Universidade Federal do Rio de Janeiro

Programa de Pós Graduação em Informática

(Instituto de Matemática / Núcleo de Computação Eletrônica)

Geração Automática de Interfaces a partir de Descritores Enriquecidos

Dissertação de Mestrado

Aluna:
Marcelle Mussalli Cordeiro

Orientadores:
Prof. Dr. Carlo Emmanoel Tolla de Oliveira
Profª. Drª. Vanessa Braganholo Murta

Rio de Janeiro, RJ – Brasil
2009

GERAÇÃO AUTOMÁTICA DE INTERFACES A PARTIR DE DESCRITORES ENRIQUECIDOS

MARCELLE MUSSALLI CORDEIRO

PPGI-IM/UFRJ

Mestrado em Informática

Orientadores:

Prof. Dr. Carlo Emmanoel Tolla de Oliveira, Ph.D.

Profª. Drª. Vanessa Braganholo Murta, Ph.D.

Rio de Janeiro, RJ – Brasil

2009

GERAÇÃO AUTOMÁTICA DE INTERFACES A PARTIR DE DESCRITORES ENRIQUECIDOS

Marcelle Mussalli Cordeiro

Dissertação submetida ao corpo docente do Instituto de Matemática e do Núcleo de Computação Eletrônica da Universidade Federal do Rio de Janeiro - UFRJ, como parte dos requisitos necessários à obtenção do grau de Mestre em Informática.

Aprovada por:

Prof._____ (Orientador)
Carlo Emmanoel Tolla de Oliveira, Ph.D.

Profª _____ (Orientadora)
Vanessa Braganholo Murta, Ph.D.

Prof._____
Fernando Silva Pereira Manso, Ph.D.

Profª._____
Cláudia Maria Lima Werner, D.Sc.

Profª._____
Maria Luiza Machado Campos, Ph.D.

Prof._____
Geraldo Bonorino Xexéo, D.Sc.

Rio de Janeiro, RJ – Brasil
2009

CORDEIRO, MARCELLE MUSSALLI.

Geração Automática de Interfaces a partir de Descritores Enriquecidos / Marcelle Mussalli Cordeiro. Rio de Janeiro: UFRJ / IM / PPGI, 2009.

132 pp.

Dissertação (Mestrado em Informática) – Universidade Federal do Rio de Janeiro - UFRJ, IM / PPGI, 2009.

Orientador: Prof. Carlo Emmanoel Tolla de Oliveira e Profª. Vanessa Braganholo Murta

1. Geração automática de sistemas. 2. Tipos elementares de dados. 3. XML.

I. PPGI/IM-NCE/UFRJ. II. Título (série).

Aos meus pais.

AGRADECIMENTOS

À Deus, por abençoar a mim e a minha família todos os dias, essencial para a conclusão deste trabalho.

Aos meus queridos pais, pelo grande apoio em todas as formas possíveis, por tornarem esse aprendizado mais fácil, proporcionando um ambiente confortável, por entenderem os momentos mais difíceis e por sempre terem uma palavra reconfortante que ajudasse a seguir em frente.

Ao meu irmão, pelo carinho que muitas vezes é escondido, mas que sabemos que sempre está presente.

À minha querida madrinha, pelo grande carinho e por estar sempre presente em todos os momentos da minha vida. Pelos conselhos, que me ajudam a me posicionar melhor diante da vida.

Ao meu amado namorado e melhor amigo, Renato. Por estar ao meu lado em todo momento. Por tornar a minha vida mais serena e feliz. Pelo apoio, pela ajuda e força nos momentos mais difíceis.

À minha querida tia, que infelizmente não está mais entre nós para assistir a realização deste trabalho, mas que sei que estaria orgulhosa e muito feliz por mim.

Ao orientador, Prof. Carlo Emmanoel, que sempre acreditou em mim e me deu a oportunidade de aprender durante toda a fase de mestrado. Pelo grande amigo que sempre demonstrou ser, sempre pronto a aconselhar e me fazer sentir mais confortável diante das dificuldades da vida.

À orientadora, Profª. Vanessa Braganholo Murta, pela excelente orientação dedicada durante todo o desenvolvimento desta dissertação.

Aos professores Fernando Manso, Cláudia Werner, Maria Luiza Machado e Geraldo Xexéo por aceitarem participar da banca examinadora deste trabalho.

Aos amigos Gilson Tavares e Ricardo Storino, gerentes do projeto SIGA, pela oportunidade que me concederam de aprendizado durante o tempo de estágio. Obrigada principalmente por sempre procurarem manter um ambiente de trabalho confortável para a equipe.

Aos amigos da equipe SIGA, que trabalharam comigo e tornaram o desenvolvimento deste trabalho mais prazeroso.

RESUMO

CORDEIRO, Marcelle Mussalli. Geração Automática de Interfaces a partir de Descritores Enriquecidos. Orientadores: Prof. Carlo Emmanoel Tolla de Oliveira e Profª Vanessa Braganholo Murta. Rio de Janeiro: UFRJ/PPGI/IM/NCE, 2009. Dissertação (Mestrado em Informática).

Desenvolver software de qualidade assegurada, com elevada produtividade, dentro do prazo e custos estabelecidos tem sido um grande desafio da Engenharia de Software. A implementação da geração automática de código tende a reduzir o tempo gasto em desenvolvimento e manutenção do sistema, incorporando qualidade ao processo. Isto porque a geração automática produz um ganho considerável em produtividade devido à diminuição da duplicação de código e à construção de descrições abstratas precisas e formais. Aliado à geração automática, temos o desenvolvimento baseado em componentes (DBC) ganhando força como paradigma de desenvolvimento, visto que prega a técnica de reutilização de componentes. Esta dissertação propõe a geração automática de interfaces a partir de descritores enriquecidos. Tais descrições são enriquecidas através da criação de notações que, quando adicionadas às entidades, forneçam a informação necessária à geração automática. Dessa forma, é proposta uma especificação de modelo mais completa, baseada em Tipos Elementares de Dados, com o objetivo de adicionar significado ao dado, incorporando qualidade e minimizando os erros provocados pela duplicação de código. O Sistema de Gestão Acadêmica da UFRJ (SIGA) foi utilizado como exemplo de aplicação, e os resultados são mostrados e discutidos nesse trabalho.

ABSTRACT

CORDEIRO, Marcelle Mussalli. Automatic Generation of user interfaces from Enhanced Descriptors. Advisors: Prof. Carlo Emmanoel Tolla de Oliveira e Profª Vanessa Braganholo Murta. Rio de Janeiro: UFRJ/PPGI/IM/NCE, 2009. Thesis (Master of Computer Science).

Software development with assured quality and high productivity, respecting the deadline and costs of projects has been a great challenge for Software Engineering. The implementation of automatic code generation tends to reduce the time of software development and maintenance, incorporating quality to the process. Automatic code generation increases the productivity due to the reduction of code duplication and to the construction of accurate and formal abstract descriptions. Besides automatic code generation, the component-based development (DBC) is gaining force as a development paradigm, because it is based on components reuse. The aim of this work is to propose the automatic generation of user interfaces from enhanced descriptors. Such descriptions are enriched through notations that, when added to the entities, supply the information needed for automatic generation. Thus, this works proposes a more complete model specification, based on Elementary Types of Data, with the aim to add meaning to the data, incorporating quality and minimizing errors caused by code duplication. The Academic Management System of UFRJ (SIGA) was used as a case study. The results are showed and discussed in this work.

Lista de Figuras

Lista de Tabelas

Lista de Abreviaturas

CEG	Conselho de Ensino de Graduação
CEPEG	Conselho de Ensino para Graduados
DBC	Desenvolvimento Baseado em Componentes
DDD	Domain Driven Development
DRE	Divisão de Registro de Estudantes
EA	Engenharia de Aplicação
ED	Engenharia de Domínio
EJB	Enterprise Java Beans
FODA	Feature-Oriented Domain Analysis
HTML	Hyper Text Markup Language
HTTP	Hypertext Transfer Protocol
JSP	Java Server Pages
J2EE	Java 2 Enterprise Edition
MDA	Model Driven Architecture
MVC	Model-View-Controller
PIM	Platform Independent Model
PSM	Platform Specific Model
RSEB	Reuse-Driven Software Enginnering Business
SEI	Software Engineering Intitute
SIGA	Sistema Integrado de Gestão Acadêmica
UFRJ	Universidade Federal do Rio de Janeiro
UML	Unified Modeling Language
VO	Value Object
XML	Extensible Markup Language
XP	eXtreme Programming

SUMÁRIO

1. Introdução

Desenvolver software de qualidade assegurada, com elevada produtividade, observando os prazos e recursos estabelecidos para o projeto tem sido um grande desafio da Engenharia de Software. O surgimento de novas técnicas de produção de software aumenta a demanda de sistemas cada vez mais complexos. Com tamanho crescimento da indústria de software, percebemos muitas semelhanças entre produtos, até mesmo naqueles com propósitos diferentes.

As semelhanças encontradas levam-nos a atestar que os softwares seguem um determinado padrão operacional. Em um único produto, podemos verificar que, entre seus serviços, existem operações similares. Estas similaridades podem acarretar em duplicação de código e em inconsistências de interface visual.

Operações comuns necessitam de um padrão de implementação. Este pode ser alcançado através da geração automática de código, que produz um ganho considerável em produtividade. Isto porque se baseia na diminuição da duplicação de código e na construção de descrições abstratas precisas e formais, incorporando qualidade ao processo.

Nesta dissertação, apresentaremos o Hércules (PAIS, 2004) como arcabouço de geração automática de sistemas de informação, que provê uma forma padronizada de descrever os serviços de um sistema utilizando a linguagem XML (W3C, 2009). Este engenho melhora a qualidade das descrições dos serviços, já que impõe um formato de implementação aos desenvolvedores.

Apesar disso, as descrições são muito extensas, de difícil compreensão e não reutilizáveis, acarretando em uma prática de duplicação de código, em especial nas descrições da vista dos serviços, visto que, como o sistema obedece a um padrão operacional, são muito parecidas. Com o intuito de mitigar esses problemas, propomos neste trabalho uma extensão do Hércules através de um modelo de descrição de interface menos extenso e mais descritivo, facilitando assim a sua compreensão. Este modelo foi construído com base nos ideais de reutilização de componentes. Assim, teremos a construção de componentes de vista para o sistema, que irão compor a descrição da interface de um determinado serviço desse sistema e poderão ser utilizados na descrição de qualquer outro serviço.

Baseando-se na importância de componentes reutilizáveis para a construção de sistemas cada vez mais complexos utilizando o mínimo de recursos, surgiu a proposta de Desenvolvimento Baseado em Componentes (DBC) (HEINEMAN, COUNCILL, 2001). Essa vem ganhando adeptos e é uma das propostas mais promissoras para a melhoria da produção de software nos próximos anos (WERNER et al., 1999). A tecnologia conhecida como Engenharia de Domínio (SEI, 2007) provê suporte ao DBC. Essa é dividida em fases e abrange todo o processo de desenvolvimento, desde a especificação do domínio até a implementação de artefatos. Apontamos neste trabalho, o problema da explosão fatorial de artefatos provocada pela proposta atual da Engenharia de Domínio. Com o foco na fase de implementação de artefatos, em especial, artefatos de vista do sistema, propomos um modelo de construção desses

artefatos de forma a mitigar o problema da explosão fatorial. Assim, além da construção de artefatos reutilizáveis voltados para a vista do sistema, propomos que tal construção seja realizada de uma forma que não gere uma explosão de artefatos na variabilidade. Com isso, artefatos que possuem a mesma essência, mas com algum comportamento variável deixam de ser artefatos diferentes, ou seja, teremos um único artefato com comportamentos diferenciados, mas sem perder a sua essência.

Com o objetivo de melhorar a manutenção de sistemas, com foco na apresentação visual, a proposta deste trabalho aborda a geração automática de interfaces de sistemas de informação a partir de descritores enriquecidos da vista. Para isso, propomos uma refatoração no arcabouço Hércules no sentido de diminuir a extensão e enriquecer a descrição da vista dos serviços. Tais descrições são enriquecidas através da criação de notações que, quando adicionadas às entidades, forneçam a informação necessária à geração automática. Dessa forma, é proposta uma especificação de modelo da vista mais completa, o que chamamos de descritor enriquecido. O modelo da vista é o que compreende as informações que serão apresentadas ao usuário. Através do enriquecimento de informação nos descritores visuais, almejamos um código mais inteligível, reduzindo assim a duplicação de código.

Para o desenvolvimento do descritor enriquecido, apresentado neste trabalho, tivemos como base a terceira fase da Engenharia de Domínio, ou seja, a implementação de artefatos que podem ser reutilizados por todo o sistema, atendendo ao maior número de serviços possível, e também por outros sistemas.

Dessa forma, construímos artefatos que são responsáveis por descreverem a apresentação visual de uma determinada informação no sistema, porém utilizamos uma solução arquitetural capaz de mitigar o problema da explosão fatorial, aumentando o reúso. Os artefatos são construídos de acordo com os requisitos visuais do sistema. É feito um levantamento dos tipos de visualização dos dados do sistema e para cada tipo é criado um artefato capaz de descrevê-lo. Esta solução é baseada no conceito de Tipos Elementares de Dados (LEITE, 2006), que são tipos complexos, ou seja, derivados de tipos primitivos ou de outros Tipos Elementares já construídos.

O Tipo Elementar representa um conjunto de valores e operações reconhecidas como válidas sobre elementos deste conjunto. Em outras palavras, os Tipos Elementares são tipos de dados definidos através de metadados que vão prover mais significado ao dado, ajudando a garantir a sua confiabilidade. Estes Tipos Elementares formarão uma biblioteca de tipos reusáveis. Utilizando tipos elementares de dados, pretende-se melhorar a captura da apresentação do negócio dentro do sistema e, com isso, diminuir a duplicação de código, além de reduzir a margem de erros devido a inconsistências.

Este trabalho está organizado em 8 capítulos. O primeiro deles corresponde a esta introdução, que exibe a motivação e o objetivo deste trabalho, além de sua organização. O Capítulo 2 apresenta estudos realizados na área de desenvolvimento de sistemas de informação que serviram de base para o desenvolvimento deste trabalho. O Capítulo 3 apresenta a pesquisa realizada sobre trabalhos relacionados com os temas levantados. O Capítulo 4 apresenta,

em detalhes, a proposta deste trabalho, um mecanismo para agilizar o desenvolvimento de sistemas de informação garantindo segurança e qualidade. Além disso, mostra como o modelo proposto atende às necessidades apontadas no Capítulo 2. O Capítulo 5 apresenta, em detalhes, a implementação e os ajustes necessários nos arcabouços que serviram como base para a implementação da proposta. O Capítulo 6 apresenta o exemplo de aplicação utilizado neste trabalho. Para isso foi dividido em duas seções principais: a primeira apresenta o SIGA - Sistema de Gestão Acadêmica da UFRJ; e a segunda apresenta a aplicação do mecanismo proposto nos casos de uso do SIGA. O capítulo 7 apresenta uma avaliação do mecanismo proposto, levando em conta sua aplicação no exemplo de aplicação apresentado no sétimo capítulo. Por fim, apresentamos no capítulo 8 a conclusão com uma visão geral deste trabalho, assim como suas contribuições, dificuldades levantadas e suas perspectivas futuras.

2. Referencial Teórico

A duplicação de trechos de código fonte representa não só um risco à integridade do software, mas também um fator de queda de produtividade. A adoção de uma solução arquitetural capaz de converter a duplicação em uma forma de reúso de software é uma maneira de se mitigar este problema. O reúso de software é um tema da engenharia de software que tem recebido destaque especial nas últimas décadas (REINEHR, 2008).

A técnica de reúso possibilita o reaproveitamento de grande parte do software produzido e informações associadas em projetos futuros, diminuindo o custo e aumentando a produtividade no desenvolvimento e na própria evolução do produto. A idéia básica é que componentes de software sejam projetados e implementados de forma que possam ser reusados em muitos sistemas diferentes (BRAGA, 2005).

Com a crescente demanda por aplicações cada vez mais complexas, surgem técnicas de desenvolvimento que possuem o objetivo de vencer tais barreiras de complexidade. O desenvolvimento baseado em componentes (DBC) (HEINEMAN, COUNCILL, 2001) surge como uma proposta para que, através do trabalho em equipe, exista uma melhor gerência do desenvolvimento, controle e garantia da qualidade, possibilitando reutilização de componentes e redução do tempo necessário para o desenvolvimento de *software*. Segundo Staa (2000), com o objetivo de assegurar a qualidade de novos programas e reduzir o custo de

desenvolvimento, é recomendável o uso de componentes já comprovados e reutilizáveis.

Apresentaremos neste capítulo as linhas de pesquisa que mais têm se destacado sobre reutilização de artefatos, como a Engenharia de Domínio e o Desenvolvimento Baseado em Componentes. Apontaremos uma fragilidade de tais propostas a fim de justificar a escolha da arquitetura empregada no modelo proposto. Esse modelo foi arquitetado de forma a mitigar o problema gerado pela utilização da engenharia de domínio na construção de sistemas de informação. As duas primeiras seções deste capítulo apresentam o quão importante é para um sistema que o seu domínio seja bem compreendido para sua evolução e manutenção, além de esclarecer que o mesmo domínio pode ser visto de maneiras diferentes de acordo com o usuário que o requisita, impondo dessa forma um determinado grau de variabilidade ao sistema.

2.1 Domínio do Problema

O domínio é um dos elementos fundamentais em um sistema de software. O domínio de uma aplicação consiste na captura das entidades e conceitos do mundo real que pertencem ao espaço do problema. Essas entidades e conceitos são representados como classes e seus vários tipos de relacionamentos. Dessa forma, o domínio está difundido em todas as partes do sistema.

Entre as definições do termo domínio está a palavra *conhecimento*. Em ciência da computação, o domínio do problema é o conhecimento do universo de informações e regras de um problema que se pretende solucionar com um sistema

de software (CORRÊA, 2005). O domínio do problema é o que compreende a essência da informação, e, portanto, deve ser bem representado e bem compreendido.

A definição do domínio do problema começa a partir do levantamento de requisitos junto ao cliente. Devido a dificuldades decorrentes de problemas de comunicação e até mesmo de entendimento do próprio cliente com relação ao que deseja em seu sistema, nem todo o conhecimento é obtido em primeira instância, abrindo a possibilidade de que mais regras e informações sejam adicionadas posteriormente. É imprescindível que o modelo da informação, responsável por representar uma parte do domínio, seja capaz de absorver alterações no entendimento do domínio do problema.

Um domínio bem conhecido é fundamental para o bom funcionamento do sistema, ou seja, é preciso conhecer aquilo que se pretende resolver. A deficiência na compreensão do problema gera futuras alterações no domínio do sistema. Com uma boa representação do domínio do problema, eventuais alterações se tornam mais simples de serem realizadas (CORRÊA, 2005).

Mesmo quando o domínio do problema extraído de um caso de uso parece claro e definido, esse pode ainda sofrer alterações. A forma com que o domínio será visualizado dependerá daquilo que cada usuário do sistema quer ver e de como quer ver. O modelo da informação surge da relação entre o observador (funcionalidade) e o observado (domínio).

2.2 Conhecimento Empírico

Muitos filósofos acreditavam que o conhecimento era obtido apenas através da razão. Outros começaram a considerar a construção do conhecimento através de experiências. O conhecimento empírico é construído através de observação, experiência. É a percepção do mundo através dos sentidos (BERGMAN, 2004). Deste modo, não há como garantir a coincidência da percepção de um mesmo fato por várias pessoas, pois cada uma a inferiu subjetivamente sob o arcabouço de sua psique individual e única. Mesmo quando os sentidos limitam a obtenção de um conhecimento, há casos em que este conhecimento obtido é suficiente para o emprego que se pretende dar.

A obtenção da informação por meios limitados pode ser comparada com a formação do domínio nas diferentes partes de um sistema de software. Os recursos e a finalidade de cada camada do sistema representam os sentidos que contribuirão para a construção do conhecimento. O domínio utilizado em cada camada poderia ser considerado como conhecimento obtido através desses sentidos. Cada usuário do sistema percebe a informação de maneira diferente. O resultado é uma visão do domínio mais apropriada a cada uma delas.

Para abordar este assunto, iremos apresentar o poema de autoria de John Godfrey Saxe, chamado *Os cegos e o elefante* (SAXE, 2008). Baseado em uma história tradicional da Índia, o poeta conta a aventura de seis homens que, apesar de cegos, foram ver um elefante. Cada um, por observação, deveria satisfazer sua curiosidade. A Figura 2.1 ilustra o tema do poema.

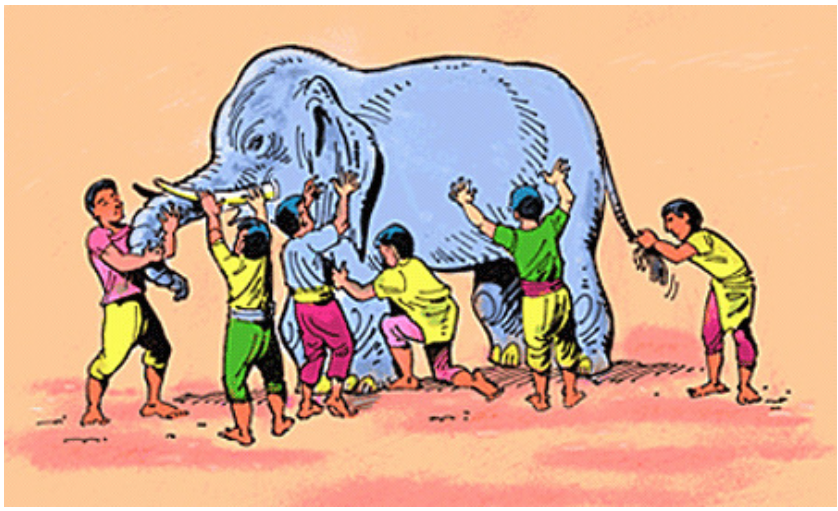

Figura 2.1: Os cegos e o elefante (SAXE, 2008)

A Figura 2.2 ilustra o desenho do elefante feito a partir das descrições de cada homem poema e logo a seguir apresentamos o texto. Os cegos, comunicando suas impressões, chegaram a compor a imagem de um elefante. Porém, nenhum dos pontos de vista individuais dava conta de uma descrição verdadeira de um elefante.

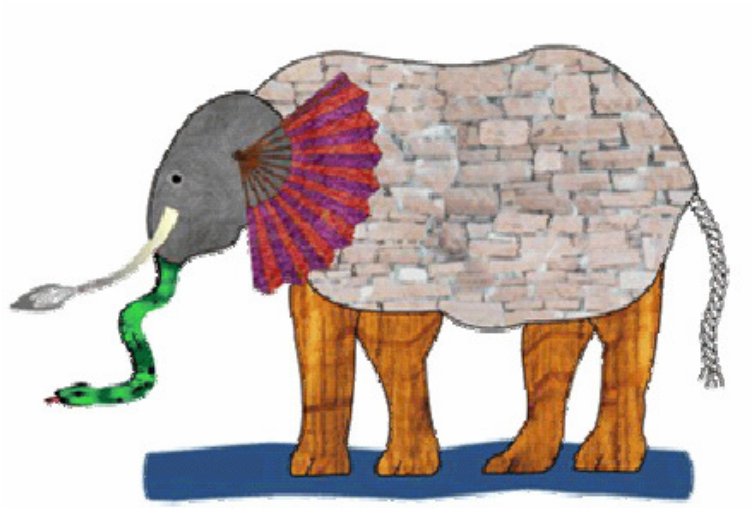

Figura 2.2:O elefante dos seis homens (SAXE, 2008)

OS CEGOS E O ELEFANTE

Eram seis homens do Industão
Que gostavam muito de aprender
E que foram um elefante ver
(Apesar de serem todos cegos)
Cada um fez sua observação
Na tentativa de se satisfazer

O primeiro se aproximou do elefante
E quase a cair
Contra o lado firme do gigante
Logo começou a bramir
"Deus me ajude, mas o Elefante
é muito parecido com uma parede!"

O segundo, sentindo a presa,
Gritou, "Oh, o que temos aqui,
Tão redondo, liso e afiado?
Para mim está claro
Que a maravilha do Elefante
Se parece com uma lança!"

O terceiro se aproximou do animal,
E aconteceu de tocar
Na tromba com as suas mãos,
E assim sem medo falou
"Agora eu vejo", ele disse, "o Elefante
Se parece com uma serpente!"

O quarto alcançou uma pata
E a sentiu próximo do joelho,
"Que maravilhosa besta é esta
É bem direta" ele disse;
"Está claro que o Elefante
Se parece muito com uma árvore!"

O quinto, que tocou a orelha
Disse "Até o mais cego
Pode dizer que ele se parece mais,
Negue quem puder,
O maravilhoso Elefante
e muito parecido com um abanador!"

O sexto logo começou
A tentar agarrar a fera,
quando o rabo se movimentou
E o sentimento que ele sentiu:
"Eu vejo", ele disse, "o Elefante
Se parece com um corda!"

E estes homens do Industão,
Discutiram alto e por muito tempo,
Cada um a sua própria opinião
Mantendo uma firme posição,
E apesar de cada um estar certo em parte
Todos estavam errados!

Moral:
Muitas vezes em guerras teológicas,
Os que as disputam, penso eu,
Criticam a ignorância
que dos outros ouviu
e afirmou sobre um elefante
Que nenhum deles jamais viu!

Podemos fazer uma analogia para sistemas de software com relação à dificuldade de se descrever um novo software. Devemos lembrar que, na maioria das vezes, um sistema é projetado para atender um conjunto de usuários com perfis e necessidades diferentes. Assim, cada subconjunto de usuários enxerga o novo software de acordo com suas próprias necessidades, ignorando as demais

funcionalidades. O sistema será então composto da combinação das diferentes visões.

No poema, percebemos que o que realmente importa é que cada homem ficou satisfeito com a sua interpretação. Com software, funciona da mesma forma, o domínio pode ser apresentado de maneiras diferentes, dependendo da funcionalidade ou do usuário que requisitou a informação. O importante é que cada usuário do sistema fique satisfeito com a sua interpretação do domínio.

2.3 Padrões de Projeto

O desenvolvimento de software orientado a objetos possui grande motivação e ênfase no reúso de software. Nessa perspectiva, a percepção de que os sistemas de informação são estruturalmente parecidos leva a uma área da engenharia de software denominada de Padrões de Software (LARMAN 2004, HELM 2000, BUSCHMANN 1996). Tais padrões têm sido pesquisados na última década como uma forma promissora de reúso, não somente de código, mas também de projeto, análise, arquitetura e processo de desenvolvimento.

O uso de padrões de software traz inúmeras vantagens na modelagem e implementação de um software. Dentre estas, a possibilidade de projetar soluções mais rapidamente e com qualidade, já que os padrões são soluções comprovadamente eficientes para problemas já conhecidos. Além disso, visa principalmente flexibilidade, organização e reaproveitamento de código, o que resulta em maior produtividade, qualidade e facilidade de manutenção das aplicações assim desenvolvidas.

2.4 Desenvolvimento Baseado em Componentes

Cada vez mais os setores da economia mundial dependem do uso de aplicações. O crescimento do mercado de software tem exigido a construção de aplicações complexas em espaços reduzidos de tempo e com qualidade assegurada. Dependendo da aplicação, um mal funcionamento é inadmissível, podendo causar danos de grandes proporções, financeiras e, em alguns casos, vitais, colocando a vida de pessoas em risco. Dessa forma, o grande desafio da indústria de software tem sido alcançar a capacidade de desenvolver seus produtos de forma rápida, segura e com qualidade.

Apoiado nas técnicas de reúso e padrões de software, com a finalidade de reduzir a complexidade do desenvolvimento, assim como custos, de forma que a qualidade seja assegurada, o desenvolvimento baseado em componentes (DBC) tem sido considerado como uma boa técnica de construção de software. Isto é baseado no fato de o desenvolvimento ser guiado por componentes já existentes, devidamente testados e com qualidade verificada. Dessa forma, é gasto menor tempo no desenvolvimento de fato, tempo este melhor aproveitado no entendimento real dos requisitos do sistema, por exemplo.

O desenvolvimento baseado em componentes (DBC) está cada vez mais ganhando adeptos e é uma das propostas promissoras para a melhoria da produção de software nos próximos anos. Segundo Werner et al. (2005), o DBC tem como objetivo a definição de componentes interoperáveis, com interfaces bem definidas, evidenciando os tipos de relacionamentos permitidos por estes

componentes. Dessa forma, a complexidade no desenvolvimento é reduzida, assim como os custos, através da reutilização de componentes exaustivamente testados.

Existem inúmeras definições para componentes. Uma definição que demonstra bem a idéia apresentada neste trabalho é que “componente é um elemento que pode ser utilizado por diversos programas e possui identidade própria” (STAA, 2000). Dessa forma, veremos que componentes se comportam como serviços que são disponibilizados por meio de uma interface bem definida. Assim, podem ser reutilizados por diferentes programas.

2.5 A Variabilidade

Nos tempos atuais, é notável o crescimento da indústria de software, impulsionado pela também crescente demanda do mercado. A principal característica exigida para as aplicações hoje desenvolvidas é que sejam flexíveis. Como o mercado sofre modificações o tempo todo, é importante que os softwares utilizados acompanhem tamanha evolução, por isso flexibilidade é uma qualidade necessária em sistemas.

Uma das formas mais utilizadas para alcançar a flexibilidade em sistemas de software cada vez mais complexos é a implementação de variabilidade. Existem diversas técnicas de implementação que fornecem diferentes graus de variabilidade ao software. Neste ponto, é importante salientar que “as técnicas que fornecem níveis mais elevados de variabilidade também implicam processos mais complexos de engenharia” (BRAGANÇA, MACHADO, 2004).

Com o objetivo de atingir níveis mais elevados de flexibilidade, a indústria de software tem adotado técnicas de implementação de variabilidade suportadas em tempo de execução das aplicações. Exemplos de padrões da indústria para componentes de software que suportam este tipo de implementação de variabilidade são COM, XPCOM, Java 2, CORBA e .Net, permitindo que diversas implementações possam ser utilizadas para uma determinada interface funcional. Isso significa que componentes de software que são desenvolvidos para uma determinada interface funcional podem ser instalados e dinamicamente utilizados nos pontos de variabilidade da aplicação.

