Universidade Federal do Rio de Janeiro
Instituto de Matemática
Núcleo de Computação Eletrônica

Integridade e Translação Representacional das Entidades

Dissertação de mestrado

Aluna:
Bárbara de Oliveira Brasil Corrêa

Orientador:
Prof. Dr. Carlo Emmanoel Tolla de Oliveira

Rio de Janeiro
2005

Bárbara de Oliveira Brasil Corrêa

Integridade e Translação Representacional das Entidades

Universidade Federal do Rio de Janeiro
Instituto de Matemática
Núcleo de Computação Eletrônica

Orientador:
Prof. Dr. Carlo Emmanoel Tolla de Oliveira
Ph. D., University College, 1993

Rio de Janeiro
2005

Integridade e Translação Representacional das Entidades

Bárbara de Oliveira Brasil Corrêa

Dissertação submetida ao corpo docente do Instituto de Matemática – IM / Núcleo de Computação Eletrônica – NCE - Universidade Federal do Rio de Janeiro - UFRJ, como parte dos requisitos necessários à obtenção do grau de Mestre.

Aprovada por:

_____

Prof. Dr. Carlo Emmanoel Tolla de Oliveira - Orientador
Ph. D., University College, 1993

_____

Prof. Dr. Eber Assis Schmitz
Ph. D., Imperial College, 1980

_____

Prof. Dr. Ivan Mathias Filho
D. Sc., Pontifícia Universidade Católica do Rio de Janeiro, 2002

Rio de Janeiro
2005

C82448 Corrêa, Bárbara de Oliveira Brasil.
Integridade e Translação Representacional das Entidades / Bárbara de Oliveira Brasil Corrêa. – Rio de Janeiro, 2005.

111 f.: il.

Dissertação (Mestrado em Informática) – Universidade Federal do Rio de Janeiro - UFRJ, Instituto de Matemática / Núcleo de Computação Eletrônica, 2004.

Orientador: Carlo Emmanoel Tolla de Oliveira

1. Arquitetura de Software.- Teses. 2. UML.- Teses. 3. Projeto Hércules - Teses. I. Carlo Emmanoel Tolla de Oliveira (Orient.). II. Universidade Federal do Rio de Janeiro. Instituto de Matemática. Núcleo de Computação Eletrônica. III. Título.

CDD

DEDICATÓRIA

Ele diz que eu sou muito romântica... mas como não ser? Nossa história é linda! E foi construída, sim, com muito, muito amor, mas, sobretudo, com respeito, zelo, compreensão, e muitos outros requisitos que possibilitaram essa linda história.

Você, querido, colore a minha vida e a enche de alegria!

Sei que a conclusão desse trabalho também é uma alegria para você, que, como sempre, compartilha comigo todos os bons e maus momentos.

Muito obrigada por você existir!

AGRADECIMENTOS

Agradeço...

Em primeiro lugar a Deus, nosso pai bondoso que me deu mais essa oportunidade de aprendizado.

Aos meus queridos pais que fizeram essa caminhada até aqui possível, sobretudo com muito amor e dedicação. E ao meu irmãozinho que é super carinhoso comigo.

Ao meu amado marido, que também é o meu melhor amigo, e está sempre ao meu lado me apoiando e dando força, principalmente nos momentos mais difíceis.

Ao meu querido orientador, que acreditou em mim e me ofereceu todo o auxílio para a conclusão desse trabalho.

A minha grande amiga Ana Paula, que esteve sempre presente em todos os momentos com palavras de incentivo e sempre pronta a me ajudar em todos os sentidos.

A querida Tia Deise que teve a maior paciência do mundo comigo.

Aos amigos da equipe SIGA, que trabalharam comigo e de alguma forma contribuíram para este trabalho.

RESUMO

CORRÊA, Bárbara de Oliveira Brasil. **Integridade e Translação Representacional de Entidades.** Orientador: Carlo Emmanoel Tolla de Oliveira. Rio de Janeiro: UFRJ/IM-NCE, 2004. Dissertação (Mestrado em Informática).

Na medida que a complexidade das aplicações aumenta, fica mais difícil evitar que o caos se instale no código de implementação. Os serviços que se entremeiam através das entidades do domínio sofrem de efeitos em cascata em cada etapa do desenvolvimento ou da manutenção. Cada serviço deve ter acesso a uma vista muito específica do domínio para realizar suas tarefas, impondo forma ao domínio segundo seus próprios requisitos. A integridade do domínio deve então ser protegida dos riscos de codificação incidental advindos de várias camadas do sistema. A proteção necessária deve ser provida por uma arquitetura que encapsula o domínio da interferência dos requisitos individuais ao mesmo tempo em que especializa a vista do domínio para cada serviço. Este trabalho propõe uma arquitetura de translado de objetos que permite que as partes individuais da aplicação mantenham sua própria perspectiva do domínio, preservando-o de intrusão mútua. Cópias do domínio, representando o modelo requerido, são geradas e entregues a cada caso de uso. Uma abstração é definida para a tríade MVC, permitindo a programação declarativa dos aspectos do domínio. O resultado é a preservação do domínio da injeção de aspectos dos diversos serviços e a utilização de modelos derivados, específicos para cada serviço. A adaptabilidade do modelo derivado proporciona a automatização de algumas tarefas da aplicação. Este resultado permite que o barramento de translado de objetos de modelo seja gerado automaticamente por técnicas modelagem guiada por arquitetura.

ABSTRACT

CORRÊA, Bárbara de Oliveira Brasil. **Integridade e Translação Representacional de Entidades.** Orientador: Carlo Emmanoel Tolla de Oliveira. Rio de Janeiro: UFRJ/IM-NCE, 2004. Dissertação (Mestrado em Informática).

As applications scale in complexity, the harder it gets to avert chaos from pouring into implementation. Services crosscutting across domain entities suffer from riplle breaking effects at each development or maintenance step. Each service or presentation layer must have access to a very specific view of the domain to accomplish its tasks, thus shaping the domain into its own requirements. Domain integrity must then be protected from effluent coding hazards adjoining from system implementation at various layer. To achieve this, there must be an architecture that encapsulates the domain from interference of individual requirements whilst tailors domain view to each service. This work proposes an object conveyance architecture which enables each individual parts of the application to keep its own perspective of the domain preserving it from mutual intrusion. Proxy clones are defined and delivered to each use case according to required behaviour. A MVC triad abstraction is defined for declarative programming, allowing for automatic generation system backbone through model driven architecture techniques. The result is the domain preservation from the diverse services aspects injection and the use of derived models, specific for each service. The adaptability of the derived model provides the automatization of some tasks of the application. This result allows the automatic generation of the object transfer bus using model driven architecture techniques.

FIGURAS

Figura 1 - Modelos em formatos diferentes 4
Figura 2 - Enfoque da Plataforma Hércules neste trabalho 5
Figura 3 - Barramento de translação 5
Figura 4 – Influências dos serviços sobre o domínio 9
Figura 5 - Modelo dinâmico 10
Figura 6 – Os cegos e o elefante 13
Figura 7 - O elefante dos seis homens 14
Figura 8 - Modelo-Vista-Controle 18
Figura 9 - MVC recursivo 20
Figura 10 - Diagrama de dependência sem injeção de dependência 21
Figura 11 - Diagrama de dependência com o montador 22
Figura 12 - Botões com título extraído de um RDF. 25
Figura 13 - Framework Struts 26
Figura 14 - Zeus, Hera e Olimpo 31
Figura 15 - Tela do serviço Ativar Turmas 35
Figura 16 - Diagrama de estados 37
Figura 17 - Telas resultantes do diagrama de estados 37
Figura 18 - Controlador Hércules 38
Figura 19 - Fábrica de controladores 39
Figura 20 - Serviço de Dados Pessoais do SIGA 40
Figura 21 - Formatos do domínio 42
Figura 22 - Meta-modelo 43
Figura 23 - Representação da aplicação Chronos 44
Figura 24 - Apresentação de Chronos e seu modelo 45
Figura 25 – Parte do domínio de Chronos em Zeus 48
Figura 26 - Parte do domínio do sistema com entidades participantes do caso de uso de Inscrição em Disciplinas 53
Figura 27 - Parte do domínio do caso de uso de Inscrição em Disciplinas 53
Figura 28 - Diagrama de classes da descrição do domínio de Hera 56
Figura 29 - Construção do domínio do caso de uso de Inscrição em Massa 57
Figura 30 - Container de componentes de Hera 61
Figura 31 - Etapas de marcação e aplicação de cosméticos. 62
Figura 33 - Seleção da Previsão de Turmas 65
Figura 34 - Lista de resultados da Previsão de Turmas 66
Figura 35 – Modelo de persistência de Inscrição em Massa 71
Figura 36 - Tela Edição de Plano de Estudos 72
Figura 37- Modelo para apresentação da informação no Struts 73
Figura 38 - FormBean que traz os dados da requisição para a edição de Plano de Estudos 74
Figura 39 - Modelo inicial da Inscrição em Massa em Hera 75
Figura 40 - Modelo com os atributos necessários à apresentação 76
Figura 41 - Entidades complementares para o modelo de Zeus 77

LISTA DE ABREVIATURAS

API	Application Public Interface
DOM	Document Object Model
EJB	Enterprise Java Beans
GUI	Graphical User Interface
HTML	Hyper Text Markup Language
IDE	Integrated Development Environment
J2EE	Java 2 Platform, Enterprise Edition
JDOM	Java Document Object Model
JSP	Java Server Pages
MVC	Model-View-Controller
SIGA	Sistema Integrado de Gestão Acadêmica
UFRJ	Universidade Federal do Rio de Janeiro
UML	Unified Modeling Language
URI	Uniform Resource Identifier
URL	Uniform Resource Locator
XML	Extensible Markup Language
XUL	XML User Interface Language

SUMÁRIO

1 INTRODUÇÃO 1

1.1 Influências sobre o domínio de uma aplicação 1
1.2 O caos instalado 2
1.3 Controle das forças 3

2 REPRESENTAÇÃO DA INFORMAÇÃO 7

2.1 Modelo da Informação 8
2.2 Domínio do problema 10
2.3 Translação do Domínio 11
2.3.1 Conhecimento empírico 12
2.3.2 Modelo da Informação para persistência 14
2.3.3 Modelo da Informação para apresentação 15
2.3.4 Modelo da Informação para controle 16
2.4 Estilo Arquitetônico Modelo-Vista-Controle 17
2.5 MVC Recursivo 19
2.6 Injeção de dependências 20
2.7 Resource Description Framework (RDF) 23
2.8 Apache Struts 25

3 PLATAFORMA HÉRCULES E O DOMÍNIO EM ZEUS 29

3.1 *Framework* Hércules em Zeus 31
3.2 Zeus: camada de vista 32
3.2.1 Apresentação Zeus 33
3.2.2 Controle Zeus 36
3.2.3 Modelo em Zeus 39
3.3 Representando o domínio em Zeus 41
3.3.1 Especificação do domínio 42
3.3.2 Estruturação do domínio 44
3.3.3 Comportamento no domínio 46
3.3.4 Representação do domínio 47

4 O DOMÍNIO E SUA REPRESENTAÇÃO EM HERA 50

4.1 Face do Domínio em Hera 52
4.2 Construindo o domínio do serviço 54
4.3 Programação orientada a componentes 59
4.4 A camada de modelo em Hera 60
4.5 Caracterização do Modelo 62
4.6 Cosméticos 63
4.6.1 Pesquisa e apresentação de resultado 64
4.6.2 Restrições de acesso 68

5 AVALIAÇÃO NO ESTUDO DE CASO: INSCRIÇÃO EM MASSA 70

5.1 Vantagens do framework Hercules em relação ao Struts 72
5.2 Melhorias observadas em Hera 75

6 CONCLUSÃO E TRABALHOS FUTUROS 81

REFERÊNCIAS BIBLIOGRÁFICAS 85

1 INTRODUÇÃO

1.1 Influências sobre o domínio de uma aplicação

O domínio é um dos elementos fundamentais em um sistema de software. Assim como o sangue percorre o corpo levando nutrientes, o domínio leva informação a todas as partes de uma aplicação. O domínio de uma aplicação consiste na captura de 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. O domínio está difundido em todas partes do sistema.

Em sistemas de software de pequeno porte existe uma tendência à concentração de operações em um mesmo conjunto de entidades. Como conseqüência o domínio da aplicação é obrigado a se apresentar em um formato que atenda a todas as operações ao mesmo tempo. À medida que os sistemas se tornam mais complexos essa abordagem passa a representar uma ameaça à harmonia do sistema, pois mais tempo e esforço podem ser empregados para lidar com o tratamento diferenciado do domínio do que com o problema original.

Modificações no domínio da aplicação atrapalham o processo de desenvolvimento quando mais de um serviço interage com a mesma parte do domínio. Os serviços podem ser considerados como forças independentes influenciando um objeto em comum, neste caso o domínio da aplicação, gerando um desenvolvimento caótico. Pequenas alterações na aplicação tomam grandes proporções, tornando difícil prever a conseqüência de uma intervenção e despendendo muito trabalho para contornar os eventuais erros gerados. A perda do controle das forças atuantes prejudica a integridade do domínio que fica deformado pela concentração de operações, levando a uma implementação engessada e dificultando posteriores alterações.

Em aplicações de grande porte cresce o número de entidades e relacionamentos. Cresce também a necessidade de controle sobre essas entidades. Porém cada serviço enxerga o domínio de uma maneira particular, onde algumas informações são importantes e outras não. Para adaptar o domínio ao seu contexto, um serviço molda as entidades impondo formatos que outros serviços, que compartilham as mesmas entidades, são obrigadas a aceitar. Por sua vez, os serviços são coagidos a se adaptarem ao domínio da aplicação. Esse efeito cascata contribui para uma poluição no cenário do serviço uma vez que algumas associações ou até mesmo atributos de uma determinada entidade podem ser irrelevantes ao serviço.

1.2 O caos instalado

Manter, expandir ou até mesmo testar uma aplicação, onde operações influenciam o domínio da aplicação sem qualquer controle, são tarefas desafiadoras. O domínio, que deveria ser claro e enxuto, fica distorcido e sobrecarregado ao tentar atender às diversas forças do sistema. Os serviços, obrigados a se adaptarem ao domínio distorcido, perdem sua identidade.

É difícil identificar quais serviços interagem com uma determinada parte do domínio, por isso uma alteração em um serviço pode interferir em muitos outros. As regras de negócio, que deveriam estar isoladas, ficam misturadas a código de formatação do domínio dificultando o trabalho do programador e a realização de testes. A integridade do domínio fica comprometida, pois este absorve as influências dos diversos serviços. Informações colocadas indiscriminadamente sem padronização prejudicam a flexibilidade e manutenibilidade do sistema levando a uma implementação caótica onde a desordem e a imprevisibilidade prevalecem.

À medida que as aplicações se tornam mais complexas fica ainda mais difícil evitar a deformação do domínio e conseqüente surgimento de código reparador para os erros provocados por alterações na aplicação. O surgimento de novos serviços provoca interferência nos serviços pré-existentes que interagem com a mesma parte do domínio.

Muitas operações sobre o domínio se repetem de uma aplicação para outra. Tarefas como transladar o domínio para apresentação ou transladá-lo para a persistência

poderiam ser reaproveitadas poupando tempo e evitando erros. Contudo a confusão gerada pela influência desorganizada das forças que atuam sobre o domínio inviabiliza o reaproveitamento entre aplicações. As responsabilidades ficam indefinidas complicando o isolamento de partes do sistema como componentes.

1.3 Controle das forças

Sistemas de software complexos, onde a influência de forças sobre o domínio dá origem a tantos problemas, carecem de uma abordagem diferente. Este trabalho não pretende isolar o domínio das forças que o influenciam, pois domínio e forças estão inseridos em um sistema que precisa estar em equilíbrio. Esse equilíbrio será dado pelo controle de forças e preparação do domínio para uma determinada finalidade.

Cada serviço do sistema visualiza o domínio de uma forma particular. O domínio deve ser preparado para ser visualizado por um serviço, ou seja, o serviço somente visualizará a informação que for imprescindível para seu funcionamento. Todavia o domínio deve ser capaz de representar a informação da maneira mais adequada ao serviço. O domínio utilizado em um serviço poderá até ser utilizado por outro serviço se não houver necessidade de alterá-lo. Visualizações diferentes requerem domínios diferentes.

Serviços distintos utilizando o mesmo domínio devem ser gerenciados. O gerenciamento torna possível a configuração das forças que atuam sobre o domínio. Adicionar ou remover um serviço não deverá causar nenhuma modificação nos serviços já existentes ou no domínio. Operações que não exijam modificações no meio em que estão inseridas podem ser vistas como componentes, podendo ser reutilizadas em outra aplicação.

Na intenção de atingir esses objetivos, esse trabalho propõe uma arquitetura para preservar a integridade do domínio da aplicação criando domínios particulares para cada serviço. Os domínios particulares são transladados de uma parte a outra do sistema. Além disso, provê a configuração de serviços que atuam sobre o mesmo domínio.

A arquitetura proposta neste trabalho é uma parte da Plataforma Hércules para o adequado tratamento do domínio de uma aplicação. A Plataforma Hércules foi desenvolvida para facilitar a construção de sistemas de informação sofisticados, que demandam a execução de diversos casos de uso. Essa plataforma provê uma padronização no desenvolvimento, favorecendo a redução de erros e agilizando todo o processo. Tal padronização é obtida com a utilização de arquiteturas que constituem a plataforma em conjunto com engenhos de geração automática.

A Plataforma Hércules utiliza o conceito de MVC recursivo (Pais, 2004). Uma aplicação que utiliza o estilo arquitetural MVC é dividida em modelo, vista e controle. O conceito de MVC recursivo permite que cada uma dessas partes possa ser subdividida em modelo, vista e controle novamente. O domínio da aplicação está espalhado por essas partes e precisa estar adequado a cada uma delas.

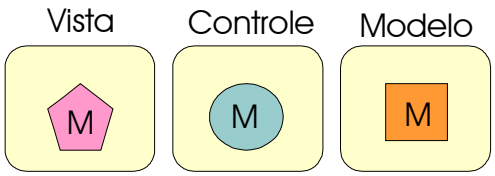


Figura 1 - Modelos em formatos diferentes

Ana Paula Pais em seu trabalho de dissertação (PAIS, 2004) apresentou uma solução para a vista da Plataforma Hércules, com enfoque no controle do caso de uso. Complementando a Plataforma Hércules, este trabalho fornece uma padronização para a construção do domínio da aplicação. O objetivo deste trabalho está nos modelos internos da tríade MVC externa, especificamente nos modelos residentes na vista e no controle, como mostra a Figura 2. Também oferece mecanismos para configurar serviços sobre o domínio. Muitas operações, uma vez configuradas, ficaram transparentes para os programadores. Contudo, a arquitetura proposta para a representação do domínio introduz maior complexidade aumentando a quantidade de código. Para facilitar o desenvolvimento, é sugerido, como trabalho futuro, a construção de uma ferramenta de geração automática. A geração automática se baseia no conceito de MDA (Kleppe et al, 2003), definindo perfis para aplicações sobre a Plataforma Hércules. A partir de uma modelagem

UML é possível gerar parte do código necessário para a implementação de um caso de uso.

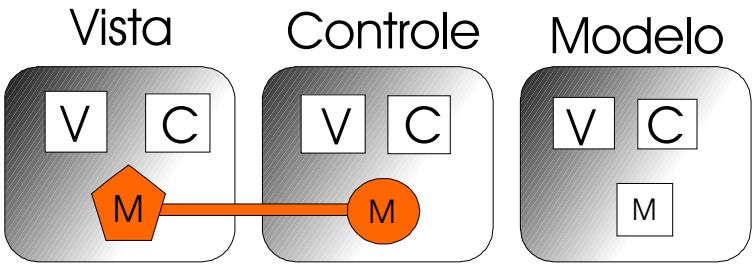


Figura 2 - Enfoque da Plataforma Hércules neste trabalho

O emprego de uma arquitetura capaz de transladar o domínio padroniza o desenvolvimento de casos de uso, preservando o domínio da aplicação e torna transparente e configurável algumas operações. Há um aumento da possibilidade de reuso e manutenibilidade do sistema. A geração automática reduz o tempo de desenvolvimento e reduz o aparecimento de erros. Além disso, incentiva a descoberta e documentação de novos perfis de aplicação.

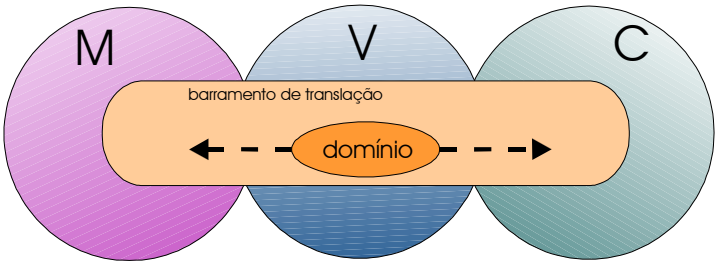


Figura 3 - Barramento de translação

Este trabalho está organizado em seis capítulos. O primeiro deles corresponde a esta introdução, que exibe a motivação e o objetivo deste trabalho, além de sua organização.

O segundo capítulo contextualiza “Modelo da Informação” e “Domínio do Problema” e apresenta visões da informação nas partes de um sistema. Além disso, apresenta uma revisão de algumas das principais arquiteturas, tecnologias e frameworks que

têm sido utilizadas na construção dos sistemas de informação, servindo de base para o desenvolvimento deste trabalho.

O terceiro capítulo apresenta, em detalhe, a proposta deste trabalho para a representação da informação na camada de vista. Este capítulo mostra como o modelo proposto atende as necessidades apontadas no capítulo dois.

O quarto capítulo apresenta, em detalhe, a proposta deste trabalho para a representação da informação na camada de controle. Este capítulo mostra como a preservação da informação pode auxiliar na construção de engenhos para automatizar a translação do domínio e como a injeção de dependência pode auxiliar a adição de informação extra no modelo.

O quinto capítulo apresenta a avaliação deste trabalho, apresentando seus pontos positivos e negativos. Duas comparações são feitas: uma em relação ao framework STRUTS e outra da camada de controle do framework Hércules sem a contribuição deste trabalho.

O sexto capítulo corresponde à conclusão. Esse capítulo apresenta uma visão geral deste trabalho, assim como suas contribuições e perspectivas futuras.

2 REPRESENTAÇÃO DA INFORMAÇÃO

Sistemas de software lidam com informação a todo o momento. Ora apresentam a informação, ora atribuem alterações, sempre aplicando regras sobre a informação. Em sistemas que seguem o paradigma de Orientação a Objetos essa informação é representada por um conjunto de objetos e relacionamentos. Existem algumas técnicas para organizar essa informação, como por exemplo, a modelagem relacional, a modelagem UML, a separação da aplicação em camadas, etc. Porém, muitas vezes, essas representações não oferecem toda informação necessária a uma serviço. Surgem informações complementares e fora de contexto para suprir essa carência. A pobreza na representação prejudica a interação do sistema com a informação, à medida que esta é acessada em situações diferentes. Com essa profusão de informações se torna difícil identificar o domínio e o papel dele em cada parte da aplicação.

O domínio da aplicação deve estar claro para os desenvolvedores poderem implementar as operações. No momento da implementação de um caso de uso, o desenvolvedor precisa ter acesso à informação contida na especificação deste. Informações que surgem durante a implementação devem poder ser adicionadas sem comprometer o a representação do domínio. O objetivo é fazer com que o desenvolvedor se concentre apenas nas regras de negócio. Contudo, nem sempre a representação da informação utilizada pelo desenvolvedor é apropriada para um determinado serviço. A representação adequada é importante para que o serviço possa interagir melhor com a informação. Todas as informações importantes para o desenvolvimento e funcionamento da aplicação devem estar inseridas dentro de um contexto apropriado.

No momento de desenvolvimento de um caso de uso, pode surgir a necessidade de criar entidades, relacionamentos e atributos, antes ignorados. Informações inseridas de forma desorganizada em decorrência da ausência de uma estrutura capaz de comportá-las polui o cenário de desenvolvimento. O aumento do poder

representacional do domínio permite que as informações fiquem concentradas num contexto harmônico. O domínio resultante é uma combinação do domínio já conhecido do sistema com um domínio intrínseco da camada em que está sendo trabalhado.

Operações, tais como apresentação das informações ao usuário ou mecanismos de persistência, são isoladas e utilizam distintas representações do domínio. Disponibilizar diferentes visões do domínio de acordo com o serviço facilita a manutenção e posterior extensão da aplicação. O domínio da aplicação claro e bem representado facilita o desenvolvimento, uma vez que o desenvolvedor sabe exatamente de onde tirar as informações para a implementação de um caso de uso.

A melhor representação para a informação depende da operação a ser realizada. É fundamental saber o papel que o domínio vai desempenhar a cada momento para propor uma representação adequada. O papel do domínio não muda apenas de um caso de uso para outro, muda também entre camadas. Um domínio contendo determinadas informações pode ser transladado para outra camada preservando essas informações em um outro formato. Por exemplo, um objeto *Pessoa* com seus atributos é explodido em uma coleção de campos na GUI.

2.1 Modelo da Informação

Modelo é o termo utilizado para nomear um dos elementos da tríade MVC. No entanto, a camada modelo não deve ser entendida simplesmente como informação, mas como formato da informação, ou então *modelo da informação*. O modelo da informação deve responder a seguinte pergunta: O q*ue se precisa saber, para se fazer o que se faz*?

Modelar é dar forma à informação dando origem a um modelo. Modelo é a forma, e a informação é a essência. Platão acreditava que o mundo físico e os objetos ao nosso redor de nós são meras cópias da substância real, ou essência. Ele dizia que o aquilo que percebemos por meio de nossos sentidos é apenas uma cópia da forma real. Talvez porque os nossos sentidos são os meios que dispomos para absorver determinada informação. Assim também é em um sistema de software, cuja informação pretendida encontra-se sob um formato. Este formato depende de como

as partes irão interagir com essa informação, ou seja, que meios o sistema dispõe para acessar uma determinada informação.

A informação, vista como essência, pode conceber diversas formas dependendo do meio em que está inserida. Mais de um modelo pode ser necessário para representar a mesma informação em diferentes partes do sistema. A finalidade para qual uma determinada informação está sendo utilizada é que irá ditar o formato mais adequado.

Quanto mais próxima a forma estiver da realidade melhor será o entendimento do modelo. O paradigma de orientação a objetos proporciona uma representação da informação mais próxima do mundo real, onde objetos encapsulam dados e comportamentos. Em sistemas orientados a objetos, o modelo da informação é geralmente representado por entidades, atributos e relacionamentos. Para ilustrar, considere um sistema de gestão acadêmica. Um serviço de previsão de turmas pode ter como modelo uma turma com seus horários e locais. No mesmo sistema, um serviço de inscrição em disciplinas pode ter como modelo uma turma com alunos inscritos. Em cada caso a informação sobre turma é vista em formatos diferentes (Figura 4).

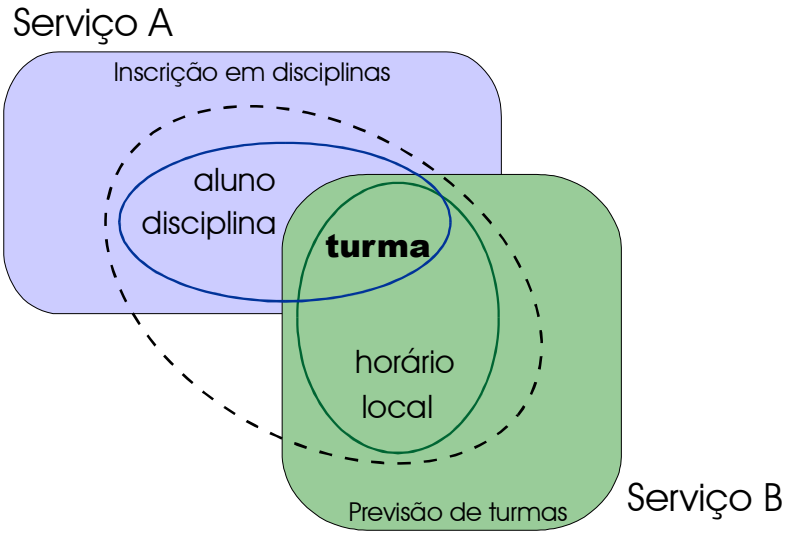


Figura 4 – Influências dos serviços sobre o domínio

A camada de modelo da tríade MVC tem como função primeira apresentar a informação em uma forma que seja apropriada dentro do seu contexto. Além disso, de acordo com a complexidade do sistema, esta camada pode oferecer mecanismos para gerenciar o comportamento e o conteúdo do modelo da informação. A camada de modelo recebe mensagens do controle e seu estado é observado pela vista.

No MVC recursivo a função da camada de modelo não se altera. No entanto, devido à mudança de contexto, o modelo da informação sofre alterações. No sistema como um todo, o modelo da informação é um modelo dinâmico, que muda dependendo do contexto em que está inserido. Os modelo são representações para as variações das características comportamentais do ambiente ou sistema. Uma questão deve ser respondida: Q*uando é necessário saber, o que se precisa saber, para se fazer o que se faz*?

