos gerados durante o processo de desenvolvimento e esta em anexo para consulta.

As características avaliadas conforme o laudo são: usabilidade, funcionalidade, confiabilidade, eficiência, portabilidade sendo que as principais no contexto desta pesquisa são a portabilidade e eficiência, pois são requisitos não-funcionais exigidos pelo usuário durante a coleta de requisitos. Para cada um desses requisitos os entrevistados apontaram se o sistema de controle de acesso atende ou não os requisitos não-funcionais e também verifica se esse item não foi avaliado.

Todos os entrevistados fazem parte de uma equipe de cinco funcionários que atendem aos chamados internos e externos para inclusão de usuários em diversos sistemas desenvolvidos no Estado do Pará, como sistema de Protocolo e sistema de Patrimônio, o sistema de controle de acesso está sendo utilizado desde 2010.

Entre as opções existentes a maioria dos entrevistados considera que os requisitos estão sendo atendidos, conforme ilustra a Figura 5-14.

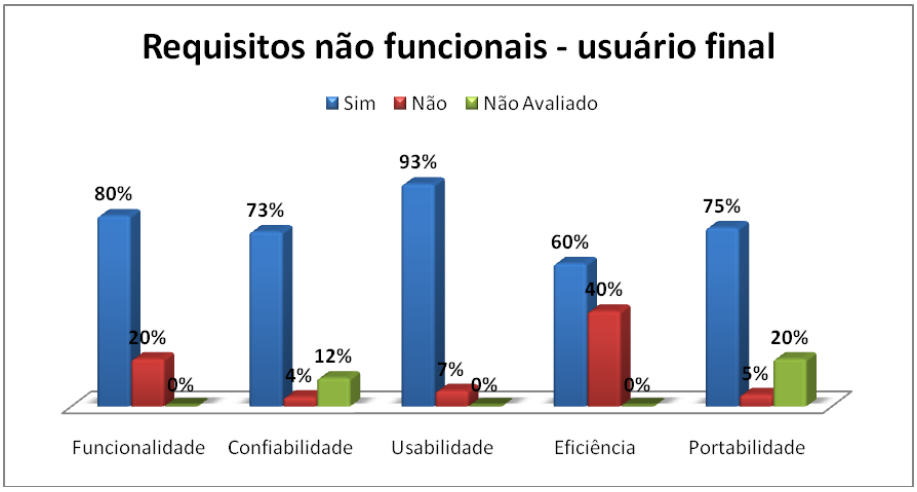

Figura 5-14 Requisitos não-funcionais.

Diante dos resultados apresentados, verifica-se que o sistema atende em mais de 60% todos os atributos avaliados, isso demonstra que o sistema foi bem aceito e atende aos requisitos estabelecidos pelo usuário.

Uma atenção especial deve ser dada aos requisitos de eficiência e portabilidade, pois estão diretamente relacionados aos requisitos não-funcionais exigidos pelo cliente. O requisito eficiência apesar da maioria afirma que existe um grau de eficiência no sistema, quase a metade discordou o que sugere uma adequação a fim de alcançar melhores resultados.

5.3 LIMITAÇÃO DA AVALIAÇÃO

A arquitetura apresentada nesse estudo foi aplicada no desenvolvimento de mais de 20 *software*s no Estado, sendo o sistema de controle de acesso apenas um ponto de partida dentro da fábrica de *software*. Apesar da avaliação dessa pesquisa ter acontecido em um ambiente real, não foi possível realizar avaliações quantitativas por meio da aplicação de ferramentas de coleta de métricas especificamente para desempenho, como por exemplo, o Jmeter (Hanzen, 2011) que permite fazer comparações entre sistemas e assim oferecer mais credibilidade aos resultados.

Outra limitação encontrada nesta avaliação é que as respostas podem ter sofrido alterações em detrimento de aspectos emocionais por parte dos entrevistados e como conseqüência pode ter ocorrido alterações no resultado obtido.

5.4 ANÁLISE DO PROBLEMA E SOLUÇÃO PROPOSTA PARA A ARQUITETURA

A arquitetura proposta foi aplicada no sistema de controle de acesso, porém foi observado que dentre os requisitos levantados, apenas os requisitos RNF01 e RNF02 não foram cobertos eficientemente pela arquitetura.

O requisito não-funcional priorizado pelo cliente foi a eficiência, verificou-se que à medida que pode resolver esta questão é a medida de tempo de resposta do sistema, ou seja, analisar quantas requisições o serviço consegue processar em menor período de tempo.

Como durante a fase de requisitos o cliente estabeleceu um valor de desempenho necessário para atender as necessidades, o arquiteto projetou a arquitetura para que fosse realizado um mínimo de 100 requisições por minuto baseando-se na sua experiência com esse tipo de aplicação. Esse valor foi utilizado como indicador de desempenho, porém diante dos resultados obtido com a avaliação, percebeu-se que o processamento das atividades estava abaixo do valor esperado, por isso, a arquitetura foi ajustada a fim de reverter este quadro.

5.4.1 Solução Proposta para a Arquitetura

As informações eliciadas durantes as reuniões de avaliação junto aos arquitetos foram importantes para propor os principais ajustes descritos abaixo:

- **Substituição da camada *Facade* pelo padrão *Application Service:*** A camada *Application Service* (Alur., 2003) é um padrão J2EE que centraliza a lógica de negócios de diversos componentes da camada em diversos grupos ou pontos de acesso. É a porta de entrada para efetuar uma transação. Funciona como ouvinte da camada de apresentação, ou seja, toda ação efetuada pelo usuário do sistema será “ouvida” pelo *Application Service* e repassada para o *Business Object* executar;

  A descentralização de acesso do *Application Service*, permite que o cliente tenha acesso apenas no domínio de seu interesse, isolando assim o acesso ao restante da aplicação, resultando em segurança e alto desempenho, uma vez que só é compilado o grupo de métodos referente ao serviço acessado. Diferentemente do padrão Facade que sobrecarrega o processamento por meio de um único ponto de acesso;

- **Substituição da camada *Business* pelo *Business Object:*** A camada *Business Object* é responsável por implementar a camada de lógica de negócios da aplicação e fornece os serviços para o objeto de negócio específico (Alur, 2003). As classes encontradas neste pacote são responsáveis por executar as transações que a aplicação cliente necessita e instanciar os componentes de persistência;

Esta camada utiliza tanto seus próprios métodos quanto os métodos disponibilizados por outros *Business Objects* permitindo o reuso dos serviços implementados.