As plataformas de software como Java 2 e .Net já suportam técnicas mais rebuscadas de implementação de variabilidade, como a geração em tempo de execução de componentes de software e a sua instalação. Outros sistemas, como o JBoss (FLEURY, 2003), aplicam tais técnicas através dos mecanismos de introspecção e a capacidade de compilação de código em tempo de execução.

A técnica de introspecção (do inglês, *Introspection*) (SUN MICROSYSTEMS, 2009) permite o carregamento dinâmico de tipos, a instanciação dinâmica de objetos e também a execução dinâmica de métodos, além da geração dinâmica de código. A importância dessas características é que podem trazer quase todas as possibilidades existentes nos ambientes de desenvolvimento durante as fases anteriores à distribuição do software para fases posteriores a esta.

A adoção de técnicas de variabilidade não visa apenas o suporte de construção de múltiplas aplicações a partir de uma base comum, podendo também ser aplicada no desenvolvimento de uma única aplicação. Neste caso, o objetivo é acrescentar flexibilidade e capacidade de evolução à aplicação, isto é, facilitar a possibilidade de alterações na aplicação. Segundo Bragança e Machado (2004), de uma forma geral, quando se considera que a implementação de uma característica da aplicação pode vir a ser alterada no futuro, então deve-se adotar uma técnica de implementação de variabilidade.

2.6 A Engenharia de Domínio

Um dos fatos mais conhecidos da engenharia de software é que os requisitos de sistemas sempre mudam, ou seja, os sistemas estão em constante evolução (SOMMERVILLE, 2007). Todo sistema, com o tempo, passa por um processo de defasagem tanto em relação à tecnologia utilizada quanto às próprias necessidades dos usuários. É crescente a necessidade de atualizações e adição de novas funcionalidades ao sistema.

A adoção de técnicas de implementação de variabilidade, particularmente técnicas suportadas em tempo de execução, promete níveis de flexibilidade e evolução mais elevados para as aplicações. Por outro lado, com a adoção de tais técnicas, é elevado também o nível de complexidade desses sistemas, podendo dificultar o desenvolvimento e manutenção. Uma forma de gerir essa complexidade está na adoção de métodos de engenharia de domínio, aplicados em paralelo com o método de engenharia de aplicação. Bragança e Machado

(2004) apresentam uma abordagem centrada na adoção de métodos de desenvolvimento de software baseados em engenharia de domínio como forma de abordar o desenvolvimento de aplicações complexas com elevado grau de variabilidade.

A Engenharia de Domínio é uma técnica que provê suporte ao desenvolvimento baseado em componentes (DBC). É um processo que visa identificar, representar e implementar artefatos reutilizáveis de um domínio (SEI, 2007). Trata-se de uma abordagem sistemática que vai desde a especificação do domínio até a implementação de artefatos. O objetivo é fornecer um grupo de artefatos reutilizáveis dentro de um domínio específico que podem ser aplicados no desenvolvimento de novos produtos.

A Engenharia de Domínio se baseia na busca de características comuns e variabilidade para alcançar o conhecimento de um determinado domínio. O processo é dividido em três etapas: Análise do Domínio, Projeto do Domínio e Implementação do Domínio (SEI, 2007). Cada uma destas etapas gera um produto final, modelo de domínio, modelo de projeto e artefatos reusáveis, respectivamente. É, portanto, na terceira etapa que são implementados os artefatos que podem ser aplicados no desenvolvimento de outros sistemas. Este esquema pode ser observado na Figura 2.3. Abordaremos cada etapa nas seções a seguir.

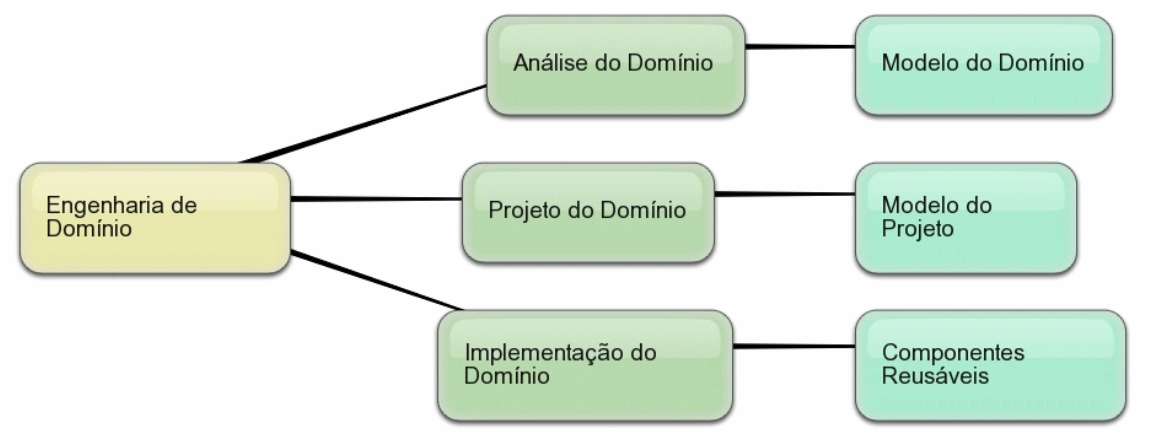

Figura 2.3: Engenharia de Domínio

2.6.1 Análise do Domínio

A Análise de Domínio é a etapa responsável pela identificação, coleção, organização e representação de informações relevantes ao domínio. Essa fase é composta das atividades referentes à definição do domínio, tais como: definição do escopo do domínio, descrição dos componentes do domínio, modelagem do diagrama de integração dos componentes e especificação dos modelos estruturais do domínio.

A Figura 2.4 ilustra que a partir do conhecimento do domínio, temos como produto final desta etapa o Modelo do Domínio. Nesta etapa, é feita a busca pelas semelhanças e variabilidades com o objetivo de ter um maior entendimento do domínio. Dessa forma, temos o conhecimento e a representação do domínio, resultando no Modelo do Domínio como produto final.

Figura 2.4: Análise do Domínio

2.6.2 Projeto do Domínio

O Projeto do Domínio é um processo de desenvolvimento de um Modelo de Projeto. Tal processo possui como base os produtos resultantes da Análise do Domínio. O objetivo é desenvolver uma arquitetura genérica de sistema, sem a preocupação quanto à implementação, que suporte a reutilização de artefatos no domínio.

As atividades realizadas nessa fase são: definição das arquiteturas suportadas no domínio, refinamento dos modelos estruturais e especificação da interface básica entre os componentes. A Figura 2.5 mostra que a partir da Arquitetura do Sistema, gerada nessa fase, e do Modelo do Domínio, gerado na etapa anterior, é especificado o Modelo do Projeto.

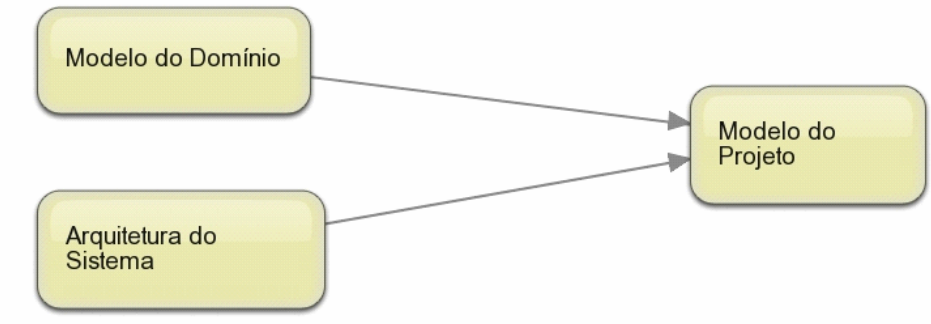

Figura 2.5: Projeto do Domínio

2.6.3 Implementação do Domínio

Tendo como base o Modelo do Projeto formado na etapa anterior, podemos observar na Figura 2.6 que, nesta fase, ocorre o processo de identificação e implementação de artefatos reutilizáveis, por exemplo, componentes de *software*. Esta implementação é baseada em padrões de projeto, já que são soluções desenvolvidas e testadas para problemas freqüentes durante o desenvolvimento de *software*, aumentando assim a produtividade e a qualidade.

É desenvolvida também uma biblioteca de componentes, que visa atender a necessidade de uma qualificação e classificação dos componentes, facilitando a sua busca e recuperação. Essa fase compreende, portanto, as atividades de codificação e testes dos componentes, além do empacotamento do código legado.

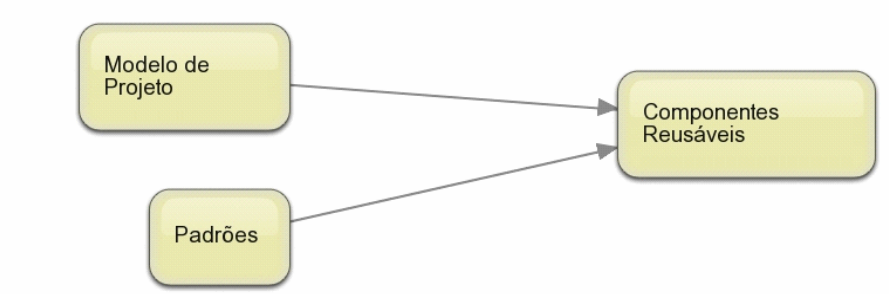

Figura 2.6: Implementação do Domínio

Os produtos da engenharia de domínio irão influenciar nas atividades e, conseqüentemente, no desenvolvimento dos produtos da engenharia de aplicação. Esta trata do estudo das melhores técnicas para a produção de aplicações específicas no contexto do desenvolvimento baseado em reutilização.

Assim, os requisitos da aplicação são analisados conforme o modelo de domínio a fim de gerar a especificação do sistema, e a arquitetura será gerada com base no modelo de projeto. Finalmente, o desenvolvimento da aplicação será baseado na biblioteca de componentes reusáveis.

2.7 A Engenharia de Aplicação

A engenharia de aplicação é o estudo das melhores técnicas, processos e métodos para a produção de aplicações no contexto do desenvolvimento baseado em reutilização (GRISS et al., 1998). A engenharia de domínio e a engenharia de aplicação trabalham de maneira colaborativa no sentido de troca de recursos. Os artefatos produzidos pela engenharia de domínio podem ser reutilizados na engenharia de aplicação para cada nova aplicação construída dentro do domínio e esta fornecerá conhecimento para refinar os mesmos artefatos do domínio ou para construir novos artefatos.

Segundo Krueger (1992), para que uma técnica de reutilização de software possa ser efetiva, ela deve reduzir a distância cognitiva entre o conceito inicial de um sistema e sua implementação final. Sendo assim, no contexto de um processo de desenvolvimento de software baseado em reutilização, o reúso nas fases iniciais do desenvolvimento deve ter como resultado a reutilização de componentes na implementação da aplicação. Ou seja, deve haver uma relação estreita entre os conceitos de um domínio nas fases iniciais do processo de desenvolvimento e os componentes codificados que serão utilizados na implementação da aplicação (BRAGA, 1999). Assim, o processo de reutilização só

será efetivo quando for utilizado um processo de engenharia de domínio capaz de criar componentes de qualidade em todas as fases do desenvolvimento e um processo correspondente de engenharia de aplicação que seja capaz de utilizá-los.

Os produtos resultantes da Engenharia de Domínio (Modelo do Domínio, Projeto do Domínio e Biblioteca de Componentes) irão influenciar nas atividades, e, portanto, no desenvolvimento dos produtos da Engenharia de Aplicação. Isto porque é na Engenharia de Aplicação que temos as aplicações específicas, onde os componentes do domínio são efetivamente utilizados na construção de aplicações reais. Os requisitos do sistema são analisados conforme o modelo do domínio, gerando assim a especificação do sistema. A arquitetura será gerada com base no modelo do projeto (ou arquitetura do domínio). Finalmente, o desenvolvimento da aplicação será realizado com base na biblioteca de componentes reusáveis. A Figura 2.7 (FOREMAN, 1996) apresenta o esquema descrito, conhecido como o modelo dos dois ciclos de vida.

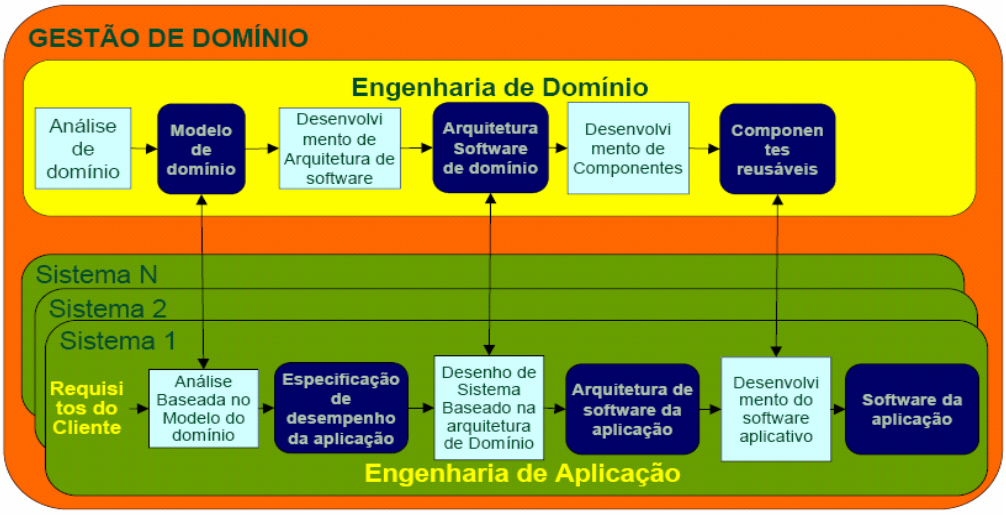

Figura 2.7: As Engenharias de Domínio e Aplicação. Fonte: (FOREMAN, 1996)

2.8 A Variabilidade e as Engenharias de Domínio e Aplicação

Quando apresentamos o conceito de variabilidade, pudemos observar que o grau de variabilidade quantifica a capacidade de uma aplicação suportar alterações. Portanto, esse constitui também uma medida da flexibilidade da aplicação. Sendo assim, a introdução de pontos de variabilidade no ciclo de vida do software aumenta a flexibilidade das aplicações. Como exemplo, podemos citar técnicas de suporte de variabilidade em tempo de execução, como carregamento dinâmico de bibliotecas, que tornam possível a introdução de novas variantes sem ser necessária a alteração das aplicações.

Na seção anterior, vimos como a variabilidade pode auxiliar no processo de desenvolvimento e evolução de um software. Por outro lado, é importante observar que o uso da variabilidade com o objetivo de aumentar a flexibilidade da aplicação somente é aconselhado nos casos em que não é necessário regressar

às fases mais iniciais do ciclo de vida do software. Caso isso aconteça, o processo de desenvolvimento do software pode se tornar mais complexo.

Com o objetivo de controlar o nível de complexidade que as técnicas de implementação de variabilidade podem impor ao processo de desenvolvimento, Bragança e Machado (2004) apresentam uma proposta baseada na engenharia de domínio. Trata-se de um processo que visa identificar, representar e implementar artefatos reutilizáveis de um domínio (SEI, 2007). Exemplos de métodos que aplicam princípios de engenharia de domínio são o FODA (KANG, 1990), sigla para *"Feature-Oriented Domain Analysis"* e o RSEB (GRISS *et al*., 1998), sigla para *"Reuse-Driven Software Enginnering Business"*. Tais métodos são aplicados apenas quando diversas aplicações partilham o mesmo domínio.

A proposta apresentada por Bragança e Machado (2004) mostra que ao invés de adotar a Engenharia de Domínio como um processo que visa suportar o desenvolvimento de diversas aplicações de um domínio, esse pode ser adotado para apoiar o desenvolvimento de variantes e respectivos pontos de variabilidade na Engenharia de Aplicação comum. Nesse caso, consideramos cada ponto de variabilidade como um domínio. A Figura 2.8 apresenta as interações possíveis entre a Engenharia de Domínio e a Engenharia de Aplicação. Os artefatos produzidos na engenharia de domínio podem ser reutilizados na Engenharia de Aplicação.

A Engenharia de Aplicação atua de forma paralela à Engenharia de Domínio, sendo a última responsável pela construção dos componentes reutilizáveis que

representam o domínio (processo conhecido como desenvolvimento para reúso). Na Engenharia de Aplicação, os componentes construídos na Engenharia de Domínio são utilizados para o desenvolvimento de um produto de software (desenvolvimento com reúso).

Na Figura 2.8, as setas mostram a colaboração entre as engenharias de domínio e de aplicação, sendo essa fornecedora de entradas para a engenharia de domínio, principalmente em forma de conhecimento do domínio. Assim, existe uma troca: de um lado estão os componentes, que possibilitam a reutilização; de outro está o conhecimento, que possibilita o refino dos mesmos componentes ou a criação de novos. Tais componentes gerados na engenharia de domínio são aqueles comuns, reutilizados nas aplicações, e aqueles que implementam os pontos de variabilidade e que são usados apenas em algumas aplicações, necessários quando o objetivo é construir aplicações flexíveis e que possam evoluir mais facilmente.

Apesar das vantagens oferecidas pela engenharia de domínio, de forma a controlar a complexidade das aplicações que fazem uso do conceito de variabilidade, a proposta atual provoca uma grande proliferação de componentes, visto que é baseada na declaração estática, portanto irrevogável. Isto faz com que cada nova alteração em algum componente leve à necessidade de criação de um segundo componente. Apresentaremos nos capítulos seguintes uma proposta de declaração dinâmica, que visa mitigar esse problema de proliferação de componentes, inserindo o conceito de adjetivos no desenvolvimento de softwares.

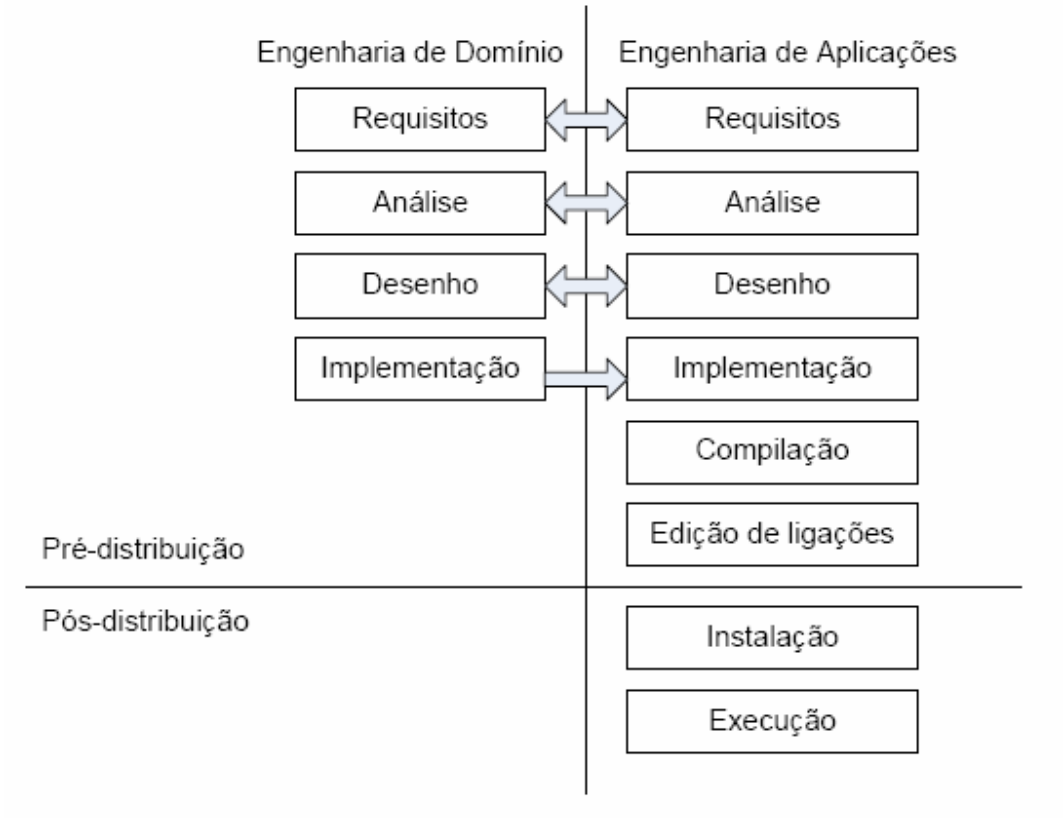

Figura 2.8: Engenharia de Domínio e Engenharia de Aplicação. Fonte: (BRAGANÇA, MACHADO, 2004)

2.9 Considerações Finais

Neste capítulo, apresentamos um estudo realizado da literatura já existente sobre os pontos que levantamos durante a execução deste trabalho. Este estudo serviu como base para o desenvolvimento do mecanismo aqui proposto. Explicitamos, ainda, os conceitos e as tecnologias envolvidas na elaboração desse mecanismo. No próximo capítulo, apresentaremos os trabalhos existentes nas áreas de geração automática de sistemas e engenharia de domínio.

3. Trabalhos Relacionados

Neste capítulo, apresentamos trabalhos que serviram como fonte de pesquisa para os assuntos abordados por esta dissertação. Na primeira seção discutimos trabalhos de grande relevância para o estudo de técnicas de geração automática de código. Na segunda seção, apresentaremos o Hércules, que será utilizado como arcabouço de geração automática neste trabalho. Uma das contribuições deste trabalho está na extensão do arcabouço. Na terceira seção, apresentaremos o Megara, uma proposta de utilização da arquitetura MDA (do inglês, *Model Driven Architecture*) (WARMER, KLEPPE, BAST, 2003) para o arcabouço Hércules,

3.1 Geração Automática de Código

Uma importante fonte de pesquisa pode ser encontrada em (MOLINA, 2003), onde é proposta uma extensão para a especificação de interface gráfica sobre modelos conceituais orientados a objetos. O autor apresenta um método para geração de interface gráfica a partir de uma linguagem mais abstrata aliada à geração de código. O objetivo principal é aumentar a qualidade e produtividade durante todo o processo de desenvolvimento.

O interesse na geração automática de código é crescente. Em (PELECHANO, PASTOR, INSFRÁN, 2002), é apresentado um processo de geração de código a partir de modelos conceituais. O processo é baseado na orientação a objetos e no uso de padrões de projeto. Além disso, é proposto um

modelo formal para especificar padrões conceituais, como especializações, classes e tipos de agregações, seguindo o conceito da linguagem UML.

Em (JIANCHENG, XUDONG, LEI, 2007), é proposto um modelo de interface gráfica que suporta a geração automática para promover o desenvolvimento de software baseado em modelos de apresentação de sistemas. O objetivo do modelo é tornar a interface gráfica mais intuitiva e de fácil aceitação. Na pesquisa, modelos de dados e padrões de projeto relacionados com interface gráfica são isolados como componentes de modelagem da interface, facilitando a apresentação e construção da vista.

Segundo Krueger (1992), o processo de desenvolvimento de software pode ser dividido em duas etapas: desenvolvimento para reutilização e desenvolvimento com reutilização. Entre as abordagens que provêem suporte ao desenvolvimento para reutilização, destaca-se o método Catalysis (D' SOUZA, WILLS, 1998). O processo de DBC em Catalysis compreende a identificação dos requisitos do domínio do problema para o qual se deseja construir os componentes, prossegue com a especificação dos componentes com base nos requisitos identificados, e finalmente conclui com o desenvolvimento dos componentes. Baseia-se no princípio de refinamentos sucessivos, para obter componentes com maior grau de reutilização e mais fáceis de serem reutilizados.

Levy (1986) afirma que, para valer a pena o seu emprego, o custo de desenvolver e manter um gerador de artefatos deve ser superado pela economia obtida através da redução do custo de construção e manutenção dos artefatos

gerados. Para alcançar essa meta, foram desenvolvidos processos de construção baseados em meta-geradores, que são geradores de geradores de artefatos. O paradigma Draco (NEIGHBORS, 1984) é um exemplo de meta-gerador baseado em domínios, que se destaca pela definição de conceitos sobre análise de domínio e transformações. “Em (FREEMAN,1987), percebemos que o paradigma Draco pode ser caracterizado por diversos pontos de vista: originalmente, foi desenvolvido para prover um método para construir software a partir de componentes preexistentes; pode ser utilizado na construção de geradores de software capazes de gerar e manter um conjunto de sistemas similares; e finalmente como um conjunto de linguagens de domínios utilizadas no processo de construção de sistemas, através de um mecanismo transformacional capaz de gerar um programa executável a partir de uma especificação descrita em uma linguagem específica a um domínio (DSL – Domain Specific Language)” (BERGMANN, 2003). No paradigma Draco, os elementos reutilizáveis, chamados de domínio, são definidos por linguagens formais. Essas linguagens são construídas com o objetivo de encapsular objetos e operações do domínio, e assim possibilitar a descrição da sintaxe (através de uma gramática) e da semântica (através de transformações e refinamentos). Essas linguagens vão atuar no desenvolvimento semi-automático de código executável. O engenho envolve transformações inter e intra-domínios, que compreendem refinamentos e otimizações.

Além de sua importância como sistema transformacional, o Draco mostrou-se um ambiente propício para realização de diferentes experimentos na área de

Engenharia de Software. Desde a sua primeira versão, o Draco passou por várias evoluções destacando-se a Máquina Draco-PUC (LEITE, SANT'ANNA, FREITAS, 1994), que tem sido atualizada e disponibilizada para diferentes grupos de pesquisa, tornando-se instrumento de diversos trabalhos (FILETO et. al, 1996; FREITAS, LEITE, 1997; SANT'ANNA, LEITE, PRADO, 1998). A Máquina Draco-PUC implementa o paradigma Draco com o objetivo de construir uma implementação real por tradução de todas as linguagens do domínio, sendo os domínios executáveis os alvos das transformações. Segundo Freitas e Leite (1997), o processo em que uma especificação escrita em um domínio fonte atinge um domínio alvo pode passar por inúmeros domínios intermediários. Esta propriedade diferencia o Draco-PUC de um Gerador de Geradores simples, pois, além de construir o Gerador, a máquina Draco-PUC permite que isto seja feito reutilizando outros domínios, o que acreditamos facilitar bastante o trabalho de construção das transformações de traduções.

Em (PRADO, LEITE, BERGMANN, 1996) é apresentada a construção e o uso de um ambiente de apoio ao desenvolvimento orientado a objetos. O ambiente integra uma interface gráfica com o sistema transformacional da máquina Draco-PUC. Esse trabalho se destaca pela utilização da tecnologia de transformação que permite a geração automática de código C++ a partir de especificações em alto nível de abstração. Dessa forma é apresentado um uso real para a utilização da Máquina Draco-PUC na geração de código.

Um exemplo de ferramenta CASE (do inglês, *Computer-Aided Software Engineering*) que se destaca na fase de implementação de componentes é a

CASE Bold (BOLDSOFT, 2009). Essa ferramenta é voltada para a construção de componentes dirigida pelo modelo (MDA). Os modelos UML da aplicação que será construída (ou dos componentes da aplicação) devem ser criados em uma ferramenta que os persista na linguagem MDL (do inglês, *Modeling Domain Language*) (BOOCH et al, 1998) ou XMI (do inglês, *XML Metadata Interchange*) (OMG, 2007). A partir desses modelos, a ferramenta Bold é capaz de gerar código orientado a objetos. Contudo, não se trata de uma ferramenta flexível, já que não são permitidas alterações no processo de geração de código.

Bossonaro et al (2003) apresenta um estudo sobre a geração de código em uma linguagem orientada a objetos a partir de modelos UML persistidos em MDL. O caso apresentado trata da construção de um sistema de cardiologia e apresenta uma comparação da utilização das abordagens Draco-PUC e a ferramenta Bold. O estudo conclui que é possível automatizar a implementação dos componentes utilizando ambas as ferramentas, porém "a ferramenta Draco-PUC é uma opção mais viável, tendo em vista ser uma ferramenta que gera código em qualquer linguagem de programação, permitindo a geração de código mais otimizado, facilitando a manutenção" (BOSSONARO et al, 2003). Além disso, a Draco-PUC permite a interação do engenheiro de software nas transformações para direcionar o processo de geração de código, sendo assim mais flexível.

Issa (2006) apresenta um trabalho cujo objetivo é produzir um método de transformação e uma ferramenta de apoio ao desenvolvimento de software baseado no paradigma MDA. Tal ferramenta deve permitir a modelagem e a geração de parte de aplicações, incluindo as interfaces com o usuário. Como não

faz parte do escopo do MDA tratar questões ligadas a interfaces, o foco do trabalho foi voltado em como tratar a modelagem de interface dentro do MDA. Para isso, foi desenvolvido um método de desenvolvimento e geração de interfaces (MDGI), que compreende a criação de dois modelos independentes de tecnologia: um que descreve o domínio do sistema; e outro que descreve apenas a interação do sistema com o usuário. Além disso, o MDGI necessita de um modelo responsável pela transformação do primeiro modelo no segundo. Com essa separação de modelos, o trabalho provê uma modelagem de interação com o usuário em alto nível, ou seja, independente de tecnologia, que pode ser reaproveitado, além de o mecanismo de transformação permitir que o conhecimento de utilização de determinada tecnologia seja embutido na própria transformação.

Gohil (2001) e Shirota (1997) apresentam com sucesso a idéia de geração de código relativo a interfaces de usuário utilizando metadados extraídos da própria estrutura de modelo de dados do sistema. Mrack (2008) por sua vez, apresenta uma proposta para geração de interfaces de usuário em tempo de execução também baseada em metadados provenientes do próprio modelo de dados do sistema, porém sem a geração de código fonte. A idéia apresentada é a geração direta das interfaces durante a execução do sistema. Assim, os metadados são traduzidos nos elementos da interface do usuário durante a própria execução do sistema por um mecanismo interpretador. Além disso, o trabalho acrescenta os conceitos de conhecimento empírico, heurísticas sobre o

sistema e regras de inferência de forma a aumentar a qualidade das interfaces geradas, reduzindo assim os custos de construção.