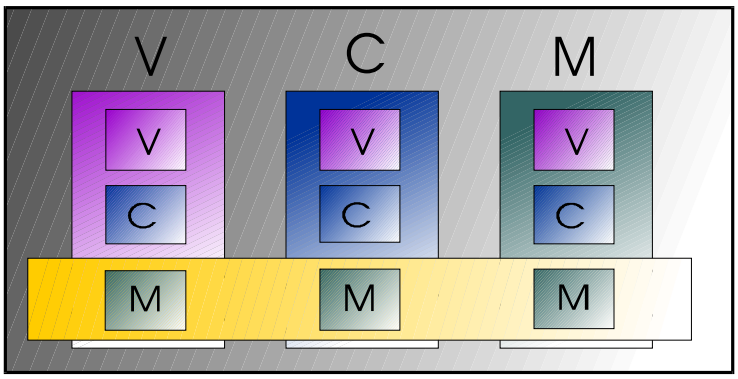


Figura 5 - Modelo dinâmico

2.2 Domínio do problema

Entre as definições do termo domínio está a palavra *conhecimento*. Para a matemática, domínio significa *totalidade dos pontos onde uma função é definida*. Há uma correspondência entre essas definições e o significado de domínio em relação à expressão *domínio do problema* para a ciência da computação. Domínio do problema é o conhecimento do universo de informações e regras de um problema que se pretende solucionar com um sistema de software.

O domínio do problema compreende a essência da informação. Além disso, regras de negócio que regem a informação obtida com o estudo do problema também

fazem parte do domínio. Por exemplo, em um sistema de gestão acadêmica, uma turma tem alunos inscritos. A turma e os alunos são dados, valores. Porém, neste exemplo, uma turma pode ter no máximo 20 alunos. Quando o vigésimo aluno se inscreve a turma fica fechada. Isso é uma regra que impõe um controle sobre a quantidade de alunos inscritos. Esta regra também faz parte do domínio do problema.

O domínio de um problema deve ser bem representado e bem compreendido. A partir de análise dos requisitos junto ao cliente o domínio do problema vai sendo traçado. Nem todo o conhecimento é obtido em primeira instância, abrindo a possibilidade de que mais regras e informações sejam adicionas posteriormente. O modelo da informação é responsável por representar uma parte desse domínio e deve ser capaz de absorver alterações no entendimento do domínio do problema. Algumas das regras são representadas por elementos de controle, outras pelo próprio modelo. As regras absorvidas pelo modelo são aquelas inerentes à natureza dos dados. Ainda no exemplo do sistema de gestão acadêmica, os alunos só podem fazer inscrição em turmas em um determinado período do ano letivo. Este é um caso em que o controle está fora do escopo do modelo, pois essa regra não controla uma entidade e sim um serviço. No entanto, toda turma precisa ter um nome, o que representa um controle da própria natureza de uma turma.

Um domínio bem conhecido é fundamental para o bom funcionamento do sistema, ou seja, é preciso conhecer aquilo que se pretende resolver. No entanto, muitas vezes o problema não é bem compreendido nem pelo cliente, gerando alterações no domínio do problema. Com uma boa representação do domínio do problema, eventuais alterações se tornam mais simples de serem realizadas.

2.3 Translação do Domínio

Sistemas de grande porte gerenciam um conjunto volumoso de informações. Tratar toda essa informação do mesmo jeito em todas as partes do sistema contribui para que as funcionalidades fiquem com responsabilidades excessivas para obter informação de modelos inapropriados. O domínio, na tentativa de atender todas as funcionalidades, fica deformado. O conhecimento do domínio do problema em cada

caso de uso permite a definição de um domínio enxuto e objetivo. O estudo do fluxo da informação no sistema possibilita transladar o domínio em formatos mais apropriados para cada camada deste sistema.

As perguntas: O q*ue se precisa saber, para se fazer o que se faz*? E Q*uando é necessário saber, o que se precisa saber, para se fazer o que se faz*? São respondidas analisando os requisitos de um caso de uso e os momentos de em que a informação precisa ser acessada.

Mesmo quando o domínio do problema extraído de um caso de uso parece claro, o domínio ainda sofre algumas alterações no momento de implementação. As formas em que esse domínio será visualizado dependerá daquilo que cada parte do sistema quer ver e de como quer ver. O modelo da informação surge da relação entre o observador (funcionalidade) e o observado (domínio).

2.3.1 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 parte do sistema sente a informação de maneira diferente. O resultado é uma visão do domínio mais apropriada a cada uma delas.

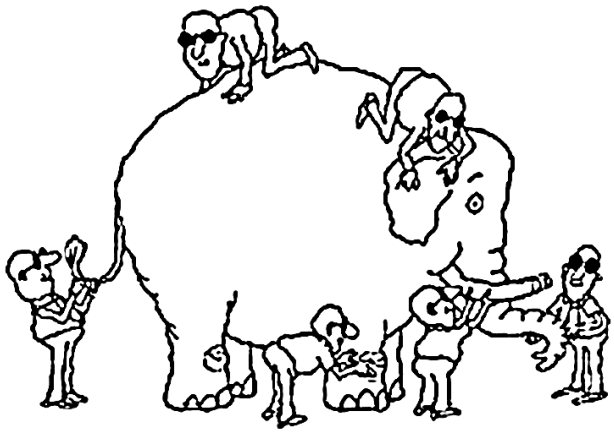


Figura 6 – Os cegos e o elefante

O poema (SAXE, 2005) abaixo, de autoria John Godfrey Saxe, é utilizado para ilustrar interpretações diferentes em relação a um objeto. O poema é baseado em um antigo conto indiano.

OS CEGOS E O ELEFANTE

Eram seis homens do Hindustão
Inclinados para aprender muito,
Que foram ver o Elefante
(Embora todos fossem cegos)
Que cada um, por observação,
Poderia satisfazer sua mente.

O Primeiro aproximou-se do Elefante,
E aconteceu de chocar-se
Contra seu amplo e forte lado
Imediatamente começou a gritar:
"Deus me abençoe, mas o Elefante
É semelhante a um muro",
O Segundo, pegando na presa,
Gritou, "Ohl O que temos aqui
Tão redondo, liso e pontiagudo?
Para mim isto é muito claro
Esta maravilha de elefante
É muito semelhante a uma lança"

O Terceiro aproximou-se do animal
E aconteceu de pegar
A sinuosa tromba com suas mãos.
Assim, falou em voz alta:
"vejo", disse ele, "o Elefante
E muito parecido com uma cobra!"

O Quarto esticou a mão, ansioso
E apalpou em torno do joelho.
"Com o que este maravilhoso animal
Se parece é muito fácil", disse ele:
"Está bem claro que o Elefante
É muito semelhante a uma árvore"

O Quinto, por acaso, tocou a orelha,
E disse: "Até um cego
Pode dizer com o que ele se parece:
Negue quem puder,
Esta maravilha de Elefante
É muito parecido com um leque!"

O Sexto, mal havia começado
A apalpar o animal,
Pegou na cauda que balançava
E veio ao seu alcance. "Vejo", disse ele,
"o Elefante
é muito semelhante a uma corda!"

E assim esses homens do Hindustão
Discutiram por muito tempo,
Cada um com sua opinião,
Excessivamente rígida e forte.
Embora cada um estivesse, em parte,
certo,
Todos estavam errados!

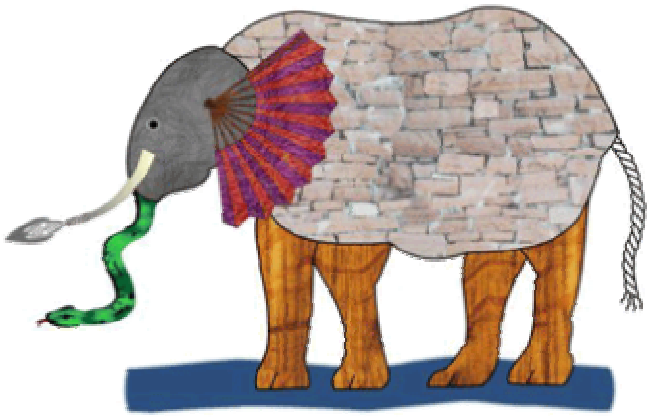


Figura 7 - O elefante dos seis homens

Os cegos, comunicando suas impressões, chegaram a "compor" a imagem de elefante. Porém, nenhum dos pontos de vista individuais dava conta de uma descrição "verdadeira" de um elefante. Mas o que realmente importa é que cada um ficou satisfeito com a sua interpretação. Em um sistema de software também é assim; o importante é que cada parte do sistema fique satisfeita com a sua interpretação (ou visão) do domínio.

2.3.2 Modelo da Informação para persistência

Atualmente o projetista de um sistema de informação tem a sua disposição uma variada gama de opções de mecanismos de persistência. Entre eles estão mecanismos que utilizam banco de dados e os que armazenam objetos na memória, como, por exemplo, o Prevayler (SANTOS et al, 2005). Cada mecanismo possui suas particularidades que são incompatíveis entre si. Alterar de um mecanismo para outro é uma tarefa complicada quando o sistema se encontra numa fase mais avançada de desenvolvimento.

Normalmente esses mecanismos de persistência oferecem controles de transações, meio para realizar consultas com linguagens específicas (exemplo SQL), etc. O domínio da camada de modelo deve incorporar informações específicas para estas operações de persistência. O modelo utilizado para representar esse domínio deve

prover os meios necessários para ser manipulado pelos engenhos responsáveis pelas operações de persistência.

A criação de um modelo específico para a camada de persistência poderia prover uma padronização para o armazenamento da informação do sistema. Um modelo adequado inserido numa arquitetura capaz de lidar com os diversos meios de persistência de forma transparente facilitaria o trabalho do programador. Estudos mais aprofundados sobre o modelo de persistência farão parte de trabalhos futuros do *framework* Hércules.

A partir do estudo de perfis de serviços, as entidades podem ser modeladas para formar o domínio necessário para a camada de modelo externa do MVC recursivo. A proposta dessa camada é oferecer uma arquitetura capaz persistir a informação do sistema utilizando mecanismos de persistência configuráveis. Mais de um mecanismo poderá ser utilizado no mesmo sistema. Além disso, o mecanismo de persistência de todo um sistema poderá ser trocado por outro com um mínimo (ou talvez nenhum) esforço do programador.

2.3.3 Modelo da Informação para apresentação

Geralmente é pela apresentação que o usuário expõe e entende as funcionalidades da aplicação. O modelo que surge na especificação de vista nem sempre é o mesmo utilizado para a aplicação das regras de negócio do caso de uso. A apresentação geralmente precisa de informações auxiliares como pesquisas; conjunto de dados para preencher componentes visuais como, por exemplo, caixas de seleção; textos meramente informativos; etc.

O modelo da camada de vista externa do MVC recursivo deve ser capaz de comportar todas as informações necessárias à apresentação, além de ser sensível à interação do usuário. Este modelo, de acordo com o estilo arquitetônico MVC, não deve ter conhecimento de vista. Porém ele deve prover os meios necessários para a comunicação com os componentes de vista. Os elementos de modelo devem registrar os interessados na alteração de seu conteúdo para serem notificados quando estas ocorrerem.

Uma implementação inteligente da camada de modelo da camada de vista pode dispensar a intervenção desnecessária da camada de controle externa. Operações simples podem ser resolvidas na própria camada de vista. Clone de elementos de coleção e simples mudança de tela (conjunto de componentes visíveis pelo usuário) são exemplos de operações que podem ser resolvidas na própria camada de vista.

O modelo da camada de vista deve possibilitar também a renderização em diferentes tipos de interface gráfica. Com a utilização de um modelo próprio, a camada de vista pode comportar implementações para apresentação em diversas interfaces. A implementação de componentes visuais específicos para a interface desejada seria suficiente para permitir ao projetista escolher a renderização mais conveniente para apresentação.

2.3.4 Modelo da Informação para controle

A camada de controle é a camada que provavelmente apresentará o domínio mais próximo do mundo real. O programador terá acesso a esse modelo para a aplicação de regras de negócio. Por isso, o modelo da camada de controle deve ser simples e de fácil compreensão para o programador, permitindo agilidade na implementação dos casos de uso.

Ao mesmo tempo em que a camada de controle oferece ao programador um modelo enxuto, ela deve manter informações complementares transparentes. Algumas operações podem ser configuradas para atuar sobre o modelo sem a intervenção do programador. Essas operações geralmente requerem informações adicionais para serem executadas.

A translação do domínio é um exemplo de operação automatizada. A passagem do domínio da camada de controle para a camada de vista freqüentemente requer informações auxiliares como, por exemplo, visibilidade de elementos do modelo. Quando o domínio chega da apresentação à camada de controle pode ocorrer a necessidade de registrar alguma informação de controle proveniente da interação do usuário. Um exemplo é a identificação de elementos que foram apagados (excluídos) pelo usuário para posterior validação pela camada de controle.

É importante também manter o conhecimento de construção do domínio da camada de controle a partir do domínio da camada de modelo para possibilitar a automatização de algumas operações relacionada a armazenagem e recuperação de dados. Esta informação será fundamental para a translação do domínio para a camada de modelo.

2.4 Estilo Arquitetônico Modelo-Vista-Controle

Os projetistas de sistemas de informação estão constantemente buscando técnicas para auxiliar no desenvolvimento dos sistemas. Soluções que reduzam o acoplamento e facilitem o desenvolvimento são bem-vindas, mesmo quando há um aumento no número de classes no sistema. Acoplamento caracteriza o relacionamento de um módulo com outros módulos, isto é, ele mede a interdependência entre dois módulos. Quanto mais conexões entre módulos, mais dependentes eles estão (YOURDON, 1979). Através de objetos com responsabilidades específicas é possível obter maior flexibilidade na construção dos sistemas. O estilo arquitetônico Modelo-Vista-Controle (MVC) (BURBECK, 2005) sugere uma separação da aplicação em dados que representam o mundo real, sua apresentação e controle.

O estilo arquitetônico MVC teve sua origem no *Smalltalk* e vem sendo amplamente utilizado no desenvolvimento de sistemas de informação. No MVC, o controle, vista e modelo estão explicitamente separados e são gerenciados por três tipos de objetos, cada qual especializado em sua tarefa. O modelo, vista e controle envolvidos na tríade MVC precisam se comunicar entre si para a aplicação gerenciar a interação com o usuário. A comunicação entre a vista e seu controle é direta, pois Vista e Controle foram desenvolvidos para trabalharem juntos. Por outro lado, a comunicação entre vista e modelo é mais sutil. O modelo apenas recebe notificações da vista. A Figura 8 mostra as responsabilidades de cada elemento da tríade.

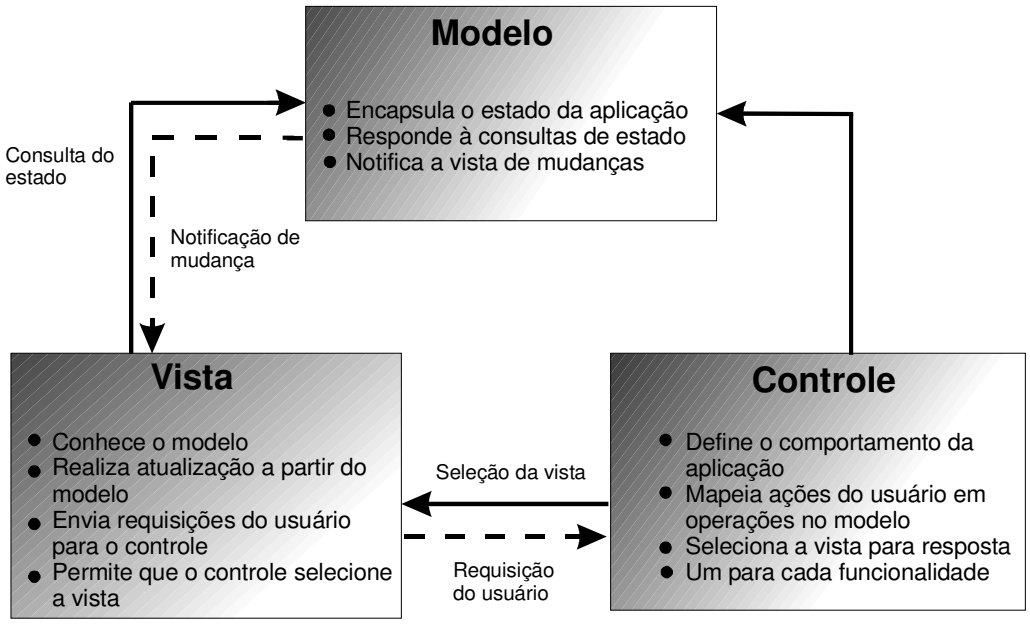


Figura 8 - Modelo-Vista-Controle

Vista. Responsável pela apresentação do modelo ao usuário. Notifica o modelo de mudanças ocorridas pela interação do usuário. É diretamente alterada pelo controle.

Controle. O controlador recebe as requisições do usuário, executa operações e decide que estado do caso de uso será apresentado ao usuário. O controlador também gerencia os dados do modelo.

Modelo. O modelo armazena a informação que é apresentada ao usuário. Não conhece a vista nem o controle. É responsável por informar, indiretamente, a vista de mudanças no seu conteúdo para que ela possa se atualizar. O controle é capaz de intervir no modelo alterando ou requisitando mudanças no seu conteúdo. Em ambos os casos o modelo é passivo, apenas sofrendo a manipulação por parte do controle e da vista.

A utilização do estilo arquitetônico MVC divide o sistema em camadas definindo as responsabilidades de cada uma delas. Essa divisão contribui para o desacoplamento das funções de apresentação, controle e representação do conteúdo apresentado ao usuário. Com as camadas separadas segundo o MVC se torna mais simples a alterar ou mesma a substituir a implementação de alguma camada. Por exemplo, a

apresentação feita em Swing pode ser alterada por uma apresentação Web sem que seja preciso modificar as camadas de modelo e controle.

2.5 MVC Recursivo

O padrão de projeto MVC está sendo bastante utilizado na construção de frameworks no mercado de software. Como exemplo o STRUTS (CAVANESS, 2004) e o SPRING (JOHNSON, 2005) utilizam MVC para separar um sistema em três camadas: modelo, vista e controle. Com a divisão do sistema em camadas as responsabilidades ficam bem melhor definidas facilitando o desenvolvimento. Em sistemas de grande porte, onde modificações e adição de novas funcionalidades ocorrem com freqüência, essa simples separação em três camadas se torna frágil para comportar a complexidade do sistema. Uma camada muitas vezes precisa conhecer detalhes das outras duas camadas, o que gera uma dependência indesejável.

Sistemas mais sofisticados requerem uma arquitetura mais robusta que suporte alterações sem grandes problemas. Esta arquitetura deveria oferecer maior flexibilidade permitindo que partes da aplicação fossem vistas como componentes passíveis de substituição. O aumento de flexibilidade é um dos benefícios do emprego do padrão de projeto MVC. Aumentar a granulosidade da arquitetura organiza melhor as responsabilidades formando blocos com tarefas bem definidas sem necessidade de interferência de outras camadas.

O conceito de MVC recursivo (PAIS, 2004, p.52) introduz uma divisão mais profunda da aplicação, onde cada camada do MVC original pode ser subdividida em modelo, vista e controle novamente. As subcamadas que surgem têm a mesma finalidade das camadas do MVC original, porém estão relacionadas ao contexto onde estão inseridas. Em cada camada exterior a informação é visualizada e tratada de maneira diferente. As subcamadas surgem de acordo com a necessidade. Uma subcamada pode ser eventualmente omitida em uma determinada camada caso não se justifique sua existência.

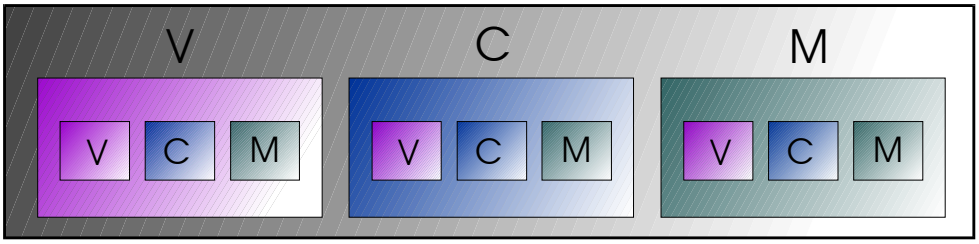


Figura 9 - MVC recursivo

O Modelo, onde quer que esteja, é responsável pela representação da informação. No entanto a representação da informação pode mudar de acordo com a finalidade, dando origem a mais de uma representação para a mesma informação. Na camada de vista a subcamada de modelo reúne informações para apresentação ao usuário. As regras de negócio são aplicadas sobre a subcamada de modelo da camada de controle. A subcamada de modelo da camada de modelo representa as informações no formato que estas serão persistidas e compartilhadas com todas as operações da aplicação.

Quanto mais complexa é uma aplicação, mais benefícios são obtidos pelo aumento de granulosidade. A classificação das responsabilidades facilita a manutenção da aplicação, pois define pontos de intervenção de acordo com sua natureza, desestimulando, por exemplo, codificação de regras de negócio em uma natureza de apresentação. A aplicação do MVC recursivo possibilita o aumento de granulosidade sem perder os conceitos já disseminados de modelo, vista e controle. As camadas se tornam mais autônomas dispensando a interferência das outras camadas. Com o MVC recursivo o código fica mais padronizado possibilitando futuras automatizações.

2.6 Injeção de dependências

À medida que as aplicações se tornam mais volumosas, há uma tendência a fragmentação da aplicação em pequenas partes para executarem tarefas bem específicas. O desenvolvimento de componentes que ofereçam serviços simples vem crescendo e com eles cresce também o potencial de reutilização entre aplicações. Os componentes são combinados para dar origem a uma aplicação coesa. No entanto, a maneira como esses componentes se integram pode gerar

dependências indesejáveis. O padrão de projeto Injeção de Dependências apresenta soluções para evitar tais problemas. Surgiram no mercado de software alguns *containers* que adotam esse padrão para fazer a ligação entre os componentes.

Injeção de Dependências é uma estratégia de tirar o controle de dependências de dentro para fora das classes. Ao invés de uma classe instanciar internamente cada objeto que necessita, ela simplesmente recebe as referências. Os objetos necessários são passados a ela. Também é muito comum encontrar na literatura essa estratégia com o nome de Inversão de Controle (Inversion of Control, ou IoC). Para Fowler (2005), o termo Inversão de Controle é genérico demais e deixa as pessoas confusas, por isso ele adotou em seu artigo a nomenclatura Injeção de dependências para este padrão.

Para ilustrar, seja o exemplo de um filme que vai ser dirigido por um diretor de fotografia. A classe Filme precisa do serviço da interface Diretor, e DiretorFotografia é uma classe que implementa Diretor. Sem a injeção de dependência, o diagrama de dependência resultante pode ser observado na Figura 10.

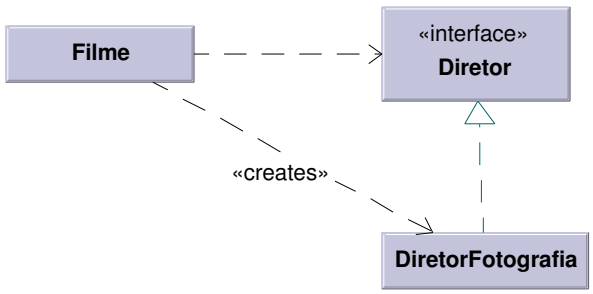


Figura 10 - Diagrama de dependência sem injeção de dependência

A idéia básica da Injeção de dependência é ter um objeto separado, o montador (*assembler*), que popula um campo de um objeto com uma implementação apropriada. A Figura 11 mostra o diagrama de dependência do exemplo acima com o montador.

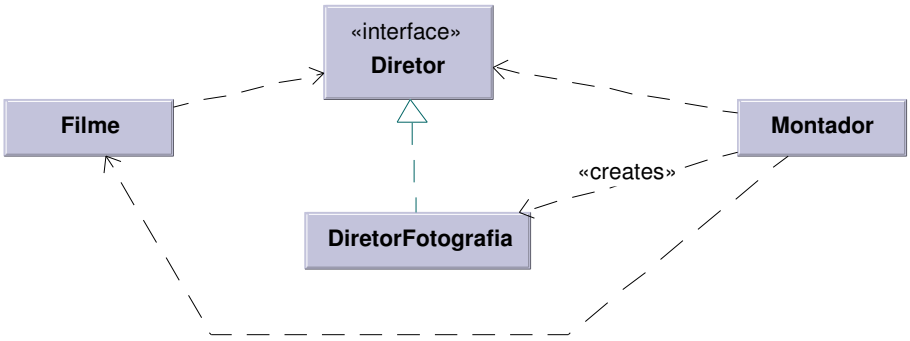


Figura 11 - Diagrama de dependência com o montador

Existem três tipos principais de Injeção de dependência, a saber, Injeção por Construtores, Injeção por Métodos Set (métodos de atribuição) e Injeção por Interfaces. A Injeção por interface requer mais trabalho para montar os componentes e as dependências, pois é preciso escrever as interfaces que serão utilizadas. Por esse motivo, os *containers* mais recentes optaram pelos outros tipos de injeção. A escolha entre injeção por construtores e injeção por métodos set esbarra no conceito de boas práticas da programação orientada a objetos. Na visão de Beck (1996), é preferível criar objetos válidos em tempo de construção. Construtores com parâmetros oferecem uma visão clara do que se pretende criar. Outra vantagem da injeção por construtores é proteger os campos que devem permanecer inalterados. A opção por injeção por métodos set é mais justificável quando existem muitos parâmetros a serem passados no construtor.

Este trabalho utiliza o container PicoContainer, que preconiza a injeção por construtores, embora também disponibilize injeção por métodos set.

PicoContainer

O PicoContainer foi escolhido para este trabalho por ser um container leve (e sem dependência externa) e de fácil uso. Aplicando o padrão Injeção de Dependência, o PicoContainer gerencia os componentes deixando o código menos acoplado. Ele é responsável por instanciar os componentes e associá-los aos componentes dependentes. Os componentes são implementações de classes Java comuns. Não é necessário implementar nenhuma API para a utilização deste container.

Além da facilidade de utilização, o PicoContainer oferece algumas funcionalidades interessantes como por exemplo o gerenciamento de ciclo de vida de um componente através da interface *Startable*. O PicoContainer pode ser facilmente estendido para abranger outras funcionalidades através da criação de novos mecanismos semelhantes ao de gerenciamento de ciclo de vida.

O PicoContainer foi utilizado neste trabalho como meio de configurar os agentes que atuariam na transformação do domínio. Estes agentes foram implementados como componentes pequenos e com tarefas simples. Através do PicoContainer, esses componentes são combinados para atingir a transformação necessária a um determinado contexto.

2.7 Resource Description Framework (RDF)

A linguagem XUL (XML User-interface Language) (XUL, 2005) é um padrão aberto criado pelo grupo de desenvolvedores do navegador Mozilla (MOZILLA, 2005). XUL é uma linguagem multi-plataforma para descrever interfaces gráficas com o usuário em aplicações. O XUL provê a capacidade de criar a maioria dos elementos encontrados em interfaces gráficas modernas. É genérico o suficiente para que possa ser aplicado às necessidades especiais de certos dispositivos e poderoso o suficiente para que desenvolvedores possam criar interfaces sofisticadas com ele. O conteúdo exibido pode ser criado a partir de arquivos XUL ou com dados vindos de uma fonte de dados. Uma opção de fonte de dados é a utilização de arquivo RDF.

O RDF (RDF, 2005) (ZANETE, 2005) (HYATT, 2005) , como XUL, é uma linguagem baseada em XML. O exemplo abaixo mostra um modelo RDF simples listando uma tabela com três registros e três campos sobre animais.

```
<RDF:RDF xmlns:RDF="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
         xmlns:ANIMALS="http://www.some-fictitious-zoo.com/rdf#">
  <RDF:Seq about="urn:animals:data">
    <RDF:li>
       <RDF:Description about="urn:animals:lion">
         <ANIMALS:name>Lion</ANIMALS:name>
         <ANIMALS:species>Panthera leo</ANIMALS:species>
         <ANIMALS:class>Mammal</ANIMALS:class>
       </RDF:Description>
    </RDF:li>
    <RDF:li>
       <RDF:Description about="urn:animals:tarantula">
         <ANIMALS:name>Tarantula</ANIMALS:name>
         <ANIMALS:species>Avicularia avicularia</ANIMALS:species>
         <ANIMALS:class>Arachnid</ANIMALS:class>
       </RDF:Description>
```

```
    </RDF:li>
    <RDF:li>
       <RDF:Description about="urn:animals:hippopotamus">
         <ANIMALS:name>Hippopotamus</ANIMALS:name>
         <ANIMALS:species>Hippopotamus amphibius</ANIMALS:species>
         <ANIMALS:class>Mammal</ANIMALS:class>
       </RDF:Description>
    </RDF:li>
  </RDF:Seq>
</RDF:RDF>
```

Neste exemplo, três registros foram descritos, um para cada animal. Cada marcador *"RDF:Description"* descreve um único registro. Em cada registro, três campos são escritos, "*name"*, "*species"* e "*type"*. Não é preciso ter os mesmos campos para todos os registros, mas faz mais sentido que seja assim. A cada um dos três campos foi dado um *namespace* "ANIMALS", cuja URL foi declarada no marcador RDF.

Os elementos "seq" e "li" são usados para especificar que os registros estão em uma lista. Isso é muito parecido com a forma como listas são declaradas em HTML. O elemento "seq" é usado para indicar que os elementos estão ordenados, ou em uma seqüência. Ao invés do elemento "seq", pode ser usado o elemento "bag" para indicar que os dados estão desordenados, e "alt" para indicar dados onde cada registro especifica valores alternativos.

Para permitir a criação de elementos baseados em dados RDF, é preciso prover um modelo simples que será duplicado para cada elemento que é criado. Em essência, só se faz o primeiro elemento, sendo os elementos remanescentes construídos a partir dele. O modelo é criado usando o elemento *template*. Dentro dele, são colocados os elementos XUL desejados para representar cada item construído. O trecho XUL abaixo popula um *vbox* com botões cujo título é extraído de um RDF. O resultado pode ser visualizado na Figura 12.

```
<vbox datasources="rdf:bookmarks" ref="NC:BookmarksRoot" flex="1">
  <template>
    <button uri="rdf:*" label="rdf:http://home.netscape.com/NC-rdf#Name"/>
  </template>
</vbox>
```

Figura 12 - Botões com título extraído de um RDF.