- **Inclusão da camada *Factory:*** Esta camada é responsável por criar objetos que devem obedecer a certos critérios dentro de uma lógica de criação complexa (Larman, 2007). Este padrão permite que as criações dos objetos DAO sejam feitos de forma transparente. Uma das vantagens de utilizar esse padrão é que ela indica qual implementação de persistência (DAO) será utilizada pela aplicação a partir de um único ponto de acesso, o que diminui o retrabalho no caso de manutenção da aplicação;

De uma maneira geral, esta arquitetura além de mais simples adiciona performance no tratamento de seus objetos, interoperabilidade e reusabilidade, isso se deve principalmente aos fatores listados abaixo (ver Figura 5-15):

- para cada funcionalidade disponibilizada existe um *Application Service*, evitando a centralização em um único ponto de acesso e em conseqüência sua sobrecarga, resultando na diminuição no tempo de resposta no processamento das operações;
- inclusão da camada de *Business Object* que além de ser responsáveis pelas regras de negocio, permite que os serviços implementados no BO sejam reutilizados em outros BO permitindo um relacionamento cíclico entre os mesmos, evitando assim reescrita de código de uma mesma função em BOs distintos;
- inclusão da camada *Factory* que permite a partir de um único ponto indicar qual DAO será utilizado na aplicação, isso aumenta a produtividade na fase de desenvolvimento e manutenção. Em outras

arquiteturas se houver mudanças na implementação no DAO a ser utilizado, o desenvolvedor tem que apontar em cada BO que utiliza o DAO a nova implementação, ocasionando aumento de retrabalho. Esta camada também possui como vantagem a reutilização da instância do DAO que esta na memória auxiliando na melhoria do desempenho.

Depois dos ajustes elaborados, a nova solução foi implementada e aplicada no desenvolvimento de um serviço de geração e avaliação de mapas conceituais que são acessados via *Web Service* pelo Laboratório de Ensino de SQL (LabSQL) (Lino *et al.*, 2007) afim de testar os ajustes aplicados na arquitetura. A descrição detalhada do serviço citado pode ser encontrada no apêndice C deste trabalho.

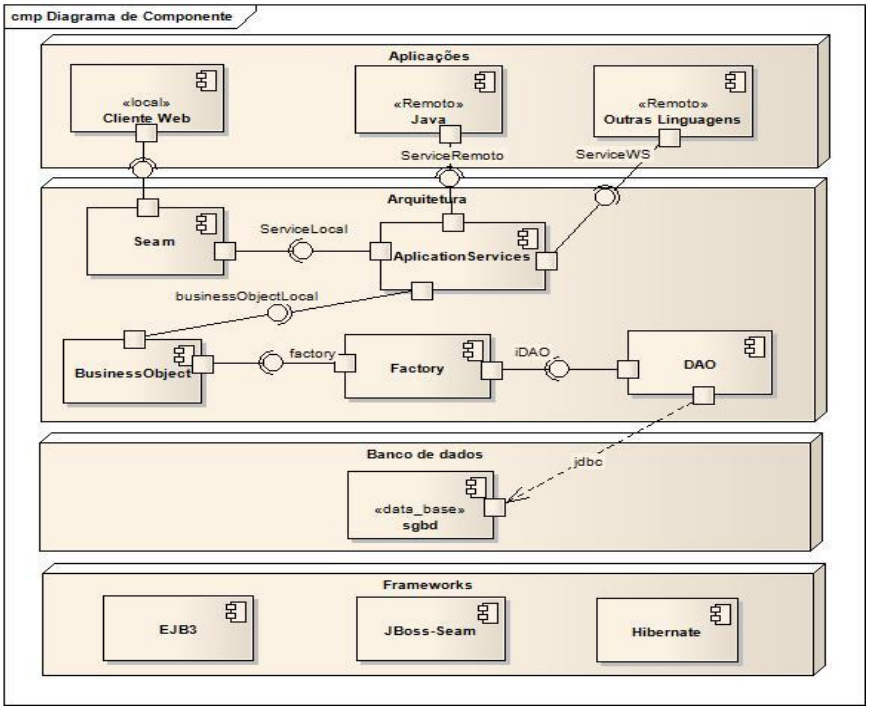

Figura 5-15 Arquitetura otimizada.

CAPÍTULO 6
CONCLUSÃO

O capítulo 6 apresenta as considerações finais do trabalho, faz uma síntese dos resultados obtidos com a implementação do estudo de caso, além de propor trabalhos futuros.