Nesta seção apresentamos um estudo detalhado sobre geração automática de código e os benefícios conseqüentes dessa prática no desenvolvimento de sistemas. Cada um apresenta a sua proposta de especificação, seja através de uma metodologia ou através de um modelo formal. O método Catalysis foi de suma importância para apoiar uma das etapas do desenvolvimento desta dissertação, o processo de modelagem dos componentes visuais de acordo com os requisitos do sistema. Os trabalhos relacionados ao paradigma Draco mostram a importância que a orientação a objetos tem ganhado nos últimos tempos para o desenvolvimento de software e em especial a geração automática aliada à prática de reuso de artefatos. Apresentamos também a aplicação da abordagem MDA através da ferramenta Bold, além de um estudo comparativo entre esta e a máquina Draco-PUC. Outro estudo importante sobre a utilização de MDA no desenvolvimento de sistemas foi o trabalho que apresentou o modelo MDGI, uma proposta de utilização do MDA com foco maior em interfaces com o usuário. Apresentamos também estudos sobre geração automática de interfaces a partir de metadados do modelo de dados.

A abordagem proposta por este trabalho se diferencia dos demais devido ao foco na implementação dos componentes, e não na modelagem destes. No MDA, o foco é centrado no modelo e não no código-fonte (ISSA et al, 2006). Neste trabalho não trabalharemos na modelagem dos componentes e em como os modelos resultantes poderiam servir de entrada para uma ferramenta de geração

de código. Trabalhamos diretamente na implementação destes componentes utilizando a linguagem de programação, ou seja, trabalhamos em um nível diferente de abstração. Os trabalhos referentes à geração automática de interfaces de sistemas se assemelham ao objetivo desta dissertação pelo fato de serem baseados na interpretação de metadados provenientes do modelo de dados do sistema para a construção dessas interfaces. Esta dissertação também prega a interpretação de dados que são descritos no modelo da vista de acordo com a sua apresentação na interface com o usuário. Porém, além de possuir o foco inteiramente na codificação do sistema, a implementação é proposta de uma forma mais descritiva, com o objetivo de alcançar um melhor mapeamento dos requisitos na aplicação, possibilitando a reutilização desses componentes, e melhorando consequentemente sua manutenção.

Com base no paradigma de geração automática de código, esta dissertação propõe a descrição de interfaces de um sistema através de um descritor enriquecido de dados, de forma a prover maior reúso e um código mais inteligível. O importante é combater a proliferação de componentes e tornar o código o mais auto-explicativo possível. Para isso, propomos a especificação de tipos elementares de dados, que são levantados de acordo com os requisitos do sistema que será construído. Esses tipos são especificados através de métodos como uma forma de prover maior reúso de código.

Nas próximas seções apresentaremos os projetos que serviram como base para o desenvolvimento desta dissertação. O primeiro arcabouço apresentado foi utilizado como engenho de geração automática de código durante o

desenvolvimento deste trabalho. Veremos no decorrer deste estudo que esse arcabouço teve suas funcionalidades estendidas com a finalidade de melhorar a manutenção do sistema, provendo um código mais inteligível. O trabalho seguinte é de grande importância pela apresentação de outra abordagem existente na área de geração automática, a MDA.

3.2 Arquitetura de Controle Hércules

O Projeto "*Arquitetura de Controle Hércules: a base para a geração automática de sistemas de informação com ênfase na camada de controle"* (PAIS, 2004) constitui uma plataforma para desenvolvimento de sistemas de informação. O Hércules desenvolve um arcabouço para a produção de sistemas que possibilita a geração automática de código a partir de diagramas UML (PAIS, OLIVEIRA, 2001). Permite assim que o desenvolvedor concentre-se na lógica de negócio, deixando as tarefas repetitivas para o engenho de geração automática. O código fonte é gerado automaticamente a partir de descrições de alto nível, como UML e metadados.

A arquitetura do Hércules propõe a divisão do sistema em módulos que representam os casos de uso do sistema (PAIS, 2004). Casos de uso são cenários que descrevem interações que um ator (usuário ou sistema) realiza com o sistema (FOWLER, 2005). O arcabouço impõe um padrão de implementação que possibilita a tradução do roteiro de um caso de uso em código fonte. A lógica contida no cenário de caso de uso determina como o ator solicita a realização de funcionalidades ao sistema, e como o sistema deve apresentar o resultado

correspondente (PAIS, 2004). Assim, todos os desenvolvedores seguirão os mesmos passos para a construção dos módulos e qualquer membro da equipe poderá dar manutenção em qualquer parte do sistema.

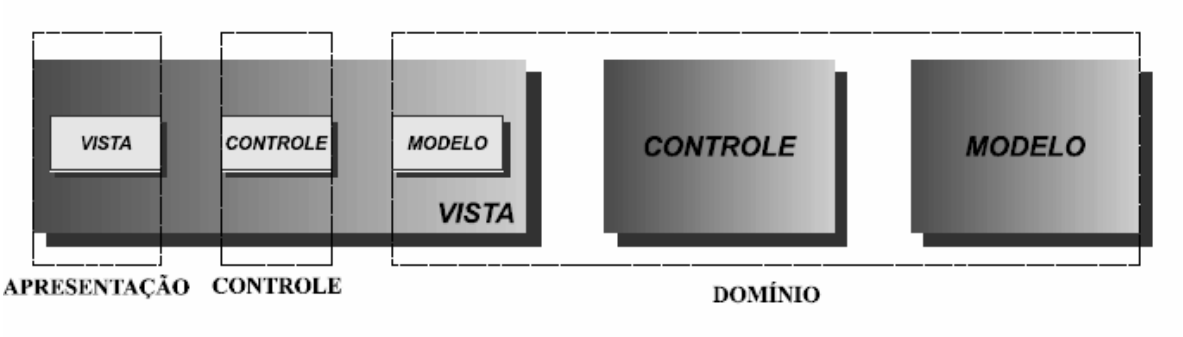

Figura 3.1: Aplicação recursiva do padrão MVC na arquitetura do Hércules.
Fonte: (PAIS, 2004)

A Figura 3.1 expõe a aplicação recursiva do padrão MVC (do inglês, *Model-View-Controller*) (SUN MICROSYSTEMS, 2008) dando ênfase à camada de vista, proposta pelo Hércules. A Vista é dividida em três partes: modelo, vista e controle. A camada de modelo é constituída das informações das entidades que serão mostradas durante o caso de uso (espelho da entidade). A camada de vista é responsável pela apresentação e pela captura de solicitações do usuário. A camada de controle é responsável pelo recebimento das solicitações vindas da vista e pelo acionamento da execução das regras de negócio correspondentes implementadas no controle da aplicação.

A arquitetura do Hércules utiliza os conceitos da arquitetura MVC e a aplicação recursiva da tríade para estabelecer a divisão lógica em domínio, apresentação e controle. A camada de domínio é formada pelo Modelo, pelo Controle e pelo Modelo da Vista, contendo as entidades de domínio, as regras de

negócio e sendo responsável por prover o espelho da entidade. A camada de Controle é constituída pelo controlador de caso de uso e a camada de Apresentação é constituída pela vista da Vista.

O Hércules estabelece um conjunto de especificações de alto nível que são utilizadas pelos desenvolvedores para descrever as três partes do caso de uso: a apresentação, fluxo de execução do caso de uso e os espelhos de entidade. Essas descrições são fornecidas ao controlador do caso de uso durante sua inicialização, instruindo o seu funcionamento. A Figura 3.2 apresenta o esqueleto de especificação do caso de uso descrito em um documento XML. Dessa forma, o desenvolvedor descreve o caso de uso, que é formado por três seções: *view*, *control* e *model*.

```
<mvc>
    <view/>
    <control/>
    <model/>
</mvc>
```

Figura 3.2: Especificação Abstrata do caso de uso

As marcas (*tags*) do documento XML podem possuir atributos que auxiliam na descrição do elemento representado pela marca. A seção view contém os elementos que compõem a interface com o usuário. O nome da marca especifica

o tipo do componente gráfico que deve ser construído. Todos os componentes visuais que exibem uma informação que está armazenada em um elemento do modelo possuem um atributo chamado *datasources* e um chamado *ref*. O atributo *datasources* indica o espelho de entidade e o *ref* indica a propriedade do espelho de entidade que deve ser exibida. Além disso, essa seção mostra a maneira como os elementos devem estar organizados na tela. Na Figura 3.3 é mostrado um trecho de uma especificação.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<mvc>
     <view id="dummyId" title="emissaoHistoricoAndBoletim">
          <tabbox id="tabbedMenu">
               <tabs>
                    <tab label="Lista" selected="false" />
               </tabs>
               <tabpanels>
                    <tabpanel id="Lista">
                         <hbox align="center" id="listaAlunos" width="98">
                              <grid id="tableListaAlunos"
                                        datasources="br.ufrj.siga.historico.emissao.vo.AlunoListVO"
                                        ref="listItensVO" class="listTable">
                                   <columns>
                                        <column value="Nome" />
                                        <column value="Matrícula" />
                                        <column value="Curso" />
                                   </columns>
                                   <rows>
                                        <template>
                                             <row>
                                                  <label id="nome"
                                                       datasources="br.ufrj.siga.historico.emissao.AlunoListItemVO"
                                                       ref="nome"
                                                       class="normal"/>
                                                  <label id="matricula"
                                                       datasources="br.ufrj.siga.historico.emissao.AlunoListItemVO"
                                                       ref="matricula"
                                                       class="normal"
                                                       onclick="emitir"/>
                                                  <label id="curso"
                                                       datasources="br.ufrj.siga.historico.emissao.AlunoListItemVO"
                                                       ref="curso"
                                                       class="normal" />
                                             </row>
                                        </template>
                                   </rows>
                              </grid>
                              <caption label="Escolha uma Matrícula:" />
                         </hbox>
                    </tabpanel>
               </tabpanels>
          </tabbox>
     </view>
     ...
</mvc>
```

Figura 3.3: Exemplo de uma seção view

O Hércules é utilizado pelos sistemas com a finalidade de padronizar a implementação de seus casos de uso. Desta forma, existe um arquivo XML para cada caso de uso. Isto permite que os desenvolvedores não precisem se

preocupar em escrever código HTML, que é gerado a partir da descrição contida no elemento <view> do documento. O Hércules possibilita, portanto, a padronização da interface a partir de descrições abstratas de vista. Apesar disto, este arcabouço não resolve o problema de maneira geral, já que ainda existe duplicação de código, ao passo que o que ocorre é uma cópia da descrição da vista de serviços que já existe para outros serviços semelhantes. Assim, pode-se notar uma deficiência neste arcabouço, visto que este não possibilita o reúso de descrições de vista.

A proposta desta dissertação visa estender o arcabouço Hércules de forma a mitigar o problema da duplicação de código, melhorando assim a manutenção do sistema. Com foco na especificação da vista dos casos de uso, apresentamos um mecanismo de descrição baseado em um Modelo da Vista que utiliza descritores enriquecidos de informação. Através da utilização do novo Modelo da Vista, pretendemos reduzir consideravelmente a extensão das descrições.

3.3 Megara: Uma Ferramenta para o Desenvolvimento de Sistemas Baseado em Modelos

O crescimento do número de linhas de código e da complexidade dos sistemas torna cada vez mais difícil o atendimento da demanda do mercado com a qualidade requerida e com o prazo determinado. Adicionalmente, o número crescente de usuários introduz novos requisitos como a escalabilidade do sistema e a personalização do serviço. Com isso, há uma quantidade maior de informação que deve ser compartilhada entre os desenvolvedores (PEREIRA, 2006).

O projeto Megara (PEREIRA, 2006), desenvolvido no PPGI/UFRJ prega que a melhor forma de compartilhamento de informações é através de representações abstratas, que escondam detalhes irrelevantes em um determinado contexto. Apesar disso, os curtos prazos impostos pelo mercado normalmente impedem que a etapa de modelagem, quando realizada, seja mantida atualizada com a etapa de desenvolvimento. A distância entre os modelos e o código pode ser diminuída com a atualização dos diagramas, no entanto, o valor adicionado com a atualização é questionável, já que qualquer nova mudança se inicia no código (WARMER, KLEPPE, BAST, 2003).

Com o objetivo de mitigar esse problema, o projeto Megara apresenta uma proposta de geração automática de artefatos a partir de representações abstratas da especificação do sistema. Dessa forma, a etapa de modelagem é valorizada, além de elevar o nível de abstração do desenvolvimento. Tal estudo foi baseado na proposta de Arquitetura Dirigida por Modelo, MDA (do inglês, *Model Driven Architecture*) (WARMER, KLEPPE, BAST, 2003).

A MDA propõe a separação entre a especificação das operações do sistema e os detalhes das funcionalidades de uma plataforma específica. Além disso, como seu próprio nome sugere, o objetivo da MDA é manter a arquitetura do sistema descrita através de modelos. Porém, as regras de transformação são definidas através de linguagens ou ferramentas de transformação, das quais podemos citar a linguagem QVT-Partners (QVT-PARTNERS, 2003) e a ferramenta AndroMDA (AndroMDA, 2009).

O projeto Megara propõe um processo de desenvolvimento baseado em modelos com o diferencial de incluir um modelo representativo das definições das transformações. A descrição das regras em diagramas UML (FOWLER, 2005) aumenta o nível de abstração no processo de desenvolvimento de software, tornando mais fácil para o arquiteto do sistema a modificação da transformação e trazendo mais flexibilidade para os artefatos gerados.

Como prova de conceito, o projeto Megara utilizou o arcabouço Hércules, apresentado na seção 3.1. A partir de um modelo independente de plataforma, é gerado um modelo específico para a plataforma Hércules, utilizando regras de transformação descritas em diagramas UML. A abordagem substitui a descrição do caso de uso em arquivos XML, como defendido no projeto Hércules, pela descrição utilizando diagramas UML. As regras de transformação representam a arquitetura da interface gráfica do sistema, ou seja, o modelo de transformação é representado por um diagrama de classes que descreve como deverão ficar as telas do sistema.

O processo proposto é apresentado na Figura 3.4. Os desenvolvedores do sistema modelam as regras de negócio do caso de uso sem que detalhes de uma plataforma específica sejam considerados (PIM, do inglês *Platform Independent Model*). A ferramenta Megara gera o modelo específico (PSM, do inglês *Platform Specific Model*) baseado no modelo da plataforma e nas regras de transformação. O desenvolvedor poderá alterar os modelos gerados antes de solicitar que a ferramenta gere os artefatos necessários para a execução de um caso de uso na plataforma Hércules (PEREIRA, 2006).

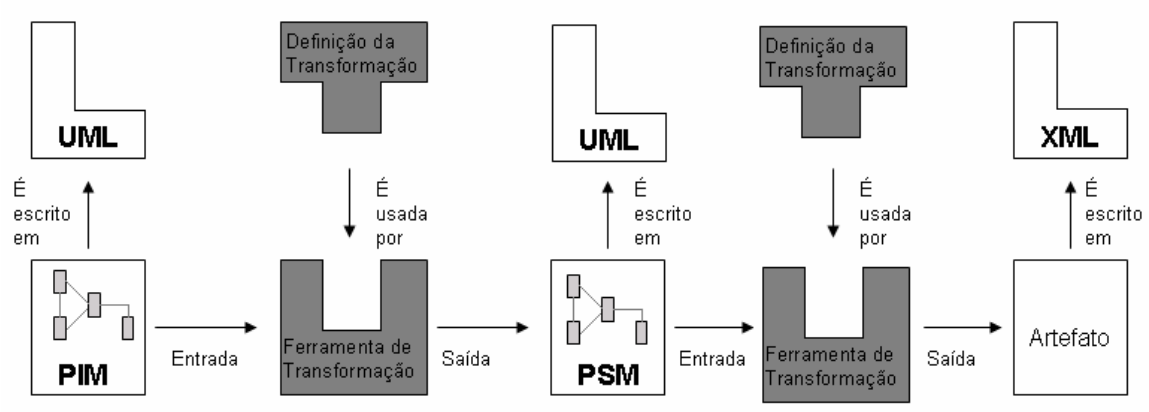

Figura 3.4: Etapas do processo Megara. Fonte: (PEREIRA, 2006)

O processo apresentado complementa a proposta da arquitetura dirigida por modelo, aumentando dessa forma o nível de abstração do processo de desenvolvimento e a flexibilidade do código gerado. Por outro lado, foram encontradas algumas fragilidades no processo, das quais é importante citar: o digrama da camada de Controle não é gerado automaticamente; e o modelo de Domínio não é contemplado pelo processo. Além disso, no principal sistema em que o Hércules é utilizado (apresentaremos o sistema no nosso exemplo de aplicação), houve uma grande resistência quanto à utilização do projeto Megara, devido ao desenvolvimento ser baseado em diagramas UML.

3.4 Considerações Finais

Neste capítulo, apresentamos trabalhos existentes na área de geração automática de sistemas. Vimos que o projeto Megara foca na UML para a construção de sistemas através dos conceitos de MDA; e o arcabouço Hércules, também é baseado na linguagem UML para a construção de sistemas, com a diferença de não utilizar modelos UML, mas descrições em linguagem XML, o que

não resolve o problema da duplicação de código. Este trabalho propõe a geração automática de interfaces a partir de descrições enriquecidas do modelo, substituindo a declaração estática utilizada pelo Hércules por uma declaração dinâmica que provê maior reúso de descritores e um código mais inteligível, com o objetivo de melhorar a manutenção do sistema. No próximo capítulo, apresentaremos o mecanismo proposto.

4. Geração Automática de Código a partir de Descritores Enriquecidos

Uma vez definidos os requisitos a serem atendidos, a construção de uma aplicação é vista como uma tarefa repetitiva para muitos desenvolvedores de software. Apesar do problema a ser tratado a cada novo sistema ser sempre diferente, as estruturas de dados e o código gerado para materializar os sistemas normalmente não mudam tanto de uma situação para outra. Em alguns casos, a implementação de um sistema de informação é feita a partir de uma cópia de outra aplicação construída anteriormente, deixando ao desenvolvedor a sensação de estar sempre tratando de estruturas muito parecidas.

As técnicas apresentadas no Capítulo 2 são utilizadas como uma maneira de mitigar este problema. A área que mais tem chamado à atenção, sendo um grande foco e fonte de pesquisas é a engenharia de domínio. Neste capítulo, apresentaremos os conceitos e implementações de variabilidade e engenharia de domínio, além do principal problema conseqüente desta área, a grande proliferação de artefatos, devido à metodologia utilizada.

A proposta atual da engenharia de domínio provoca uma explosão de componentes na variabilidade, já que é baseada na declaração estática, portanto irrevogável, utilizando atributos. A explosão fatorial ocorre por causa da existência dos atributos nos componentes. Como os atributos são estáticos, então a necessidade de eliminar um atributo de um componente já existente leva à criação de um novo componente. O projeto Hércules, apresentado no Capítulo 3, é um exemplo de arcabouço que segue essa proposta de declaração estática. Isto

porque, como vimos, para cada nova tela do sistema é necessária a criação de uma nova descrição, mesmo que seja uma tela idêntica a outra existente no sistema.

A proposta deste trabalho é introduzir um mecanismo de geração automática para construir a maior parte da interface gráfica de uma aplicação de forma a reduzir a explosão fatorial de artefatos através de uma especificação do Modelo da Vista de nível mais alto com um significado mais definido. Para que isso seja possível, propomos a criação de especificações da vista mais completas usando Tipos Elementares de Dados (LEITE, 2006) especificados através de uma declaração comportamental. Esta declaração deixaria de usar apenas atributos para usar métodos, criando dessa forma uma especificação dinâmica.

A idéia da especificação dinâmica é a criação de adjetivos na especificação da aplicação e a introdução destes adjetivos nos componentes. Dessa forma, teremos componentes que podem ser adjetivados, deixando de ter a criação de novos componentes para cada nova necessidade. Com isso, almejamos a redução da dimensionalidade, melhoria da qualidade e precisão do desenvolvimento, devido ao melhor mapeamento entre o negócio e a implementação.

Os tipos elementares de dados podem ser utilizados para a definição de qualquer tipo de dado comum do sistema. Este trabalho focou no desenvolvimento de tipos elementares de dados para os componentes de vista de um determinado

sistema. Para isso, basta realizar um levantamento de todos os componentes de vista do sistema. Cada componente dará origem a um único tipo elementar.

Para um melhor entendimento da proposta deste trabalho, apresentaremos um exemplo de uma loja virtual. Trata-se de uma loja de livros que disponibiliza a consulta *online* dos livros disponíveis para o usuário. Além de realizar consulta, o usuário tem a opção de imprimir os resultados apresentados. A Figura 4.1 apresenta o resultado de uma busca realizada por um usuário.

Figura 4.1: Apresentação dos resultados de uma consulta realizada no sistema

Imaginemos que o sistema apresentado na Figura 4.1 possui um padrão operacional. Assim existe uma tela de consulta, que permite que o usuário informe os dados do livro, e os resultados são apresentados conforme podemos visualizar. Isso demonstra que é possível gerar automaticamente a descrição da vista a partir da entidade exibida na tela. Bastaria apenas identificar, na entidade, quais atributos devem ser exibidos, em que ordem e quais os títulos dos campos que devem ser exibidos. Este trabalho propõe a criação de notações, que quando adicionadas às entidades, forneçam a informação necessária à geração da descrição de componentes visuais.

As notações atuam tanto nas entidades como nos seus atributos, constituindo um conjunto de descritores visuais do modelo. Uma notação específica foi introduzida para indicar onde os descritores de modelo podem ser acoplados. Os descritores de modelo especificam características que podem ser passadas para apresentação da entidade e descritores específicos para atributos. Os descritores de atributo indicam como o atributo se relaciona com o encaixe. Este mecanismo de geração permite que se use automatização onde se aplicava uma cópia do código. O código estático que seria duplicado deve ser convertido em um descritor de modelo e atributos.

4.1 O Descritor Enriquecido

No Capítulo 2, apresentamos a importância de a aplicação ser capaz de evoluir constantemente. Para que o Modelo da Vista esteja preparado para tais

alterações, de forma a não prejudicar o restante do sistema, é importante que seja de fácil entendimento para qualquer integrante da equipe de desenvolvimento. Para que isso seja possível, propomos um modelo adicionado de informação, trata-se do descritor enriquecido. No Modelo da Vista, o significado dos dados é enriquecido através de anotações e os tipos de dados são suplementados com descritores implementados em forma de métodos.

O aprimoramento do modelo permite maior expressividade na especificação, beneficiando o entendimento do domínio e permitindo, quando necessário, a adição de novas informações de maneira rápida e segura. O modelo é baseado no padrão de negócios *Value Object*, que chamaremos apenas de VO no decorrer deste trabalho. Padrão de projeto J2EE, o VO (ALUR, 2001) é um objeto que é passado através da rede ao invés de passar cada atributo separadamente, aumentando, dessa forma o desempenho. São classes que possuem apenas atributos e seus respectivos métodos *getters* e *setters*. Tal padrão é útil, pois encapsula todas as informações sobre um determinado registro, evitando que seja feito um grande número de chamadas remotas e com isso diminuindo o tráfego pela rede (ALUR, 2001).

Este padrão possui o objetivo de especificar as propriedades de uma entidade para um determinado caso de uso. Tais propriedades são necessárias para obter informações requeridas pelo usuário, além de modificá-las. Além disso, o VO está intimamente ligado à apresentação do serviço. Isto porque as propriedades contidas no VO, em sua maioria, são aquelas que serão expostas ao usuário. Por essa razão, o VO é utilizado neste trabalho como o responsável por

descrever o Modelo da Vista dos casos de uso de um sistema. Assim, o descritor enriquecido agrega descritores que refinam o tipo do VO e seus atributos.

Na Figura 4.2, percebe-se que o enriquecimento do VO se dá a partir de bibliotecas de tipos já definidos. Estes tipos são parametrizáveis de modo a personalizar o VO para que atenda os requisitos do seu papel na aplicação. A seguir iremos explorar os detalhes da Figura, explicitando cada componente.

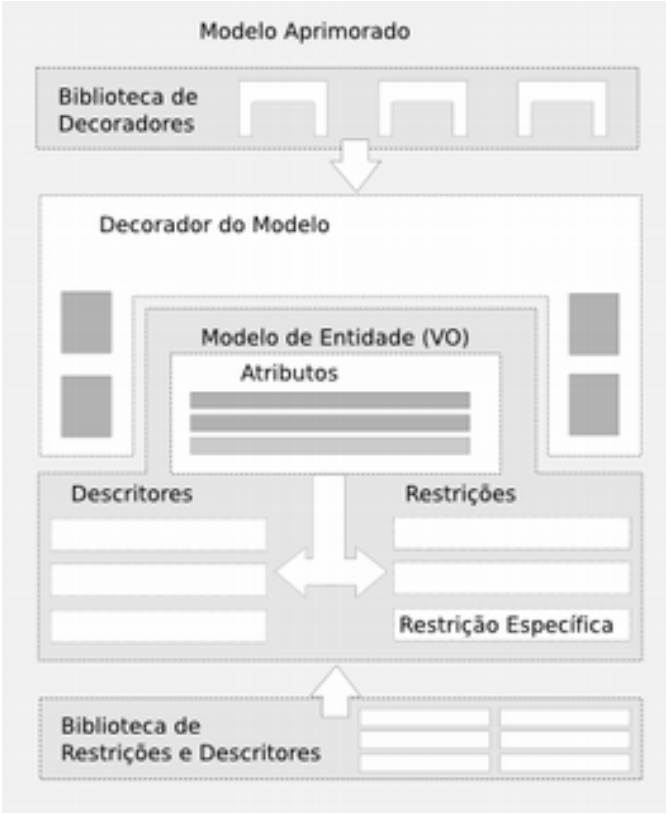

Figura 4.2: Descritor Enriquecido

O elemento “Decorador do Modelo” é o enriquecimento atribuído ao VO como um todo. Ele define o papel que este VO estará representando no caso de uso. Este papel define em que sub-caso este VO será apresentado e que tipo de ação será atribuída a cada instância. Os papéis disponíveis são disponibilizados

pelo componente “Biblioteca de Decoradores” e são levantados a partir do padrão emergente do conjunto de casos de uso da aplicação. Estes decoradores são parametrizados com informações que estão associadas ao sub-caso, provendo maior significado ao VO, tornando-o assim mais inteligível.

Para facilitar o entendimento desta biblioteca e de como o VO é decorado, vamos imaginar o sistema “O Livreiro Digital” apresentado na Figura 4.1. Nessa figura visualizamos a tela de apresentação dos itens do livro consultado anteriormente pelo usuário. A tela de consulta não será exibida neste trabalho, mas imaginemos que seja uma tela onde o usuário é capaz de informar alguns dados sobre o livro para efetuar a busca. Dessa forma, até agora, temos duas descrições de vista para este sistema: uma tela de consulta e uma tela que apresenta os dados resultantes da consulta. A proposta deste trabalho é realizar a descrição dessas telas em classes VO. Assim, a tela de consulta seria descrita em um VO e a tela de resultados em outro. Com a finalidade de marcar cada VO com o seu respectivo objetivo, usamos a “Biblioteca de Decoradores” e o engenho chamado de “Anotação” (do inglês, *Annotation*) do Java 5. Criamos, portanto, dois decoradores: um para anotar o VO que será responsável pela descrição da tela de consulta; e outro, que será responsável pela descrição da tela de resultados. No Capítulo seguinte apresentaremos melhor a proposta do decorador do VO para esse exemplo da loja virtual. Além disso, apresentaremos também maiores detalhes de implementação no capítulo de exemplo de aplicação.

Além do decorador, o VO é composto de atributos que serão de tipos específicos do sistema. Estes tipos são os Tipos Elementares de Dados, que

serão apresentados com maior detalhamento na próxima seção. O conjunto dos tipos elementares de um sistema compõe o componente “Biblioteca de Restrições e Descritores”.

Os tipos elementares estão fatorados em duas naturezas de especificação, sendo uma natureza de forma e outra de conteúdo. Temos assim os elementos “Descritores”, “Restrições” e “Restrição Específica”. O Descritor é o próprio tipo elementar. Este é composto de Restrições, propriedades que caracterizam este tipo de acordo com os requisitos do sistema. Além disso, existe a Restrição Específica, um engenho utilizado para estender o comportamento do tipo elementar sem que este perca a sua essência.

Este modelo permite uma variação maior dos tipos finais sem aumentar o número de tipos originais na biblioteca. O VO suporta a especificação de mapas que associam tipos elementares aos seus atributos. Os tipos são parametrizados de modo a permitir uma variabilidade nos aspectos visuais e de consistência da informação.

O VO será descrito utilizando os tipos elementares de dados, o que significa uma substituição da declaração por atributos por uma declaração por métodos, possibilitando maior reúso de componentes e incorporando significado ao código. O reúso de componentes é garantido pelo engenho de tipos elementares, que irão compor o modelo da vista do sistema. Assim, o mesmo modelo da vista poderá ser apresentado de formas diferentes de acordo com a necessidade, como veremos na próxima seção. Podemos imaginar esse tipo de declaração como o

acoplamento de adjetivos ao modelo. Dependendo da funcionalidade, o modelo terá adjetivos diferentes. Essa é uma maneira de se mitigar o problema da criação de novos modelos sempre que alguma alteração é necessária no modelo já existente.

Para que isso seja possível, utilizamos uma maneira de construção do objeto que inclui ou exclui as suas propriedades de acordo com os parâmetros informados na construção. Explicitaremos melhor esta forma de construção do objeto no capítulo seguinte, que trata da implementação da proposta apresentada por este capítulo.

4.2 Disciplinando o Reúso com Tipos Elementares

A criação de tipos elementares de dados é encorajada para o estabelecimento de padrões no desenvolvimento de software. A idéia surgiu como uma forma de amarrar o padrão e torná-lo mais efetivo, facilitando a sua implementação. Assim, além de adicionar maior informação às propriedades que serão exibidas ao usuário, isto é feito de uma forma mais segura, ao passo que é comprovada a corretude do tipo elementar.

Analisemos agora o caso do sistema “O Livreiro Digital”, apresentado na Figura 4.1, na introdução deste capítulo. Podemos perceber que existe a opção de impressão na tela de consulta do livro. Esta opção apresentará uma nova tela com os dados apresentados da mesma forma que a tela anterior, excluindo apenas as propagandas do sistema.

Se representarmos a entidade “livro”, apresentada na tela de consulta, utilizando VO, percebemos que este mesmo VO pode ser aplicado na tela de impressão da página. Assim, aplicaremos este trabalho na automatização da geração da vista deste sistema. Para isso, levantamos os requisitos de componentes visuais do sistema e, para cada componente, criamos um tipo elementar de dado. Dessa forma, para descrever a apresentação de novos serviços do sistema, o desenvolvedor apenas utiliza um artifício. Isto significa que toda a engenharia deste artifício fica transparente para o desenvolvedor.

Analisando a interface gráfica do sistema “O Livreiro Digital”, apresentada na Figura 4.1, e levantando os componentes visuais desse sistema, percebemos a necessidade de criação do componente de vista que é apresentado como texto puro na tela. Este componente é utilizado em vários pontos do sistema. Começamos então com o desenvolvimento deste componente da forma mais comum utilizada no sistema. Assim, geramos o tipo elementar “*LabelDescriptor*”, um tipo genérico utilizado para apresentar texto puro na sua forma mais usual.

Na Figura 4.3, apresentamos uma proposta de diagrama de classes formado para o engenho de tipos elementares. Este diagrama é baseado na caracterização dos tipos de acordo com a sua funcionalidade no sistema. Assim, como este trabalho propõe uma extensão para o arcabouço Hércules, realizamos a divisão dos tipos de acordo com a aba onde será apresentado. Podemos notar o tipo elementar “*LabelDescriptor*”. Este tipo é uma especialização do tipo “*ColumnDescriptor*”, o que o caracteriza como um tipo utilizado apenas na descrição de uma vista do tipo lista de resultados (ou aba de lista, no caso do

Hércules). Por sua vez, este último tipo é uma especialização do tipo "*Descriptor*", responsável pela definição de todos os tipos elementares.

Quando for necessária a criação de um tipo que define a vista de consulta, por exemplo, basta que esse tipo herde de "*Descriptor*", como o componente "*FieldDescriptor*" apresentado na Figura 4.3. O mesmo serve para os tipos de dados específicos, como "*TextRestrictionDescriptor*", que é caracterizado como um tipo elementar utilizado para definir um atributo que será exibido na tela de consulta; e "*LinkDescriptor*", que é caracterizado como um tipo elementar utilizado para definir um atributo que será exibido na tela de lista como um link. Como os tipos elementares variam de acordo com os requisitos visuais do sistema, o diagrama apresentado é apenas uma proposta de organização dos tipos baseada na proposta do arcabouço Hércules em conjunto com o sistema "O Livreiro Digital" que utilizamos como exemplo. Ou seja, este diagrama pode ser diferente quando aplicado em outros sistemas.

No mesmo exemplo apresentado na Figura 4.1, percebemos a necessidade de criar uma exceção para o componente "*LabelDescriptor*", já que em alguns pontos do sistema seria necessário apresentar o texto com um tamanho diferente do usual. Para isso, ao invés de criarmos um novo componente, por exemplo, "*LabelDescriptorMaior*", utilizamos o mesmo tipo elementar criado anteriormente. O que ocorre nesse caso é que esse tipo elementar terá o seu comportamento estendido, com a adição de um adjetivo "tamanho". Com isso, através de parâmetros informados durante a descrição da vista, um mesmo tipo elementar pode se apresentar com certa distinção, mas sem perder a sua essência.

Apresentamos na Figura 4.4 a vista do sistema destacando o tipo elementar utilizado.

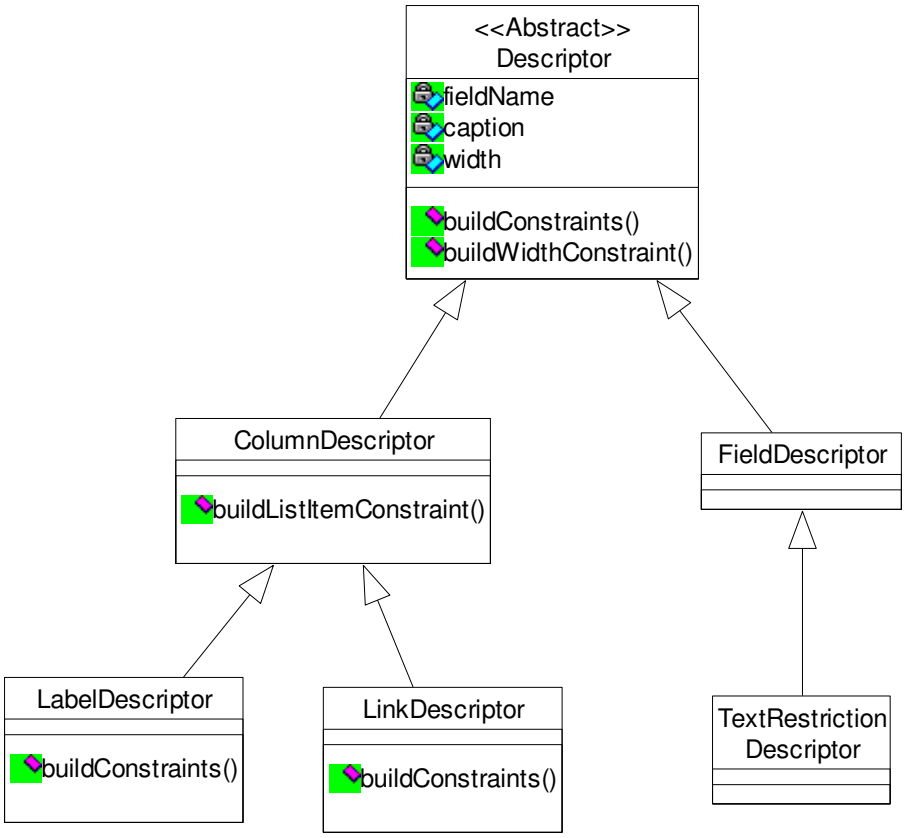

Figura 4.3: Diagrama de classes simplificado para o engenho de tipos elementares

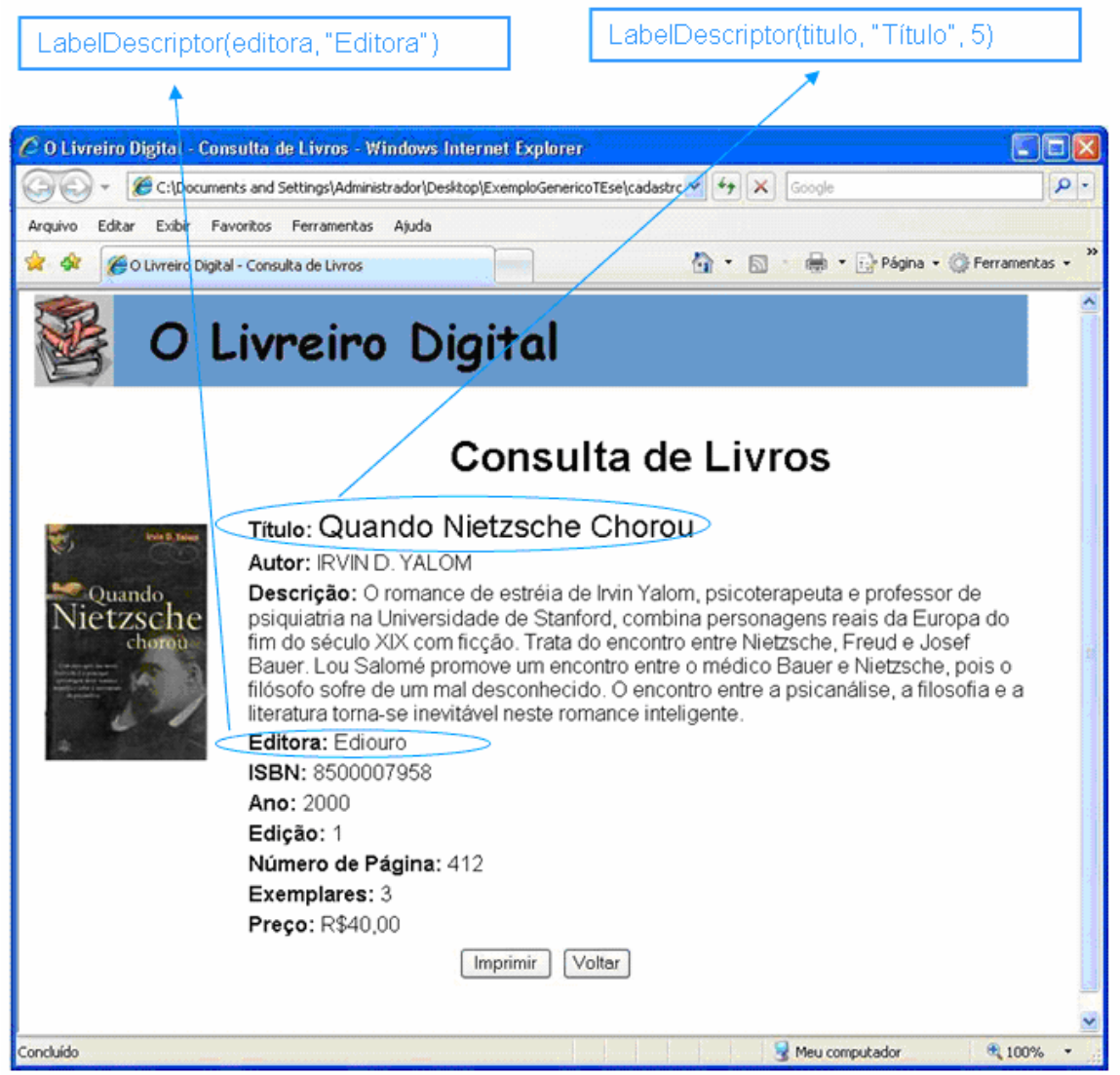

Figura 4.4: Comportamento variável do tipo elementar

Analisando a Figura 4.4, percebemos que os campos “Título” e “Editora” são descritos pelo mesmo tipo elementar “*LabelDescriptor*”. Isto significa que ambos serão apresentados como texto puro na vista. Apesar disso, percebemos uma diferença na declaração do tipo, evidenciando o comportamento variável de apresentação deste. A diferença será informada através do terceiro parâmetro do

tipo elementar, que significa o tamanho do texto apresentado. O campo “Editora” dispensa que o tamanho do texto seja informado na sua declaração, já que será o tamanho padrão do tipo elementar. Já o campo “Título” informa que o tamanho será diferente e de valor “5”. O primeiro parâmetro corresponde à propriedade do VO e o segundo corresponde ao título do campo.

4.3 Considerações Finais

Apresentamos neste capítulo o descritor enriquecido proposto por este trabalho para a construção de interfaces de sistemas de informação. Essa proposta é baseada no objetivo de diminuir a explosão fatorial de componentes decorrente da variabilidade existente nos sistemas e do comportamento estático da declaração atual. Esse modelo é composto de anotações e tipos elementares de dados, que utilizam uma declaração comportamental, possibilitando maior reúso de componentes. O próximo capítulo apresentará o mecanismo, que é composto de uma proposta de implementação para este descritor enriquecido, assim como todo o engenho necessário para que um determinado sistema possa traduzir este modelo em componentes próprios.

5. Apresentação e Implementação do Mecanismo

Este trabalho propõe um mecanismo que possibilite a construção da interface gráfica de um sistema baseado na descrição enriquecida do Modelo da Vista. Utilizamos o arcabouço Hércules como o engenho para geração automática de sistemas. Para o suporte ao mecanismo em um sistema de informação, foram necessárias alterações no comportamento do Hércules, adicionando novas funcionalidades. Este capítulo apresenta a implementação realizada neste arcabouço, assim como as demais implementações necessárias para a utilização do mecanismo em um sistema de informação que utilize os arcabouços Hércules e Chameleon. O Chameleon será apresentado na seção 5.3.1 e é o responsável pela tradução do novo descritor enriquecido em componentes reconhecidos pelo Hércules.

Para isso é necessário enriquecer a informação disponível para a especificação do Modelo da Vista. Dentro dos recursos da linguagem Java, isto significa a utilização de anotações, que permitem introduzir um princípio de meta-modelagem. No Modelo da Vista, o significado dos dados é enriquecido através de anotações e os tipos de dados são suplementados com descritores implementados em forma de métodos.

Este descritor enriquecido impõe novas condições que precisam ser combinadas com os arcabouços existentes. Por conta disso, neste capítulo são descritas as modificações necessárias para tal. Estas modificações requerem a

modificação do código existente, além da criação de novas classes que dêem suporte ao novo descritor enriquecido.

A implementação deste trabalho foi baseada na idéia de automatização da descrição da vista de sistemas de informação. Vimos, na seção 3.1, que o arcabouço Hércules possui uma engenharia de construção da vista dos sistemas baseada em descrições feitas em arquivos XML. Este tipo de descrição é baseado na especificação declarativa, ou estática, o que gera duplicação de código, já que não possibilita reúso. Isto significa que para cada serviço do sistema, existirá um arquivo XML específico, mesmo que os serviços tenham a mesma apresentação visual.

A proposta deste trabalho é substituir a descrição da vista especificada na linguagem XML pelo descritor enriquecido, apresentado na seção 4.1. Dessa forma, passamos a descrever a vista utilizando VO e bibliotecas de tipos, ou tipos elementares de dados. A notação necessária para modificar essa especificação é descrita neste capítulo juntamente com a reengenharia das partes que concorrem para a complementação da especificação omitida no arquivo XML.

5.1 Implementação do Descritor Enriquecido

O descritor enriquecido dependerá da criação de duas bibliotecas de tipos: uma para entidades, e uma para atributos. Conforme podemos observar na Figura 4.2, apresentada na seção 4.1, a biblioteca de entidades corresponde ao elemento “Biblioteca de Decoradores” do modelo, aqui representado pelo VO; e a biblioteca de atributos corresponde ao elemento “Biblioteca de Restrições e Descritores”. A

primeira é utilizada para marcar o VO de acordo com a sua funcionalidade no sistema e a segunda é utilizada para estender o VO adicionando maior significado aos dados e provendo tipos de dados específicos para o sistema. Veremos nas seções seguintes os engenhos utilizados para esses fins.

5.1.1 Biblioteca de Entidade

A biblioteca de entidade é responsável por classificar o modelo (VO) para um determinado fim. A implementação escolhida foi na forma de anotações do Java 5. A vantagem da anotação é que não interfere na sintaxe do VO, sendo completamente externa a este. Além disso, com a utilização da anotação, é adicionado significado ao código, já que toda a informação relevante está contida na classe.

Para melhor entendimento, usaremos o sistema “O Livreiro Digital” como exemplo. A Figura 4.1, exposta na introdução do capítulo 4, apresentou o resultado de uma consulta de livro realizada por um usuário. A forma como foi apresentado este resultado é a mesma para todo o tipo de consulta do sistema. Ou seja, se o sistema possuir um serviço de consulta de revistas, por exemplo, o resultado da consulta seria apresentado utilizando o mesmo aspecto visual, modificando apenas o modelo de dados utilizado, que nesse caso não seria mais “livro”. Com isso, podemos perceber que a construção da vista pode ser automatizada.

Para a automatização da construção da vista que apresenta os resultados de uma consulta, criamos uma anotação. Esta é destinada a decorar o VO

responsável pela descrição visual da tela. Assim, através da anotação é possível identificar o VO como um artifício utilizado para descrição da tela que exibe os resultados de uma consulta. A Figura 5.1 exibe a implementação deste VO.