O RDF provê uma maneira interessante de reunir dados locais e remotos para serem apresentados em uma descrição XUL. No entanto, a representação dos dados é muito simplificada. A riqueza de representação de interface gráfica oferecida pelo XUL não é encontrada na representação do modelo pelo RDF. Há uma carência de informação quanto a entidades, relacionamentos e atributos. Além disso, a ligação dos elementos de vista com o modelo é feita apenas no sentido modelo-vista. A vista não é capaz de notificar o modelo de alterações no conteúdo. Visando enriquecer o modelo informações pertinentes, este trabalho apresenta uma proposta para a representação do modelo de forma a favorecer o trabalho das camadas da aplicação.

2.8 Apache Struts

Apache Struts (CAVANESS, 2004) é um framework livre utilizado para criação de aplicações WEB em Java. O framework Struts não é uma tecnologia em específico, mas sim um conjunto destas, que possibilita o desenvolvimento de aplicações WEB, utilizando-se do modelo MVC-2 (SESHADRI, 1999). Este modelo é implementado no Struts através de um Servlet que controla a aplicação, o *ActionServlet*, sendo este o responsável pelo controle do fluxo entre as páginas JSP (Java Sever Page) e as diversas camadas da aplicação. É implementado ainda o padrão MVC usando ActionForwards e ActionMappings para manter as decisões do fluxo do controle fora da camada de apresentação.

Esta definição bem clara das camadas em uma aplicação, permite a distribuição do trabalho, de forma que o designer pode desenvolver as páginas JSP's sem se

preocupar com o restante da aplicação, e a sua integração será bastante simples através da utilização das *taglibs* do próprio Struts.

A Figura 13 apresentada o fluxo normal de uma aplicação Struts e quais tecnologias estão envolvidas neste determinado momento.

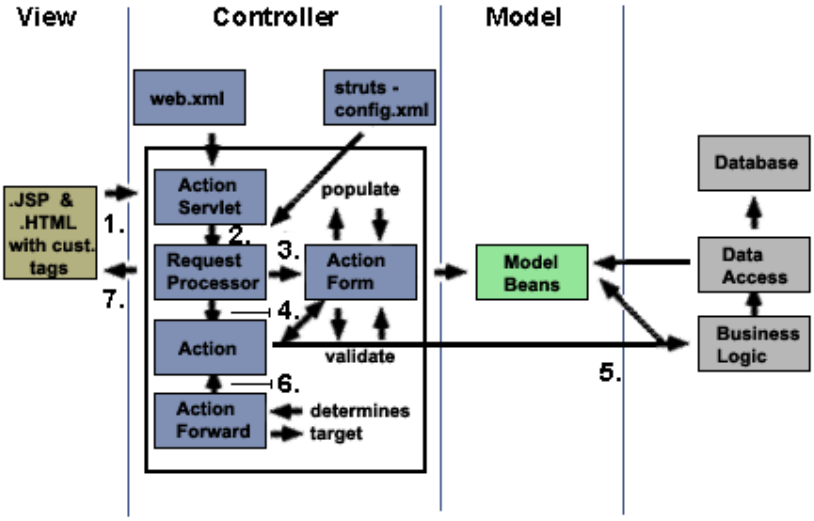


Figura 13 - Framework Struts

1. Cada solicitação HTTP tem que ser respondida com uma resposta HTTP. Desta forma inicia-se uma aplicação que utiliza o Struts, com uma solicitação HTTP. Esta solicitação, normalmente é definida como *alguma_coisa.do* .

2. A solicitação *alguma_coisa.do* é mapeada no arquivo *struts-config.xml*. Este arquivo é lido por um *ActionServlet* que lê este arquivo na inicialização da aplicação e cria um banco de objetos com o arquivo de configuração (HUSTED et al, 2004). No arquivo de configuração são definidos os *Actions* para cada solicitação.

3. O *ActionServlet* define o *Action* correspondente para a solicitação. Um *Action* pode validar a entrada e acessar a camada de negócios para recuperar as informações nos bancos de dados e outros serviços de dados.

4. A requisição HTTP pode ser feita também através de um formulário HTML. Em vez de fazer com que cada *Action* retire os valores do campo da solicitação, o *ActionServlet* coloca a entrada em um *JavaBean* , estes *JavaBeans* são definidos como *FormBeans* no Struts e estendem a classe *org.apache.struts.action.ActionForm*.

5. O *Action* interage com a camada de negócio onde uma base de dados poderá ser atualizada.

6. Em geral, o Struts não apresenta a resposta em si, mas envia a solicitação para outro recurso como uma página JSP. O Struts fornece uma classe *ActionForward* que pode ser usada para armazenar o caminho para uma página sob um nome lógico (HUSTED et al, 2004). Desta forma o endereço ficará oculto para o usuário, este visualizará apenas o nome definido para o caminho.

No âmbito deste trabalho, é interessante observar como o Struts lida com a informação que vem e volta para tela. Para obter os dados da requisição, o Struts utiliza os *FormBeans* e para dar a resposta é comum a utilização de *JavaBeans* comuns. Um *FormBean* também é um *JavaBean*, com o detalhe que este estende uma classe específica do framework Struts, o *ActionForm* (*org.apache.struts.action.ActionForm*).

Os *JavaBeans* são classes Java de acordo com um conjunto de padrões de construção que facilitam o uso com ferramentas de desenvolvimento entre outros componentes (HUSTED et al, 2004). Um *JavaBean* pode ser ainda definido como um componente de software reutilizável escrito em Java. Para ser qualificado como *JavaBean*, a classe tem que ser concreta e pública e ter um construtor sem argumentos. Os *JavaBeans* expõem os campos internos como propriedades fornecendo métodos públicos que seguem um padrão de projeto consistente. Sabendo que os nomes patenteados seguem esse padrão, as outras classes Java são capazes de usar a introspecção para descobrir e manipular as propriedades *JavaBean* (HUSTED et al, 2004).

Para cada formulário da aplicação que utiliza o Struts, deverá ser construído um *FormBean*. Cada campo do formulário terá o seu devido atributo na classe, juntamente com os métodos acessores e transformadores. Quando um formulário HTML é enviado, as duplas nome-valor são obtidas pelo controlador Struts e aplicadas em um *FormBean*. O Struts compara então os nomes das propriedades deste *Bean* com os nomes das duplas de entrada, quando coincidem, o controlador define a propriedade para o valor da dupla correspondente. As propriedades extras são ignoradas. As propriedades ausentes mantêm o seu valor padrão. Estes pares de entrada são compostos pelos campos do formulário HTML.

Este mecanismo de obtenção dos dados implementado pelo Struts é simples e prático, pois libera o desenvolvedor do trabalho extrair a informação pretendida do objeto de requisição. Contudo, esta mesma facilidade não é observada na geração da resposta. A informação a ser apresentada é obtida de JavaBeans pelos JSPs. Pode haver a necessidade de ter mais de um JavaBean para completar a informação de resposta. Dessa forma, são dois os lugares de especificação de modelo utilizado: os FormBeans descritos no struts-config.xml e os JavaBeans utilizados para obter a resposta descritos nos JSPs.

Outra característica do *FormBean* é ser uma classe plana, com propriedades simples, que reúne todas as informações vindas da requisição. É um modelo engessado, que não é capaz de representar coleções ou relacionamentos entre entidades. Por exemplo, uma tela para adicionar telefones de uma pessoa de forma dinâmica, seria complicado representar com o *FormBean*, pois teria que se conhecer previamente a quantidade de telefones para colocar as propriedades correspondentes no *FormBean*.

O framework Struts é similar ao framework Hércules em grau de dificuldade, pois ambos envolvem um conjunto de tecnologias, podendo o aprendizado tornar-se bastante dispendioso. Apesar destas dificuldades, as tecnologias e padrões adotados são fatores motivantes para aprendizado e utilização de ambos. No entanto, em relação à proposta de modelo para informação da camada de vista, o Hércules supera o Struts oferecendo maior flexibilidade e uma estrutura mais sofisticada que facilita o trabalho do desenvolvedor.

3 PLATAFORMA HÉRCULES E O DOMÍNIO EM ZEUS

O mercado de software apresenta uma tendência à especificação de Sistemas de Informação cada vez mais complexos e volumosos. Nem sempre os prazos são dilatados para acompanhar o crescimento de requisitos, causando atraso na entrega. Muitas vezes a preocupação com o cumprimento da agenda não deixa tempo hábil para o projetista especificar uma arquitetura robusta e flexível, prejudicando a manutenibilidade do sistema. Apesar de se observar muitas semelhanças no desenvolvimento entre serviços, a ausência de uma ferramenta apropriada faz com que o programador realize muito trabalho repetitivo. Esse trabalho quase mecânico contribui para aumentar a probabilidade de surgimento erros no código. Uma solução é o uso de ferramentas que permitam automatizar etapas do processo de desenvolvimento.

A experiência em desenvolvimento de sistemas de informação de grande porte permitiu que um grupo de alunos da UFRJ desse início ao projeto Hércules (PAIS, OLIVEIRA, LEITE, 2001) (PAIS, OLIVEIRA, 2001) (PAIS, BRASIL, OLIVEIRA, 2002). O projeto Hércules consiste de um *framework* para desenvolvimento de sistemas de informação visando possibilitar a geração automática de código a partir de diagramas UML (Unified Modeling Language) (UML, 2005). Este enfoque permite que o desenvolvedor se concentre apenas com o negócio do sistema. O objetivo da plataforma Hércules é padronizar o desenvolvimento de sistemas de informação bem como oferecer engenhos capazes lidar com as tarefas repetitivas da aplicação. Este trabalho é parte integrante da Plataforma Hércules.

Uma das principais preocupações da Plataforma Hércules foi possibilitar a renderização do sistema em diferentes interfaces gráficas. Por esse motivo, a construção da Plataforma Hércules iniciou com a camada de vista do padrão MVC. Nesta camada foi aplicado o MVC recursivo dando origem a outras três camadas modelo, vista e controle. Esta plataforma foi edificada com bases em características como flexibilidade na apresentação do sistema, rapidez no desenvolvimento e

facilidade de adaptação a mudanças nos requisitos. Um conjunto de descrições de alto nível foi estabelecido para que os desenvolvedores especificassem a apresentação, o controle do caso de uso e o modelo da vista. Engenhos especializados examinam essas descrições e geram parte do código fonte responsável pelo funcionamento da aplicação.

Até o momento apenas a renderização em HTML foi concluída. A renderização em Swing encontra-se em fase de implementação. A especificação de uma descrição abstrata para a apresentação despertou o interesse de outros alunos dando origem a ferramentas para auxiliar na construção de tal descrição. O projeto Meg (PEREIRA, 2003) foi desenvolvido para auxiliar a construção do XML da primeira versão da descrição de apresentação. É uma ferramenta que à medida que se escreve o XML ela apresenta uma amostra de como ficaria a interface gráfica. Outro projeto, o XULBuilder (BRASIL, 2003) surgiu na época da segunda e atual versão que adota o padrão de descrição de interface gráfica XUL (XUL, 2005). O XULBuilder gera a especificação em XUL a partir de uma ferramenta visual.

A plataforma Hércules está sendo utilizada há aproximadamente quatro anos no Sistema Integrado de Gestão Acadêmica (SIGA) na UFRJ. Durante esse período muitas mudanças foram feitas na implementação como, por exemplo, a redução de JavaScript ao mínimo necessário e a alteração da descrição de vista adaptada ao XUL. Os serviços, que antes eram feitos de maneira diversificada e desorganizada, ficaram padronizados.

Apesar das mudanças no decorrer do tempo, a plataforma Hércules não expandiu para as camadas de controle e de modelo. Este trabalho propõe um *framework* que implementa um barramento de transporte da informação ao longo das camadas da aplicação. Este modelo se mostrou funcional para a camada de vista, porém não é prático para as outras camadas. No SIGA, o modelo na camada de controle, em muitos serviços, é inexistente; em outros é utilizada uma implementação simples do padrão de projeto *ValueObject* (ALUR, CRUPI, MALKS, 2001, p. 261). A camada de modelo adota o formato exigido pelo mecanismo de persistência, neste caso o EJB (MONSON-HAEFEL, BURKE, LABOUREY, 2004). Além do modelo da camada de

vista, este trabalho apresentará uma proposta para a representação do domínio no contexto de um serviço no *framework* Hércules.

3.1 *Framework* Hércules em Zeus

A Plataforma Hércules adota o estilo arquitetônico MVC. Contudo, ela se aprofunda um pouco mais na divisão de camadas, adotando o MVC recursivo. O MVC recursivo provoca uma extensão na nomenclatura das camadas, por exemplo, "camada de modelo da camada de modelo" ou "camada de controle da camada de vista". Para efeito de simplicidade ao fazer referência a uma camada, a Plataforma Hércules batizou suas camadas mais externas. As camadas de vista, controle e modelo receberam o nome de Zeus, Hera e Olimpo respectivamente.

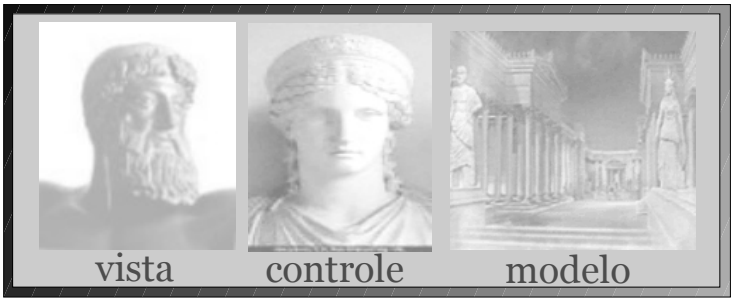


Figura 14 - Zeus, Hera e Olimpo

O desenvolvimento da Plataforma Hércules se iniciou por Zeus. O objetivo de Zeus é distanciar o programador de aspectos de apresentação de um serviço, fazendo com que este se concentre nas regras de negócio. A implementação de Zeus sofreu influência de conhecimento da arquitetura Swing do Java (que adota o estilo arquitetônico MVC) e de controle através de Máquinas de Estado, obtidas através de diagramas de estado UML.

A seguir é feita uma descrição mais detalhada das camadas internas de Zeus. Pais, em sua dissertação de mestrado (PAIS, 2004), abordou as subcamadas de controle e de vista. A subcamada de modelo de Zeus ficou a cargo deste trabalho.

3.2 Zeus: camada de vista

A interface gráfica é o cartão de visita de um sistema. Por dentro, o sistema pode estar funcionando sem erros, mas se a apresentação não agradar o usuário certamente ele não ficará satisfeito. Por isso, o programador dedica muito tempo construindo uma interface gráfica “amigável”. Existem muitas opções de renderização tanto para aplicações *desktop* como para aplicações WEB. No entanto, quando o programador começa a desenvolver em um tipo de renderização, por exemplo, HTML, e tem que mudar o trabalho desperdiçado é muito grande.

Sistemas que apresentam baixo acoplamento com as classes de vista permitem que alterações na interface gráfica impactem o mínimo no funcionamento do sistema. A interface gráfica é apenas o meio de apresentação do sistema. Um sistema que está funcionando adequadamente com uma determinada renderização deveria continuar funcionado caso essa renderização fosse intercambiada por uma outra qualquer. Mesmo que o sistema continuasse funcionando, o programador teria novamente o trabalho de construir as telas para esta renderização e inclusive aprender a sintaxe e os recursos de uma nova linguagem.

A interação do usuário com a interface gráfica gera eventos que devem ser tratados pelo sistema. É necessário conhecer o comportamento dos componentes de vista para obter os eventos gerados. Cada tipo diferente de renderização requer o aprendizado de mais informação quanto aos componentes e à forma como eles tratam os eventos. Muitos eventos, que poderiam ser tratados no cliente, são propagados sobrecarregando o servidor com tarefas exclusivas de vista.

A camada Zeus foi desenvolvida para lidar com os aspectos de configuração de telas e controle de eventos gerados a partir da interação do usuário independente da renderização escolhida para a aplicação. Com a arquitetura utilizada em Zeus o desenvolvimento se torna padronizado, pois as classes que o programador deve implementar possuem papéis bem definidos. Outras soluções foram estudadas na dissertação de Pais, contudo não se mostraram satisfatórias para o problema (PAIS, 2004).

Zeus provê o controle de estados e eventos de um caso de uso através de uma descrição abstrata simples, porém abrangente, das telas, e utiliza uma representação eficiente de modelo. Esses artefatos surgiram com a aplicação do MVC recursivo caracterizando a divisão de modelo vista e controle em Zeus. Não é necessário escrever código para componentes de vista, apenas descrevê-los e associá-lo aos respectivos componentes de modelo. Zeus contém engenhos capazes de entender a descrição MVC que facilitam o trabalho de construção da vista de um serviço.

A primeira versão de Zeus foi construída em *JavaScript*, causando uma sobrecarga nos clientes. Essa versão foi utilizada por algum tempo, porém com muitas reclamações quanto à demora na renderização das páginas. Essa insatisfação fomentou a construção da segunda e atual versão de Zeus. Esta versão foi implementada com classes Java que formam as camadas de vista, controle e modelo de Zeus. Cada camada possui engenhos que permitem que o programador use uma descrição abstrata. A versão atual trouxe para o *Web Server* a carga de processamento, que na versão em *JavaScript* estava no cliente. Esta última versão está sendo usada há mais de dois anos no SIGA.

3.2.1 Apresentação Zeus

A construção das telas de um sistema requer tempo e conhecimento da biblioteca de componentes utilizada. Muitas empresas possuem profissionais especializados em design das telas dos serviços de um sistema. O conhecimento necessário envolve comportamento dos componentes, atribuição de valores, tratamento de eventos entre outros. Além disso, para trocar uma renderização por outra é preciso construir tudo novamente com outra biblioteca de componentes.

Para alcançar o objetivo de possibilitar diferentes renderizações foi preciso especificar uma descrição abstrata de interface gráfica. A partir dessa descrição foi construída uma biblioteca Java de componentes visuais correspondendo aos elementos contidos na especificação da descrição. Para uma determinada renderização é preciso construir uma biblioteca específica de componentes visuais, bem como o mecanismo de tradução da biblioteca abstrata para a específica. A biblioteca específica para HTML foi a primeira a ficar pronta e está sendo utilizada no

SIGA. A adoção de uma especificação abstrata permitiu que o programador se limitasse a aprender somente uma renderização – a renderização Hércules.