6.1 CONCLUSÕES

Em termos gerais, esta dissertação apresentou uma arquitetura que desenvolve aplicações distribuídas em Java de maneira eficiente. Também contemplou requisitos de qualidade importantes, que de acordo com (Sordi *et al*., 2006) fazem parte de uma boa arquitetura, como: desempenho, reusabilidade, manutenibilidade, entre outros. Resultados quantitativos e qualitativos confirmam o atendimento dos requisitos não-funcionais pela arquitetura proposta

No contexto do desenvolvimento de sistemas distribuídos a arquitetura apresentada tem como diferencial integrar *frameworks* consagrados na plataforma Java, como EJB e Hibernate no desenvolvimento de aplicações distribuídas. A aplicação da arquitetura trouxe os seguintes benefícios:

a) Para a empresa:

- unificação, padronização e reuso de soluções implementadas por diferentes equipes facilitando a manutenção e evolução de sistemas;
- maior rapidez no desenvolvimento de novos sistemas;
- capacitação dos técnicos envolvidos a partir do conhecimento adquirido com os padrões de mercado, o que também elevou a auto-estima da equipe.

b) Para o Estado:

- desenvolvimento ágil de soluções com menor custo, permitindo dispensar mais recursos para projetos da área fim de cada órgão;
- maior integração das soluções adotadas pelos mais diversos órgãos do Estado, o que facilita o combate às fraudes e corrupção.

6.2 TRABALHOS FUTUROS

Alguns trabalhos futuros podem ser realizados na área da pesquisa na qual este trabalho foi centrado, indicando algumas possíveis melhorias na arquitetura que podem torná-la mais completa e adequada para atender um maior número de requisitos de qualidade:

- **Planejar novos padrões de projeto:** Com a codificação corriqueira de certas funções. Percebeu-se a necessidade de padronizar os padrões de

desenvolvimento comumente usados a fim de obter maior produtividade durante o desenvolvimento. Como exemplo de padrões sugeridos, cita-se, função “listar objetos em um combo”, que para obter uma lista de objetos, acessar as camadas *Facade*, serviço e *business* é irrelevante e podem ser desconsideradas para a implementação desta função, esta simplificação resultaria em produtividade, outro padrão pode ser concebido também, quando um cliente local acessar os serviços, neste caso ele dispensaria a fachada e acessaria direto o serviço ganhando performance na execução de suas transações;

- **Aplicar um método de avaliação de arquitetura de *software***: O processo de desenvolvimento e avaliação da arquitetura obedeceu ao processo institucionalizado na empresa, resultados mais acurados podem ser obtidos por meio da aplicação de um método de avaliação próprio para arquiteturas de *software*, como ATAM, SAAM, entre outros;
- **Coletar métricas especificas para avaliação de melhoria da arquitetura:** Como os atributos de qualidade não são considerados operacionais, a avaliação é geralmente realizada qualitativamente, porém existem algumas métricas definidas para avaliar atributo de qualidade específicos como: métrica de utilização de serviços, que podem ser usadas para avaliar a melhoria da arquitetura e métricas de Rahman (2004) para avaliação estrutural da arquitetura, baseada na medição de componentes;
- **Transformar em *framework* o módulo de controle de acesso**: O *framework* pode ser implementado a fim de utilizá-lo de forma não intrusiva, isto é, não requerer que classes da aplicação (modelo de domínio) fiquem "penduradas" nas *interface*s/classes do *framework* (POJO), esta transformação torna o *framework* leve e facilita a reutilização destes módulos.

REFERÊNCIAS

ACME, The Acm, **Architectural Description Language**. School of Computer Science, Carnegie-Mellon University, Disponível em: <http://www.cs.emu.edu/~acme/>. Acesso em 01 março de 2011.

ALEXANDER Christopher; ISHIKAWA, Sara; SILVERSTEIN, Murray. **A Pattern Language**, Oxford Press, Oxford, R. Unido, 1977.

ALLEN, Robert; GARLAN, David. **Formalizing Architectural Connection**. In Proc 16th International Conference on *Software* Engineering, 1994.

ALMEIDA, Rodrigo Rebouças. **Uma Arquitetura de *Software* Para Arranjos Produtivos Locais**. 2005, Dissertação - Centro de Ciências e Tecnologia, Universidade Federal de Campina Grande, Campina Grande.

ALUR, Deepak; MALKS, Dan; CRUPI, John. **Core J2EE Patterns: Best Practices and Design Strategies**, Second Edition. Pearson, 2003.

BAHSOON, Rami; EMMERICH, Wolfgang. **Evaluating *Software* Architectures: Development, Stability, and Evolution**, Proceedings of ACS/IEEE Int. Conf. on Computer Systems and Applications. July, 2003. Tunis, Tunisia

BARBACCI, Mario. **Architecture analysis Techniques and When to Use Them**. Relatorio Técnico CMU/SEI-2002-TN-005, *Software* Engineering Institute (SEI), Pittsburgh, USA, 2002.

BASS, Len; CLEMENTS, Paul; KAZMAN, Rick. ***Software* Architecture in Practice**, 1 ed., Addison-Wesley, 2004.

BASS, Len et al. **Product Line Practice** Workshop Report. Tecnhical Report. Software Engineering Institute, Carnigie Mellon University, 1997.

BAUER, Christian; KING Gavin. **Java Persitence with Hibernate**. Manning, 2006.

BOOCH, Grady; RUMBAUGH, James; JACOBSON, Ivar. **The Unified Modeling Language User Guide**. Addison-Wesley, 1999.

BORYSOWICH, Craig. **An example of Using Design Patterns**. Disponível em: <http://it.toolbox.com/blogs/enterprise-solutions/an-example-of-using-design-patterns-21172>. Acesso em 3 de feveveiro de 2011.

BURKE, Bill; MOSON-HAEFEL, Richard. **Enterprise javaBeans 3.0**. 5th. Peason.2007.

BUSHMANN, Frank et al. **Pattern-Oriented Software Architecture – A system of patterns**, volume 1; Wiley; 2001.

BUSCHMANN, Frank; HENNEY, Kevlin; SCHIMIDT, Douglas. **Pattern-Oriented *Software* Architecture – A pattern language for Distributed Computing**. Volume 4. Wiley 2007.

BUSHMANN, Frank; HENNEY, Kevlin. **A Distributed Computing Pattern Language - Introduction To The Language Component Partitioning Patterns Resource Management Patterns**. Proceedings of Seventh European Pattern Languages of Programming Conference – Euro PLoP 2002.