```
@ListTabbedPaneRenderizable(
            formTitle = "Consulta de Livros",
            tabSheetID = "listaLivro")
public class ListaVO implements Serializable, FieldConstrainable {

      public Collection livros = new LinkedList();
      private static UnmodifiableFieldsConstraints constraints;

      static{

            FieldsConstraints staticConstraints = VOConstraintsUtilities.
                        buildTableConstraints(ListaVO.class,"livros");

            constraints = new
                        UnmodifiableFieldsConstraints(staticConstraints);
      }

      public final IFieldsConstraints getConstraints() {
            return constraints;
      }
}
```

Figura 5.1: VO anotado para descrição visual

Analisando a Figura 5.1, podemos verificar que o VO está decorado através da anotação chamada *"ListTabbedPaneRenderizable"*, o que significa que este VO é responsável pela descrição visual da lista resultante de uma consulta realizada em um determinado serviço do sistema. Podemos observar também que a anotação compreende a determinação de dois valores: *"formTitle"* e *"tabSheetID"*. Estes atributos fazem parte dos metadados do serviço, visto que o primeiro corresponde ao título que será apresentado ao usuário, e o segundo corresponde ao identificador da descrição. Veremos nas próximas seções a necessidade da criação desse último atributo.

De maneira análoga à apresentação visual da lista de resultados de uma consulta realizada em algum serviço do sistema, temos também a apresentação visual da própria consulta. A consulta de livros e a de revistas seria apresentada ao usuário da mesma forma, mudando apenas os campos requeridos para consulta, ou seja, o modelo de dados utilizado. Assim, percebemos também a possibilidade de automatização da construção da vista da tela de consulta.

O procedimento de implementação da automatização para a tela de consulta é o mesmo que o explicitado para a tela de lista. Nesse caso, criamos uma nova anotação, chamada *"QueryTabbedPaneRenderizable",* destinada à descrição visual da consulta, marcando o VO para este fim. O engenho responsável por interpretar o VO deverá ser capaz de diferenciar as anotações e dirigir a construção corretamente. Assim, criamos duas classes distintas: uma que descreve o formato da lista; e outra que descreve o formato da consulta. Para isso, criamos uma hierarquia de classes, como podemos visualizar na Figura 5.2, onde temos uma classe chamada "*ViewClass*", que engloba o engenho em comum da construção da vista e as classes filhas, "*Lista**ViewClass*" e "Selecao*ViewClass*", que possuem aspectos específicos de apresentação visual.

O engenho responsável pela interpretação do VO e determinação correta da "*ViewClass*" que será apresentada é um filtro implementado utilizando o arcabouço Chameleon, que será apresentado na seção 5.3.1. O filtro então delega para a "*ViewClass*" a responsabilidade de interpretar os descritores de entidade e atributos do VO, a fim de construir os componentes visuais. Apresentaremos melhor a utilização do filtro neste engenho na seção 5.3.

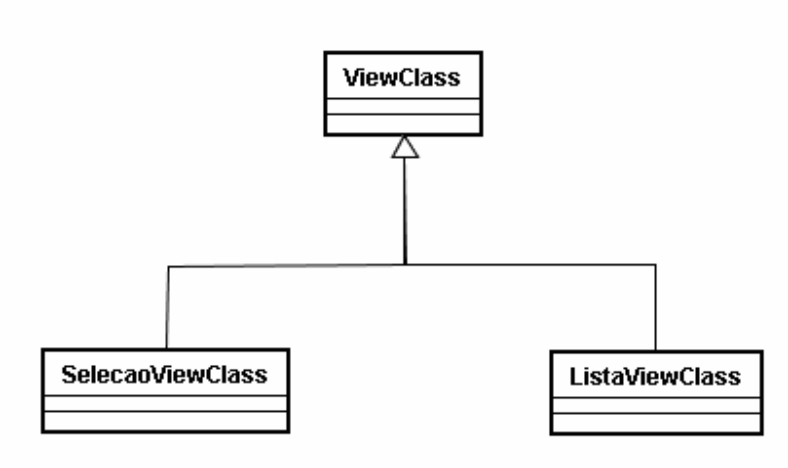

Figura 5.2: Hierarquia das classes de vista

Sobre a biblioteca de entidade e as anotações apresentadas nessa seção, é importante salientar que estes artefatos são independentes do sistema que será construído. As anotações ditam apenas qual tipo de tela será construída e detalhes de apresentação, como título da tela. Neste trabalho, utilizamos o arcabouço Hércules como engenho de geração automática de código, portanto, as anotações são referentes à construção da vista através de abas (Lista e Seleção). Qualquer sistema que utilizasse o Hércules poderia também utilizar tais anotações. Por outro lado, caso o sistema não utilize o Hércules, bastaria que ele utilizasse um engenho similar de interpretação do descritor enriquecido e construção da vista.

5.1.2 Biblioteca de Atributo

A biblioteca de atributos foi organizada em forma de métodos de uma classe utilitária, compondo os tipos elementares de dados. Estes métodos têm parâmetros que permitem refinar a especificação do tipo. O método retorna o tipo

construído, que é a representação interna do que seria especificado por um documento XML.

Esta forma de desenvolvimento é muito usada atualmente em sistemas que seguem a abordagem DDD, acrônimo para “Desenvolvimento Dirigido por Domínio”, do inglês *Domain-Driven Design* (EVANS, 2003). Esta abordagem visa disciplinar o desenvolvimento de software. Reúne conceitos, técnicas e princípios com foco no domínio e na lógica do domínio para criar um modelo de domínio mais semântico, utilizando para isso o padrão de projeto VO. Entre suas vantagens, está o melhor entendimento do desenvolvedor com relação ao negócio, o que evita erros e custos altos em manutenção.

A biblioteca de atributo também é responsável por adicionar significado à descrição da vista. Através de tipos elementares de dados, os atributos do VO são declarados de acordo com suas características de apresentação. Para isso, são criados tipos elementares de dados correspondentes a cada apresentação visual existente no sistema.

A alteração implementada no VO é a implantação de novos membros estáticos, que permitem a criação dos tipos elementares e sua posterior recuperação e construção dos componentes visuais correspondentes. A partir das descrições implementadas no VO, o filtro realiza o processo de interpretação e geração das descrições visuais. Essas irão compor o conjunto de encaixes que será usado para completar o processo de construção da vista.

Vimos na Figura 4.4, apresentada na seção 4.2, um exemplo de tipo elementar, chamado *"LabelDescriptor"*, criado para o sistema "O Livreiro Digital". Este tipo corresponde a campos do tipo texto que são compostos de um título e o resultado da consulta. Assim, no VO que descreve a apresentação da vista, ao invés de existir apenas um atributo *"titulo"* de um tipo genérico (como o *String*, do Java), este atributo será declarado como "*LabelDescriptor*", adicionando significado ao dado. A Figura 5.3 exibe uma proposta de implementação para o VO responsável pela construção da vista da tela de lista do sistema. O exemplo se limitou à descrição dos campos *título* e *editora* para diminuir a extensão do código.

```
public class LivroListItemVO implements Serializable, FieldConstrainable
{

   /* Lista de atributos do vo */
   public String titulo;
   public String editora;

   /* Inicialização dos constraints */
   private static UnmodifiableFieldsConstraints constraints;
   static{

      FieldsConstraints staticConstraints =
            VOConstraintsUtilities.buildRowTableConstraints(
                 LivroListItemVO.class,
                 new VOConstraintsUtilities.ColumnDescriptor[]{
                     new VOConstraintsUtilities.LabelDescriptor("titulo",
                      "Título", 5),
                     new VOConstraintsUtilities.LabelDescriptor("editora",
                      "Editora")
                 }
            );

            constraints = new
                 UnmodifiableFieldsConstraints(staticConstraints);
      }

   public IFieldsConstraints getConstraints() {
      return constraints;
   }
}
```

Figura 5.3: Implementação do VO que descreve a lista do sistema "O Livreiro Digital"

Analisando a Figura 5.3, percebemos o uso de um bloco estático para definir as restrições (*constraints*). O bloco estático é executado antes da execução do construtor do objeto, apenas uma vez. Assim, quando o VO for carregado para a memória, são definidos os tipos elementares de cada atributo contido neste, já com as informações visuais.

Podemos notar que ambas as propriedades, *titulo* e *editora* são descritos através do mesmo tipo elementar "*LabelDescriptor"*. A variação do comportamento deste tipo está ilustrada no último parâmetro de sua declaração, onde o atributo *título* informa o tamanho da fonte que será utilizada na construção do componente visual. Para o atributo *editora*, não foi informado o tamanho da fonte, já que será utilizado o tamanho padrão para este tipo elementar.

O engenho para a definição destas restrições está na classe "*VOConstraintsUtilities*". Através do método "*buildRowTableConstraints*", são informados os tipos elementares de cada atributo, utilizando seus próprios construtores. Este método também é responsável por delegar para cada tipo elementar específico que defina as suas restrições através dos parâmetros passados em seu construtor.

Sobre a biblioteca de atributo e os tipos elementares de dados apresentados nessa seção, é importante salientar que estes artefatos são construídos de acordo com a necessidade do sistema que será construído. Como o foco deste trabalho está na apresentação visual de um sistema, os tipos elementares refletem os tipos de componentes visuais existentes no sistema.

Nesta seção, utilizamos o exemplo do sistema “O Livreiro Digital” e vimos a necessidade da construção de um tipo elementar que chamamos de “*LabelDescriptor*”, e será responsável pela apresentação de uma informação em formato texto. Para outro sistema, esse tipo elementar poderia também ser utilizado no mesmo formato ou em um outro formato, de acordo com o requisito do cliente, portanto existe uma grande possibilidade de os tipos elementares serem distintos para cada sistema. Como no caso da biblioteca de entidade, qualquer sistema que utilizasse o Hércules poderia também utilizar tipos elementares, desde que esses fossem construídos de acordo com os requisitos do sistema em questão. De forma análoga, caso o sistema não utilize o Hércules, bastaria que ele utilizasse um engenho similar de interpretação do descritor enriquecido e construção da vista.

5.2 Adaptações no arcabouço Hércules

O arcabouço sofreu adaptações a fim de possibilitar a construção de vistas padronizadas através de descrições adicionais provenientes do VO. Este arcabouço possui um controlador que é responsável, entre outros, pela montagem da vista. Antes deste trabalho, era possível apenas fornecer a descrição completa da apresentação do caso de uso através de um documento XML.

Analisando a Figura 5.4, podemos entender melhor a arquitetura deste modelo. Quando o usuário faz uma requisição, esta é transferida para o Hércules, que cria o controlador requisitado. Este executa o processo de inicialização obtendo o descritor do caso de uso, que nesse caso é um documento XML. O

controlador Hércules é responsável por interpretar a descrição do caso de uso e construir os componentes visuais do sistema.

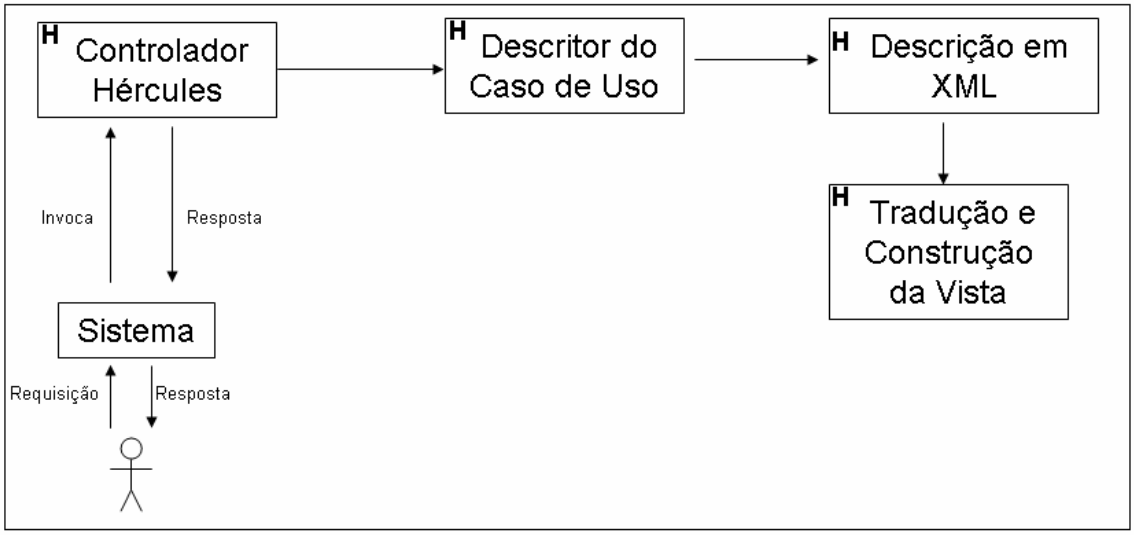

Figura 5.4: Arquitetura original do Hércules

Com a automatização, passou a existir uma extensão da descrição da vista. Assim, existe a descrição obrigatória, implementada através da linguagem XML. Esta pode ser estendida através de uma descrição adicional, opcional, e apenas para os serviços que seguem um padrão de apresentação, implementada utilizando VO. Esta técnica então substitui a descrição que antes era feita no XML pela descrição implementada no VO. Para isso, foi necessária uma modificação na interpretação do descritor em XML. Isto porque tal descritor passa a ter partes "ocas", lacunas que serão preenchidas posteriormente. Essas lacunas são representadas no XML através do atributo "*class*" que terá o valor de "*proxy*". Assim, a especificação em XML foi estendida para que seja possível indicar que a descrição de um determinado componente ainda não é conhecida. O componente

passa a ser descrito apenas por um identificador. Esse identificador, associado ao tipo do componente, indicará qual o encaixe adequado à lacuna.

Com mais de uma fonte para a descrição da vista, foram necessárias algumas modificações também no controlador. Esse verifica se existem as duas fontes e, em caso afirmativo, utiliza um artifício implementado no arcabouço para "colar" as duas descrições. Esta cola utiliza o padrão Visitor (GAMMA, 1999) com a finalidade de percorrer cada componente descrito, encaixando os componentes relacionados. Dessa forma, as lacunas existentes no descritor XML são preenchidas através da descrição adicional.

O descritor da vista utiliza o padrão de projeto Composite (GAMMA, 1999), e possui elementos de meta-descrição. Ele pode ser gerado em tempo de execução ou dinamicamente a partir de descrições em XML. Para a evolução deste trabalho, as classes de meta descrição foram estendidas. Por outro lado, não houve modificação nos componentes visuais já existentes.

A descrição da vista em XML foi estendida para aceitar a existência de lacunas. Tais lacunas foram traduzidas em forma do padrão de projeto "*Proxy*" (GAMMA, 1999). Foi criado o componente "*TabSheetProxy*" que nada mais é que o próprio componente original "*TabSheet*" usado para construir as abas na apresentação do serviço. Dessa forma, preencher as lacunas significa associar a referência ao elemento *proxy* correspondente. Com o *proxy* já conectado, o controlador está preparado para ordenar a criação dos componentes visuais a partir da descrição.

A Figura 5.5 apresenta um exemplo de descrição utilizando o XML com lacunas para o exemplo do sistema “O Livreiro Digital”. Percebemos a presença do atributo “*class*”, indicando que a descrição da vista não será especificada neste documento, ou seja, que o XML contém lacunas. Percebemos também o valor do atributo “*id*”, que deve ser igual ao atributo “*tabSheetID*” da anotação utilizada para marcar o VO como descritor da vista, apresentado na Figura 5.1.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<mvc>
      <view id="dummyId" title="consultaLivro">
            <tabbox id="tabbedMenu">
                  <tabs>
                        <tab label="Lista" selected="false" />
                  </tabs>
                  <tabpanels>
                        <tabpanel id="listaLivro" class="proxy"/>
                  </tabpanels>
            </tabbox>
      </view>
</mvc>
```

Figura 5.5: Exemplo de descrição utilizando a linguagem XML

A fim de concluir a apresentação das alterações sofridas pelo arcabouço Hércules, apresentamos na Figura 5.6 a nova arquitetura, proposta por este trabalho. Como vimos, agora temos dois modelos de descrição da vista do caso de uso. O controlador Hércules continua sendo o responsável pela construção dos componentes visuais, mas agora o controlador é capaz de interpretar uma descrição em XML que contém lacunas. A tradução é feita utilizando um componente “*proxy*” enquanto o componente original não é construído. Quando o Hércules recebe a invocação para construir os demais componentes visuais (descritos através de tipos elementares de dados nas classes VO), o controlador deve ser capaz de encaixar os componentes e, assim, formar a vista final.