O XML abaixo apresenta trechos da descrição[1] do serviço “Ativar Turmas” do SIGA.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<mvc>
  <view id="previsao_turma" title="Ativação de Turmas">
      <tabbox id="tabbedMenu">
         <tabs>
            <tab label="Seleção" selected="true" />
            <tab label="Lista" selected="false" />
            <tab label="Mensagem" selected="false" />
         </tabs>
         <tabpanels>
            <tabpanel id="Selecao">
            (...)
            </tabpanel>
            <tabpanel id="Lista">
               <hbox width="98" align="center" id="mainBoxtabbedMenu">
                  <vbox id="formList" align="center" width="100%">
                     <box id="dummyId" align="center" value="5">
                        <button id="btn1" command="ListState-1" type="default" label="" />
                        <button id="btn2" command="ListState-2" type="default" label="" />
                        <button id="btn3" command="ListState-3" type="default" label="" />
                     </box>
                     <grid id="tableListaTurma" ref="list" class="listTable">
                        <columns>
                           <column />
                           <column value="Código" />
                           <column value="Nome" />
                           <column value="Segmentação" />
                           <column value="Situação" />
                        </columns>
                        <rows>
                           <template>
                              <row>
                                 <checkbox id="ativar" datasources="ListBean" ref="ativar" />
                                 <label id="Codigo" datasources="ListBean" ref="codigoTurma"
                                        class="normal" />
                                 <label id="nome" datasources="ListBean" ref="nome"
                                        class="normal" />
                                 <label id="Segmentacao" datasources="ListBean"
                                        ref="segmentacao" class="normal" />
                                 <label id="Situacao" datasources="ListBean" ref="situacao"
                                        class="normal" />
                              </row>
                           </template>
                        </rows>
                     </grid>
                     <caption />
                  </vbox>
               </hbox>
            </tabpanel>
            <tabpanel id="Mensagem">
            (...)
            </tabpanel>
         </tabpanels>
         <caption label="Ativação de Turmas" />
      </tabbox>
   </view>
</mvc>
```

Engenhos responsáveis pela interpretação da descrição de vista geram os componentes visuais e fazem as associações necessárias com as outras camadas.

A Figura 15 - Tela do serviço Ativar Turmas mostra a tela resultante da descrição XML acima. Nesta tela aparecem componentes como tabelas, botões e abas organizadoras.

Figura 15 - Tela do serviço Ativar Turmas

Os componentes visuais podem ou não estar associados a elementos de modelo. O componente visual que for associado a um elemento de modelo se torna observador deste, recebendo notificação quando mudanças ocorrerem no modelo. As interações do usuário com a interface são comunicadas ao modelo observado, para que este possa tomar as devidas providências. A especificação abstrata oferece meios para organização de componentes através de *layouts* (exemplo: vbox e hbox). Componentes mais complexos como tabelas e caixas de seleção tem comportamentos adaptados à necessidade dos Sistemas de Informação (PAIS, 2004). Além de todas as facilidades descritas, a camada de vista ainda oferece *Look and Feel* (SWING, 2005) configurável. A cores e formas contidas na Figura 15 está com *Look and Feel* ao sabor do SIGA.

[1] O manual completo da descrição MVC do *framework* Hércules pode ser obtido no sítio do Hércules (PAIS, 2005).

É através da camada de vista que o usuário tem contato com uma face do domínio do sistema. Essa face é representada em Zeus pela camada de modelo. Em Zeus, o modelo registra a interação do usuário e permanece alterado até ser analisado por Hera. No entanto a camada de controle tem poder para manipular este modelo caso haja necessidade.

3.2.2 Controle Zeus

Sistemas de Informação são composições de casos de uso. Um caso de uso representa uma funcionalidade do sistema e é descrito através de um roteiro. Ana Paula Valente Pais enxerga os clientes de um Sistema de Informação como roteiristas, cuja função é especificar diálogos entre os elementos do sistema. Em sua dissertação, Pais aponta a camada de controle como o local apropriado para a encenação do roteiro.

> "A camada de controle torna-se a peça fundamental para a realização do roteiro do sistema, constituindo o local ideal para a 'encenação' de tal roteiro. Elementos da camada de controle coordenam elementos das camadas de apresentação e domínio durante a realização de cada um dos diálogos. Somente os elementos da camada de controle detêm conhecimento sobre qual parte do roteiro está sendo executada, assim como sabem exatamente quais são as falas presentes no roteiro." (PAIS, 2004, p. 45)

O diagrama de estados da UML é um artefato muito poderoso para representação do roteiro de um caso de uso. Entretanto, este diagrama geralmente é subutilizado no momento de implementação do caso de uso. A configuração de estados e eventos permite distinguir momentos do caso de uso, bem como quais eventos podem ocorrer a cada momento. Aplicações que não mantêm o estado têm maior dificuldade em acompanhar caminhos alternativos do roteiro de caso de uso. A replicação de código e a intervenção direta em componentes visuais são freqüentemente observadas nesse tipo de abordagem.

Na Figura 16, o evento "inscrever" só ocorre no estado "InscricaoCoordenador". Embora o conteúdo a ser apresentado tanto para aluno como para coordenador seja o mesmo, somente quando o usuário for coordenador é que deverá aparecer o botão responsável por disparar o evento "inscrever", como está representado na Figura 17.

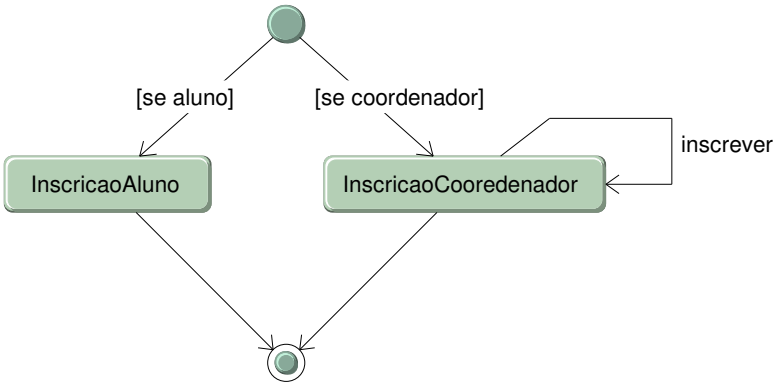


Figura 16 - Diagrama de estados

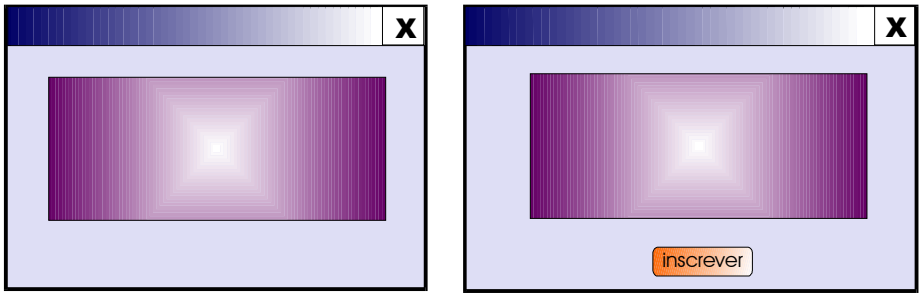


Figura 17 - Telas resultantes do diagrama de estados

A simulação de uma máquina de estados para controlar o fluxo do caso de uso é a tarefa central da camada de controle de Zeus. Para cada caso de uso o programador constrói um controlador composto de estados e eventos. O controlador possui uma configuração de quais eventos podem ocorrer em cada estado e quais ações e transições um determinado evento pode desencadear. A Figura 18 mostra o diagrama de classes com alguns relacionamentos da classe Controller (que representa o controlador). As classes ControllerContext e ControllerDispatcher auxiliam o controlador em algumas tarefas (PAIS, 2004).

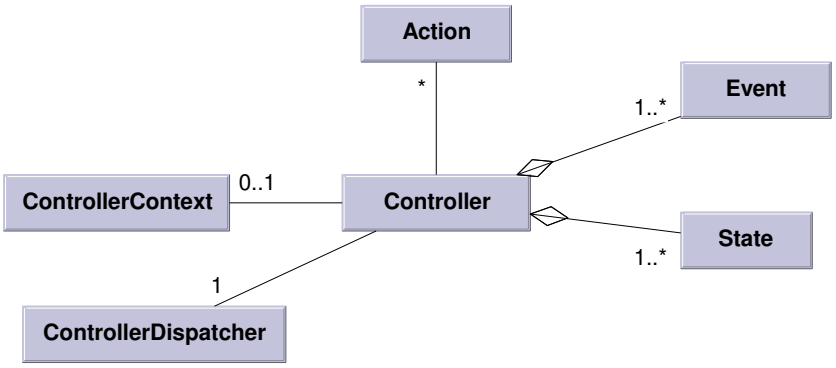


Figura 18 - Controlador Hércules

Com base no estilo arquitetônico MVC, a camada de controle conhece as camadas de vista e modelo. Ela recebe a descrição do caso de uso (usualmente através de XML) e dispara tarefas como a construção da apresentação e associação de elementos de modelo aos componentes visuais. A camada de controle recebe eventos disparados pelos componentes visuais que alimentam a máquina de estados. Ações podem ser realizadas pelo controlador antes de propagar os eventos para Hera. Alguns eventos podem ser resolvidos sem que seja necessário ser propagado.

O controlador tem acesso ao domínio do sistema através do modelo de Zeus. A informação lá contida pode ser manipulada pelo controlador a qualquer momento. Não é aconselhável implementar regras de negócio em Zeus. Se uma regra de negócio é particular a um serviço, então esta regra deve estar situada em Hera. Por outro lado, se a regra de negócio é válida para o domínio, independente do serviço, esta regra deve estar no Olimpo.

Em aplicações de grande porte, muitos controladores são instanciados. Em alguns casos, são instanciados muitos de um mesmo tipo. Para gerenciar a instanciação e utilização de controladores a camada de controle possui uma fábrica (GAMMA, et al, 1995). Quando um caso de uso é invocado, a fábrica identifica a requisição e devolve o respectivo controlador. A fábrica mantém a informação de controladores que estão em uso e controladores ociosos para futuro reaproveitamento. Quando um caso de uso é invocado é reservado para ele um container. Os casos de uso que puderem ser alcançados a partir desse primeiro ficam empilhados no container. O

container mantém o estado dos controladores empilhados. A Figura 19 apresenta os relacionamentos da fábrica com as classes *Container* e *Controller*.

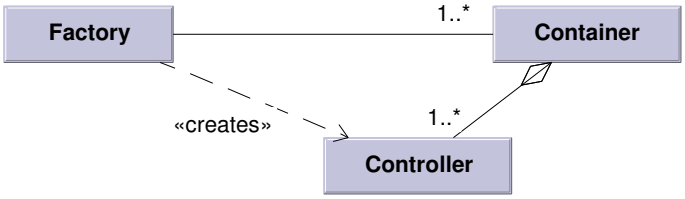


Figura 19 - Fábrica de controladores

3.2.3 Modelo em Zeus

Os clientes de Sistemas de Informação buscam agilidade nas telas dos serviços, aspirando ter, em uma mesma tela, acesso a diversas funcionalidades. Serviços complexos requerem um mecanismo sofisticado de armazenamento de informação. O risco de ocasionar uma sobrecarga visual faz com que as interações sejam disponibilizadas em etapas. A Figura 20 mostra um exemplo de uma tela do SIGA onde a informação foi distribuída em abas. Alguma informação pode estar repetida em mais de um lugar como recurso de contextualização para o usuário. Alterações em uma informação devem refletir em todas as aparições dessa informação na tela. Além disso, as informações que não são mais visualizadas em um dado momento devem estar armazenadas aguardando a conclusão da interação para que possam ser processadas.

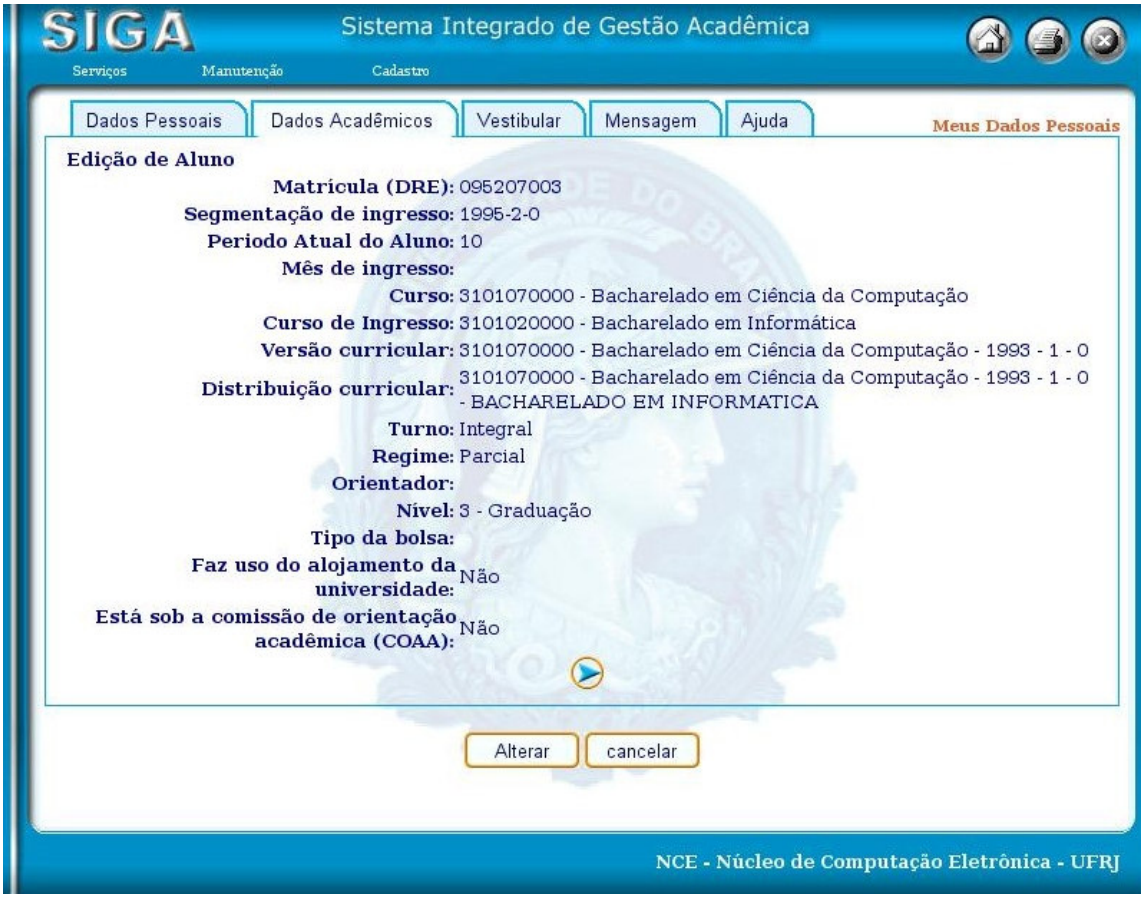


Figura 20 - Serviço de Dados Pessoais do SIGA

Este trabalho mostrará que a camada de modelo de Zeus fornece a estrutura apropriada para lidar com a informação requerida pelos casos de uso. Essa informação se refere a entidades, propriedades e relacionamentos cuja representação é feita através de classes de meta informação pertencentes à camada de modelo. A informação armazenada é manipulada por engenhos especializados em realizar determinadas operações, complementando a camada de modelo. O modelo é independente da apresentação podendo um mesmo dado ser representado por mais de um componente visual. Alterações no modelo são refletidas em todos os componentes visuais associados.

Seguindo o princípio arquitetural do Swing do Java, a camada de modelo não tem conhecimento das outras camadas. A camada de modelo apenas faz registro de observadores interessados em mudanças na informação. Os componentes visuais são os principais observadores dos elementos da camada de modelo. Quando

ocorre alguma mudança na informação, os respectivos observadores são notificados. Por outro lado, os componentes visuais conhecem o elemento de modelo que apresentam e enviam notificações de interação diretamente ao modelo, sem que seja necessária a intervenção do controle. O modelo é facilmente acessível pelo controle, para que este possa manipular ou disparar alguma operação sobre o modelo.

Os engenhos da camada de modelo foram criados para realizar operações básicas como atualização de um modelo por outro modelo, iniciação do modelo e algumas transformações. Os engenhos de transformação implementados são dois: um traduz o modelo em XML e vice-versa e outro traduz em *ValueObjects* e vice-versa.

A especificação e construção dessa camada fazem parte deste trabalho como representação do domínio em Zeus. A camada de modelo de Zeus trás uma contribuição importante para a facilidade de implementação e manutenção dos serviços desenvolvidos utilizando o *framework* Hércules.

3.3 Representando o domínio em Zeus

A essência da informação de um sistema adquire formas diferentes em cada camada. É como se uma camada não possuísse todos os “sentidos” necessários para absorver a informação por completo. O domínio fica condicionado à interpretação de cada camada vestindo a forma apropriada para ser compreendido. Parte da informação se perde, todavia é uma parte sem utilidade para a camada.

O domínio presente em Zeus é aquele que compreende a informação que deve ser apresentada. Em alguns casos a informação a ser apresentada pode se parecer com a informação a ser processada e a informação a ser armazenada. Contudo, na maioria dos casos as informações utilizadas em cada camada apresentam diferenças, descrevendo distintas faces para o domínio do sistema. Em Zeus é comum surgir informação que não faz parte do domínio do sistema. Essa informação é resultante da aplicação semântica trazida por Hera. Um exemplo é a criação de modelo para representar mensagens informativas que tem por finalidade colocar o usuário ciente do estado de um serviço.

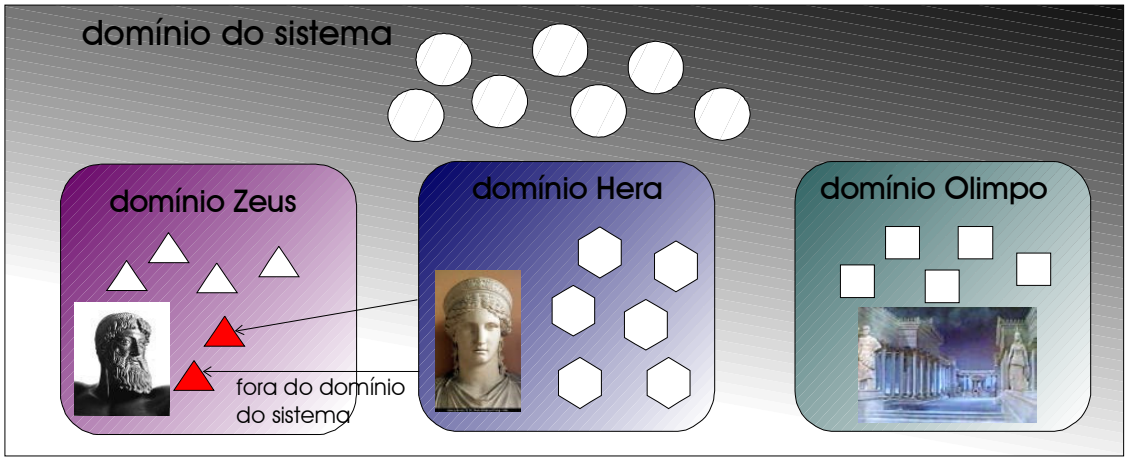


Figura 21 - Formatos do domínio

O domínio em Zeus é representado através de um meta-modelo. Esse meta-modelo é entendido pelas camadas de vista e controle de Zeus. Este é o único modelo aceito por Zeus. As camadas Hera e Olimpo podem trabalhar com este modelo apesar de não ser o mais adequado.

3.3.1 Especificação do domínio

Desde o início do projeto Hércules imperou a idéia de desenvolver uma aplicação independente de sua apresentação. Para isso era preciso encapsular os dados, de forma abstrata, em uma linguagem que pudesse ser entendida universalmente. Nesta época, estudos apontaram o XML como sendo uma boa opção para o transporte de dados. Pesquisas sobre interoperabilidade e representação de dados envolvendo XML levaram ao conhecimento da proposta de Johnson (2005). Em seus artigos ele propõe uma representação de entidades utilizando XML. Esta proposta foi adotada no *framework* Hércules sofrendo ao longo do tempo algumas modificações.

O XML é utilizado apenas como meio de transporte para a informação entre Zeus e Hera. Em Zeus, seja qualquer implementação de renderização, essa informação é extraída do XML e passa a ser representada por um meta-modelo correspondente especificado para este XML. Estender a utilização do XML além do mero transporte de informação não é adequado, pois sua manipulação é trabalhosa e limitada. A utilização de um meta-modelo trouxe maior flexibilidade para a implementação do

framework e tornou dispensável o aprendizado de manipulação de XML pelo programador, uma vez que este é gerado e interpretado por engenhos automáticos do *framework* Hércules.

O modelo de Zeus é composto por diversas classes, mas encontra nas classes apresentadas pelo diagrama da Figura 22 maior representatividade. Para aproveitar os recursos da orientação a objetos, todos os elementos de modelo herdam da classe *AbstractElement*. A classe *JavaBean* representa uma entidade do domínio e a classe *Element* seus atributos. A classe *Composite* indica quando um atributo pode ser composto de outros elementos, por exemplo, as coleções. Isso ocorre quando uma entidade tem um relacionamento "um para muitos" com outra entidade. Quando esse relacionamento é "um para um" é representado por um elemento *JavaBean* dentro de um *JavaBean*.

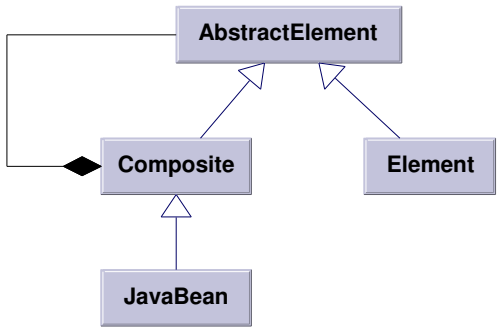


Figura 22 - Meta-modelo

A Figura 23 mostra, através de estereótipos, como é representado o modelo da aplicação Chronos. Esta aplicação tem como objetivo gerenciar projetos segundo o processo de desenvolvimento Extreme Programming (XP) (TELES, 2004). As entidades são representadas pela classe *JavaBean*. Os relacionamentos de composição "um para muitos" dão origem às propriedades *interacoes*, *estorias* e *tarefas* do tipo *Composite* em *Projeto*, *Iteracao* e *Estoria* respectivamente. Atributos simples como nome, *dataInicio* e *dataFim* de *Iteracao* são mapeados para *Element*.

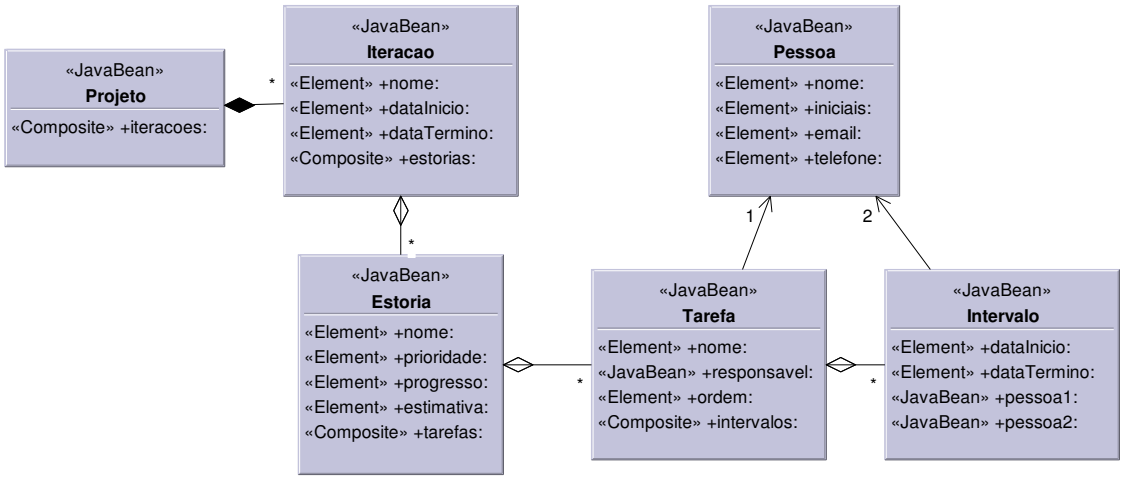


Figura 23 - Representação da aplicação Chronos

Esse meta-modelo, além de representar as entidades, seus atributos e seus relacionamentos, permite o registro de observadores e visitantes para sua estrutura através dos padrões de projeto Observer e Visitor (GAMMA, HELM, JOHNSON, VLISSIDES, 1995).

3.3.2 Estruturação do domínio

A proposta de Johnson (JOHNSON, 2005) se constituiu em uma idéia inicial para a representação do modelo em Zeus. No entanto, essa proposta foi aperfeiçoada à medida que surgiam obstáculos à representação da informação. Um obstáculo foi a representação de múltiplas referências a uma mesma entidade. Outro importante obstáculo foi a representação de referências nulas. O modelo de Zeus oferece recursos para representar essa informação.

A aplicação Chronos pode ser vista como exemplo onde as referências e os elementos nulos seriam necessários. Em um dado projeto do Chronos, uma *Pessoa* "A" é responsável por uma *Estoria* e participa de uma *Tarefa*. Se cada uma dessas pessoas for uma instância diferente de *JavaBean*, ao atualizar, por exemplo, o *email* de "A" será preciso atualizar todas as instâncias. Por esse motivo o modelo de Zeus possui a classe *JavaBeanRef* cuja função é referenciar um *JavaBean*. Dessa forma, no exemplo mencionado, só é preciso atualizar apenas um único ponto.

A representação de elementos nulos pode parecer estranha, mas é muito importante para a reconstrução dos *JavaBeans*. Uma referência pode estar nula, por exemplo, no momento que aparece na tela pela primeira vez, contudo se o usuário preencher esse campo este deixará de ser nulo. Ao deixar de ser nulo, o *JavaBean* correspondente deve ser reconstruído a partir de um modelo que é conhecido pela representação do elemento nulo.

Essas representações de informação são utilizadas tanto na comunicação com Hera como na apresentação. Os componentes visuais são associados a elementos de modelo, observando tudo que acontece com eles. A Figura 24 mostra a associação de elementos de modelo do Chronos com sua apresentação.

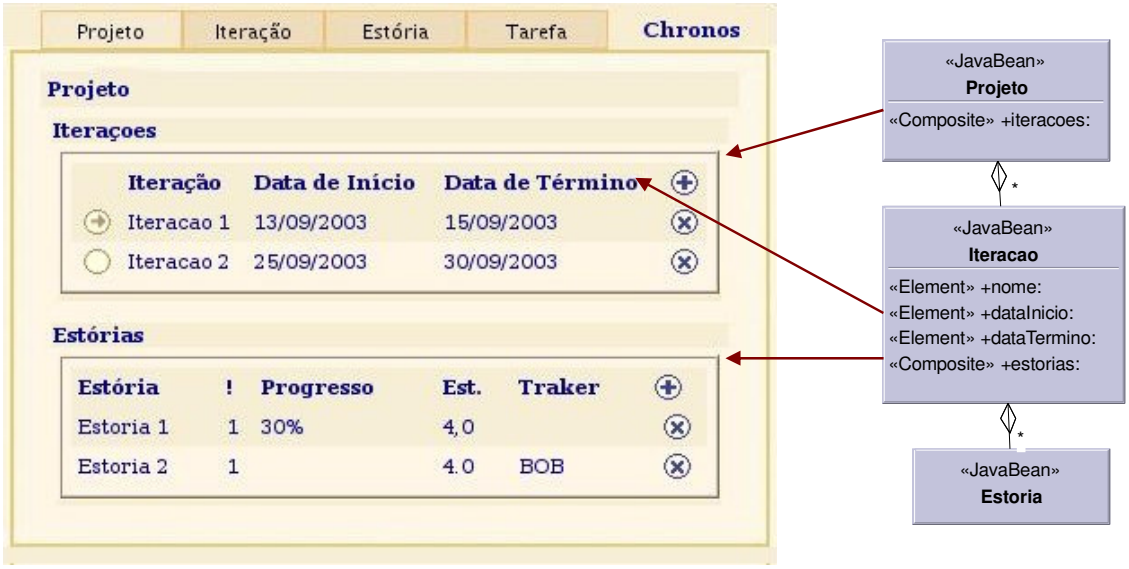


Figura 24 - Apresentação de Chronos e seu modelo

A propriedade *iteracoes*, representada por *Composite*, está associada a um componente visual do tipo tabela, como pode ser visualizado na Figura 24. Essa tabela apresenta tantas linhas quantos elementos estão inseridos em *iteracoes*. Caso o número de elementos em *iteracoes* altere, ou seu conteúdo não seja mais o mesmo, o componente de vista se encarrega de atualizar a tela sem que o programador precise fazer nada. Na mesma figura está representada uma relação mestre-detalhe, dada pelo relacionamento entre *Iteracao* e *Estoria*. Quando um

elemento de *Iteracao* é selecionado na tabela superior, a tabela inferior apresenta o conteúdo de *estorias* correspondente ao elemento selecionado.

Os elementos de modelo também podem ter seu acesso restringido através da propriedade booleana *readOnly*. Quando esta propriedade possui valor verdadeiro, o componente visual que apresenta o elemento de modelo correspondente deve ser renderizado de forma que não seja possível a alteração deste elemento. Por exemplo, se uma caixa de texto estiver associada a um elemento de modelo apenas de leitura, este componente visual se renderizará como texto simples.

A vista depende do modelo. Uma vez feita a ligação com elementos de modelo, os componente visuais não são mais manejados. Os componentes visuais atualizam a informação apresentada com base em notificações que recebem do modelo. O programador tem apenas a preocupação de manter atualizada a informação contida no modelo.

3.3.3 Comportamento no domínio

A manipulação do modelo pela camada de controle de Zeus revelou a freqüente ocorrência de algumas operações. Estas operações são simples, mas que sem o recurso adequado consomem tempo e esforço do programador. A estruturação do modelo permitiu que algumas operações fossem automatizadas. Sobre a arquitetura de modelo seguindo o padrão *Composite* (GAMMA, HELM, JOHNSON, VLISSIDES, 1995) foi possível construir classes no padrão *Visitor* (GAMMA, HELM, JOHNSON, VLISSIDES, 1995) para desempenhar algumas funções. Estão incluídas nessas operações a atualização e a iniciação do modelo.

No início da execução de um serviço, Zeus recebe a definição do modelo que será utilizado. Depois de concluída uma interação do usuário, os dados contidos no modelo são passados para Hera, onde são aplicadas as regras de negócio. Hera valida a informação recebida, altera o que for necessário e transmite a informação de volta para Zeus. Esse processo é repetido até que o serviço encerre. Zeus deve usar a informação retornada para atualização de seu modelo. Para isso foram implementadas algumas classes responsáveis por essa atualização. O modelo pode ser atualizado a partir de um XML (com especificação igual ao que foi utilizado na

construção do modelo) ou por um conjunto de classes simples que seguem o padrão *ValueObject* (ALUR, 2001).

Outra operação muito freqüente em serviços é a iniciação ou “limpeza” do modelo. Com a utilização de um *Visitor* construído para realizar a limpeza do modelo é possível iniciar toda uma hierarquia de elementos de modelo, bastando para isso passar o *Visitor* para o elemento do topo. Outra operação comum é a clonagem de elementos de modelo. Cada classe é capaz se clonar e obter clone das classes internas. A preservação do conteúdo do modelo na clonagem é opcional.

Essas automatizações vieram agilizar operações simples e corriqueiras no modelo. Na medida em que for surgindo necessidade, mais operações poderão ser automatizadas. A arquitetura do modelo facilita a implementação de mecanismos capazes de percorrer sua estrutura. Operações como essas acontecem sob o comando da camada de controle de Zeus.

3.3.4 Representação do domínio

Em Zeus o domínio encontra uma forma singular de representação. A informação foi moldada de acordo com as necessidades de apresentação. A partir do domínio do sistema é extraído o domínio utilizado em Zeus, preservando sua integridade em um formato particular. Além de formatos diferentes, o domínio significativo para Zeus pode não ser o mesmo que é utilizado em Hera ou Olimpo.

O exemplo da aplicação Chronos apresenta uma situação em que uma entidade precisa ser representada em Zeus, porém é dispensável para Hera. Este é o caso da entidade *PessoaCol*, que pode ser visualizada na Figura 25. A entidade *PessoaCol* surgiu da necessidade de representar uma coleção de pessoas da qual se escolherá a pessoa para ser responsável pela *Tarefa*. Para representar esse caso na tela podem estar envolvidos até quatro elementos de modelo: o *Composite* que irá preencher a caixa de seleção, o valor a ser apresentado na caixa de seleção, o valor a ser atribuído pela seleção de um elemento, o elemento de modelo que irá receber esse valor (no exemplo da Figura 25 o elemento *nome* de *Pessoa* desempenha dois papeis).

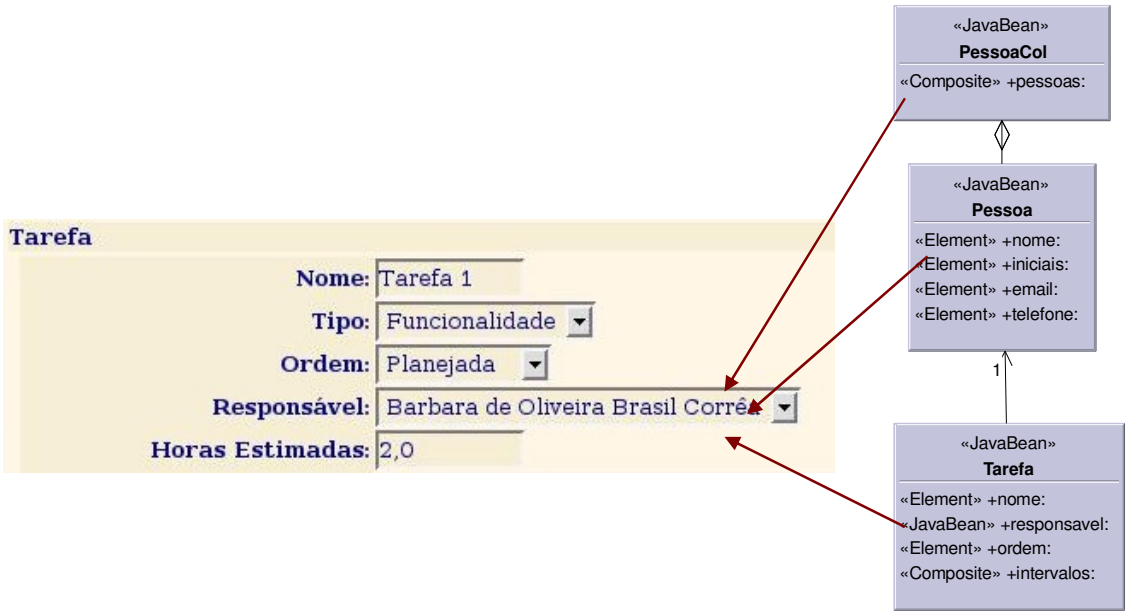


Figura 25 – Parte do domínio de Chronos em Zeus

Apesar das diferenças, a informação transita entre as camadas. Para que uma camada compreenda a informação transmitida de outra camada, mesmo estando em formato diferente, alguma coisa deve ser comum a ambas. Se uma entidade *Iteracao* em Zeus tem uma propriedade *nome* do tipo *String*, uma entidade *Iteracao* em Hera não poderá ter uma propriedade *nome* do tipo *Integer*. Um XML semelhante ao apresentado abaixo é utilizado como meio de transporte da informação entre Zeus e Hera.