CALDAS, Vanessa; FAVERO, Eloi. **Uma Proposta de Avaliação Automática de Mapas Conceituais para Ambientes de Ensino a Distância**, XXXV Conferencia Latinoamericana de Informática, Pelotas, 2009.

CARVALHO, Arthur Gonçalves. Uma **Proposta de Arquitetura de *Software* Genérica para Mobile Games Utilizando J2ME**. Disponível em: http://www.unibratec.com.br/jornadacientifica/diretorio/DEFININDO.pdf, Acesso em 21 de junho de 2011.

CELEPAR, Companhia de Informática do Paraná – CELEPAR, **Framework Pinhão**. Disponível em: <http://www.frameworkpinhao.pr.gov.br/modules/conteudo/conteudo.php?conteudo=5#pfc_CelRUP_arquitetura.pdf>. Acesso em 18 de abril de 2011.

CHUNG, Lawrence et al. **Non-Functional Requirements in Software Engineering**, 1 ed., Kluwer Academic Publishers, 1999.

CLEMENTS, Paul et al. **Documenting software architecture : views and beyond.** The SEI series in software engineering. Addison-Wesley, Boston, 2002.

CONSOLINE, Gisele. **Uma Avaliação da influência da Arquitetura no Desempenho de Sistemas de *Software***. 2006, Dissertação de Mestrado - Instituto de computação. Universidade de Campinas, Campinas.

COULOURIS, George; DOLLIMORE, Jean; KINDBERG, Tim. **Distributed Systems - Concepts and Design**. 4th edition. Addison-Wesley, 2005

FERNANDES, Aguinaldo; TEXEIRA, Descartes. **Fábrica de Software: Implantação e gestão de operações**, Atlas, São Paulo, 2004.

FLEURY, Marc; STARK, Scott; NORMAN, Richards; **JBoss® 4.0 The Official Guide**, JBoss Inc, 2005.

FOWLER, Martin et al. **Patterns of Enterprise Application Architecture.** Addison Wesley. Nov, 2002.

GAMMA, Eric et al. **Design Patterns: Elements of Reusable Object-Oriented Software.** Addison-Wesley; 2003.

GARLAN, David; PERRY, Dewayne. **Introduction to the Special Issue on *Software* Architecture**, IEEE Transactions on *Software* Engineering (April), 1995.

HARRISON, Neil. **Patterns of Architecture Reviews**, EuroPLoP ,Disponível em <http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.3.5618&rep=rep1&type=pdf>. Acesso em 12 de abril de 2011

HIRA, Celio. **ARQUIMEDIA: uma proposta de arquitetura de *software* para terminais de acesso à TV digital interativa**. 2008, Dissertação de Mestrado - Centro de Informática, Universidade Federal de Pernambuco, Recife.

HOPHE, Gregor; WOOLF, Bobby. **Enterprise Integration Patterns**; Addison-Wesley; 2003.

INTERNATIONAL ORGANIZATION FOR STANDARDIZATION, **ISO/IEC 9126, Tecnologia de Informação – Engenharia de Software – Qualidade de Produto**, 2003.

JACOBSON, Ivar GRISS, Martin; JONSSON, Patrik**. *Software* Reuse – Architecture Process and Organization for Business Success**, New York: Ed. Addison-Wesley, 1997.

JAZAYERI, Mehdi et al. ***Software* Architecture for Product Families – Principles and Practice**. Addison-Wesley, 2000.

HANSEN, Keid. **Load Testing your Applications with Apache JMeter**.Tutorial, disponível em <http://javaboutique.internet.com/tutorials/JMeter/>. Acessado em 13 de fevereiro de 2011.

KANDT, Ronald. ***Software* Enginnering Quality Pratices**. Auerbach Publications, Taylor & Francis Group, 2006.

KAZMAN, Rick et al. **The architecture tradeoff analysis method.** In Proc. of 4th IEEE Inter. Conf. on Engineering, 1998.

KAZMAN, Rick. **SAAM: A Method for Analyzing the Properties of Software** Architectures, Proceedings of the 16th International Conference on Software Engineering, Sorrento, Itália,1994.

KRAFZIG, Dirk; KARL, Banke; SLAMA, Dirk. **Enterprise SOA: Service-Oriented Architecture Best Practices**. Indianapolis: Prentice Hall, 2004.

LALANDA, Philippe. **Shared Repository Pattern**. In: Conference on Patterns Languages of Programs, Monticello, Illinois, EUA, 1998

LARMAN, Craig. **Utilizando UML e Padrões: uma introdução à análise e ao projeto orientado a objetos**. 3ª edição, Porto Alegre: Bookman, 2007.

LAZILHA, Fabricio. **Uma proposta de arquitetura de linha de produtos para sistemas de gerenciamento de workflow**, 2002. Dissertação - Programa de Pós-graduação em computação, Universidade Federal do Rio Grande do Sul, , Porto Alegre. Disponível em: <http://www.lume.ufrgs.br/handle/10183/2626>. Acesso em 03 de janeiro de 2011.

LINO, Adriano et al. **Avaliação automática de consultas SQL em ambiente virtual de ensino-aprendizagem.** 2ª Conferência Ibérica de Sistemas e Tecnologias de Informação, 2007.

METTALA, Eick; GRAHAM, Marc. **The Domain-Specific *Software* Architecture**, SEI Technical Report CMU/SEI-92-SR-009, Carnegie Mellon University, Estados Unidos, 1992.

MICROSOFT, **DATA TRANSFER OBJECT Microsoft Patterns & Practices**. 2009. Disponível em: <http://msdn.microsoft.com/en-us/library/ms978717.aspx>. Acesso em 17 de março de 2011.