Veremos na próxima seção o responsável por invocar o controlador Hércules para a construção dos encaixes visuais.

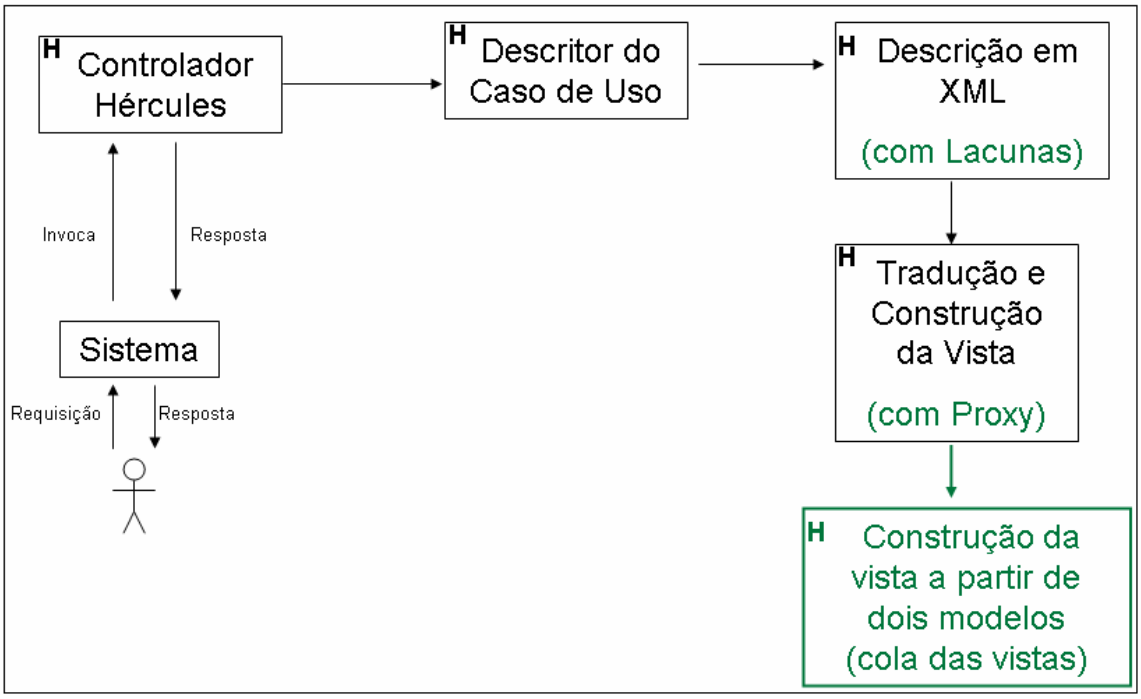

Figura 5.6: Arquitetura de construção da vista após a alteração do Hércules

5.3 Engenho de Construção dos Encaixes da Vista

Apresentamos nesta seção a segunda fase de construção da vista de um sistema. Para isso, utilizamos o arcabouço Chameleon, responsável por interceptar as requisições do usuário e auxiliar na tradução do modelo da vista a fim de construir os encaixes necessários para montar a vista.

5.3.1 Chameleon

É comum encontrarmos sistemas de informação que possuem restrições quanto ao acesso a determinadas funcionalidades e serviços. As aplicações em

geral não são desenvolvidas para um único usuário, mas para um grupo de usuários que, na maioria das vezes, possuem necessidades diferentes. Neste contexto, foi proposto o arcabouço Chameleon (LEMOS, 2005), que provê o controle de acesso à aplicação. Os usuários são divididos em classes, e, para cada classe, são delegados perfis, ou seja, permissões de acesso.

O Chameleon introduziu a idéia de filtros com o objetivo de permitir o controle de acesso de uma forma mais simplificada. Com o uso de filtros, não há a necessidade de implementar o controle de acesso em cada serviço, possibilitando assim que o desenvolvimento de um serviço fique voltado para a codificação das regras de negócio pertinentes ao seu objetivo. Segundo Lemos (2005), uma camada, sobre a qual os serviços são construídos, ficaria responsável por interceptar as requisições, e submetê-las a uma lista de filtros antes de executar o serviço de fato. Ligado a essa camada está um filtro que, baseado nos perfis do usuário, deve permitir ou negar acesso ao serviço requisitado.

O objetivo inicial do projeto era o controle de acesso, mas a arquitetura de filtros foi estendida para funcionar como uma intermediação entre a requisição do usuário e o tratamento da informação circulante. O Chameleon se propõe a funcionar como um Proxy seletivo que se interpõe entre os clientes e os serviços ofertados pelo sistema, injetando funcionalidades ubíquas em todos eles.

Os filtros são interpostos entre um protocolo, sendo o HTTP (W3C, 2009) um padrão mundialmente estabelecido, e o *Servlet* (HUNTER, CRAWFORD, 2003), objeto que atende à requisição do usuário. Assim, todas as requisições,

antes de atingir o objeto *Servlet*, atravessam a cadeia de filtros que estão declarados para o ponto que está sendo invocado. Dessa forma, pode-se agregar valor tanto à requisição quanto à resposta.

Durante a interceptação de uma requisição, cada filtro tem controle total sobre os elementos que a compõem. Assim, quando uma dada ação é invocada, o controle é passado ao arcabouço. Uma cadeia de filtros é montada, e cada filtro tem a chance de inspecionar e modificar tanto a requisição em si, quanto a reposta produzida pela ação concreta. A modificação dos elementos que compõem a requisição se dá pela decoração dos objetos modificados, ou seja, para um filtro modificar a resposta, por exemplo, basta que ele passe adiante o objeto de resposta decorado, de modo que qualquer resposta escrita neste objeto passará por ele antes de atingir o objeto resposta original (LEMOS, 2005).

Naturalmente a lista de filtros que deve fazer parte da cadeia de interceptação deve ser configurada dentro do conjunto de metadados da ação. Desta forma, novos filtros podem ser desenvolvidos e incluídos na execução de qualquer ação. Com essa configuração há também um ganho de desempenho, já que é possível excluir determinados filtros de algumas ações, nos casos em que a interposição dos mesmos seja desnecessária. Esta configuração de metadados de ações é realizada por meio de entidades em bancos de dados relacionais, ou seja, persistida fora do código que implementa a ação em si, aumentando a flexibilidade, uma vez que modificações nos metadados podem ser feitas através da própria aplicação.

5.3.2 Utilização do Chameleon na construção da vista

Vimos na seção anterior que o arcabouço Chameleon provê um esquema, composto de uma camada de filtros, capaz de interceptar a requisição do usuário ao sistema com o objetivo de modificar a resposta, adicionando funcionalidades ou tomando alguma decisão. Apresentamos na Figura 5.7, a arquitetura de um sistema que utiliza os arcabouços Chameleon e Hércules. O processo de construção do caso de uso continua o mesmo para o Hércules, mas agora, através de uma camada de filtros cadastrados na ação inicial do serviço requisitado, o arcabouço Chameleon é capaz de se comunicar com o Hércules, modificando a resposta ao usuário, adicionando funcionalidades, por exemplo.

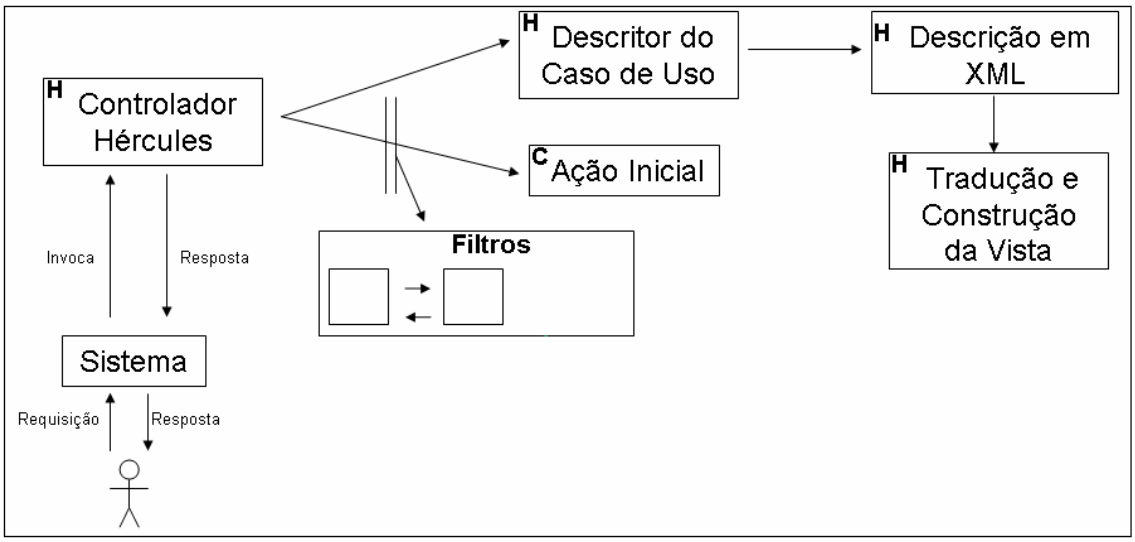

Figura 5.7: Arquitetura original de construção do sistema utilizando o Chameleon

O arcabouço Chameleon foi utilizado como engenho para tradução da descrição adicional em componentes de vista reconhecidos pelo Hércules. Assim, como vimos na seção anterior, o Hércules apenas interpreta a descrição realizada

na linguagem XML e gera um conjunto parcial de componentes que formarão a vista. A participação do Chameleon na construção dos encaixes da vista é dividida em duas etapas: a primeira compreende a criação das telas; e na segunda, o Chameleon invoca o engenho do Hércules criado para interpretar a descrição adicional e construir os componentes visuais que irão compor o serviço. Assim, inicialmente, o Chameleon é o responsável por interpretar o VO em busca do decorador de entidade (anotação). Se o VO estiver anotado, o Chameleon constrói os chamados elementos "*proxy*", referentes às "*viewClasses*", ou seja, constrói a aba de lista ou seleção do serviço, dependendo da anotação utilizada no VO. Este esquema simplificado está ilustrado na Figura 5.8.

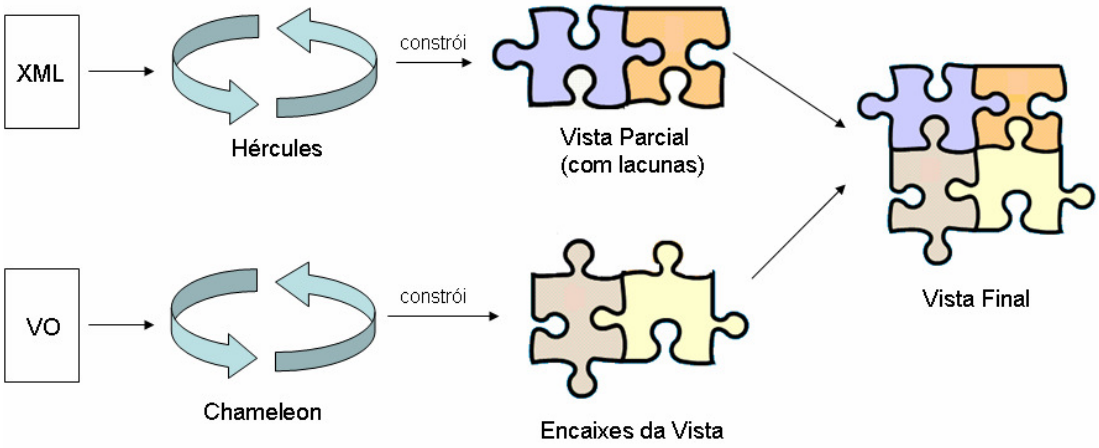

Figura 5.8: Esquema simplificado da construção da vista do sistema utilizando o mecanismo

Para a construção de partes da vista de maneira dinâmica, foi criado um filtro. O arcabouço Chameleon estrutura os serviços de um sistema em forma de ações. Assim, os serviços devem disparar uma ação inicial que é responsável pela criação das classes VO. A função do filtro é interceptar a resposta da ação e criar

os elementos que preencherão as lacunas da vista, a partir dos descritores adicionados as estas classes VO.

Para interceptar uma requisição, uma Cadeia de Responsabilidade (*Chain of Responsability*) (GAMMA, 1999) é organizada logo no início da execução do serviço. Nesta cadeia estão presentes filtros que podem implementar as mais diversas funcionalidades. Uma destas funcionalidades, como visto, foi o controle de acesso implementado pelo arcabouço Chameleon.

O objetivo do filtro é verificar, na resposta da ação inicial, qual VO possui as informações necessárias para a construção da vista. O filtro fica então responsável por construir as interfaces visuais. Quando um serviço é implementado, deve possuir uma descrição para ação inicial, esta ação contém todas as classes VO que serão usadas pelo serviço. Para que a automatização funcione, é necessário cadastrar o filtro para esta ação inicial, o que significa cadastrar metadados para as ações de cada serviço.

A utilização da anotação apresentada na seção 5.1.1 para marcar o VO possibilita a ação do filtro. Assim, o VO fica anotado com informações que servirão de validação durante a construção da vista. Esta foi a forma encontrada para relacionar a parte da vista descrita no VO com a outra parte relacionada, descrita no XML. Com isso, foi extinto o risco de inconsistência, já que, utilizando anotação, foi criado um atributo para o VO que nada mais é que um identificador da aba a ser construída. Por isso, conforme observamos na seção 5.2, o valor deste atributo deve ser o mesmo indicado no XML que contém a outra parte da

vista a ser construída. Assim, como podemos verificar na Figura 5.9:, o filtro valida que não há mais de um VO com mesmo identificador na resposta da ação e identifica o VO que possui o mesmo identificador que está especificado no XML. A anotação é usada no VO para definir a "*viewClass*" (aba) que será construída e as partes customizáveis da vista, como o título que aparecerá ao usuário. Como a automatização foi realizada para as abas de seleção e lista, existe uma anotação diferente para cada aba. A vista será construída pelo filtro de acordo com a anotação que o VO implementa.

```
private void verificaVODescricaoLista(Collection vosViewClasses, Map
tabSheetsIDMap, Object vo) throws ActionException {

      ListTabbedPaneRenderizable voLista = null;
      try{
         voLista = vo.getClass().
                   getAnnotation(ListTabbedPaneRenderizable.class);
      }catch(ClassCastException e){
      //não faz nada... o VO pode não ser destinado à descrição da lista
      }

      if(voLista != null){
            if((tabSheetsIDMap.get(voLista.tabSheetID())) != null){
                  throw new ActionException(
                        "Erro ao construir a descrição da vista: não é
                        permitido mais de um VO com mesmo ID.");
            }
            tabSheetsIDMap.put(voLista.tabSheetID(), voLista);

            try{
               ListaViewClass listaViewClass = new ListaViewClass(
                                                voLista.tabSheetID(),
                                                voLista.tabSheetTitle(),
                                                voLista.formTitle());
               listaViewClass.setVO(vo);
               vosViewClasses.add(listaViewClass);
            } catch (BuilderViewException e) {
               throw new ActionException(e.getMessage(), e);
            }
      }
}
```

Figura 5.9:Trecho da implementação do filtro

Neste trecho apresentado na Figura 5.9, o filtro verifica se o VO que está no contexto é destinado à descrição da tela de lista. Para isso, o VO deve implementar a anotação chamada de "*ListTabbedPaneRenderizable*", como se pode ver no código. Assim, se o VO for determinado para este fim, é criada a classe responsável pela construção da aba de lista, chamada de "*ListaViewClass*", e esta recebe como parâmetros as informações que estão contidas no VO através da anotação. Como se pode ver, são três os parâmetros passados, "*tabsheetID*" é aquele que identifica a lacuna que será preenchida no descritor XML e os demais são aqueles que aparecerão na vista, como títulos.

Além de construir a aba de lista e de seleção, o filtro é também responsável por invocar o novo engenho do Hércules para que os demais componentes visuais sejam construídos (a partir da tradução dos metadados definidos através dos tipos elementares para cada atributo do VO) e encaixados nos componentes previamente construídos (a partir da descrição em XML). A Figura 5.10 ilustra essa nova arquitetura proposta.

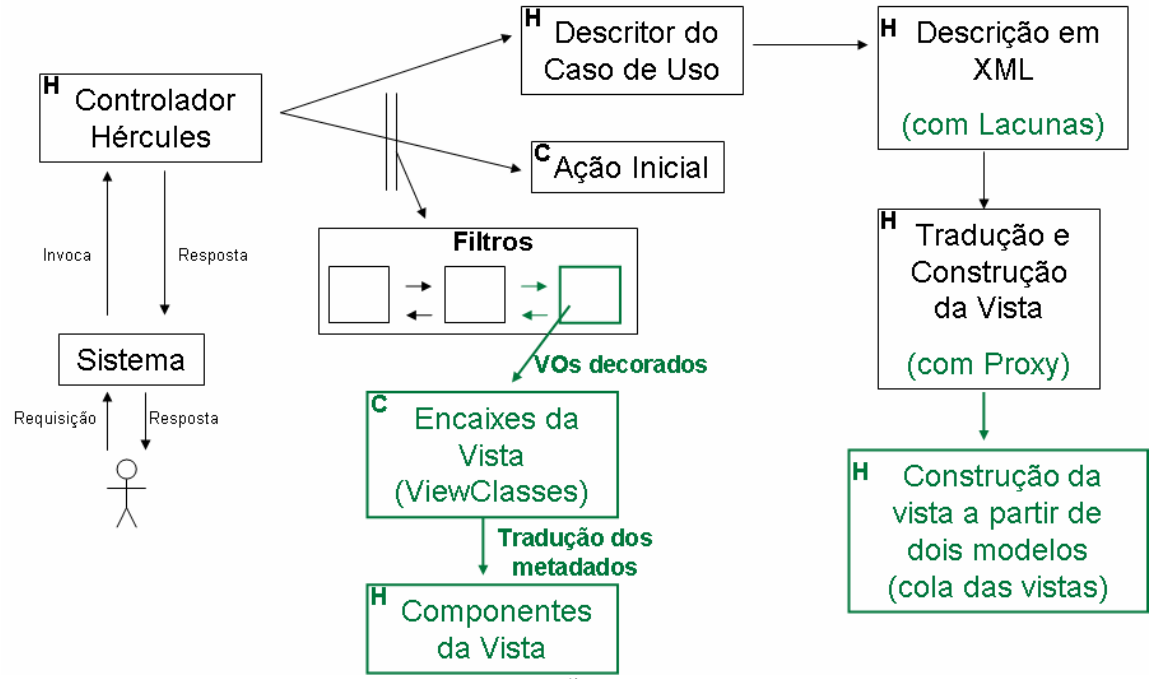

Figura 5.10: Representação da arquitetura proposta

5.4 Mecanismo

Neste capítulo, vimos que o mecanismo proposto para construção de interfaces de sistemas de informação engloba três fatores principais: o descritor enriquecido; a construção da descrição em XML contendo lacunas; e a utilização do engenho criado através do arcabouço Chameleon para a construção dos componentes que serão encaixados na vista. Nesta seção, apresentaremos a consolidação dos passos necessários para a utilização do mecanismo proposto por este trabalho.

Em suma, para a construção de um serviço em um sistema de informação, são necessários os seguintes passos:

- Criação do documento XML com lacunas, conforme apresentado na seção 5.2.

- Criação do VO responsável pela descrição da vista e utilização da anotação para decorar o VO de acordo com a tela que será responsável por construir, conforme apresentado na seção 5.1.1.

- Criação do VO que contém os atributos que serão exibidos na vista. Esse VO deve herdar da interface "*FieldConstrainable*" para associar seus atributos aos tipos elementares através do construtor estático. Isso é feito utilizando o engenho construído para a definição de tipos, conforme apresentamos na seção 5.1.2.

- Cadastro do filtro do Chameleon no engenho de ações responsáveis pela execução do serviço. Conforme apresentamos na seção 5.3, o filtro verifica a descrição enriquecida do VO e constrói os componentes que serão encaixados nas respectivas lacunas do XML.

Para o exemplo do sistema "O Livreiro Digital", o VO utilizado para apresentar a vista é descrito utilizando o descritor enriquecido apresentado na seção 5.1. Como neste exemplo apresentamos os possíveis resultados de uma consulta realizada pelo usuário, foi necessária a criação de duas classes VO: um VO que engloba todos os itens resultantes; e outro que apresenta os atributos de cada item. Assim, o primeiro VO utilizou a biblioteca de entidade para ser marcado como a classe que descreve a tela de lista de livros; e o segundo VO utilizou a biblioteca de atributos, através dos tipos elementares de dados, para descrever

como os itens dos livros serão apresentados na tela. Vimos que a tela de apresentação do livro e a tela de impressão apresentada ao usuário são idênticas com relação à apresentação dos itens do livro. Este trabalho provê a reutilização das descrições, logo, não será necessário descrever a tela de impressão, basta utilizar o mesmo VO criado anteriormente.

5.5 Considerações Finais

Neste capítulo, apresentamos o mecanismo de construção de interfaces e a implementação inicialmente proposta para a aplicação do mecanismo em um sistema de informação. Para isso, explicitamos a criação de uma biblioteca de decoradores e outra de descritores, que serão utilizadas para descrever o VO e assim adicionar maior significado à classe. O mecanismo proposto por este trabalho implicou na extensão do arcabouço Hércules, adicionando um novo engenho para a geração automática da vista de casos de uso. Caso o sistema utilize o arcabouço Chameleon para intermediar a informação que é passada pelo Hércules, criamos um filtro responsável pela interpretação dessa informação e construção da vista adicional para o Hércules. No próximo capítulo apresentaremos o sistema utilizado em nosso exemplo de aplicação como prova de conceito do mecanismo proposto.

6. Exemplo de Aplicação

Os arcabouços Hércules e Chameleon, citados nas seções 3.1 e 6.3.1, respectivamente, são utilizados na construção do Sistema Integrado de gestão acadêmica da Universidade Federal do Rio de Janeiro – SIGA/UFRJ, aqui apresentado como nosso exemplo de aplicação. Está capítulo foi dividido em duas seções principais: a primeira é composta da apresentação do SIGA; e a segunda apresenta os detalhes de utilização do SIGA como exemplo de aplicação do mecanismo proposto por este trabalho.

6.1 Sistema Integrado de Gestão Acadêmica

O Sistema Integrado de Gestão Acadêmica da Universidade Federal do Rio de Janeiro – SIGA/UFRJ é atualmente acessado por mais de 60.000 usuários, dentre eles, alunos, professores e secretarias, ou seja, toda a comunidade acadêmica. Trata-se de um sistema de grande porte que agrega todos os serviços de registro acadêmico da UFRJ, disponibilizando serviços on-line aos seus usuários de acordo com os perfis estabelecidos para cada um. Ao estudante é permitido, por exemplo, inscrição em disciplinas e acesso ao boletim e histórico escolar. Para o corpo docente, o sistema possibilita, por exemplo, o lançamento de graus e freqüências. Para as secretarias acadêmicas, é possível o lançamento e alteração de graus, trancamento de matrícula, dentre outros.

Além dos serviços da Divisão de Registro de Estudantes (DRE), existem outras seções que nunca tiveram apoio de um sistema informatizado e estão

sendo acrescentadas ao SIGA. O sistema se expande a outros departamentos da Pró-reitoria de Graduação e da Pró-reitoria de Pós-Graduação e Pesquisa, como controle de diplomas, pagamento de bolsas, controle de alojamento e cadastro dos cursos de extensão.

O SIGA está em desenvolvimento ao longo de alguns anos, tendo iniciado o processo em 2000 e estando em produção desde 2001. Por ser um sistema de grande porte, que atende a um número elevado de usuários (toda a comunidade acadêmica da UFRJ), o SIGA é um sistema que está em contínuo desenvolvimento e evolução. Isto porque as regras de negócio sofrem constantes modificações requisitadas diretamente pelos Conselhos Superiores: Conselho de Ensino de Graduação (CEG), no caso dos cursos de graduação, e Conselho de Ensino para Graduados (CEPEG), para os programas de Mestrado e Doutorado da Universidade. Além disso, há uma grande requisição de novos serviços e de acertos por parte dos próprios usuários, como secretarias acadêmicas e coordenadores de curso.

Nas próximas seções, apresentaremos o processo de desenvolvimento utilizado, além dos padrões impostos, justificando a adaptação do sistema ao mecanismo proposto. Por último, apresentaremos a arquitetura de interação do sistema com os arcabouços descritos.

6.1.1 Processo de Desenvolvimento

O SIGA é desenvolvido inteiramente em formato *web*, na plataforma Java J2EE (*Java 2 Entreprise Edition*), cujo principal diferencial é a ênfase dada à

utilização de padrões de projeto (*Design Patterns*). A plataforma Java J2EE surgiu com o objetivo de padronizar e simplificar a criação de aplicações empresariais. Para isso, propõe um modelo, onde componentes J2EE (páginas JSP, Servles, EJB´s, etc) escritos pelos usuários da plataforma, podem fazer uso de serviços providos por esta, os quais simplificam sua implementação e possibilitam maior foco no negócio (SINGH, 2002).

O uso de padrões de projeto traz inúmeras vantagens na modelagem e implementação de um software. Dentre estas, a possibilidade de projetar soluções mais rapidamente e com qualidade já que os padrões são soluções comprovadamente eficientes para problemas já conhecidos. Além disso, visa principalmente flexibilidade, organização e reaproveitamento de código, o que resulta em maior produtividade, qualidade e facilidade de manutenção das aplicações assim desenvolvidas.

Por ser um sistema de âmbito acadêmico, e que sofre contínua evolução, a equipe de desenvolvimento se modifica constantemente. Em vista disso, o SIGA utiliza XP (*eXtreme Programming*) (BECK, 2004) como processo de desenvolvimento. Assim, a equipe é composta por aproximadamente 20 (vinte) desenvolvedores e 2 (dois) gerentes de projeto.

O XP possibilita maior disseminação de informação entre os integrantes da equipe, já que enfatiza a comunicação, realimentação e simplicidade, advogando práticas como a posse coletiva do código. Além disso, é dada ênfase também ao desenvolvimento guiado por testes automatizados. Isto provê o entendimento do

maior número possível de integrantes para qualquer parte do sistema, impedindo a criação de “ilhas de conhecimento”.

6.1.2 Padrão Operacional

Serviços que possuem finalidades diferentes podem apresentar o mesmo padrão de funcionamento. O uso de um padrão operacional em um sistema é bastante benéfico, visto que os usuários devem sentir-se confortáveis e seguros diante da interface do sistema. Um sistema deve possuir uma interface com o usuário que seja consistente e intuitiva, permitindo assim que o usuário crie um modelo mental da forma como o sistema se comporta, o que reduz o custo de treinamento e suporte. O padrão operacional do SIGA impõe um roteiro de caso de uso composto de um conjunto de operações básicas: consultar, editar, inserir, alterar, excluir e limpar.

A utilização de um padrão operacional é imprescindível para o SIGA, não só pela grande quantidade de serviços disponíveis, mas também devido ao número elevado de usuários. Isto porque o acesso às funcionalidades do sistema é ditado pelas permissões que cada usuário possui. Isto quer dizer que o acesso ao sistema é liberado para o usuário de uma forma limitada de acordo com o seu perfil. Assim, um determinado usuário, que teve seu acesso expandido, não vai demorar a se adaptar às novas funcionalidades que lhe foram oferecidas. Por exemplo, um professor possui a permissão de lançar grau para determinada turma ministrada por este; caso este professor se torne coordenador de curso, este poderá lançar grau para qualquer turma ministrada pelo curso coordenado.

A criação dos serviços no SIGA segue padrões de implementação e interface com o usuário. A apresentação do serviço é dividida em abas que representam o estado do caso de uso. A maioria dos serviços segue este padrão de operação:

- O usuário realiza uma consulta (aba “Seleção”);
- São listados os possíveis resultados decorrentes da consulta (aba “Lista”);
- O usuário seleciona um dos itens apresentados;
- Toda a informação relacionada ao item selecionado é apresentada ao usuário, que pode modificá-la (aba “Edição”);

Caso exista alguma mensagem de aviso ou erro, esta é apresentada na aba “Mensagem”; e, caso exista alguma ajuda cadastrada para o serviço, será apresentada na aba “Ajuda”. A Figura 6.1 apresenta um exemplo de serviço do sistema SIGA.