```
<JavaBean class="Projeto" delete="false">
  <properties>
    <property name="iteracoes" value="" type="java.util.Collection">
      <collection>
        <JavaBean class="Iteracao" delete="false">
          <properties>
            <property name="descricao" value="" type="java.lang.String" />
            <property name="dataInicio" value="13/09/2003" type="java.lang.String" />
            <property name="nome" value="Iteracao 1" type="java.lang.String" />
            <property name="dataFim" value="15/09/2003" type="java.lang.String" />
            <property name="estorias" value="" type="java.util.Collection">
              <collection>
                <JavaBean class="Estoria" delete="false">
                  (...)
                </JavaBean>
              </collection>
            </property>
          </properties>
        </JavaBean>
        <JavaBean class="Iteracao" delete="false">
          (...)
        </JavaBean>
      </collection>
    </property>
  </properties>
</JavaBean>
```

Trabalhar diretamente com o domínio do sistema é um desperdício de esforço uma vez que cada parte do sistema utiliza apenas um subconjunto dessa informação. Além disso, corre-se o risco de sobrecarregar o domínio do sistema com particularidades de cada camada, fazendo com que outras camadas sejam obrigadas a lidar com essa sobrecarga. Essa interferência no domínio do sistema e suas implicações geram o caos entre as camadas. Esse mesmo caos pode ser observado entre os serviços, caso um interfira no domínio utilizado pelo outro. O importante é que cada parte do sistema veja o domínio de forma particular e esteja satisfeita com a visualização que teve, assim como na fábula dos cegos.

4 O DOMÍNIO E SUA REPRESENTAÇÃO EM HERA

Na Plataforma Hércules, Hera é a camada de controle da tríade MVC. Em Hera ocorre a execução das regras de negócio dos casos de uso. É nesta camada que o programador concentra a maior parte do seu esforço na implementação de um caso de uso. Por isso o domínio em Hera deve ser bem representado para facilitar o desenvolvimento bem como posteriores modificações.

A informação pretendida por Hera não corresponde necessariamente à utilizada por Zeus. O domínio em Hera é aquele indispensável à aplicação das regras de negócio. Informações em Hera podem não estar contidas em Zeus e vice-versa. No entanto, eventualmente em casos de uso simples, podem ser idênticas. Esse conteúdo depende do que deve ser apresentado e do que deve ser processado pelas regras do caso de uso.

A diversidade de informações encontra-se também entre serviços[2] em Hera. Apesar de existir entidades de domínio participando de mais de um serviço, a informação pretendida em cada um deles poder ser diferente. Se cada serviço acomodar a informação que precisa em uma entidade compartilhada obrigará aos demais se adaptarem a tais informações num efeito dominó gerando o caos no sistema.

A maneira tradicional de lidar com informação termina poluindo toda a implementação do sistema. O impacto dessa poluição generalizada ecoa na manutenibilidade do sistema. A proposta desse trabalho é transladar a informação entre Zeus e Hera utilizando em cada camada um modelo que possa refletir melhor as informações convenientes. E ainda individualizar o domínio utilizado em cada serviço em Hera, de forma que cada serviço visualize apenas a face do domínio indispensável para a execução de suas tarefas.

[2] Neste contexto, serviço é entendido como a parte do caso de uso de aplicação de regras de negócio, localizado em Hera.

Hera acumula também a tarefa de embutir no domínio algumas informações indispensáveis para Zeus. Essas informações são, por exemplo, atributos somente de leitura que não podem ser alterados ou atributos que são parâmetros de consulta. Tais informações podem ser configuradas e até mesmo ser extraídas de um banco de dados. Em Hera foi desenvolvido um mecanismo capaz de absorver essas configurações sem comprometer a integridade do domínio. Essas informações são utilizadas principalmente para automatizar algumas tarefas do serviço que ocorrem com freqüência, diminuindo o trabalho do programador. A informação deve ser tratada para que quando chegue ao programador este apenas se preocupe com a aplicação das regras de negócio.

A preparação do domínio em Hera é como uma maquiagem. A face do domínio utilizada em um serviço é maquiada para refletir as informações complementares necessárias ao caso de uso. Assim cada serviço embute as informações que precisa sem interferir no domínio de outros serviços. A integridade do domínio do sistema fica preservada.

Para viabilizar essa idéia foram desenvolvidos componentes em Hera com o objetivo de manipular a informação. Os participantes do processo de transformação desempenham metaforicamente os papéis de maquiador, esteticista e cosmético. O esteticista estipula como vai ser a maquiagem. O maquiador faz a maquiagem no domínio utilizando os cosméticos. O domínio pode ser maquiado com tantos cosméticos quanto forem preciso para complementar a informação necessária ao caso de uso.

O modelo adotado para representar o domínio em Hera tem como características primárias absorver a informação conveniente de Zeus, acomodar informação complementar e fornecer uma visão clara e particular do domínio para um determinado serviço. O modelo utilizado em Zeus translada para o modelo utilizado em Hera e vice-versa. A camada Olimpo ainda se encontra em fase de pesquisa, portanto sem modelo específico para translação. Essa abordagem localiza as alterações do domínio no escopo de um serviço, sem que essas alterações possam prejudicar a o domínio do sistema como um todo.

4.1 Face do Domínio em Hera

O domínio em sistemas de grande porte tende a ser volumoso. No entanto, um caso de uso utiliza, de fato, apenas um pequeno subconjunto dessa informação. Durante o desenvolvimento de um caso de uso pode surgir necessidade de adicionar informações que originalmente não pertencem ao domínio do sistema, mas facilitam a implementação do caso de uso. A combinação de informações trazidas da persistência com informações adicionadas na implementação formam o domínio de um serviço. Esse domínio deve ser representado por um modelo claro e simples para o programador e conter a informação essencial para a execução do serviço.

Para ilustrar como o domínio do sistema pode se modificar até chegar ao domínio de um caso de uso são apresentadas a Figura 26 e a Figura 27 contendo algumas entidades participantes do caso de uso Inscrição em Disciplinas do SIGA. A Figura 26 mostra entidades do domínio do sistema relacionadas a Turma. Entidades como *AtividadeAcademica*, *Curso* e *Servidor* possuem muitos atributos dispensáveis ao caso de uso. A Figura 27 representa o domínio utilizado pelo caso de uso. Este domínio é um subconjunto do domínio do sistema. Além disso, entidades pequenas podem ser representadas como atributos em suas associações, como, por exemplo, o atributo *situacao* em *Turma* e o atributo *nivel* em *Curso* da Figura 27.

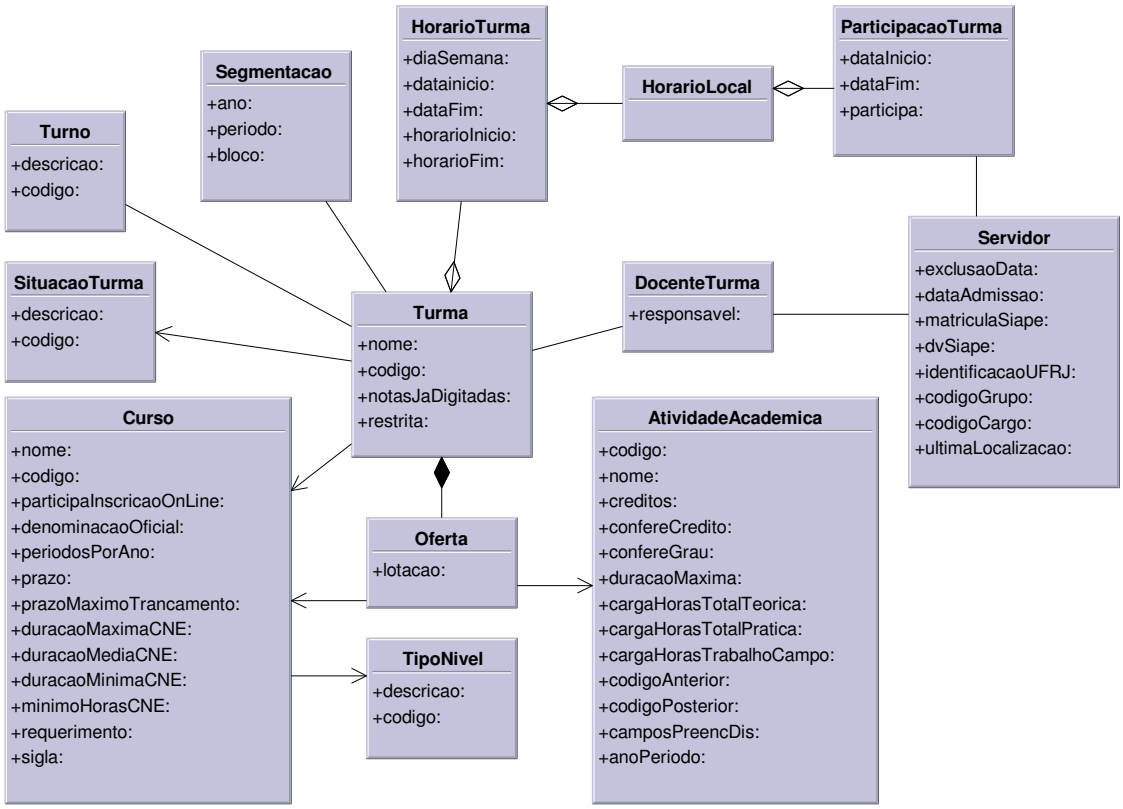


Figura 26 - Parte do domínio do sistema com entidades participantes do caso de uso de Inscrição em Disciplinas

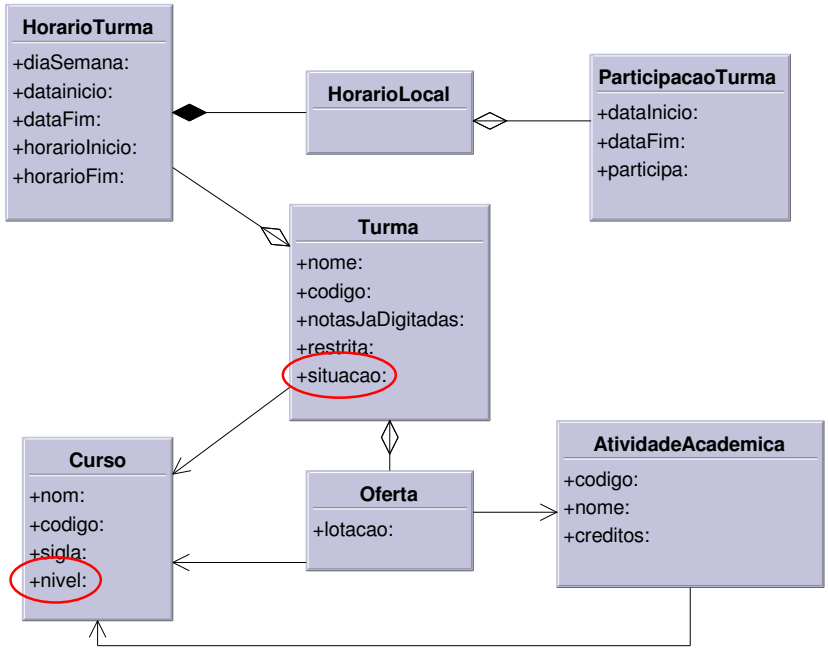


Figura 27 - Parte do domínio do caso de uso de Inscrição em Disciplinas

A construção do domínio de um caso de uso envolve conhecimento de como a entidade foi construída e de qual fonte os dados foram obtidos. A preservação desse conhecimento permite que algumas tarefas possam ser automatizadas. Para isso o modelo empregado na representação desse domínio deve ser capaz de guardar tais informações. O meta-modelo utilizado em Zeus, apesar de poder ser utilizado em Hera, não oferece os recursos adequados, pois sua implementação é voltada para os requisitos de apresentação.

Uma alternativa muito utilizada em sistemas que utilizam a Plataforma Hércules é a representação do domínio através de classes seguindo o padrão *ValueObject* (VOs). Os VOs são classes simples que guardam apenas os valores que devem ser persistidos. O desenvolvimento de casos de uso com Vos é mais ágil do que com um meta-modelo, como o utilizado em Zeus. A utilização de VOs permite que classes e atributos sejam acessados diretamente. Contudo, associar informação complementar a essas classes e atributos se torna mais complicado em relação a um meta-modelo.

O modelo para o domínio em Hera precisa ter como características a agilidade dos VOs e a flexibilidade de um meta-modelo. A preservação do conhecimento utilizado na construção das entidades utilizadas em um caso de uso permite que tarefas como recuperação de dados e persistência possam ser automatizadas. Essa abordagem não somente melhora o desenvolvimento, como também aumenta a manutenibilidade do sistema.

4.2 Construindo o domínio do serviço

O desenvolvimento de serviços utilizando VOs representou uma importante fonte de estudo de como o domínio de um serviço é formado. Analisando a composição de VOs de um serviço foram observadas significativas diferenças em relação aos modelos utilizados nas outras camadas. O conteúdo de um VO pode ser um subconjunto de dados originados da persistência combinados com dados novos. A experiência com VOs enriqueceu esta pesquisa trazendo a tona algumas características para um modelo adequado para Hera. O modelo em Hera deve ser capaz de:

- Juntar mais de um atributo de entidades do sistema para formar um atributo da entidade do caso de uso.
- Ter atributos de diferentes entidades do sistema em uma mesma entidade do caso de uso.
- Indicar o tipo de elemento que fará parte de uma coleção.
- Indicar o tipo de um elemento associado (multiplicidade 1).
- Indicar atributos que não fazem parte de entidades do sistema.
- Indicar conjunto de valores (atributos ou entidades) que podem preencher um atributo da entidade do caso de uso.

Com base nestes requisitos este trabalho propõe uma arquitetura para a descrição do modelo em Hera. A Figura 28 mostra o diagrama com as classes utilizadas para a representação da descrição do modelo. A classe *Stunt*[3] representa uma entidade de um serviço. Esta classe é composta de propriedades simples e propriedades que representam associações entre *Stunts*. Cada propriedade simples, representada pela classe *SimpleProperty*, pode ser constituída por qualquer combinação de campos de entidades fonte com textos simples. A classe *SourceEntity* representa entidades fonte, ou seja, entidades do domínio do sistema de onde são extraída informações para formar o domínio do serviço. Os relacionamentos entre entidades fonte são indicados através da classe *Relationship*. A descrição do domínio (*DomainDescription*) é composta de classes do tipo *Stunt* e *SourceEntity*.

[3] *Stunt* quer dizer dublê em inglês. Neste contexto, o dublê poupa a entidade de domínio do sistema de participar diretamente em um serviço.

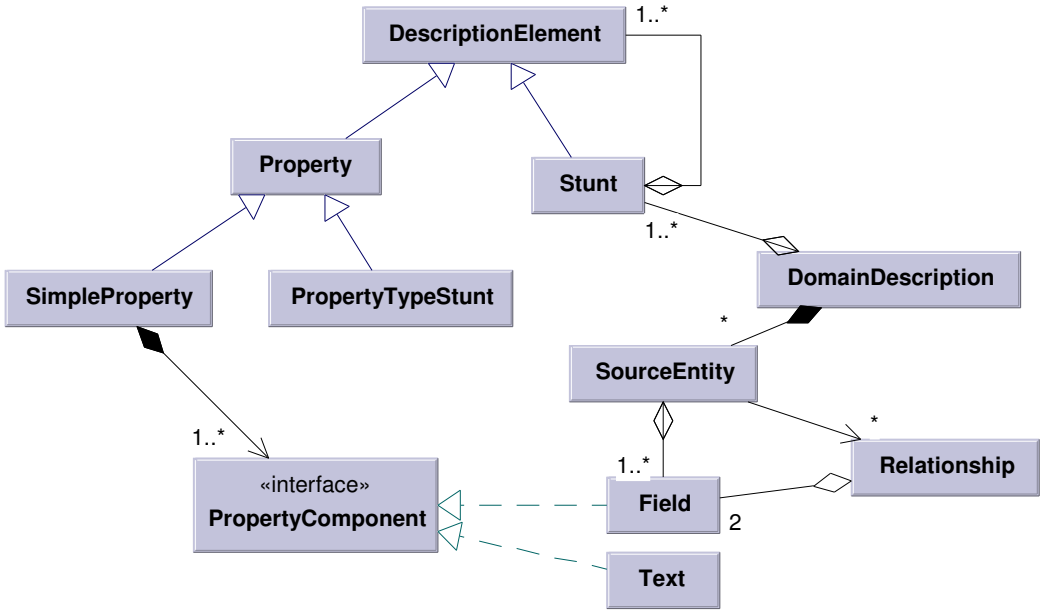


Figura 28 - Diagrama de classes da descrição do domínio de Hera

Com esse conjunto de classes é possível armazenar a informação necessária para a construção do domínio. A Figura 29 mostra, a título de ilustração, como é construído o domínio para o caso de uso Inscrição em Massa. Este caso de uso tem como objetivo definir um plano de estudos para alunos de um determinado curso. O plano de estudos tem uma coleção de ofertas planejadas para inscrição e é válido para uma determinada segmentação[4]. Poucos atributos das entidades de domínio do sistema aparecem no domínio deste caso de uso. Em particular, o atributo *segmento* em *Segmentacao* representa uma junção de três atributos da entidade *SegmentacaoEntityBean*, separados pelo caractere "-".

[4] Segmentação é o nome que se dá, no SIGA, para o período de tempo que pode ser identificado por um ano, um período e um bloco.

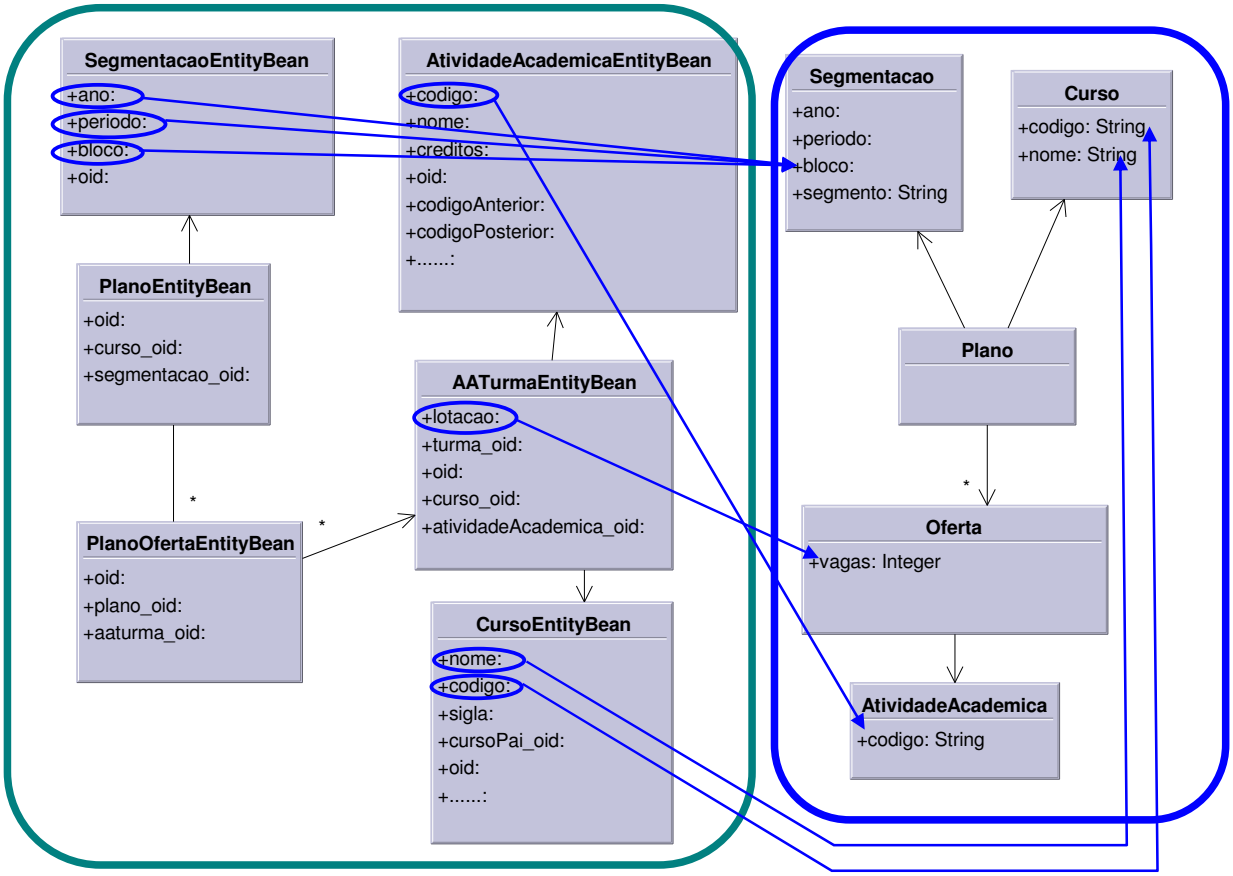


Figura 29 - Construção do domínio do caso de uso de Inscrição em Massa

Além da instanciação direta das classes do diagrama da Figura 28, a informação de construção do domínio pode também ser obtida através de XML. Para isso foi especificado um formato para o XML para armazenar as informações desejadas. O XML abaixo se refere ao caso de uso de Inscrição em Massa.

```
<query-stunt>
  <source>
    <entity name="PlanoEntityBean" alias="plano" />
    <entity name="CursoEntityBean" alias="curso" />
    <entity name="PlanoOfertaEntityBean" alias="planoOferta" />
    <entity name="AATurmaEntityBean" alias="oferta" />
    <entity name="AtividadeAcademicaEntityBean" alias="atividadeAcademica" />
    <entity name="SegmentacaoEntityBean" alias="segmentacao" />
  </source>
  <relationship>
    <equals>
      <field name="curso_oid" entity="plano" />
      <field name="oid" entity="curso" />
    </equals>
    <equals>
      <field name="oid" entity="plano" />
      <field name="plano_oid" entity="planoOferta" />
    </equals>
    <equals>
      <field name="oid" entity="oferta" />
      <field name="aaturma_oid" entity="planoOferta" />
    </equals>
    <equals>
      <field name="atividadeAcademica_oid" entity="oferta" />
```