O'BRIEN, Liam; SMITH, Deennis. **MAP and OAR Methods: Techniques for Developing Core Assets for *Software* Product Lines from Existing Assets**, Technical Note, CMU/SEI-2002-TN-007, Estados Unidos, 2002.

OMG, **Unified Modeling Language.** Disponível em: <http://www.uml.org/>. Acesso em 29 de maio de 2011.

PERRY, Dewayne; WOLF, Alexander. **Foundations for the study of *software* architecture**, SIGSOFT Softw. Eng. Notes, v. 17, n. 4 (1992), pp. 40-52, 1992.

PRESSMAN, Roger. **Engenharia de *Software***. 6th ed. McGraw Hill. 2006.

PRODEPA, Processamento de dados do Pará, Diretoria de desenvolvimento de sistemas. **Catálogo de Padrões Arquiteturais**. Belém, 2010a.

PRODEPA, Processamento de Dados do Estado do Pará. **Arquitetura de Referência para Desenvolvimento de *Software* na PRODEPA**. Belém, 2010b.

QUEIROZ, Ana Emilia; BRAGA, Mauricio; GOMES, Alex. **Design de ajudas inteligentes em *interface*s educativas**. In: Simpósio Brasileiro sobre Fatores Humanos e Computacionais, 2008, Porto Alegre. ACM International Conference Proceeding Series; Vol. 189, 2008.

RAPIDE, Computer Science Lab**, DRAFT Guide to Rapide 1.0 – Laguagem Reference Manuals**, Rapide Design Team – Stanford University, 1997.

RIBEIRO, Andre Luiz**. Processo de Avaliação da Qualidade de Arquitetura de *Software***. 2005. Dissertação - Centro de Informática, Universidade Federal de Pernambuco, Recife.

SCHIMIDT, Douglas et al. **Pattern-Oriented Software Architecture – Pattern for Concurrent and Networked Objects**, volume 2; Wiley, 2000.

SERPRO, SERVICO DE PROCESSAMENTO FEDERAL, **Demoiselle Framework**, Disponível em: http://www.*framework*demoiselle.gov.br/menu/*framework*/download-1/das, Acesso em 30 de março de 2009.

SHAW, Mary; GARLAN, David. ***Software* Architecture: Perspectives on an Emerging Discipline**. 1 ed. New Jersey, Prentice-Hall, 1996.

SHAW, Mary et al. **Abstractions for *Software* Architectures and Tools to Suport Them**, IEEE Transactions on *Software* Engineering, vol. 21, no. 4, 1995.

SOMMERVILLE, Ian. **Engenharia de *software***. 6 ed. Rio de Janeiro: Addison Wesley, 592p. 2006

SORDI, José; MARINHO, Bernadete; NAGY, Marcio. **Benefícios da Arquitetura de *Software* Orientada a Serviços para as Empresas: Análise da Experiência do ABN AMRO Brasil**. Revista de Gestão da Tecnologia e Sistemas de Informação, 2006.

STA, Stanford Research Institute. **A Strucutural Architecture Description Laguage**. Disponível em: <http://www.sdl.sri.com/dsa/sadl-main.htm>. Acesso: 11 jun 2010

STOERMER, Christoph; O´BRIEN, Liam. **MAP – Mining Architectures for product line evaluations**, In Proceeding of working IEEE/IFIP Conferenceon *software* Architecture, Washington, DC, USA. IEEE Computer Society, 2001.

SUN, **Core J2EE Pattern Catalog – Data Access Object**. Site Oficial da Sun, Disponível em: http://java.sun.com/blueprints/corej2eepatterns/Patterns/DataAccessObject.html>. Acesso em 14 de janeiro de 2011.

SZYPERSKI, Clemens. **Component *Software* – Beyond Object-Oriented Programming**, Addison-Wesley, 1998.

THOMÉ, Adriana; FERREIRA, Maurício; CUNHA, João. **SICSDA – uma arquitetura de *software* distribuída, configurável e adaptável aplicada às várias missões de controle de satélites**. Instituto Nacional de Pesquisas Espaciais (INPE), São José dos Campos, 2004.

TIBONI, Antônio; LISBOA, Flávio; MOTA, Luciana. **Uma plataforma livre para padronização do desenvolvimento de sistemas no Governo Federal**, In: XXIX Congresso da Sociedade Brasileira de Computação, I Workshop de Computação Aplicada em Governo Eletrônico. Bento Gonçalves – RS, 2009

VAROTO, Ana Cristina. **Visões em Arquitetura de Software**. 2002, Dissertação - Instituto de Matemática e Estatística da Universidade de São Paulo, São Paulo.

VASCONCELOS, Aline Pires. **Uma Abordagem para Recuperação de Arquitetura de *Software* Visando sua Reutilização em Domínios Específicos**,2004, Exame de Qualificação, COPPE, UFRJ, Rio de Janeiro.

XAVIER, José Ricardo. **Criação e Instanciação de Arquiteturas de *Software* Específicas de-Domínio no Contexto de Uma Infra-Estrutura de Reutilização**. 2001, Dissertação - COPPE/UFRJ, Programa de Engenharia de Sistemas e Computação, Rio de Janeiro.

YUAN, Michael; HEUTE Thomas. **JBoss Seam: Simplicity and power beyond Java EE,** Pretince Hall, 2007

Apêndice A – Serviço de geração e avaliação de mapas conceituais

A Arquitetura otimizada, subsidiou o desenvolvimento de um serviço para geração e avaliação de mapas conceituais, que são implementações de *webservices* disponibilizadas para o LabSQL conforme mostra a Figura A-1. As telas aqui mostradas fazem parte do LabSQL e acessam o serviço disponibilizado pelo serviço proposto.