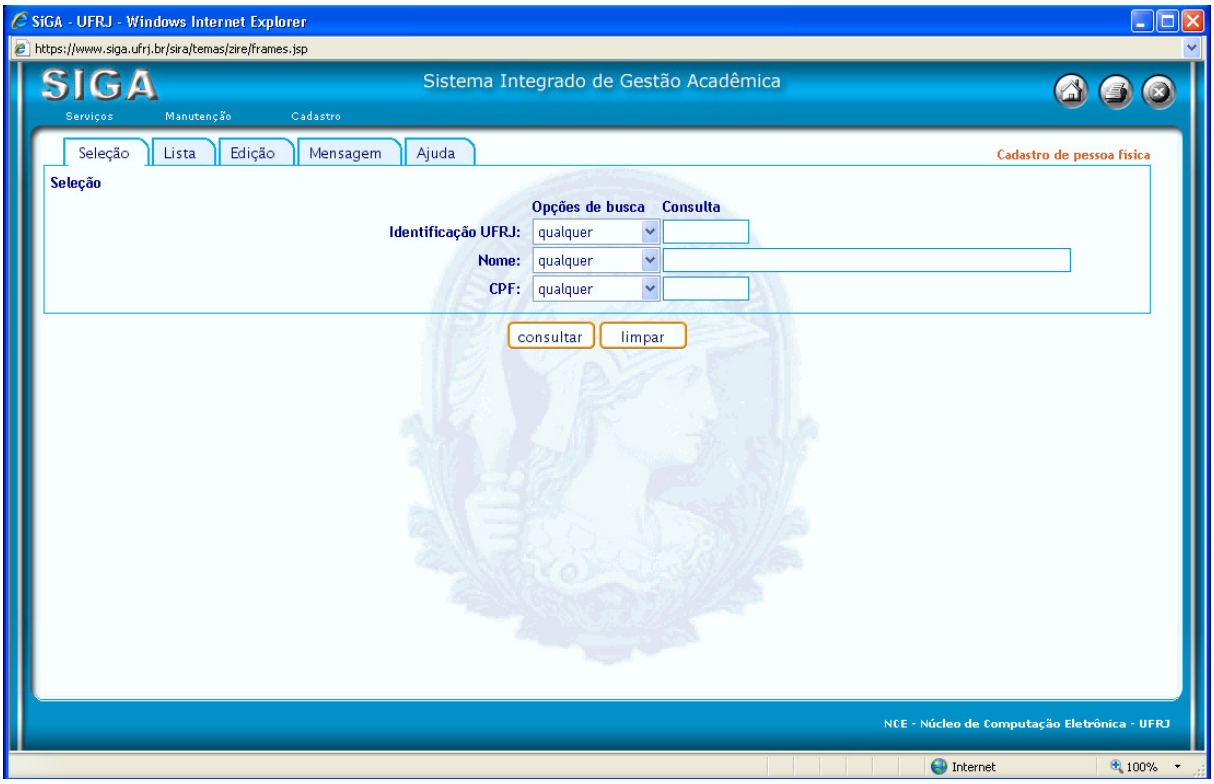

Figura 6.1: Apresentação do sistema SIGA

6.1.3 Padrão de Interface

Serviços que possuem o mesmo padrão de operação devem possuir o mesmo padrão de interface gráfica. As abas de seleção e lista têm sempre o mesmo padrão de disposição de elementos, indicando ao usuário que o comportamento dessas abas será o mesmo em todos os serviços nos quais elas são disponibilizadas. O usuário sabe que essas abas irão se comportar da mesma maneira em qualquer serviço. É importante que essas telas tenham rigorosamente a mesma forma gráfica para que o usuário não tenha a menor dúvida. A ausência de um título, por exemplo, pode deixar o usuário inseguro e com dúvidas.

O exemplo do SIGA mostra como o problema da variabilidade afeta sistemas grandes. A profusão de serviços muito semelhantes induz à criação de diversas variantes de um mesmo caso de uso, poluindo o código com trechos semelhantes. Estas semelhanças, quando percebidas pelo desenvolvedor, sugerem que o trabalho possa ser reduzido através da cópia e edição do diferencial relevante. Apesar de poupar trabalho, esta prática evidencia uma fragilidade arquitetural da solução que não provê uma proposta em um nível mais alto.

A solução *ad hoc* dos desenvolvedores enuncia que existe uma semelhança no código que pode ser explorada. A solução de cópia, apesar de prover uma solução funcional imediata, pode não ser eficiente ao longo prazo. À medida que o sistema evolui, o código comum pode sofrer modificações que irão afetar todas as partes onde estiver copiado. Isto exigiria uma varredura criteriosa de todo o código fonte para corrigir todas as versões. A tarefa de localizar estas versões é difícil de ser automatizada, pois não se sabe quais partes foram variadas para se construir uma expressão regular de busca. Isto pode resultar num alto custo de revisão manual de todo código. Ademais, numa busca manual, códigos podem escapar à inspeção visual, resultando numa perigosa fonte de defeitos.

A variabilidade, neste caso, demanda por uma solução em nível mais alto. Esta solução deve ser capaz de representar padrões de uso comum que sejam identificados pelos desenvolvedores. Os desenvolvedores poderão então capturar as semelhanças nestes padrões, que formarão o corpo de todas as variantes.

Estes padrões devem aceitar uma forma de especialização que permita registrar a variabilidade necessária para cada caso específico. O registro dos padrões comuns e suas variantes devem ser feitos de uma forma centralizada. Com isto, na ocorrência de uma evolução no código, bastará editar este registro para que todas as variantes fiquem em conformidade. Uma solução como esta reduziria a margem de erros acidentais e proveria uma melhor compreensão do sistema como um todo.

6.1.4 Utilização dos arcabouços no SIGA

O SIGA é desenvolvido baseado nos dois arcabouços explicitados nas seções 3.1 e 6.3.1. Dessa forma, o Hércules resolve o problema da padronização de caso de uso, ou seja, o código relativo à vista é separado de qualquer tipo de validação ou regra de negócio. Por outro lado, o Chameleon fornece um engenho de filtros que funciona como um interceptador, tratando as informações de acordo com a necessidade da requisição feita pelo usuário.

Apesar de tais benefícios, os arcabouços não resolvem por completo o problema. Isto porque o que ocorre é a prática por parte dos desenvolvedores de geração de código duplicado. A existência de serviços muito semelhantes torna o desenvolvimento uma prática de copiar implementações já existentes e alterar as partes onde os serviços se distinguem. A Figura 6.2 apresenta um esquema da arquitetura utilizada no SIGA.

Como exemplo de ocorrência de duplicação de código, podemos citar os serviços de Histórico Escolar, onde são consultados os boletins dos alunos da

universidade. Estes são serviços semelhantes que apresentam classes de usuário diferentes. Assim, a disponibilidade de funcionalidades do serviço de Histórico varia de acordo com a classe de usuário que o requisita. Desta forma, os serviços terão a mesma descrição da vista, sendo uma adaptada para a classe de usuário correspondente, ou seja, com descrições adicionais à outra. O que ocorre nesse caso é a duplicação de código entre serviços semelhantes.

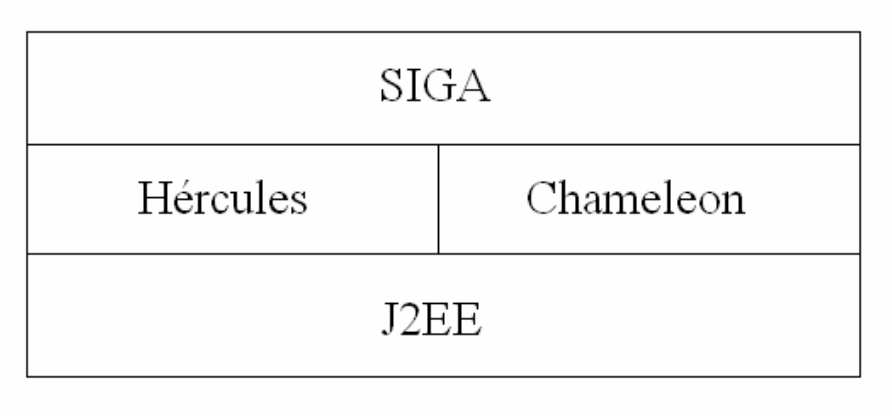

Figura 6.2: Arquitetura do SIGA

6.2 Aplicação do mecanismo no SIGA

O exemplo de aplicação foi baseado nos casos de uso do Sistema de Gestão Acadêmica (SIGA), apresentado na Seção 6.1 deste trabalho. O mecanismo de construção da vista aqui proposto foi aplicado inicialmente nos serviços correspondentes à consulta de Histórico escolar, já que são serviços de menor complexidade. Um estudo realizado no sistema constatou que as abas de seleção e lista seguem um padrão visual em todos os serviços. Assim, começamos o trabalho de automatização para a aba de lista e, quando finalizada, a automatização foi estendida também à aba de seleção.

O serviço de Histórico, quando acessado por um aluno, possui apenas uma aba, aquela referente à lista de matrículas do aluno. Porém, tal serviço, quando acessado por certas classes de usuário, possui maior funcionalidade, apresentando também a aba de consulta ao usuário. Assim, este usuário tem acesso ao Histórico de qualquer aluno.

Percebe-se, então, que os serviços de Histórico são exemplos de serviços semelhantes que apresentam diferença de acordo com a classe de usuário. No caso de um perfil com maior privilégio de acesso, além da aba de lista, será apresentada também a aba de seleção. Porém, a aba de lista é apresentada da mesma maneira para qualquer perfil. Observamos na seção 3.1 que, com a utilização do arcabouço Hércules, as interfaces dos serviços são descritas em arquivos XML. Com isso, o que acontecia era a duplicação de código no que dizia respeito à descrição da aba de lista. Com a automatização, tal descrição é feita apenas uma vez no VO e utilizada para ambos os serviços.

6.2.1 Implementação da Proposta

Vimos na seção 6.1.3, que o padrão visual do SIGA impõe um padrão rígido para a disposição dos elementos das suas abas. Como podemos verificar na Figura 6.3, a aba de lista, por exemplo, sempre exibe uma coleção de elementos, que correspondem a instâncias de uma determinada entidade. A coleção é sempre organizada na forma de uma tabela. As colunas dessa tabela são formatadas conforme os atributos dessa mesma entidade. Além disso, um desses

atributos é exibido como link, em todas as linhas da tabela, enquanto os demais elementos são exibidos como texto simples.

O exemplo da Figura 6.3 demonstra que é possível gerar automaticamente a descrição da aba de lista a partir da entidade exibida na tela. Bastaria apenas identificar na entidade quais atributos devem ser exibidos, em que ordem, quais títulos devem ser exibidos nas colunas da tabela, e qual atributo deverá ser apresentado como link. Além disso, este trabalho propõe a criação de notações, utilizando tipos elementares de dados, que quando adicionadas às entidades, forneçam a informação necessária à geração da descrição de componentes visuais.

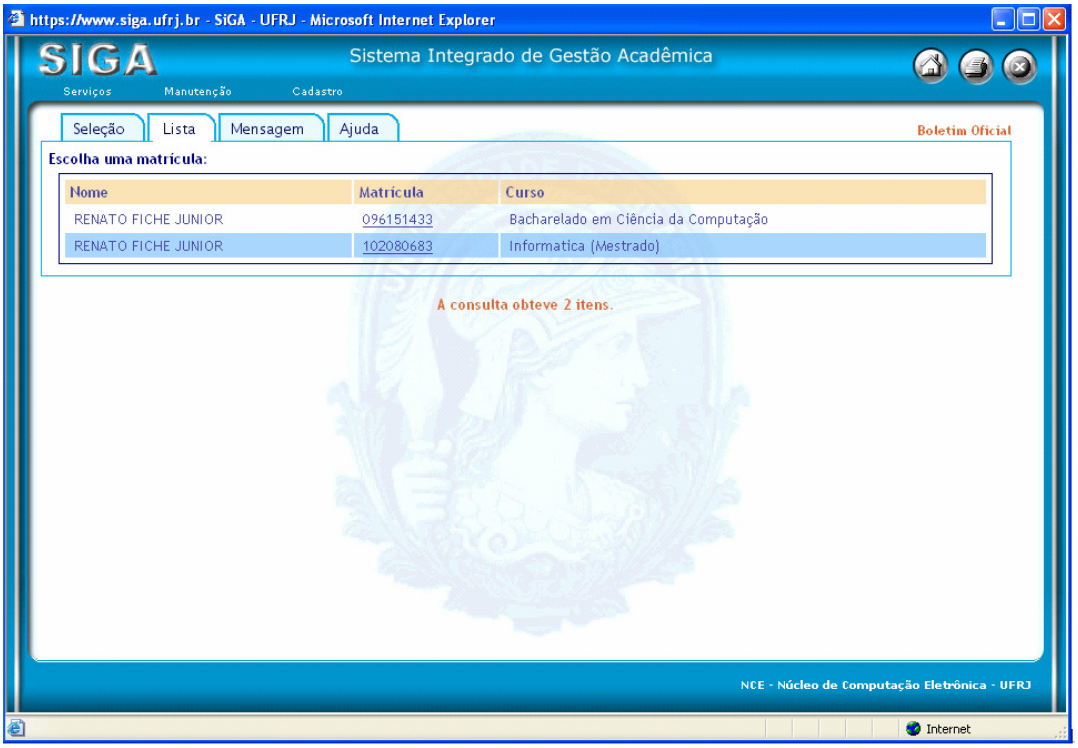

Figura 6.3: Apresentação da aba de lista do serviço de histórico escolar

Para possibilitar a automatização da geração da vista, foram levantados os requisitos de componentes visuais de interação com o usuário e, para cada componente, foi criado um tipo elementar. Cada tipo possui um comportamento particular e foi criado de acordo com o padrão do sistema, ou seja, a forma em que é apresentado no sistema. Porém, se houver necessidade de um novo comportamento no sistema para determinado tipo, este pode ser estendido, de maneira análoga ao exemplo apresentado na seção 4.2.

Para esclarecer melhor este comportamento variável, basta visualizar o exemplo representado na Figura 6.4. No serviço de Histórico Escolar para o acesso com maior privilégio, é exibida uma aba de consulta com campos cujos valores serão informados pelo usuário. Por exemplo, os campos matrícula e nome. Ambos são apresentados como uma composição de um texto com o nome do campo e uma caixa de texto, onde será informado o valor para a consulta. Portanto, ambos implementam o mesmo tipo elementar, mas com o comportamento variável de apresentação. Tal diferença será informada através do parâmetro para o tipo elementar, como pode ser observado na Figura 6.4. O primeiro parâmetro corresponde à propriedade do VO, o segundo corresponde ao texto e o terceiro corresponde ao tamanho da caixa de texto.

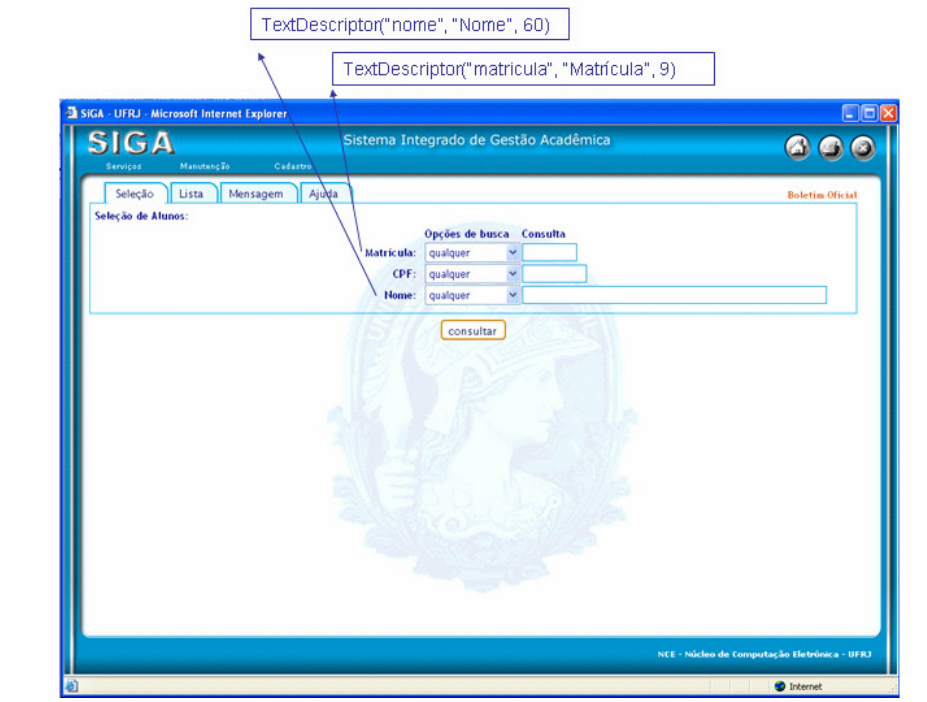

Figura 6.4: Comportamento variável do tipo elementar

O sistema SIGA utiliza o arcabouço Hércules como engenho de descrição da vista, assim esta é definida utilizando a linguagem XML. Com a implementação do mecanismo proposto por este trabalho, o descritor enriquecido redefine partes da descrição gráfica que são agora omitidas na especificação declarativa. A engenharia utilizada para possibilitar esta substituição na especificação da vista, apresentada no capítulo 6, é baseada no uso de anotações e tipos elementares de dados.

Vimos na seção 6.3 que a apresentação visual dos serviços do SIGA é disposta em abas. Um estudo realizado no sistema constatou que as abas de seleção e lista seguem um padrão visual em todos os serviços. Dessa forma, este trabalho foi aplicado na descrição visual dessas duas abas. Para isso, foram

criados dois tipos de anotação: uma destinada a decorar o VO responsável pela descrição visual da aba de lista, e outra destinada à descrição visual da aba de seleção. Dessa forma, a anotação é responsável por identificar o tipo de aba que o VO irá descrever. A Figura 6.5 apresenta um exemplo de um VO que descreve a vista da aba de lista de um serviço do SIGA.

Analisando a Figura 6.5, podemos verificar que o VO está decorado através da anotação chamada *"ListTabbedPaneRenderizable"*. Vimos o uso desta anotação no capítulo 5 com a aplicação no sistema "O Livreiro Digital". De forma análoga, significa que este VO é responsável pela descrição da aba de lista de um determinado serviço do sistema. Apresentamos também, no capítulo 6, a criação de um filtro que será o responsável por interpretar a descrição implementada no VO e dirigir a construção correta dos componentes visuais. O sistema SIGA também utiliza esse esquema de filtro.

```
@ListTabbedPaneRenderizable(
            formTitle = "Lista",
            tabSheetID = "Lista")
public class ListaVO implements Serializable, FieldConstrainable {

      public Collection itens = new LinkedList();
      private static UnmodifiableFieldsConstraints constraints;

      static{

            FieldsConstraints staticConstraints = VOConstraintsUtilities.
                        buildTableConstraints(ListaVO.class,"itens");

            constraints = new
            UnmodifiableFieldsConstraints(staticConstraints);
      }

      public final IFieldsConstraints getConstraints() {
            return constraints;
      }
}
```

Figura 6.5: VO anotado para descrição visual da aba de lista

6.2.2 Construção da aba de lista

Apresentamos na Figura 6.6 a aba de lista do serviço de histórico escolar do sistema SIGA. Trata-se de uma tabela que contém os resultados da consulta realizada após a requisição do serviço pelo usuário. No caso de usuários com maior privilégio de acesso, a consulta é realizada na aba de seleção, de acordo com os dados fornecidos; já no caso de alunos, não existe a aba de seleção, a consulta é realizada na requisição do serviço, pois o aluno só pode visualizar o seu próprio histórico.

A Figura 6.6 mostra que a tabela pode ser composta de vários itens. Quando o serviço é requisitado através da aba de seleção, é possível consultar o histórico de mais de um aluno, por exemplo, históricos de alunos do curso de Informática (Mestrado). Além disso, quando o serviço é requisitado pelo próprio aluno, também é possível que haja mais de um histórico como resultado, já que a pessoa pode ter mais de uma matrícula na universidade. Um exemplo desse caso é apresentado pela Figura 6.6, onde a aluna possui a matrícula do curso de Bacharelado em Ciência da Computação (já concluído) e a matrícula do curso de mestrado em Informática. Dessa forma, para a construção da aba de lista, é necessária uma classe que contém um atributo do tipo coleção que irá englobar os itens resultantes da consulta.

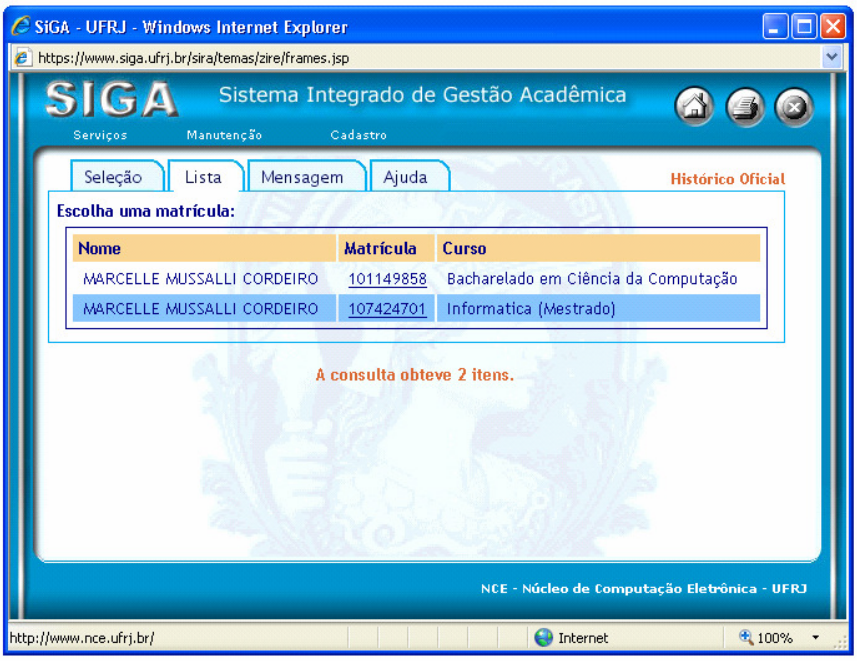

Figura 6.6: Apresentação da aba de lista no serviço de histórico escolar para usuário com maior privilégio

Conforme já observado, utilizamos o padrão de projeto *Value Object* para representar nossas entidades. Antes da implementação do mecanismo aqui proposto, utilizando a declaração estática, o VO continha apenas o atributo citado, sem qualquer significado embutido, conforme podemos visualizar na Figura 6.7.

```
public class AlunoListVO implements Serializable {

      public Collection listItensVO;
}
```

Figura 6.7: VO que englobará os itens a serem apresentados na lista

Após a automatização, o VO, representado na Figura 6.8, é preenchido de significado, sendo decorado para uma determinada função no sistema. O mecanismo de preenchimento da aba de lista é o mesmo para todo o sistema,

assim, transformamos esse VO em uma classe utilitária, que fará parte da construção da interface de todos os serviços do SIGA. Analisando a Figura 6.8, percebemos o uso de anotação, novidade do Java 5. Como vimos na seção 5.1, o objetivo é marcar o VO como uma classe utilizada para a construção da aba de lista, além de possibilitar a descrição da vista através dos atributos *"formTitle"* e *"tabSheetID"*.

```
@ListTabbedPaneRenderizable(
            formTitle = "Lista",
            tabSheetID = "Lista")
public class ListaVO implements Serializable, FieldConstrainable {
      public Collection itens = new LinkedList();
      private static UnmodifiableFieldsConstraints constraints;
      static{
            FieldsConstraints staticConstraints = VOConstraintsUtilities.
                        buildTableConstraints(ListaVO.class,"itens");
      constraints=new UnmodifiableFieldsConstraints(staticConstraints);
      }
      public final IFieldsConstraints getConstraints() {
            return constraints;
      }
}
```

Figura 6.8: VO utilizado para descrever a aba de lista

Percebe-se, na Figura 6.8, o uso de uma engenharia chamada de *constraints*, disponível no arcabouço Hércules. Esta foi necessária para que o atributo fosse marcado para determinado fim, ou seja, o atributo que possui os itens a serem apresentados na aba de lista. Além disso, foi criada uma classe utilitária para facilitar o engenho, que, neste caso, faz uso do método específico para aba de lista.

Vimos na seção 5.1 que o atributo *"formTitle"* define o título que irá aparecer na aba apresentada ao usuário. Utilizando o VO utilitário na descrição da aba de lista do serviço de histórico teria "Lista" como título. Para tornar o serviço

mais descritivo, podemos criar um novo VO que apenas herda do utilitário, alterando o valor do atributo que define o título. Um exemplo disso é ilustrado na Figura 6.9, onde foram alterados os valores do título e do identificador informados no documento XML.

```
@ListTabbedPaneRenderizable(
            formTitle = "Escolha uma matrícula: ",
            tabSheetID = "AlunoLista")
public class AlunoListVO extends ListaVO<AlunoListItemVO> {}
```

Figura 6.9: VO utilizado para descrever a aba de lista de forma mais descritiva

Além do VO que contém a coleção de itens que serão apresentados, é necessário também um VO que representa cada um desses itens. Da mesma forma que o VO anterior, antes do mecanismo aqui proposto, este VO possuía apenas os atributos que seriam apresentados como colunas na aba de lista, utilizando tipos de dados primitivos, oferecidos pela linguagem Java. Podemos visualizar a representação do VO da aba de lista do serviço de histórico escolar na Figura 6.10.

```
public class AlunoListItemVO implements Serializable {

      public String matricula;
      public String nome;
      public String curso;
}
```

Figura 6.10: VO que possui os atributos necessários à apresentação da aba de lista

A simples leitura da classe acima não deixa claro sua utilização no sistema. Com o intuito de embutir maior significado no código, construímos o VO apresentado na Figura 6.11. Nesta descrição, podemos observar o uso dos tipos

elementares de dados, apresentados no capítulo 5. Assim, cada informação do item é descrita no VO de acordo com a sua apresentação na vista.

No exemplo da Figura 6.11, devido ao significado injetado, podemos interpretar que a aba de lista será composta por três colunas: nome, matrícula e curso. Além disso, com o uso de tipos elementares, é possível entender que os campos “Nome” e “Curso” serão do tipo texto, meramente informativo, e o campo “Matrícula” será do tipo *link*, onde o usuário poderá clicar para gerar o histórico.

```
public class AlunoListItemVO implements Serializable, FieldConstrainable
{

      /* lista de atributos do vo */
      public String nome;
      public String matricula;
      public String curso;  // nome do curso atual

      /* inicialização dos constraints */
      private static UnmodifiableFieldsConstraints constraints;
      static{

          FieldsConstraints staticConstraints =
                  VOConstraintsUtilities.buildRowTableConstraints(
                        AlunoListItemVO.class,
                        new VOConstraintsUtilities.ColumnDescriptor[]{
                          new VOConstraintsUtilities.
                              LabelColumnDescriptor("nome", "Nome"),
                          new VOConstraintsUtilities.
                              LinkColumnDescriptor("matricula",
                              "Matrícula", "emitir"),
                          new VOConstraintsUtilities.
                              LabelColumnDescriptor("curso", "Curso")
                        }
            );

            constraints = new
                  UnmodifiableFieldsConstraints(staticConstraints);
      }


      public IFieldsConstraints getConstraints() {
            return constraints;
}}
```

Figura 6.11: VO que contém a descrição da aba de lista para o serviço de Histórico

Como vimos na seção 3.1, o arcabouço Hércules possui um engenho de descrição do caso de uso do serviço que utiliza arquivos XML. Tendo finalizado a construção das classes que descrevem o conteúdo da aba, basta apenas descrever o XML. A Figura 6.12 apresenta o XML que descreve a vista do serviço de histórico escolar.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<mvc>
  <view id="dummyId" title="emissaoHistoricoAndBoletim">
    <tabbox id="tabbedMenu">
      <tabs>
        <tab label="Lista" selected="false" />
      </tabs>
      <tabpanels>
        <tabpanel id="Lista">
          <hbox align="center" id="listaAlunos" width="98">
            <grid id="tableListaAlunos"
                  datasources="br.ufrj.siga.historico.
                               emissao.vo.AlunoListVO"
                  ref="listItensVO" class="listTable">
            <columns>
                  <column value="Nome" />
                  <column value="Matrícula" />
                  <column value="Curso" />
            </columns>
            <rows>
              <template>
                <row>
                  <label id="nome" datasources="br.ufrj.siga.historico.
                                    emissao.AlunoListItemVO"
                                   ref="nome" class="normal"/>
                 <label id="matricula" datasources="br.ufrj.siga.
                                     historico.emissao.AlunoListItemVO"
                       ref="matricula" class="normal" onclick="emitir"/>
                  <label id="curso" datasources="br.ufrj.siga.historico.
                                    emissao.AlunoListItemVO"
                                   ref="curso" class="normal" />
                </row>
              </template>
            </rows>
            </grid>
      <caption label="Escolha uma Matrícula:" />
            </hbox>
        </tabpanel>
      </tabpanels>
            </tabbox>
      </view>
</mvc>
```

Figura 6.12: Descrição da aba de lista utilizando a linguagem XML

Na descrição apresentada na Figura 6.12, podemos identificar as colunas apresentadas na Figura 6.13, que exibe a aba de lista do serviço de histórico escolar acessado por um aluno, ou seja, classe de usuário que possui acesso somente aos próprios dados, e não de terceiros. Com isso, não existe, no XML, a descrição da aba de seleção, utilizada para consulta de histórico de demais alunos.

Figura 6.13: Apresentação do serviço de Histórico Escolar para o perfil de aluno

Comparando a Figura 6.6 com a Figura 6.13, podemos perceber que a única diferença entre estas é a ausência ou presença da aba de seleção e o título do serviço. A aba de lista é apresentada exatamente da mesma forma em ambos

os serviços. Isto significa que acontecia uma duplicação de código no que diz respeito à descrição da aba de lista. Para cada serviço de acesso ao histórico escolar, dependendo do usuário que fez a requisição, existia um arquivo XML com a mesma descrição para a aba de lista.

Com a implementação deste trabalho, a descrição em XML passou a conter uma lacuna que é preenchida pela descrição realizada nas classes VO. Assim, as classes VO responsáveis pela descrição da aba de lista poderão ser utilizadas por todos os serviços de acesso ao histórico escolar, sem que haja necessidade de criar novas classes com o mesmo comportamento. A ligação entre essas classes e o XML é feita através do atributo *"tabSheetID"*, informado na anotação, conforme observamos no capítulo 5. Sendo assim, o XML de vista se limitará a:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<mvc>
      <view id="dummyId" title="emissaoHistoricoAndBoletim">
            <tabbox id="tabbedMenu">
                  <tabs>
                        <tab label="Lista" selected="false" />
                  </tabs>
                  <tabpanels>
                        <tabpanel id="AlunoLista" class="proxy"/>

                  </tabpanels>
            </tabbox>
      </view>
      ...
</mvc>
```

Figura 6.14: XML de vista do serviço de Histórico

Para indicar que a vista do serviço não será descrita através do XML, foi necessário alterar o comportamento do arcabouço Hércules, com a criação do atributo *"class"*. Assim, para que fique indicado que a descrição da vista será

implementada através de um VO, é necessário que o atributo *"class"* esteja indicado com o valor *"proxy"*. Além disso, para que seja identificado o VO responsável pela descrição desta parte correspondente da vista, é necessário que seu atributo *"tabSheetID"* esteja indicado com mesmo valor que o *id* da *tabpanel*, indicado no XML. O responsável por esta verificação é o filtro criado para este trabalho. Uma vez que esses VO´s são passados para a ação inicial, o filtro intercepta a ação procurando pelos VO´s necessários. Assim, é obrigatório o cadastro do filtro em tal ação.