```
      <field name="oid" entity="atividadeAcademica" />
    </equals>
    <equals>
      <field name="segmentacao_oid" entity="plano" />
      <field name="oid" entity="segmentacao" />
    </equals>
  </relationship>
  <presentation>
    <stunt name="inscricaoEmMassa.model.Plano" alias="plano">
      <property name="nome" type="java.lang.String">
        <field name="nome" entity="plano" />
      </property>
      <property name="curso" type="inscricaoEmMassa.model.Curso">
        <stunt name="inscricaoEmMassa.model.Curso">
          <property name="nome" type="java.lang.String">
            <field name="nome" entity="curso" />
          </property>
          <property name="codigo" type="java.lang.String">
            <field name="codigo" entity="curso" />
          </property>
        </stunt>
      </property>
      <property name="ofertas" type="java.util.Collection">
        <stunt name="inscricaoEmMassa.model.Oferta">
          <property name="vagas" type="java.lang.Integer">
            <field name="lotacao" entity="oferta" />
          </property>
          <property name="disciplina" type="inscricaoEmMassa.model.AtividadeAcademica">
            <stunt name="inscricaoEmMassa.model.AtividadeAcademica">
              <property name="codigo" type="java.lang.String">
                <field name="codigo" entity="atividadeAcademica" />
              </property>
            </stunt>
          </property>
        </stunt>
      </property>
      <property name="segmentacao" type="inscricaoEmMassa.model.Segmentacao">
        <stunt name="inscricaoEmMassa.model.Segmentacao">
          <property name="segmento" type="java.lang.String">
            <field name="ano" entity="segmentacao" />
            <text value="-" />
            <field name="periodo" entity="segmentacao" />
            <text value="-" />
            <field name="bloco" entity="segmentacao" />
          </property>
        </stunt>
      </property>
    </stunt>
  </presentation>
</query-stunt>
```

O meta-modelo apresentado guarda o conhecimento de construção do domínio. Este meta-modelo é utilizado por Hera para permitir automatização de algumas tarefas comuns. A partir dessa informação, Hera constrói o domínio para o serviço. O programador continuará trabalhando com objetos simples no padrão *ValueObject*. Hera se encarrega de atualizar o modelo utilizado pelo programador. No entanto, o serviço pode precisar adicionar mais informação ao domínio. Neste caso a informação será adicionada ao meta-modelo e não aos VOs. Esse gênero de informação que o serviço precisa adicionar ocorre com freqüência em muitos casos de uso. A configuração de tarefas para adicionar tais informações é desejável para automatizar o processo. A utilização de componentes para desempenhar tais tarefas se mostrou o mecanismo mais apropriado.

4.3 Programação orientada a componentes

A programação orientada a objetos é a base para a tão almejada reutilização. Contudo, é preciso esmero ao projetar classes para desempenhar determinada tarefa quando a intenção é utilizá-las em diversos projetos. Uma classe muito dependente de outras pode ser comparada a um prato de espaguete: é difícil puxar um só fio de espaguete sem trazer outros juntos. Por isso a reutilização está mais relacionada com a disciplina de programação. Deve-se tomar o cuidado de desenvolver classes simples, com baixo acoplamento e com uma tarefa específica. Essas, entre outras, são características de componentes.

Atualmente está em voga a Programação Orientada a Componentes (Component Oriented Programming, COP). COP não representa uma mudança de paradigma, apenas tem como princípio o desenvolvimento de componentes visando a reutilização. Componentes são pequenas entidades responsáveis por elas mesmas que objetivam ser totalmente substituíveis sem esforço. Um componente é decomposto em sua interface e sua implementação. Sua interface é o que será visível ao mundo externo, ou seja, as aplicações que farão uso do serviço. Sua implementação é uma caixa preta, o que permite sua fácil substituição.

Parsons, em seu artigo (PARSONS, 2005), enumera as características de um componente:

- Tem interface simples e bem definida
- É um objeto, sendo que seus dados e métodos são combinados formando uma unidade.
- Exibem um nível de especialização de funcionalidade (freqüentemente obtidos através de configuração), com métodos apropriados de ciclo de vida para prover a funcionalidade desejada.
- É desenvolvido na expectativa de ser reutilizado, embora o contexto de reuso seja imprevisível.

A utilização de componentes em Hera é apropriada para adicionar informação complementar ao domínio de um serviço. O domínio obtido a partir da descrição das entidades participantes de um caso de uso está preparado para agregar novas informações. O serviço detém o conhecimento de que gênero de informações irá

precisar. Essas informações caem sobre o modelo como uma maquiagem. Fazendo uma analogia, um determinado serviço pode maquiar o domínio com baton, blush e sombra, outro pode apenas maquiar com baton. Para cada cosmético existe um componente que sabe como utilizá-lo para maquiar a face do domínio. Um conjunto de componentes é disponibilizado para o serviço escolher quais atuarão sobre seu domínio.

Características como facilidade de substituição e possibilidade de configuração ampliam o poder de representação do domínio em Hera. Uma mesma informação pode chegar ao domínio de diferentes formas apenas trocando o componente utilizado por outro semelhante. No entanto, a configuração dos componentes deve ser feita de forma ordenada por um mecanismo capaz de gerenciar os componentes.

4.4 A camada de modelo em Hera

No círculo de desenvolvimento de softwares orientados a objetos o termo acoplamento é bastante conhecido. “O acoplamento (...) refere-se ao grau de interdependência entre diferentes classes de seu projeto, ou seja, ao quanto uma determinada classe depende de outras classes.” (VILLELA, OLIVEIRA, VERÍSSIMO, 2004). Quanto mais baixo for o acoplamento maior é a flexibilidade de reutilização do código. Somente a utilização de componentes não garante o baixo acoplamento das classes, pois existem diversas maneiras de implementar a comunicação entre as elas. O acoplamento fraco é uma conseqüência do uso do padrão “Injeção de dependência” também conhecido como “Inversão de controle”.

A injeção de dependência faz uso de uma classe externa responsável pelo gerenciamento dos componentes. Essa classe, chamada de montador (*assembler*), conhece os componentes envolvidos em um determinado serviço e gerencia seus ciclos de vida, configurações e dependências. O montador faz invocações nas classes nos momentos certos com um propósito específico. Existem disponíveis no mercado de software diversas implementações na forma de containers leves para esse padrão.

Na camada de modelo de Hera, componentes são construídos para executarem alguma tarefa sobre o domínio do caso de uso. No entanto, a camada de modelo

precisa estar preparada para que não ocorra um acoplamento forte entre as classes. É preciso saber o momento adequado de utilizar cada componente e gerenciar seus ciclos de vida. Neste contexto, a adoção do padrão Injeção de dependência é adequada como solução.

Dentre os containers de injeção de dependência disponíveis no mercado o *PicoContainer* foi escolhido para fazer parte deste trabalho. Como o próprio nome sugere o PicoContainer é muito leve e simples de ser usado. Este container foi utilizado como base para a construção de um container mais especializado para gerenciamento dos componentes de Hera. O container em Hera desempenha o papel de maquiador gerenciando os componentes que representam cosméticos para a face do domínio.

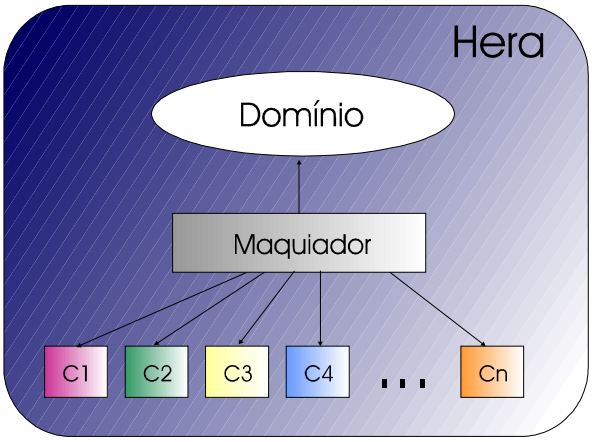


Figura 30 - Container de componentes de Hera

O maquiador obtém o meta-modelo de descrição do domínio a partir de um componente do tipo *DescriptionReader* nele registrado. Por enquanto, apenas um componente desse tipo foi implementado. Este componente lê uma descrição do domínio em XML e constrói o meta-modelo (DomainDescription). Com base nessa descrição o domínio é construído ficando assim pronto para a aplicação de cosméticos.

A aplicação de cosméticos é feita seguindo a idéia do controle de ciclo de vida implementado no PicoContainer, onde os componentes que participam desse controle implementem a interface *Startable*. Esses componentes são instanciados

pelo container e recebem uma chamada para o método *start*. Em Hera foram desenvolvidos dois tipos de controle: marcação e aplicação de cosméticos. A marcação de cosméticos identifica os elementos de modelo que devem ser marcados e realiza a marcação. A aplicação analisa a necessidade de alterações no domínio com base nos elementos marcados, por exemplo, criação de entidades auxiliares.

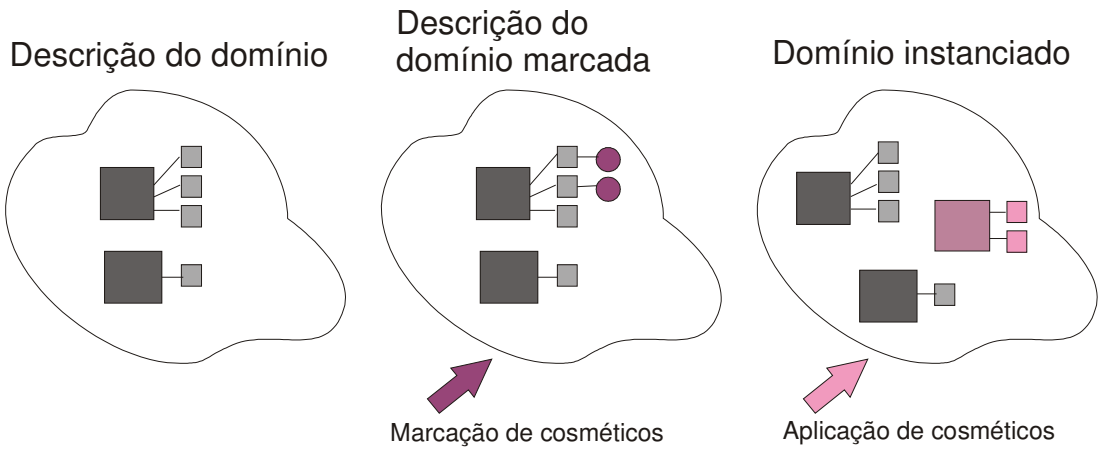


Figura 31 - Etapas de marcação e aplicação de cosméticos.

A utilização de um container de injeção de dependência possibilitou que o domínio ficasse completamente desacoplado dos componentes. O PicoContainer, apesar de simples, ofereceu as funcionalidades suficientes para a implementação de um container para gerenciamento dos componentes de Hera. Os componentes são instanciados pelo container, segundo prévia configuração, e aplicados sobre o domínio do serviço.

4.5 Caracterização do Modelo

A descrição do domínio na camada de modelo de Hera permite a construção de um modelo simples e adequado para a interação do programador. Neste domínio estarão presentes as informações imprescindíveis à aplicação das regras de negócio do caso de uso. Entretanto, a camada de modelo pode executar outras operações onde seja necessária a complementação desse domínio com mais informações. Para isso o domínio precisa estar preparado com a informação necessária à

execução de tais operações. Preservada a visão simplificada para o programador, a camada de modelo deverá oferecer um domínio modificado, caracterizado de acordo com o papel que deve desempenhar no contexto das operações internas.

As informações são adicionadas no domínio através de componentes registrados no maquiador. Essas informações são adicionadas ao domínio construído a partir da descrição e não na descrição, pois a descrição é utilizada durante todo o caso de uso para construir ou atualizar o domínio sempre que necessário. Um exemplo de operação da camada de modelo é “voltar valor antigo”. Para essa operação é necessário guardar o valor antigo dos atributos que deverão ser restaurados.

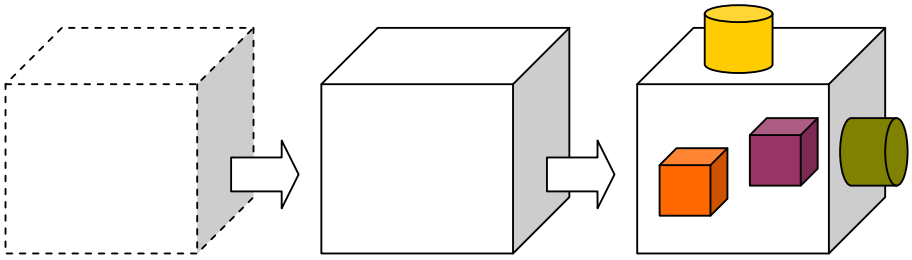


Figura 32 – Descrição, domínio Básico e domínio caracterizado

Para proceder à maquiagem o maquiador precisa de um roteiro de maquiagem. O roteiro é estabelecido pelo registro de componentes no maquiador. A maquiagem é feita utilizando a metáfora de cosméticos. O maquiador sabe o momento certo de realizar a maquiagem. Após a maquiagem o domínio fica pronto para a execução das operações.

4.6 Cosméticos

A construção de um meta-modelo com a informação de domínio facilita a manipulação de dados e a translação do domínio entre camadas do *framework* Hércules. Na camada Hera, o domínio representado pelo meta-modelo é utilizado para aplicação de regras de negócio do caso de uso. Entretanto, existe outro tipo de regra que estão dissociadas do negócio de caso de uso. Estas, em geral, indicam comportamento de um atributo ou uma entidade do domínio do caso de uso.

O registro desse comportamento é útil tanto para a camada Hera como para a translação do domínio para outras camadas. O controle de escopo é um exemplo em que uma regra é utilizada na camada Hera. Um atributo que possui uma regra de controle de acesso segundo uma localidade precisa passar pelo crivo de um controlador de escopo situado em Hera. Uma regra de um atributo que precisa ser preenchido a partir da escolha de um elemento dentro de um conjunto é importante na translação do domínio para a camada Zeus.

Este trabalho apresenta uma proposta para a representação desse comportamento. Essas regras são traduzidas em marcações no meta-modelo do domínio do caso de uso. A marcação é feita num passo intermediário entre a montagem do domínio e a utilização, de fato, da informação contida nele.

A proposta desse trabalho se baseia na metáfora de uma maquiagem onde os cosméticos marcam a face, neste caso a face do domínio que é enxergada pelo caso de uso em Hera. Podem ser desenvolvidos tantos cosméticos quanto forem preciso. Cada cosmético aplica um determinado comportamento no domínio. Por exemplo, um cosmético de consulta marca os elementos do domínio que são parâmetros de consulta. Até o momento foram implementados dois tipos de cosméticos, o de consulta e o de restrição de acesso.

4.6.1 Pesquisa e apresentação de resultado

A informação manipulada pelos Sistemas de Informação deve estar a serviço dos usuários atendendo suas necessidades. O usuário deve ser capaz de obter a informação que deseja de forma eficiente. Dentro do universo de informação de um sistema, o usuário pode definir critérios para selecionar um subconjunto de seu interesse. Por exemplo, em um sistema de gerência de projeto, um gerente pode estar interessado em saber quais são as tarefas em atraso. Ou seja, neste caso o usuário deseja obter informação baseada no critério data de conclusão da tarefa.

No sistema SIGA é comum ocorrer casos de uso apresentando uma etapa de pesquisa da informação e outra subseqüente de apresentação de resultado. Esta última etapa apresenta os itens correspondentes ao resultado da pesquisa de acordo com os critérios especificados. Quando um item de resultado é selecionado o

sistema o prepara para a etapa de edição. É geralmente nesta etapa que estão concentradas as informações necessárias a aplicação das regras de negócio do caso de uso. Em Hera as etapas de pesquisa e apresentação são meras auxiliares na obtenção da informação necessária a aplicação das regras de negócio. As figuras abaixo mostram o conteúdo das abas de Seleção e Lista do caso de uso Previsão de Turmas do SIGA. Na aba Edição, o usuário pode visualizar uma turma selecionada na aba de Lista e alterá-la ou criar uma nova turma. É nesta etapa que ocorrem as operações principais deste caso de uso, ou seja, criação e alteração de turmas.

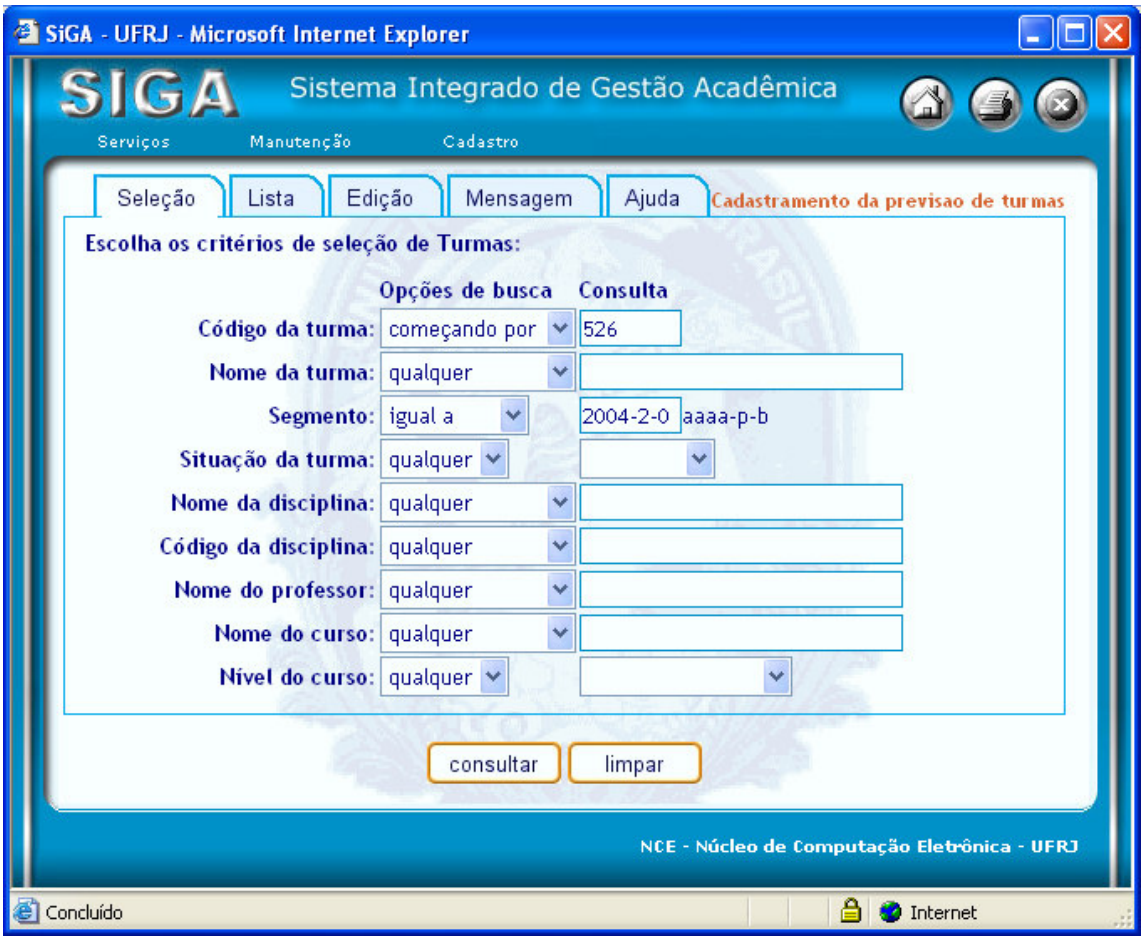
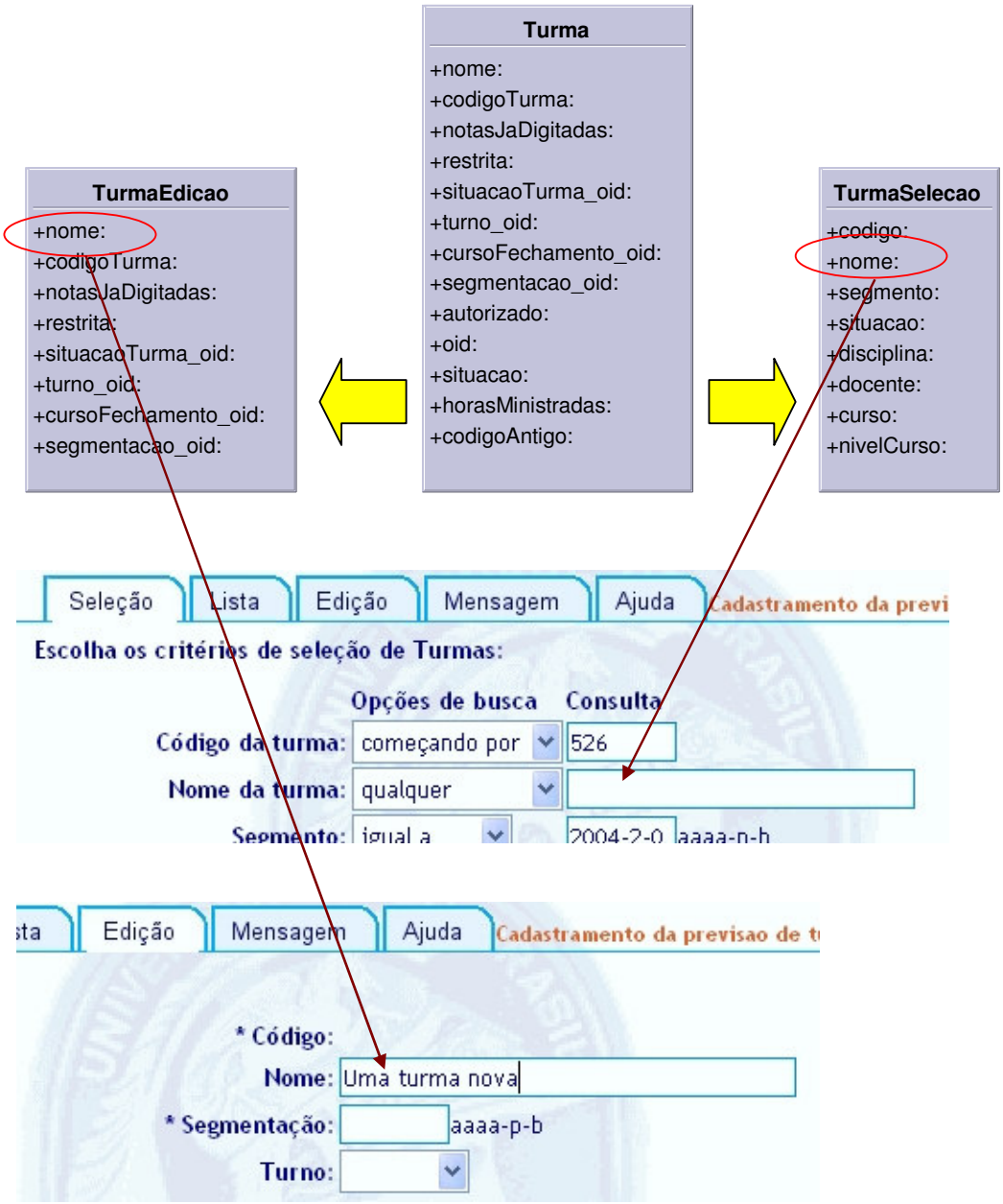


Figura 33 - Seleção da Previsão de Turmas

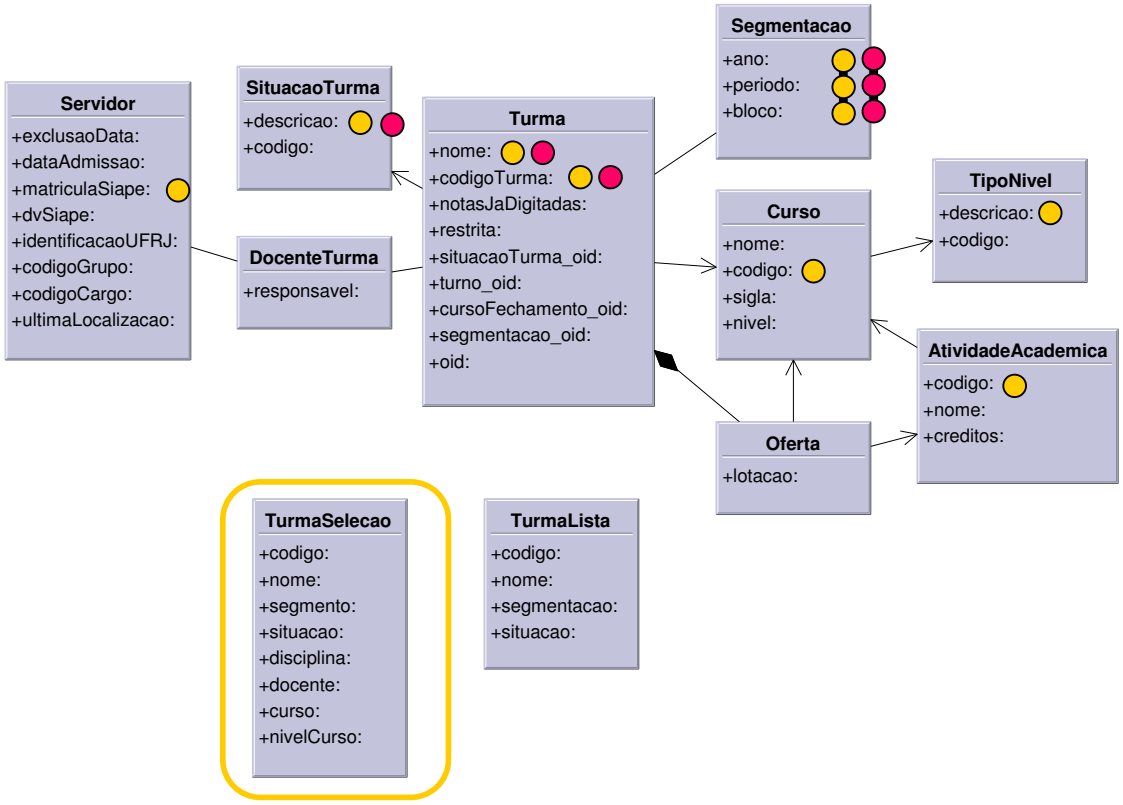
Figura 34 - Lista de resultados da Previsão de Turmas

A visão dessas etapas na camada Zeus é diferente. Enquanto Hera visualiza como domínio a informação necessária à aplicação de regras de negócio, Zeus entende como domínio toda informação que precisa ser apresentada na tela. Por esse motivo o modelo em Zeus precisa discriminar as entidades envolvidas na pesquisa e apresentação de resultado. Apesar dessas entidades possuírem atributos teoricamente comuns com as entidades da etapa de edição na prática eles precisam ser distintos, pois os componentes visuais estão diretamente associados a elementos de modelo. Se um atributo da etapa de pesquisa fosse o mesmo da etapa edição as duas etapas refletiriam o mesmo valor. Como geralmente não é esse o efeito desejado, é definida uma entidade para a etapa de pesquisa, outra para etapa de apresentação de resultado e outra para edição.



Entretanto, Hera é responsável por enviar a informação que Zeus precisa para construir seu modelo. Ao mesmo tempo a informação que Zeus precisa é dispensável para Hera. Com o objetivo de evitar o aparecimento de informação desnecessária à aplicação de regras de negócio foi desenvolvido um cosmético especial. Esse cosmético marca o domínio identificando os elementos que representarão critérios de pesquisa e os elementos que farão parte da apresentação de resultado.

Em alguns casos de uso do SIGA foi observada a presença de duas pesquisas, por isso o domínio deve ser capaz de ser marcado por mais de um cosmético de pesquisa e apresentação. Um cosmético desse tipo possui uma identificação que dará origem a duas entidades auxiliares: uma para a pesquisa e outra para a apresentação do resultado.



O cosmético que marca os elementos de modelo participantes da pesquisa e apresentação de resultado evita o aparecimento de informação inconveniente no momento da aplicação de regras de negócio. Além disso, facilita a configuração e possíveis alterações de tais propriedades. Com base nessa marcação é possível transladar para Zeus o domínio apropriado para a apresentação do caso de uso na GUI.

4.6.2 Restrições de acesso

O perfil de um usuário de um Sistema de Informação indica o que este usuário pode acessar no sistema. Uma informação pode oferecer permissão de escrita para um

perfil e apenas de leitura para outro. Esta regra está relacionada ao perfil do usuário que está utilizando o serviço independente das regras de negócio do caso de uso.

No *framework* Hércules, a camada Zeus interpreta elementos de modelo com restrição de acesso. Um componente visual associado a um elemento de modelo com restrição “apenas leitura” é renderizado de forma que não seja permitida edição, mesmo que sua forma original seja editável. Por exemplo, o componente visual que representa uma caixa de texto que apresenta o CPF de uma pessoa e este CPF é “apenas leitura”, a caixa de texto renderiza como texto comum.

Essa informação quanto ao acesso está no modelo. A informação que chegar à camada Zeus vem da camada Hera. Porém a informação que não é interessante para a execução das regras de negócio de um caso de uso deve ser agregada com cuidado para não poluir o modelo de Hera.

Com o objetivo de isolar a informação de restrições de acesso do modelo utilizado pelo caso de uso foi implementado um cosmético especializado em guardar (registrar) tal informação. O cosmético de restrição de acesso simplesmente indica a acessibilidade de um elemento do modelo. Quando o modelo é transladado para Zeus essa informação é interpretada e incorporada ao modelo de Zeus.

A implementação do caso de uso em Hera fica enxuta, pois dispensa a preocupação com a restrição de acesso. Um componente foi desenvolvido no *framework* Hércules para lidar com a restrição de acesso baseada em perfil de usuário. Este componente pode ser utilizado por ser utilizado por qualquer caso de uso, bastando para isso registrar o componente no maquiador. Novos componentes podem ser desenvolvidos para restrições de acesso baseadas em outro critério.

5 AVALIAÇÃO NO ESTUDO DE CASO: INSCRIÇÃO EM MASSA

O problema da poluição do domínio em Sistemas de Informação de grande porte deu origem a uma proposta para a construção de domínios específicos para as diferentes camadas do sistema. Baseado no *framework* Hécules este trabalho apresenta uma arquitetura para as camadas de modelo de Zeus e Hera. A implementação dessa arquitetura permite a aplicação prática em um caso de uso real visando avaliar os benefícios esperados e possíveis dificuldades encontradas. O caso de uso a ser estudado é "Inscrição em Massa" do sistema SIGA, brevemente apresentado na seção 4.2.

Numa universidade os cursos são divididos em períodos e cada período tem uma grade de disciplinas proposta. A cada *inscrição em disciplinas*[5]*,* muitos alunos se inscrevem no mesmo grupo de disciplinas, principalmente no primeiro período do curso. O caso de uso Inscrição em Massa tem como objetivo definir um plano de estudos para alunos de um curso em um segmento. O plano de estudos reúne as disciplinas que os alunos serão inscritos. O serviço oferecido ao usuário possui dois momentos principais: a escolha de disciplinas que farão parte do plano de estudos e a inscrição de alunos neste plano. Esta avaliação observará apenas a etapa de definição do plano de estudos.

[5] Evento da universidade previsto em calendário acadêmico.

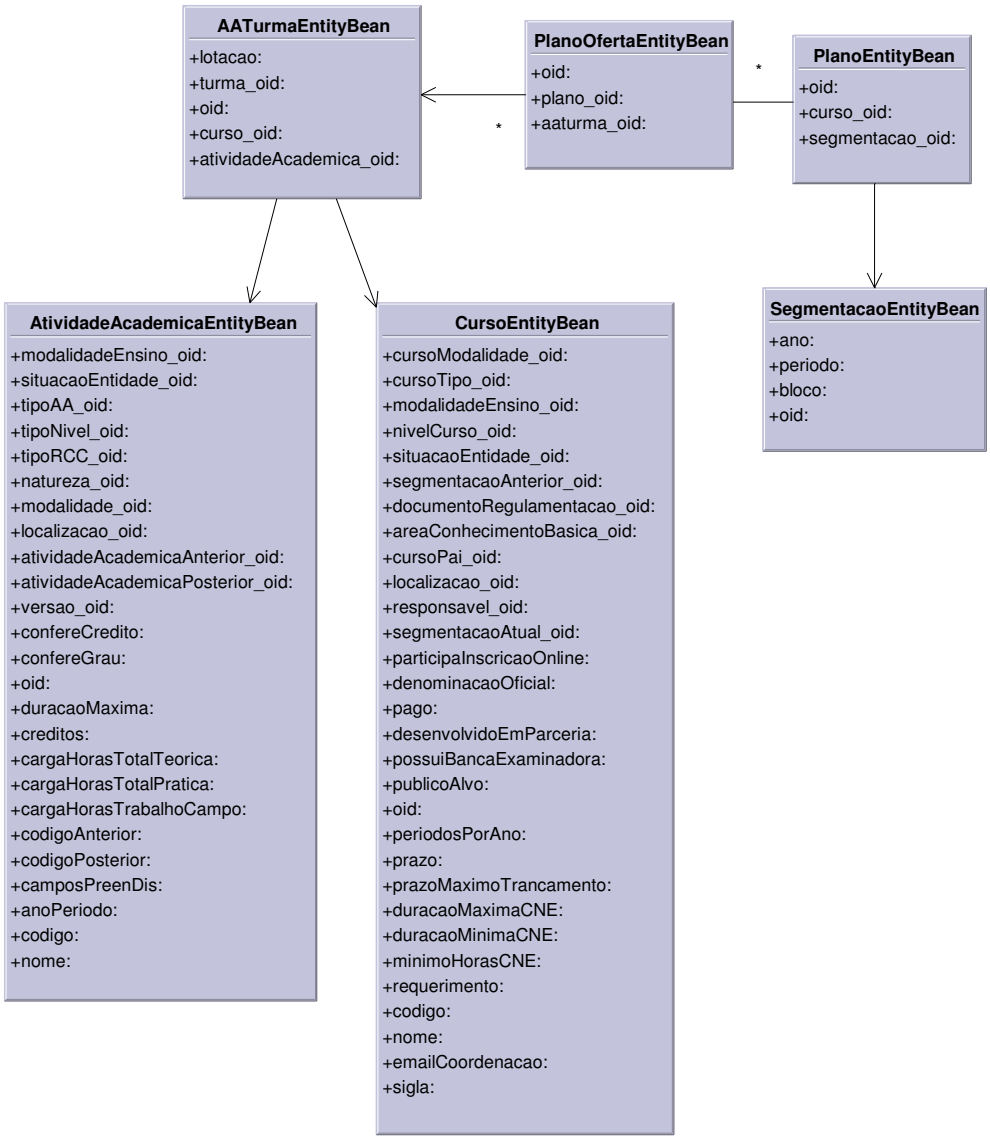


Figura 35 – Modelo de persistência de Inscrição em Massa

A Figura 35 apresenta o trecho da modelagem das entidades da camada de persistência que participam da Inscrição em massa.

Esta avaliação contempla dois aspectos do framework Hércules. O primeiro aspecto mostra as vantagens do framework Hércules em relação ao framework Struts, especificamente no que diz respeito ao modelo da camada de vista. Essa comparação é feita visto que o framework Hércules não existe sem o modelo de Zeus, o que torna inviável a comparação dentro do mesmo framework. Contudo, em Hera, é possível fazer tal comparação, pois o tratamento dado ao modelo em Hera

por este trabalho é uma sofisticação do modelo que era utilizado anteriormente. Este é o segundo aspecto desta avaliação.

5.1 Vantagens do framework Hercules em relação ao Struts

Como descrito na seção 2.8, o Struts utiliza *JavaBeans* para representar o modelo na camada de vista, em especial os *FormBeans* para obter a informação vinda na requisição. Utilizando o exemplo do caso de uso Inscrição em Massa, particularmente a tela de Edição de Plano de Estudos (Figura 36), serão analisados os modelos envolvidos.

Figura 36 - Tela Edição de Plano de Estudos

Para apresentar a tela de edição do Plano de Estudos, previamente selecionado em outra tela, está representada na Figura 37 uma possível estrutura de *JavaBeans*. Apenas uma parte da informação do modelo de persistência é utilizada na construção do modelo em Zeus.

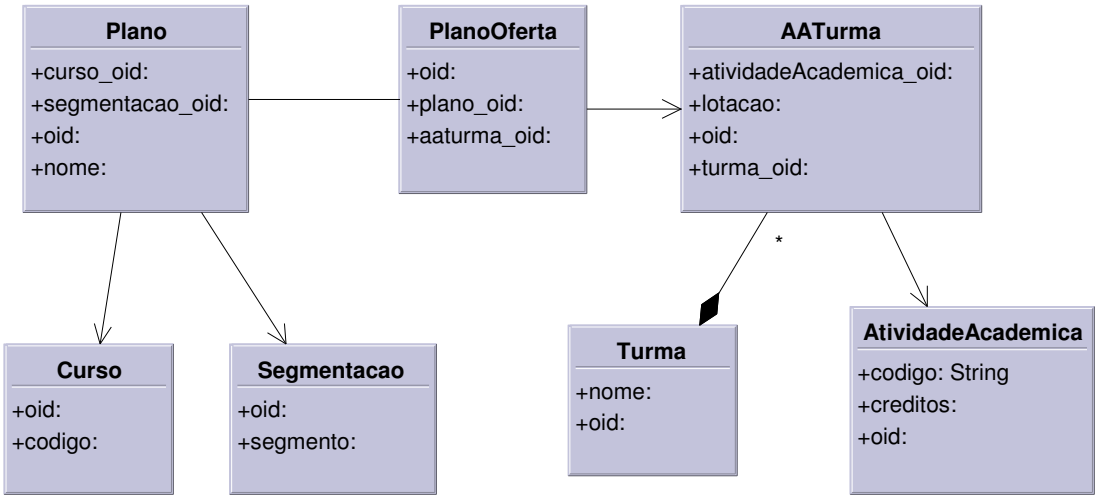


Figura 37- Modelo para apresentação da informação no Struts

O controle da aplicação disponibiliza os JavaBeans para as páginas JSPs acessar e apresentar os dados na tela. Quando o cliente faz a requisição, o framework STRUTS espera que um ActionForm seja preenchido com os dados desta requisição. O ActionForm é uma classe simples que tem atributos com os mesmos nomes dos parâmetros da requisição. A Figura 38 mostra como ficaria o ActionForm responsável por trazer os dados da tela de edição do Plano de Estudos.

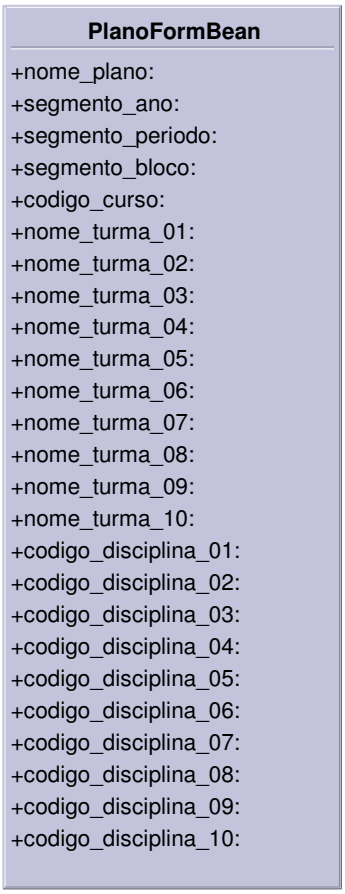


Figura 38 - FormBean que traz os dados da requisição para a edição de Plano de Estudos

A representação de modelo sugerida pelo STRUTS tem alguns inconvenientes. Os atributos que antes estavam separados em diversas classes, facilitando seu entendimento e programação, neste modelo encontram-se todos juntos. As disciplinas contidas no Plano de Estudos deveriam ser uma coleção de tamanho indefinido. No entanto, é necessário indicar como atributo no ActionForm os parâmetros que vêm na requisição para o framework poder armazenar. Além disso, o framework Hércules possui componentes de interface preparados para lidar com modelo que contenha relacionamentos, facilitando a programação da interface com o usuário e por vezes até economizando codificação. Um exemplo é a coleção de turmas do Plano de Estudos: basta indicar para o componente tabela que o que vai preenchê-lo será um modelo de coleção. Neste caso não é necessário fazer iteração dos elementos da coleção.

No framework Hércules é utilizado, para trazer os dados da requisição, o mesmo modelo que é utilizado para apresentar os dados na tela. A forma de representação do modelo no Hércules é muito mais rica, possibilitando a representação de relacionamentos sem necessidade de intervenção do programador para tratar os dados vindos da requisição. Os componentes da interface com o usuário foram desenvolvidos especialmente para lidar com algumas características dos modelos como, por exemplo, enviar na requisição o elemento selecionado em uma coleção.

5.2 Melhorias observadas em Hera

Na implementação do caso de uso em Hera, assim como em Zeus, apenas uma parte da informação do modelo de persistência é realmente utilizada. Somente a informação necessária à criação de um plano de estudos e à apresentação para o usuário será aproveitada na construção do modelo de Hera. Essa informação representa o domínio do caso de uso Inscrição em Massa. Como um modelo específico é criado para atender as necessidades do caso de uso, o formato adotado não influencia o modelo de outros casos de uso. A figura abaixo mostra o modelo na fase inicial de construção contendo as informações necessárias à criação do plano de estudos.

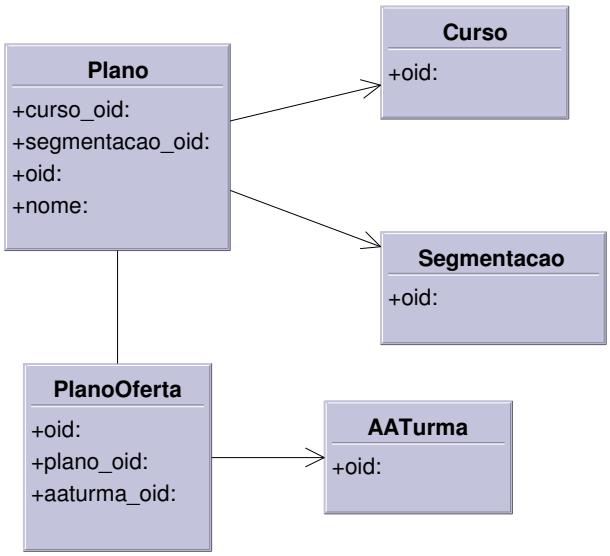


Figura 39 - Modelo inicial da Inscrição em Massa em Hera

A interface gráfica especificada pelo cliente do caso de uso orientará a complementação do modelo até então descrito. Com base na informação que deverá ser apresentada ao usuário, novos atributos e entidades podem ser adicionados ao modelo de Hera. A Figura 40 apresenta o modelo sugerido com base nas telas protótipo do Anexo II.

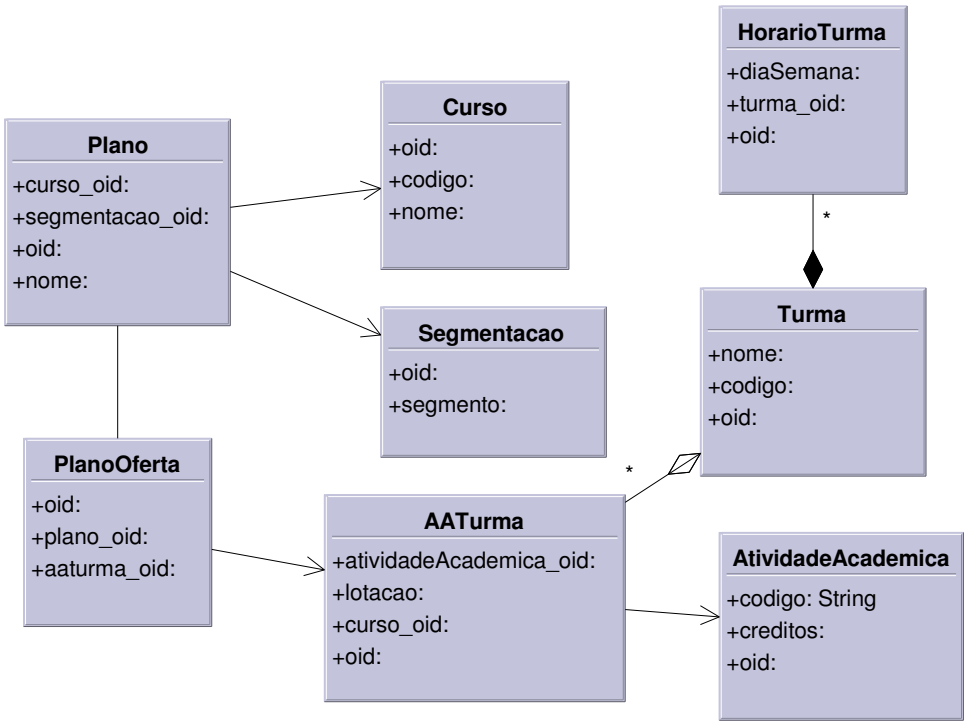


Figura 40 - Modelo com os atributos necessários à apresentação

Em Zeus os atributos de pesquisa e apresentação de resultado devem ser colocados em entidades auxiliares, distintas das entidades utilizadas na edição. Isso porque os componentes de vistas estão associados a elementos de modelo, o que torna possível um mesmo elemento de modelo estar sendo apresentado por mais de um componente de vista. Por exemplo, se um campo na aba de “Seleção” estiver associado ao mesmo elemento de modelo que um campo da aba “Edição”, quando alterar em uma aba automaticamente aparecerá alterado na outra também. As entidades da Figura 41 complementam o modelo de Hera apresentado na Figura 40 dando origem ao modelo requerido por Zeus.

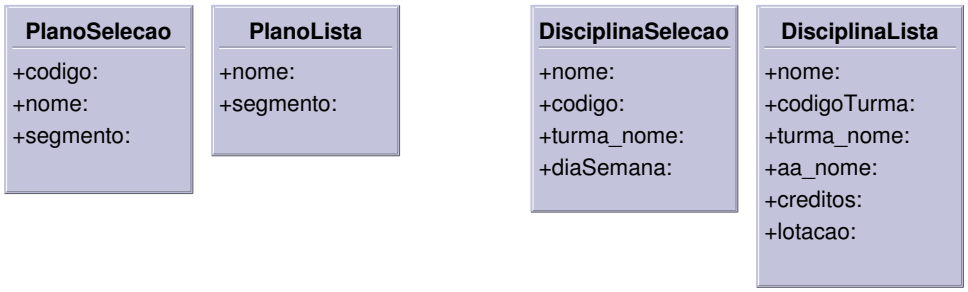


Figura 41 - Entidades complementares para o modelo de Zeus

Este trabalho propõe a construção do modelo de Hera a partir de uma descrição em XML. Esse modelo seria um subconjunto da informação contida no modelo da aplicação, no qual estão contidas todas as informações necessárias a toda a aplicação. Dessa forma, o modelo de Hera é construído voltado para o caso de uso em que está inserido, contudo o modelo original é preservado. O XML abaixo contém a descrição do modelo para o caso de uso Inscrição em massa.