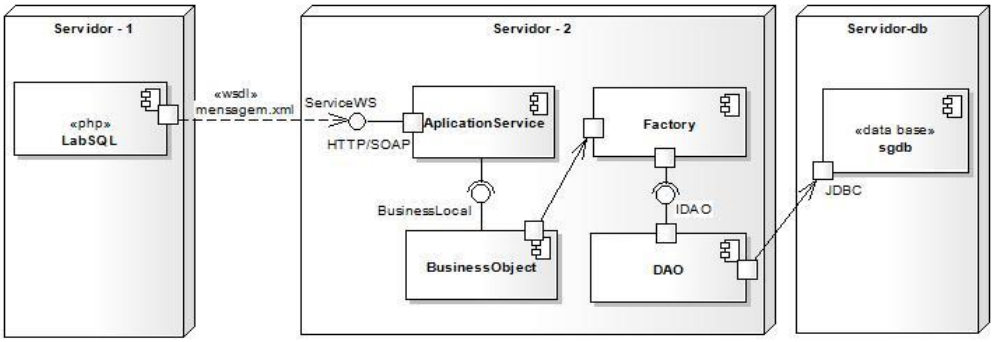

Figura A-1 Integração entre as aplicações via web service

O serviço para geração e avaliação de mapas conceituais possui como requisitos principais:

a) manter mapa conceitual: Esta funcionalidade permite que usuários com perfil de professor ou estudante gerem mapas conceituais, porem só o professor pode cadastrar seu mapa como modelo, isto é, o mapa que serve de referência na correção dos mapas dos estudantes;

b) corrigir mapa conceitual: Esta funcionalidade compara o MC do estudante com o MC base previamente cadastrado pelo professor e compara com os mapas conceituais dos demais estudantes gerando uma nota de acordo com seu número de acertos. Caso o aluno acrescente um conceito inválido o mesmo é diferenciado.

O componente desenvolvido tem como propósito principal a avaliação automática de MCs a partir da análise de similaridade dos MCs dos estudantes contra um modelo de resposta do professor, retornando um escore da avaliação quantitativa. Este resultado é obtido por meio da análise via bigrama da formação conceito-link ou link-conceito e análise via trigrama da formação dos arcos conceito-link-conceito ou link-conceito- link (Caldas & Favero, 2009).

As entidades que compõem o domínio do negocio são apresentadas na Figura A-2 a seguir:

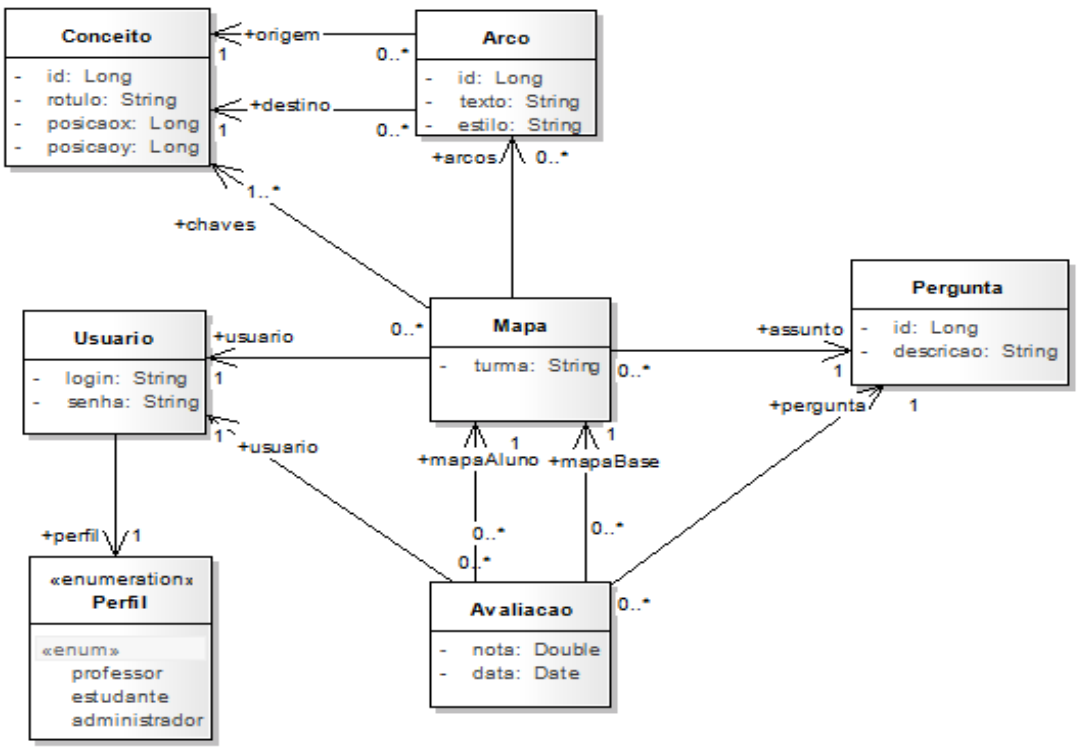

Figura A-2 Diagrama de classes – Modelo de Negócios.

Os serviços como, cadastrar mapas conceituais, avaliar mapas conceituais entre outros do módulo citado, são disponibilizados por uma *interface webservice* (ServiceWS), essa *interface* é acessada pelo LabSQL que consegue assim fornecer esses serviços a partir de sua aplicação a seus usuários finais.

A Figura A-3 ilustra o fluxo de execução para a funcionalidade cadastrar mapas, observe que cada camada possui uma *interface* (ServiceWS, *Business*MapaLocal e IDAOMapa) que define quais serviços estão disponibilizados e sua respectiva implementação (AplicationService, *Business*MapatImpl, IDAOMapaImpl) com exceção da camada factory que já disponibiliza seus métodos públicos na própria classe.