6.2.3 Construção da Aba de Seleção

O processo de automatização para a aba de seleção ocorreu de maneira semelhante ao descrito acima para a aba de lista. Assim, será mostrada de forma sucinta a aplicação da geração automática da vista para a aba de seleção. Como já foi explicitado, o serviço de histórico escolar no SIGA, para usuários com perfil de maior privilégio, possui uma aba de seleção para que seja realizada a consulta do histórico. Assim, o usuário informa os dados, e requisita o histórico a partir desses dados.

O caso da aba de seleção diferencia da aba de lista porque é necessário apenas um VO. Além disso, foi criada uma anotação diferenciada com o objetivo de marcar corretamente a aba que está sendo construída. De acordo com a anotação que o VO possui, o filtro vai interpretar essa informação e construir a aba correspondente. A aba de seleção do serviço de histórico é mostrada na Figura 6.15.

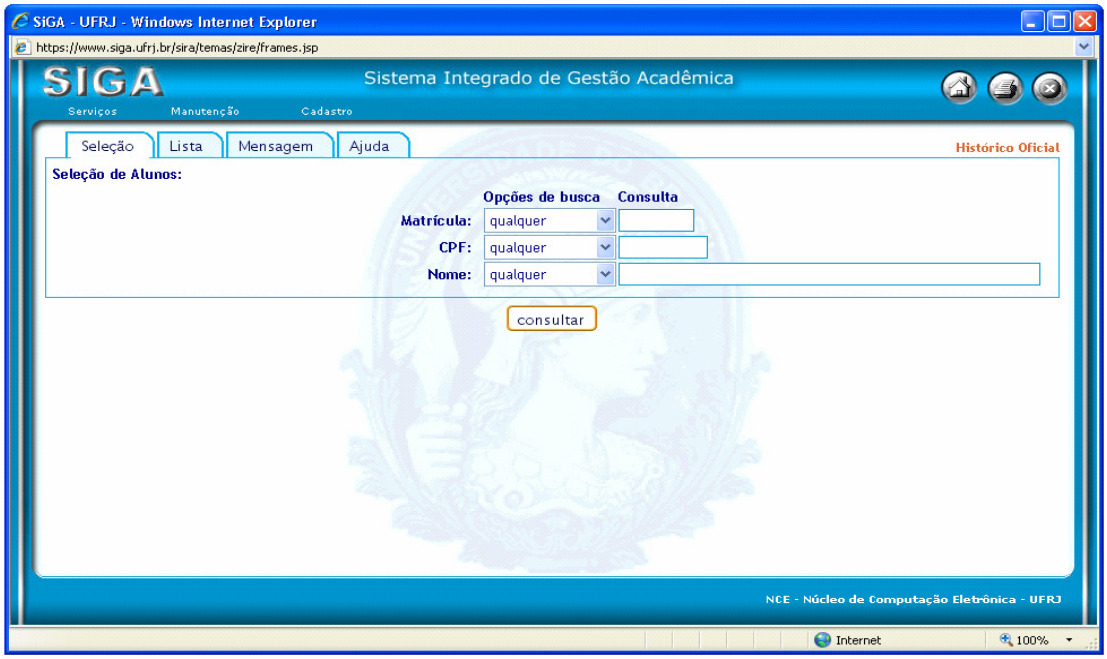

Figura 6.15: Aba de seleção do serviço de histórico escolar

Como no caso da aba de lista, antes deste trabalho, o VO não apresentava nenhum significado quanto aos seus atributos. Podemos visualizar na Figura 6.16 o VO utilizado na descrição da aba de seleção do serviço de histórico, apresentado na Figura 6.15.

```
public class AlunoQueryVO implements Serializable, FieldConstrainable{

      public String nome;
      public String nomeRestricao = "qualquer";

      public String matricula;
      public String matriculaRestricao = "qualquer";

      public String cpf;
      public String cpfRestricao = "qualquer";

}
```

Figura 6.16: VO destinado à aba de seleção para o serviço de histórico

Além do VO apresentado na Figura 6.16, a descrição da vista era implementada utilizando o descritor em XML da seguinte forma:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<mvc>
  <view id="dummyId" title="emissaoHistoricoAndBoletim">
    <tabbox id="tabbedMenu">
      <tabs>
        <tab label="Seleção" selected="false" />
        <tab label="Lista" selected="false" />
      </tabs>
      <tabpanels>
        <tabpanel id="Selecao">
          <hbox id="pnlEmissaoHistoricoAndBoletimBorder" class="bevel" align="center" width="98">
            <grid value="1" id="formSelectEmissaoHistoricoAndBoletim">
              <columns>
                <column class="shadow" flex="2" align="end"/>
                <column class="light" flex="1" align="start" value="Opções de busca" />
                <column class="light" flex="3" align="start" value="Consulta" />
              </columns>
              <rows>
                <row>
                  <label id="lbl_matricula" class="caption" value="Matrícula:"/>
                  <menulist id="matriculaRestricao" datasources="br.ufrj.siga.historico.emissao.vo.AlunoQueryVO"
                            ref="matriculaRestricao">
                  <menupopup>
                    <menuitem value="qualquer" label="qualquer"/>
                    <menuitem value="comecando por" label="começando por"/>
                    <menuitem value="contendo" label="contendo"/>
                    <menuitem value="terminando por" label="terminando por"/>
                    <menuitem value="igual a" label="igual a"/>
                  </menupopup>
                  </menulist>
                  <textbox id="matricula" datasources="br.ufrj.siga.historico.emissao.vo.AlunoQueryVO"
                           ref="matricula" maxlength="9" rows="1" cols="9" type="text" multiline="false" />
                </row>
                <caption label="Seleção de Alunos:"/>
              </grid> </hbox>
        </tabpanel>
        <tabpanel id="AlunoLista" class="proxy"/>
      </tabpanels>
  </tabbox>
</view>
```

Figura 6.17: Trecho da descrição da vista do serviço de histórico utilizando XML

Na descrição apresentada na Figura 6.17, foi omitida a descrição de todos os campos, devido à sua extensão. Dessa forma, foi mostrada apenas a descrição do primeiro campo (Matrícula), que se apresenta como: o texto 'Matrícula', uma lista de opções para consulta e uma caixa de texto, onde será informada a matrícula do aluno. As descrições dos demais campos apresentados na tela de seleção são feitas de maneira análoga.

A descrição da aba de seleção utilizando arquivo XML é muito extensa devido a grande duplicação de código, já que para cada campo, é necessário criar uma lista de opções, com os valores “qualquer”, “começando por”, “igual a”, “contendo” e “terminando por”. Isto significa 13 (treze) linhas de código em arquivo XML para cada campo, ou seja, para o exemplo da Figura 6.15, seriam 39 (trinta e nove) linhas de arquivo XML necessárias para descrever apenas os campos “matrícula”, “cpf” e “nome”, além das demais descrições necessárias.

Com a implementação deste trabalho, a descrição da aba de seleção passou a ser implementada utilizando VO. Com isso, diminuiu consideravelmente a descrição em XML, além de ter aumentado o índice de reutilização de código. A Figura 6.18 apresenta a descrição da vista utilizando VO.

Percebe-se na Figura 6.18, a presença de anotação, marcando o VO como descritor da aba de seleção para tal serviço. Além disso, pode-se ver também a presença do tipo elementar *“TextRestrictionDescriptor”* para descrever os campos da aba de seleção. Da mesma maneira que ocorre para a aba de lista, existem tipos elementares para cada campo existente na aba de seleção. Essa diferenciação de tipos elementares para abas foi implementada utilizando conceitos de tipos. Assim, a classe utilizada para descrever a vista no VO reconhece os tipos elementares que podem ser descritos para cada aba através de tipos que são aceitos nos parâmetros do método utilizado.

```
@QueryTabbedPaneRenderizable(
            formTitle = "Seleção de Alunos:",
            tabSheetID = "Selecao")
public class AlunoQueryVO implements Serializable, FieldConstrainable{

      public String nome;
      public String nomeRestricao = "qualquer";

      public String matricula;
      public String matriculaRestricao = "qualquer";

      public String cpf;
      public String cpfRestricao = "qualquer";

      private static UnmodifiableFieldsConstraints constraints;

      static {

          FieldsConstraints staticConstraints
                VOConstraintsUtilities.buildQueryFormConstraints(
                 AlunoQueryVO.class,
                 new VOConstraintsUtilities.
                      TextRestrictionDescriptor("matricula",
                      "Matrícula", 9),
                 new VOConstraintsUtilities.
                      TextRestrictionDescriptor("cpf", "CPF", 11),
                 new VOConstraintsUtilities.
                      TextRestrictionDescriptor("nome", "Nome", 70)
                );
      constraints = new UnmodifiableFieldsConstraints(staticConstraints);
      }

      public IFieldsConstraints getConstraints() {
            return constraints;
      }
}
```

Figura 6.18: Descrição da aba de seleção do serviço de histórico

Com a descrição da vista implementada no VO, o XML de vista do serviço de histórico ficou reduzido ao apresentado na Figura 6.19.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<mvc>
     <view id="dummyId" title="emissaoHistoricoAndBoletim">

          <tabbox id="tabbedMenu">

               <tabs>
                    <tab label="Seleção" selected="false" />
                    <tab label="Lista" selected="false" />
               </tabs>

               <tabpanels>
                    <tabpanel id="Selecao" class="proxy"/>
                    <tabpanel id="AlunoLista" class="proxy"/>
               </tabpanels>
          </tabbox>
     </view>
</mvc>
```

Figura 6.19: Descrição da vista do serviço de histórico utilizando o mecanismo

6.2.4 Reutilização de descrições

O SIGA possui um serviço chamado Registro de Tese, que possibilita o lançamento de dissertações defendidas para os cursos de pós-graduação. Tal serviço apresenta a descrição da aba de seleção da mesma maneira que o serviço de histórico escolar descrito no tópico anterior. Antes deste trabalho, o que acontecia era a duplicação da descrição da vista nos respectivos XML's de vista, já que não é possível a reutilização dessas partes de código. A Figura 6.20 apresenta o XML de vista com parte da descrição da aba de seleção do serviço de registro de tese.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<mvc>
 <view id="dummyId" title="registroTese">
   <tabbox id="tabbedMenu">
    <tabs>
      <tab label="Selecao" selected="true" />
      <tab label="Lista" selected="false" />
      <tab label="Edicao" selected="false" />
    </tabs>
    <tabpanels>
        <tabpanel id="Selecao">
            <box id="pnlQuestionarioBorder" class="bevel" align="center" width="98">
            <grid value="1" id="formSelectQuestionario">
            <columns>
              <column class="shadow" flex="2" align="end"/>
              <column class="light" flex="1" align="start" value="Opções de busca" />
              <column class="light" flex="3" align="start" value="Consulta" />
            </columns>
            <rows>
              <row>
              <label id="lbl_DRE" class="caption" value="Matrícula:"/>
              <menulist id="matriculaAlunoRestricao"
                        datasources="br.ufrj.siga.historico.tese.action.vo.consulta.ConsultaRegistroTeseVO"
                        ref="matriculaAlunoRestricao">
                    <menupopup>
                        <menuitem value="qualquer" label="qualquer"/>
                        <menuitem value="comecando por" label="começando por"/>
                        <menuitem value="contendo" label="contendo"/>
                        <menuitem value="terminando por" label="terminando por"/>
                        <menuitem value="igual a" label="igual a"/>
                    </menupopup>
              </menulist>
              <textbox id="matriculaAluno"
                        datasources="br.ufrj.siga.historico.tese.action.vo.consulta.ConsultaRegistroTeseVO"
                        ref="matriculaAluno" maxlength="9" rows="1" cols="9" type="text" multiline="false" />
              </row>
            </rows>
            <caption label="Seleção de Tese:"/>
      </grid>
   </box>
</tabpanel>
```

Figura 6.20: Descrição do serviço de Registro de Tese utilizando XML

Analisando a descrição acima, percebemos a necessidade do VO que contém os atributos que serão apresentados para consulta. Este VO é apresentado na Figura 6.21. Podemos perceber que o VO é composto apenas de atributos, sem qualquer significado embutido.

```
public class ConsultaRegistroTeseVO implements Serializable {
      public String nome;
      public String nomeRestricao = "qualquer";

      public String matricula;
      public String matriculaRestricao = "qualquer";

      public String cpf;
      public String cpfRestricao = "qualquer";
}
```

Figura 6.21: VO do serviço de registro de tese

Este trabalho possibilitou a reutilização da descrição da vista do serviço de histórico escolar, visto que esta foi implementada utilizando VO. Assim, o mesmo VO que descreve a aba de seleção para o serviço de histórico é também utilizado para a descrição da aba de seleção do serviço de registro de tese.

Com a finalidade de melhorar a apresentação do serviço, tornando-o mais descritivo, não utilizamos o VO do serviço de histórico, criamos um novo VO que herda do primeiro e altera apenas a anotação que dará título ao serviço. A Figura 6.22 ilustra o VO utilizado.

```
@QueryTabbedPaneRenderizable(
            formTitle = "Seleção de Tese:",
            tabSheetID = "Selecao")
public class ConsultaRegistroTeseVO extends AlunoQueryVO {
}
```

Figura 6.22: Descrição da aba de seleção do serviço de Registro de Tese

Como podemos perceber, a descrição da vista do serviço de registro de tese foi simplificada, visto que apenas utiliza um artifício já existente no sistema. Este artifício, representado aqui pelo VO *"AlunoQueryVO"*, torna-se o que chamamos de VO utilitário, visto que é utilizado no sistema por serviços diferentes. Podemos perceber que a diferença entre os VOs apresentados na Figura 6.18 e Figura 6.22 é apenas no âmbito visual, customizando o serviço com a modificação do seu título. Se os serviços fossem apresentados de maneira mais geral, ou seja, com títulos menos descritivos, por exemplo, apenas "Seleção", o mesmo VO poderia ser usado por ambos, sem a necessidade de criação de um novo VO.

Com a aplicação do mecanismo, descrevendo a vista no VO, o XML de vista do serviço de registro de tese se limita à descrição representada na Figura 6.23.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<mvc>
     <view id="dummyId" title="registroTese">
          <tabbox id="tabbedMenu">
               <tabs>
                    <tab label="Seleção" selected="true" />
                    <tab label="Lista" selected="false" />
                    <tab label="Edição" selected="false" />
               </tabs>
               <tabpanels>
                    <tabpanel id="Selecao" class="proxy"/>
                    <tabpanel id="Lista" class="proxy" />
                    <tabpanel id="Edicao">

                         <!--Descrição da aba de Edição-->

                    </tabpanel>
               </tabpanels>
          </tabbox>
     </view>
</mvc>
```

Figura 6.23: Descrição da vista do serviço de Registro de Tese utilizando o mecanismo

6.3 Considerações Finais

Apresentamos neste capítulo o sistema SIGA. Vimos que este sistema possui padrões de interface e operacional, o que possibilita a aplicação de um engenho de geração automática de sistemas de informação. Vimos também que, apesar de utilizar o arcabouço Hércules para automatizar a construção da vista, este não resolve o problema por completo, existindo ainda muita duplicação de código na descrição de serviços semelhantes. Dessa forma, o sistema SIGA é um

ótimo exemplo de aplicação para a aplicação do mecanismo proposto por este trabalho, com o objetivo de mitigar o problema de duplicação de código.

Este capítulo apresentou a aplicação do mecanismo proposto por este trabalho em um sistema real, o sistema SIGA, apresentado no capítulo 6. Para o exemplo de aplicação, escolhemos aplicar o mecanismo na descrição da vista das abas de seleção e lista do serviço de consulta ao histórico escolar. Posteriormente, aplicamos o mecanismo no serviço de registro de tese, que possui semelhanças visuais com o serviço de histórico. Para esclarecer as diferenças em termos de implementação, apresentamos como os serviços eram descritos antes e após este trabalho. No próximo capítulo apresentaremos a avaliação deste trabalho, baseada neste exemplo de aplicação.

7. Avaliação

Segundo Staa (2000), entende-se por avaliação o processo de medir alguma coisa em acordo com uma métrica, usando um procedimento ou técnica que depende de julgamento humano. O processo de medição deve levar em conta métricas e deve utilizar um procedimento isento de julgamento humano. Por fim, a métrica é um padrão de medição, que identifica a grandeza a ser medida.

Na engenharias tradicionais, medição é um assunto corriqueiro e bem entendido. Já na engenharia de software, medição ainda carece de muito amadurecimento (ZUSE, 1998). Por outro lado, a medição possui um papel muito importante no desenvolvimento de software com o fim de garantir um controle de qualidade e introduzir melhorias no processo.

Com a finalidade de verificar a validade do mecanismo proposto por este trabalho, utilizamos algumas métricas que determinam se houve melhoria ou degradação no processo de desenvolvimento de software após a adoção da técnica. Para garantir resultados confiáveis, as métricas foram balanceadas de acordo com os critérios levantados durante o desenvolvimento deste trabalho. Tais critérios são baseados no número de linhas de código e índice de satisfação por parte dos desenvolvedores com o novo mecanismo.

As métricas utilizadas neste trabalho são divididas em quantitativas e qualitativas. Sobre dados quantitativos, realizamos uma comparação do processo de desenvolvimento utilizado antes do mecanismo proposto e após a adoção

desta. O objetivo neste caso é apresentar o aumento de reutilização de código e conseqüente diminuição de erros inseridos no sistema. Sobre os dados qualitativos, realizamos um questionário com os desenvolvedores do sistema, com o fim de fazer um levantamento sobre a satisfação com relação à manutenibilidade, facilidade de uso e aumento de produtividade, entre outros.

7.1 Avaliação Quantitativa

Com o fim de realizar uma análise quantitativa sobre as modificações impostas pelo mecanismo proposto, utilizamos como métrica o número de linhas de código. Entendemos por número de linhas de código, a quantidade de linhas de código escritas na classe, desconsiderando linhas em branco e quebras de linhas. O objetivo de tal métrica é mostrar que, com a utilização do mecanismo, houve uma significativa redução no número de linhas de código inseridas no desenvolvimento dos serviços e, portanto, menor número de possíveis erros inseridos no sistema. Isto se deve ao fato do aumento de reúso de código já testado e com corretude comprovada. Ao longo desta seção, apresentaremos uma comparação quantitativa entre o código necessário antes da aplicação deste trabalho e o código utilizando o mecanismo aqui proposto.

7.1.1 Serviço de Histórico Escolar – Aba de Lista

Comecemos pelo primeiro exemplo apresentado no exemplo de aplicação, o serviço de consulta ao histórico escolar dos alunos. Vimos que utilizando a declaração estática, a descrição da aba de lista do serviço era realizada em três etapas: a construção de um VO que contém uma coleção de itens, que será

posteriormente preenchida pela ação do serviço; um segundo VO que contém os itens que serão apresentados ao usuário; e a descrição de toda a interface gráfica em um arquivo XML, utilizando a engenharia do arcabouço Hércules. Tais arquivos são apresentados na Figura 6.7, Figura 6.10 e Figura 6.12, respectivamente, no nosso exemplo de aplicação. Realizando a contagem de linhas de código necessárias para a construção dessa interface, chegamos aos números apresentados na Tabela 7.1.

Tabela 7.1: Análise quantitativa da aba de lista do serviço de histórico escolar

Aba de Lista do Histórico Escolar	VO – Coleção de Itens	VO – Itens da Lista	XML
Linhas de Código	3	6	33

Como podemos verificar na Tabela 7.1, no total, são necessárias 42 (quarenta e duas) linhas de código para construir a interface gráfica da aba de lista do serviço de consulta ao histórico escolar dos alunos. Observamos também que era muito comum a duplicação de código. Serviços diferentes que possuíam a mesma interface gráfica não faziam uso das mesmas classes.

Utilizando o mecanismo proposto neste trabalho, reduzimos consideravelmente a quantidade de linhas de código necessárias para a implementação de novos serviços, além de proporcionar um maior reúso. O principal exemplo de reúso diz respeito ao VO que possui a coleção de itens da lista, que pode ser utilizado por qualquer serviço do sistema, tratando-se portanto de uma classe utilitária do sistema. Sendo assim, não contabilizamos mais a implementação desta classe no desenvolvimento de um serviço. A Tabela 7.2

apresenta a contagem de linhas de código necessárias para o desenvolvimento do mesmo serviço de consulta ao histórico escolar dos alunos utilizando o mecanismo aqui proposto.

Tabela 7.2: Análise quantitativa da aba de lista do serviço de histórico escolar utilizando o mecanismo

Aba de Lista do Histórico Escolar	VO – Coleção de Itens	VO – Itens da Lista	XML
Linhas de Código	2	14	13

Analisando a Tabela 7.2, percebemos a redução significativa de novas linhas de código necessárias ao desenvolvimento do serviço, totalizando em 29 (vinte e nove). A coluna que apresenta o VO de coleção de itens aparece como opcional porque este novo VO só será criado se houver necessidade de customização do serviço, alterando por exemplo, o título que é exibido na aba.

Chamamos a atenção em especial para a redução do arquivo XML, isto porque este arquivo é utilizado apenas para um único serviço, não podendo ser reutilizado. A descrição feita no arquivo XML é também grande fonte de duplicação de código, onde a maioria dos desenvolvedores apenas copiava descrições já existentes de outros serviços e adaptava para o novo serviço. Em entrevistas realizadas com os desenvolvedores do sistema, constatamos que a maioria não se sentia confortável com a descrição da interface em arquivos XML, o que, aliado à duplicação de código, dificultava a manutenção do sistema e aumentava a possibilidade de novos erros serem inseridos.

7.1.2 Serviço de Histórico Escolar – Aba de Seleção

O segundo exemplo apresentado no exemplo de aplicação trata da descrição da aba de seleção do serviço de consulta ao histórico escolar do aluno, disponível para usuários com maior privilégio de acesso ao sistema, por exemplo, secretarias. Para a descrição da aba de seleção são necessários apenas dois arquivos, um VO que contém os atributos que formarão a consulta e o XML que descreve a interface gráfica. Estes arquivos são apresentados na Figura 6.16 e Figura 6.17, respectivamente, no exemplo de aplicação. A contagem de linhas de código necessárias para a descrição dessa aba é apresentada na Tabela 7.3.

Tabela 7.3: Análise quantitativa da aba de seleção do serviço de histórico escolar

Aba de Seleção do Histórico Escolar	VO – Itens de Consulta	XML
Linhas de Código	8	64

Observamos na Tabela 7.3 que, no total, são necessárias 72 (setenta e duas) linhas de código para construir a interface gráfica da aba de seleção do serviço de consulta ao histórico escolar dos alunos. Assim como no caso da descrição da aba de lista, era muito comum a duplicação de código, onde serviços diferentes que possuíam a mesma interface gráfica não faziam uso das mesmas classes.

A Tabela 7.4 apresenta a nova contagem de linhas de código necessárias na implementação da consulta ao histórico escolar utilizando o mecanismo aqui proposto. Além disso, chamamos à atenção que é encorajada a prática de reúso de classes. No próximo exemplo, apresentaremos um caso de uso do sistema

SIGA que faz uso do VO de itens de consulta, apesar de serem serviços diferentes.

Tabela 7.4: Análise quantitativa da aba de seleção do serviço de histórico utilizando o mecanismo

Aba de Seleção do Histórico Escolar	VO – Itens de Consulta	XML
Linhas de Código	16	15

Mais uma vez, é visível a redução do número de linhas de código necessárias ao desenvolvimento do serviço, caindo para 31 (trinta e um). Mais uma vez, observamos a significativa redução no tamanho do arquivo XML e aumento de linhas no VO. Tal aumento está alinhado à adição de significado ao código.

7.1.3 Serviço de Registro de Tese – Aba de Seleção

Nesta seção, abordaremos o terceiro exemplo de caso de uso do sistema SIGA utilizado em nosso exemplo de aplicação, o serviço utilizado para registro de teses. Como vimos na seção 6.1.2, este serviço possui uma aba de seleção utilizada para a consulta de teses já cadastradas, que é apresentada da mesma maneira que a aba de seleção do serviço de histórico escolar.

Observamos que, com a utilização do mecanismo proposto por este trabalho, é possível reutilizar toda a descrição já implementada para o serviço de histórico no serviço de registro de tese. Com isso, todo o tempo que seria necessário para a implementação da aba de seleção deste serviço não é mais

consumido, diminuindo, portanto, o número de linhas de código inseridas no sistema. Para fins de comparação, iremos apresentar as diferenças de implementação deste serviço em três formas: antes da aplicação do mecanismo, como era feito originalmente no desenvolvimento do sistema; depois da aplicação do mecanismo sem reutilizar a descrição já implementada em outro serviço; e depois da aplicação do mecanismo, reutilizando a descrição existente.

Como vimos na seção anterior, a descrição da aba de seleção é realizada através de dois arquivos: um XML, que descreve a interface gráfica, e o VO, que contém os atributos que formarão a consulta. Estes arquivos são apresentados na Figura 6.20 e Figura 6.21, respectivamente, no exemplo de aplicação. A contagem de linhas de código necessárias para a descrição dessa aba utilizando o mecanismo original, com descrição estática, é apresentada na Tabela 7.5.

Tabela 7.5: Análise quantitativa do serviço de registro de tese utilizando a descrição estática

Aba de Seleção do Registro de Tese (Antes)	VO – Itens de Consulta	XML
Linhas de Código	8	64

Observamos na Tabela 7.5 que, no total, são necessárias 72 (setenta e duas) linhas de código para construir a interface gráfica da aba de seleção do serviço de registro de tese. Chamamos a atenção mais uma vez para o elevado número de descrição em XML, o que aumentava a duplicação de código e a prática de “copiar, colar e adaptar” descrições existentes. Como podemos perceber na Tabela 7.3 e na Tabela 7.5, não por acaso, as descrições da aba de

seleção para os serviços de histórico escolar e registro de tese apresentam o mesmo número de linhas de código, já que são idênticas.

Apresentaremos agora a contagem realizada na implementação deste serviço utilizando o mecanismo aqui proposta. Neste caso, apenas para comparação, não utilizamos o reúso das classes já existentes, criando novas classes específicas para o serviço, ignorando o fato de já haver uma classe com a mesma descrição. A contagem é apresentada na Tabela 7.6.

Tabela 7.6: Análise quantitativa do serviço de registro de tese utilizando o mecanismo

Aba de Seleção do Registro de Tese (Sem reúso)	VO – Itens de Consulta	XML
Linhas de Código	16	15

Observamos na Tabela 7.6, que, utilizando o mecanismo proposto por este trabalho, são necessárias, no total, 31 (trinta e uma) linhas de código. Mais uma vez, é visível a diferença na quantidade de código novo sendo inserido no sistema para cada novo serviço desenvolvido.

Por último, apresentaremos a contagem realizada no processo de implementação do serviço levando em conta que existe outro serviço com a mesma descrição no sistema. Aplicamos então o mecanismo aqui proposto para desenvolver o serviço de registro de tese, reutilizando a descrição do serviço de histórico escolar. Os dados são apresentados na Tabela 7.7.

Tabela 7.7: Análise quantitativa do serviço de registro de tese utilizando o mecanismo com reúso de código

Aba de Seleção do Registro de Tese (Com reúso)	VO – Itens de Consulta (Opcional)	XML
Linhas de Código	2	15

Observamos na Tabela 7.7, que, utilizando o mecanismo com reutilização de código já existente, são necessárias apenas 17 (dezessete) linhas novas de código, sendo que o VO de consulta é opcional, podendo fazer uso daquele já existente para o serviço de histórico escolar.

A significativa redução no número de linhas de código novas embutidas no sistema no desenvolvimento de cada serviço revela um aumento na qualidade do software e a diminuição de novos erros inseridos. A qualidade é garantida devido ao reúso de classes já testadas e comprovadas corretas, facilitando ainda a manutenção do código. O reúso também diminui o número de erros inseridos no sistema visto que é comprovado que a cada 10 (dez) linhas adicionadas ao código fonte de um sistema, são inseridos novos erros no sistema. A Figura 7.1 ilustra um gráfico que apresenta a evolução do desenvolvimento de software nas três fases propostas nesta seção.