```
<query-stunt>
  <source>
    <entity name="br.ufrj.siga.inscricao.PlanoEntityBean" alias="plano" />
    <entity name="br.ufrj.siga.ensino.CursoEntityBean" alias="curso" />
    <entity name="br.ufrj.siga.inscricao.PlanoOfertaEntityBean" alias="planoOferta" />
    <entity name="br.ufrj.siga.turma.AATurmaEntityBean" alias="oferta" />
    <entity name="br.ufrj.siga.turma.AtividadeAcademicaEntityBean" alias="aa" />
    <entity name="br.ufrj.siga.org.SegmentacaoEntityBean" alias="segmentacao" />
    <entity name="br.ufrj.siga.turma.TurmaEntityBean" alias="turma" />
    <entity name="br.ufrj.siga.turma.HorarioTurmaEntityBean" alias="horarioTurma" />
  </source>
  <relationship>
    <equals>
      <field name="curso_oid" entity="plano" />
      <field name="oid" entity="curso" />
    </equals>
    <equals>
      <field name="oid" entity="plano" />
      <field name="plano_oid" entity="planoOferta" />
    </equals>
    <equals>
      <field name="oid" entity="oferta" />
      <field name="aaturma_oid" entity="planoOferta" />
    </equals>
    <equals>
      <field name="atividadeAcademica_oid" entity="oferta" />
      <field name="oid" entity="aa" />
    </equals>
    <equals>
      <field name="segmentacao_oid" entity="plano" />
      <field name="oid" entity="segmentacao" />
    </equals>
    <equals>
      <field name="oid" entity="turma" />
      <field name="turma_oid" entity="oferta" />
    </equals>
    <equals>
      <field name="oid" entity="turma" />
      <field name="turma_oid" entity="horarioTurma" />
    </equals>
  </relationship>
  <presentation>
```

```
    <stunt name="br.ufrj.siga.inscricao.inscricaoEmMassa.model.Plano" alias="plano">
      <property name="nome" type="java.lang.String">
        <field name="nome" entity="plano" />
      </property>
      <property name="curso" type="br.ufrj.siga.inscricao.inscricaoEmMassa.model.Curso">
        <stunt name="br.ufrj.siga.inscricao.inscricaoEmMassa.model.Curso">
          <property name="nome" type="java.lang.String">
            <field name="nome" entity="curso" />
          </property>
          <property name="codigo" type="java.lang.String">
            <field name="codigo" entity="curso" />
          </property>
        </stunt>
      </property>
      <property name="ofertas" type="java.util.Collection">
        <stunt name="br.ufrj.siga.inscricao.inscricaoEmMassa.model.Oferta">
          <property name="vagas" type="java.lang.Integer">
            <field name="lotacao" entity="oferta" />
          </property>
          <property name="disciplina"
type="br.ufrj.siga.inscricao.inscricaoEmMassa.model.AtividadeAcademica">
            <stunt name="br.ufrj.siga.inscricao.inscricaoEmMassa.model.AtividadeAcademica">
              <property name="codigo" type="java.lang.String">
                <field name="codigo" entity="aa" />
              </property>
            </stunt>
           </property>
          <property name="turma" type="br.ufrj.siga.inscricao.inscricaoEmMassa.model.Turma">
            <stunt name="br.ufrj.siga.inscricao.inscricaoEmMassa.model.Turma">
              <property name="codigoTurma" type="java.lang.String">
                <field name="codigoTurma" entity="turma" />
              </property>
              <property name="nome" type="java.lang.String">
                <field name="nome" entity="turma" />
              </property>
              <property name="horariosTurma" type="java.util.Collection">
                <stunt name="br.ufrj.siga.inscricao.inscricaoEmMassa.model.HorarioTurma">
                  <property name="diaSemana" type="java.lang.String">
                    <field name="diaSemana" entity="horarioTurma" />
                  </property>
                </stunt>
              </property>
            </stunt>
          </property>
        </stunt>
      </property>
      <property name="segmentacao"
type="br.ufrj.siga.inscricao.inscricaoEmMassa.model.Segmentacao">
        <stunt name="br.ufrj.siga.inscricao.inscricaoEmMassa.model.Segmentacao">
          <property name="segmento" type="java.lang.String">
            <field name="ano" entity="segmentacao" />
            <text value="-" />
            <field name="periodo" entity="segmentacao" />
            <text value="-" />
            <field name="bloco" entity="segmentacao" />
          </property>
        </stunt>
      </property>
    </stunt>
  </presentation>
</query-stunt>
```

A utilização do *framework* Hércules tem como objetivo evitar a poluição do modelo por entidades com informação duplicada e que não tem importância na aplicação das regras de negócio. Neste sentido, os cosméticos são utilizados para registrar os atributos que farão parte da pesquisa e apresentação de resultado. Duas pesquisas compõem o serviço: a pesquisa de plano e a pesquisa de disciplinas. Para cada uma

existe uma apresentação de resultado. Pode ser especificado qualquer número de pesquisas. O XML abaixo descreve os cosméticos utilizados.

```
<cosmetics>
    <cosmetic name="selection" id="consultaPlano">
        <mark field="curso.codigo" entity="plano"/>
        <mark field="curso.nome" entity="plano"/>
        <mark field="segmentacao.segmento" entity="plano"/>
    </cosmetic>
    <cosmetic name="selectionResult" id="consultaPlano">
        <mark field="nome" entity="plano"/>
        <mark field="segmentacao.segmento" entity="plano"/>
    </cosmetic>
    <cosmetic name="selection" id="consultaDisciplina">
        <mark field="ofertas.oferta.curso.codigo" entity="plano"/>
        <mark field="ofertas.oferta.turma.codigoTurma" entity="plano"/>
        <mark field="ofertas.oferta.aa.codigo" entity="plano"/>
        <mark field="ofertas.oferta.turma.horariosTurma.horarioTurma.diaSemana"
entity="plano"/>
    </cosmetic>
    <cosmetic name="selectionResult" id="consultaDisciplina">
        <mark field="ofertas.oferta.curso.nome" entity="plano"/>
        <mark field="ofertas.oferta.turma.codigoTurma" entity="plano"/>
        <mark field="ofertas.oferta.turma.nome" entity="plano"/>
        <mark field="ofertas.oferta.aa.nome" entity="plano"/>
        <mark field="ofertas.oferta.aa.creditos.diaSemana" entity="plano"/>
        <mark field="ofertas.oferta.lotacao" entity="plano"/>
    </cosmetic>
</cosmetics>
```

O componente que faz a aplicação do cosmético de pesquisa e apresentação de resultado deve ser registrado no maquiador. O trecho de código abaixo faz o registro do componente no maquiador.