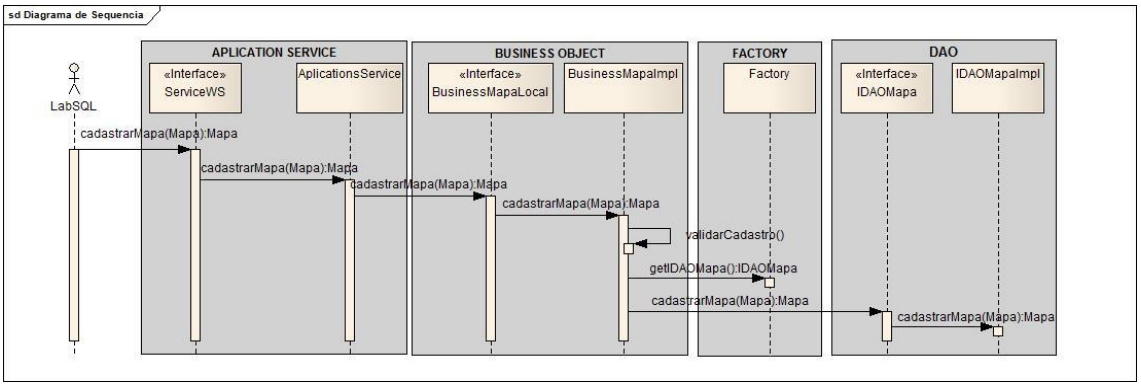

Figura A-3 Diagrama de seqüência - Método cadastrar mapa.

A Figura A-4 mostra a tela de resultado da avaliação de mapas conceituais produzida pelo estudante, como pode ser observado, foi gerada uma nota para questão 217 que define o conceito de banco de dados relacional, sendo que além de apresentar o histórico de notas para a questão o sistema também aponta o conceito que não faz parte da definição.

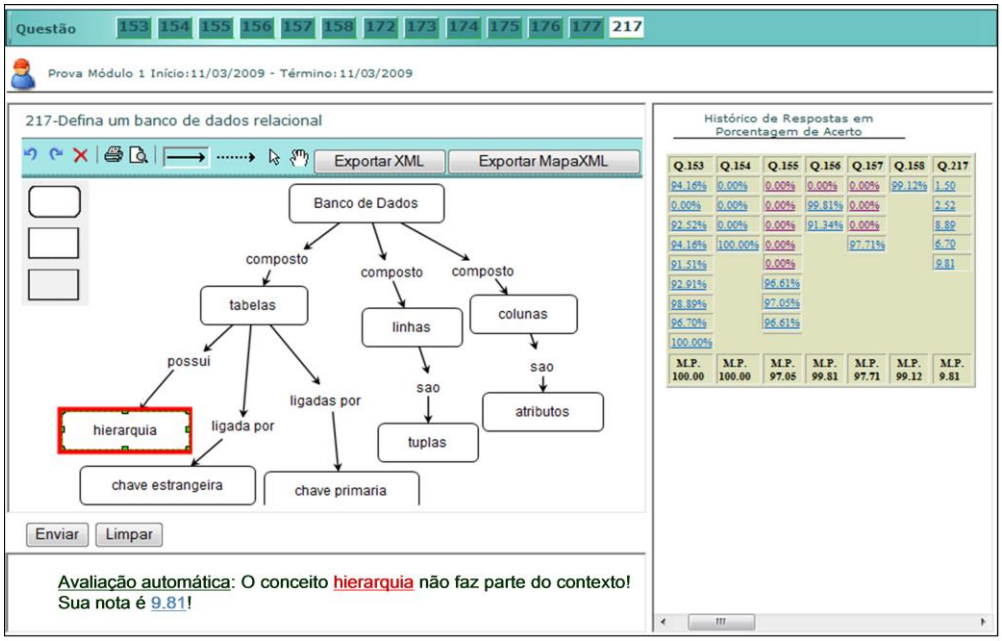

Figura A-4 Resultado da avaliação do mapa conceitual do aluno.

A versão do LabSQL que contém a funcionalidade dos Mapas Conceituais foi utilizada por várias turmas diferentes e os resultados obtidos até o momento são satisfatórios. Essa versão foi testada por 12 (doze) turmas, nos níveis de graduação e pós-graduação da Universidade do Oeste do Pará (UFOPA), durante o período de 2 (dois) semestres letivos.

Anexo A – Laudo de Avaliação do Produto

Laudo de Avaliação do Produto	
Nome do Produto:	
Avaliador:	
Data da Avaliação:	

Critérios Avaliados:

CRITERIO	Significado	Sub-caracteristica	Pergunta chave	Resultado
Funcionalidade	Evidencia o conjunto de funções que atendem ás necessidades explícitas e implícitas para a finalidade a que se destina o produto	Adequação	Propõe-se a fazer o que é apropriado?	( ) Sim ( ) Não ( ) Não Avaliado
		Acurácia	Faz o que foi proposto de forma correta?	( ) Sim ( ) Não ( ) Não Avaliado
		Interoperabilidade	Interage com os sistemas especificados?	( ) Sim ( ) Não ( ) Não Avaliado
		Conformidade	Está de acordo com as normas, leis etc?	( ) Sim ( ) Não ( ) Não Avaliado
		Segurança de acesso às informações	Evita acesso não autorizado aos dados?	( ) Sim ( ) Não ( ) Não Avaliado
Confiabilidade	Evidencia a capacidade do produto de manter seu desempenho ao longo do tempo e em condições estabelecidas.	Maturidade	É imune a falhas?	( ) Sim ( ) Não ( ) Não Avaliado
		Tolerância a falhas	Ocorrendo falhas, como ele reage?	( ) Sim ( ) Não ( ) Não Avaliado
		Recuperabilidade	É capaz de recuperar dados em caso de falha?	( ) Sim ( ) Não ( ) Não Avaliado
Usabilidade	Evidencia o relacionamento entre o nível de desempenho do produto e a quantidade de recursos utilizados, sob condições estabelecidas	Intelegibilidade	É fácil entender o conceito e a aplicação?	( ) Sim ( ) Não ( ) Não Avaliado
		Apreensibilidade	É fácil aprender a usar?	( ) Sim ( ) Não ( ) Não Avaliado
		Operacionalidade	É fácil de operar e controlar?	( ) Sim ( ) Não ( ) Não Avaliado
Eficiência/ °	Evidencia a facilidade para a utilização do produto	Tempo	O tempo de resposta na geração de consultas e relatórios, está de acordo com o esperado?	( ) Sim ( ) Não ( ) Não Avaliado
		Recursos	O tempo de resposta do sistema às suas ações é satisfatório?	( ) Sim ( ) Não ( ) Não Avaliado
Portabilidade	Evidencia a capacidade do produto de ser transferido de um ambiente para outro	Adaptabilidade	É fácil adaptar a outros ambientes?	( ) Sim ( ) Não ( ) Não Avaliado
		Capacidade para ser instalado	É fácil instalar em outros ambientes?	( ) Sim ( ) Não ( ) Não Avaliado