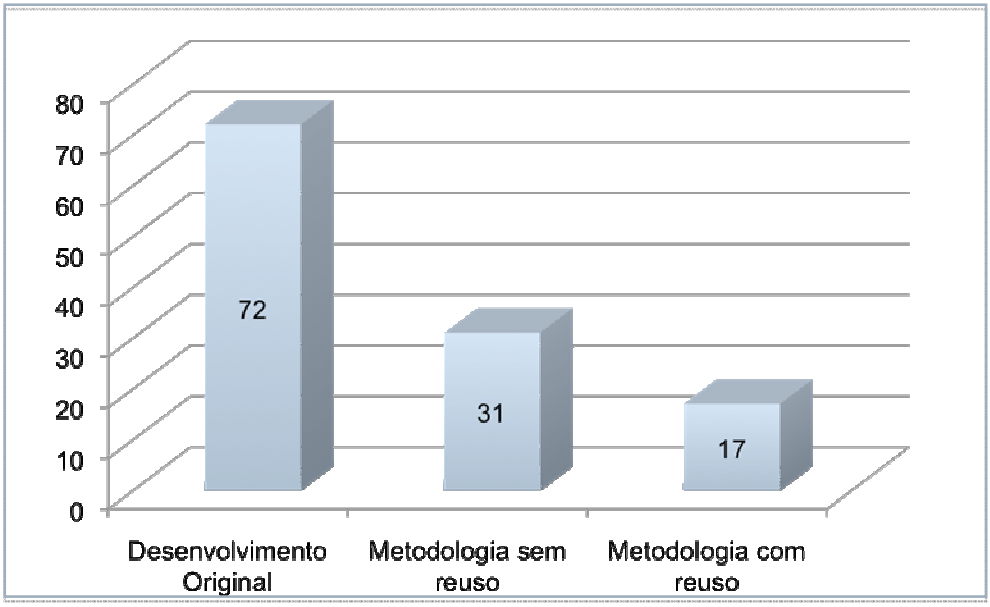

Figura 7.1: Evolução do desenvolvimento de software em quantidade de linhas de código para o serviço de Registro de Tese

7.2 Avaliação Qualitativa

Com a finalidade de obter uma avaliação qualitativa sobre o mecanismo proposto por este trabalho, realizamos um questionário com os desenvolvedores do sistema SIGA, utilizado em nosso exemplo de aplicação. Utilizamos questões relacionadas ao desenvolvimento do sistema antes da aplicação do mecanismo e após esta. Com isso, foi possível realizar o levantamento da satisfação dos desenvolvedores com relação à manutenibilidade, facilidade de uso e aumento de produtividade.

Para a aplicação do questionário, entramos em contato com a equipe do sistema SIGA e agendamos uma reunião. Antes de aplicar o questionário, a autora deste trabalho realizou uma breve apresentação sobre o trabalho e as modificações apresentadas no capítulo de exemplo de aplicação. Além disso, por

se tratar de um sistema acadêmico, é constante a modificação da equipe de desenvolvedores. Dessa forma, os desenvolvedores que conheceram o mecanismo, mas já não fazem mais parte da equipe do sistema SIGA, foram contatados via correio eletrônico.

Adotamos para o questionário, a escala *Likert* (LIKERT, 1932), que é um instrumento eficiente de coleta do grau de intensidade da opinião dos participantes da pesquisa em relação ao assunto proposto (REA, PARKER, 2000). Assim, cada uma das questões foi apresentada com cinco variações (graus de intensidade) para as respostas: Discordo *Fortemente*, *Discordo*, *Não tenho opinião, Concordo e Concordo Fortemente.* Estas variações são ancoradas em 5 pontos, assim são definidos os valores 1, 2, 3, 4 e 5, respectivamente. Além disso, este tipo de escala é bipolar, variando em polaridade semântica positiva e negativa. A primeira vai ao encontro das hipóteses deste trabalho; e a segunda vai de encontro às hipóteses deste trabalho. No caso de uma afirmação ser negativa, os pontos são invertidos.

A Tabela 7.8 apresenta o questionário aplicado com as informações da escala *Likert*. A última coluna da Tabela indica quais questões foram criadas para validar a coerência das respostas de cada participante. Além do questionário, foi reservado um espaço para que o participante pudesse definir com suas próprias palavras a impressão que teve sobre o mecanismo.

Tabela 7.8: Questionário aplicado aos desenvolvedores do sistema SIGA

	Questões	Polaridade Semântica	Qual questão é validada
1	A equipe tem facilidade em entender descrições puramente em XML, como eram descritos os serviços antes do trabalho.	NEGATIVA	
2	A equipe reconhece que a prática “copia-cola-adapta” provoca duplicação de código, o que dificulta a manutenção.	POSITIVA	
3	É interessante para a equipe que a descrição em XML seja reduzida.	POSITIVA	1
4	A equipe acredita que, com a redução da descrição em XML, o desenvolvimento dos serviços torna-se mais produtivo.	POSITIVA	3
5	O uso de descrições em VO torna o código mais inteligível.	POSITIVA	
6	O uso de descrições em VO facilita o entendimento do código, e, consequentemente, sua manutenção.	POSITIVA	5
7	O mecanismo proposto aumenta o nível de reúso do código.	POSITIVA	
8	Há aumento na qualidade de software devido ao reúso de código.	POSITIVA	7
9	O mecanismo é de fácil aplicação no desenvolvimento do software, podendo substituir o mecanismo anterior.	POSITIVA	
10	Em uma escala de 1 a 10, qual o seu nível de satisfação com o mecanismo?	POSITIVA	

Para análise das informações coletadas através do questionário, utilizamos o software Microsoft Excel 2003. Apresentamos na Tabela 7.9, a distribuição das respostas em porcentagem para cada pergunta do questionário. Em seguida, apresentamos na Figura 7.2, o gráfico de barras gerado pela ferramenta. Analisando os dados, podemos perceber que o mecanismo teve uma repercussão positiva e, portanto, bem aceita entre os desenvolvedores do sistema.

Tabela 7.9: Análise das respostas do questionário em porcentagem

Questão	Discordo Fortemente	Discordo	Não tenho opinião	Concordo	Concordo Fortemente
1	25	50	25	0	0
2	0	0	12,5	25	62,5
3	0	0	0	0	100
4	0	0	0	0	100
5	0	0	0	25	75
6	0	0	0	50	50
7	0	0	0	12,5	87,5
8	0	0	0	12,5	87,5
9	0	0	0	12,5	87,5
10	0	0	0	12,5	87,5

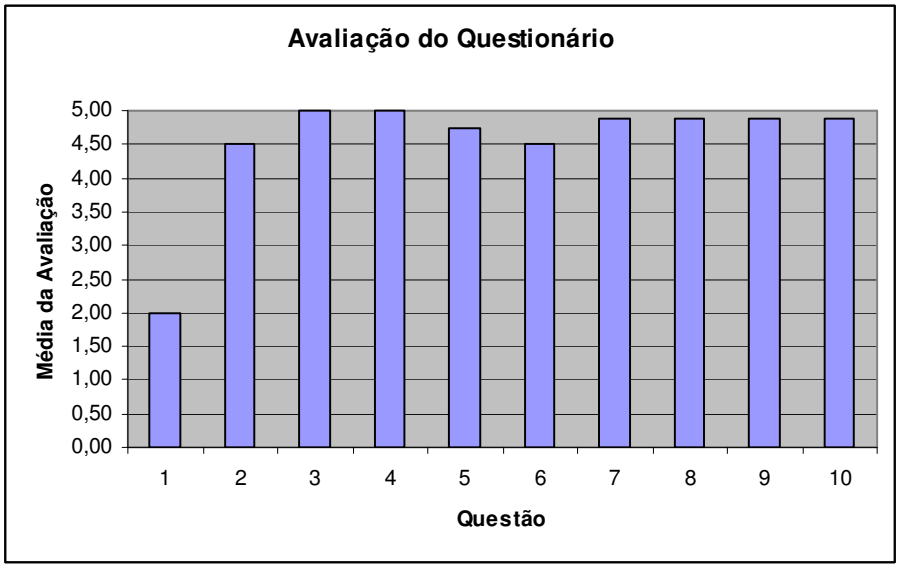

Figura 7.2: Gráfico de barras com a análise do questionário

Além da avaliação levantada para cada questão, dois participantes discorreram comentários gerais sobre a implantação do mecanismo proposto por este trabalho no sistema SIGA. Seguem os comentários:

> **"O mecanismo realmente acelerou o processo de desenvolvimento e entendimento do software e tornou mais prazerosa a manutenção do mesmo."**
>
> **"O Hércules usa uma linguagem própria e prende o desenvolvedor a criação de um XML para as telas do SIGA, mas os XMLs são muito extensos e dificultam até mesmo o entendimento do desenvolvedor. Este novo mecanismo define uma nova forma bem mais adequada para a construção das telas com o uso de VO's, facilitando consideravelmente a construção das telas do Siga."**

7.3 Considerações Finais

Apresentamos neste capítulo a avaliação do mecanismo proposto por este trabalho aplicada no sistema SIGA. Para que a avaliação fosse completa, dividimos em avaliação quantitativa e avaliação qualitativa. Na primeira, fizemos uma comparação em nível de linhas de código, ilustrando as diferenças sofridas em termo de código fonte. Na segunda, levantamos a opinião e satisfação dos desenvolvedores do sistema quanto ao seu novo mecanismo de descrição de serviços. Apresentaremos no próximo capítulo a conclusão deste trabalho.

8. Conclusão

Nos tempos atuais, é notável o crescimento da indústria de software, impulsionado pela também crescente demanda do mercado. Isto faz com que o prazo de desenvolvimento e manutenção de sistemas de informação seja cada vez menor. Para isso, uma das principais qualidades requeridas para um sistema é que seja altamente flexível. Aliado a esse fato, vimos também neste trabalho a existência de padronização operacional e de interface gráfica dos sistemas de informação. Estes fatos tornam possível a aplicação de um engenho de geração automática de sistemas.

Um estudo realizado demonstrou que a melhor forma de alcançar flexibilidade em sistemas de informação é através de técnicas de implementação de variabilidade, particularmente aquelas suportadas em tempo de execução, tais como introspecção, infra-estruturas de componentes e carregamento dinâmico de bibliotecas. Por outro lado, as técnicas que fornecem níveis mais elevados de variabilidade também implicam processos mais complexos de engenharia (BRAGANÇA, MACHADO, 2004). Apresentamos algumas dessas técnicas na seção 2.5. Vimos também no Capítulo 2 que uma forma de mitigar o problema da complexidade está na adoção de métodos de engenharia de domínio, aplicados em paralelo com o método de engenharia de aplicação. Porém, a proposta atual, baseada na declaração estática de componentes gera o problema da explosão fatorial destes.

Apresentamos os trabalhos que serviram como base para o desenvolvimento desta dissertação. Os mais relevantes são os arcabouços Hércules e Megara. Vimos que o arcabouço Hércules propõe a geração automática de sistemas como uma forma de melhoria da qualidade do desenvolvimento, porém não resolvia o problema da duplicação de código. O arcabouço Megara propõe um desenvolvimento baseado em MDA utilizando o arcabouço Hércules, porém este também apresentou deficiências quanto à facilidade de descrição dos serviços.

O mecanismo apresentado neste trabalho complementa o arcabouço Hércules, que, como vimos, utiliza a linguagem XML na descrição dos serviços de um sistema. Esta forma de descrição, baseada no comportamento estático, não resolvia o problema de duplicação de código por completo. A cada novo serviço do sistema, é necessário escrever um novo arquivo XML, mesmo que a descrição seja exatamente igual a outra já existente. Assim, apresentamos uma forma de descrever parte do sistema através de modelos enriquecidos, possibilitando o reúso desses modelos.

As contribuições trazidas com a proposição do mecanismo são: aumento do nível de abstração do processo de desenvolvimento; flexibilidade do código; representação e reutilização da arquitetura da interface do sistema; melhorias no arcabouço Hércules; possibilidade de reúso de componentes e adição de significado ao código, tornando-o mais inteligível, o que melhora a manutenção do sistema.

A utilização do mecanismo elevou o nível de abstração, pois ao invés de descrever a interface gráfica em arquivos XML, os desenvolvedores apenas constroem um modelo que é enriquecido de informação. A implementação do engenho, apresentada no Capítulo 6, possibilitou a construção de componentes de vista reconhecidos pelo Hércules.

Há flexibilidade no código visto que é realizado um levantamento de todos os componentes do sistema e criados tipos elementares que definirão esses componentes. Quando um novo componente é semelhante a algum outro existente, basta estender o comportamento desse último, sem que sua essência se perca.

Vimos no exemplo de aplicação um exemplo que comprova o aumento do reúso de componentes na descrição do sistema. Nos casos em que os serviços são semelhantes, não existe mais a necessidade de escrever um novo arquivo XML, basta apenas utilizar a descrição já existente, em classes VO. Além disso, essas classes foram enriquecidas de significado, sendo de fácil percepção o seu objetivo no sistema.

O sistema SIGA foi utilizado no exemplo de aplicação visto que é um ótimo exemplo de sistema que segue padrões de interface e operacional. Além disso, existia uma forte prática de copiar descrições realizadas em linguagem XML cada vez que novos serviços surgiam, o que demonstrava a grande duplicação de código existente no sistema.

A validação do mecanismo proposto por este trabalho foi realizada através da avaliação quantitativa e qualitativa. A primeira levou em conta detalhes de implementação necessários para a implantação do mecanismo em um sistema, quantificando o reúso através de linhas de código. A segunda foi realizada através de um questionário aplicado aos desenvolvedores do sistema SIGA, a fim de colher suas impressões e satisfação quanto ao novo mecanismo a ser aplicada no sistema. Em ambas as avaliações, chegamos a resultados positivos, apresentados através de gráficos e cálculos estatísticos.

Uma das fragilidades encontradas no mecanismo e que poderá ser solucionada com trabalhos futuros é o uso efetivo da declaração comportamental de componentes. Este trabalho se baseou na declaração comportamental, utilizando métodos, mas, com o objetivo de diminuir o escopo da implementação, ainda foram necessários os atributos.

Outra fragilidade encontrada está na gerência dos tipos elementares de dados. Em casos de sistemas maiores, a gerência dos tipos elementares pode ser uma tarefa dispendiosa e levanta a necessidade de um engenho capaz de gerenciar esses tipos. Além disso, este trabalho pode ser automatizado através da criação de um arcabouço que suporte a criação de novos tipos de dados. Os tipos disponíveis atualmente são os que já estão implementados nas bibliotecas existentes. Para criar novos tipos, todo o trabalho tem de ser feito manualmente. Poderia ser desenvolvido um sistema de meta-tipos que permita criar um tipo usando uma abstração de nível mais alto.

Para a adaptação deste trabalho ao arcabouço Hércules, foi necessário utilizar o arcabouço Chameleon como um engenho de tradução de classes VO em componentes reconhecidos pelo Hércules. Isto poderia ser solucionado por um trabalho futuro que incorporasse ao Hércules o engenho de tradução. Dessa forma, o Hércules poderia ser estendido para trabalhar também com classes VO.

Referências Bibliográficas

ALUR, D.; CRUPI, J.; MALKS, D., **Core J2EE Patterns**: Best Practices and Design Strategies. 2. ed. Sun Microsystems Press, 2001. 627 p.

AndroMDA, **AndroMDA: The Open Source MDA Tool**. Disponível em: http://www.andromda.org/. Acesso em jun. 2009

BECK, K., **Extreme Programming Explained**: Embrace Change. 2. ed. Addison Wesley Professional, 2004. 190 p.

BERGMANN, U., **Evolução de cenários através de um mecanismo de rastreamento baseado em transformações**. 2003. 211p. Tese (Doutorado em Informática) – Pontifícia Universidade Católica do Rio de Janeiro, Rio de Janeiro, RJ, Brasil.

BERGMAN, G., **Filosofia de Banheiro**: sabedoria dos maiores pensadores mundiais para o dia-a-dia. São Paulo: Madras, 2004. 144 p.

BOLDSOFT, **BOLD Software**. http://www.boldsoft.com/. Acesso em ago. 2009.

BOSSONARO, A., MORAES, J., FONTANETTE, V., GARCIA, V., PRADO, A., **Implementações de Frameworks de Componentes dirigidas por Modelos do Método Catalysis**. In: The Fourth Congress of Logic Applied to Technology (LAPTEC'2003), 2003, Marília. Proceedings of The Fourth Congress of Logic Applied to Technology (LAPTEC'2003), 2003. v. 1. p. 1-12.

BOOCH, G., RUMBAUGH, J., JACOBSON, I., **The Unified Modeling Language User Guide**. Addison-Wesley Professional. 1 ed. 1998. 512 p.

BRAGA, R., **Busca e Recuperação de Componentes em um Ambiente de Reúso**. 2000. Tese (Doutorado em Engenharia de Sistemas e Computação) – COPPE, Universidade Federal do Rio de Janeiro, Rio de Janeiro, RJ, Brasil.

BRAGA, R., et al., **Introdução aos Padrões de Software**. *IV Conferência Latino-Americana em Linguagens de Padrões para Programação*, Campos do Jordão, Brasil, 2005.

BRAGA, R., WERNER, C., **Odyssey-DE**: Um Processo para Desenvolvimento de Componentes Reutilizáveis, *X Conferência Internacional em Tecnologia de Software (X CITS)*, pp.177-194, Curitiba, Maio, 1999.

BRAGANÇA, A., MACHADO, R., **Engenharia de Domínio no Suporte ao Aumento de Flexibilidade nos Sistemas de Software**. *IV Conferência para a Qualidade nas Tecnologias de Informação e Comunicações – QUATIC'2004*, Porto, Portugal, 2004.

BUSCHMANN, F., MEUNIER, R., ROHNERT, H., SOMMERLAD, P., STAL, M., **Pattern-Oriented Software Architecture**, John Wiley and Sons, New York, NY, 1996

CORRÊA, B., **Integridade e Translação Representacional das Entidades**. 2005. 111 p. Dissertação (Mestrado em Informática) – Programa de Pós-Graduação em Informática, Universidade Federal do Rio de Janeiro, Rio de Janeiro. 2005.

D' SOUZA, D., WILLS, A. **Objects, Components and Frameworks with UML: The Catalysis(SM) Approach**. Addison-Wesley, 1998, 816 p.

EVANS, E., **Domain-Driven Design**: Tackling Complexity in the heart of software. Addison-Wesley, 2003. 560 p.

FILETO, R., MEIRA, C., COSTA, C., MASSHURÁ, S., **A Construção de um Gerador de Programas Aplicativos segundo Conceitos de Análise de Domínios**, Anais do X Simpósio Brasileiro de Engenharia de Software; Ed. SBC; 1996; pp 119-135.

FLEURY, M., REVERBEL, F., **The Jboss Extensible Server**. ACM/IFIP/USENIX International *Middleware* Conference, Rio de Janeiro, 2003. Disponível em: http://www.cc.gatech.edu/classes/AY2007/cs4365_spring/mirror/jboss-mw2003.pdf. Acesso em: fev. 2009.

FOREMAN, J. **Product Line Based Software Development – Significant Results, Future Challenges**. Software Technology Conference, Salt Lake City, UT, 1996.

FOWLER, M. **UML Essencial**. 3. ed. Bookman, 2005. 160 p.

FREEMAN, P. **A Conceptual Analysis of the Draco Approach to Constructing Software Systems**, IEEE Transactions on Software Engineering, SE-13(7), p 830-844, July 1987.

FREITAS, F., LEITE, J. C. S. P., **Aplicando reuso de software na construção de ferramentas de engenharia reversa**, Anais do XI Simpósio Brasileiro de Engenharia de Software; Ed. SBC, 1997, pp 265-280.

GAMMA, E. et al. **Design Patterns**: Elements of reusable object-oriented software. Addison-Wesley, 1999. 395 p.

GOHIL, B., **Automatic GUI Generation**. 2001. 58 p. Dissertação (Bacharelado em Engenharia de Software) – Universidade Bournemouth, Reino Unido.

GRISS, M., FAVARO, J., Alessandro, M., **Integrating Feature Modeling with the RSEB**. *V International Conference on Software Reuse*, Victoria, Canada, 1998.

Disponível em: http://www.favaro.net/john/home/publications/rseb.pdf. Acesso em: fev. 2009.

HELM, R., VLISSIDES, J., GAMMA, E., JOHNSON, R., **Padrões de Projeto**: Soluções Reutilizáveis de Software Orientado a Objetos, Bookman Companhia, 2000.

HEINEMAN G. T., COUNCILL W. T., **Component-Based Software Engineering**: Putting the pieces together, Addison-Wesley, 2001, 818 p.

HUNTER, J., CRAWFORD W. **Java Servlet Programming**. 2 ed. O´reilly. 2003. 780 p.

ISSA, L., **Desenvolvimento de Interface com Usuário Dirigido por Modelos e Geração Automática de Código**. 2006. 77 p. Dissertação (Mestrado em Ciência da Computação) – Instituto de Ciências Exatas, Universidade Federal de Minas Gerais, Minas Gerais, Belo Horizonte.

JIANCHENG, W., XUDONG, L., LEI, L., **A Model of User Interface Design and Its Code Generation**. Information Reuse and Integration, IEEE International Conference, Las Vegas, Estados Unidos, 2007, pp. 128-133, INSPEC: 9824209

Kang, K., Cohen, S., Hess, J. Novak, W., Peterson, A. **Feature-Oriented Domain Analysis (FODA) Feasibility Study**. Relatório Técnico. CMU/SEI-90-TR-021.

KNIGHT, J., MYERS, E., **Improved Inspection Technique**, Communications of the ACM, v.36 n.11, pp. 51-61, Novembro, 1993.

KRUEGER, C., **Software Reuse**. *ACM Computing Surveys*, v. 24, n. 2, pp. 131-183, New York, NY, USA, 1992.

LARMAN, C., **Utilizando UML e Padrões**: Uma introdução a Análise e ao Projeto Orientados a Objetos. 2.ed. Bookman Companhia, 2004.

LEITE, M. N. C. **Definição de Tipos de Dados na Modelagem de Negócios**: uma proposta de ferramenta de captura de metadados para geração automática de código. 2006. 110 p. Dissertação (Mestrado em Informática) – Programa de Pós-Graduação em Informática, Universidade Federal do Rio de Janeiro, Rio de Janeiro.

LEITE, J. C. S. P., SANT'ANNA, M., FREITAS, F. G., **Draco-PUC: A Technology Assembly for Domain-Oriented Software Development**. IEEE Internacional Conference on Software Reuse, 1994, pp 94-101.

LEMOS, R.G.S. **Restrição Temporal de Acesso**: proposta de implementação de requisito semi-funcional para controle de acesso. 2005. 67 p. Monografia (Bacharelado em Ciência da Computação) – Instituto de Matemática, Universidade Federal do Rio de janeiro, Rio de Janeiro.

LEVY, L. **A Metaprogramming Method and Its Economic Justification.** IEEE Transactions on Software Engineering 12(2), 1986, pp 272-277.

LIKERT, R. **A Technique for the Measurement of Attitudes**, Arquivos de Psicologia, Vol. 140, jun. 1932. 55 p.

MAIA, N., **Odyssey-MDA: Uma Abordagem para a Transformação de Modelos**. 2006. Dissertação (Mestrado em Engenharia de Sistemas e Computação) – COPPE, Universidade Federal do Rio de Janeiro, Rio de Janeiro.

MILLER, N. **A Engenharia de Aplicações no Contexto da Reutilização baseada em Modelos de Domínio**. 2000. Dissertação (Mestrado em Engenharia de Sistemas e Computação) – COPPE, Universidade Federal do Rio de Janeiro, Rio de Janeiro.

MOLINA, P. J., **Especificación de Interfaz de Usuario: de los requisitos a la generación automática**. 2003. Tese (Doutorado) – Departamento de Sistemas Informáticos y Computación, Universidad Politecnica de Valencia, Valencia, Espanha, 402 p.

MRACK, M., **Geração Automática e Assistida de Interface de Usuário em Tempo de Execução**. 2008. Dissertação (Mestrado em Computação) - Universidade Federal do Rio Grande do Sul, Rio Grande do Sul.

NEIGHBORS, J., The Draco Approach to Constructing Software from Reusable Components, IEEE Transactions on Software Engineering, SE-10, p 564- 573, Set. 1984.

OLIVEIRA, H., ROCHA, C., GONÇALVES, K., SOUZA C. **Utilização de Sistemas Críticos nas Atividades de Engenharia de Domínio e Aplicações.** XV Simpósio Brasileiro de Engenharia de Software pp. 21-35, Rio de Janeiro, Outubro, 2001.

OMG, **XML Metadata Interchange**. Disponível em: http://www.omg.org/technology/documents/formal/xmi.htm. 2007. Acesso em ago.2009.

PAIS, A. P. V. **Arquitetura de Controle Hércules**: a base para a geração automática de sistemas de informação com ênfase na camada de controle. 2004. 121 p. Dissertação (Mestrado em Informática) – Programa de Pós-Graduação em Informática, Universidade Federal do Rio de Janeiro, Rio de Janeiro.

PAIS, A. P. V., OLIVEIRA, C. E. T. **Enhancing UML Expressivity towards Automatic Code Generation.** Proc. VII International Conference on Object-Oriented Information Systems - OOIS'01. Calgary, Canadá: Springer-Verlag. 2001, v. 1, pp. 335-344.

PELECHANO, V., PASTOR, O., INSFRÁN, E., **Automated Code Generation of Dynamic Specializations: an approach based on design patterns and formal**

techniques. Data & Knowledge Engineering, Elsevier, 2002, v. 40, nl. 3, pp. 315-353.

PEREIRA, L. A., **Megara**: Uma Ferramenta para o Desenvolvimento de Sistemas Baseado em Modelos. 2006. 71 p. Dissertação (Mestrado em Informática) – Programa de Pós-Graduação em Informática, Universidade Federal do Rio de Janeiro, Rio de Janeiro.

PRADO, A. F., LEITE, J. C. S. P., BERGMANN, U., **Desenvolvimento de Sistemas Orientados a Objetos Utilizando o Sistema Transformacional Draco-PUC**. Anais do X Simpósio Brasileiro de Engenharia de Software, 1996, São Carlos. São Carlos : Universidade Federal de São Carlos, 1996. v. I. p. 173-188.

PRIETO-DIAZ, R., ARANGO, G., **Domain Analysis Concepts and Research Directions**. PRIETO-DIAZ, R., ARANGO, G. (eds), *Domain Analysis and Software Systems Modeling*, IEEE Computer Society Press, 1991.

QVT-PARTNERS, **MOF Query/Views/Transformations, Revised Submission**. OMG Document: ad/2003-08-18. 2003. Disponível em: http://tratt.net/laurie/research/publications/papers/qvtpartners1.1.pdf. Acesso em: jun. 2009.

REA, M. L., PARKER, A. R. **Metodologia da pesquisa:** do planejamento à execução. São Paulo, Pioneira, 2000.

REINEHR, S. S. **Reúso Sistematizado de Software e Linhas de Produto de Software no Setor Financeiro**: Estudos de Caso no Brasil. 2008. 327 p. Tese (Doutorado em Engenharia) – Departamento de Engenharia de Produção, Escola Politécnica da Universidade de São Paulo, São Paulo.

SANT'ANNA, M., LEITE, J. C. S. P., PRADO, A.; **A Generative Approach to Componentware**, Proceedings of International Workshop on Component Based Software Engineering; 1998;

SAXE, J. G. **The Blind Men and the Elephant**. Disponível em: <http://www.wordinfo.info/words/index/info/view_unit/1/?letter=B&spage=3>. Acesso em: out. 2008.

SEI, **Domain Engineering: A Model-Based Approach.** Software Engineering Institute, 2007. Disponível em: http://www.sei.cmu.edu/domain-engineering/index.html. Acesso em: jan. 2008.

SHIROTA, Y.; IIZAWA, A.. **Automatic GUI generation from database schema information**. Systems and Computers in Japan, Tokyo, v. 28, n.5 , p. 1-10, dez. 1997.

SIMOS, M., ANTHONY, J., **Weaving the Model Web: A Multi-Modeling Approach to Concepts and Features in Domain Engineering**, Proceedings of the 5th International Conference on Software Reuse (ICSR-5), ACM/IEEE, pp. 94-102, Victoria, Canadá, Junho, 1998.

SINGH, I. et al. **Designing Enterprise Applications with the J2EE Platform**. 2. ed. Prentice Hall PTR, 2002. 352 p.

SOMMERVILLE, **Engenharia de Software**. 8.ed. Addison Wesley, São Paulo, Brasil, 2007, 524 p.

STAA, A. V. **Programação Modular**: Desenvolvendo programas complexos de forma organizada e segura. Rio de Janeiro: Campus. 4. ed., 2000, 690 p.

SUN MICROSYSTEMS, **Java BluePrints – Model-View-Controller**. Disponível em: <http://java.sun.com/blueprints/patterns/MVC-detailed.html>. Acesso em jan. 2008.

SUN MICROSYSTEMS, **Introspection**. Disponível em: <http://java.sun.com/docs/books/tutorial/javabeans/introspection/index.html>. Acesso em jun. 2009.

WARMER, J., KLEPPE, A., BAST, W. **MDA Explained: The Model Driven Architecture**: Pratice and Promise. Addison-Wesley, 2003, 192 p.

WERNER, C., MATTOSO, M., BRAGA R., BARROS, M., MURTA, L., DANTAS, A. **Odyssey**: Infra-estrutura de Reutilização baseada em Modelos de Domínio. *XIII Simpósio Brasileiro de Engenharia de Software, Caderno de Ferramentas*, pp. 17-20, Florianópolis, Outubro, 1999.

WERNER, C., BRAGA, R., **A Engenharia de Domínio e o Desenvolvimento Baseado em Componentes**. GIMENES, I.M.S., HUZITA, E.H.M. (ed.), Desenvolvimento Baseado em Componentes: Conceitos e Técnicas, Rio de Janeiro, Ciência Moderna, 2005.

W3C, **HTTP** - Hypertext Transfer Protocol. Disponível em: http://www.w3.org/Protocols. Acesso em: jun. 2009.

W3C, XML – Extensible Markup Language. Disponível em: http://www.w3.org/XML/. Acesso em: ago. 2009.

ZUSE, H. **A Framework of Software Measurement**. Walter de Gruyter (ed.), Berlim, 1998. 755 p.