```
makeup.registerComponentImplementation(SelectionModelBuilder.class);
```

Uma vez acomodado o domínio no modelo de Hera, este domínio será transladado para Zeus na apresentação do caso de uso ao usuário. A informação deverá passar para o formato esperado em Zeus. Com base na informação dos cosméticos novas entidades e atributos surgirão na translação do domínio de Hera para Zeus resultando na combinação dos modelos apresentados nas Figura 40 e Figura 41. O usuário interage com a informação apresentada na tela até requisitar uma operação. O domínio sofre nova translação, agora de Zeus para Hera, e esta executa a operação requisitada retornando o resultado para Zeus. Zeus atualiza o seu domínio segundo as informações vindas de Hera. E assim esses passos ocorrem sucessivamente até o caso de uso ser encerrado.

A descrição do modelo de Hera em XML e a utilização de cosméticos para adicionar informação e comportamento complementar ao modelo foram melhorias implementadas em relação à forma original que Hera tratava o modelo. Na maneira antiga, era preciso escrever todas as classes na translação para Zeus, inclusive as

classes que representam a pesquisa e resultado de pesquisa. Não era conservada a informação da origem do modelo e por isso, a cada translação para a persistência era necessário codificar o mapeamento da informação dos modelos. O Anexo III apresenta um trecho de código que obtém as informações do modelo de Hera e faz as devidas operações para refletir as mudanças no modelo de persistência. Com as melhorias implementadas em Hera, o modelo utilizado é gerado a partir da descrição XML. As classes auxiliares derivadas dos cosméticos também são geradas, mas ficam transparentes ao programador. O Maquiador se encarrega de gerenciar essas classes.

Com essa proposta, cada camada possui um modelo mais apropriado para acomodar o domínio de acordo com as suas necessidades. Em Hera o programador tem acesso a um domínio mais claro e enxuto para a aplicação das regras de negócio. Além disso, a implementação é mais padronizada facilitando tanto o desenvolvimento quanto a manutenção do caso de uso. Os demais casos de uso não sofrem impacto pelo fato da Inscrição em massa utilizar o domínio em um formato diferente.

Em Zeus, apesar do domínio ser mais complexo, o programador dificilmente precisará lidar com essa informação diretamente. A descrição da vista de Zeus é realizada através de XML contendo as ligações dos componentes visuais com os elementos de modelo. O Anexo I contém o XML com a descrição da vista da Inscrição em massa. Zeus é utilizado no desenvolvimento dos casos de uso do SIGA desde 2003.

Apesar das vantagens conquistadas com utilização do *framework* Hércules, alguns aspectos no processo de desenvolvimento podem ser melhorados. Código e arquivos XML podem ser gerados automaticamente, no todo ou em parte, através de uma modelagem mais completa do caso de uso.

6 CONCLUSÃO E TRABALHOS FUTUROS

A constante busca por agilidade no processo de desenvolvimento de sistemas de software leva os analistas, a cada vez mais, investigar mecanismos para facilitar o desenvolvimento e manutenção dos sistemas. Em sistemas de grande porte a divisão em camadas é um recurso bastante utilizado para facilitar o desenvolvimento. No entanto, o domínio do sistema não recebe a devida atenção em todas as camadas. O domínio do sistema é o protagonista desta trama e seu papel deve ser cuidadosamente analisado em cada parte do sistema. Este trabalho busca um aprimoramento na visualização do domínio nas diversas camadas de um Sistema de Informação, baseado no *framework* Hércules.

Mover a informação entre cada parte do sistema é como cruzar uma barreira. Cada parte da aplicação tem objetivos e necessidades diferentes. A ubiqüidade do domínio mal interpretada engessa a informação abrindo espaço para o caos do sistema. O fato de o domínio estar presente em toda a aplicação não significa que este precise ter um formato constante. À medida que os sistemas crescem, essa abordagem gera uma deformação expressiva no domínio do sistema. Se cada parte do sistema tenta moldar o domínio segundo seus objetivos as outras partes necessariamente vão ter que se adaptar a essas modificações. Esse processo pode se agravar dependendo do tamanho do sistema gerando o caos tanto no desenvolvimento quanto na manutenção.

Cada parte do sistema enxerga o domínio de forma particular como na fábula dos cegos e o elefante. Todos os cegos ficaram satisfeitos com a sua própria interpretação independente das demais. O mesmo acontece com as camadas de um sistema. Para uma determinada camada somente interessa o domínio no formato adequado para a execução de suas tarefas. No entanto, esse domínio precisa transpor as barreiras do sistema transladando entre as camadas.

Este trabalho é uma contribuição para o *framework* Hércules, com o objetivo de fornecer representações e adequado tratamento para o domínio do sistema. O conceito de MVC recursivo é aplicado permitindo a divisão nas camadas modelo-vista-controle recursivamente. Ao longo deste trabalho é apresentada uma proposta para a representação do domínio para as camadas Zeus (camada de vista externa) e Hera (camada de controle externa). Entre uma camada e outra o modelo que representa a informação de domínio se altera. Cada parte do sistema tem domínio apropriado às suas tarefas evitando o caos da aplicação.

Na camada Zeus o objetivo é a apresentação da informação para o usuário. Dessa forma, apenas a informação de domínio que precisa ser apresentada fará parte do domínio de Zeus. Componentes visuais se conectam a elemento de modelo para refletir a interação do usuário. O modelo em Zeus representa as entidades, atributos e relacionamentos necessários à apresentação do caso de uso. A camada de modelo possui também engenhos responsáveis por executar algumas operações no modelo como, por exemplo, a limpeza e atualização.

Na camada Hera o objetivo é a aplicação das regras de negócio do caso de uso. O modelo de Hera além de representar as entidades, atributos e relacionamentos também aceita a adição de informação complementar. Essa informação é traduzida em uma marcação no modelo que pode ser utilizada para diversos fins. Neste trabalho foram desenvolvidos dois componentes que interpretam a marcação do modelo. Componentes dessa natureza são gerenciados por um container de componentes baseado no PicoContainer. As informações contidas no modelo são utilizadas tanto na camada Hera como também na translação da informação para outras camadas.

Dois componentes foram desenvolvidos como cosméticos para maquiar a face do domínio em Hera. O componente de “Restrição de Acesso” é responsável por marcar o escopo de acesso dos elementos do domínio. O componente de “Pesquisa e Apresentação de Resultado” marca os elementos que corresponderão a critérios de pesquisa e elementos que farão parte da apresentação do resultado da pesquisa.

Para avaliar a proposta desse trabalho, o caso de uso “Inscrição em Massa” foi implementado utilizando o *framework* Hércules. O componente de pesquisa e apresentação de resultado implementado para marcação do domínio foi utilizado neste estudo de caso.

A camada de modelo Zeus foi utilizada no projeto desenvolvido pelo NCE para o CREA-RJ (Conselho Regional de Engenharia, Arquitetura e Agronomia do Rio de Janeiro), no projeto de gestão acadêmica desenvolvido para a Unigranrio e está sendo utilizada no SIGA há dois anos. Esta proposta tem apresentado excelente funcionamento. O modelo utilizado para a representação do domínio se mostrou apropriado às necessidades de Zeus. As operações realizadas sobre o domínio apresentaram funcionamento correto. As interações do usuário refletiram no domínio sendo, no momento oportuno, transmitidas para Hera. A camada de modelo de Zeus permitiu que operações simples de interface fossem executadas sem intervenção de Hera.

Na camada de modelo de Hera foi utilizado o container de componentes baseado no PicoContainer, utilizado como maquiador do domínio. Uma descrição XML do domínio foi o recurso utilizado para a formação do meta-modelo do domínio. Informações importantes sobre a construção do domínio ficaram preservadas no modelo de Hera. A aplicação do cosmético de pesquisa e apresentação de resultado pelo maquiador possibilitou a configuração dos elementos de modelo participantes da etapa de seleção e lista da apresentação. As informações complementares registradas no meta-modelo não interferiu no domínio utilizado na aplicação das regras de negócio do caso de uso. O domínio em Hera ficou claro e enxuto facilitando o desenvolvimento e reduzindo esforço do programador.

A adoção de um modelo específico para a camada Zeus e outro para a camada Hera permitiu a adaptação do formato da informação adequado a cada camada sem influência entre camadas. Esta mesma conquista se verificou entre casos de uso. Cada caso de uso implementado em Hera possui uma visão do domínio apropriada a aplicação das suas regras de negócio. O controle de forças atuantes sobre o domínio evitou o caos no sistema.

A camada Olimpo, referente à camada de modelo externa do MVC recursivo, ainda encontra-se em fase de pesquisa. Trabalhos futuros poderão complementar este trabalho apresentando um modelo adequado para as operações de persistência do *framework* Hércules.

Algum esforço ainda foi observado na configuração do modelo e dos cosméticos. Novas formas de obter a informação estão sendo estudadas para dinamizar o processo. Como trabalho futuro, um estudo mais aprofundado em “Arquitetura orientada a modelo” (Model Driven Architecture - MDA) (MDA, 2005) (THOMAS, 2005) pode trazer importante contribuição ao projeto Hércules. O MDA é um modo de separar a arquitetura da aplicação de sua implementação. O MDA consiste em especificar regras para a construção de uma aplicação, baseado no modelo de uma aplicação genérica associado ao modelo específico da aplicação que se deseja construir. Para a construção da aplicação é preciso traçar o perfil da aplicação descrevendo regras. Com o auxílio do MDA será possível a construção automatizada do domínio adequado a cada camada do sistema.

REFERÊNCIAS BIBLIOGRÁFICAS

ALUR, D., CRUPI, J., MALKS, D. **Core J2EE Patterns**. Pearson Education; 1ª edição, 2001. 496 p.

BECK, K. **Smalltalk Best Practice Patterns**. Prentice Hall PTR, 1996. 240 p.

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

BRASIL, A.C. **XUL Builder**: Montando interfaces multiuso. 2003. Projeto final de curso (Bacharelado em Informática), Universidade Federal do Rio de Janeiro, Rio de Janeiro.

BURBECK, S. **Applications Programming in Smalltalk-80 (TM)**: How to use Modelo-View-Controller. Disponível em http://st-www.cs.uiuc.edu/users/smarch/st-docs/mvc.html Último acesso em: 15 mar. 2005.

CAVANESS, C. **Programming Jakarta Struts, 2nd Edition.** O'Reilly Media, 2004. 450 p.

FOWLER, M., **Inversion of Control Containers and the Dependency Injection pattern**. Disponível em http://www.martinfowler.com/articles/injection.html Último acesso em 03/01/2005.

GAMMA, E., HELM, R. , JOHNSON, R. , VLISSIDES, J. **Design Patterns**. Addison-Wesley Professional, 1995. 395 p.

HIBERNATE. Disponível em: http://www.hibernate.org/. Último acesso em: 20 mar. 2005.

HUSTED, T., DUMOULIN, C., FRANCISCUS, G., WNTERFELDT, D. **Struts in Action** . Greenwich: Manning Publications, 2004. 630 p.

Java BluePrints: Model-View-Controller. Disponível em: http://java.sun.com/blueprints/patterns/MVC-detailed.html. Último acesso em: 20 mar. 2005.

JDOM. Disponível em: http://www.jdom.org/. Último acesso em: 20 mar. 2005.

JOHNSON, M., **XML JavaBeans, Part 1**. Disponível em http://www.javaworld.com/jw-02-1999/jw-02-beans.html. Último acesso em 21 jan. 2005.

_____, **XML JavaBeans, Part 2**. Disponível em http://www.javaworld.com/javaworld/jw-03-1999/jw-03-beans.html. Último acesso em 21 jan. 2005.

_____, **XML JavaBeans, Part 3**. Disponível em http://www.javaworld.com/javaworld/jw-07-1999/jw-07-beans.html. Último acesso em 21 jan. 2005.

JOHNSON, R., HOELLER,J., ARENDSEN, A., RISBERG, T, SAMPALEANU, C. **Professional Java Development with the Spring Framework**. Wrox, 2005. 672 p.

KLEPPE, A., WARMER, J., BAST W. **MDA Explained: The Model Driven Architecture: Practice and Promise.** Addison-Wesley, 2003. 176 p.

HYATT, D. **XUL and RDF**: The Implementation of the Application Object Model. Disponível em: http://www.mozilla.org/xpfe/xulrdf.htm. Último acesso em: 15 mar. 2005.

Model View Controller. Disponível em: http://www.object-arts.com/EducationCenter/Overviews/MVC.htm. Último acesso em: 20 mar. 2005.

MONSON-HAEFEL, R., BURKE, B., LABOUREY, S. **Enterprise JavaBeans**. O'Reilly; 4ª edição, 2004. 800 p.

MOZILLA. Disponível em: http://www.mozilla.org/. Último acesso em 15 mar. 2005.

Os Pensadores: Platão. São Paulo: Editora Nova Cultural, 1999. 190 p.

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. 132 p. Dissertação (Mestrado em Informática) – Núcleo de Computação Eletrônica, Universidade Federal do Rio de Janeiro, Rio de Janeiro.

PAIS, A.P.V.; BRASIL, B.; OLIVEIRA, C.E.T.; TAVARES, G. **A Responsive Client Architecture with Local Object Behavior Deployment** . Proc. VIII International Conference on Object-Oriented Information Systems - OOIS'02. Montpellier, França: Springer-Verlag. 2002, p. 470-481.

PAIS, A. P. V.; CORRÊA, B. O. B. **Hércules – Apresentação**. 2003. Disponível em: http://hercules.nce.ufrj.br. Acesso em: 10 jan. 2005.

PAIS, A.P.V.; OLIVEIRA, C.E.T. **Enhancing UML Expressivity Towards Atomatic Code Generation** . Proc. VII International Conference on Object-Oriented

Information Systems - OOIS'01. Calgary, Canadá: Springer-Verlag. 2001, v. 1, p. 335-344.

PAIS, A.P.V.; OLIVEIRA, C.E.T., LEITE, P.H.P.M. **Robustness Diagram**: A Bridge Between Business Modeling And System Design . Proc. VII International Conference on Object-Oriented Information Systems - OOIS'01. Calgary, Canadá: Springer-Verlag. 2001, v. 1, p. 530-539.

PARSONS, R., **Components and the World of Chaos**. Disponível em www.martinfowler.com. Último acesso em 21 jan. 2005.

PEREIRA, L.A. **Geração Automática de Interfaces Visuais para Sistemas de Informação**. 2003. Projeto final de curso (Bacharelado em Informática), Universidade Federal do Rio de Janeiro, Rio de Janeiro.

PICOCONTAINER. Disponível em: http://www.picocontainer.org. Último acesso em: 20 mar. 2005.

PRESSMAN, R. S. **Engenharia de Software**. Graw Hill, 1995. 1056 p.

RDF. Disponível em: http://twiki.im.ufba.br/bin/view/XUL/TutorialXULCap06 Último acesso em: 15 mar. 2005.

ROSEMBERG, D.; SCOTT, K. **Use case driven object modeling with UML**: A pratical approach. Addison Wesley Longman, 1999. 165 p.

SANTOS, C. A. L. G. DOS, OLIVEIRA, R. DE. **Persistência de Objetos em Sistemas Prevalentes.** Disponível em: http://meusite.mackenzie.com.br/rogerio/tgi/2003PrevalenciaA.PDF Último acesso em 20 mar. 2005.

SAXE, J.G. **The Blind Men and the Elephant**. Disponível em: http://www.wordfocus.com/word-act-blindmen.html. Último acesso em: 20 mar. 2005.

SESHADRI, G. **Understanding JavaServer Pages Model 2 architecture.** JavaWorld.com, 1999. Disponível em: http://www.javaworld.com/javaworld/jw-12-1999/jw-12-ssj-jspmvc.html

SCOTT, K. **O Processo Unificado Explicado**. 1.ed. Rio de Janeiro: Bookman, 2003. 160 p.

SWING: The Swing Tutorial. Disponível em http://java.sun.com/docs/books/tutorial/uiswing. Último acesso em 20 mar. 2005.

TELES, V.M. **Extreme Programming**. Novatec Editora, 2004. 320 p.

THOMAS, D. **MDA**: Revenge of the Modelers or UML Utopia? Disponível em: http://www.martinfowler.com/ieeeSoftware/mda-thomas.pdf. Último acesso em: 20 mar. 2005.

UML: Unified Modeling Language. Disponível em: http://www.uml.org. Último acesso em 20 mar. 2005.

VILLELA, Carlos, OLIVEIRA, Daniel Quirino, VERÍSSIMO, Hamílton. **Desmistificando a Inversão de Controle**. Mundo Java, Rio de Janeiro, n. 07, ano II, p. 20-27, 2004.

XUL. Disponível em: http://twiki.im.ufba.br/bin/view/XUL. Último acesso em: 10 mar. 2005.

YOURDON, E., CONSTANTINE, L. **Structured Design**. Prentice Hall, Englewood Cliffs, N.J., 1979.

ZANETE, N.H. **Introdução ao RDF**: Resource Description Framework. Disponível em: http://www.faccar.com.br/zanete/zaneteRDF.htm. Último acesso em: 15 mar. 2005.

ANEXO I

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE mvc SYSTEM "http://labase.nce.ufrj.br/hercules/recursos/dtd/mvcWithXul.dtd">

<mvc>
  <view>
    <tabbox id="inscricaoEmMassa">
      <tabs>
        <tab label="Consulta" selected="true" />
        <tab label="Planos" selected="false" />
        <tab label="Criar/Editar Plano" selected="false" />
        <tab label="Inscrição" selected="false" />
        <tab label="Mensagem" selected="false" />
      </tabs>
      <tabpanels>
        <tabpanel id="formularioDeConsultaDePlanos">
          <vbox id="boxFormularioDeConsultaDePlanos" align="center" width="98%">
            <grid id="gridFormularioDeConsultaDePlanos" value="2">
              <columns>
                <column align="end" class="shadow" flex="2"/>
                <column align="center" class="light" flex="1" value="Opções de busca"/>
                <column align="start" class="light" flex="7" value="Consulta"/>
              </columns>
              <rows>
                <row>
                  <label id="lblCodigoCurso" class="caption" value="Código do Curso:" />
                  <menulist id="codigoCursoRestricao"
datasources="br.ufrj.siga.inscricao.inscricaoEmMassa.InscricaoEmMassaSessionBean$QueryVO"
ref="codigoCursoRestricao">
                    <menupopup>
                      <menuitem value="qualquer" label="qualquer" />
                      <menuitem value="comecando por" label="começando por" />
                      <menuitem value="contendo" label="contendo" />
                      <menuitem value="terminando por" label="terminando por" />
                      <menuitem value="igual a" label="igual a" />
                    </menupopup>
                  </menulist>
                  <textbox id="codigoCurso"
datasources="br.ufrj.siga.inscricao.inscricaoEmMassa.InscricaoEmMassaSessionBean$QueryVO"
ref="codigoCurso" maxlength="11" rows="1" cols="12" type="text" multiline="false" />
                </row>
                <row>
                  <label id="lblNomeCurso" class="caption" value="Nome do Curso:" />
                  <menulist id="cursoRestricao"
datasources="br.ufrj.siga.inscricao.inscricaoEmMassa.InscricaoEmMassaSessionBean$QueryVO"
ref="cursoRestricao">
                    <menupopup>
                      <menuitem value="qualquer" label="qualquer" />
                      <menuitem value="comecando por" label="começando por" />
                      <menuitem value="contendo" label="contendo" />
                      <menuitem value="terminando por" label="terminando por" />
                      <menuitem value="igual a" label="igual a" />
                    </menupopup>
                  </menulist>
                  <textbox id="curso"
datasources="br.ufrj.siga.inscricao.inscricaoEmMassa.InscricaoEmMassaSessionBean$QueryVO"
ref="curso" maxlength="45" rows="1" cols="11" type="text" multiline="false" />
                </row>
                <row>
                  <label id="lblSegmento" class="caption" value="Segmento:" />
                  <menulist id="segmentoRestricao"
datasources="br.ufrj.siga.inscricao.inscricaoEmMassa.InscricaoEmMassaSessionBean$QueryVO"
ref="segmentoRestricao">
                    <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="segmento"
datasources="br.ufrj.siga.inscricao.inscricaoEmMassa.InscricaoEmMassaSessionBean$QueryVO"
ref="segmento" maxlength="10" rows="1" cols="11" type="text" multiline="false" />
              </row>
            </rows>
            <caption label="Seleção de planos:"/>
          </grid>
        </vbox>
      </tabpanel>
      <tabpanel id="listaDePlanos">
        <box id="boxListaDePlanos" align="start" value="0" width="98%">
          <grid id="gridListaDePlanos"
datasources="br.ufrj.siga.inscricao.inscricaoEmMassa.InscricaoEmMassaSessionBean$ListVO"
ref="planos" class="listTable">
            <columns>
              <column value="Nome do plano" flex="3" class="light"/>
              <column value="Segmentação" flex="2"/>
            </columns>
            <rows>
              <template>
                <row>
                  <label id="nomePlano"
datasources="br.ufrj.siga.inscricao.inscricaoEmMassa.InscricaoEmMassaSessionBean$ListVO$ListIt
emVO" ref="nomePlano" class="normal" onclick="ListState-Editar" />
                  <label id="segmentacaoPlano"
datasources="br.ufrj.siga.inscricao.inscricaoEmMassa.InscricaoEmMassaSessionBean$ListVO$ListIt
emVO" ref="segmentacaoPlano" class="normal" />
                </row>
              </template>
            </rows>
          </grid>
          <caption label="Planos de estudo"/>
        </box>
      </tabpanel>

      <tabpanel id="formularioDeAlteracaoDePlano">
        <wizard id="card1">
          <wizardpage id="DadosPlano">
            <vbox id="dummBox" align="start">
              <grid id="tabelaDisciplinas" class="gridLayout">
                <columns>
                  <column class="shadow" flex="1"/>
                  <column class="normal" flex="3"/>
                </columns>
                <rows>
                  <row>
                    <label id="lblNomePlano" value="Nome do plano: "/>
                    <textbox id="nome" cols="50" maxlength="50"
datasources="br.ufrj.siga.inscricao.inscricaoEmMassa.InscricaoEmMassaSessionBean$PlanoVO"
ref="nome"/>
                  </row>
                  <row>
                    <label id="lblSegmento" value="Segmento (ano - periodo - bloco): "/>
                    <box id="dummy">
                      <textbox id="anoSegmento" cols="5" maxlength="4"
datasources="br.ufrj.siga.inscricao.inscricaoEmMassa.InscricaoEmMassaSessionBean$PlanoVO"
ref="anoSegmento"/>
                      <label id="separador" value=" - "/>
                      <textbox id="periodoSegmento" cols="2" maxlength="1"
datasources="br.ufrj.siga.inscricao.inscricaoEmMassa.InscricaoEmMassaSessionBean$PlanoVO"
ref="periodoSegmento"/>
                      <label id="separador" value=" - "/>
                      <textbox id="blocoSegmento" cols="2" maxlength="1"
datasources="br.ufrj.siga.inscricao.inscricaoEmMassa.InscricaoEmMassaSessionBean$PlanoVO"
ref="blocoSegmento"/>
                    </box>
                  </row>
                  <row>
                    <label id="lblCurso" value="Código do Curso:"/>
                    <textbox id="codigoCurso" cols="7" maxlength="10"
datasources="br.ufrj.siga.inscricao.inscricaoEmMassa.InscricaoEmMassaSessionBean$PlanoVO"
ref="codigoCurso"/>
                  </row>
                </rows>
```

```
                </grid>
                <grid id="ofertasTable" class="plainTable"
datasources="br.ufrj.siga.inscricao.inscricaoEmMassa.InscricaoEmMassaSessionBean$PlanoVO"
ref="ofertas">
                  <columns>
                    <column value="Turma" flex="5"/>
                    <column value="Disciplina" align="end" flex="2"/>
                    <column value="Creditos" align="end" flex="1"/>
                    <column value="Vagas" align="end" flex="1"/>
                  </columns>
                  <rows>
                    <template>
                      <row>
                        <label id="nomeTurma"
datasources="br.ufrj.siga.inscricao.inscricaoEmMassa.InscricaoEmMassaSessionBean$OfertaVO"
ref="nomeTurma"/>
                        <label id="codigoAA"
datasources="br.ufrj.siga.inscricao.inscricaoEmMassa.InscricaoEmMassaSessionBean$OfertaVO"
ref="codigoAA"/>
                        <label id="creditos"
datasources="br.ufrj.siga.inscricao.inscricaoEmMassa.InscricaoEmMassaSessionBean$OfertaVO"
ref="creditosAA"/>
                        <label id="vagas"
datasources="br.ufrj.siga.inscricao.inscricaoEmMassa.InscricaoEmMassaSessionBean$OfertaVO"
ref="vagasOferta"/>
                      </row>
                    </template>
                  </rows>
                  <commands>
                    <command class="ADD" oncommand="procurarOfertas"/>
                  </commands>
                </grid>
              </vbox>
            </wizardpage>
            <wizardpage id="Selecao">
              <vbox width="100" align="center" id="mainBoxSelecao">
                <box class="bevel" id="selectLayoutBevelBorder" width="100" value="0">
                  <grid value="1" id="formSelect">
                    <columns>
                      <column flex="2" class="shadow" align="end" />
                      <column flex="1" class="light" align="start" value="Opções de busca" />
                      <column flex="3" class="light" align="start" value="Consulta" />
                    </columns>
                    <rows>
                      <row>
                        <label id="lbl_cursoRestricao" class="caption" value="Nome do curso:"
/>
                        <menulist id="cursoRestricao"
datasources="br.ufrj.siga.inscricao.inscricaoEmMassa.InscricaoEmMassaSessionBean$OfertaQueryVO
" ref="cursoRestricao">
                          <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="curso"
datasources="br.ufrj.siga.inscricao.inscricaoEmMassa.InscricaoEmMassaSessionBean$OfertaQueryVO
" ref="curso" maxlength="45" rows="1" cols="30" type="text" multiline="false" />
                      </row>
                      <row>
                        <label id="lbl_nivelCursoRestricao" class="caption" value="Nível do
curso:" />
                        <menulist id="nivelCursoRestricao"
datasources="br.ufrj.siga.inscricao.inscricaoEmMassa.InscricaoEmMassaSessionBean$OfertaQueryVO
" ref="nivelCursoRestricao">
                          <menupopup>
                            <menuitem value="qualquer" label="qualquer" />
                            <menuitem value="igual a" label="igual a" />
                          </menupopup>
                        </menulist>
                        <menulist id="nivelCurso"
datasources="br.ufrj.siga.inscricao.inscricaoEmMassa.InscricaoEmMassaSessionBean$OfertaQueryVO
" ref="nivelCurso">
                          <menupopup>
```

```
                          <menuitem value="3" label="Graduação" />
                          <menuitem value="4" label="Extensão" />
                          <menuitem value="5" label="Aperfeiçoamento" />
                          <menuitem value="6" label="Especialização" />
                          <menuitem value="7" label="Mestrado" />
                          <menuitem value="8" label="Doutorado" />
                          <menuitem value="9" label="Pós-doutorado" />
                        </menupopup>
                      </menulist>
                    </row>
                    <row>
                      <label id="lbl_codigoAARestricao" class="caption" value="Código da
disciplina:" />
                      <menulist id="codigoAARestricao"
datasources="br.ufrj.siga.inscricao.inscricaoEmMassa.InscricaoEmMassaSessionBean$OfertaQueryVO
" ref="codigoAARestricao">
                        <menupopup>
                          <menuitem value="qualquer" label="qualquer" />
                          <menuitem value="comecando por" label="começando por" />
                          <menuitem value="contendo" label="contendo" />
                          <menuitem value="terminando por" label="terminando por" />
                          <menuitem value="igual a" label="igual a" />
                        </menupopup>
                      </menulist>
                      <textbox id="codigoAA"
datasources="br.ufrj.siga.inscricao.inscricaoEmMassa.InscricaoEmMassaSessionBean$OfertaQueryVO
" ref="codigoAA" maxlength="8" rows="1" cols="8" type="text" multiline="false" />
                    </row>
                    <row>
                      <label id="lbl_TurmaRestricao" class="caption" value="Número da
turma:" />
                      <menulist id="codigoTurmaRestricao"
datasources="br.ufrj.siga.inscricao.inscricaoEmMassa.InscricaoEmMassaSessionBean$OfertaQueryVO
" ref="numeroTurmaRestricao">
                        <menupopup>
                          <menuitem value="qualquer" label="qualquer" />
                          <menuitem value="comecando por" label="começando por" />
                          <menuitem value="contendo" label="contendo" />
                          <menuitem value="terminando por" label="terminando por" />
                          <menuitem value="igual a" label="igual a" />
                        </menupopup>
                      </menulist>
                      <textbox id="codigoTurma"
datasources="br.ufrj.siga.inscricao.inscricaoEmMassa.InscricaoEmMassaSessionBean$OfertaQueryVO
" ref="numeroTurma" maxlength="10" rows="1" cols="10" type="text" multiline="false" />
                    </row>
                    <row>
                      <label id="lbl_diaSemanaRestricao" class="caption" value="Dia da
semana:" />
                      <menulist id="diaSemanaRestricao"
datasources="br.ufrj.siga.inscricao.inscricaoEmMassa.InscricaoEmMassaSessionBean$OfertaQueryVO
" ref="diaSemanaRestricao">
                        <menupopup>
                          <menuitem value="qualquer" label="qualquer" />
                          <menuitem value="igual a" label="igual a" />
                        </menupopup>
                      </menulist>
                      <menulist id="diaSemana"
datasources="br.ufrj.siga.inscricao.inscricaoEmMassa.InscricaoEmMassaSessionBean$OfertaQueryVO
" ref="diaSemana">
                        <menupopup>
                          <menuitem value="2" label="segunda-feira" />
                          <menuitem value="3" label="terça-feira " />
                          <menuitem value="4" label="quarta-feira " />
                          <menuitem value="5" label="quinta-feira " />
                          <menuitem value="6" label="sexta-feira " />
                          <menuitem value="7" label="sábado " />
                          <menuitem value="1" label="domingo " />
                        </menupopup>
                      </menulist>
                    </row>
                  </rows>
                </grid>
              </box>
              <box id="boxBotao" align="center">
                <button id="botaoConsultar" command="listarOfertas" label="consultar"
type="action"/>
```

```
              </box>
              <caption label="Seleção de turmas: " />
            </vbox>
          </wizardpage>
          <wizardpage id="Lista">
            <grid id="formList" value="1">
              <rows>
                <row>
                  <grid id="tableListaInscricao"
datasources="br.ufrj.siga.inscricao.inscricaoEmMassa.InscricaoEmMassaSessionBean$OfertaListVO"
ref="ofertas" class="checkTable">
                    <columns>
                      <column value="Curso" />
                      <column value="Número da turma" />
                      <column value="Nome da turma" />
                      <column value="Disciplina" />
                      <column value="Créditos" />
                      <column value="Lotação" />
                    </columns>
                    <rows>
                      <template>
                        <row>
                          <label id="nomeCurso"
datasources="br.ufrj.siga.inscricao.inscricaoEmMassa.InscricaoEmMassaSessionBean$OfertaVO"
ref="nomeCurso" class="normal" />
                          <label id="codTurma"
datasources="br.ufrj.siga.inscricao.inscricaoEmMassa.InscricaoEmMassaSessionBean$OfertaVO"
ref="codTurma" class="normal" />
                          <label id="nomeTurma"
datasources="br.ufrj.siga.inscricao.inscricaoEmMassa.InscricaoEmMassaSessionBean$OfertaVO"
ref="nomeTurma" class="normal" />
                          <label id="codigoAA"
datasources="br.ufrj.siga.inscricao.inscricaoEmMassa.InscricaoEmMassaSessionBean$OfertaVO"
ref="codigoAA" class="normal" />
                          <label id="creditosAA"
datasources="br.ufrj.siga.inscricao.inscricaoEmMassa.InscricaoEmMassaSessionBean$OfertaVO"
ref="creditosAA" class="normal" />
                          <label id="vagasOferta"
datasources="br.ufrj.siga.inscricao.inscricaoEmMassa.InscricaoEmMassaSessionBean$OfertaVO"
ref="vagasOferta" class="normal" />
                        </row>
                      </template>
                    </rows>
                    <commands>
                      <command label="marcar tudo" class="CHECK_ALL" />
                      <command label="desmarcar tudo" class="UNCHECK_ALL" />
                      <command label="retirar marcados da lista"
class="REMOVE_CHECKED_ITEMS" />
                      <command label="adicionar à inscrição" class="EXTERNAL"
oncommand="adicionarOfertasAoPlano" />
                    </commands>
                  </grid>
                </row>
              </rows>
            </grid>
          </wizardpage>
        </wizard>
      </tabpanel>

      <tabpanel id="formularioDeInscricaoDeAlunos">
        <wizard id="inscricao">
          <wizardpage id="Inscricao">
            <vbox id="dummBox" align="center" width="100" >
              <caption label="Formulário de inscrição"/>
              <grid id="tabelaMatriculasDRE"
datasources="br.ufrj.siga.inscricao.inscricaoEmMassa.InscricaoEmMassaSessionBean$AlunosVO"
ref="alunos" class="plainTable">
                <columns>
                  <column value="Matrícula DRE" />
                </columns>
                <rows>
                  <template>
                    <row>
                      <textbox id="matriculaDRE"
datasources="br.ufrj.siga.inscricao.inscricaoEmMassa.InscricaoEmMassaSessionBean$AlunoVO"
ref="matriculaDRE" />
                    </row>
```

```
                  </template>
                </rows>
              </grid>
              <box id="dummBoxButton" align="center" width="100" value="3">
                <button id="btRecuperarAlunos" command="recuperarAlunos"></button>
              </box>
            </vbox>
          </wizardpage>
          <wizardpage id="Confirmacao">
            <vbox id="dummBox" align="center" >
              <caption label="Confirmação de inscrição"/>
              <box id="dadosPlanoBorder" class="bevel" width="100">
                <grid id="dadosPlano" class="gridLayout">
                  <columns>
                    <column class="shadow" flex="1"/>
                    <column class="light" flex="3"/>
                  </columns>
                  <rows>
                    <row>
                      <label id="lblNomePlano" value="Nome do plano: " class="caption"/>
                      <label id="nome"
datasources="br.ufrj.siga.inscricao.inscricaoEmMassa.InscricaoEmMassaSessionBean$PlanoVO"
ref="nome"/>
                    </row>
                    <row>
                      <label id="lblSegmento" value="Segmento (ano - periodo - bloco): "
class="caption"/>
                      <box id="dummy">
                        <label id="anoSegmento"
datasources="br.ufrj.siga.inscricao.inscricaoEmMassa.InscricaoEmMassaSessionBean$PlanoVO"
ref="anoSegmento"/>
                        <label id="separador" value=" - "/>
                        <label id="periodoSegmento"
datasources="br.ufrj.siga.inscricao.inscricaoEmMassa.InscricaoEmMassaSessionBean$PlanoVO"
ref="periodoSegmento"/>
                        <label id="separador" value=" - "/>
                        <label id="blocoSegmento"
datasources="br.ufrj.siga.inscricao.inscricaoEmMassa.InscricaoEmMassaSessionBean$PlanoVO"
ref="blocoSegmento"/>
                      </box>
                    </row>
                  </rows>
                </grid>
              </box>
              <grid id="tabelaMatriculasDRE"
datasources="br.ufrj.siga.inscricao.inscricaoEmMassa.InscricaoEmMassaSessionBean$AlunosVO"
ref="alunos" class="listTable">
                <columns>
                  <column value="Matrícula DRE" flex="1"/>
                  <column value="Nome do aluno" flex="3"/>
                </columns>
                <rows>
                  <template>
                    <row>
                      <label id="matriculaDRE"
datasources="br.ufrj.siga.inscricao.inscricaoEmMassa.InscricaoEmMassaSessionBean$AlunoVO"
ref="matriculaDRE" />
                      <label id=" nome"
datasources="br.ufrj.siga.inscricao.inscricaoEmMassa.InscricaoEmMassaSessionBean$AlunoVO"
ref="nome" />
                    </row>
                  </template>
                </rows>
              </grid>
              <box id="dummBoxButton" align="center" width="100" value="3">
                <button id="btConfirmarInscreverAlunos"
command="confirmarInscricoes"></button>
              </box>
            </vbox>
          </wizardpage>
        </wizard>
      </tabpanel>
      <tabpanel id="mensagem">
        <hbox id="mainBoxtabbedMenu" align="center" width="98%">
          <grid id="formMessage" value="1">
            <rows>
              <row>
```

```
                  <grid id="tableMessage" ref="message" class="listTable">
                    <columns>
                      <column value="Mensagem"/>
                    </columns>
                    <rows>
                      <template>
                        <row>
                          <label id="mensagem" datasources="MensagemBean" ref="mensagem"
class="normal" />
                        </row>
                      </template>
                    </rows>
                  </grid>
                </row>
              </rows>
            </grid>
            <caption label="Mensagens do sistema"/>
          </hbox>
        </tabpanel>
      </tabpanels>
    </tabbox>
  </view>
  <model>
    <EDIT>
      <JavaBean
class="br.ufrj.siga.inscricao.inscricaoEmMassa.InscricaoEmMassaSessionBean$QueryVO">
        <properties>
          <property type="java.lang.String" name="codigoCurso" />
          <property type="java.lang.String" name="curso" />
          <property type="java.lang.String" name="segmento" />
          <property type="java.lang.String" value="qualquer" name="codigoCursoRestricao" />
          <property type="java.lang.String" value="qualquer" name="cursoRestricao" />
          <property type="java.lang.String" value="qualquer" name="segmentoRestricao" />
        </properties>
      </JavaBean>
      <JavaBean
class="br.ufrj.siga.inscricao.inscricaoEmMassa.InscricaoEmMassaSessionBean$OfertaQueryVO"
delete="false">
        <properties>
          <property name="codigoAARestricao" value="qualquer" type="java.lang.String" />
          <property name="diaSemanaRestricao" value="qualquer" type="java.lang.String" />
          <property name="nivelCursoRestricao" value="qualquer" type="java.lang.String" />
          <property name="cursoRestricao" value="qualquer" type="java.lang.String" />
          <property name="curso" value="" type="java.lang.String" />
          <property name="diaSemana" value="" type="java.lang.String" />
          <property name="codigoAA" value="" type="java.lang.String" />
          <property name="numeroTurmaRestricao" value="qualquer" type="java.lang.String" />
          <property name="nivelCurso" value="" type="java.lang.String" />
          <property name="numeroTurma" value="" type="java.lang.String" />
        </properties>
      </JavaBean>
      <JavaBean
class="br.ufrj.siga.inscricao.inscricaoEmMassa.InscricaoEmMassaSessionBean$ListVO"
delete="false">
        <properties>
          <property name="planos" type="java.util.Collection">
            <collection>
            </collection>
          </property>
        </properties>
      </JavaBean>
      <JavaBean
class="br.ufrj.siga.inscricao.inscricaoEmMassa.InscricaoEmMassaSessionBean$AlunosVO"
delete="false">
        <properties>
          <property name="alunos" type="java.util.Collection">
            <collection>
              <JavaBean
class="br.ufrj.siga.inscricao.inscricaoEmMassa.InscricaoEmMassaSessionBean$AlunoVO"
delete="false">
                <properties>
                  <property name="nome" value="" type="java.lang.String" />
                  <property name="aluno_oid" value="" type="java.lang.String" />
                  <property name="matriculaDRE" value="" type="java.lang.String" />
                </properties>
              </JavaBean>
            </collection>
```

```
          </property>
        </properties>
      </JavaBean>
      <JavaBean
class="br.ufrj.siga.inscricao.inscricaoEmMassa.InscricaoEmMassaSessionBean$OfertaListVO"
delete="false">
        <properties>
          <property name="ofertas" type="java.util.Collection">
            <collection/>
          </property>
        </properties>
      </JavaBean>
      <JavaBean
class="br.ufrj.siga.inscricao.inscricaoEmMassa.InscricaoEmMassaSessionBean$PlanoVO">
        <properties>
          <property name="oid" type="java.lang.String"/>
          <property name="nome" type="java.lang.String"/>
          <property name="curso_oid" type="java.lang.String"/>
          <property name="nomeCurso" type="java.lang.String"/>
          <property name="codigoCurso" type="java.lang.String"/>
          <property name="segmentacao_oid" type="java.lang.String"/>
          <property name="anoSegmento" type="java.lang.String"/>
          <property name="periodoSegmento" type="java.lang.String"/>
          <property name="blocoSegmento" type="java.lang.String"/>
          <property name="ofertas" type="java.util.Collection">
            <collection/>
          </property>
        </properties>
      </JavaBean>
    </EDIT>
  </model>
</mvc>
```

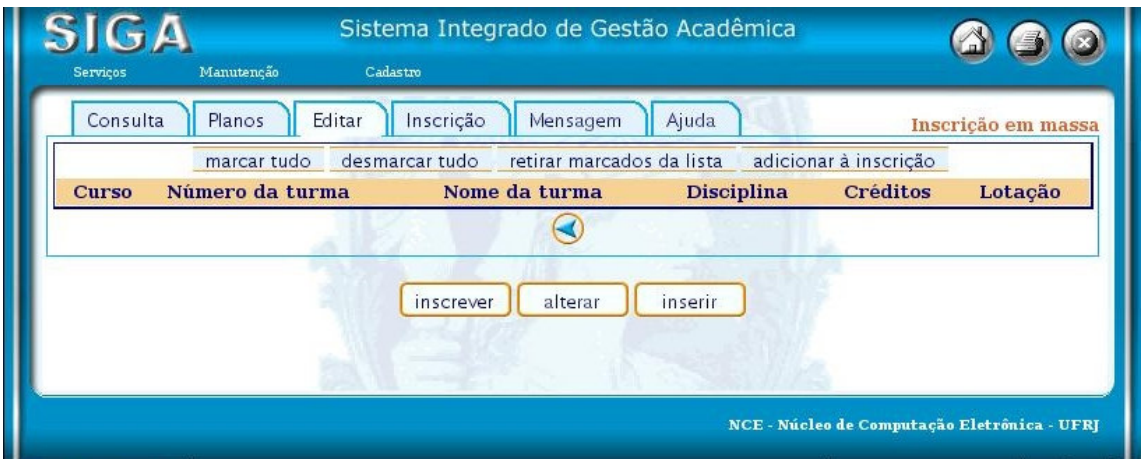
ANEXO II


SIGA
Sistema Integrado de Gestão Acadêmica
Serviços
Manutenção
Cadastro
Consulta
Planos
Editar
Inscrição
Mensagem
Ajuda
Inscrição em massa
Seleção de planos:
Opções de busca
Consulta
Código do Curso:
qualquer
Nome do Curso:
qualquer
Segmento:
qualquer
consultar
NCE - Núcleo de Computação Eletrônica - UFRJ


SIGA
Sistema Integrado de Gestão Acadêmica
Serviços
Manutenção
Cadastro
Consulta
Planos
Editar
Inscrição
Mensagem
Ajuda
Inscrição em massa
Nome do plano
Segmentação
Plano 2
2005-2-0
Plano 1
2005-1-0
NCE - Núcleo de Computação Eletrônica - UFRJ


SIGA
Sistema Integrado de Gestão Acadêmica
Serviços
Manutenção
Cadastro
Consulta
Planos
Editar
Inscrição
Mensagem
Ajuda
Inscrição em massa
Nome do plano:
Plano 2
Segmento (ano - periodo - bloco):
2005
-2
-0
Código do Curso:
3101070
Turma
Disciplina
Creditos
Vagas
inscrever
alterar
inserir
NCE - Núcleo de Computação Eletrônica - UFRJ


SIGA
Sistema Integrado de Gestão Acadêmica
Serviços
Manutenção
Cadastro
Consulta
Planos
Editar
Inscrição
Mensagem
Ajuda
Inscrição em massa
Seleção de turmas:
Opções de busca
Consulta
Nome do curso:
contendo
ciência da computação
Nível do curso:
qualquer
Código da disciplina:
qualquer
Número da turma:
qualquer
Dia da semana:
qualquer
consultar
inscrever
alterar
inserir
NCE - Núcleo de Computação Eletrônica - UFRJ


SIGA
Sistema Integrado de Gestão Acadêmica
Serviços
Manutenção
Cadastro
Consulta
Planos
Editar
Inscrição
Mensagem
Ajuda
Inscrição em massa
marcar tudo
desmarcar tudo
retirar marcados da lista
adicionar à inscrição
Curso
Número da turma
Nome da turma
Disciplina
Créditos
Lotação
inscrever
alterar
inserir
NCE - Núcleo de Computação Eletrônica - UFRJ

ANEXO III

```
//Criando entidade de Plano de Estudos
HomeFactory contexto = new LocalHomeFactory();
PlanoEstudoEntityHome planoHome =
   (PlanoEstudoEntityHome)PortableRemoteObject.narrow(
   contexto.lookup(EjbReferences.PLANO_ESTUDO), PlanoEstudoEntityHome.class );
PlanoEstudoEntity planoEntity = planoHome.findByPrimaryKey(plano.getOid());

planoEntity.setNome(plano.getNome());

//Buscando entidade de Curso
CursoEntityHome cursoHome = (CursoEntityHome)PortableRemoteObject.narrow(
   contexto.lookup(EjbReferences.CURSO), CursoEntityHome.class );
CursoEK cursoEK = new CursoEK(plano.getCurso().getCodigo());
CursoEntity cursoEntity = cursoHome.findByExternalKey(cursoEK);

planoEntity.setCurso(cursoEntity);


//Buscando Segmentacao
SegmentacaoEntityHome segmentacaoHome =
  (SegmentacaoEntityHome)PortableRemoteObject.narrow(
  contexto.lookup(EjbReferences.SEGMENTACAO), SegmentacaoEntityHome.class );

//Separando em ano, período e bloco
StringTokenizer segmentacaoTk = new
   StringTokenizer(plano.getSegmentacao().getSegmento(),"-");
String ano = segmentacaoTk.hasMoreTokens()?segmentacaoTk.nextToken():"";
String periodo = segmentacaoTk.hasMoreTokens()?segmentacaoTk.nextToken():"";
String bloco = segmentacaoTk.hasMoreTokens()?segmentacaoTk.nextToken():"";

SegmentacaoEK segmentacaoEK = new SegmentacaoEK(ano,periodo,bloco);

SegmentacaoEntity segmentacaoEntity =
    segmentacaoHome.findByExternalKey(segmentacaoEK);


//Limpar a coleção atual de ofertas e criar novas de acordo com o
//que foi preenchido na tela
PlanoOfertaEntityHome planoOferaHome =
   (PlanoOfertaEntityHome)PortableRemoteObject.narrow(
   contexto.lookup(EjbReferences.PLANO_OFERTA), PlanoOfertaEntityHome.class );
Collection ofertas = planoOferaHome.findByPlano(plano.getOid());
apagarOfertas(ofertas);
Iterator iter = plano.getOfertas().iterator();
while(iter.hasNext()){
  Oferta oferta = (Oferta)iter.next();
  PlanoOfertaEntity planoOfertaEntity = planoOfertaHome.Create();

  // Buscar AtividadeAcademica
  AtividadeAcademicaEntityHome atividadeAcademicaHome =
    (AtividadeAcademicaEntityHome)PortableRemoteObject.narrow(
```

```
        contexto.lookup(EjbReferences.ATIVIDADE_ACADEMICA),
        AtividadeAcademicaEntityHome.class );
      AtividadeAcademicaEK atividadeAcademicaEK = new
        AtividadeAcademicaEK(oferta.getDisciplina().getCodigo());
      AtividadeAcademicaEntity atividadeAcademicaEntity =
        atividadeAcademicaHome.findByExternalKey(atividadeAcademicaEK);

      //Buscar Turma
      TurmaEntityHome turmaHome = (TurmaEntityHome)PortableRemoteObject.narrow(
        contexto.lookup(EjbReferences.TURMA), TurmaEntityHome.class );
      TurmaEK turmaEK = new TurmaEK(oferta.getTurma().getCodigoTurma());
      TurmaEntity turmaEntity = turmaHome.findByExternalKey(turmaEK);

      //Buscar AATurma
      AATurmaEntityHome aaTurmaHome =
        (AATurmaEntityHome)PortableRemoteObject.narrow(
        contexto.lookup(EjbReferences.AATURMA), AATurmaEntityHome.class );
      AATurmaEK aaTurmaEK = new
        AATurmaEK(atividadeAcademicaEntity.getOid(),turmaEntity.getOid());
      AATurmaEntity aaTurmaEntity = aaTurmaHome.findByExternalKey(aaTurmaEK);


      planoOfertaEntity.setAATurma(aaTurmaEntity);
      planoOfertaEntity.setPlano(planoEntity);

    }
```