Anexo B - Questionário

Perfil
( ) Analista ( ) Projetista ( ) Desenvolvedor

Qual seu conhecimento sobre a arquitetura de *software* padrão utilizada na empresa?
( ) Excelente ( ) Bom ( ) Razoável ( ) Insuficiente

Sobre a arquitetura de *software*, qual a sua opinião:

Quanto a modularidade (independência funcional entre componentes)

1. A arquitetura esta projetada a fim de prover modularidade?

( ) Totalmente ( ) Atende ( ) Atende Pouco ( ) Não Atende

2. As camadas possuem uma *interface* de comunicação bem definida entre as mesmas?

( ) Totalmente ( ) Atende ( ) Atende Pouco ( ) Não Atende

Quanto a extensibilidade (capacidade de suportar a adesão de novos componentes e *framework*s)

3. A arquitetura permite adicionar camadas sem prejuízo para sua estrutura?

( ) Totalmente ( ) Atende ( ) Atende Pouco ( ) Não Atende

4. A arquitetura permite adicionar *framework*s sem prejuízo para sua estrutura?

( ) Totalmente ( ) Atende ( ) Atende Pouco ( ) Não Atende

Quanto a reusabilidade (capacidade de reutilização de módulos do sistema em outras aplicações)

5. A arquitetura pode ser reutilizada no desenvolvimento de diferentes sistemas distribuídos?

( ) Totalmente ( ) Atende ( ) Atende Pouco ( ) Não Atende

6. Os *framework*s podem ser utilizados em diferentes projetos?

( ) Totalmente ( ) Atende ( ) Atende Pouco ( ) Não Atende

Quanto a manutenibilidade (característica do componente ser modificado.)

7. A arquitetura possui um processo de manutenção padrão? Ou seja todos seguem um mesmo procedimento para fazer manutenção no *software*?

( ) Totalmente ( ) Atende ( ) Atende Pouco ( ) Não Atende

8. Com a aplicação da arquitetura de *software* é possível executar a manutenção de sistemas com rapidez?

( ) Totalmente ( ) Atende ( ) Atende Pouco ( ) Não Atende

9. Os *framework*s (muiraquitã, seam, entre outros) exigidos pela arquitetura são adequados às complexidades dos requisitos funcionais exigidos?

( ) Totalmente ( ) Atende ( ) Atende Pouco ( ) Não Atende

Em relação ao *software* desenvolvido com a arquitetura de *software*, qual sua opinião:

10. O usuário precisa ter profundo conhecimento na área para utilizar o *software*?

( ) Totalmente ( ) Atende ( ) Atende Pouco ( ) Não Atende

11. O *software* tem capacidade de continuar o processamento com grandes volumes de dados?

( ) Totalmente ( ) Atende ( ) Atende Pouco ( ) Não Atende

12. O *software* é conciso nos resultados, passando confiança ao usuário?

( ) Totalmente ( ) Atende ( ) Atende Pouco ( ) Não Atende

13. O *software* é capaz de desempenhar todas as funções necessárias para sua execução?

( ) Totalmente ( ) Atende ( ) Atende Pouco ( ) Não Atende

Quanto à portabilidade (característica de o componente ser transferido de um ambiente para outro.

14. O *software* pode ser facilmente modificado para atender as necessidades do usuário?

( ) Totalmente ( ) Atende ( ) Atende Pouco ( ) Não Atende

15. O *software* tem capacidade para operar em ambientes diferentes?

( ) Totalmente ( ) Atende ( ) Atende Pouco ( ) Não Atende

16. O *software* tem capacidade de continuar funcionando sem sofrer modificações quando da troca de ambiente?

( ) Totalmente ( ) Atende ( ) Atende Pouco ( ) Não Atende

17. O *software* permite adicionar/excluir funções com facilidade?

( ) Totalmente ( ) Atende ( ) Atende Pouco ( ) Não Atende

Quanto à eficiência (Comportamento em relação ao tempo)

18. O tempo de resposta do *software* é adequado em relação ao volume de dados envolvido?

( ) Totalmente ( ) Atende ( ) Atende Pouco ( ) Não Atende

19. O tempo de resposta é adequado à complexidade das funções do *software*?

( ) Totalmente ( ) Atende ( ) Atende Pouco ( ) Não Atende

20. O tempo de respostas para realização de consultas é satisfatório?

( ) Totalmente ( ) Atende ( ) Atende Pouco ( ) Não Atende

21. Ao aplicar a arquitetura de *software* na programação de sistemas, o tempo total para programação:

( ) Aumentou ( )Diminuiu ( ) Não fez diferença

Que ajustes você faria na arquitetura para melhorar a qualidade do produto final desenvolvido?